FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Beiträge zum Wissenschaftlichen Rechnen Ergebnisse des Gaststudentenprogramms 2000 des John von Neumann-Instituts für Computing

Rüdiger Esser, Daniel Mallmann (Hrsg.)

FZJ-ZAM-IB-2000-15

November 2000

(letzte Änderung: 28.11.2000)

Vorwort

Die Ausbildung im Wissenschaftlichen Rechnen ist neben der Bereitstellung von Supercomputer-Leistung und der Durchführung eigener Forschung eine der Hauptaufgaben des John von Neumann-Instituts für Computing (NIC) und hiermit des ZAM als wesentlicher Säule des NIC. Um den akademischen Nachwuchs mit verschiedenen Aspekten des Wissenschaftlichen Rechnens vertraut zu machen, führten wir in diesem Jahr erstmals während der Sommersemesterferien ein Gaststudentenprogramm durch. Entsprechend dem fächerübergreifenden Charakter des Wissenschaftlichen Rechnens waren Studenten der Natur- und Ingenieurwissenschaften, der Mathematik und Informatik angesprochen. Die Bewerber mußten das Vordiplom abgelegt haben und von einem Professor empfohlen sein.

Die neun vom NIC ausgewählten Teilnehmer, unter ihnen eine Studentin aus Griechenland und je ein Teilnehmer aus Rußland und China, kamen für zehn Wochen, vom 7. August bis zum 13. Oktober, ins Forschungszentrum. Die meisten von ihnen wollen sich auf die Gebiete Computational Physics oder Computational Chemistry spezialisieren. Sie beteiligten sich hier an den Forschungs- und Entwicklungsarbeiten des ZAM und der Forschungsgruppe Komplexe Systeme des NIC und wurden jeweils einem Wissenschaftler zugeordnet, der mit ihnen zusammen eine Aufgabe festlegte und sie bei der Durchführung anleitete.

Die Gaststudenten und ihre Betreuer waren:

Timo Betcke, Hamburg	Inge Gutheil
Boris Bierwald, Duisburg	Stefan Birmanns
Zoe Cournia, Athen	Godehard Sutmann
Timm Höhr, Konstanz	Peter Grassberger, NIC
Jun-Mei Shi, Erlangen	Astrid Goeke, Bernhard Steffen
Carsten Urbach, Berlin	Bernd Körfgen
Oliver Vormoor, Göttingen	Rudolf Berrendorf, Godehard Sutmann
Jochen Werth, Duisburg	Peter Grassberger, NIC
Rouslan Zinetoulline, Duisburg	Ulrich Detert, Felix Wolf

Zu Beginn ihres Aufenthalts erhielten die Gaststudenten eine viertägige Einführung in die Programmierung und Nutzung der Parallelrechner im ZAM. Da sich die Teilchensimulation als wichtiges gemeinsames Thema herausstellte, wurde von Godehard Sutmann zusätzlich ein Seminar über parallele Techniken der Molekulardynamik-Simulation veranstaltet. Um den Erfahrungsaustausch untereinander zu fördern und den Fortgang der Arbeiten zu dokumentieren, präsentierten die Gaststudenten selbst zur Mitte und am Ende ihres Aufenthalts ihre Aufgabenstellung und die erreichten Ergebnisse. Sie verfaßten zudem Beiträge mit den Ergebnissen für diesen Internen Bericht des ZAM.

Wir danken den Teilnehmern für ihre engagierte Mitarbeit - schließlich haben sie geholfen, einige Forschungsarbeiten des ZAM weiterzubringen - aber auch den Betreuern, die tatkräftige Unterstützung dabei geleistet haben.

Da bei der Durchführung dieses ersten Gaststudentenprogramms des NIC organisatorisches Neuland betreten werden mußte, war seitens Verwaltung und ZAM manch administrative Schwierigkeit gemeinsam zu bewältigen. Das NIC dankt allen, die daran mitgewirkt haben, insbesondere dem Verein der Freunde und Förderer des Forschungszentrums und der Firma SGI für die finanzielle Unterstützung. Es ist zu hoffen, daß das erfolgreiche Experiment künftig wiederholt werden kann, schließlich ist die Förderung des wissenschaftlichen Nachwuchses dem Forschungszentrum ein besonderes Anliegen.

Weitere Informationen über das Gaststudentenprogramm, auch über eine mögliche Fortführung, findet man unter http://www.fz-juelich.de/zam/gaststudenten.

Jülich, November 2000 Rüdiger Esser, Daniel Mallmann

Inhalt

Timo Betcke:
Performance analysis of various parallelization methods
for BLAS3 routines on cluster architectures 1

Boris Bierwald:
Online Visualisierung von Molekulardynamik-Simulationen 17

Zoi Cournia:
The Ewald Summation method: Calculating long-range interactions 27

Timm Höhr:
Simulationen ungeordneter dipolarer Systeme
mit der Methode des "Parallel Tempering" 39

Jun-Mei Shi:
Parallelization of the Multigrid Solver for Flows in Complex Geometries:
FASTEST2D-LBR 57

Carsten Urbach:
Installation and Test of the DRAMA library on the ZAMpano cluster 73

Oliver Vormoor:
Simulation of colloidal systems in aqueous solution 93

Jochen Werth
Einfluss elektrischer Ladungen auf Stabilität und
Agglomerationsvorgänge in Suspensionen 109

Rouslan Zinetoulline:
Synchronisation der Uhren auf dem ZAMpano-Cluster 119

Performance analysis of various parallelization methods for BLAS3 routines on cluster architectures

Timo Betcke

Technische Universität Hamburg-Harburg

t.betcke@tu-harburg.de

Abstract: Traditional parallel computers are either based on the shared memory or the distributed memory model. The new ZAMpano Cluster of the Central Institute for Applied Mathematics at the Research Center Jülich provides both techniques of memory management. On every node of the cluster either OpenMP, a thread based parallelization technique for shared memory systems, or MPI, which is based on message passing between different processes, is available. The nodes exchange data by using the MPI protocol. For performing operations in linear algebra it is either possible to use only MPI communication even when processors of the same node have to share data or to use a hybrid data distribution model based on OpenMP on one node and MPI for the communication between nodes. In this article the performance of both parallelization models is analyzed for the ZAMpano Cluster and for an HP 9000-N Class Enterprise Cluster at the Technical University of Hamburg-Harburg by measuring the performance of ScaLAPACK matrix-multiplication routines and my own OpenMP routines based on BLAS3. The target will be to create an efficient BLAS3 matrix-multiplication which takes advantage of both worlds, OpenMP and MPI.

1 Introduction

With the upcoming of the new generation of massively parallel systems built as clusters the question arises which programming model is suitable for such systems. On the one hand traditional message passing is possible on clusters and leads to good results but on the other hand the shared memory architecture on every node allows a shared memory parallelization model like OpenMP. So the question whether there is a gain by using shared memory parallelization on every node and message passing between nodes arises because this sort of hybrid parallelization leads to new efforts to parallelize programs in contrast to an only message passing based parallelization like MPI which is already widely used and efficient implementations exist for a great variety of algorithms.

The basic idea for examining this new hybrid parallelization is taking a widely used and well understood algorithm which has already been implemented to MPI and then change it for taking advantage of the special cluster architecture. A good algorithm for such an analysis is the matrix multiplication. It is quite easy to understand, already parallelized in MPI and it is the basic algorithm under most of the implementations for linear algebra operations.

2 Background

2.1 Programming Models

2.1.1 OpenMP

OpenMP is a programming model for shared memory systems. It is based on the Fork-Join-Model for parallel execution. A program using OpenMP starts with one single thread called the *master thread*. The master thread executes in a serial region until a parallel construct is encountered. At this point a team of threads is created by the master thread which execute the parallel region. At the end of the parallel region the different threads stop working and the master thread continues the single execution of the program. In the parallel region the different threads are able to modify the same variables in memory except for a thread private variable. OpenMP is usable for a wide variety of problems but ideal for working on large arrays in memory. Every thread works on its own part of the array and communication between threads is possible by changing common variables. For further information see [1].

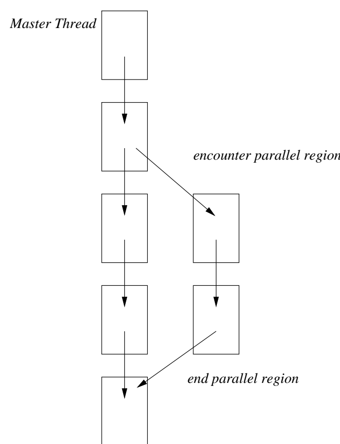


Figure 1. Fork-Join Model used by OpenMP

2.1.2 MPI

MPI [2] is a message passing interface which realizes the message passing paradigm for parallel computers. It is mainly developed for distributed memory systems but may also be used on shared memory systems if an MPI library is available. In contrast to OpenMP MPI is process based. Thus there are a number of independent processes which all have their own resources sending messages to each other through MPI library calls. Every process is identified by a unique number from the MPI system. But it is not necessary to write a special program for every single process. Normally the tasks that have to be performed by every process are very similar among the processes. There exist only slight differences. Thus it is only necessary to write one program for all processes and perform different parts of the code on different processes by using some sort of selection statements like `if` or `select case`. This programming paradigm is called SPMD (Single Program Multiple Data). The program is compiled in the usual way but must be started through an MPI start routine which starts the program as a number of new processes and distributes them among the processors in the system. For example in figure 2 you can see a small code fragment which represents a ring communication with all MPI processes. If the logical

number of the process is 0, a message is sent to process number one and then it is waiting for a message from the last process to logically close the ring. Every other process receives one message and then forwards it to the next process in the logical order of the ring. The last process with number n-1 sends its message back to the root process zero. The names root and slave processes are only logical and based on the function of the program and on the rank every process receives from the MPI system. Normally every process is assigned to a different processor by the MPI system. In this way real parallel execution is reached.

```
if (myID==0){ /* root process */
MPI_Send(DATA,COUNT,DATATYPE,1,TAG,MPI_COMM_WORLD);
MPI_Recv(DATA,COUNT,DATATYPE,n-1,TAG,MPI_COMM_WORLD,STATUS);
}
else{ /* slave processes */
MPI_Recv(DATA,COUNT,DATATYPE,myID-1,TAG,MPI_COMM_WORLD,STATUS);
MPI_Send(DATA,COUNT,DATATYPE,(myID+1) mod n,TAG,MPI_COMM_WORLD);
}
```

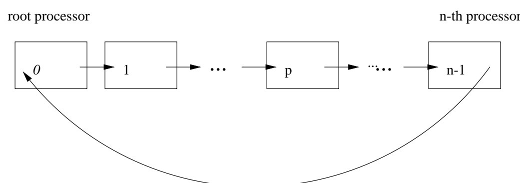


Figure 2. Example for a simple MPI ring communication with n processes

2.2 Used Libraries

2.2.1 BLACS

The BLACS (Basic Linear Algebra Communication Subprograms [3]) are a message passing library for linear algebra oriented problems. It offers an easy to use interface for matrix distribution and exists for a great variety of different hardware architectures The interface to the BLACS library is independent from the message passing libraries which are implemented on the system as the basis of the BLACS library. Thus it allows developing easy portable parallel applications for linear algebra problems.

2.2.2 BLAS

The Basic Linear Algebra Subprograms (BLAS [4]) are a library for solving elementary problems in linear algebra. Optimized versions of this library exist for almost all systems. Thus every program that uses BLAS calls will automatically perform nearly optimal on different platforms. The BLAS library consists of three parts. The Level 1 BLAS define functions for vector-vector operations like addition or dot product. The Level 2 BLAS define functions for matrix-vector operations like rank-1 updates of a matrix or matrix-vector multiplication. The highest level are the Level 3 BLAS routines. They provide matrix-matrix operations like matrix multiplication or rank-k updates of a matrix. The names Level 1,2 and 3 result from the complexity of the operations. Level 1 BLAS define $O(n)$ operations, Level 2 BLAS $O(n^2)$ operations and Level 3 BLAS $O(n^3)$ operations. To achieve a high FLOPS-rate it is necessary to use Level 3 BLAS operations which are

thus most efficient. The following table will show the reasons for this behaviour at the example of typical operations from all 3 levels.

Level	Operation	Number of floating point op.	memory accesses	ratio
1	$\alpha x + y \rightarrow y$	$2n$	$3n + 1$	$\frac{2}{3}$
2	$\alpha Ax + \beta y \rightarrow y$	$2n^2 + O(n)$	$n^2 + 3n + 2$	2
3	$\alpha AB + \beta C \rightarrow C$	$2n^3 + O(n^2)$	$4n^2 + 2$	$\frac{n}{2}$

x and y are vectors of dimension n, α and β are scalars and A,B and C are matrices of dimension n.

The important column is the ratio between the number of floating point operations and the number of memory accesses. In order to achieve as many FLOPS as possible it is important that the ratio is as high as possible because memory accesses cost a lot of time. The best ratio is delivered by the BLAS3 matrix multiplication. It is even scaling higher when the matrix dimension grows. And if in this case the algorithms can be optimized for the cache of the computer architecture to minimize the number of memory accesses BLAS Level 3 algorithms can achieve nearly peak performance on the system. This is the main reason why my examination of hybrid algorithms for linear algebra is focused on the matrix multiplication. It is the most efficient algorithm in the set of BLAS routines and is thus widely used in different applications.

2.2.3 PBLAS

The Parallel BLAS (PBLAS [5]) routines are the parallel versions of the BLAS. They are targeted for parallel vector-vector, matrix-vector and matrix-matrix operations. The PBLAS libraries use the BLACS routines for communication between processes and the BLAS for performing calculation in a process. So if highly optimized versions of the BLACS and the BLAS libraries exist on the machine the user can expect an optimal performance of his applications by using PBLAS.

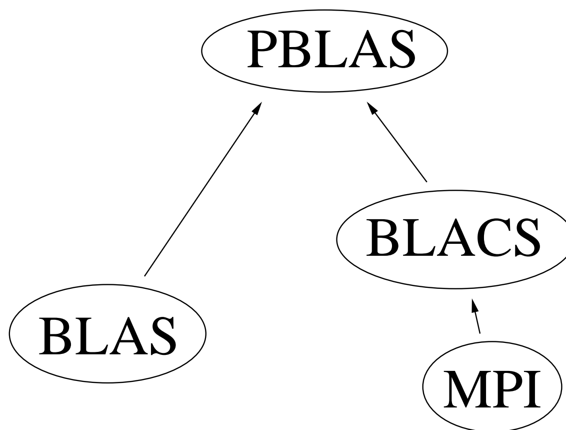


Figure 3. Relationship between BLACS,BLAS,PBLAS and MPI

2.3 Data Distribution Strategies

2.3.1 Data Distribution for the MPI based PBLAS library

For the Parallel BLAS routine DGEMM for matrix multiplication a two dimensional block cyclic data distribution model was chosen. It allows very good load balancing for various

problems and is thus the recommended distribution model for higher level ScaLAPACK [6] routines which are built upon the PBLAS routines. To show the great advantage of the block cyclic distribution model I will first introduce some more simple distribution models and discuss their disadvantages which will then directly lead to the idea of the block cyclic data distribution model. For a more detailed explanation of the following parallelization models and implementation hints see [6].

The one dimensional block column distribution: The probably most simple way of distributing a matrix is to cut it into p pieces as shown in figure 4 where p is the number of processes and give every process one of the matrix pieces. The disadvantage of this method

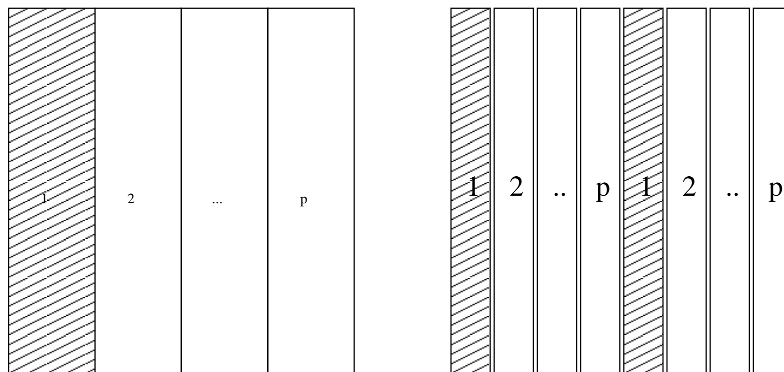


Figure 4. One dimensional block column distribution and one dimensional block cyclic distribution

is the bad load balancing. Suppose a LU-decomposition of the matrix. Then process 1 is first finished while the other processes still have to work. A possible solution would be distributing the data in a cyclic manner.

The one dimensional cyclic column distribution: In this distribution model column k is assigned to process 1+((k-1) mod p). This method allows better load-balancing, because every process has multiple parts of the matrix. The obvious disadvantage is that BLAS3 operations are no longer possible. So this distribution scheme can not be a satisfying solution. Thus we have to combine the cyclic and the block distribution schemes.

The one dimensional block cyclic column distribution: The idea is simple. Instead of distributing single columns in a cyclic manner, the cyclic distribution is applied to complete blocks. So BLAS3 operations are made possible and we have a pretty good load-balancing through the cyclic distribution scheme. But there are still situations in which the block cyclic column distribution leads to poor load-balancing (see [6]). So in general a solution is preferred where not only the columns are distributed among the processes but also the rows are distributed in a cyclic manner.

The two dimensional block cyclic distribution: In this scheme we order our p processes in a 2-dimensional $P_r \times P_c$ array and distribute the block rows and columns in a cyclic manner. In figure 5 the block cyclic distribution is shown at the example of a 5×5 matrix which shall be distributed to a 2×2 processor grid with a block size of 2×2. The two-dimensional block cyclic distribution is the recommended distribution scheme for PBLAS

and ScaLAPACK. It includes the previously examined distribution schemes as special cases and allows a nearly optimal load balancing for a great variety of problems because of its high flexibility.

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

a_{11}	a_{12}	a_{15}	a_{13}	a_{14}
a_{21}	a_{22}	a_{25}	a_{23}	a_{24}
a_{51}	a_{52}	a_{55}	a_{53}	a_{54}
a_{31}	a_{32}	a_{35}	a_{33}	a_{34}
a_{41}	a_{42}	a_{45}	a_{43}	a_{44}

Figure 5. Two dimensional block cyclic distribution of a 5×5 Matrix on a 2×2 processor grid with a block size of 2×2. The left figure shows the distribution in 2×2 blocks and the right figure shows the resulting block cyclic distribution of the blocks onto the process grid.

2.3.2 Data Distribution for OpenMP Parallelization

Data distribution in OpenMP parallelized programs does not cover the question how data is optimally distributed between different memories because as already explained in 2.2.1, all OpenMP threads access the same memory. Also the question of optimal data distribution in OpenMP shall only be discussed for the matrix multiplication. During my work I examined 2 different types of data distribution - the one dimensional block distribution and the two dimensional block distribution.

The one-dimensional block distribution: This distribution scheme is almost identical to the one dimensional block distribution when using MPI. Thus the matrix is divided into p blocks where p is the number of OpenMP threads. This distribution scheme has a poor load balancing for most of the problems in linear algebra but performs well for matrix multiplication. The idea is that when the matrices A and B shall be multiplied only the matrix B and the resulting matrix C are distributed in the one-dimensional block distribution scheme. Thus we have

$$B = (b^1, b^2, ..., b^p)$$

which leads to

$$A * B = (A * b^1, A * b^2, ..., A * b^p) = (c^1, c^2, ..., c^p)$$

So all OpenMP threads perform one matrix multiplication of always approximately the same complexity. This leads to a very efficient load balancing.

The two-dimensional block distribution: In this distribution the matrix C which holds the result of $A * B$ is separated into a $p_r \times p_c$ grid such that $p_r * p_c = p$. Every thread is assigned to one block of this grid. The matrices A and B are also separated into fitting grids in the way that every block C_{ij} equals $\sum_{k=1}^{p_c} A_{ik} * B_{kj}$. Every thread (p,q) in the $p_r \times p_c$ grid has to perform p_c matrix multiplications and $p_c - 1$ matrix additions. In figure 6 we have a simple case for this distribution. There are 4 OpenMP threads and so we divide the matrix C into a 2×2 grid in which every thread calculates one block of the matrix C. In contrast to the one-dimensional block distribution every thread has to perform two matrix multiplications and two additions (The matrix C might already contain data which has to be added to $A * B$). Thus this distribution scheme is more complex than the first one but may perform better in cached systems because the chance

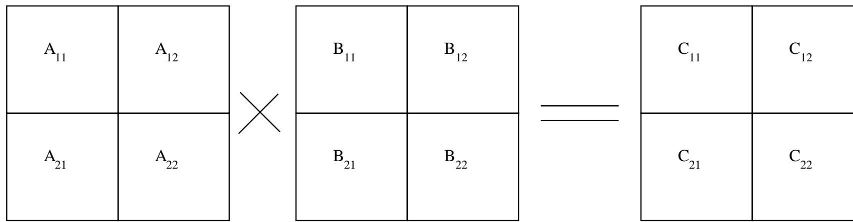


Figure 6. Two-dimensional block distribution for OpenMP

that data fits completely into the cache is higher than in the one-dimensional distribution scheme which works with larger block sizes. So it is not sure which of these two distribution schemes performs better on a given architecture.

3 Implementation of the hybrid parallelized matrix multiplication

The basis for the hybrid matrix multiplication algorithm was the PBLAS driver PDGEMM. It already provides an efficient MPI implementation of the matrix multiplication. As it is shown in Figure 3 the basis of PBLAS are calls to standard BLAS functions. So the idea was to create one MPI process per cluster node and change the calls to the standard BLAS matrix multiplication to calls to my own OpenMP parallelized matrix multiplication. After experimenting with the one dimensional and the two dimensional data distribution schemes for OpenMP matrix multiplication I decided to implement the one dimensional OpenMP distribution scheme because of the higher performance compared to the two dimensional schemes on the HP 9000 test platform. The data distribution model between the cluster nodes was implemented as the recommended two-dimensional block cyclic scheme which normally performs best for ScaLAPACK [6] routines.

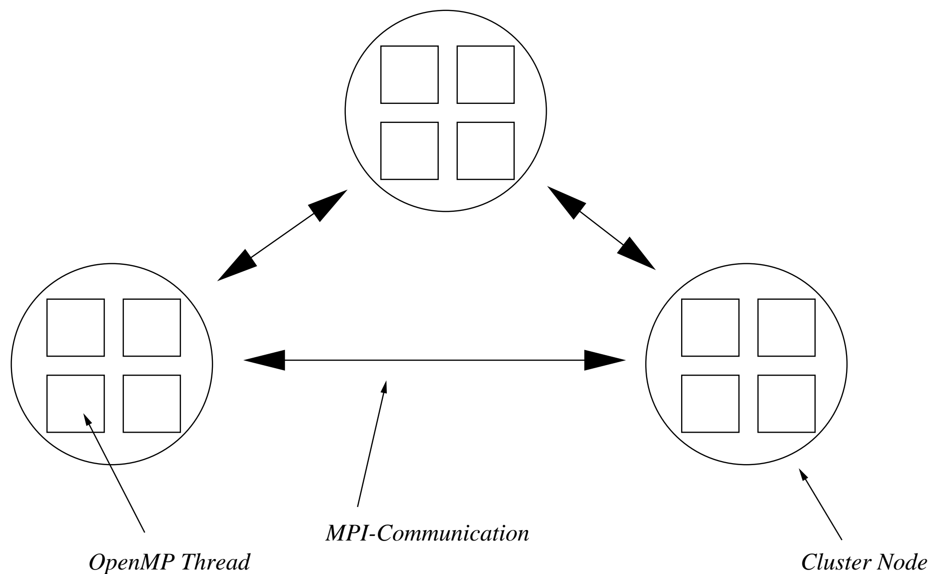


Figure 7. Idea of hybrid parallelization

4 Results of Performance Analysis

4.1 Test Environment

The tests were performed on a linux cluster based on Pentium III XEON processors with 550 MHz and on a HP 9000 N-Class Enterprise Cluster.

The linux cluster called ZAMpano is a project of the Central Institute for Applied Mathematics at the Research Centre Jülich [7]. Every node of the cluster consists of 4 processors and 2 GB RAM. The whole cluster has 9 nodes from which 8 are compute nodes and one is a service node for interactive work. The nodes are connected via Fast Ethernet and Myrinet. MPI communication uses the Myrinet. The operating system of the cluster is SuSE Linux 6.3. The compilers used for the performance analysis on the ZAMpano cluster were the Portland Group C and Fortran compilers. As BLAS library the library of the ATLAS project [9] was used because of its good performance and stability on Pentium III systems.

The HP 9000 N-Class Enterprise Cluster resides at the Technical University of Hamburg-Harburg [8] and consists of 5 nodes with 8 HP PA-Risc processors per node. The operating system used is HP-UX 11.0. The compilers used were the HP Fortran90 and C compilers. For OpenMP parallelization Guide 3.9 from KAI was used. The BLAS library used was the Veclib from HP which provides stable and good performing BLAS routines on HP systems.

4.2 Performance of the single processor matrix multiplication

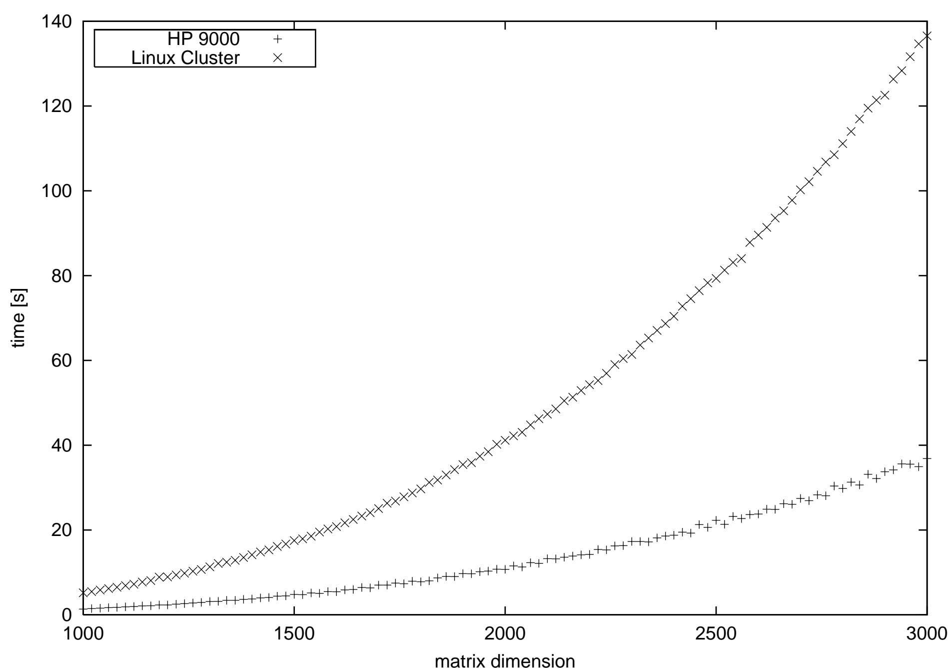


Figure 8. The single processor BLAS performance on the HP9000 compared to the Linux Cluster

Figure 8 shows the performance of the BLAS matrix multiplication on both test systems. The peak performance of the HP PA-Risc processors used in the HP 9000 is about 1.7 GFLOPS. The performance reached with the matrix multiplication is about 1.4 to 1.5 GFLOPS. So compared to the peak performance the HP 9000 performs very well. The

peak performance of the Pentium III Xeon processor is about 550 MFLOPS from which 380 to 400 MFLOPS are reached with the ATLAS BLAS matrix multiplication. Compared to the peak performance this is not as good as the HP processor with the HP Veclib library. But it is a very good value for a free BLAS library which is not developed by Intel.

4.3 Comparison of the one-dimensional and two-dimensional OpenMP data distribution

All tests for the performance of OpenMP parallelization in this chapter were performed on 4 processors thus the environment variable

```
OMP_NUM_THREADS
```

was set to 4 for both systems on which the tests were performed.

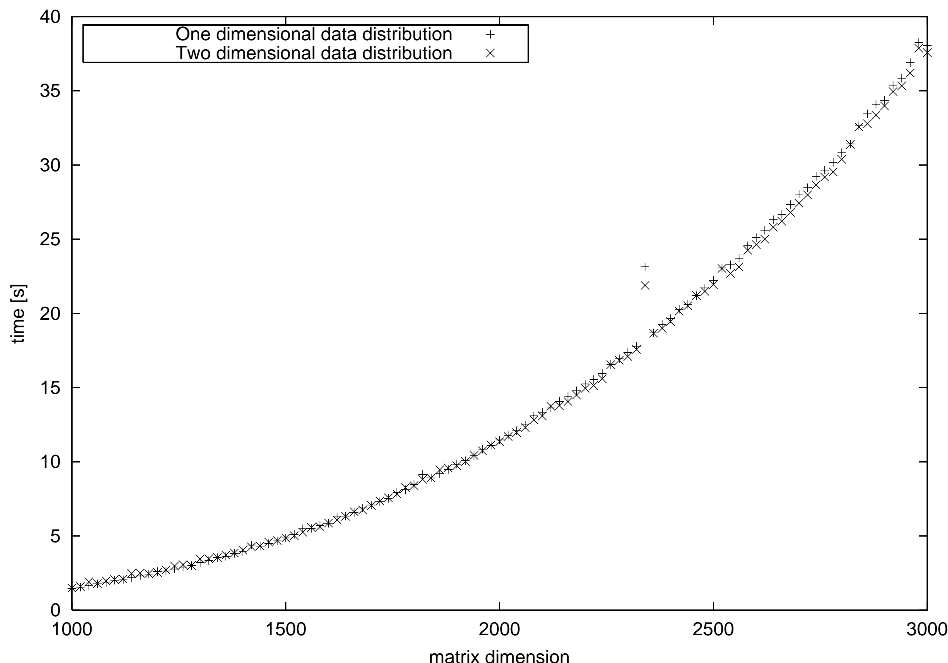


Figure 9. OpenMP performance with the one dimensional and two dimensional data distribution on the ZAMpano cluster

On the HP9000 the one dimensional data distribution obviously performs much better than the two dimensional matrix distribution (see figure 10). The great loss of performance in this case is a quite surprising result. The performance of the two dimensional data distribution is even worse than the single processor BLAS matrix multiplication as you can see in figure 11. The performance comparison of the two data distribution schemes on the Linux cluster looks much more like one would expect it. In figure 9 you can see that they both perform almost identical. But this behaviour also strongly depends on the BLAS library used. At the beginnning of the tests I used the BLAS library developed for the Intel ASCI Option Red Supercomputer instead of ATLAS and they showed a quite different behaviour on the Linux cluster. The one dimensional data distribution performed obviously better than the two dimensional scheme using the Intel libraries. But unfortunately the Intel BLAS libraries were too unstable on the Linux cluster. New libraries are being developed for Pentium III Xeon processors but they were not yet ready to use at the time when the tests were done.

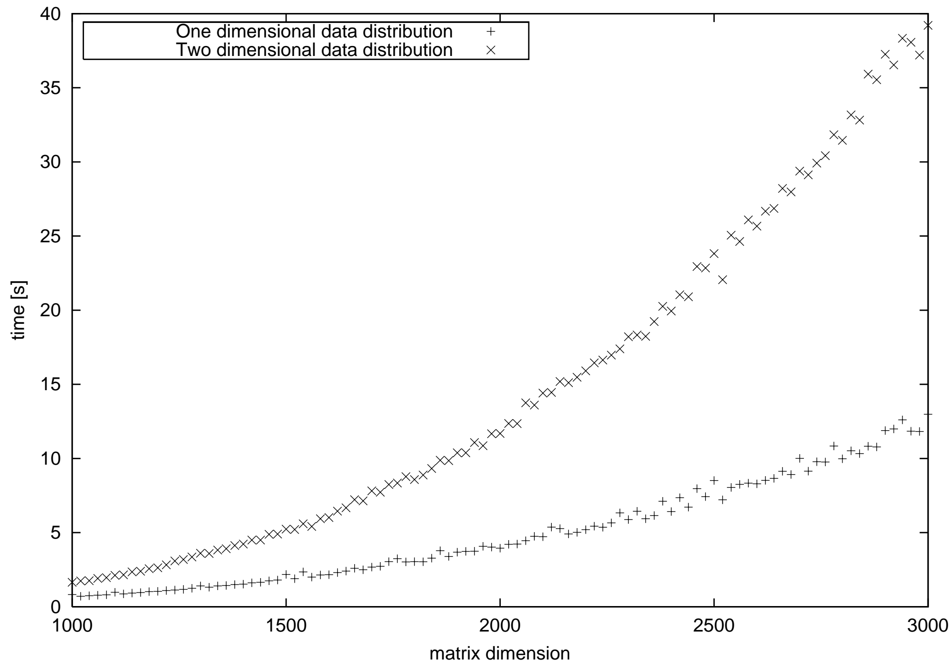


Figure 10. OpenMP performance with the one dimensional and two dimensional data distribution on the HP9000

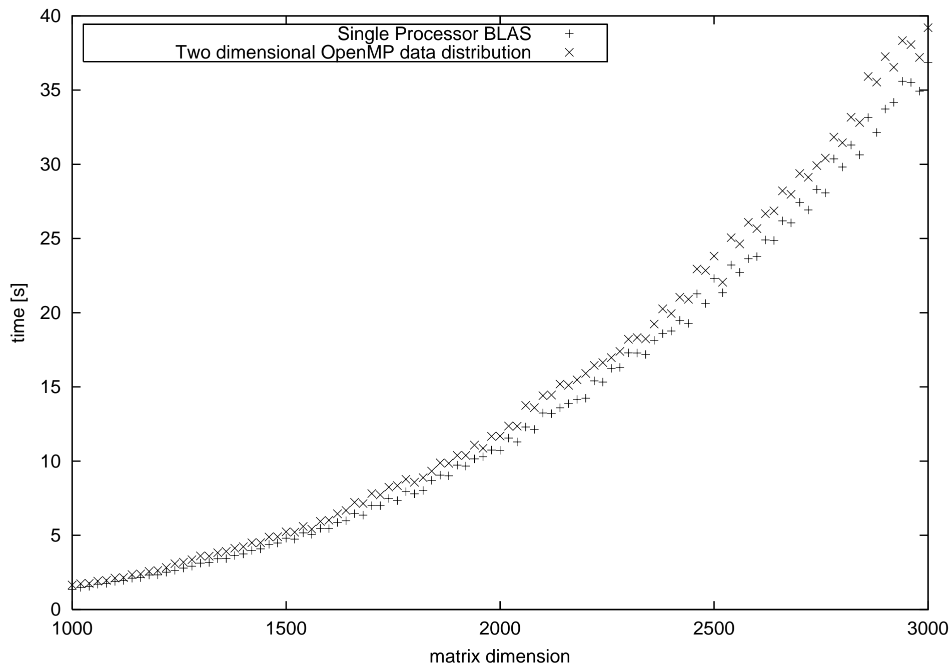


Figure 11. Comparison of the single processor BLAS performance and the two dimensional data distribution for OpenMP on the HP 9000

A possible explanation for the bad results using the two dimensional matrix distribution on the HP 9000 could be problems with the HP VECLIB libraries. But such problems can not explain this huge loss of performance. The other possibilities are problems with the cache or the OpenMP preprocessor by KAI. Investigations in both directions could be interesting for future analyses. For the following tests only the one dimensional distribution for OpenMP was chosen because of its better behaviour on the HP 9000.

4.4.1 Chosing the right block size

As explained in 2.2.3 the two dimensional block cyclic distribution for PBLAS heavily depends on the block size. So the first question is what block size is suitable for the hybrid parallelized matrix multiplication. Remember that the global distribution for the hybrid parallelization is the same as for Parallel BLAS. Only the local calls for single BLAS operations are exchanged with my own OpenMP matrix multiplication. In [6] a block size from about 64 is recommended. But this is a common suggestion and may differ from system to system and it is only a suggestion for the standard Parallel BLAS routines. So when using a hybrid parallelization the recommended block size may differ completely from the suggested number of 64. In order to illustrate the influence of different block sizes I have tested the MPI based PBLAS matrix multiplication and my own hybrid parallelized one with a 4000×4000 matrix and various block sizes on the HP 9000. Both tests were performed with 2 different process grids. For the normal PBLAS matrix multiplication a 5×5 and a 4×4 grid were taken and for the hybrid parallelized matrix multiplication a 1×5 grid with 5 OpenMP threads on every process of the grid and a 2×2 grid with 4 OpenMP threads on every process was taken.

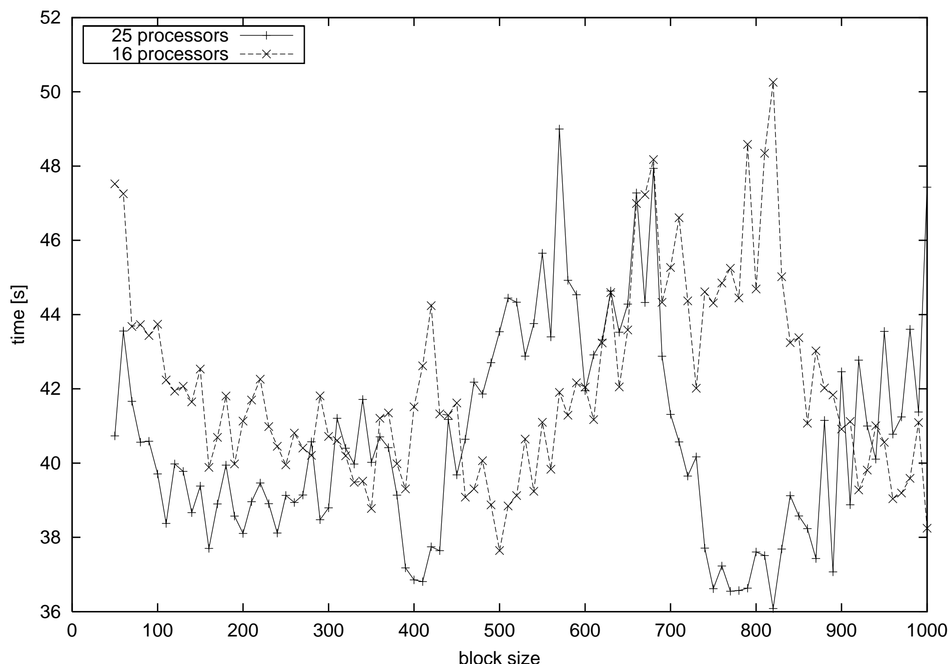


Figure 12. MPI based PBLAS matrix multiplication with various block sizes on the HP 9000 of two 4000×4000 matrices

The data points in figure 12 for the normal PBLAS matrix multiplication are connected via lines for better clarity. The results show that there is almost no speedup from 16 to 25 processors for a 4000×4000 matrix. This shows the overhead of communication which becomes significant when going from 16 to 25 processors. The ideal block sizes for 16 processors lie around 500 and 1000. This is as expectedbecause with such block sizes the matrix fits ideal into the 16 processor grid. Corresponding to this the ideal block sizes for 25 processors are 400 and 800 which lead to a good matrix distribution on 25 processors. The great range of measured times for both processor grids show the importance for chosing the right block size. For the hybrid parallelized matrix multiplication the situation

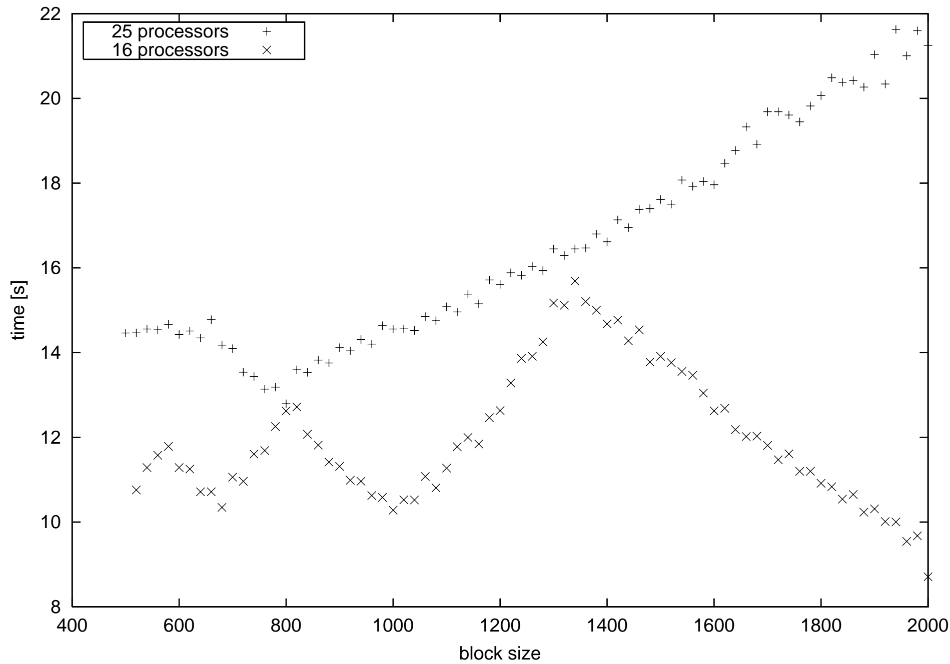


Figure 13. hybrid parallelized matrix multiplication with various block sizes on the HP 9000 of two 4000×4000 matrices

is quite similar. Again block sizes which fit to the matrix dimension and the processor grid perform well. I concentrated only on block sizes from 500 up to 2000 in this case because smaller block sizes lead to very small matrices which have to be calculated by every OpenMP thread. And block sizes greater than 2000 lead to the situation that some of the processes of the 2×2 process grid do not receive any work because of the two dimensional cylic data distribution between the processes.

Due to severe stability problems on the ZAMpano Linux cluster the corresponding results for the Linux cluster can not be given here.

4.4.2 Comparing the performance of hybrid parallelized and standard PBLAS matrix multiplication

The performance tests in this part were done on the ZAMpano cluster and on the HP 9000 cluster with 16 processors. The following tabular shows the grid sizes for the two parallelization models.

Parallelizaton Model	Number of processors	Process grid size	Local block size
MPI	16	4×4	60/400
Hybrid	16	2×2	1000

For the Hybrid parallelization on 16 processors each process forks into 4 threads. The MPI parallelized matrix multiplication from the PBLAS library was tested with the local block size of 60 on ZAMpano and with the local block size of 400 on the HP 9000 which seem to be quite good block sizes for these systems after several measurements with various block sizes. Figure 14 shows the performance of hybrid and MPI parallelized matrix multiplication on the ZAMpano cluster. The curve for the MPI parallelized matrix multiplication shows a normal behaviour and scales like $O(n^3)$ as one would expect it to do. But the hybrid matrix multiplication shows a quite strange behaviour. It scales completely different. For receiving the growth rate I performed a linear regression on the data and

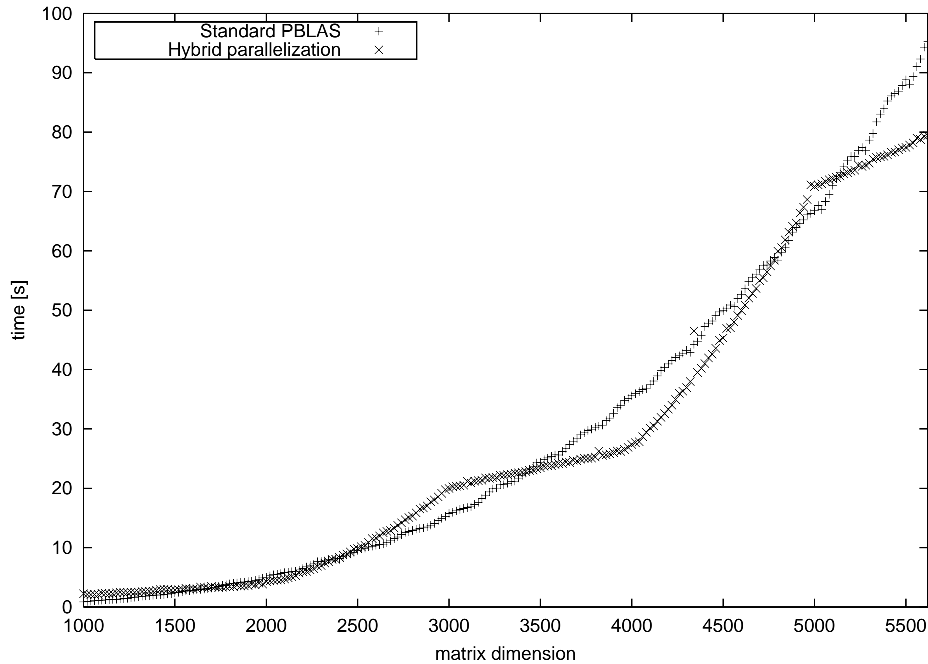


Figure 14. Performance comparison for the hybrid and MPI parallelized matrix multiplication on the ZAMpano cluster with a local MPI blocksize of 60

received following results:

Matrix dimension	Scaling rate $O(n^k)$
1000 - 2000	k = 0.9278
2000 - 3000	k = 3.9804
3000 - 4000	k = 0.9911
4000 - 5000	k = 4,3225
5000 - 5620	k = 0.9805

Due to stability issues higher dimensions than 5620 could not be calculated on the ZAMpano cluster. The behaviour of the scaling factor is quite astonishing but explicable through load inbalances in the dimensions from 2000 to 3000 and 4000 to 5000. In these ranges the first process column of the process grid receives a lot more data than the second column. This leads to high growing communication among the processes of the first process column and certainly to more calculations perfomed in the first process column in addition to the growing number of calculations and communications between the processor columns. In the matrix dimension range from 3000 to 4000 and from 5000 to 6000 the second processor column is filled with more data until it has exactly the same amount of data as the first column at the matrix dimensions 4000 and 6000. So the communication and the number of calculations in the first column stay constant but in the second column they reach the niveau of the first process column when the matrix dimension grows. As conclusion the additional time needed for multiplying the complete matrices for growing matrix dimensions in these ranges is only based on additional communication and calculations between the columns and thus the scaling factor is much smaller in the ranges from 3000 to 4000 and from 5000 to 6000. For the MPI based matrix multiplication the larger process grid and the smaller block size lead to an overall better load balancing. So these effects do not play such a role.

In figure 15 the hybrid performance of both clusters is compared. With the hybrid paral-

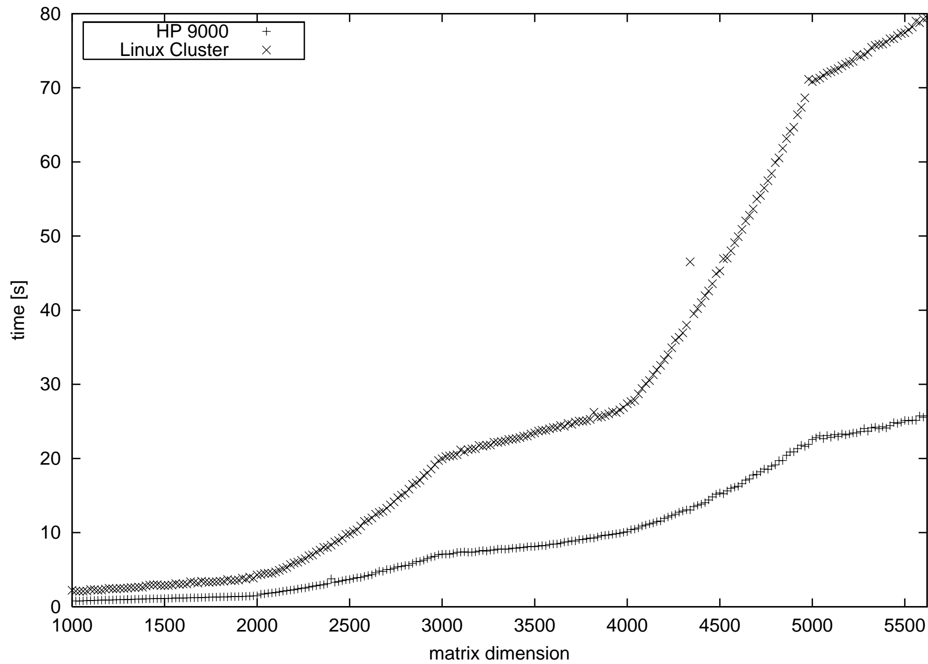


Figure 15. Hybrid parallelization with 16 processors on the ZAMpano cluster and on the HP 9000 cluster

lelization both clusters reach at a dimension of 5620 about 48% of their peak performance. The tests on the ZAMpano cluster could not go to higher dimensions because of its instability when using hybrid parallelization. But these good results change dramatically when comparing the pure PBLAS performance of both systems which only uses MPI parallelization. Compared to its peak performance the HP 9000 cluster performs awful when

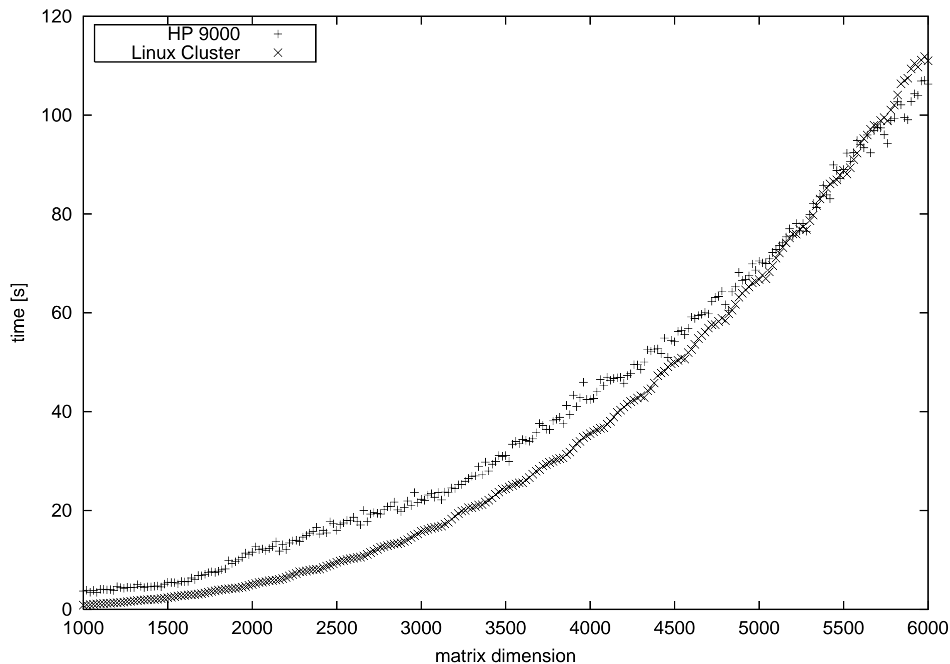


Figure 16. MPI based matrix multiplication on the HP 9000 cluster with 16 processors and on the ZAMpano cluster with 16 processors

using only MPI as you can see in figure 16. The reason for this is still unclear. So the first guess was that the communication network itself on the HP 9000 is slower than the

Myrinet on the ZAMpano cluster. But measurements of the duration of MPI_Send and MPI_Recv operations showed the opposite. So it could be a problem with the interaction between the BLACS library and the MPI implementation from HP. For hybrid parallelization the communication network does not play a great role. Most of the communication is performed through OpenMP threads which share the same memory. Only some MPI calls have to be done for inter-process communication. This is shown in figure 17 where the number of MPI_Send calls for the MPI parallelized matrix multiplication is compared with the hybrid parallelization. The number of calls on the MPI based parallelization are up to 20 times as high as the number of calls of hybrid parallelization. So the hybrid parallelization may deliver a great advantage on clusters with slow communication networks. One reason is the shorter block size when using MPI parallelization. But the main reason is the smaller process grid of 2×2 processes in the hybrid parallelization compared to the 4×4 grid when using MPI parallelization. Thus the hybrid parallelization saves a great part of the MPI communication which is necessary for the parallel matrix multiplication.

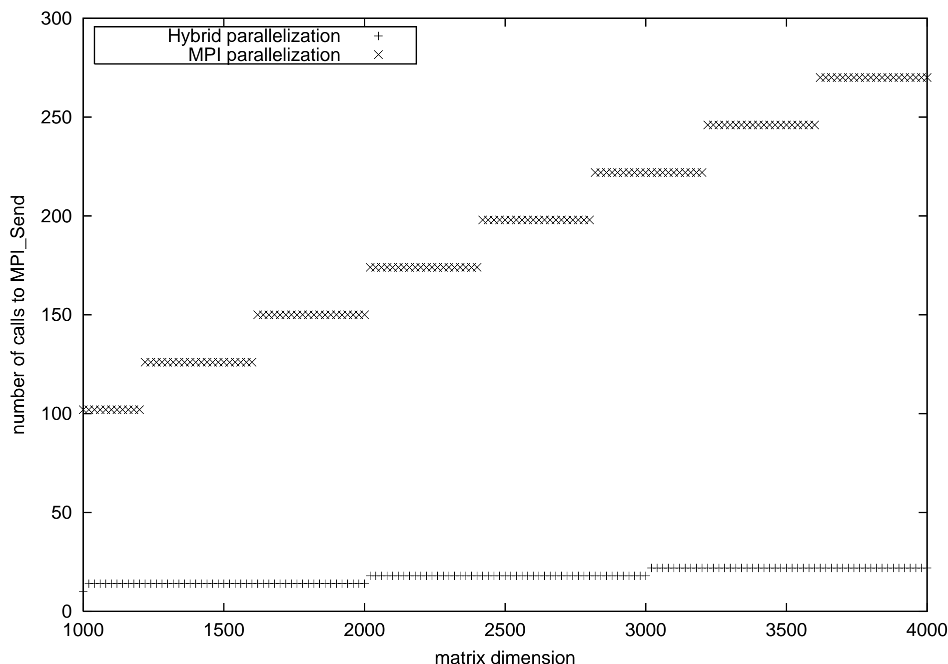


Figure 17. Comparison of the number of calls to MPI_Send for the hybrid parallelization and for the MPI parallelization on 16 processors with a block size of 400 for MPI and 1000 for hybrid parallelization

4.5 Summary

The performance of the hybrid parallelized matrix multiplication compared with the performance of the MPI based matrix multiplication depends very much on the given hardware architecture. If there is a good environment for MPI available which consists of a fast communication network like Myrinet and good MPI drivers, the performance gain does not compensate for the higher development efforts which hybrid parallelization requires. This is clearly shown for the ZAMpano cluster on which the MPI performance and the hybrid performance are almost equal (as shown in figure 14). But on multiprocessor computers which are connected via slower network technology like Ethernet, hybrid parallelization may lead to a great gain in performance because it drastically reduces the necessary MPI traffic as shown in figure 17. So the hybrid parallelization allows building well perform-

ing clusters by taking cheap SMP-systems and connecting them with standard network technology. In this way it may be easy to build well performing systems with a very small budget. But the great problem is that standard libraries like PBLAS and ScaLAPACK first have to be reprogrammed in order to take advantage of the hybrid parallelization. So the question is to create better MPI drivers which automatically recognize if a message has to be sent locally to another processor or through the communication network. On the systems I used the MPI implementation always used the network. Better MPI implementations are planned and already in beta phase which take advantage of shared memory routines by using the memory bus for internal message passing. The performance may not be as high as the hybrid performance. If the performance gap between hybrid parallelized systems and new improved MPI implementations is only small, porting standard libraries to use hybrid parallelization won't pay any more. But this has to be tested when such MPI implementations are available. Until then the hybrid parallelization is a very good alternative for using clusters which are connected only via standard network technology.

5 Acknowledgements

I want to thank my advisors, Mrs. Gutheil and Dr. Grotendorst for the support they gave me during the time at the Research Center Jülich and Mr. Egerer for his help whenever I had problems with the Unix environment. These persons made my time here a rich experience which will surely influence my further studies.

References

1. *OpenMP* http://www.openmp.org
2. *Message Passing Interface Forum* http://www.mpi-forum.org
3. *BLACS - Basic Linear Algebra Communication Subprograms* http://www.netlib.org/blacs
4. *BLAS - Basic Linear Algebra Subprograms* http://www.netlib.org/blas
5. *PBLAS - Parallel Basic Linear Algebra Subprograms* http://www.netlib.org/scalapack
6. *L. S. Blackford, J. Choi and others* ScaLAPACK User's Guide SIAM 1997
7. *ZAMpano - ZAM Parallel Nodes* http://zampano.zam.kfa-juelich.de
8. *HP 9000 N-Class Enterprise Server Cluster at the Technical University of Hamburg-Harburg* http://www.tu-harburg.de/rzt/tuinfo/ausorg/para/
9. *ATLAS - Automatically Tuned Linear Algebra Software* http://www.netlib.org/atlas

Online Visualisierung von Molekulardynamik-Simulationen

Boris Bierwald

Gerhard-Mercator-Universität Duisburg
Fachbereich 10 - Physik
E-mail: boris@eddy.uni-duisburg.de

Zusammenfassung: In diesem Paper, das als Abschlußbericht des Gaststudentenprogramms 2000 des ZAM entstand, wird zunächst das Konzept der Interaktiven Molekulardynamik-Simulation vorgestellt: die Möglichkeit, den Simulationsprozeß steuern und in ihn eingreifen zu können. Es wird dann auf konkrete Aspekte bei der Implementierung einer Online-Visualisierung eingegangen, die die Visualisierung von MD-Simulationen zur Laufzeit gestattet. Zum Abschluss wird das Grundgerüst für eine einfache Beispielrealisierung einer Online-Visualisierung vorgestellt.

1 Einleitung

Molekulardynamik-Simulationen sind in der Regel nicht-interaktive Anwendungen, bei denen der Benutzer nach ihrem Start keine Einflußmöglichkeiten auf den Ablauf hat. Anfangskonfiguration und Startparameter werden vor Simulationsbeginn festgelegt, eine Auswertung und Kontrolle der gewonnenen Daten findet erst nach Ablauf der Simulation statt. Dies gilt insbesondere für aufwendige, auf Parallelrechnern laufende Simulationen, die oft ein enormes Maß an Rechenzeit beanspruchen.
Diesem - nicht nur bei MD-Simulationen, sondern allgemein bei rechenaufwendigen Simulationen üblichem - traditionellen Konzept steht das Konzept der 'Interaktiven Simulation' oder des 'Computational Steering' gegenüber. Hier wird der Simulationsprozeß mit Möglichkeiten der Anwenderinteraktion ausgestattet. Es ist somit möglich, den Simulationsprozeß zu steuern und in ihn einzugreifen.
Die Vorteile eines solchen Konzepts sind unmittelbar einsichtig: Zunächst ist eine einfache Kontrolle über die laufende Simulation möglich, die es erlaubt, einen durch fehlerhafte Anfangskonfiguration oder falsche Parameter unbrauchbaren Simulationverlauf abzubrechen und so keine kostbare Rechenzeit zu verschwenden. Die weitergehende Möglichkeit, interaktiv in den Simulationsprozess einzugreifen und sich die Resultate dieses Eingriffs unmittelbar visuell darstellen zu lassen, kann dem Anwender einen neuen Einblick in die simulierten Prozesse liefern und ermöglicht das Testen von Simulationsparametern, Konfigurationen oder Randbedingungen, die bisher nur durch zeitaufwendige Zyklen von 'Ändern der Parameter', 'Lauf der Simulation' und 'Visualisierung der Resultate' möglich waren. Ob eine solche Interaktion möglich und sinnvoll ist, hängt jedoch nicht zuletzt vom benötigten Rechenaufwand ab, d.h. ob die unmittelbaren Resultate einer Interaktion dem Benutzer in einem angemessenen Zeitrahmen zur Verfügung gestellt werden können. Mit der immer weiter voranschreitenden Leistungssteigerung von Parallelrechnern ist jedoch abzusehen, daß auch bei sehr rechenintensiven Simulationen diese Hürde mit der Zeit kleiner wird. Es ist somit denkbar, eine auf einem entfernten Großrechner laufende

Simulation 'fernzusteuern', indem eine auf der lokalen Workstation laufende Visualisierungsapplikation als Benutzer-Interface zur Simulation dient.
Ein Beispiel für eine solche Interaktive MD-Simulation ist die Molekulardynamik-Visualisierung VMD (Visual Molecular Dynamics) [1] der Theoretical Biophysics Group der University of Illinois, die in Verbindung mit der am selben Institut entwickelten parallelen MD-Simulation NAMD eine interaktive Manipulation biomolekularer Systeme durch Auftragung von externen Kräften erlaubt. VMD unterstützt dazu Virtual Reality Techniken wie die Eingabe über haptische Interfaces zur Rückkopplung von Simulationsdaten.

2 Allgemeine Kriterien der Online-Visualisierung

Es werden nun die der Realisierung einer Online-Visualisierung zugrundeliegenden Konzepte und Kriterien erläutert. Unter dem Begriff der Online-Visualisierung wird im folgenden die Darstellung von Simulationsdaten auf entfernten Rechnern zur Laufzeit der Simulation verstanden, in Unterscheidung zur Interaktiven Simulation, bei der eine interaktive Beeinflussung der Simulation möglich ist.
Um größtmögliche Flexibilität in der Benutzung der Visualisierung zu erreichen, sollte ein Ankoppeln an eine laufende Simulation zu jedem Zeitpunkt der Simulation möglich sein. Insbesondere wäre es ein entscheindener Funktionalitätsverlust, wenn der Visualisierungsprozeß bereits zum Start der Simulation existieren müßte, wie dies in der derzeitigen Implementierung von VMD der Fall ist. Es sollte möglich sein, sich mehrfach an eine Simulation an- und wieder abzukoppeln, um ihren Verlauf punktuell kontrollieren zu können.
Als ein weiteres Kriterium muß die Performance der Simulation durch das neue Feature soweit wie möglich unbeeinträchtigt bleiben. Rechenzeit ist in der Regel knapp, und eine Erweiterung wird meist nur dann eingesetzt, wenn sie zumindest in den wichtigsten Leistungsmerkmalen einer Anwendung keine Nachteile mit sich bringt. Die Idee, in eine Simulation die notwendigen Routinen zur Online-Visualisierung fest einzubauen, und diese nur bei Bedarf über eine Visualisierungs-Anwendung zu nutzen, unterstreicht dieses Problem und teilt es in zwei Teilaspekte. Zum einen die Performance im gewöhnlichen Simulationsablauf, ohne Nutzung der Visualisierungsmöglichkeiten, und zum anderen die Performance bei aktiver Visualisierung. Performance-Einbußen bei aktiver Visualisierung sind im Falle einer reinen Online-Visualisierung von geringerer Bedeutung, da der größte Teil eines Simulationslaufs in der Regel ohne Visualisierung stattfinden wird. Prinzipiell sind bei der Übertragung von Daten Leistungseinbußen unvermeidbar, da mit Netzwerkverkehr im Vergleich zur Rechnergeschwindigkeit große Latenz- und Übertragungszeiten verbunden sind.
Die Portabilität der Visualisierungs-Anwendung scheint wichtig, da ihre Funktionen in unterschiedlichen Benutzerumgebungen zur Verfügung stehen sollten. Auf der Implementierungsebene folgt daraus die Forderung nach Konvertierung von architekturspezifischen Darstellungsformaten wie Byteorder-Konvertierung und der Einführung systemunabhängiger Datentyplängen.
Anforderungen an den Quellcode sollten die Erweiterbarkeit der einzelnen Komponenten und die einfache Integrierbarkeit der Kommunikationsroutinen in vorhandene Simulationen sein. Eine Erweiterbarkeit stellt sicher, daß das bereits vorhandene Gerüst an unterschiedliche Anwendungsfälle angepasst werden kann, und neue Funktionen hinzugefügt werden können. Dies schließt die Erweiterung der Visualisierungs- sowie der Kommunikationselemente ein.

Die Beispielrealisierung wurde für **DMMD** (Distributed Memory Molecular Dynamics)

[2] geschrieben, einer am ZAM entstandenen modular aufgebauten, in Fortran 90 geschriebenen parallelen Molekulardynamiksimulation, die zur Zeit für die Simulation von Mischungen von Atomen mit Lennard-Jones Potentialen genutzt wird. Ein durch das verteilte Konzept von DMMD auftauchendes Problem ist die Übertragungsstrategie der auf den einzelnen Knoten verteilt vorhandenen Daten. Der modulare Aufbau von DMMD hat die Integration der Kommunikations-Schnittstelle in die Simulation durch deren Realisierung als Fortran 90 Modul erleichtert.

3 Kommunikation

Die zwischen den Kommunikationspartnern ausgetauschten Nachrichten wurden in verschiedene Nachrichtentypen funktional unterteilt. Dazu wird jede Nachricht mit einer ID gekennzeichnet, die im Nachrichten-Header enthalten ist und Auskunft über ihren Inhalt gibt. Anhand dieser ID, die auf der Empfängerseite abgefragt werden kann, können die Nachrichten entsprechend behandelt werden. Nachrichten sind prinzipiell in beide Richtungen möglich. Bisher sind zwei Nachrichtentypen zum Verbindungsaufbau, sowie ein Nachrichtentyp zur Übertragung von Koordinatensätzen implementiert. Der Verbindungsaufbau wird über das Versenden eines SIZE-Paketes von der Simulation abgewickelt, das globale Parameter wie die Anzahl an Atomen und die Größe des Simulationsgebietes enthält. Nachfolgend wird in der gleichen Richtung ein PROPERTIES-Paket versandt, das Informationen über die einzelnen Atome zur Verfügung stellt, wie deren Radius und Typ, um diese Eigenschaften nicht mit jedem Koordinatensatz erneut übertragen zu müssen. Das Nachrichtenpaket für Koordinatensätze indiziert die Atome zusätzlich, um auch bei Umordnung der Reihenfolge (was z.B. bei parallelen MD-Simulationen möglich ist) der Atome im Speicher eine eindeutige Identifizierung einzelner Atome und ihre Zuordnung zu der Liste der Eigenschaften zu ermöglichen.

3.1 Kommunikationsbibliothek

Für die Kommunikation zwischen Simulation und Visualisierung wurde die Bibliothek **Visit** (Visualization Interface Toolkit) [3] verwandt, die für einen ähnlichen Zweck am ZAM im Rahmen eines Teilprojekts des "Gigabit Testbed West" entstand. Als Kommunikationsprotokoll unterhalb der Anwendungsschicht (OSI-Modell, siehe z.B. [4]) kommt die TCP/IP-Protokoll-Suite zum Einsatz, da eine plattformübergreifende Unterstützung

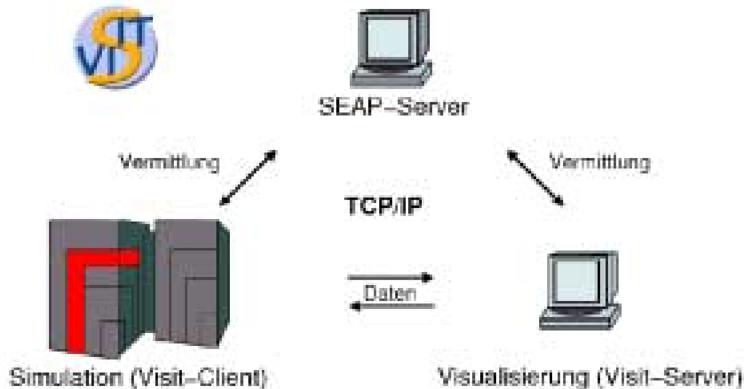


Abbildung 1. Visit-Kommunikationsmodell

vorhanden ist. Visit ist prinzipiell jedoch dazu konzipiert, auch andere Protokolle zur Datenübertragung zu unterstützen. Visit stellt ein Application Programming Interface (API) für C, Fortran und Perl zur Verfügung.
Zum Verbindungsaufbau stellt Visit den Host- und Portvermittlungsservice Seap (Service Announcement Protocol) bereit, um die direkte Angabe von Host- und Portadresse zu umgehen. Zur Datenübertragung unterstützt Visit bis zu vierdimensionale Arrays, eine Byteorderkonvertierung findet automatisch statt.
Das Visit-Kommunikationsmodell ist ein Client-Server-Modell, in dem die Simulation die Client-Seite einnimmt, die Visualisierung die Server-Seite. Die Kommunikation geht grundsätzlich vom Client aus. Das API ist aufgeteilt in ein Client- und ein Server-API, so daß die Rollen im Kommunikationsprozeß eindeutig festgelegt sind. Die Vermittlungs- und Authentifizierungsinstanz übernimmt der Seap-Server, der typischerweise auf einem dritten Host im Hintergrund läuft, siehe dazu Abb. 1. Bei diesem meldet sich ein VisualisierungsServer mit einer Service-Kennung und einem Passwort ('Cookie') an, worauf sich ein Simulations-Client unter Angabe dieses Service-Cookie-Paares über den Seap-Server mit der Visualisierung verbinden kann.
Da Kommunikationsanfragen im Visit-Modell grundsätzlich vom Client ausgehen, liegt dem Kommunikationsschema zwingend ein Polling-Betrieb des Clients zugrunde, womit hier ein in regelmässigen Abständen durchgeführter Test auf Vorhandensein eines Visualisierungs - Servers bezeichnet wird.

3.2 Server-Seite

Folglich besteht die Aufgabe des Servers darin, nach einer Initialisierung auf eingehende Verbindungsanfragen zu warten.
Dazu ist die Server-Applikation in zwei funktionale Einheiten aufgeteilt, die jeweils durch einen Thread realisiert wurden und so unabhängig voneinander ausgeführt werden: eine Kommunikationseinheit und eine Visualisierungseinheit. Um diese sauber voneinander zu trennen, kommunizieren beide über eine Queue. Der Kommunikationsthread gibt so die empfangenen Daten an den Visualisierungsthread weiter, der diese verarbeitet. Der Vorteil einer solchen Implementierung besteht in der prinzipiellen Austauschbarkeit jeder der Komponenten, z.B. falls eine andere Kommunikationsbibliothek erforderlich sein sollte. Es wurde aus Portabilitätsgründen das Posix-Thread-API verwandt.

3.3 Client-Seite

Der deutlich mehr Überlegung erfordernde Teil der Kommunikations - Implementierung liegt allerdings bei der Simulation, vor allem um den oben genannten Anforderungen der Vermeidung von Performance-Einbußen zu genügen.
Im Falle einer strikt auf dem Distributed-Memory-Konzept basierenden verteilten MD - Simulation wie DMMD muß vermieden werden, sämtliche Konfigurationsdaten auf einem Knoten zu sammeln. Mit der Eigenschaft der massiv-parallelen Architektur der Cray T3E, daß jeder einzelne Knoten den logischen Endpunkt einer TCP/IP-Verbindung darstellt, und nicht der gesamte Rechner, ergibt sich die Frage nach der Übertragungsstrategie der lokalen Daten. Verschiedene Möglichkeiten sind hier denkbar (siehe Abb. 2).
Einerseits kann prinzipiell von jedem Knoten eine Netzwerkverbindung zur Visualisierung aufgebaut werden, über die die jeweils lokalen Daten unabhängig voneinander übertragen werden. Praktisch scheidet diese Möglichkeit in unserem Fall aus, da das Visit-API die

gleichzeitige Verbindung einer Visualisierung mit mehreren Clients nicht vorsieht. Für einen solchen Ansatz wäre auch eine zusätzliche Komponente zur Verwaltung der einzelnen Verbindungen notwendig. Diesem Ansatz steht ein weiteres Problem im Weg, daß die Anzahl offener TCP-Sockets eines T3E-Systems über die Anzahl maximal geöffneter File-Deskriptoren begrenzt ist, sowohl global wie auch pro Benutzer (auf den Cray T3E-Rechnern des ZAM auf 255). Dadurch wird eine maximale Anzahl an Knoten gesetzt, mit der dieser Ansatz lauffähig wäre. Vorteil einer solchen Lösung wäre allerdings eine komplett parallele und unsynchronisierte Durchführung der Kommunikation auf der Client-Seite.
Die zweite Möglichlichkeit besteht darin, die Kommunikationsverbindung zwischen dem Server und nur einem Master-Knoten des Clients aufzubauen, der zum Kommunikationszeitpunkt sequentiell die Daten aller Knoten über MPI erhält und diese jeweils verschickt. Offensichtlicher Nachteil dieser Möglichkeit ist die sequentielle Übertragung der Daten, sowie die Synchronisation der Knoten, die einen Performance-Bottleneck darstellen würden.
Eine dritte Strategie, die beide Ansätze verbindet, wäre schließlich die Realisierung eines Kommunikations - Prozesses, der für jeden Simulations-Prozeß auf der Client-Seite einen neuen Thread anlegt, und die Daten so unabhängig überträgt (siehe auch [5]).
Es wurde bisher nur die Strategie implementiert, die Daten in einer Anzahl von Sendevorgängen zu übertragen, die der Anzahl an Knoten entspricht. Der simulationsinterne Datenaustausch findet über MPI statt. Abb. 3 zeigt hierzu die für einen Zeitschritt der Simulation benötigte Zeit über der Anzahl an Prozessoren aufgetragen. Die Messung wurde für eine Mischung aus zwei Lennard-Jones-Fluiden mit einer Gesamtteilchenzahl von 2048 und einer Polling-Frequenz von eins (was bedeutet, daß in jedem Zeitschritt ein Polling bzw. Kommunikation stattfindet) durchgeführt. Die drei Kurven entsprechen der ursprünglichen DMMD-Version und der Version mit implementiertem Online-Visualisierungs-Modul mit und ohne vorhandenem Visualisierungs-Server. Wie zu erwarten war, steigt bei vorhandenem Server die Kommunikations-Zeit der aktuellen Implementierung mit wachsender Prozessor-Anzahl deutlich an. Bereits ab acht Prozessoren wächst die absolute zur Ausführung eines Zeitschritts benötigte Zeit an, so daß diese Version lediglich für sehr kleine Knotenzahlen anwendbar ist. Ein weiterer Performance-Nachteil ist die im Vergleich zur reinen Simulationszeit (untere Kurve) nicht zu vernachlässigende Polling-Zeit (mittlere Kurve). Diese beträgt relativ konstant 30ms und entsteht durch die Dauer eines Aufrufs der Visit Connect-Routine. Diese Zeitdauer ist ausschließlich auf der Cray T3E zu beobachten, auf anderen Architekturen liegt sie im Bereich von 1-3 ms. Es ist anzunehmen, daß sie auch auf dem Cray-System auf ähnliche Werte reduziert werden kann. Der Synchronisations-Overhead spielt im Vergleich dazu (zumindest bis zu 32 Knoten) eine

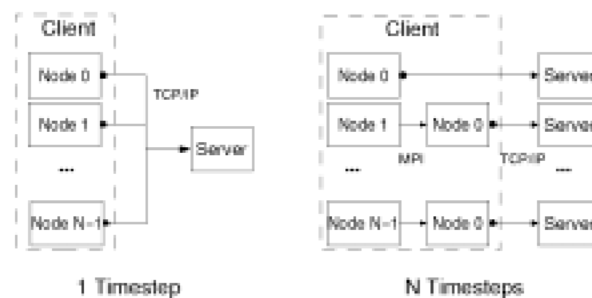


Abbildung 2. Strategien der Übertragung der auf den Knoten verteilten Daten.

untergeordnete Rolle. Ein Vergleich dieser Ergebnisse mit einer der andere Implementierungsstrategien wäre interessant, konnte jedoch aus Zeitgründen nicht durchgeführt werden. Bei diesen Ergebnissen ist jedoch einerseits zu bedenken, daß in der Regel das Polling sowie - bei aktiver Visualisierung - eine Übertragung von Daten nicht mit jedem Simulations-Zeitschritt durchgeführt wird, sondern nur mit einer festzulegenden Polling-Frequenz. Andererseits wird, wie bereits oben erwähnt, die Visualisierung selten während eines gesamten Simulationslaufs aktiviert sein. Typisch wäre wohl eher die punktuelle Benutzung der Visualisierung. Beide Aspekte relativieren die Performance-Einbußen.
Das Polling ist derzeit über einen Aufruf der 'Connect'-Funktion des Visit API realisiert. Dieser kehrt sofort zurück, falls beim Seap-Server kein entsprechender Service angemeldet ist. Eine andere Strategie, um die Polling-Zeit zu reduzieren, wäre eine Realisierung der Simulation als Server-Seite, und entsprechend der Visualisierung als Client-Seite. Auf dem nun als Server fungierenden Simulations-Host würde ein 'horchender Socket' geöffnet, der auf Verbindungen warten würde und mittels nichtblockierender Kommunikationsroutinen (ein Beispiel hierfür wäre die Select-Funktion des Unix Netzwerk API) abgefragt werden könnte. Dazu wäre allerdings eine vollkommen andere Kommunikationsstruktur erforderlich, als sie hier unter Verwendung der Visit-Bibliothek beschrieben wird. Dieser Ansatz vermeidet zwar Round Trip Time (RTT) zum Seap-Server; ob er deutliche Timing-Vorteile bringt, müßte allerdings erst noch gezeigt werden.

4 Visualisierung

Bei der Realisierung der 3D-Visualisierungs-Komponenten wurde das am ZAM von Stefan Birmanns und Herwig Zilken entwickelte Toolkit **SVT** (Small VR Toolkit) eingesetzt. Diese C++ Klassenbibliothek ermöglicht eine schnelle und unkomplizierte Entwicklung von OpenGL-Visualisierungen ohne den bei direkter Benutzung des (üblicherweise verwendeten) GL Utility Toolkits GLUT anfallenden Overhead, und unterstützt diverse Virtual Reality Erweiterungen wie 3D-Brille und Headtracking. Zusätzlich stellt es über Verwendung der User Interface Bibliothek GLUI Widgets zur Erzeugung von graphischen Benutzeroberflächen (GUIs) bereit.
Die Visualisierung der MD-Konfigurationen erfolgt über eine Darstellung der Atome durch einfache 3D-Kugeln. Dies ist im Fall einer Darstellung der von DMMD simulierten Systeme zunächst ausreichend, da DMMD zur Zeit ausschließlich kurzreichweitige, anisotrope

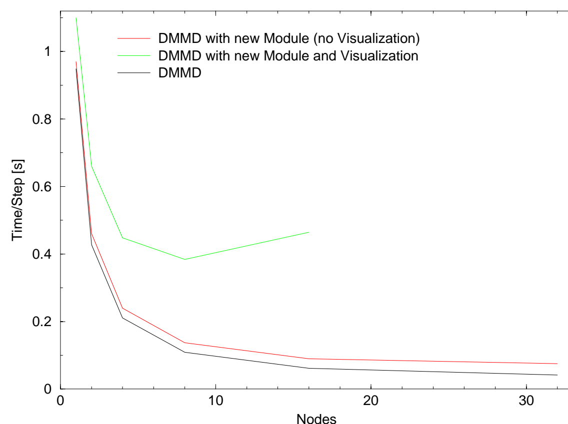


Abbildung 3. Performance des DMMD-Moduls zur Online-Visualisierung

Lennard-Jones Wechselwirkungspotentiale verwendet,

$$V_{LJ}(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \quad ,$$

über die den Atomen annähernd der Radius

$$r_{min} = \frac{\sigma}{2}2^{1/6}$$

zugeordnet werden kann.

Die Atome sowie die Menge aller Atome basieren auf allgemein gehaltenen Klassen. Die Radien der Atome werden relativ zu ihren Abständen maßstabsgetreu dargestellt, ihre Farben drücken ihren mit den Atom-Eigenschaften übermittelten Typ aus. Die Anzahl der dargestellten Atome entspricht dem beim Verbindungsaufbau empfangenen Wert.
Ein Aspekt bei der Entwicklung der Visualisierung war die Möglichkeit der Darstellung großer Teilchenzahlen, die auch professionelle Visualisierungshardware bei Verwendung nicht-optimierter Algorithmen vor ein Performance-Problem stellt. Aus diesem Grund wurde eine bereits optimierte SVT-Klasse zur Darstellung von Kugeln benutzt, die nicht auf der Sphere-Klasse der OpenGL Utilities-Bibliothek aufbaut, sondern einen optimierten Tesselations-Algorithmus sowie Back Face Culling verwendet. Der Tesselations-Algorithmus setzt die Kugeln aus einer variablen Anzahl von Polygonen (Dreiecken) zusammen, die von der derzeitigen relativen Größe des Kugel-Objekts im Verhältnis zum sichtbaren Bildausschnitt abhängt. Somit wird eine weiter entfernte Kugel aus weniger Polygonen zusammengesetzt als Kugeln im Vordergrund (zu sehen in Abb. 4), was deutliche Performance-Steigerung bei großen Kugelzahlen, folglich also kleinen Kugelradien, bedeutet. Es wurden versuchsweise zu den drei vorhandenen Detailstufen („Levels of Detail“) weitere Stufen für gröbere Darstellungen hinzugefügt, was aber keine deutlichen Performance-Steigerungen lieferte. Über das Back Face Culling werden nur diejenigen Polygone gezeichnet, deren Flächennormale in Richtung der Betrachterposition zeigt.
Allerdings sind der durch diese Optimierungen gesteigerten Anzahl von maximal darstellbaren Atomen Grenzen gesetzt - bei einer Zahl von einigen tausend ist selbst auf dem SGI Onyx2 - Graphikrechner des ZAM eine stockende Darstellung zu beobachten. Teilchenzahlen jenseits dieser Größenordnung müßten mit Hilfe anderer Visualisierungsverfahren, z.B. über Dichte- oder Geschwindigkeitsverteilung dargestellt werden. Eine Visualisierung durch Kugeln würde dem Betrachter darüber hinaus bei deutlich höheren Teilchenzahlen auch kaum mehr aussagekräftige Informationen liefern.

5 Dokumentation der entstandenen Anwendung

Dieser Abschnitt gibt eine Übersicht über die Installation und den derzeitigen Funktionsumfang der Online-Visualisierung für DMMD.
Die Online-Visualisierung besteht aus zwei Komponenten: einem Fortran 90-Modul für DMMD auf der Client-Seite (hier werden grundsätzlich die in Abschnitt 3.1 eingeführten Begriffe benutzt), und der in C++ geschriebenen Visualisierung als Realisierung der Server-Komponente. Zusätzlich steht der Code eines minimalen Test-Clients zur Verfügung, der Koordinatensätze erzeugt und an einen vorhandenen Server sendet.
Es wurde versucht, den Quellcode ausführlich zu kommentieren, um Änderungen und Erweiterungen zu erleichtern. Die Kommentare im Quellcode des Servers erlauben die

Dokumentationserstellung mit dem Dokumentationssystem DOC++. Diese wird mit einem

```
make doc
```

im entsprechenden Verzeichnis generiert. Auf eine Beschreibung der Programm- und Klassenstruktur wird hier deshalb verzichtet.

5.1 Installation

Auf der **Client-Seite** ist zunächst das Visit-Paket in das DMMD-Verzeichnis zu installieren. Dies geschieht wie üblich durch

```
./configure && make
```

im Visit-Unterverzeichnis. (Es reicht aus, wenn die Sourcen für die Visit-Bibliothek kompiliert werden. Bei der Kompilation der Demo-Programme kann es zur Zeit unter Umständen noch zu Problemen kommen.) Das aus dem File *'imd.F90'* bestehende Fortran-Modul ist in das DMMD-Verzeichnis zu kopieren und das DMMD-Makefile anzupassen, indem das Objekt-File *'imd.o'* zur Liste der DMMD-Objekt-Files (Variable `SUBOBJS`) hinzugefügt wird und die Variable `LIBS` um die visit-Bibliothek erweitert wird (`-lvisit`, evtl. mit Pfadangabe). Anschließend ist in der DMMD-Datei *'main.F90'* das imd-Modul mittels der Anweisung '`use imd`' zugänglich zu machen und an das Ende des Main-Loops der Funktionsaufruf '`call imd_send_coords()`' anzufügen.
Es können nun die Visit-spezifischen Parameter, insbesondere das zur Authentifizierung benötigte Schlüsselpaar `service` und `cookie`, im Quellcode des imd-Files verändert werden. Weiter ist die Angabe der Sendefrequenz von Konfigurations-Koordinatensätzen zur Zeit nur über die Variable `n_send_coords` einzustellen, die die Anzahl an Zeitschritten angibt, nach denen die Simulation jeweils versucht, mit einem eventuell vorhandenen Visualisierungs-Server zu kommunizieren.
Nach diesen Modifikationen muß DMMD wie üblich mit

```
make dmmd
```

übersetzt werden. Das DMMD-Modul wurde bisher ausschließlich mit Konfigurationen von zwei Atomsorten im Atom Decomposition Modus auf der Cray T3E getestet.
Der Testclient wurde auf allen Systemen, die auch der Server unterstützt, getestet. Er wird durch ein '`make`' im entsprechenden Unterverzeichnis kompiliert.
Die **Server-Seite** besteht aus der Visualisierungs-Applikation. Zusätzlich werden die beiden Bibliotheken Visit und SVT benötigt. Sie sind in das Basisverzeichnis des Servers zu kopieren und evtl. neu zu kompilieren. Nach Eintragen der lokalen Plattform in die Datei *machine.mk* wird die Visualisierung mit einem

```
make
```

im Unterverzeichnis übersetzt. Einige SVT-Parameter wie der Wechsel zwischen Fullscreen- und Window-Modus sowie die Unterstützung des Stereo-Modus und Headtracking können in der Datei *.svtrc* im Server-Verzeichnis verändert werden. Der Server wurde unter Irix6/Onyx2, Solaris2.7/Sparc und Linux2.2.10/i386 getestet.
Sowohl auf dem Client- wie auch auf dem Server-Host muß im Home-Verzeichnis eine Datei *'.visitrc'* erstellt werden. Eine Vorlage sollte sich im Visit-Verzeichnis befinden, in ihr ist Hostname und Portnummer eines vorhandenen Seap-Servers einzutragen. Sollte dieser noch nicht existieren, ist er mit dem Perl-Script *'seap_server'* des Visit-Paketes zu starten.

Auf der Client-Seite wird DMMD nach erfolgreicher Neuübersetzung inclusive des neuen Moduls wie gewohnt gestartet. Es wird automatisch alle `n_send_coords` Zeitschritte versucht, eine Verbindung zu einem Visualisierungs-Server aufzubauen. Falls dies gelingt, werden in entsprechenden Abständen die Koordinaten-Datensätze aller Atome versandt. Der Server wird mit

```
server <service> <cookie>
```

gestartet, oder, falls die Default-Werte übernommen werden sollen, mit

```
server  .
```

Die Spezifizierung der Authentifizierungsparameter über die Kommandozeile stellt nur eine vorübergehende Lösung dar, und setzt prinzipiell den durch sie realisierten Sicherheitsmechanismus außer Kraft, da der Authentifizierungsschlüssel jedem lokalen Nutzer des Systems über ein '`ps`' zugänglich ist.
Nach dem Betätigen des Connect-Buttons meldet sich die Visualisierung beim Seap-Server an und wartet auf die Anfrage eines Clients. Der Server blockiert momentan in diesem Zustand, bis ein Client vorhanden ist.
In Abb. 4 ist die graphische Benutzeroberfläche (GUI, Graphical User Interface) der Applikation zu sehen. Die einzelnen Benutzerelemente werden im folgenden kurz erläutert.

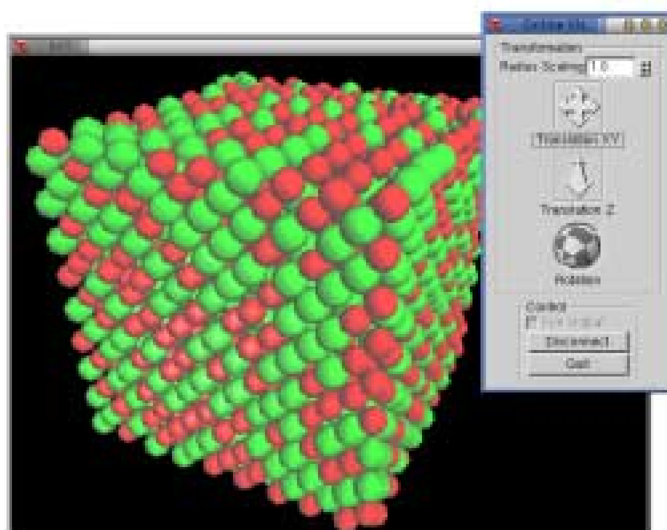


Abbildung 4. GUI der Visualisierungs-Applikation

Der **Radius-Scale**-Spinner dient zur Skalierung der Kugelradien, wobei ein Faktor von eins eine zu den Atomabständen maßstabsgetreue Skalierung bedeutet. Diese Option ist mitunter bei großen Atomzahlen sinnvoll, wo eine Reduzierung der Radien die Übersichtlichkeit erhöhen kann.
Die zwei **Translation**s-Elemente und das **Rotation**s-Element dienen zur Veränderung der Betrachterposition bzw. Verschiebung der Atom-Konfiguration. Die Geschwindigkeit der Translationen kann durch Halten der Shift- bzw. Ctrl-Taste erhöht bzw. verlangsamt werden. Die XY-Translation kann durch Halten der Alt-Taste auf eine Richtung eingeschränkt werden. Das Rotations-Element kann durch Halten der Ctrl- bzw. Alt-Taste auf

horizontale bzw. vertikale Drehungen eingeschränkt werden.
Ein Aktivieren der **File Out**-Checkbox schreibt sämtliche empfangenen Koordinatensätze der folgende Verbindung indiziert in ein ASCII-File mit dem Namen *'imd.dat'*. Vorhandene Files werden überschrieben.
Mit dem **Connect/Disconnect**-Button wird die Visualisierung beim Seap-Server an- bzw. abgemeldet, also eine Verbindung entweder versucht herzustellen oder getrennt. Der **Quit**-Button beendet den Visualisierungs-Server, ohne die Simulation anzuhalten.

6 Zusammenfassung und Ausblick

Durch die Implementierung einer einfachen Online-Visualisierung für die verteilte MD-Simulation DMMD ist einerseits eine portable Visualisierungsanwendung entstanden, die im derzeitigen Zustand, besonders unter Verwendung der am ZAM vorhandenen Virtual Reality-Ausstattung, zur anschaulichen Darstellung laufender Simulationen dienen kann. Andererseits besteht die Möglichkeit, dieses Grundgerüst um weitere Funktionen zu erweitern. Denkbar sind hier in erster Linie Komponenten zur Interaktion mit der Simulation, aber auch erweiterte Möglichkeiten der Darstellung. Darüber hinaus wäre es denkbar, die Visualisierung mit geringem Aufwand in andere MD-Simulationen zu integrieren.
Es wurde aber auch gezeigt, daß in einigen Bereichen weitere Untersuchungen notwendig sind, um effektivere Lösungen zu finden. Beispiele sind die Performance-Auswirkungen auf die Simulation, andere Konzepte der Kommunikationsstruktur sowie Konzepte der graphischen Darstellung sehr großer Teilchenzahlen.

Literatur

1. W.Humphrey, A.Dalke, K.Schulten: VMD: Visual Molecular Dynamics. Journal of Molecular Graphics 14:33-38, 1996. Siehe auch http://www.ks.uiuc.edu/
2. N.Attig, M.Lewerenz, G.Sutmann, R.Vogelsang: DMMD - A Molecular Dynamics Program for MPP-Systems. Proceed. of the 5th European SGI/CRAY MPP - workshop, Bologna, 1999.
3. Th.Eickermann, W.Frings: Visit - A Visualization Interface Toolkit. Technical Information. ZAM, Forschungszentrum Juelich, 2000. Siehe auch http://www.fz-juelich.de/zam/visit
4. W.R.Stevens: Programmieren von UNIX-Netzwerken. Hanser Verlag, 2000.
5. S.Rathmayer: Visualization and Computational Steering in Heterogeneous Computing Environments. Euro-Par 2000, LNCS 1900, pp. 47-56, 2000.

The Ewald Summation Method: Calculating long-range interactions

Zoe Cournia

University of Athens, Greece
Department of Chemistry,
Laboratory of Physical Chemistry
zoe@mailbox.gr

Abstract: An implementation of the Ewald summation method for handling long-range interactions in classical particle simulations is presented. The accurate description of long-range electrostatic forces is necessary for a realistic simulation of the structure and dynamics of polar or charged chemical systems. The Ewald summation method is proven to provide controlable results for the electrostatic interactions in such systems, when treated under periodic boundary conditions. The algorithm has been implemented in a Fortran 90 module. It is a "stand-alone" module, getting all relevant information via parameter lists from the calling program and thus can be easily included into existing Molecular Dynamics or Monte Carlo programs. As an example it is added to the parallel program DMMD, developed at the Central Institute for Applied Mathematics (ZAM) at Forschungszentrum Juelich. The module has been tested for a system of a molten salt.

1 Introduction

The treatment of long-range interactions in molecular simulations is a longstanding problem. The correct and accurate calculation of long-range forces is of considerable importance for the reliability of modern computer simulations of polar/charged systems. Approximate methods [1], although simple and requiring little computational time, are proven to produce unrealistic results in certain physical systems, e.g. solvated macromolecules.

The Ewald summation technique is a well established and accepted method for accurate calculation of long-range interactions in periodic systems which consist of ions or molecules including partially charged interaction sites [2]. Optimized techniques have also been developed, such as the particle-mesh Ewald method and multipole-based Ewald summation methods [2] to perform fast and reliable simulations of large systems. The method, which was developed in 1921 by Ewald [3], was originally aimed to handle ionic crystals in solid state physics. Later on the method was successfully applied to liquids and biomolecular systems. Although the Ewald summation method is widely accepted, some drawbacks must be pointed out, which include additional computational effort and complexity in the implemention of the method, and also the possibility of artificial enhanced periodicity effects for finite size systems.

In this report a Fortran 90 implementation of the method is presented, which was worked out at the Central Institue for Applied Mathematics (ZAM) at the Research Centre

Juelich. The algorithm was coded in a stand-alone module, and thus, it can be easily included into different Molecular Dynamics or Monte Carlo simulation programs. As a demonstration the module was included and tested in the DMMD program, a modular molecular dynamics software developed at ZAM, which has been specifically designed for massively parallel architectures [4].

2 Theoretical Background

2.1 Interactions between particles

The total potential energy of a system can be separated into two terms: the one which takes into account short-range repulsion and dispersion forces and the other which accounts for the long-range electrostatic potential

$$U = U_1 + U_2 \tag{1}$$

To understand the spatial extent of the two terms of the potential one has to check the convergence criteria of the potential form. If we ignore screening effects, an upper limit for the potential energy of a system is given by

$$U = \int u(r)d\mathbf{r} = 4\pi \int_R^\infty r^2 u(r)dr \tag{2}$$

If the integral diverges the potential is called a long-range potential. For example we can consider the Coulomb potential:

$$u_{ij} = \frac{1}{4\pi\varepsilon_0}\frac{q_i q_j}{r_{ij}} \tag{3}$$

where q is the charge of the particle i or j respectively. The integral for the Coulomb potential then becomes $\int_R^\infty rdr$, i.e. it diverges. Thus, following this reasoning one can describe a long-range force as one in which the spatial interaction falls off no faster than r^{-d}, where d is the dimensionality of the system.

On the other hand, the integral of a short-range potential is upper-bounded, e.g. the Lennard-Jones potential:

$$u_{ij}^{LJ} = 4\epsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right) \tag{4}$$

which gives a result of integration for $U \approx \frac{c}{R^3}$, which is finite.

In the category of long-range interactions are charge-charge interactions between ions and dipole-dipole interactions between molecules.

For later use we mention here that screening effects might affect electrostatic interaction. A free charge causes opposite-charged ions to gather around it in a cloud-like surrounding. This causes the effect of screening which may result in an effective potential which can often be approximated with a Debye-like potential form:

$$u_{ij} = \frac{1}{4\pi\varepsilon_0} q_i q_j \frac{e^{-\kappa r_{ij}}}{r_{ij}} \tag{5}$$

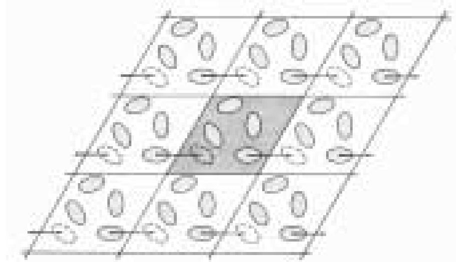


Figure 1. A two dimensional periodic system.

2.2 Handling long-range interactions

In MD simulations the long-range interactions are the most time consuming ones. Since, in principle, the range of these forces is infinite, it is definitely greater than half of the box length for small systems, e.g. a simulation consisting of 500 molecules. A first approximation could then be to simply increase the size L of the central box, and the potential drops close to 0 at the boundary of the box. Even by using very modern computer systems this solution is impracticable, since the time required to run such a simulation is proportional to N^2 i.e. the computational time is increased by a factor of 64 if one doubles the box length. [2].

Many methods have been proposed to solve this problem. The two most significant are lattice methods, such as the Ewald sum, which include the interaction of an ion or molecule with all its periodic images, and reaction field methods, which assume that the interaction from molecules beyond a cutoff distance can be handled by means of averages, using macroscopic electrostatics. In this report we concentrate on the Ewald summation method, which will be analyzed in the following section.

2.3 Periodic Boundary Conditions - Lattice Sums

Periodic boundary conditions are very often used in MD simulations as an approximation of an infinite system. They are based on the assumption that the correlation length between two particles is smaller than the box length. We consider a central cubic box in our simulation, which is replicated through space to form an infinite system. During the simulation it is assumed that as a molecule leaves the central box, its periodic image will enter the central box in exactly the same way. There are no walls at the boundary of the central box and no surface molecules. This box forms an axis of the system which we use to measure the coordinates of N molecules. A two dimensional version of such a periodic system, which illustrates the periodic boundary conditions is shown in Fig.1. Thus, the number density in the central box is conserved. Using this method, it is not necessary to store the coordinates of all the images in a simulation but only the particles of the central box. In this point arises the question of whether the properties of a small, infinitely periodic system and the macroscopic system which it represents, are the same. The accuracy of such an approximation depends on the range of the intermolecular potential and the phenomenon under investigation, but it is generally considered to be a very good approximation.

The Coulombic potential of such a periodic system can be written then as:

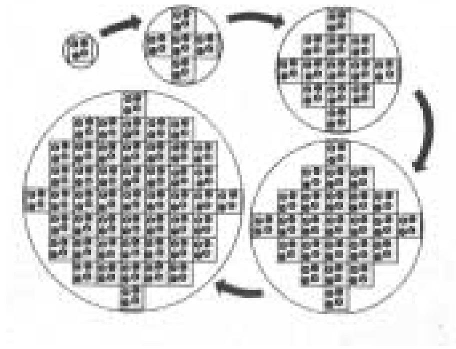


Figure 2. Building up the sphere of simulation boxes.

$$U = \frac{1}{4\pi\varepsilon_0}\frac{1}{2}\sum_{\mathbf{n}\neq 0}^{\prime}\sum_{i,j=1}^{N}\frac{q_i q_j}{r_{ij,\mathbf{n}}} \tag{6}$$

where the sum over $\mathbf{n}$ is the sum over all simple cubic lattice points, $\mathbf{n} = (n_x L, n_y L, n_z L)$, with n_x, n_y, n_z integers. This vector reflects the shape of the central box.

The problem is that the sum is conditionally convergent. Recall here that the result of a conditionally convergent sum is dependent on the order in which we add up the terms, and so Eq.6 cannot be handled as a convergent series. To overcome this problem a method developed by Ewald [3], called Ewald summation method, can be used.

2.4 Ewald summation

2.4.1 The Ewald Sum

The Ewald summation method is a technique for efficiently summing up the interaction between ions and all its periodic images. The principle of the Ewald summation method is that it introduces a Gaussian convergence factor e^{-sn^2} in the lattice sum by which the conditionally convergent series is split into two rapidly convergent sums plus a constant term.

$$U(s) = \frac{1}{2}\sum_{i,j=1}^{N}\sum_{\mathbf{n}}^{\prime} e^{-sn^2}\frac{q_i q_j}{\mid r_i - r_j + \mathbf{n}L \mid} \tag{7}$$

In the end of the calculation one has to perform the limit $s \to 0$, i.e. $U = lim_{s\to 0}U(s)$. The procedure consists of summing up the unit cell and its periodic images in spherical layers and built up an infinite system, as shown in Fig.2 . Following this approach, we must also specify the nature of the surrounding medium, that is its dielectric constant, ε. If we consider the surrounding material to be a conductor (i.e. tin foil), then the value of this surface term is zero. The equation that is finally obtained has the form:

$$U_{Ewald} = U^{real} + U^{recipr.} + U^0 + U^{surf.} \tag{8}$$

where:

$$U^{real} = \frac{1}{4\pi\varepsilon_0}\frac{1}{2}\sum_{i,j=1}^{N}\underbrace{\sum_{n=0}^{\infty}{}' q_i q_j \frac{\text{erfc}(\kappa|\mathbf{r}_{ij}+\mathbf{n}|)}{|\mathbf{r}_{ij}+\mathbf{n}|}}_{\text{real-space term}} \tag{9}$$

$$U^{recipr.} = \frac{1}{4\pi\varepsilon_0}\frac{1}{2}\sum_{i,j=1}^{N}\underbrace{(\frac{1}{\pi L^3})\sum_{\mathbf{k}\neq\mathbf{0}} q_i q_j \left(\frac{4\pi^2}{k^2}\right) e^{-k^2/4\kappa^2}\cos(\mathbf{k}\cdot\mathbf{r}_{ij})}_{\text{reciprocal-space term}} \tag{10}$$

$$U^0 + U^{surf.} = \underbrace{\left(\frac{\kappa}{\pi^{1/2}}\right)\sum_{i=1}^{N} q_i^2}_{\text{self term}} \quad + \quad \underbrace{\left(\frac{2\pi}{3L^3}\right)\left|\sum_{i=1}^{N} q_i\mathbf{r}_i\right|}_{\text{surface term}} \tag{11}$$

Here, $\mathbf{k}$ = (l,j,k) is a reciprocal-space vector, and $\mathbf{n}$ was defined earlier. Furthermore, the erfc(x) is the complementary error function $\left(\text{erfc}(x) = \frac{2}{\pi^{1/2}} \times \int_x^\infty e^{-t^2}dt\right)$, which is approximated as [10]

$$\text{erfc}(x) = \frac{1}{[1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4]^4} + \epsilon(x) \tag{12}$$

where a_i are constants and the error is $|\ \varepsilon(x)\ | \leq 5 \cdot 10^{-4}$. The error function decreases monotonously as x increases.

2.4.2 Physical explanation of splitting up the lattice sum

As discussed earlier, the Ewald summation method transforms the potential energy of Eq. (6) into a sum of two rapidly converging series in real and reciprocal space. A physical interpretation of this decomposition of the lattice sum is as follows. We consider a point charge which has a discontinuous potential. Then, around each point charge we introduce a charge distibution of equal magnitude and opposite sign, which spreads radially from the charge. This distribution is conventially taken to be Gaussian [2]:

$$\rho_i(\mathbf{r}) = q_i\kappa^3 exp(-\kappa^2 r^2)/\sqrt{\pi^3} \tag{13}$$

where the arbitrary parameter κ determines the width of the distribution, and $\mathbf{r}$ is the position relative to the center of the distribution. This charge distribution screens the interaction between neighbouring point-charges and limits it to short-range. To counteract this induced charge, a second Gaussian of the same sign and magnitude of the point charge is added in the reciprocal space. This is illustrated in Fig.3

This cancelling distribution is summed in reciprocal space using Fourier transforms. In other words, the Fourier tranform of the cancelling distribution is added and the total is transformed back into real space. The method also includes the interaction of the cancelling distribution centred at $\mathbf{r}$ and itself, and this so-called self-term must be also subtracted from the total.

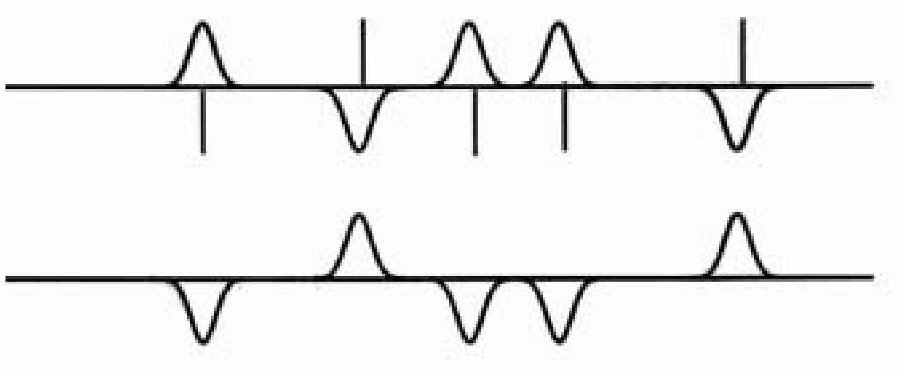


Figure 3. Charge distribution in the Ewald sum a. original point charges plus screening distribution. b. cancelling distribution.

2.4.3 Force Calculation

To calculate the force acting on a particle we can simply apply the ∇-operator to the Ewald sum potential (Eq. 8). The resulting equations are also a sum of real-space, $\mathbf{F}^{real}$, and reciprocal-space, $\mathbf{F}^{recipr.}$, contributions.

$$\mathbf{F}^{real} = q_i \sum_{j=1}^{N} \sum_{n} q_j \frac{(\mathbf{r}_{ij}, \mathbf{n})}{r_{ij,\mathbf{n}}^3} \left(\mathrm{erfc}(\kappa r_{ij,\mathbf{n}}) + \frac{2\kappa}{\sqrt{\pi}} r_{ij,\mathbf{n}} \exp(-(\kappa r_{ij,\mathbf{n}})^2) \right) \tag{14}$$

$$\mathbf{F}^{recipr.} = \frac{2q_i}{L} \sum_{j=1}^{N} \sum_{\mathbf{k} \neq \mathbf{0}} \frac{\mathbf{k}}{\mathbf{k}^2} \exp\left(-\left(\frac{\pi \mathbf{k}}{\kappa L} \right)^2 \right) \sin\left(\frac{2\pi}{L} \mathbf{k} \cdot \mathbf{r}_{ij} \right) \tag{15}$$

2.4.4 Choosing Ewald summation parameters

Three parameters control the convergence of the sums in Eq.(8): R_c, the cutoff radius of the system, k_{max}, an integer which defines the summation range in the reciprocal-space part and its number of k-vectors, and κ, the Ewald convergence parameter, which determines the rate of convergence of these series. In Eq.(9), a large value of κ, i.e. a narrow Gaussian distribution, makes the real-space sum to converge faster. On the other hand, in Eq.(10) a small value of κ causes then the reciprocal-space sum to converge very fast. So, in order to have the same accuracy in the real and reciprocal space parts, the parameters κ and k_{max} should be chosen in a proper way.

One can adjust the parameters by defining the accuracy of the calculation, e^{-p}. In doing so one gets [5] :

$$\kappa^2 r^2 = p \qquad \longrightarrow \qquad \kappa = \frac{\sqrt{p}}{R_c} \tag{16}$$

and

$$\frac{k^2}{4\kappa_{max}^2} = p \qquad \longrightarrow \qquad k_{max} = \frac{2p}{R_c} \tag{17}$$

2.4.5 Choosing the short-range potential

To account for the short range repulsion and dispersion forces one has to choose a proper potential form. The choice depends mainly on the system under investigation. For example the Lennard-Jones potential, Eq.(4) is applicable in many problems .

In the case of a molten salt like NaCl, a potential similar to a Buckingham potential is more realistic [6], due to the overestimated repulsion that appears in the LJ potential:

$$u_{ij}(r) = A_{ij} \exp[B(\sigma_i + \sigma_j - r)] + \frac{C_{ij}}{r^6} + \frac{D_{ij}}{r^8} \quad (18)$$

2.4.6 Neglecting the reciprocal-space term

It was suggested by Rycerz and Jacobs [7], that by properly choosing the simulation parameters the reciprocal space sum contribution to the total energy can be neglected entirely. Using the minimum image convention the authors estimated the contribution of reciprocal potential to the total potential to be about 1:1500. These observations were also made for molten NaCl, similar to our model. However, it was pointed out that this approach should not be applied to small systems *since in this case* the contribution of the reciprocal-space part becomes more significant, and the approximation results in artifacts. [8]

3 Programming Concept

3.1 Programming Language

The developed module is written in Fortran 90, which is a common language in scientific programming. Fortran 90, compared to older dialects of Fortran, includes modern language features such as the array language and abstract data types. The use of the modern language features like strong typing through interface checking, data and subroutine encapsulation in modules and dynamic memory allocation, is not generally applied by the contemporary compilers because of the backward compatibility with Fortran 77. The Fortran 90 standard was designed to exist alongside with older versions of Fortran, and in some cases the use of the new syntax is to a large degree optional. Thus, to avoid the use of old Fortran language features (which also include unsafe programming methods) and to diminish confusion, we adopted the restrictive F subset of Fortran 90 [9]. The F language has been proven to be highly regular, safe, reliable to use and with a high performance, especially for scientific applications.

3.2 The use of modules in Fortran 90

Fortran 90 is using the concept of modules to handle the flow of information between the subroutines [9]. The `module` provides a means of packing subprograms, global data, derived types and their associated operations, and interface blocks. Therefore, better organization and overview of the program is achieved, and everything associated with a task, may be collected into a module and accessed whenever it is needed. Parts that have to do only with internal working are declared as `private` and can only be used in the specific module. This allows the internal design to be altered without the need to change

the program that uses it and prevents accidental alteration of data. A module may also contain `use` statements that access parts or the whole of other modules. The code which was developed for calculating the long-range interactions of a system using the Ewald summation method, was written as a module and was easily included into the DMMD program.

3.3 Module Design

The structure of the module is designed to strictly follow the structure of the Ewald summation method. The module contains subroutines that are designed to calculate the real-space and the reciprocal-space parts of the Ewald sum for ions. Furthermore, the self-term is subtracted from the sum and the surface-term is zero due to the use of a conducting surrounding, as described earlier. The real function $\mathrm{erfc}(x)$ is introduced and returns the complementary error function as previously described. The arrays which depend on system specific parameters, e.g. system size or number of different particle species, are allocated in a separate subroutine.

Although the Fourier-space term is not actually used in the simulation process it is developed for reasons of completeness and also for later development of the program.

3.3.1 Organization of the module

The module is organized as follows. At the start of the module a `public` subroutine is declared which is called from the main program. This subroutine contains calls to `private` subroutines of the module, which are performing the calculations of the Ewald sum. To avoid any possible mistakes or omission of parameters the "parameter-list-check" subroutine is introduced. Thus, the call of a routine from the main program looks like:

1. Call from the main program:

```
call ewald( "real_term_list", ij_spec = ij_spec, n_pair = n_pair,  &
            list = pair_index, rij_sq = rijsq, fxyz_i = fxyz,      &
                                ................)
```

2. Call from the public subroutine `ewald`:

```
if( string == "real_term_list" ) then
if( .not. present( n_spec )  .or. .not. present( n_start )  .or.   &
                            ................. )then
call parameter_list_check( "ewald_module", "ewald", "real_term" )
endif
call real_term( rxyz, fxyz_i, fxyz_j, qq_spec, pot_spec, ij_spec,  &
                vir_spec, n_spec, n_start, n_end, r_cut_sq )
```

3. The proper subroutine is called and calculates the corresponding item.

3.3.2 Subroutines

Setup

Setup is called once in the beginning of the simulation to generate all the k-vectors required in the Ewald Sum. These vectors are used throughout the simulation in the calculation of the reciprocal-space term, to calculate the k-space contribution to the potential energy at each configuration. The wave vectors should fit the box of a unit lenght, i.e. $k = 2\pi n$.

Real-space term

The real-space term is calculated in a subroutine that returns the real-space contribution to the Ewald sum and it is called for each configuration in the simulation. Actually, two subroutines were developed to enable the possibility of using neighbor lists, too. Therefore if `(string == "real_term_list")`, then the program enables the use of a neighbor list. Otherwise if `(string == "real_term")` the program does not take into consideration the use of neighbor lists and performs calculation of the interparticle distances in every step.

Fourier-space term

Symmetry of the calculation has been used in order to reduce the computational effort. In one coordinate direction (x), the sum is symmetric and we include only positive k-vectors in the calculation [2]. Instead of calculating the quantity $\cos(\mathbf{k} \cdot \mathbf{r}_{ij})$, which is computationally very expensive, we rewrite it $\exp(i\mathbf{k} \cdot \mathbf{r}_{ij})$. Partial waves can easily be calculated then via

$$e^{inkr_{ij}} = (e^{ikr_{ij}})^n \tag{19}$$

Storing the value of $\exp(ikr_{ij})$ in a variable, avoids then to calculate n-times the exponential function. Nevertheless, the calculation of the Fourier-space part is most time-consuming in the module.

Parameter-list check

If an error occurs with the optional input parameters, the parameter-list-check subroutine prints out the location of the the problem and stops the program.

3.3.3 Principal Variables

To implement the Ewald sum the following principal variables were used:

kappa : real variable with value $\kappa = \frac{\sqrt{p}}{R_c}$
kappa represents the width of the cancelling distribution.

k_max : integer variable with value $k_{max} = \frac{2p}{R_c}$
k_max is the maximum number of allowed k-vectors required for the simulation. e^{-p} is the accuracy of the calculation, which in our case is 10^{-5}, and R_c is the cutoff radius. As mentioned in section 2.4.4, k_max and kappa are calculated according to the condition that the error in the real-space and reciprocal-space terms are comparable.

Kfactor : the maximum integer component of the k-vectors

rxyz : the coordinates of the particles
fxyz_i : force vector for index i
fxyz_j : force vector for index j
The force contribution is divided into fxyz_i and fxyz_j since it may be necessary to send i or j contributions separately to a PE.

qq_spec : the factor $\frac{q_i q_j}{4\pi\epsilon_0}$
pot_spec : the potential energy between interacting species i and j
for_spec : the total force between interacting species i and j
vir_spec : the total virial contribution

4 Implementation in DMMD

DMMD is a molecular dynamics program, developed at ZAM [4] which is structured with modules, and its design aims at massive parallel architectures. The structure of DMMD was designed to fulfil the requirements of consistent data parallelism, modular form, flexible data structure and choice for different types of algorithms. It adopts the message passing paradigm. Communication is handled in a special interface library which translates Fortran 90 concepts into more primitive mechanisms of current message passing protocols. Furthermore, it allows transparent switching between message passing libraries. The modular concept of DMMD allows easy modification and extensions. The DMMD program has been originally designed to handle only short-range interactions, so this module was used to extend the program to handle also long-range interactions.

The structure of the program remained unchanged and only minor changes were introduced, regarding additional parameters and calls to the module in the main program. Parallelization of the module was just limited to using the parallel interface of the DMMD program and to the f90_reduce command:

```
#ifdef PARALLEL
     call f90_reduce( f, f, f90_sum, 0, f90_comm_world, err )
#endif
```

5 Simulation Details and calculations

5.1 Simulation model

The computer simulation was performed by means of Molecular Dynamics, in the microcanonical ensemble (NVE), using a Verlet algorithm for the solution of the differential equations. A pure molten salt of 2048 particles was used (NaCl), at temperature $T = 1160K$, with a timestep of 0.008 ps. Equilibration time was 5.000 timesteps followed by 30.000 timesteps for statistical interpretation. The short-ranged potential that was used has the form of Eq.18.

5.2 Results

5.2.1 Potential forms

As described in section 2.4.5, a Buckingham-like potential was used to calculate the short-range part of the potential energy. In Fig.4 the potential of the interacting ions is shown, as calculated in the simulation, and is compared with the Lennard-Jones potential. It is found that a LJ potential is unrealistic for our system, due to the overestimated repulsion and the appearance of a local minimum.

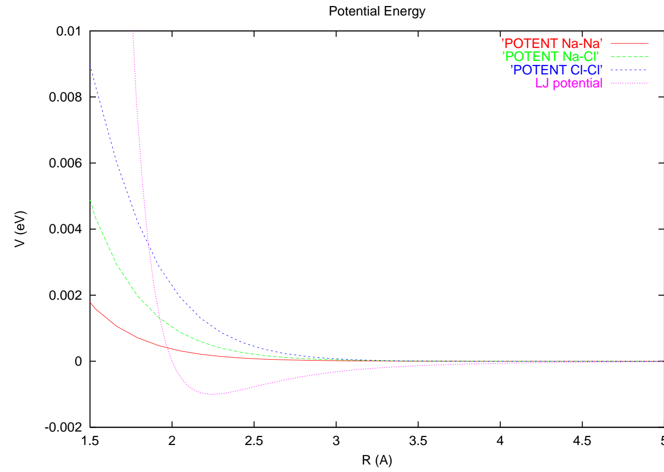


Figure 4. Potential energies of the molten salt NaCl compared with a LJ potential.

5.2.2 Radial distribution functions

The mean molecular structure of a molecular system can be studied by examining radial distribution functions, $g(r)$, of the system. Radial distribution functions are a measure for the mean number of neighboring molecules surrounding a tagged particle. These functions play an important role in the study of molecular systems since with their knowledge one can calculate various thermodynamic properties of the equilibrated system. Radial distribution functions for the molten salt NaCl are shown in Fig.5. The first peak in the diagram of a radial distribution function represents the first coordination shell, the second peak shows the second neighbor shell and so on, i.e. the local extrema, appearing in $g(r)$ are a measure for deviations of the local density, $\rho(r)$, from its macroscopic average, ρ_0. On large length scales a particle should *see* the average density in the liquid and so $\lim_{r\to\infty} g(r) = \rho(r)/\rho_0 = 1$. Fig.5 shows a typical radial distribution of a molten salt, that is a liquid-like radial distribution function.

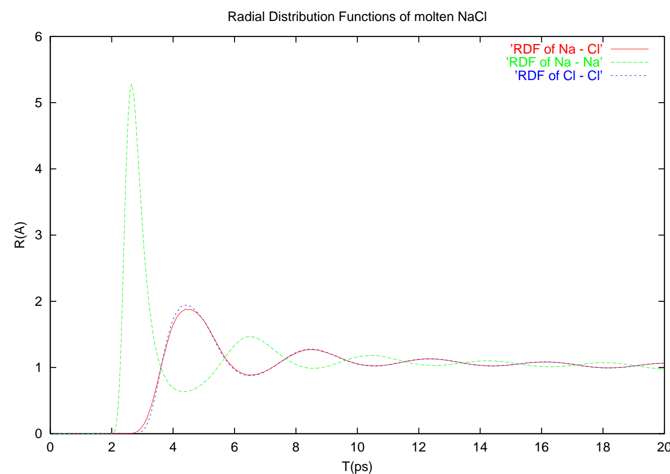


Figure 5. Radial distribution functions for the molten salt NaCl.

6 Conclusions - Prospects

The treatment of electrostatic interactions in simulations of simple and complex systems, like large biomolecules, is still an area of ongoing research, because this treatment often involves many approximations introduced in the simulation procedure. The Ewald summation technique which was implemented in a new Fortran 90 module described in the text, is a method applicable to polar / ionic systems and has shown to produce realistic results. The module is independent and can be implemented into different simulation programs.

A required improvement of the module is a better implementation of the reciprocal-space term, in order to speed up the calculation. It is also planned to characterize the potential by implementing a small analysis program that calculates the equi-potential lines. Furthermore an interesting feature would be to see the potential energy distribution of each k-component of the k-vectors. Due to the lack of time not many physical results could be presented here - this is aimed for the near future.

7 Acknowledgements

This work was carried out during the Gueststudents' Program for Scientific computing 2000, organised by ZAM and NIC. I would like to thank the organisers Prof. Hossfeld and Dr. R. Esser for enabling me to participate in the program and all staff of ZAM for excellent working conditions and for their help. I am grateful to my supervisor Dr. G. Sutmann who largely contributed in carrying out this project and helped me with useful discussions, comments and critically reading the manuscript.

References

1. M.P.Allen, D.J.Tildesley, *Computer Simulation of Liquids*
2. A.Y.Toukmaji, J.A.Board, Comput.Phys.Commun. 95 (1996), 72
3. P.Ewald, Ann.Phys.64 (1921), 253
4. N. Attig et al., *Molecular dynamics algorithms for massively parallel computers* in R.Esser et al. (*eds.*),*Molecular dynamics on parallel computers*, p. 46, World Scientific 2000
5. D. Fincham, Molec. Sim. 13 (1994), 1
6. J.W.Lewis and K.Singer, J.Chem.Soc.Faraday Trans. II 71 (1975), 41
7. Rycerz and Jacobs, Molec. Sim. 8 (1992), 197
8. Implementation of the Fourier-space part of the Ewald sum is not yet finished. For unknown reasons, when using the current version of the reciprocal space sum, the temperature and energy are increased during the simulation. Work is underway to remove this artifact of the procedure.
9. M.Metcalf, J.Reif, *The F programming language*
10. M.Abramowitz, I.Stegun, *Handbook of mathematical functions*

Simulationen ungeordneter dipolarer Systeme mit der Methode des "Parallel Tempering"

Timm Höhr

Bureschweg 4, 49635 Badbergen

thoehr@gmx.de

Zusammenfassung: Die Methode des "Parallel Tempering" wird angewendet auf ein System ungeordneter, langreichweitig wechselwirkender dipolarer Zentren. Dieses dient als Modell für ungeordnete Systeme mit lokal beweglichen, geladenen Defekten, bei denen experimentell oft ein frequenzunabhängiger dielektrischer Verlust auftritt. Hier wird jedoch die Frage untersucht, ob der Austausch von Systemzuständen zwischen mehreren, bei verschiedenen Temperaturen stattfindenden, parallelen Simulationen eine effiziente Equilibrierung ermöglicht.

1 Einleitung

Bei einer Vielzahl von Simulationen physikalischer Systeme interessiert man sich für Gleichgewichtseigenschaften. Man steht zunächst vor dem Problem, das System ins Gleichgewicht zu bringen, bzw. zu thermalisieren. Für die eigentliche Simulation, d.h. während der Aufnahme von Daten, benötigt man Zustände, die mit einer für das Gleichgewicht charakteristischen Häufigkeit auftreten. Die zugrundeliegende Wahrscheinlichkeitsverteilung kennt man jedoch in der Regel nicht.

Die einfachste und sehr häufig verwendete Methode besteht darin, einen thermalisierten Zustand durch eine vorgeschaltete Simulationsphase zu erreichen zu suchen, indem man einfach abwartet, bis – wie im Falle eines kanonischen Ensembles – die Systemenergie eines beliebig konfigurierten Anfangszustands auf einen bis auf Fluktuationen konstanten Wert abgesunken ist.

Insbesondere bei tiefen Temperaturen gestaltet sich diese Art der Thermalisierung schwierig, da das System aufgrund der niedrigen thermischen Anregungsenergien in lokalen Energieminima gefangen werden kann. Das Auffinden des Gleichgewichts dauert unter solchen Bedingungen sehr lange oder ist innerhalb einer realistischen Simulationszeit generell in Frage gestellt.

Daher ist man an schnellen und effektiven Methoden zum Erreichen eines thermalisierten Zustands interessiert.

Wir wollen hier für das Beispielsystem eines dipolaren Coulomb Gittergases eine parallele Methode untersuchen, in der gleiche Systeme parallel bei verschiedenen Temperaturen simuliert werden. Dabei soll ein Austausch von Teilchenkonfigurationen zwischen diesen parallelen Prozessen verhindern, daß diese bei tiefen Temperaturen in metastabilen Zuständen gefangen bleiben.

2 Die Methode des "Parallel Tempering"

Im folgenden wird in Anlehnung an Hukushima und Nemoto ein Verfahren beschrieben, das diese 1996 im Zusammenhang mit Monte Carlo Simulationen an Spingläsern vorgeschlagen haben [1]. Von ihnen wurde die Methode ursprünglich "Exchange Monte Carlo Method" genannt. Es wurde jedoch auch bei Molekulardynamiksimulationen erfolgreich angewendet [2].

Man betrachtet hierzu eine Anzahl M nicht wechselwirkener Replika $m = 0, .., M-1$ des gleichen Systems. Jedes System m ist in Kontakt mit einem Wärmebad der Temperatur T_m, bzw. ihrem Inversen $\beta_m = 1/k_B T_m$. Diese Temperaturen sind paarweise verschieden, es gelte $\beta_m < \beta_{m+1}$. Zusammen bilden die M Systemzustände einen Gesamtzustand $\{\mathbf{X}\} = \{\mathbf{X}_0, \cdots, \mathbf{X}_{M-1}\}$.

Die Zustandssumme $Z(\{\mathbf{X}\})$ des Gesamtsystems ist gegeben durch:

$$Z = \text{Spur } \exp\left(-\sum_{m=0}^{M-1} \beta_m H(\mathbf{X}_m)\right) = \prod_{m=0}^{M-1} Z_m \ . \tag{1}$$

Dabei bezeichnet $H(\mathbf{X})$ den allen Systemen gemeinsamen Hamiltonoperator.

Die Wahrscheinlichkeit für den Gesamtzustand $\{\mathbf{X}\} = \{\mathbf{X}_0, \cdots, \mathbf{X}_{M-1}\}$ bei $\{\beta\} = \{\beta_0, \cdots, \beta_{M-1}\}$ erhält man damit als:

$$P(\{\mathbf{X}, \beta\}) = \prod_{m=0}^{M-1} P_{\text{eq}}(\mathbf{X}_m, \beta_m) \ . \tag{2}$$

Sie ist einfach das Produkt der Einzelwahrscheinlichkeiten $P_{\text{eq}}(\mathbf{X}, \beta)$, ein System bei der inversen Temperatur β im Zustand $\mathbf{X}$ anzutreffen.

Man konstruiert nun einen Markov-Prozeß für den Austausch von Zuständen $\mathbf{X}$ und $\mathbf{X}'$. Hierzu definiert man Übergangswahrscheinlichkeiten $W(\mathbf{X}, \beta_m | \mathbf{X}', \beta_n)$ unter der Voraussetzung, daß vor diesem Austausch sich das System m bei der Temperatur β_m im Zustand $\mathbf{X}$ befand und ensprechend System n bei der Temperatur β_n im Zustand $\mathbf{X}'$. Wenn das Gesamtsystem im Gleichgewicht bleiben soll, hat man die Bedingung des detaillierten Gleichgewichts einzuhalten:

$$\begin{aligned} &P(\cdots; \mathbf{X}, \beta_m; \cdots; \mathbf{X}', \beta_n; \cdots)\ W(\mathbf{X}, \beta_m | \mathbf{X}', \beta_n) = \\ &P(\cdots; \mathbf{X}', \beta_m; \cdots; \mathbf{X}, \beta_n; \cdots)\ W(\mathbf{X}', \beta_m | \mathbf{X}, \beta_n) \ . \end{aligned} \tag{3}$$

Mit Gleichung (2) erhält man:

$$\frac{W(\mathbf{X}, \beta_m | \mathbf{X}', \beta_n)}{W(\mathbf{X}', \beta_m | \mathbf{X}, \beta_n)} = \exp(-\Delta) \tag{4}$$

mit

$$\Delta = (\beta_n - \beta_m)(E(\mathbf{X}) - E(\mathbf{X}')) \tag{5}$$

Dabei bezeichnet $E(\mathbf{X})$ die Energie des Systems im Zustand $\mathbf{X}$. Eine übliche Wahl von W ist nach Metropolis [3]:

$$W(\mathbf{X}, \beta_m | \mathbf{X}', \beta_n) = \begin{cases} 1 & \text{für } \Delta < 0 \\ \exp(-\Delta) & \text{für } \Delta \geq 0 \ . \end{cases} \tag{6}$$

Dies bedeutet, daß ein Austauschversuch immer dann angenommen wird, wenn die Energie des Systems bei der höheren Temperatur geringer ist als die des kälteren Systems. Andernfalls erfolgt ein Austausch nur mit der Wahrscheinlichkeit $\exp(-\Delta)$.

2.1 Das Simulationsverfahren

Im normalen seriellen Verfahren werden Zustandsfolgen $\mathbf{X}(t)$ generiert. Diese entsprechen z.B. im Falle von Molekulardynamiksimulationen den Koordinaten und Geschwindigkeiten der Teilchen zur diskretisierten Zeit t oder im Fall von Monte Carlo Simulationen dem Zustand des Systems nach t Sprungversuchen. Der kanonische thermische Mittelwert einer Variablen A zur Temperatur β_m ist dann einfach: $\langle A \rangle_{\beta_m} = \frac{1}{N} \sum_{i=1}^{N} A(\mathbf{X}_m(i \cdot \Delta t))$ (mit dem Zeitschritt Δt).

Beim "Parallel Tempering" Verfahren belegt man zunächst jeden von M Prozessoren eines Parallelrechners mit einem Anfangszustand $\mathbf{X}_m(t=0)$ des Systems, $m = 0, \cdots, M-1$. Jedem System wird eine feste inverse Temperatur β_m zugeordnet.

Iterativ führt man nun folgende Schritte aus:

1. Auf jedem Prozessor bzw. für jedes System wird dieselbe Anzahl von Schritten einer standardmäßigen Simulation ausgeführt.
2. Danach werden Austauschversuche von Systemzuständen zwischen den Prozessoren mit der Austauschwahrscheinlichkeit W (siehe Gl. (6)) unternommen.

Damit der oben beschriebene Algorithmus effektiv arbeitet und eine rasche Thermalisierung bewirkt, müssen folgende Bedingungen erfüllt sein:

1. Die Austauschwahrscheinlichkeit darf nicht zu klein sein. In der praktischen Ausführung bedeutet dies, daß man einen Austausch nur zwischen benachbarten Temperaturen erlaubt.
2. Ein Systemzustand soll sich hinreichend schnell auf der gesamten Temperaturleiter hin- und herbewegen.
3. Durch diese Bewegung soll das System die Erinnerung an frühere Zustände verlieren. Insbesondere darf die höchste Temperatur nicht zu niedrig gewählt sein.

Anstatt Zustände auszutauschen, kann man jedes System auf einem festen Prozessor belassen und stattdessen Temperaturen vertauschen. Diese Betrachtung ist äquivalent zur bisherigen Sichtweise. Man erhält in diesem Temperaturaustauschschema den kanonischen Mittelwert als $\langle A \rangle_{\beta} = \frac{1}{N} \sum_{i=1}^{N} \sum_{m=0}^{M-1} A(\mathbf{X}_m(i \cdot \Delta t))\, \delta_{\beta, \beta_m(i \cdot \Delta t)}$.

3 Modell-System

Das hier untersuchte Modell System ist das dipolare Coulomb Gittergas ("dipolar Coulomb lattice gas", DCLG), das von Pendzig entwickelt wurde [4].

Es ist von grundlegendem Interesse als einfaches Modell für ungeordnete Substanzen, die eine lokale Bewegung geladener Defekte aufweisen. Die wesentlichen Aspekte des Modells sind eine positionelle Unordnung in Verbindung mit langreichweitiger Wechselwirkung.

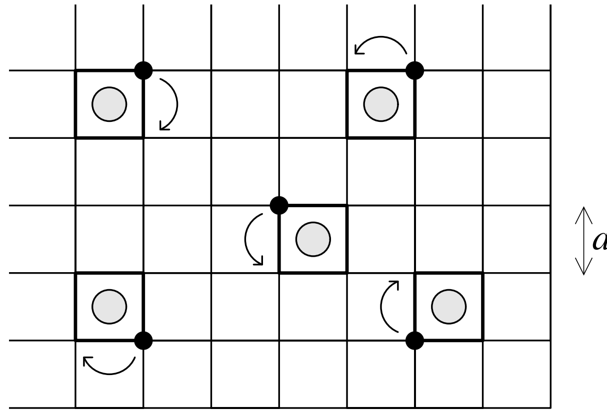


Abbildung 1. Zweidimensionale Ansicht eines Auschnitts des DCLG. Große graue Kreise stellen die unbeweglichen Gegenionen dar, kleine schwarze Kreise die mobilen Ionen.

Anhand des Modells wird die Fragestellung untersucht, unter welchen Bedingungen diese einfachen Grundannahmen zu einem frequenzunabhängigen, dielektrischen Verlust führen können [4–6]. Hierzu wurden Monte Carlo Simulationen durchgeführt (zu näheren Ausführungen und früheren Ergebnisse, siehe Anhang). Dieser Bericht beschränkt sich im weiteren jedoch auf die Thermalisierungseigenschaften des Modells.

3.1 Das dipolare Coulomb Gittergas

Das System besteht aus einer kubischen Box der Kantenlänge La, die durch ein einfach kubisches Gitter der Periode a in L^3 würfelförmige Zellen eingeteilt ist. Es gibt zwei Sorten entgegengesetzt geladener Teilchen bzw. Ionen in dieser Box.

Die sogenannten Gegenionen besetzen die Mittelpunkte zufällig ausgewählter Zellen und sind unbeweglich. Jedem Gegenion ist ein mobiles Ion zugeordnet, das sich auf den acht Eckplätzen der Zelle bewegen kann (vgl. Abb. 1). Es kann den das Gegenion umgebenden Würfel jedoch nicht verlassen. Insgesamt enthält die Box N solcher dipolarer Zentren. Daraus ergibt sich die Ionenpaarkonzentration $c = N/L^3$. Positionelle Unordnung wird durch eine zufällige Verteilung der Gegenionen erreicht. Dabei wird darauf geachtet, daß sich die besetzten Zellen in keinem Punkt berühren, damit keine zwei mobilen Ionen Zugang zu demselben Gitterplatz haben. Somit beträgt der Mindestabstand zweier Zentren $2a$.

Für das System gelten periodische Randbedingungen, d.h. , an die Systembox schließen sich in alle drei Raumrichtungen unendlich viele, gleichartige Boxen an.

Das DCLG ist eine Spezialisierung des Gegenionenmodells von Knödler auf den Hochfrequenz- bzw. Niedertemperaturfall [5]. Das vollständige Gegenionenmodell läßt im Gegensatz zum DCLG auch die Beschreibung langreichweitiger Diffusion bzw. Transports in niederfrequenten Wechselfeldern zu.

Die Temperatur ist im Verhältnis zu einer typischen Wechselwirkungsenergie anzugeben. Eine mögliche Wahl hierfür ist die Coulombenergie $E_c = q^2/(4\pi\varepsilon_0 a)$ zweier im Abstand a befindlicher Ionen [4]. In den Temperaturparameter geht zusätzlich die Konzentration c ein: $\Theta = \frac{k_B T}{E_c\, c}$.

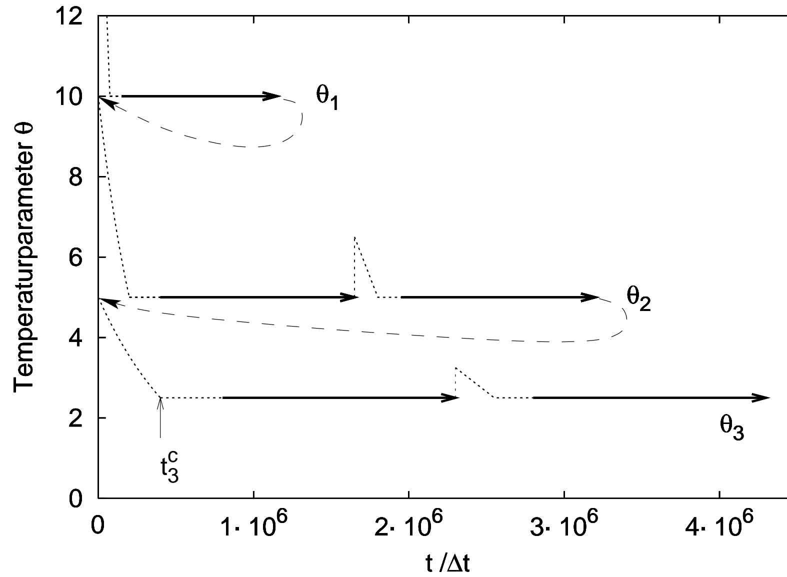


Abbildung 2. Ausschnitt aus dem Temperaturverlauf mit der Simulationszeit für Typ-II-Simulationen. Die Startzeit der Abkühlphasen ist in dieser Darstellung für jede Temperaturstufe auf 0 gesetzt. Die starken durchgezogenen Linien geben die Datenaufnahmephasen an. Folgen zwei Datenaufnahmephasen bei derselben Temperatur aufeinander, so wird das System dazwischen aufgeheizt. Unmittelbar danach werden keine Daten aufgenommen. Direkt auf die letzte Datenaufnahmephase folgt die nächste Abkühlphase. Die gestrichelten Pfeile kennzeichnen keinen Simulationszeitraum, sondern dienen der Verdeutlichung der Abfolge.

Zur Simulation der Hoppingereignisse der mobilen Teilchen wird der Standard-Metropolis-Algorithmus angewendet [3]. Die Energie des Systems wird mit der Ewaldsummation berechnet [7, 8].

3.2 Thermalisierung bei seriellen Simulationen

Bei früheren seriellen Simulationen wurden, ausgehend von zufällig verteilten Startplätzen der mobilen Ionen, zwei verschiedene Arten der Thermalisierung angewendet [6].

In einem Fall fand eine Thermalisierungsphase bereits bei der Zieltemperatur statt, bei der auch die nachfolgende Datenaufnahme durchgeführt wird. Dies entspricht einem instantanen Quench auf die Zieltemperatur.

Im anderen Fall wurden für dieselbe Unordnungskonfiguration mehrere Simulationen bei n schrittweise abgesenkten Temperaturen Θ_i $(i = 0, .., n-1)$ hintereinander ausgeführt. Zwischen Θ_i und Θ_{i+1} befindet sich eine Abkühlphase, in der die Temperatur in kleinen Schritten auf den neuen Temperaturwert Θ_{i+1} eingestellt wird (vgl. Abb. 2). Eine solche Temperaturrampe mehrerer Θ_i sollte bei einer Temperatur Θ_0 starten, die so hoch ist, daß das System problemlos thermalisiert.

Zur Unterscheidung sollen die Simulationen je nach Thermalisierungsart im folgenden als Simulationen vom Typ I (Quench) bzw. Typ II (Temperaturrampe) bezeichnet werden.

Beim Vergleich der Energien, die sich bei Verwendung der verschiedenenen Thermalisierungsarten einstellten, zeigten sich für die Konzentration $c = 1/64$ keine nennenswerten Unterschiede, wohl aber für wesentlich stärker verdünnte Systeme der Konzentration $c = 10^{-3}$.

Abbildung 3 zeigt beispielhaft für ein solches, stark verdünntes System bei dem Temperaturparameter $\Theta = 0.8$, daß der Simulationstyp II durch die Hintereinanderschaltung mehrerer Temperaturstufen in Verbindung mit dazwischenliegenden Abkühlphasen eine deutliche Absenkung der Energie gegenüber dem Verfahren I liefert. Der energetische Abstand beträgt ein Mehrfaches der Standardabweichung. Die Simulation des Typs I endete in dem dargestellten Fall sogar oberhalb einer Energie, die mittels des Verfahrens II für $\Theta = 1.25$ gefunden wurde (vgl. Abbildung 3).

Bei Temperaturen $\Theta \geq 5$ zeigten sich keine nennenswerten energetischen Unterschiede zwischen den beiden Verfahren. Für $0.8 < \Theta < 5$ liegen keine Daten des Typs I zum Vergleich vor.

Bei der Auswertung individueller Akzeptanzraten einzelner mobiler Teilchen fiel zudem auf, daß manche dieser Teilchen niemals springen. Ihr Anteil steigt mit sinkender Temperatur an auf 2.7% ($\Theta = 2.5$), ca. 5% ($\Theta = 1.25$) und ca. 10% ($\Theta = 0.8$). Es ist zu beachten, daß diese Werte natürlich von der Simulationsdauer abhängen.

Zusammenfassend läßt sich festhalten: Es konnte an einem Beispiel gezeigt werden, daß mit dem Verfahren I ein stark verdünntes System ($c = 10^{-3}$) nur unzureichend thermalisiert. Beim Verfahren II gelangt man zwar zu tieferen Energien, jedoch bleibt eine nicht unerhebliche Zahl der Zentren unbeweglich. Es ist unklar, ob deren Verharren nur eine metastabile Erscheinung ist, die sich aufgrund der begrenzten Simulationsdauer nicht als solche zeigt, oder ob sie ein wirkliches Charakteristikum des Tieftemperaturverhaltens der jeweiligen Unordnungskonfiguration darstellt.

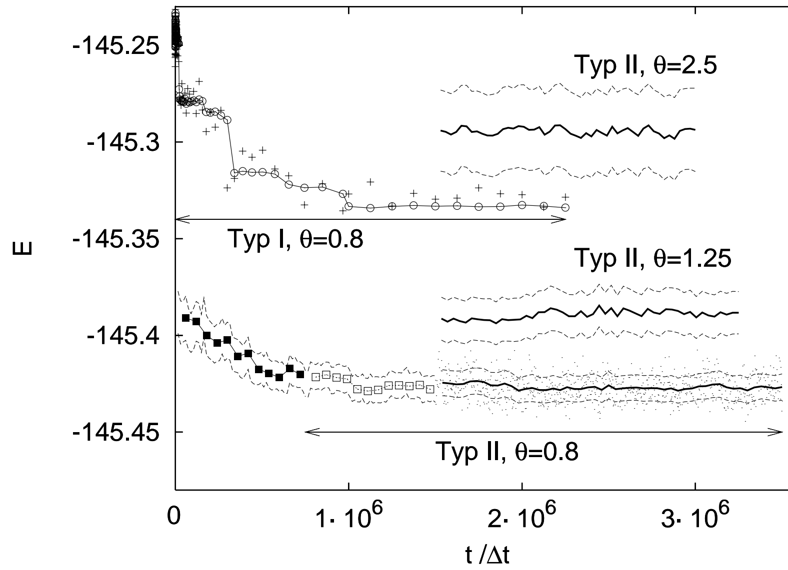


Abbildung 3. Vergleich der Simulationsverfahren I und II am Beispiel eines Systems mit $c = 10^{-3}$, $\Theta = 0.8$ und $N = 125$ Teilchenpaaren. Aufgetragen ist die Energie E des Systems als Funktion der Simulationszeit. Starke durchgezogene Linien geben den gleitenden Energiemittelwert an (Datenaufnahmephase Typ II). Die Punktwolke besteht aus herausgegriffenen Werten von E bei der tiefsten Temperatur $\Theta = 0.8$. Die gestrichelten Linien geben den Abstand einer Standardabweichung an. Darüber sind noch Mittelwert und Breite der Verteilung für die zwei vorangegangenen Temperaturstufen dargestellt. Kreise ($\circ$) geben den Mittelwert für das Verfahren I an, erhalten ebenfalls für $\Theta = 0.8$. Kreuze ($+$) kennzeichnen herausgegriffene Werte. Die Pfeile geben in beiden Situationen den Bereich an, in dem $\Theta = 0.8$ konstant ist. Zur Illustration des Verfahrens II sind für $\Theta = 0.8$ zusätzlich die Abkühlphase ($\blacksquare$) und die Thermalisierungsphase ($\square$) dargestellt.

Aufgrund dieser Fragen sollen die betreffenden Systeme nun mit dem ”Parallel Tempering” Verfahren untersucht werden.

4 Anwendung des ”Parallel Tempering” Verfahrens

In diesem Abschnitt wird die konkrete Umsetzung der ”Parallel Tempering” Methode auf das Dipolare Coulomb Gittergas beschrieben.

Auf jedem Prozessor wird dieselbe Unordnungskonfiguration der Gegenionen erzeugt, andernfalls würde ein Vergleich der Energien oder ein Austausch der Koordinaten der mobilen Ionen keinen Sinn ergeben.

Es finden nur Austauschversuche innerhalb von Paaren benachbarter Temperaturen statt. Nach einer Folge von R Monte Carlo Schritten (MCS) werden abwechselnd entweder für die geradzahligen oder die ungeradzahligen Paare Austauschversuche unternommen.

4.1 Konfigurations- versus Temperaturaustausch

Es gibt zwei Möglichkeiten der Umsetzung. Entweder vertauscht man die Anordnungen der mobilen Ionen zweier Systeme bei benachbarten Temperaturen oder man vertauscht diese beiden Temperaturen.

Die erste Variante ist die einfachste. Zur Ermittelung der Trajektorie einer Meßgröße bei einer festen Temperatur, wie z.B. der Energie und ihres Mittelwertes, braucht man sich um keine weitere Kommunikation zu kümmern. Es ist stets derselbe Prozessor zuständig. Möchte man jedoch die Bewegung eines Systems durch den Temperaturbereich verfolgen, so erfordert dies entweder bei jedem Austausch eine Rückmeldung an einen „Heimatprozessor“ (der die Informationen über den aktuellen Aufenthaltsort sammelt) oder man muß die erzeugten Daten post simulationem entsprechend aufbereiten. Ein entscheidender Nachteil dieser Variante ist der mit der Systemgröße linear wachsende Kommunikationsaufwand.

Der zweite Weg erfordert weniger Kommunikation. Was zuvor für die Trajektorie der Systemkonfigurationen im Temperaturraum galt, gilt nun für die Trajektorie einer Meßgröße bei einer festen Temperatur. Zusätzlich muß man dafür Sorge tragen, daß jeder Prozessor stets seinen Vorgänger und Nachfolger auf der Temperaturleiter kennt. Diese Buchhaltung kann man einem Masterprozessor übertragen, der die Austauschversuche koordiniert. Stattdessen wurde jedoch ein Verfahren ohne Master gewählt. Hierzu werden die Austauschversuche in zwei Hälften nacheinander abgearbeitet, damit die Zuordnung der Nachbarprozessoren nicht durcheinander gerät. Das Vorgehen ist schematisch in Abb. 4 gezeigt.

4.2 Technische Details

Das Simulationsprogramm entstand aus einer grundlegenden Umstrukturierung der wesentlichen Bestandteile eines bereits existierenden seriellen Codes. Als neuer Aspekt kam die parallele Gestaltung und Einbeziehung von Kommunikationsroutinen hinzu. Hierzu wurde das ”Message Passing Interface” (MPI) verwendet. Als Programmiersprache diente C++. Der Programmcode wurde auf zwei unterschiedlichen Systemen compiliert und zur Ausführung gebracht. Hierzu standen am Zentralinstitut für angewandte Mathematik

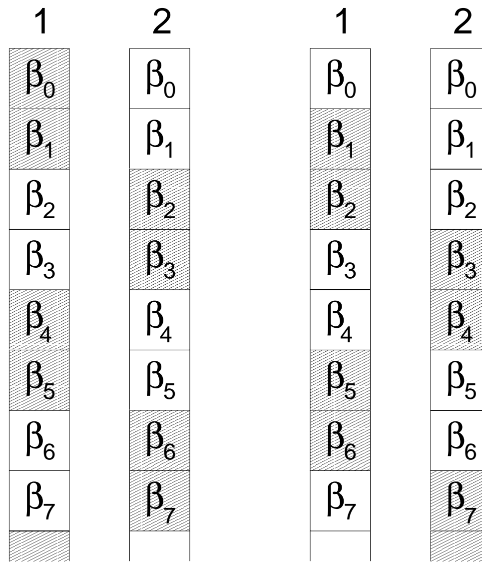


Abbildung 4. Darstellung der Austauschverfahren für geradzahlige (links) und ungeradzahlige Paare (rechts). Beide erfolgen jeweils in zwei Schritten. Schraffiert unterlegt sind die Temperaturen die an einem Austauschversuch teilnehmen. Zur einfacheren Darstellung sind die Prozessoren sind hier nach ansteigendem β geordnet.

(ZAM) des Forschungszentrums Jülich zum einen eine CRAY T3E-1200 (massiv paralleler Computer) und der SMP-Cluster ZAMpano, bestehend aus acht Vierprozessorknoten (Intel Xeon), zur Verfügung.

5 Testsimulationen und Ergebnisse

5.1 Austauschrate und geeignete Temperaturfolgen

Ein wichtiger Inputparameter ist die Menge der inversen Temperaturen $\{\beta\} = \{\beta_0, \cdots, \beta_{M-1}\}$. Die Bedingung einer nicht zu kleinen Austauschrate erfordert einen nicht zu großen Abstand zweier benachbarter Temperaturen. Wenn man einen größeren Temperaturbereich überdecken möchte, insbesondere von einer hohen Temperatur zu einer interessierenden tiefen Temperatur, so muß man M hinreichend groß wählen.

In der Regel ist durch die vorhandenen Ressourcen der Spielraum für die Anzahl M der Prozessoren beschränkt. Daher soll im weiteren von einem fest vorgegebenen M ausgegangen werden, zu dem man eine geeignete Temperaturfolge finden möchte.

Dabei darf die höchste Temperatur nicht zu niedrig gewählt werden (bzw. β_0 nicht zu groß), damit das System Gelegenheit hat, lokalen Energieminima zu entkommen.

Zur Anpassung der Temperaturfolge $\{\beta_0, \cdots, \beta_{M-1}\}$ bietet sich die Betrachtung der Austauschraten W_m für die einzelnen Temperaturpaare (β_m, β_{m+1}) an. Dazu führt man eine Testsimulation mit einer Startfolge durch. Im Anschluß daran kann man die β_m entweder von Hand anpassen oder automatisierte Verfahren anwenden.

Hukushima und Nemoto beschreiben ein Iterationsverfahren zum Auffinden geeigneter Temperaturfolgen, das für unser System jedoch keine Konvergenz erkennen ließ und bei dem zudem die tiefste Temperatur nicht vorgebbar war [1]. Abgewandelte Verfahren für

ein fest vorgebbares Temperaturintervall konnten die starken Unterschiede in den Austauschraten auch nicht beseitigen. Daher wurden nur manuell erzeugte β_m verwendet, die durch Anpassung eines Polynoms dritten Grades geglättet wurden.

Abb. 5 zeigt die Austauschraten W_m für verschiedene Temperaturfolgen. Insbesondere sieht man, daß eine äquidistante Verteilung der inversen Temperaturen keine einheitlichen Austauschraten liefert.

Ein großer Wert von M verzögert jedoch die Bewegung der Zustände auf der Temperaturleiter, da die Sprungweite in der Temperatur geringer ist. Hier ist zu prüfen, ob dies durch die größeren Austauschraten kompensiert wird. Verglichen wurden dazu die Austauschraten von Testsimulationen mit $M = 16$ bzw. $M = 32$ Prozessoren. In diesem Fall sind die Austauschraten bei doppelter Prozessoranzahl tatsächlich etwa doppelt so groß (vgl. Abb. 6).

5.2 Systemenergie

Interessant ist die Frage, welche Energien die parallele Methode im Vergleich zum bisherigen seriellen Verfahren liefert. An einer Unordnungskonfiguration wurde überprüft, daß diese zumindest nicht höher sind als in einer früheren seriellen Simulation (siehe Abb. 7).

Die "Parallel Tempering" Simulationen liefen dabei nur über eine Gesamtdauer von 250.000 MCS, inklusive eines Thermalisierungsvorlaufs von 50.000 MCS mit $R = 10$, der zur Erstellung des Histogramms nicht benutzt wurde. Bei der seriellen Simulation betrug die Vorlaufzeit insgesamt mehr als 3 Mio. MCS, verteilt auf 8 Temperaturstufen[1]. Damit ist die gesamte parallele Rechenzeit fast um einen Faktor 4 geringer[2].

Außerdem hat man Konfigurationen der mobilen Teilchen bei 16 verschiedenen Temperaturen gewonnen, gegenüber 8 beim seriellen Verfahren.

---

[1] Es bleibt allerdings zu prüfen, ob kürzere Vorlaufzeit bei der seriellen Simulation schon ähnliche Ergebnisse liefern.

[2] Der Kommunikationsaufwand ist in diesem Vergleich nicht berücksichtigt. Er ist vermutlich gering, da nur nach jeweils R MC-Schritten kleine Datenmengen zwischen Prozessorenpaaren ausgetauscht werden.

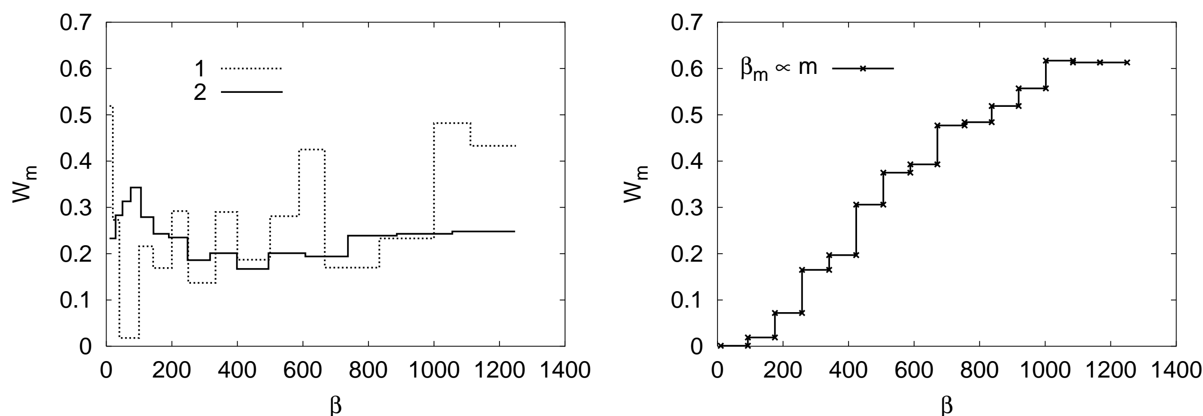


Abbildung 5. Autauschraten W_m bei $R = 10$ für verschiedene Temperaturfolgen, aufgetragen gegen β. Die Positionen der β_m entsprechen den Stufenkanten, W_m ist gegeben durch die Höhe der Stufe zwischen β_m und β_{m+1}. Links: Austauschraten vor (1) und nach einer manuellen Anpassung (2). Rechts: Austauschraten für dasselbe System für äquidistante $\beta_m \propto m$.

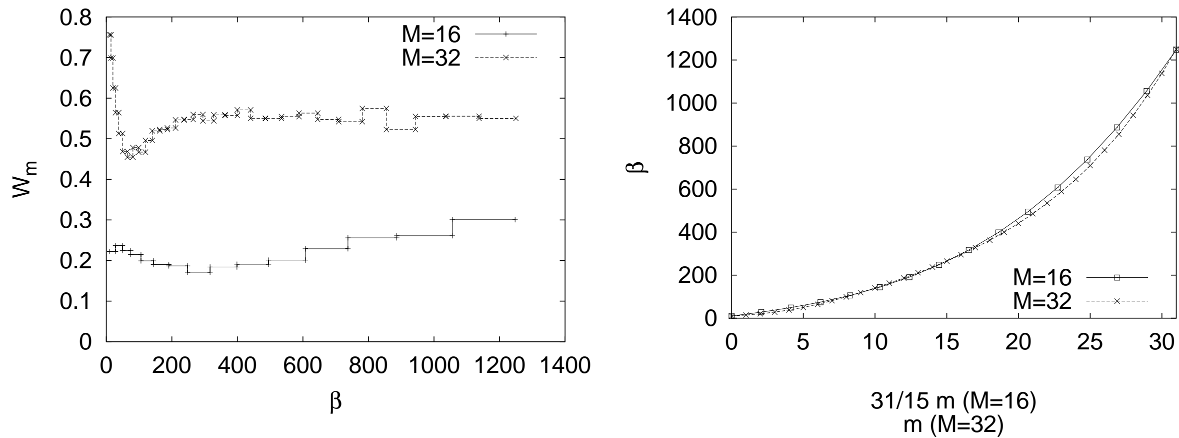


Abbildung 6. Links: Vergleich der Austauschraten für $M = 16$ und $M = 32$. Rechts: Die zugehörigen Temperaturfolgen.

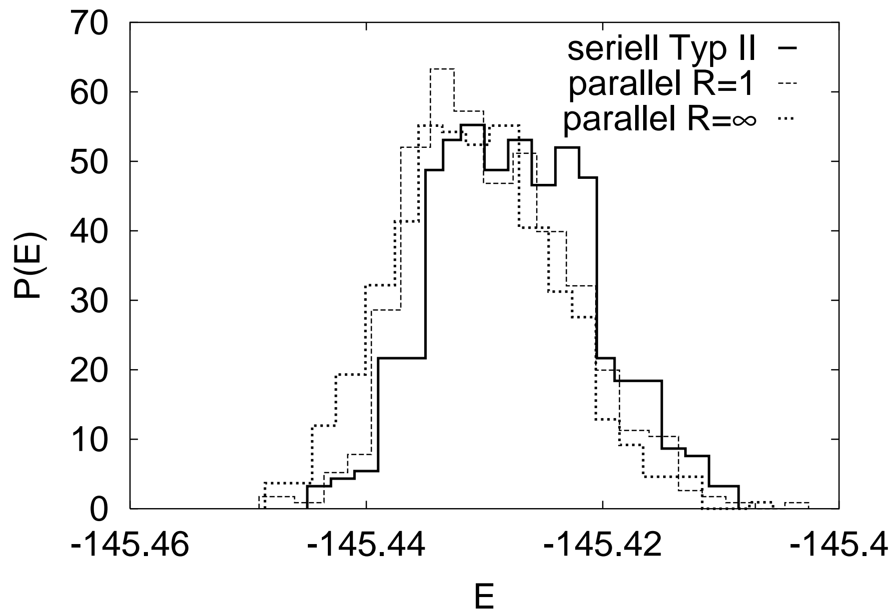


Abbildung 7. Energieverteilungen für $\beta = 1250$ ($\Theta = 0.8$) einer speziellen Unordnungskonfiguration. Vergleich des seriellen Verfahrens II mit Parallel Tempering für verschiedenes R.

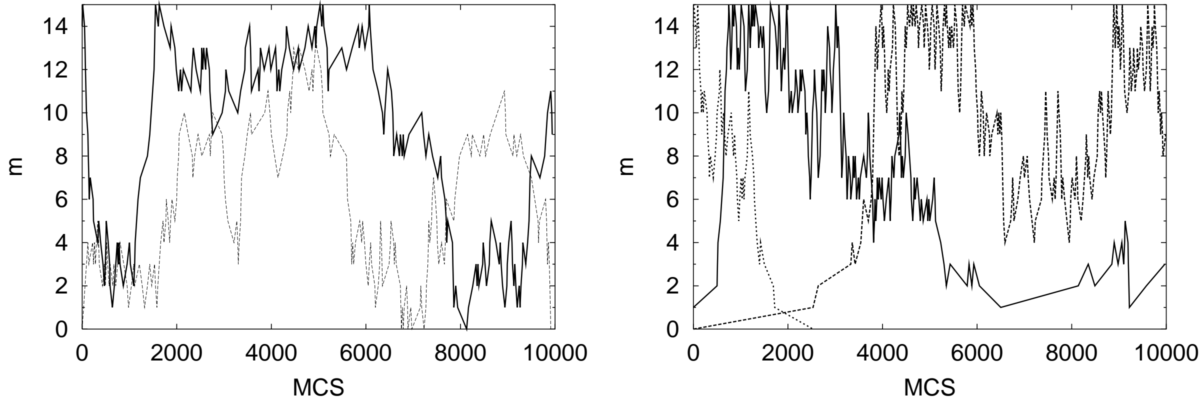


Abbildung 8. Bewegung einer Konfiguration auf der Temperaturleiter mit $R = 10$. Dargestellt ist der Index m der Temperaturstufe als Funktion der Monte Carlo Schritte (MCS). Die geraden Teilstücke der Trajektorien zeigen den Wechsel einer Temperaturstufe an. Eine lange horizontale Komponente bedeutet eine lange Verweildauer. Links: Für angepaßte Temperaturfolge. Rechts: Für äquidistante $\beta_m \propto m$.

5.3 Bewegung auf der Temperaturleiter und Relaxation

Ob das System sich hinreichend schnell auf der ganzen Länge der Temperaturleiter bewegt, kann man anhand des Stufenindex m im Verlauf der Simulation sehen. Eine ungünstige Verteilung der Austauschraten, wie bei äquidistanter Wahl der β_m, zieht für manche Systeme entsprechend lange Verweildauern auf Stufen mit niedrigen Austauschraten nach sich. Andere gelangen im Gegenzug erst gar nicht dorthin (siehe Abb. 8).

Wie zu erwarten, hängt die Beweglichkeit eines Systems stark von R ab. Für die Fälle $R = 1$ bis $R = 1000$ sind in Abb. 9 beispielhafte Trajektorien des Index m aufgeführt.

Interessant ist die Frage, wie ein solches wanderndes System relaxiert, d.h., ob und wie schnell es die Erinnerung an frühere Zustände verliert. Hierzu wurde die Korrelationsfunktion der Gesamtpolarisation $\mathbf{P}$ des Systems betrachtet:

$$C(t, \beta_m) = \overline{\mathbf{P}(0)\mathbf{P}(t)}^{(m)} \tag{7}$$

Die Polarisation des DCLG setzt sich zusammen aus den Dipolmomenten der einzelnen Ionenpaare:

$$\mathbf{P}(t) = q \sum_{i=1}^{N} (\mathbf{r}_i(t) - \mathbf{R}_i) \ , \tag{8}$$

dabei bezeichnet $\mathbf{R}_i$ den Ortsvektor des $i$ten Gegenions.

Die Zeitvariable t wird in MC-Schritten gemessen. Der hochgestellte Index $^{(m)}$ besagt, daß die Mittelung entlang des Weges durchzuführen ist, den das bei $t = 0$ auf der Temperaturstufe m startende System verfolgt. Jedes System führt gewissermaßen seine eigene Korrelationsfunktion mit.

Abb. 10 zeigt solche "Erinnerungsfunktionen" für verschiedene Parameter R. In allen vier Fällen wurden die Korrelationsdaten erst nach einer Vorlaufphase aufgenommen (50.000 MCS mit $R = 10$). Betrachtet man zunächst ein Gesamtsystem ohne Austausch ($R \to \infty$), so sieht man ein erwartungsgemäß stark unterschiedliches Relaxationsverhalten, je nach Temperatur (vgl. Abb 10). Bei der höchsten Temperatur (d.h. für β_0) relaxiert das System

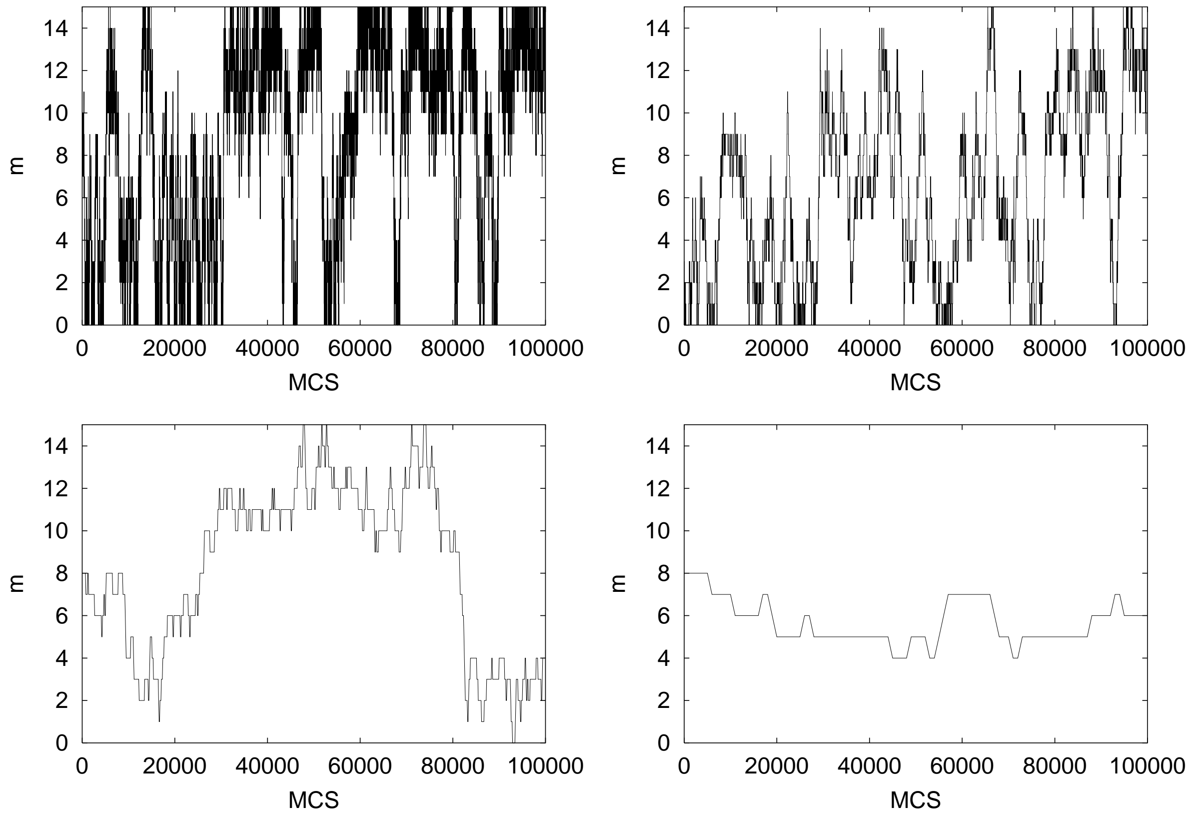


Abbildung 9. Bewegung einer Konfiguration auf der Temperaturleiter. Der Parameter R beträgt (von links oben): $1, 10, 100$ und 1000. Die Simulationsdauer betrug $2 \cdot 10^5$ MCS mit einer Vorthermalisierung bei $R = 10$ von 50.000 MCS.

wie es bei Vernachlässigung der Wechselwirkung theoretisch zu erwarten ist [6]. Man sieht eine starke Verlangsamung der Relaxation zu tieferen Temperaturen. Die Korrelationen zerfallen dort zudem nicht mehr vollständig. Der verbleibende Offset rührt her von der nicht verschwindenden mittleren Polarisation $\overline{\mathbf{P}}$, den das ungeordnete System aufgrund seiner endlichen Größe besitzt. Es bleibt ein Beitrag $\overline{\mathbf{P}}^2$ zurück. Die eigentliche Relaxation erhält man erst nach Subtraktion dieses Offsets. Die in Abb. 10 (oben links) gezeigten Daten wurden innerhalb einer relativ kurzen Simulation von $2{\cdot}10^5$ MCS erzeugt. Zu großen Zeiten hin werden statistische Ungenauigkeiten sichtbar, da dort nur wenige Werte in die Mittelung eingehen.

Für endliches, abnehmendes R bewirkt die Migration der Systemkonfigurationen eine zunehmende Vereinheitlichung der Kurvengestalt von $C(t, \beta_m)$ (vgl. Abb 10). Schnelle Relaxationen wie im seriellen Hochtemperaturfall verschwinden, auch der Offset $\overline{\mathbf{P}}^2$ nimmt offenbar ab.

Eine längere Simulation ($6 \cdot 10^5$ MCS) zeigt qualitativ das gleiche Verhalten, ebenso für eine andere Unordnungskonfiguration.

6 Zusammenfassung

Es wurde ein Programm entwickelt, mit dem die Anwendung der "Parallel Tempering" Methode als Thermalisierungsverfahren für ein Gittermodell mit positioneller Unordnung und langreichweitiger Wechselwirkung untersucht wurde.

Die Austauschraten zwischen den benachbarten Temperaturstufen hängen empfindlich von der benutzten Temperaturfolge ab. Anhand von ersten Testläufen wurde eine solche Folge manuell erstellt. Die untersuchten automatischen Verfahren zur Erstellung dieser Temperaturfolgen erwiesen sich als unzureichend.

Bei geeigneter Temperaturfolge findet eine Migration von Systemzuständen über den gesamten Temperaturbereich statt. Der wesentliche Kontrollparameter für die Frequenz voller Umläufe ist das Verhältnis R von herkömmlichen MC-Schritten und Austauschversuchen. Es sind folglich kleine Werte $R \leq 10$ zu bevorzugen.

Die "Parallel Tempering" Methode lieferte bei der interessierenden tiefen Temperatur ähnliche Energieverteilungen wie das frühere serielle Verfahren, aber tendenziell zu niedrigeren Werten verschoben. Wie groß der serielle Simulationsaufwand mindestens sein muß, wurde jedoch noch nicht systematisch untersucht. Bei der "Parallel Tempering" Methode scheint dieser – nach den bisherigen Ergebnissen – geringer zu sein.

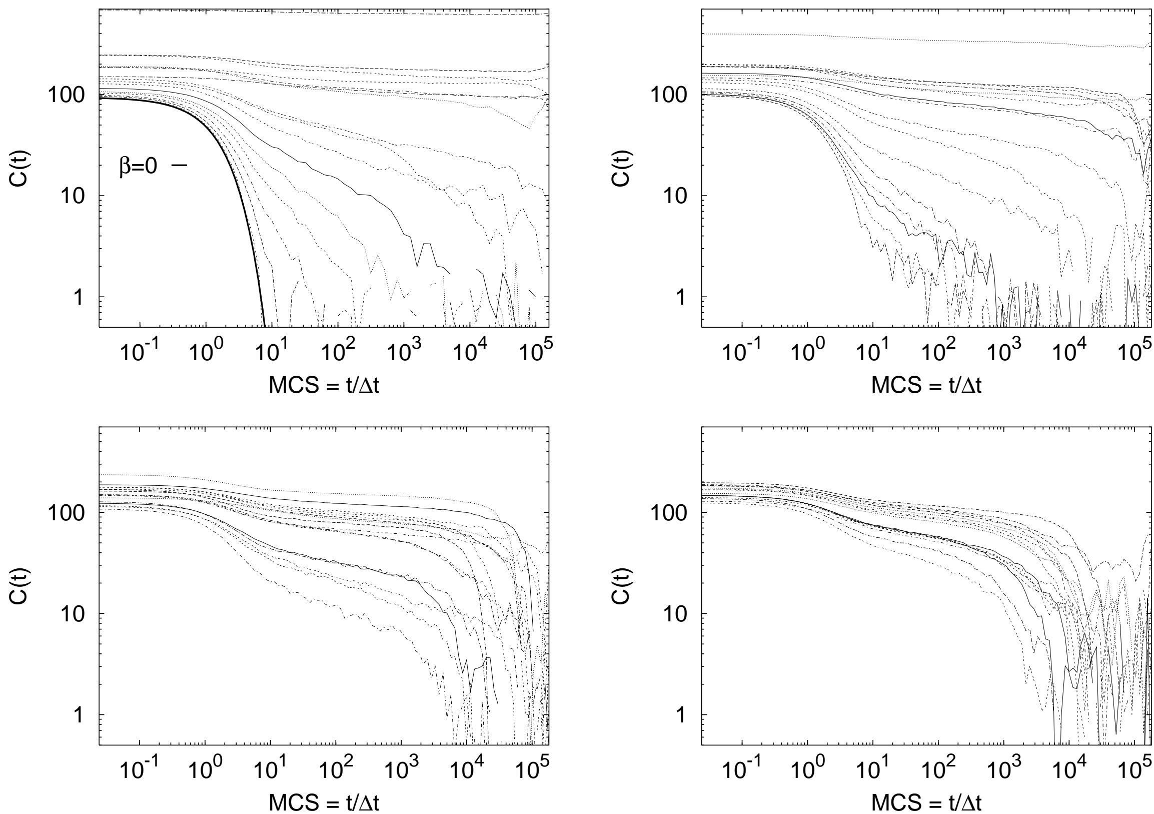


Abbildung 10. Korrelationsfunktionen $C(t, \beta_m) = \overline{\mathbf{P}(0)\mathbf{P}(t)}^{(m)}$ für eine Unordnungskonfiguration. Der Parameter R beträgt (von links oben): $\infty, 1000, 100$ und 1. Im Fall $R = \infty$ findet kein Austausch zwischen den Temperaturstufen statt. Der theoretische Verlauf für $\beta = 0$ (Hochtemperaturverhalten) ist fett eingezeichnet.

Die Relaxation der wandernden Konfigurationen wurde anhand der Polarisationskorrelationsfunktion analysiert. Die für tiefe Temperaturen typischen hohen Werte von $\overline{\mathbf{P}}^2$, treten für kleines R nicht mehr auf, das "Langzeitgedächtnis" wird reduziert. Hieraus ließ sich jedoch nicht abschließend ableiten, ob das System wirklich neue Zustände auch bei tiefen Temperaturen generiert. Zur Klärung dieser Frage sind längere Programmläufe erforderlich, um die Statistik zu großen Zeiten hin zu verbessern. Aufschluß könnte auch die gezielte Betrachtung einzelner Migrationsprozesse sowie der Anordnungen der mobilen Teilchen geben.

Die genannten Ergebnisse wurden nur für zwei Unordnungskonfigurationen des Modellsystems erzielt. Eine sicherere Bewertung der Methode erfordert die Einbeziehung mehrerer Unordnungskonfigurationen.

Mögliche Erweiterungen

Im jetzigen Algorithmus werden die Austauschpaare in fester Reihenfolge angesprochen. Eine programmtechnische Alternative wäre deren zufällige Auswahl. Durch Vorgabe der Auswahlhäufigkeit ließen sich die effektiven Austauschraten angleichen. Ein solches Verfahren käme nur zu Thermalisierungzwecken in Frage.

Zur Untersuchung der Relaxation ist anstatt der Korrelationsfunktion $\overline{\mathbf{P}(0)\mathbf{P}(t)}^{(m)}$ auch der reine Autokorrelationsanteil $\sum_i \overline{\mathbf{p}_i(t)\mathbf{p}_i(0)}^{(m)}$ denkbar. Er eignet sich vermutlich sogar besser zur Detektion von Korrelationen in der Teilchenanordnung, erfordert aber das Sammeln von Korrelationsdaten für jedes einzelne mobile Teilchen.

7 Anhang

7.1 Experimentelle Motivation des Modellsystems

Das dipolare Coulomb Gittergas greift strukturelle Aspekte auf, die man in ionenleitenden Gläsern und anderen ungeordneten dipolaren Substanzen findet. Eine weite Klasse solcher Materialien zeigt eine Frequenzabhängigkeit in der dynamischen Leitfähigkeit $\sigma = \sigma' + i\sigma''$, die nach Nowick und Mitarbeitern folgendermaßen beschrieben werden kann [9]:

$$\sigma'(\omega) = \sigma_{\rm dc} + A\omega^s + B\omega^n \qquad \text{mit } 0.6 \lesssim s \lesssim 0.7 \text{ und } n \simeq 1\ . \tag{9}$$

Dabei bezeichnet $\sigma_{\rm dc}$ die Gleichstromleitfähigkeit.

Man unterscheidet zwei Regimes: bei hohen Temperaturen oder niedrigeren Frequenzen das "Jonscher Regime" [10] mit einer Beschreibung durch einen festen Exponenten $s \approx 0.6$ bis 0.7 und thermischer Aktivierung von A, sowie bei tiefen Temperaturen oder höheren Frequenzen das "nearly constant loss regime" mit einem Exponenten nahe eins und einem nur schwach temperaturabhängigen Vorfaktor B. Ein Exponent $n = 1$ entspricht einem frequenzunabhängigen dielektrischen Verlust aufgrund der Beziehung:

$$\varepsilon'' = \operatorname{Im}\varepsilon = \frac{\sigma'(\omega) - \sigma_{\rm dc}}{\varepsilon_0\,\omega}\ . \tag{10}$$

Abbildung 11 zeigt Spektren der dynamischen Leitfähigkeit für zwei ionenleitende Gläser. Man beobachtet im linken Teil eine deutliche Ausprägung eines in der Frequenz nahezu

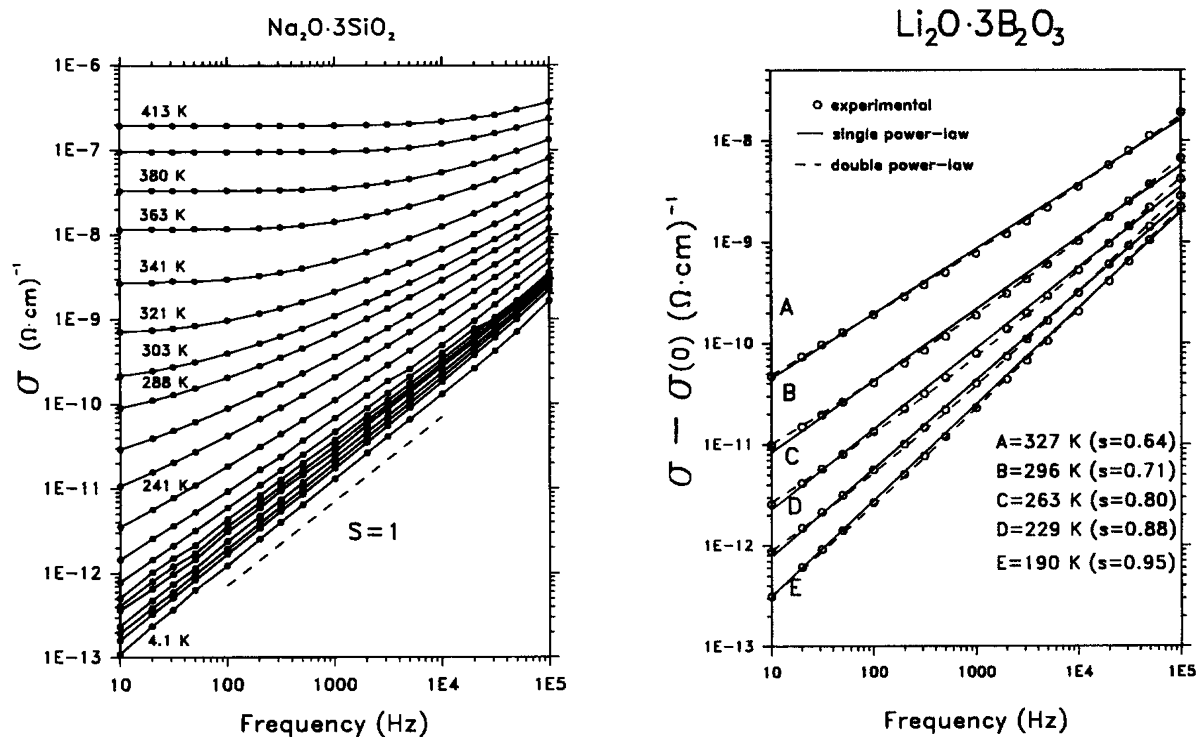


Abbildung 11. Experimentelle Leitfähigkeitsspektren ionenleitender Gläser bei verschiedenen Temperaturen nach Nowick, Lim und Vaysleyb [9]. Der linke Teil zeigt die Leitfähigkeit $\mathrm{Re}\,\sigma(\omega)$ in Abhängigkeit von der Frequenz für ein $Na_2O{\cdot}3SiO_2$-Glas bis hinab zu Heliumtemperaturen. Die gestrichelte Linie gibt einen linearen Verlauf in ω an. Im rechten Teil sind Daten eines $Li_2O{\cdot}3B_2O_3$-Glases (Symbole) dargestellt, sowie ein Vergleich unterschiedlicher Fitkurven. Der Gleichstromanteil wurde subtrahiert. Die durchgezogene Linie stellt einen Fit an Formel (9) ohne "constant loss" Beitrag dar, die erhalten Werte für s sind im Diagramm aufgelistet. Die gestrichelte Linie wurde hingegen durch eine Einbeziehung aller Beiträge gewonnen, mit festem $n = 1$ und $s = 0.61$ (der Wert für s stammt dabei aus Messungen bei höheren Temperaturen).

linearen Verhaltens zu tiefen Temperaturen bzw. hohen Frequenzen, also eines nahezu konstanten dielektrischen Verlusts. Der rechte Teil der Abbildung zeigt einen Vergleich von Fitkurven mit und ohne "constant loss" Beitrag.

7.2 Strukturelle Aspekte

Im folgenden soll Quarzglas (SiO_2) betrachtet werden. Es zeichnet sich gegenüber dem stöchiometrisch gleichartigen Quarzkristall durch das Fehlen einer periodischen Ordnung aus. Erhalten bleibt jedoch der tetraederförmige Grundbaustein aus einem Siliziumatom, umgeben von vier Sauerstoffatomen. Die Tetraeder sind über die Sauerstoffatome miteinander verbunden und bilden ein ungeordnetes Netzwerk (vgl. Abbildung 12). Zum Ionenleiter wird diese Struktur durch Dotierung mit metallischen Oxiden, oft mit Alkalioxiden, wie z.B. Na_2O, die vor dem Glasbildungsprozeß in das geschmolzene Material eingebracht werden. Die Dissoziation dieser Oxide führt zur Bildung „nichtbindender Sauerstoffatome“. Es handelt sich um Sauerstoff, der nur eine Bindung mit dem Netzwerk eingeht und daher eine negative Ladung trägt. Außerdem gibt es ungebundene, positiv geladene, metallische Ionen, die sich entlang der Zwischenräume durch das Netzwerk bewegen können.

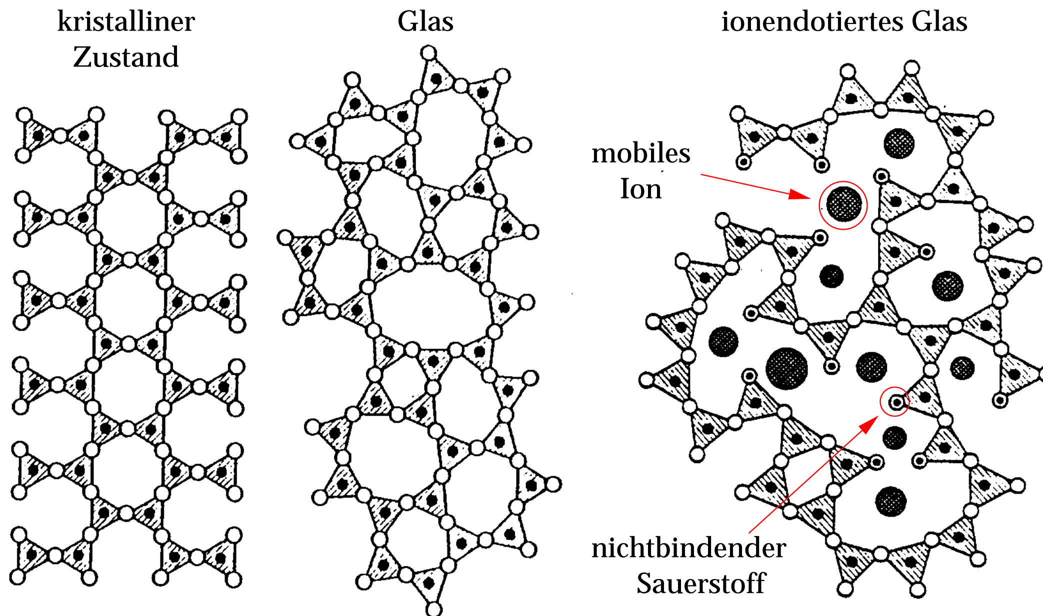


Abbildung 12. Schematische zweidimensionale Darstellung eines anorganischen Oxids, links im kristallinen Zustand, in der Mitte im ungeordneten Zustand (Glasphase) ohne gelöste Ionen und rechts als ionendotiertes Glas. Der Glasbildner (z.B. Si) ist durch den kleinen gefüllten Kreis (•), die bindenden Sauerstoffatome durch offene Kreise (◦), die nichtbindenden Sauerstoffatome durch offene Kreise mit Punkt (⊙) und die netzwerkmodifizierenden Ionen (z.B. Na^+) durch große gefüllte Kreise (●) dargestellt. Entnommen aus [11].

Bei tiefen Temperaturen ergibt sich nun die Situation, daß jedes der mobilen Teilchen innerhalb des Anziehungsbereichs eines Gegenions gefangen bleibt und nicht langreichweitig diffundieren kann. Ebenso bewirkt das Anlegen einer hochfrequenten Wechselspannung eine an die Gegenionen gebundene lokale Bewegung. Die Paare aus mobilem Ion und Gegenion stellen somit ortsfeste Dipole dar, deren räumliche Beziehung aufgrund der Netzwerks eine ungeordnete Struktur aufweist.

Das DCLG wurde speziell für diesen Fall entwickelt, d.h. tiefe Temperaturen bzw. hohe Frequenzen, ist aber auch als verallgemeinertes Modell ungeordneter Systeme mit geladenen, lokal beweglichen Defekten von Interesse.

7.2.1 Leitfähigkeit aus Fluktuationen

Im Prinzip kann man die Leitfähigkeit, ähnlich der experimentellen Situation, durch Messung des Stroms bei Anlegen eines Wechselfeldes fester Frequenz ermitteln. Diese Methode hat Knödler beim Gegenionenmodell verwendet. Zu beachten ist dabei, daß die Feldamplitude einerseits klein genug ist, damit eine lineare Betrachtung Sinn macht, und andererseits hinreichend groß ist, um ein gutes Signal-Rauschverhältnis des Stroms zu erhalten. Zudem müssen für jede Frequenz separate Simulationen durchgeführt werden [5].

Unser Modell bietet aufgrund seiner Spezialisierung die sehr praktische Möglichkeit, das Frequenzspektrum der Leitfähigkeit aus den Polarisationsfluktuationen nur einer Simulation zu gewinnen [4, 6]. Die Polarisation des DCLG setzt sich zusammen aus den Dipol-

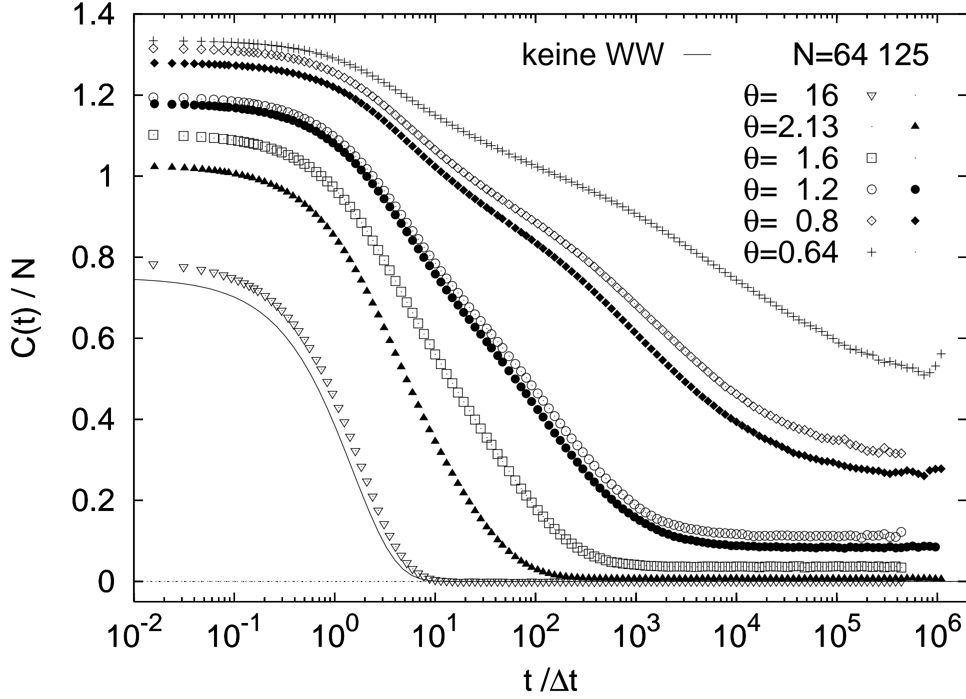


Abbildung 13. Polarisationskorrelationsfunktionen $C(t)$ normiert auf die Teilchenanzahl für $c = 1/64$, $N = 64$ bzw. 125, dargestellt für verschiedene Temperaturparameter Θ.

momenten der einzelnen Ionenpaare:

$$\mathbf{P}(t) = q \sum_{i=1}^{N} (\mathbf{r}_i(t) - \mathbf{R}_i) \ , \tag{11}$$

dabei bezeichnet $\mathbf{R}_i$ den Ortsvektor des $i$ten Gegenions.

Mithilfe der linearen Antworttheorie erhält man für den Realteil σ' der Leitfähigkeit:

$$\sigma'(\omega) = \frac{\omega^2}{3Vk_BT} \int_0^\infty S(t) \cos\omega t \, dt \ , \tag{12}$$

$$\text{mit} \quad S(t) := \left\langle\!\!\left\langle \overline{\delta\mathbf{P}(0)\,\delta\mathbf{P}(t)} \right\rangle\!\!\right\rangle = \left\langle\!\!\left\langle \overline{\mathbf{P}(0)\mathbf{P}(t)} \right\rangle\!\!\right\rangle - \left\langle\!\!\left\langle \overline{\mathbf{P}}^2 \right\rangle\!\!\right\rangle \ . \tag{13}$$

Dabei ist $S(t)$ definiert als die Korrelationsfunktion der Fluktuationen der Polarisation $\mathbf{P}$, gemittelt über verschiedene Unordnungskonfigurationen der Gegenionen (Unordnungsmittel $\langle\!\langle \cdots \rangle\!\rangle$).

Der zeitliche Mittelwert $\overline{\cdots}$ bezieht sich auf den gesamten Simulationszeitraum der Datenaufnahmephase (d. h. ohne Thermalisierungsvorläufe, vgl. Absch. 3.2). Der mittlere Polarisationsvektor $\overline{\mathbf{P}}$ wird zunächst gesondert bestimmt und sein Quadrat erst nachträglich subtrahiert (die Fluktuationen $\delta\mathbf{P} = \mathbf{P} - \overline{\mathbf{P}}$ sind in der laufenden Simulation nicht angebbar).

Abb. 13 zeigt Polarisationskorrelationsfunktionen $C(t) := \left\langle\!\left\langle \overline{\mathbf{P}(0)\mathbf{P}(t)} \right\rangle\!\right\rangle$ eines Systems der Konzentration $c = 1/64$ mit $N = 64$ bzw. $N = 125$ Teilchenpaaren bei verschiedenen Temperaturparametern Θ gemittelt über 100 Unordnungskonfigurationen der Paarzentren. Man sieht, daß sich die Relaxationen mit abnehmender Temperatur deutlich verlangsamen und nicht mehr vollständig zerfallen (vgl. Abschn. 5.3). Im Spektrum der Leitfähigkeit, das man mittels Gleichung (12) gewinnt, zeigt sich ein Bereich, in dem $\sigma'(\omega)$ ungefähr linear von der Frequenz abhängt. Dieser Bereich dehnt sich zu tiefen Temperaturen hin aus.

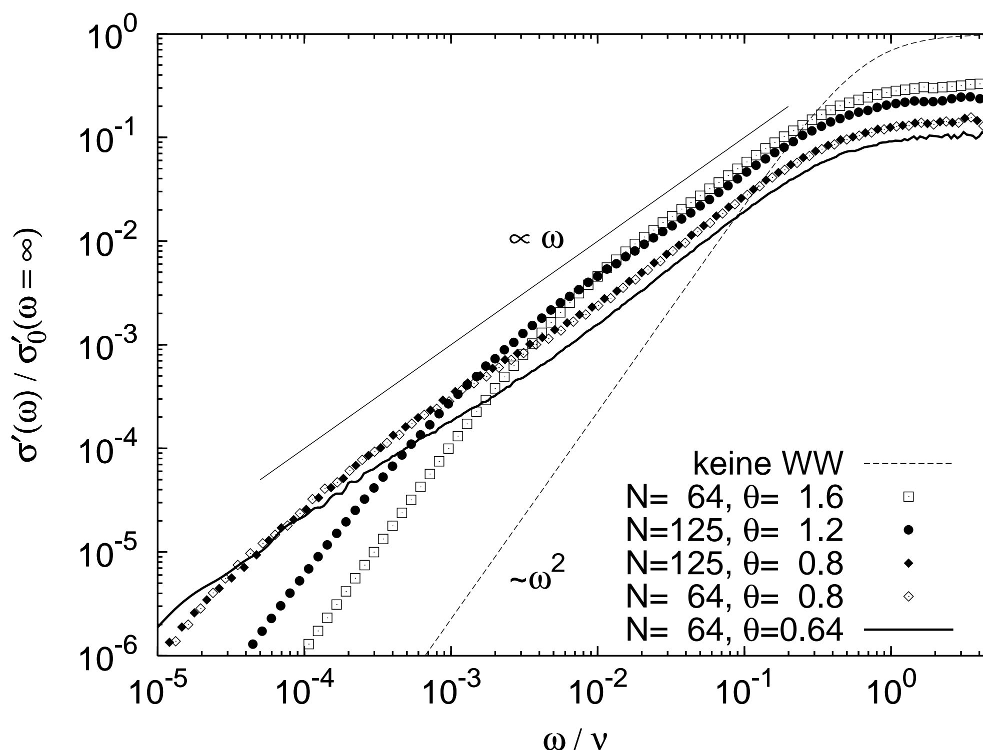


Abbildung 14. Leitfähigkeitsspektren (Realteil) für verschiedene Temperaturen bei Systemparametern $c = 1/64$, $N = 64$ bzw. 125. Die durchgezogene Gerade gibt ein ideales ω^1-Verhalten an. Die gestrichelte Kurve bezeichnet den wechselwirkungsfreien Fall. Auf dessen Hochfrequenzleitfähigkeit $\sigma_0'(\omega = \infty)$ sind alle Daten normiert.

Literatur

1. K. Hukushima und K. Nemoto, J. Phys. Soc. Jpn. **65**, 1604 (1996)
2. W. Kob, C. Brangian, T. Stühn und R. Yamamoto, e-print cond-mat/0003282
3. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller und E. Teller, J. Chem. Phys. **21**, 1087 (1953)
4. P. Pendzig, *Gittergasmodelle zur Dynamik ionenleitender Gläser und Polymere*, Dissertation Universität Konstanz, Hartung-Gorre Verlag Konstanz 1997
5. D. Knödler, *Untersuchungen eines Gittergasmodells zum Ionentransport in Gläsern*, Dissertation Universität Konstanz 1994, Hartung-Gorre Verlag 1995
6. T. Höhr, *Dipolare Relaxationen in ungeordneten Systemen*, Diplomarbeit Universität Konstanz, 2000
7. S.W. de Leeuw und J.W. Perram, Physica **107A**, 179 (1981)
8. S.W. de Leeuw, J.W. Perram und E.R. Smith, Proc. R. Soc. Lond. **A 373**, 27 (1980)
9. A.S. Nowick, B.S. Lim und A.V. Vaysleyb, J. Non-Cryst. Solids **172-174**, 1242 (1994)
10. A.K. Jonscher, Nature **267**, 673 (1977)
11. J.M. Stevels, *Handbuch der Physik*, Bd. 13, hrsg. v. S. Flügge, Springer Verlag, Berlin, 1962

Parallelization of the Multigrid Solver for Flows in Complex Geometries: FASTEST2D-LBR

Jun-Mei Shi

Friedrich - Alexander University Erlangen-Nuremberg
Institute of Fluid Mechanics (LSTM)
shi@lstm.uni-erlangen.de

Abstract: The present paper is a documentation for the parallelization of the general flow and transport problem solver FASTEST2D-LBR. Firstly, the numerical methods used in the code are explained, then the essential parts of the parallelization procedures including block structured grid partitioning, block connecting, data dependence handling at block interfaces, data structures in the communication and the implementation of the communication in the code are discussed in detail.

1 Introduction

FASTEST2D-LBR [1] is a local-block-refinement extension of the parallel multigrid flow solver FASTEST2D [2], developed at LSTM Erlangen. The code uses a *block-structured, boundary-fitted non-orthogonal* grid and the *finite volume* method for the spatial discretization of the governing equations. The basic communication structure was implemented based on TCG during the extension, but the code was not able to run in parallel due to many bugs. The aim of the present work was to implement MPI as the communication library between processors in the code, to check and to debug the code to make it run in parallel on Cray-T3E and to improve the available communication structure. Therefore, this work will need to go through the complete structure of the code, from the preprocessing (grid generation, grid partitioning, and setting up the connection between blocks) to the program I/O, the communication in the inner iteration and outer iteration and the control of the solving procedure. The communication between processors with non-matching interface is the key point of the parallelization.

2 Governing Equations and Discretization Procedure in FASTEST2D

2.1 Governing equations

FASTEST2D is a general solver for $2D$ transport phenomenon. In Cartesian coordinates system, the concerned governing equations to describe the conservation of mass, momentum, energy and specific concentrations can be expressed in the following general form:

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{unsteady term}} + \sum_i \underbrace{\frac{\partial(\rho U_i \phi)}{\partial x_i}}_{\text{convection}} - \sum_i \underbrace{\frac{\partial}{\partial x_i}\left(\Gamma_\phi \frac{\partial \phi}{\partial x_i}\right)}_{\text{diffusion}} = \underbrace{Q_\phi}_{\text{source}} \tag{1}$$

where the meaning of each term is also explained. ϕ represents the dependent variable and Γ_ϕ is the corresponding diffusion coefficient. As an example, the values of these variables for laminar flow and heat transfer in incompressible fluid are shown in Table 1.

Equation	ϕ	Γ_ϕ	Q_ϕ
Continuity	1	0	0
Momentum	U_j	μ	$-\frac{\partial P}{\partial x_j} + \frac{\partial}{\partial x_i}\left(\mu \frac{\partial U_i}{\partial x_j}\right) + \rho g_j$
Energy	$c_P T$	$\frac{\lambda}{c_P}$	$\frac{\partial P}{\partial t} - \tau_{ij}\frac{\partial U_i}{\partial x_j}$

Table 1. Values of variables in the general transport eq. (1)

It should be pointed out that the molecular momentum transport term in the momentum equation is $\frac{\partial \tau_{ij}}{\partial x_j}$, which is divided into a diffusion term and part of the source term for the convenience of numerical treatment. For incompressible Newtonian fluid, τ_{ij} is defined as:

$$\tau_{ij} = -\mu\left(\frac{\partial U_j}{\partial x_i} + \frac{\partial U_i}{\partial x_j}\right). \tag{2}$$

For the numerical solution of initial and boundary value problems of the above equations, the spatial and time discretizations should be made to approximate the differential equations by a system of discrete algebraic equations based on the numerical grids and the discrete time steps. The discretization methods are described in the following two subsections.

2.2 Spatial discretization for complex geometries

The finite-volume (FV) method is chosen in FASTEST2D for the spatial discretization of the governing equations. The starting point of this method is the integral form (3) of the general transport equation, where Ω represents the volume integration and S the surface integration, respectively. With this approach, the computational domain is divided into a finite number of contiguous control volumes (CVs), the dependent variables are defined at the centroid nodes of the CVs. The conservation equations are applied to each CV. Two levels of approximations are introduced in the discretization. One is to express the CV volume and face integrals in terms of values at some discrete locations, the other is used to interpolate values at the CV faces with the nodal values. As a result, the differential equations are discretized into a set of algebra equations based on each CV. The advantage of this approach is that it can accommodate any type of grids and the global conservation can be automatically guaranteed provided that the surface integrals are kept the same for CVs which share the interface.

$$\int_\Omega \frac{\partial(\rho\phi)}{\partial t} \mathrm{d}\Omega + \sum_{\mathrm{i}} \int_S (\rho U_i \phi) n_i \,\mathrm{d}S - \sum_{\mathrm{i}} \int_S (\Gamma_\phi \frac{\partial \phi}{\partial x_i}) n_i \,\mathrm{d}S = \int_\Omega Q_\phi \,\mathrm{d}\Omega \tag{3}$$

To solve problems involving complex geometries, *oriented* (**N**orth, **S**outh, **E**ast, **W**est), *block-structured*, *boundary-fitted non-orthogonal* quadrilateral grids are chosen in FASTEST2D for the domain discretization. The governing equations to solve are expressed in the Cartesian coordinates system in order to avoid non-conservative source terms in the momentum equations. A *colocated arrangement* of the pressure variable and velocity components in the CV center is used. An example of a typical CV is shown in Fig. 1. Based on the above definitions, the volume and surface integrals in equation (3) can be separately discretized for each CV.

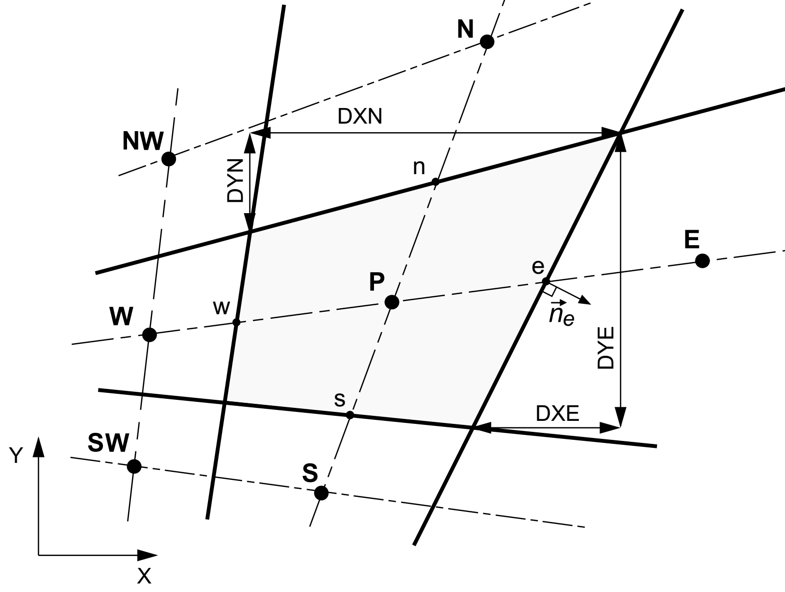


Figure 1. An example of a typical CV.

The **convective fluxes** can be obtained by the sum of convective fluxes through each CV face.

$$\sum_{\mathsf{i}} \int_S (\rho U_i \phi) n_i \, \mathrm{d}S = F^C_{\mathsf{e}} - F^C_{\mathsf{w}} + F^C_{\mathsf{n}} - F^C_{\mathsf{s}} \tag{4}$$

Firstly, these fluxes are approximated by using the midpoint rule. For example, at the east face,

$$F^C_{\mathsf{e}} = \sum_{\mathsf{i}} \int_{S_{\mathsf{e}}} (\rho U_i \phi) n_i \, \mathrm{d}S \approx \dot{m}_{\mathsf{e}} \phi_{\mathsf{e}} = \sum_{\mathsf{i}} (\rho U_i n_i)_{\mathsf{e}} S_{\mathsf{e}} \phi_{\mathsf{e}} \tag{5}$$

Then the deferred correction scheme proposed by Khosla and Rubin [3] is used for the interpolation of the values of the transport quantities at CV faces in term of those at CV-centers. This approach uses an implicit *upwind differencing scheme* (UDS) approximation combined with an explicit treatment of the difference between the UDS and the *central differencing scheme* (CDS) approximations. A factor of $0 \leq \beta \leq 1$ is used to blend the two schemes in the following way:

$$F^C_{\mathsf{e}} = F^{C(\mathsf{UDS})}_{\mathsf{e}} + \beta \, (F^{C(\mathsf{CDS})}_{\mathsf{e}} - F^{C(\mathsf{UDS})}_{\mathsf{e}})^{old} \tag{6}$$

In case of the momentum equations, the convective flux is a nonlinear term. Old values of mass fluxes (e.g., $\dot{m}_{\mathsf{e}}$) are used for the linearization.

The **diffusive fluxes** are also discretized using the midpoint rule. The corresponding expression for the east face can be written as

$$F_{\mathsf{e}}^{D} = \sum_{\mathsf{i}} \int_{S_{\mathsf{e}}} \Gamma_\phi \frac{\partial \phi}{\partial x_i} n_i \, \mathrm{d}S \approx \sum_{\mathsf{i}} \left(\Gamma_\phi \frac{\partial \phi}{\partial x_i} n_i \right)_{\mathsf{e}} S_{\mathsf{e}} \; . \tag{7}$$

A CDS-like linear interpolation is used for the calculation of the normal gradient at the CV face in eq. (7). The normal gradient is calculated in the local coordinate system, as a result, the transformation between the Cartesian and the local coordinate system will be required in the general case of a non-orthogonal and non-uniform grid. *Cross diffusion* contribution resulting from non-orthogonal grids, which contains values at nodes SW, SE, NW, NE, is treated explicitly so that the resulted algebraic equations remain five-point. The volume integral of the **source term** in eq. (3) is simply approximated as a product of the quantity value at the CV center and the CV volume, $\Delta\Omega$. This discretization is of second order accuracy (see Ferziger and Perić [4]).

$$q_{\mathsf{P}}^{\phi} = \int_{\Omega} Q_\phi \, \mathrm{d}\Omega \approx Q_{\phi,\mathsf{P}} \Delta\Omega \; . \tag{8}$$

It should be pointed out that such a treatment for the **pressure derivatives** in the source term of the momentum equations is non-conservative.

Without including the unsteady term, the above spatial discretization procedures lead to a *five-point* algebraic equation for each CV:

$$a_{\mathsf{P}} \phi_{\mathsf{P}} + \sum_{\mathsf{nb}} a_{\mathsf{nb}} \phi_{\mathsf{nb}} = q_{\mathsf{P}}^{\phi} \; , \tag{9}$$

where P is the concerned CV node and nb stands for its four neighbors E, W, N and S.

2.3 Time discretization

Based on the spatial discretization described above, eq. (3) can be rewritten as

$$\int_{\Omega} \frac{\partial (\rho\phi)}{\partial t} \mathrm{d}\Omega = q_{\mathsf{P}}^{\phi} - a_{\mathsf{P}} \phi_{\mathsf{P}} - \sum a_{\mathsf{nb}} \phi_{\mathsf{nb}} \equiv \mathbf{L}(\phi) \; . \tag{10}$$

For the discretization of the time derivative, the integral and differential operators in eq. (10) are firstly exchanged and the volume integration is approximated in terms of the mean value at the CV center:

$$\frac{\partial}{\partial t} \left[\int_{\Omega} (\rho\phi) \mathrm{d}\Omega \right] \approx \frac{\partial}{\partial t} \left[(\rho\phi)_{\mathsf{P}} \Delta\Omega \right] \; . \tag{11}$$

Then, the time derivative is discretized by a finite difference scheme. Depending on the time point, at which the time derivative is approximated (denoted t^κ), we distinguish three schemes:

$$\frac{(\rho\phi\Delta\Omega)^{n+1} - (\rho\phi\Delta\Omega)^{n}}{\Delta t} \approx \mathbf{L}(\phi)^{\kappa} \tag{12}$$

- *Euler explicit*: approximation at the beginning of the step ($\kappa = n$);

- *Euler implicit*: approximation at the end of the step ($\kappa = n + 1$);
- *Crank-Nicolson*: approximation at the middle of the step ($\kappa = n + 1/2$).

All these schemes are implemented in FASTEST2D, a detailed discussion can be found in Lange [1].

3 Solution Method of FASTEST2D

3.1 Pressure Correction Method

Since the transport equations are coupled, iterative solvers are best suited for the solution of the resulted algebraic equation system. A special feature of the Navier-Stokes equations (continuity and momentum eqs) lies in that there is no independent equation for the dependent variable pressure. A Poisson-like equation can be derived for the pressure from the N-S equations, which shows that the pressure field is physically constrained with the continuity equation. In FASTEST2D, the SIMPLE-algorithm of Patankar and Spalding [5] is used to treat this pressure-velocity coupling. It uses a predictor-corrector procedure for the conservative determination of pressure and velocity fields. The algorithm is shown in Fig. 2. The solution of the pressure in N-S equation with colocated variable arrangement requires special treatment, detailed information can be found in Ferziger and Perić [4].

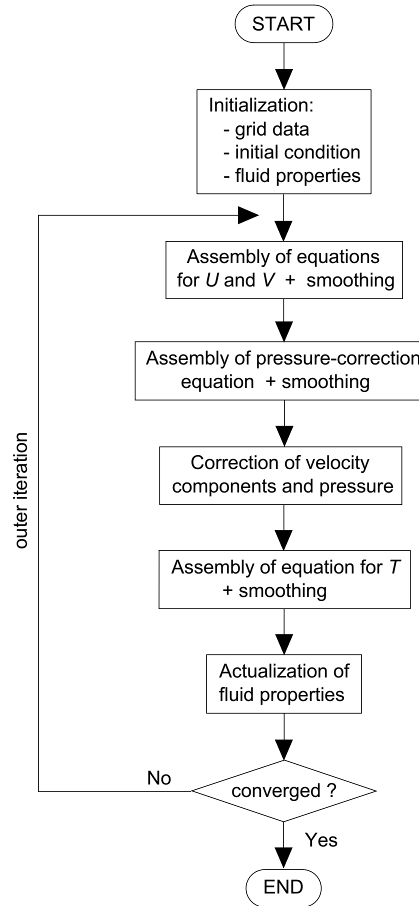


Figure 2. Iterative solution procedure of SIMPLE algorithm.

A *full approximation scheme* (FAS) of the multigrid method is implemented in FASTEST2D as the iterative solver for the algebraic equation system. An *Incomplete LU factorization method*, called Stone's [6] *strong implicit procedure* (SIP), is used as the smoother. In the SIP smoothing, the matrix of the linear equation system remains unchanged. The iteration in SIP is called inner iteration. In the iterative multigrid pressure-correction solving procedure, the coefficients of the dependent variables and the corresponding source terms are updated and the equation system is reassembled after each iteration outside the SIP solver. This type of iteration is called outer iteration. This procedure has shown to be very robust, efficient and also well suited for parallelization [7], [8].

As FASTEST2D uses *block-structured* grids, the solution procedures and the solver are separately applied to each block. It provides a good basis for the parallelization. The data dependence among neighbor blocks will be discussed in detail in the following section. For detailed discussion on the parallel variant of the SIP solver refer to Durst and Schäfer [7].

4 Parallelization

4.1 Preprocessing – Block structuring, local block refinement, block partitioning

Block-structured grids used in FASTEST2D are a useful compromise to retain the simplicity and numerical efficiency for the algorithm and the ability to handle complex geometries, conjugated problems with different properties or problems requiring various grid systems moving against each other [7]. The intention of the local block refinement is to efficiently treat problems requiring high local resolution and at the same time to retain the structured grid. Both for parallel and sequential computing, the first step in preprocessing is block structuring. That means, the computational domain is subdivided into many *geometrical blocks*. In each geometrical block, *boundary-fitted, structured grids* (very often non-orthogonal) are defined for all multi-grid levels. Grid refinements are applied to geometrical blocks where higher resolution is required than in their neighbor blocks. The principle here is to generate block-structured grids of good properties according to the geometry of the computational domain and the physical problem. The grid generating process is a recursive refinement process from the coarsest grid level to the finest one. Grid smoothing should be applied to improve the grid quality. The final grid data for each level is defined from those at the finest level. A recursive process in the opposite direction is applied to extract a subset of grid data from the finer level for the next coarser one. This measure is important in view of the multigrid method in order to have a consistent set of grid data.

Cray-T3E uses a distributed memory architecture, hence a SPMD (Single-Program-Multiple-Data)programming model is used. The second step of the preprocessing is the domain re-decomposition and the distribution of the resulting subdomains to the parallel processors. The principle here is to manage the load balancing among parallel processors. The block-structuring in the first step has provided a good basis for this purpose. A *grid partitioning* technique is applied to transform the *geometrical block structure* to a new block structure (*parallel block structure*) by means of a mapping process. In general, given NB geometrical blocks and NP working processors, three situations have to be considered in an automatic partitioning:

1. $NB = NP$: the parallel blocks are the same as the geometrical blocks, one block is assigned to each processor;

2. $NB > NP$: the geometrical blocks are re-grouped to form NP groups and the resulting groups are assigned to individual processors, see Fig. 3;

3. $NB < NP$: the geometrical blocks are partitioned into totally NP new parallel blocks to fit the number of processors, see Fig. 4.

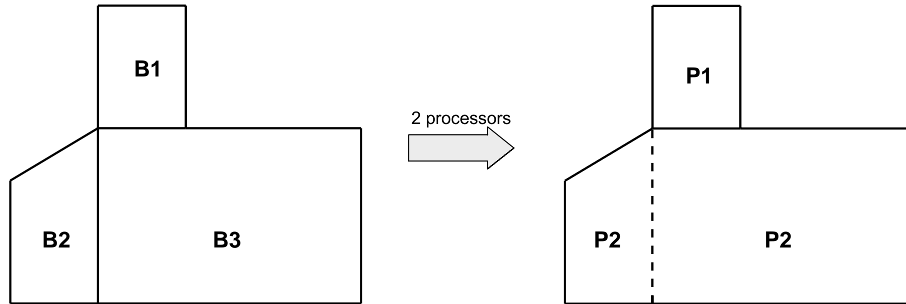


Figure 3. Example of block grouping when $NB > NP$.

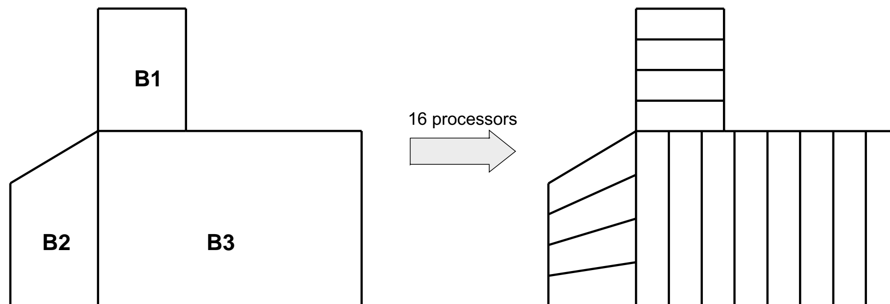


Figure 4. Example of grid partitioning when $NB < NP$.

In cases (1) and (3), one block is assigned to each processor. In case of $NB > NP$, more blocks can be assigned to one processor, hence sequential computing will be performed to the blocks by this processor. Therefore, the number of the parallel blocks(groups) is equal to the number of available processors (NP) while the number of *unit blocks* for the solver can be more than the number of geometrical blocks (NB). Instead of decomposing the domain automatically, a manual mapping between geometrical blocks and processors can also be defined. In practice, the manual partitioning might be better for an experienced user. A suitable grid partitioning is important to achieve good parallel efficiency, the principles for the optimization are discussed in detail in Durst and Schäfer [7].

4.2 Preprocessing – Setting up connection between blocks

The connection information between blocks is also established in the preprocessing and will be used for the handling of data dependence among neighbor blocks (discussed in section 4.3). Each segment of the interface along each side of two neighbor blocks is called a patch. Therefore, each segment of the interface consists of a pair of patches from two sides. For each side (E, W, S, N) of each block, the total number of the neighbor patches are calculated. For each local patch of the same side, the buffer addresses and the number

of elements for the send and receive operations are calculated for each grid level. The processor number and the block number (local in a processor) of the neighbor patch and the local number of the neighbor patch in its block and processor are identified. Then the total count (global) of the inter-patches is calculated along each side of each block. Finally, a global array is defined to record the side ID (E, W, S, N) of the send patch, the processor number, the local block number and the local patch number of the send patch and of the corresponding receive patch, respectively.

It should be pointed out that such a definition for the communication is only one-way. With this definition, each patch knows its own send and receive buffer and counts, but it doesn't have this information of its neighbor patch at the same time. As a result, in the present implementation, the required information has to be exchanged between neighbor blocks for the initialization of the communication. An indirect definition of the data structure used for the communication has to be used (defined by the neighbor block). An improvement is here necessary to use a two-way definition.

4.3 Data dependence handling at matching and non-matching block boundaries

As the solving procedure is based on the unit block, data dependence (or block coupling) among neighbor blocks occurs. In other words, a boundary CV of one unit block will need information from its neighbor CVs ($E, W, N, S, NE, NW, SE, SW$) for the computation, some of which are located in the neighbor block. Therefore, data transfer among neighbor blocks is required. In FASTEST2D, one layer auxiliary CVs is additionally defined around each block boundary to store the information of the neighbor block boundaries. The auxiliary CVs belong to the normal data structure of the blocks. With this strategy, the data in each block become self-sufficient.

In case of a **matching interface**, the boundary CVs of the two neighbor blocks match each other. Data can be simply exchanged between them. During the computation, each auxiliary CV receives the information from its corresponding CV of the neighbor block, that is, mirroring this corresponding CV for the current block. This procedure is illustrated by Fig. 5. When the two neighbor blocks are assigned to the same processor, data interexchange is realized by a copy process. Otherwise, communication (here based on MPI) is necessary. Detailed discussion on the communication between processors will be given in section 4.5.

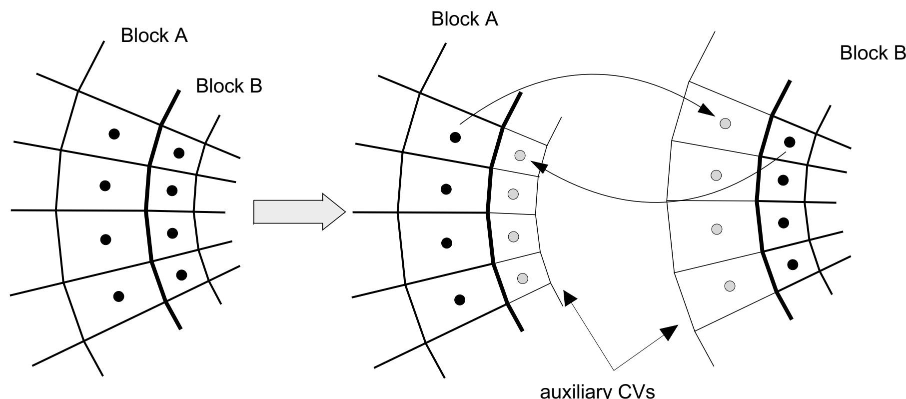


Figure 5. Block interface with auxiliary CVs.

In case of local block refinement, the boundary CVs at both sides of the block interface do not match each other(**non-matching block interfaces**). As a result, the boundary CVs of the neighbor block can not directly be used as the information provider for the auxiliary CVs of the other block. To get information from the neighbor block under the same data structure as the matching case, one layer of virtual CVs matching the current patch at the interface but located in the neighbor block are constructed to serve as information provider for the required auxiliary CVs of the current block. With the help of the virtual CVs, the data "mirroring" procedure can again be used. This idea is demonstrated in Fig. 6. As mentioned above, the auxiliary CVs belong to the data structure of the present block, but they are exact shadows of the constructed virtual CVs in the neighbor block.

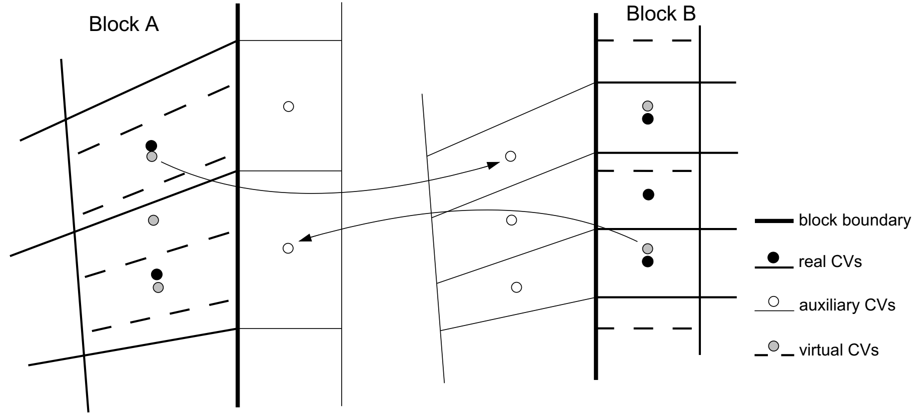


Figure 6. Data transfer from virtual to auxiliary CVs.

The virtual CVs do not belong to the normal data structure of the neighbor block, thus values of the dependent variables and the coefficients at the virtual CVs are not defined in the computation. Therefore, their values should be interpolated from the boundary CVs close to it at each iteration during the computation before the data is transferred to the corresponding auxiliary CVs of the neighbor block.

As can be noticed from Fig. 6, the number of the virtual CVs is determined by the required number of the auxiliary CVs of the current block while their geometrical properties are in the available implementation defined by fitting to the grid structure of the boundary CVs of the neighbor block (the block where virtual CVs are located). This definition simplifies the data interpolation procedure for quantities at the virtual CVs and the virtual CVs automatically coincide with the boundary CVs, thus no interpolation is needed in case of a matching patch. However, very non-smooth grids between the auxiliary CVs and the boundary CVs might result from this definition in case that the boundary grid structures on both sides of inter-patch are in large difference. As a result, large numerical errors might be caused due to this interface handling when very different grid structures at the boundary are found on both sides. An alternative is to construct the auxiliary CVs (or virtual CVs in the neighbor block) according to the grid structure of the current block. This strategy will guarantee a good connection between the auxiliary CVs with the current block. At the same time, the interpolation procedure will become more complicated. With this handling method, virtual CVs are needed both in matching and non-matching cases.

4.4 *Handling of data dependence at special block corners*

In case of non-orthogonal grids, data at the "diagonal" neighbor CVs (NW, NE, SW and SE) are needed to account for the the contribution of the cross diffusion. The corner auxiliary CVs are used to store information from these "diagonal" neighbor CVs. Therefore, corresponding virtual CVs should also be additionally constructed as the information providers. In the present case containing non-matching interfaces, the definition of the corner auxiliary CVs (or the corresponding virtual CVs) is not straightforward in the following two cases:

- junction of three blocks or T-junction;
- junction of four blocks or cross-junction.

In case of a *T-junction*, one of the blocks has two contiguous interface patches (for instance, see block A in Fig. 7). The special point here is, the two additional virtual CVs in block A are needed for the two corner auxiliary CVs of the neighor blocks (B and C). On the other hand, no additional virtual CV in block B and C is required for the two contiguous patches of block A. As a result, the send and receive buffers and counts should be exceptionally modified. In fact, following the construction rule of the virtual CVs, the additional virtual CV of a corner auxiliary CV coincides with the first or last normal virtual CV of another block which shares the same corner, but they are different virtual CVs belonging to different contiguous patches in the communication. Therefore, it seems in Fig. 7 that two send arrows set off from the "same" virtual CV.

Data transfer is only made between the neighbor patches. Therefore, in case of a *cross-junction* corner (Fig. 8), the cross data transfer (e.g., between B and C) is indirect. The reason is to avoid transfering a single data between processors, which is very inefficient. As shown in Fig. 8, two ways are possible from block B to block C depending on the sequence of the data exchange procedure, but the result is unique in case of a cross-junction with four fully matching blocks. Problem arises in case of non-matching blocks (Fig. 9). Like the fully matching cases, the value of the corner auxiliary CV depends on the sequence of the data exchange procedure. However, the value is not unique because the data transferred in different ways are from different corresponding virtual CVs, thus are different. The second problem is, that the corner auxiliary CV doesn't match the two possible virtual CVs. Therefore, the corner auxiliary CV should be modified according to the centers of the two possible virtual CVs and interpolation is applied to calculate the value of quantities at the modified corner auxiliary CV center.

4.5 *Implementation of the parallelization*

4.5.1 *Additional data structure for the communication*

Interpolation and communication are required at each inner and outer iteration to calculate the values of dependent variables and matrix coefficients for the virtual CVs in order to update the information for the corresponding auxiliary CVs of the neighbor block. Two global arrays, one integer and one real, respectively, are created for each side (W, E, S, N) in each processor. The real array is a long one-dimensional array to store the virtual data of all the patches of all blocks(in the same processor) on the same side and for all

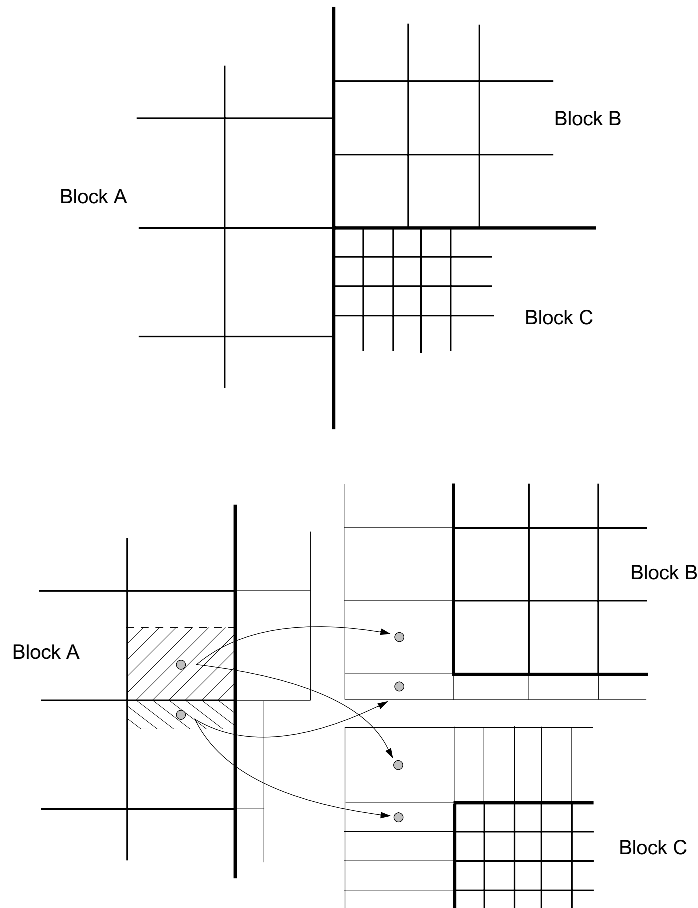


Figure 7. Definition and data transfer of virtual CVs at block corners in case of a T-junction of blocks.

grid levels as demonstrated in Fig. 10. The integer array is used for the indexing of the real array. Its two elements are pointers to the starting address and length of the patch in the real array. Two additional arrays indexed in the same way are also defined for the interpolation. One points to the global indices of the base nodes for the interpolation, the other is used to store the values of the interpolation factor of the base nodes.

These arrays are overdimensioned and the correct indexing of these arrays is a critical issue under this data structure. Firstly, the location of a patch becomes difficult to be identified in case that the array contains several patches of several blocks. The number of elements of a patch is indirectly determined by the neighbor patch (according to the number of auxiliary CVs) and should be correspondingly modified when the neighbor patch contains special corners. The same holds for the starting index of a patch. As a result, the implementation becomes more complicated than necessary, bugs may easily occur and the code becomes difficult to debug. A suitable data structure is the basis for a correct communication. Better data structure, e.g., based on each side of each block and also for each grid level, should be implemented. The improvement should begin with the preprocessing to use a two-way definition for the communication, so that a direct indexing for the virtual data array will become possible. That will also simplify the communication required for the construction of the virtual CVs in the initialization of the program.

The geometrical quantities of the virtual CVs are interpolated side by side during the initialization step of the program. Immediately after the interpolation, they are sent to the

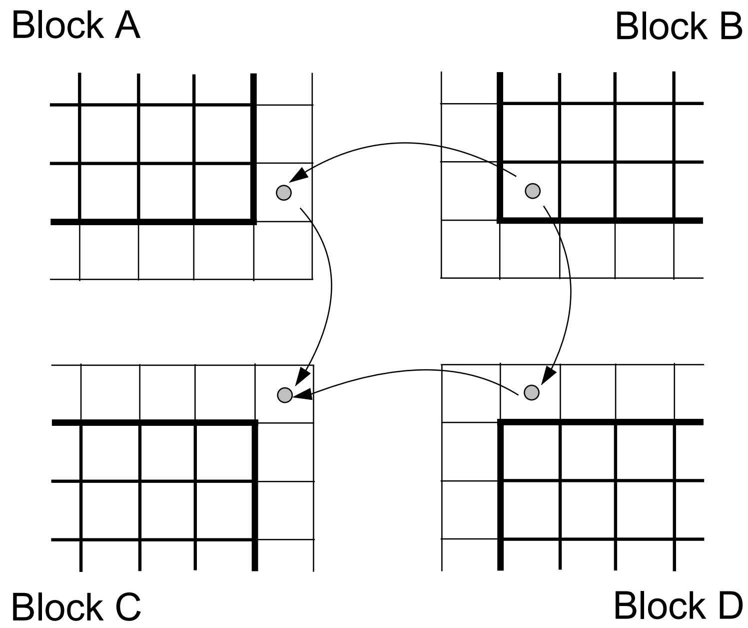


Figure 8. Two possible ways of data transfer at cross-junctions of regular fully structured blocks.

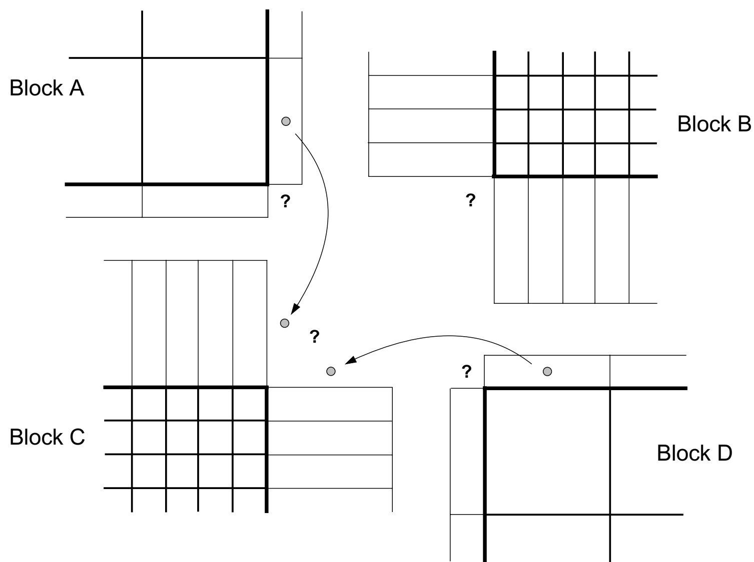


Figure 9. Data transfer problem at cross-junctions of locally refined blocks. Solution may depend on the block communication sequence.

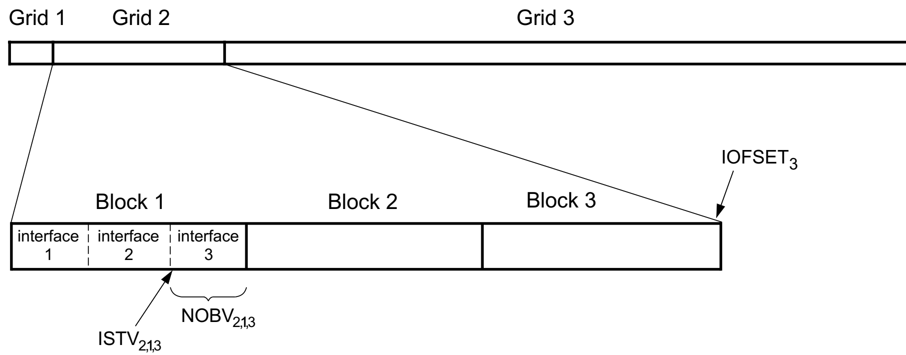


Figure 10. Example of array structure for virtual data.

corresponding auxiliary CVs in the neighbor block, and are not used any more. Therefore, the supplementary arrays needed for the interpolation are defined to be local to the subroutine.

4.5.2 Implementation of communications in the parallel Program

Communication is required in three parts of the program: I/O, initialization and solving procedure. They are listed as follows and the communication in the solving procedure is demonstrated in Fig. 11:

1. **I/O and Initialization:**
   - Initialization of communication structures for program I/O and block communications;
   - Master processor reads the input file (physical and numerical parameters, initial conditions, etc.) and broadcast these data to all processors;
   - Master processor reads the grid file which is stored in the decreasing order of the processor number (from $NP - 1$ to 0) according to the grid partitioning and mapping, and sends them to the corresponding processors;
   - In case of a restart computation, the master processor reads the restart file and sends the data to the corresponding processors with the help of the help fields. A send list is used to manage the communication;
   - Each block sends the relevant geometrical data at the interface to the neighbor block for the array indexing and to calculate the geometrical quantities of virtual CVs. After calculation, required information (e.g., center coordinates and surface projections in Cartesian basis of the virtual CVs) are sent back from the other side of the interface to define the auxiliary CVs. Communication (here based on MPI) takes place when the two neighbor blocks are assigned to the different processors.
   - Result output at outer iteration or time step: processors send the block(group) data to master processor, the master processor carries out the output according to the parallel block structure defined in the block - processor mapping procedure.

2. **Solving procedure**

- At each inner iteration in SIP solver: exchange the updated dependent variables between block interfaces; collect the absolute residue from all blocks and broadcast the convergence decision for inner iteration.
- At each outer iteration and time step: exchange the updated coefficients and dependent variable values between block interfaces; broadcast the pressure correction value at the reference point to all processors; collect the absolute residue from all blocks and broadcast the convergence decision.

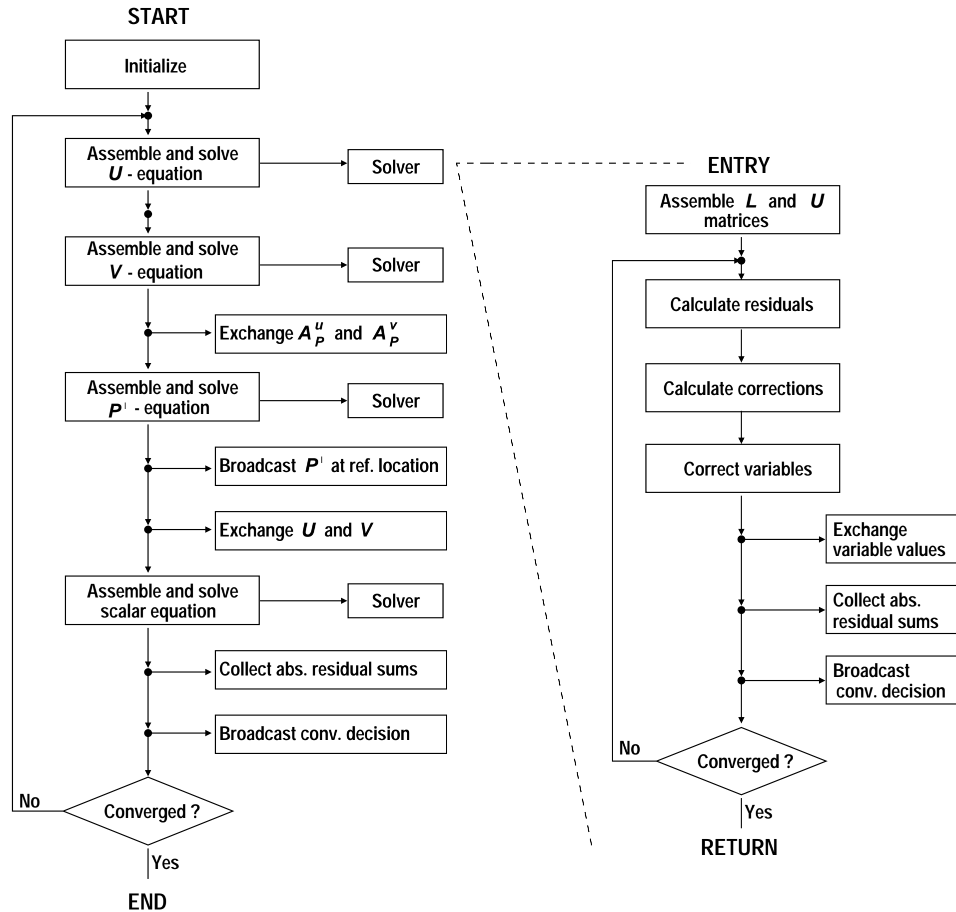


Figure 11. Flow diagram of the parallel SIMPLE and ILU algorithms

5 Summary and future improvement

Communication with MPI instead of TCG has been implemented for the multigrid solver for transport problems in complex geometries– FASTEST2D-LBR. The program was debugged and tested on the Cray T3E for several application cases. During the debug, the communication structure of the code, from the preprocessing (grid generation, grid partitioning, and setting up the connection between blocks) to the program I/O, the communication in the inner iteration and outer iteration and the control of the solving procedure were checked. The complete structure and procedure of the parallelization of this program was understood. The one-way definition for the communication in prepro-

cessing and the data structure used in communication in the present implementation was found to need improvment.

Acknowledgements

This work was supported by the Guest Student Program 2000, NIC/ZAM, which is gratefully acknowledged. Dr.-Ing C. F. Lange (Dept. of Mechanical Engineering, University of Alberta, Canada) has provided many explanations on the code in detail, which were very necessary and very helpful for the code debugging. His thesis was used as reference and several figures thereof, produced by S. Lange, are used under kind permission. Special thanks are to Dr. A. Goeke (ZAM), whose strong support and rich experience with parallel computing on Cray-T3E was a great help for the hard debug and the daily work. Both Dr. B. Steffen (ZAM) and Dr. A. Goeke have given many useful suggestions for the future improvement of the data structure in the communication and the construction of the auxiliary CVs for the data dependence. Some useful information from Dr.-Ing C. Bartels (LSTM Erlangen) and Dr.-Ing T. Weber (Computational Dynamics, Nuremberg) on the parallel structures of other FASTEST versions served as a first introduction at the beginning of the work. All these are gratefully acknowledged. Finally, I would also like to thank my supervisor Prof. Dr. Dr. h.c. F. Durst and Dr.-Ing M. Breuer at LSTM, Erlangen for the support and DAAD for the kind permission for my attendence of this program.

References

1. Lange, C. F. (1997), *Numerical predictions of heat and momentum transfer from a cylinder in crossflow with implications to hot–wire anemometry*, Ph. D. thesis, Institute of Fluid Mechanics, Friedrich–Alexander University of Erlangen–Nürnberg.
2. Flow predictor description, FASTEST Version 3.5, INVENT Computing GmbH, 3rd edition, April, 1997.
3. Khosla, P. K. and S. G. Rubin (1974), A diagonally dominant second-order accurate implicit scheme,*Comput. Fluids 2*, 207–209.
4. Ferziger, J. H. and M. Perić (1999), Computational methods for fluid dynamics, 2nd edition, Berlin: Springer.
5. Patankar, S. V. and D. B. Spalding (1972), A calculation procedure for heat, mass and momentum transfer in three dimensional parabolic flows, *Int. J. Heat Mass Transfer 15*, 1787–1806.
6. Stone, H. L. (1968), Iterative solution of implicit approximations of multi-dimensional partial differential equations, *SIAM J. Num. Anal. 5*, 530–558.
7. Durst, F. and M. Schäfer (1996), A parallel blockstructured multigrid method for the prediction of incompressible flows, *Int. J. Num. Methods Fluids 22*, 549–565.
8. Durst, F., M. Schäfer, and E. Schreck (1993), Parallelization of efficient numerical methods for flows in complex geometries, In E. H. Hirschel (Ed.), *Flow Simulation with High-Performance Computers I: DFG Priority Research Programme Results 1989-1992*, pp. 79–92. Braunschweig: Vieweg.

Installation and Test of the DRAMA library on the ZAMpano cluster

Carsten Urbach

Freie Universität Berlin,
Fachbereich Physik/ZEDV
curbach@gmx.de

Abstract: For solving large-scale problems on parallel computers one has to distribute the work uniformly to the processors, because it is very inefficient to have one processor working while others are waiting, i.e. the work load has to be balanced. Additionally for parallel problems one inevitably has communication between the processors. Therefore the communication has to be distributed uniformly on the processors as well to get a good performance.

Thus there is always the task to distribute or partition the problem between the processors such that the calculation and communication costs are minimized. In order to specify the costs for calculation and communication a so-called cost model is needed.

If the distribution has to be done only once, e.g. before the computation, it is not so important to have a good performance for the partitioning algorithms. But if the distribution has to be done several times during the calculation, as for instance in the case of adaptive mesh refinement or dynamic processes like crash test calculations, the performance of the partitioning algorithm plays a big role.

The DRAMA library is an interface for different existing libraries providing partitioning algorithms linked with a cost model for mesh based parallel applications. Mesh based means, that a complex area is discretized in (finite) elements represented by nodes.

This report is structured in the following way: After describing the theoretical basis of the finite element method and the DRAMA library the performance of the different available algorithms and the functionality of the DRAMA cost model are discussed. A finite element code was implemented to test DRAMA within an application code and first results are presented for this test.

1 Theoretical Basis

1.1 The DRAMA-Library

The DRAMA (Dynamic Re-Allocation of Meshes for parallel Finite Element Applications) library is a tool for dynamic load balancing of parallel, mesh-based applications. It originates from the DRAMA Project (Project No. 24953) funded by the European Commission within its ESPRIT Programme [1].

DRAMA is an MPI-based parallel library available as source code which provides a mesh-based interface to different partitioning algorithms. For sequential graph partitioning it uses algorithms of the Metis and the Jostle Library, for parallel graph partitioning it uses algorithms of the ParMetis and the Parallel Jostle, respectively.

The DRAMA library defines an interface which allows to specify costs for calculation and communication. Depending on these calculation and communication costs the 'optimal' repartitioned mesh is determined.

In addition it is possible to specify the calculation speed for every processor, which is useful for inhomogeneous computing environments. Furthermore the DRAMA library takes into account that application codes may contain different calculation phases with different needs for load balance. Therefore special multi-phase partitioners are provided.

1.1.1 The DRAMA Cost Model

In an abstract way it is possible to consider the mesh information (elements and/or nodes) as an undirected graph of vertices and edges, where one vertex corresponds to an element or node. If a vertex corresponds to an element, it carrys a weight $w_i^e(u)$, where i is the processor number and u is the element type; for a node k its weight is $w_i^n(k)$. Edges can carry weights c_{ij}^e, c_{ij}^n and c_{ij}^{en}, expressing dependencies (communication) between elements or between nodes or between elements and nodes.

Thus the local calculation costs have element-based and node-based contributions:

$$w_i^{tot} = \xi_e w_i^{tot,e} + \xi_n w_i^{tot,n}$$

and the communication costs have element-element, node-node and element-node based contributions:

$$c_i^{tot} = \zeta_e c_i^{tot,e} + \zeta_n c_i^{tot,n} + \zeta_{en} c_i^{tot,en}.$$

The five binary parameters ξ_e, ξ_n, ζ_e, ζ_n and ζ_{en} allow to switch between different types of data dependencies, e.g. a cost function based only on element and node calculation costs and communication between nodes would be expressed with $\{\xi_e, \xi_n, \zeta_e, \zeta_n, \zeta_{en}\} = \{1, 1, 0, 1, 0\}$.

The total costs F are given by:

$$F = \max_{i \in \{0,\ldots,p-1\}} (w_i^{tot} + c_i^{tot}),$$

where p denotes the number of processors.

So one has to specify the cost model and the parameters w_i and c_i for each processor and each phase of the application as input for DRAMA to produce a problem dependent 'optimal' distribution. For more details we refer to [2].

1.2 The Partitioners

Within the DRAMA Library different partitioning algorithms are available. Most of these algorithms belong to the ParMETIS Library, which is a MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs. Some of the algorithms belong to the Jostle and Parallel Jostle package. Furthermore a geometric partitioning module is implemented in DRAMA, that applies recursive coordinate bisection (RCB) with cost weights.

Within our project only the METIS/ParMETIS partitioners and the geometric partitioners were used, because Jostle is not a generic part of the DRAMA distribution and has

to be installed additionally. The DRAMA mesh migration module was not used, because it was still in a Beta-version. Only single phase partitioners were used, because our test problem, which will be introduced later, has only one calculation phase.

For completeness all partitioners are listed below, for more details we refer to [3] and [4].

1.2.1 Graph Partitioning

With the DRAMA routine `drama_select_graph_part_` it is possible to choose one of the available partitioning algorithms:

- Parallel single phase graph partitioner
  - ParMETIS_PartKway (PPK)
  - ParMETIS_RepartLDiffusion and ParMETIS_RepartGDifusion (LDiff/GDiff)
  - ParMETIS_RepartRemap, ParMETIS_RepartMKRemap (Remap/MKRemap)
  - Jostle parallel single phase repartitioner
- Parallel multi-phase graph partitioner
  - Moc_ParMETIS_PartKway
  - Moc_ParMETIS_SR
- Sequential partitioner
  - METIS_mCPartGraphKway and METIS_PartGraphKway (PGK)
  - Jostle sequential multi-phase partitioner

Graph Types

The meshes can be represented by a graph in different ways. Thus within DRAMA different graph types are available which can be selected with the routine `drama_select_graph_type_` [3]:

- Dual graph
  A dual graph represents a mesh in the following way: Every element becomes a vertex. Vertices are connected through edges if the elements share an 'edge' or a face.
- Combined graph
  In this graph different types of vertices occur: The first type corresponds to elements, whereas the second type corresponds to nodes. With this graph type a good representation of all types of calculation costs can be achieved.
- Extended dual graph
  The classical dual graph is extended with edges that connect vertices whose corresponding elements share one or more nodes.
- Nodal graph
  The vertices represent nodes and the edges represent the connections between the nodes.
- Generalized dual graph

1.2.2 Geometric Partitioning

The routine `drama_geometric_` partitions the distributed mesh by recursive coordinate bisectioning (RCB) with cost weights. There are two types implemented, a bucket based and a simple geometric partitioner (GeoBucket/GeoSimple).

The RCB method recursively halves the set of mesh by arranging the set along the coordinate axis for which the box bounding the points is longest and cutting orthogonal to that direction [3].

1.3 The Finite Element Method

The finite element method (FEM) is a method to solve partial differential equations (PDE), e.g. problems from structural mechanics, numerically. It follows a short introductions that does not claim completeness. For more detailed information we refer to reference [5].

1.3.1 Mathematical Background

In the following we discuss the simple example of the Poisson equation for an area G:

$$\Delta\Psi(\vec{x}) = \varphi(\vec{x}). \tag{1}$$

Here one wants to calculate Ψ for given φ and boundary conditions. For simplicity the boundary conditions are all set to zero in the derivation.

One can prove that solving the PDE (1) is equivalent to the following extremal problem:

$$\delta I = 0$$

where

$$I = \frac{1}{2}\int\limits_G \left(\left(\frac{\partial\Psi}{\partial x}\right)^2 + \left(\frac{\partial\Psi}{\partial y}\right)^2 + \left(\frac{\partial\Psi}{\partial z}\right)^2 + 2\varphi(x,y,z)\Psi(x,y,z) \right) dxdydz \tag{2}$$

One can think of two ways to solve the PDE numerically: One can approximate the geometry by splitting the area G into 'easy to handle' elements and one can make an approximative approach for the solution the PDE. The finite element method combines these two approximations as shown below.

Simplification of the Geometry

To simplify the geometry one divides the area G in discrete (finite) elements with simple geometry, e.g. squares or triangles in two dimensions or cubes in three dimensions. Within each element one makes an approximative basic approach for Ψ, usually a polynomial. Depending on the basic approach Ψ is now determined by its value at the m meshpoints $\vec{x}_i$ of the element in an unique way:

$$\Psi(\vec{x}) \to \Psi_i = \Psi(\vec{x}_i) \quad i = 1, 2, \ldots, m \tag{3}$$

Then one has to find the values Ψ_i at the nodes and one can represent Ψ as follows:

$$\Psi(\vec{x}) = \sum_{k=1}^{m} \Psi_k N_k(\vec{x}) \tag{4}$$

where the $N_k(\vec{x})$ are the shape functions that take care of the basic approach and fulfill the condition:

$$N_k(\vec{x}_j) = \begin{cases} 1 & : \quad \vec{x}_j = \vec{x}_k \\ 0 & : \quad \vec{x}_j \neq \vec{x}_k \end{cases} \qquad j, k = 1, 2, \ldots, m$$

For example with a bilinear approach in two dimensions a square finite element has four nodes and the four shape functions are given in normal coordinates ξ, η of the square as:

$$N_1(\xi,\eta) = (1-\xi)(1-\eta), \quad N_2(\xi,\eta) = \xi(1-\eta), \quad N_3(\xi,\eta) = \xi\eta, \quad N_4(\xi,\eta) = (1-\xi)\eta.$$

Ritz Method

To get an approximative solution for an extremal problem we can use the Ritz method, which makes the following approach for Ψ:

$$\Psi(\vec{x}) = \psi_0(\vec{x}) + \sum_{k=1}^{m} c_k \psi_k(\vec{x}) \tag{5}$$

with suitably chosen functions $\psi_0, \psi_1, \ldots$. Now we solve the extremal problem by minimizing with respect to the coefficients c_k. If one compares this approach with equation (4), one sees, that the representation of eq. (4) can be used as a Ritz approach; this means that the extremal problem is solved with respect to the values Ψ_i.

If one substitutes (4) into (2) it is now easy to calculate the integral for the first three terms in (2). In the rest of this report we restrict ourselves to the study of the Laplace equation $\Delta\Psi(\vec{x}) = 0$, this means we set $\varphi \equiv 0$ in the following; with this restriction the remaining integral in (2) vanishes. Because of the coordinate transformation from global coordinates $\{x, y\}$ to the normal coordinates $\{\xi, \eta\}$ of each square $dxdy$ becomes $Fd\xi d\eta$, where F is the area of the square. Since in this example only one element type is used (squares all of the same size) and all edges are parallel to the corresponding global coordinate axis F is only an overall factor. The result can be written in the following matrix equation, for a detailed derivation we refer to [5]:

$$\mathbf{S}\,\vec{\Psi} \;=\; 0 \quad \text{with} \quad \mathbf{S} = \mathbf{S_1} + \mathbf{S_3} \tag{6}$$

and

$$\mathbf{S_1} = \frac{1}{6}\begin{pmatrix} 2 & -2 & -1 & 1 \\ -2 & 2 & 1 & -1 \\ -1 & 1 & 2 & -2 \\ 1 & -1 & -2 & 2 \end{pmatrix}, \quad \mathbf{S_3} = \frac{1}{6}\begin{pmatrix} 2 & 1 & -1 & -2 \\ 1 & 2 & -2 & -1 \\ -1 & -2 & 2 & 1 \\ -2 & -1 & 1 & 2 \end{pmatrix}, \quad \vec{\Psi} = \begin{pmatrix} \Psi_1 \\ \Psi_2 \\ \Psi_3 \\ \Psi_4 \end{pmatrix}$$

With equation (6) we reduced the original problem of solving the Laplace equation to the solution of a system of linear equations, which can be calculated by means of well known algorithms.

Compilation of the global Matrix

The next step is to compile the global matrix for the area G from the matrices for all finite elements. For this one has to find a global numbering of the nodepoints, and then one extends every element matrix to the global matrix size by introducing zeros for non local entries. As a last step one has to sum over all extended element matrices. Thus the compilation of the global matrix is reduced to simply adding matrix elements depending on the connectivity between the elements.

Of course the order of nodepoints is important, because in most cases you will get only a sparse global matrix, since every node is only connected to a small number of elements and thus only to a small number of nodes. This can simplify the computation a lot if one utilizes this well. But we decided to save the time for the implementation of this, because we were primarily interested in the parallelization of the method.

1.3.2 Solving the System of linear Equations

We decided to use the iterative **Jacobi** method, because it is easy to implement:

Let $\mathbf{D}$ contain only the diagonal elements of $\mathbf{S}$ and set $\mathbf{B} := \mathbf{S} - \mathbf{D}$, so we can write:

$$\mathbf{S}\,\vec{\Psi} = 0 \quad \Leftrightarrow \quad \vec{\Psi} = -\mathbf{D}^{-1}\mathbf{B}\vec{\Psi}$$

This we can formulate as an iteration instruction:

$$\leadsto \quad \vec{\Psi}^{(i+1)} = -\mathbf{D}^{-1}\mathbf{B}\vec{\Psi}^{(i)} \tag{7}$$

1.3.3 Solving the System of linear Equations in parallel

This iterative solver is very easy to parallelize, because we can build for each processor a ‘global matrix’ in the way described above depending only on the elements which are stored locally. This ‘global matrix’ can then be used to solve the system of linear equations for the nodepoints belonging to the local processor.

In order to get the global result one has to exchange after each iteration step the values for the nodepoints lying on the boundary between two processors and take the arithmetic mean over the different values for one node, i.e. the values coming from the local iterations on the different processors.

Note that with distributing the mesh to two processors we halve the number of nodepoints on each processor, but we quarter the number of entries in the matrix! In this way the method for solving the system of linear equations is optimized naturally without too much effort for the implementation.

2 Installation and Test of the Library

2.1 Installation

For the installation on the ZAMpano only a few changes had to be made to the source code distribution of the DRAMA library. There are makefiles given with the source code for different platforms, so only one of the makefiles had to be changed to compile the

library. This makefile was added to the list of makefiles. For compilation of the code the gcc for C and the KCC for C++ were used.

During compilation there were problems within the files `drama_graph.c` and `my_malloc.c` because it is not possible to make a global initialization with a not constant object. So the line

```
FILE* out_stream = stdout;
```

leads to an error since `stdout` is no longer defined in a header file with a `#define` command, because it changed for Linux systems from the old a.out format to the new elf format.

The files were replaced with a working version. The original files were moved to `file.org`. Because it was not known in which way this object is used in the subsequent code no better solution was searched for.

The migration module, which is written in C++, could not be compiled in a standard conform way. Without this compiler option it was compiled but creating linking errors even though it was linked with the input/output routines of C++.

The next step was to make it possible to use DRAMA from C++. Therefore the makro `__cplusplus` in the header file `drama.h` was inserted to declare all routines as '`extern "C"`'.

To compile the C examples delivered with the DRAMA source it was necessary to compile without '`-DLAMP_GNU`' because this defines a '`_`' to the end of every routine name, which is neither needed nor possible, because it causes linking errors.

2.2 First Test of DRAMA

As a first test we studied the functionality and performance of the partitioning algorithms. Therefore the distribution times were measured for different element and processor numbers. Distribution time means here the wall clock time needed by the DRAMA call distributing the mesh. All measurements shown in this section were started with a random distributed mesh.

In figure 1 almost all algorithms have a nice linear scaling behaviour with increasing number of elements. Only the simple geometric partitioner (GeoSimple) shows a parabola like behaviour.

In figure 2 the distribution times for different partitioning algorithms and a problem size of 200.000 elements are given. As one expects the sequential partitioner (PGK) shows only small changes in the distribution time for different processor numbers. Additionally one can observe, that all algorithms have much communication overhead, because they all show a steep rise in wall clock time from two to four processors. For much less than 200.000 elements the effect of the communication overhead becomes important even for higher numbers of processors, which is important for the test made within an application as can be seen in the next section.

The geometric partitioner (GeoBucket) seems not to be parallelized at all because its "speedup" is bad, whereas the other parallel algorithms show a good "speedup". The

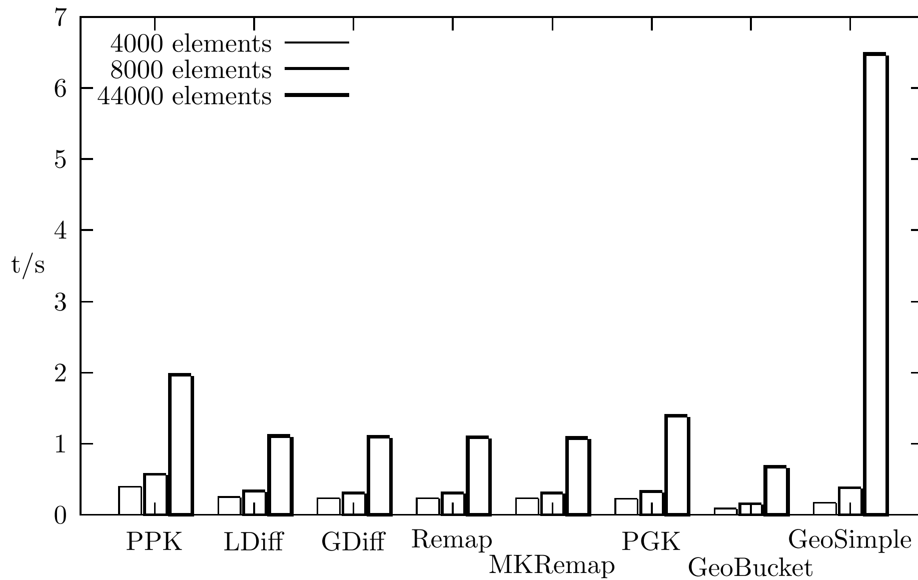


Figure 1. Distribution times for different partitioning algorithms and problem sizes

diffusion algortihm (LDiff) is always faster than the parallel Kway partitioner (PPK), and both show a better "speedup" than the others for more than 24 processors.

Note that the quality of the partitioner depends not only on the time needed for the distribution of the mesh but also on the quality of the distribution, which can be tested together with an application code.

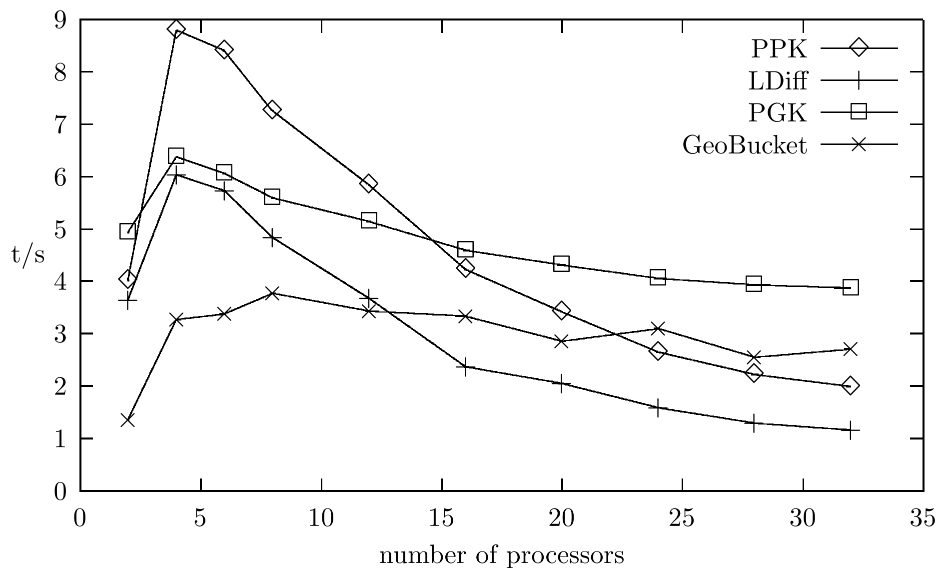


Figure 2. Distribution times for different partitioning algorithms (200.000 elements) versus the number of processors

3 Test of DRAMA within an Application

3.1 *The Application Code*

In order to test the DRAMA library within a real application code a parallel FEM code was developed that solves a system of linear equations with the Jacobi method. The algorithm of this application works as described in figure 3.

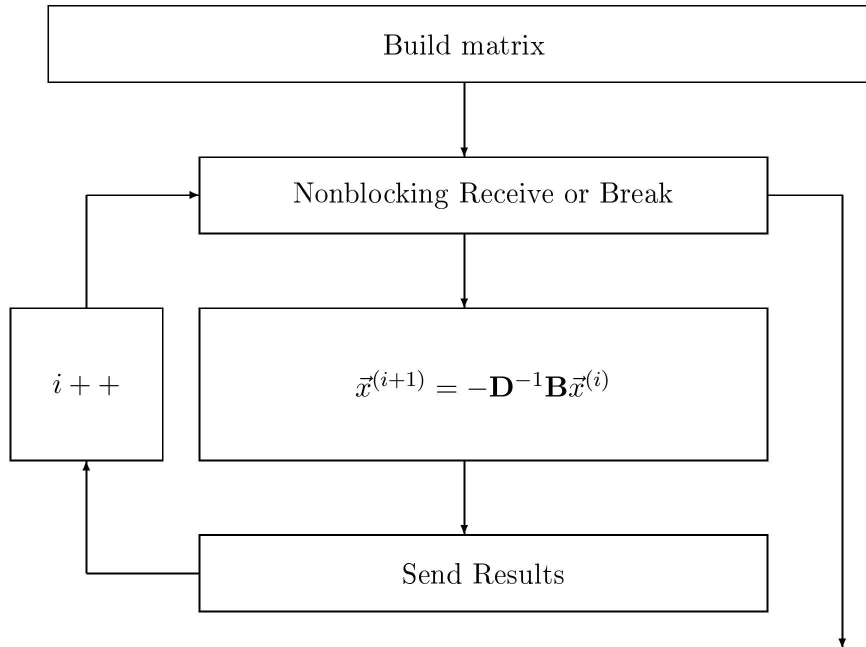


Figure 3. Block diagram for the solver kernel on one processor

This solver was used to solve the Laplace equation with Dirichlet's boundary conditions for a 2D L-shaped region G (see for instance figure 12). This region is subdivided into meshes of squares with different numbers of elements. By the variation of the number of processors between 1 and 32 it is possible to investigate the behaviour of the application for different ratios between the number of elements (per processor) and the number of boundary elements, i.e. the number of elements that share nodes with elements stored on another processor.

Speedup

First we want to discuss the speedup of the Jacobi algorithm. The results for this are plotted in figure 4. By doubling the number of processors the matrix size is not halved but quartered. Thus it is no surprise that the speedup looks like a parabola. The theoretical speedup of 1024 for 32 processors is not reached due to the communication overhead.

It is also interesting that the speedups for model sizes of 10800 and 4800 elements are identical up to 16 processors. For higher number of processors the smaller problem (4800 elements) shows the better speedup. For 12, 16, 20 and 24 processors the smallest problem

(2700 elements) shows the best speedup. The local matrix size of the problem with 10800 elements is distributed on 32 processors similar to the matrix size of the problem with 4800 elements on 16 processors and the matrix size of the problem with 2700 elements on 8 processors (see table 1). Thus our explanation for this behaviour at the present time is that this is probably a cache effect.

#P.	number of nodes n 2700 el.	4800 el.	10800 el.
2	1410.5	2480.5	5520.5
4	705.3	1240.3	2760.3
8	**352.6**	620.1	1380.1
16	176.3	**310.1**	690.1
32	88.2	155.0	**345.0**

Table 1. Mean number of nodes n per processor (matrix size n^2) for different problem sizes

For the problem size of 2700 elements the speedup is more or less constant for more than 28 processors. This happens because the matrix size per processor is already quite small. Here the communication overhead dominates over the calculation costs.

From these results one can conclude that it is not always useful to increase the number of processors as much as possible, because depending on the problem size the program will stop to gain speedup after a certain number of processors.

One might have expected that the speedup shows the opposite behaviour, i.e. the bigger problem shows the better speedup because of the better ratio between number of elements (calculation costs) and number of boundary nodes (communication costs). We gather from this that the communication plays only a minor role within our application code.

In figure 5 one can see, that if only one processor per node is used (i.e. MPI only between nodes), the code runs faster than in the case, that several processors are used on the same node (intra-node communication with MPI). If the MPI implementation used the common memory on one node, it was to expect to be vice versa. But at the present time the MPI communication within one node is going in both directions through the same network interface and therefore this communication pattern creates a bottleneck.

Convergence

In this paragraph we discuss the convergence of the Jacobi method, which is shown in figure 6. In the measurements of the time needed for the solution of the system of linear equations the iteration was stopped by the following condition for the 'rate of change' of the $d^{(i)}$:

$$d^{(i-1)} - d^{(i)} < c \quad ,$$

where the $d^{(i)}$ are defined as

$$d^{(i)} = \frac{(\vec{x}^{(i)} - \vec{x}^{(i-1)})^2}{\#nodes}$$

and the c is chosen appropriately. Only to get comparable results measuring the speedups for different partitioning algorithms the iteration was stopped in some measurements after a constant number of iterations.

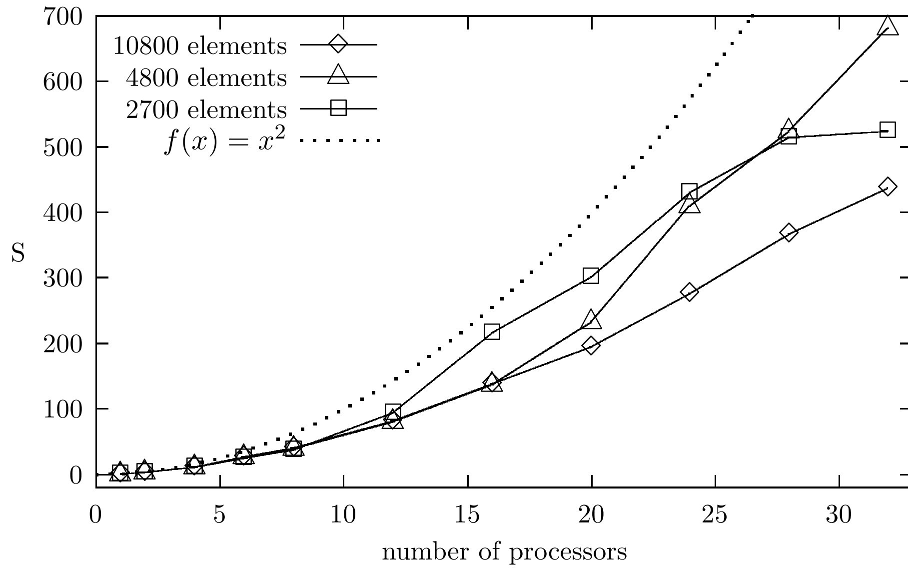


Figure 4. Speedup measured for different problem sizes versus the number of processors

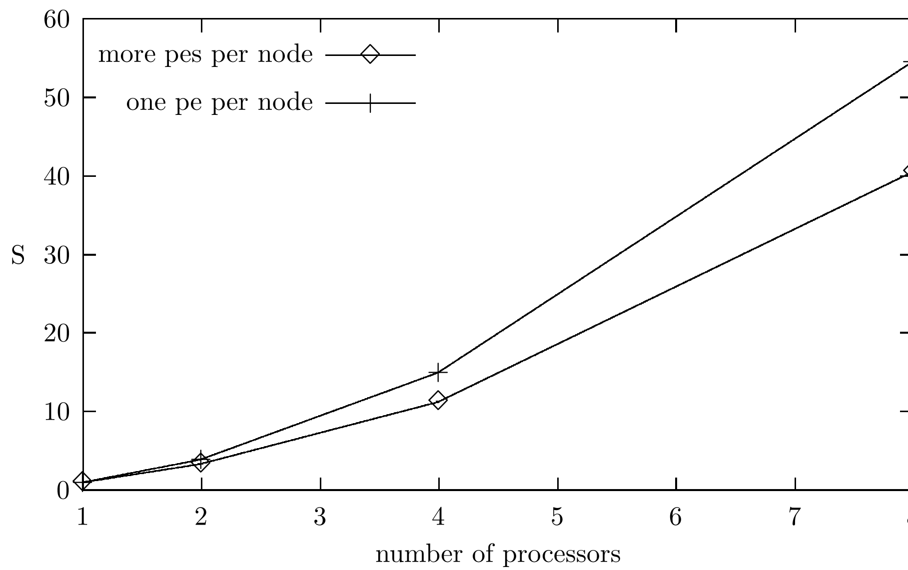


Figure 5. Comparison of the speedups from calculations with (more pes per node) and without (one pe per node) intra-node communication

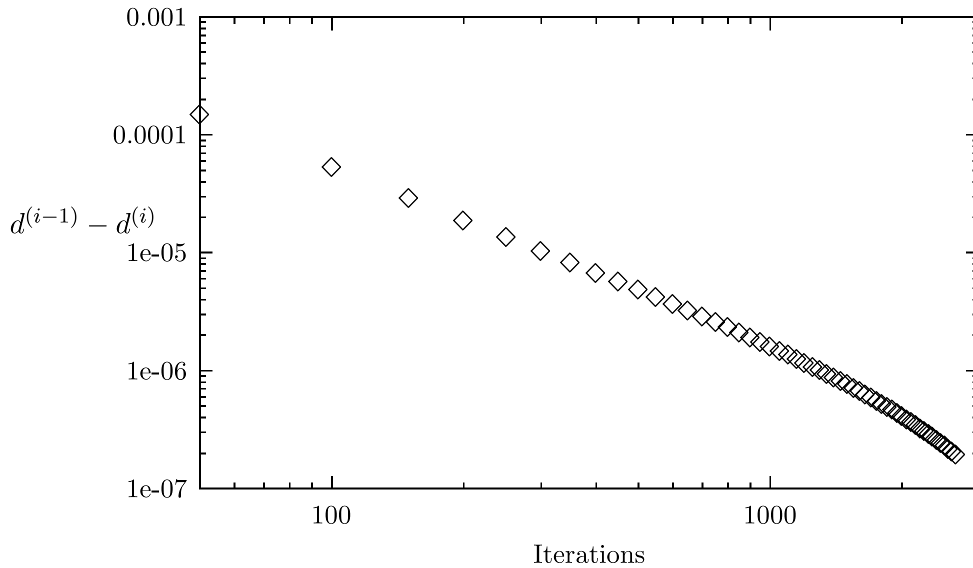


Figure 6. 'Convergence' of the Jacobi method as function of the number of iterations (logarithmic scale)

3.2 The Cost Model

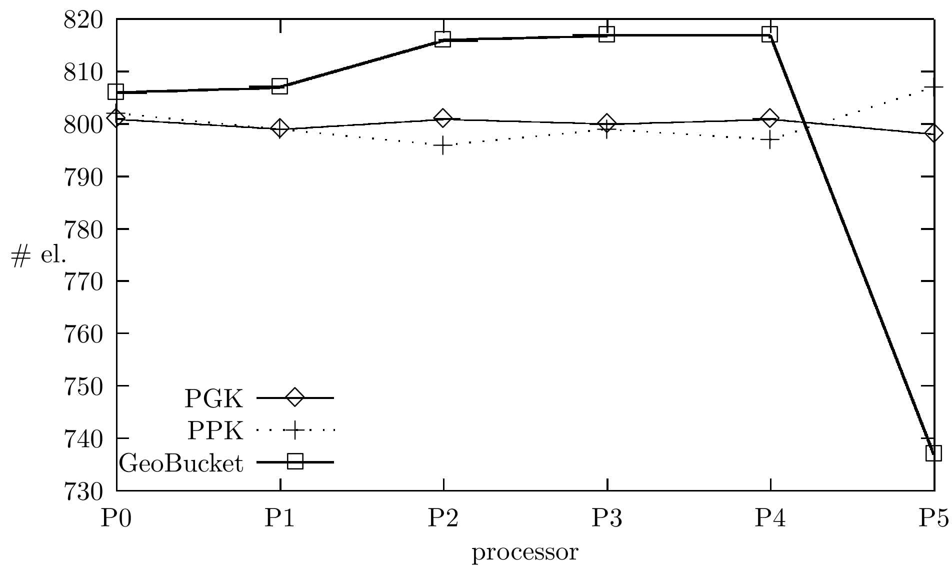


Figure 7. Distribution of 4800 elements (L-shaped area) between the processors for different partitioning algorithms

In order to investigate the cost function of the DRAMA library it is necessary to specify the cost model for our test problem. Because there is only one finite element type (squares) and one type of nodes, we decided to choose only element-element communication costs, which are controlled by the number of nodes connecting two elements.

For the calculation costs the first idea was to choose only node depending calculation costs, because each node in our problem has one degree of freedom and thus adds exactly one unknown to the system of linear equations. But within DRAMA each node is stored only

once, i.e. on one processor. This leads to an underestimation of the calculation costs for the boundary nodes (those nodes belonging to elements stored on different processors), which have to be calculated twice in every iteration step, namely on each processor. Therefore we decided to take only element depending calculation costs into account to avoid this problem.

Using these input parameters for our simple problem we were able to test successfully some of the functionality of the DRAMA cost model: By specifying (by hand) different calculation speeds for different processors, we simulated an inhomogeneous computer platform on ZAMpano; as one would expect DRAMA distributes the work load (number of elements) according to the performance of the processors, i.e. fast processors get higher numbers of elements and slower processors get less elements, respectively (see figure 12).

We have to mention that several details of the invocation of the DRAMA cost model, which are crucial for a correct behaviour, were unfortunately not included in the DRAMA documentation; thus they had to be determined in a trial and error fashion and with means of the examples delivered with DRAMA.

The best way to investigate the quality of the distribution obtained from the DRAMA partitioners is obviously the test within a real application. Nevertheless for our simple problem with only one type of finite elements the number of elements per processor and the number of boundary nodes can both be used as a first measure for the quality of the distribution.

In our test case, now under the assumption of a homogeneous computer platform, which is true for the ZAMpano, one would expect that the number of elements, as well as the number of boundary nodes are distributed equally to the processors, because this would minimize the calculation and communication costs. At the present stage our results seem to indicate that only the number of elements is distributed uniformly (see figure 7 and 8)! The reason for this behaviour is not yet understood; further tests of the DRAMA library with more complicated problems as well as a better understanding of the underlying partitioning algorithms are necessary in order to assess finally the quality of the distributions given by DRAMA.

3.3 Results for the Laplace Equation

In order to study the quality of the different partitioning algorithms the time for solving the system of linear equations was measured for different distributions. In this measurements the mesh had to be distributed from one processor to the others first. This was done in two different ways, with the bucket based geometric partitioner (GeoBucket) and with the sequential Kway partitioner (PGK). In a second step the different partitioners shown in figure 9 were applied to the two start distributions and the wall clock time for running the solver was measured. The results in figure 9 were produced with a problem size of 10800 elements on 16 processors. This measurement is shown only for 16 processors, because varying the number of processors just reproduces the relative performance of the different partitioners.

The first result to be observed in figure 9 is that the geometric pre-distribution is worse than the Kway pre-distribution (PGK) with respect to the time for running the solver. And one sees that the diffusion algorithms (LDiff/GDiff) do not change a lot of the pre-distribution, because their distributions are as bad (GeoBucket) or as good (PGK) as

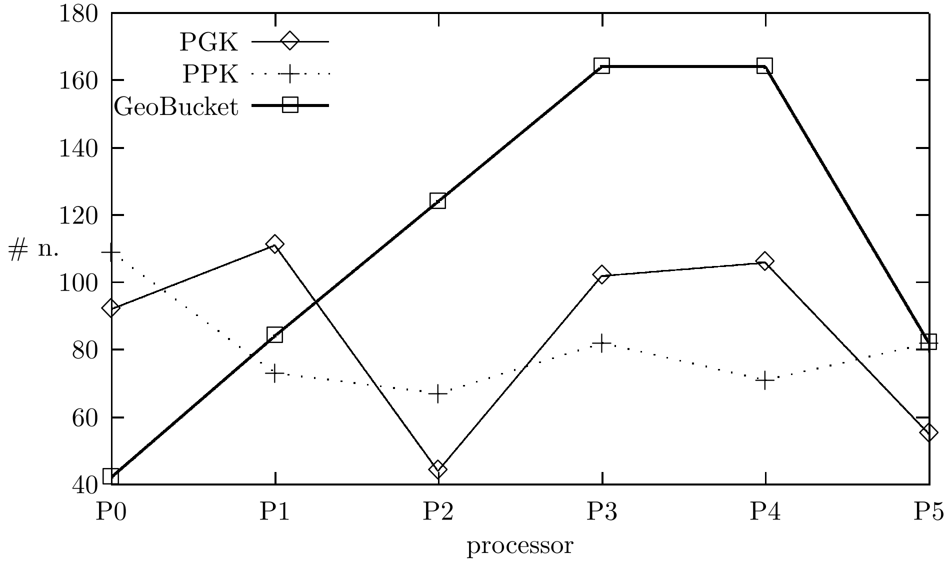


Figure 8. Distribution of boundary nodes (4800 elements, L-shaped area) between the processors for different partitioning algorithms

the pre-distribution, as can be seen as well in the figures 13 and 14 for the GeoBucket pre-distribution in figures 15 and 16 for the PGK pre-distribution.

Additionally the best overall result is achieved by the sequential partitioner (PGK), but the time for the partitioning itself is much longer than for other partitioner (see figure 2), especially for larger number of processors. The simple geometric partitioner (GeoSimple) produces also a not so good distribution (see figure 17, which indicates, that the boundary nodes are not equally distributed), but for a small number of processors the partitioner itself is very fast.

The bucket based geometric partitioner (GeoBucket) produces independent from the pre-distribution bad results, because it distributes the elements as uniformly as possible disregarding the boundary structure, i.e. it simply divides the area in strips (see figure 13). Thus there are some processors with much less boundary than others especially for the L-shaped area.

In contrast to this the remapping and the parallel Kway partitioner allow a good performance of the solver more or less independent from the pre-distribution, because they produce a more or less completely new distribution.

In figure 10 the times for the DRAMA calls are plotted in comparison with the solution times for the system of linear equations for different number of processors. One can see, that the time for the DRAMA calls reaches the order of magnitude of the time for solving the system of linear equations for a large number of processors. In this case it is probably the best strategy to take a fast partitioner that produces a maybe not not so well balanced distribution to minimize the time for running the whole program. In such a situation one can speed up the whole program even by omitting the second partitioner and using only the sequential partitioner PGK, which can be seen in figure 11, where the time for the pre-distribution with PGK is compared with the time for pre-distributing with PGK and redistributing with PPK.

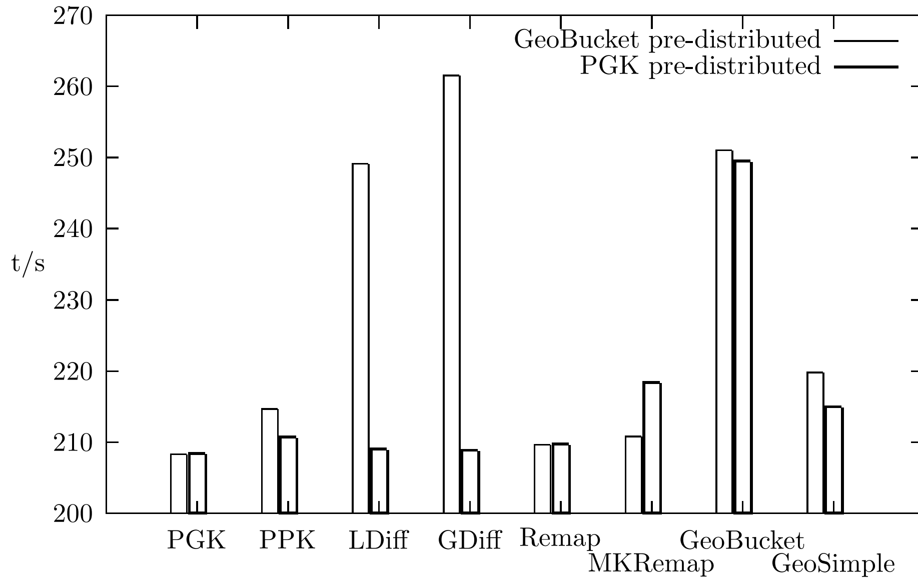


Figure 9. Comparison between wall clock times for the Jacobi Iteration starting from different distributions as described in the text

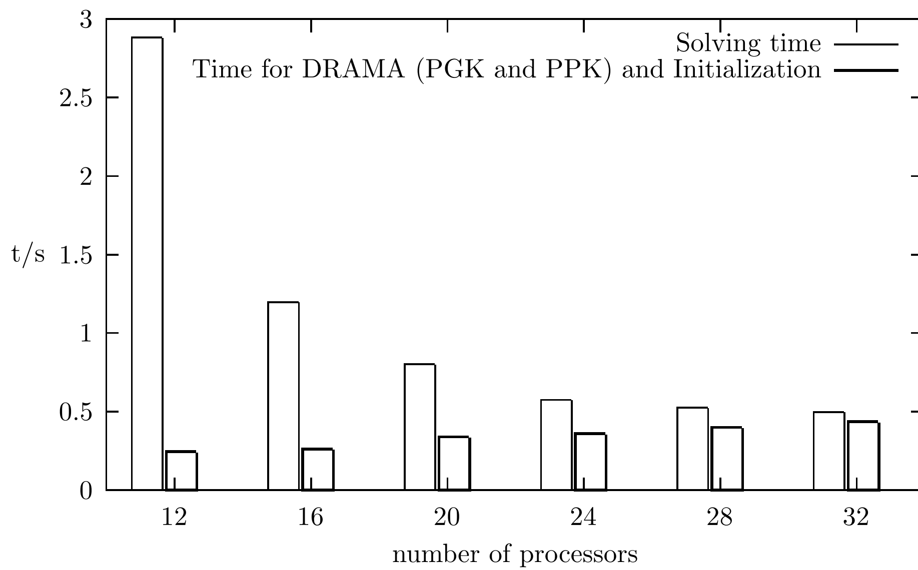


Figure 10. Comparison between wall clock times for solving the system of linear equations and the times for pre-distributing, re-distributing and initializing the mesh (2700 elements)

Moreover one can conclude from figure 11, that the first distribution with the sequential partitioner PGK will always take most of the partitioning time. And it is growing with increasing number of processors while the time for the PPK partitioner is approximately constant.

Obviously, we discuss here only relatively small sized problems and relatively small numbers of processors. It surely would be interesting to investigate the behaviour for bigger sized problems and with more processors than 32. It would be especially interesting to reach those region in figure 2 where the partitioning algorithms really benefit from increasing numbers of processors.

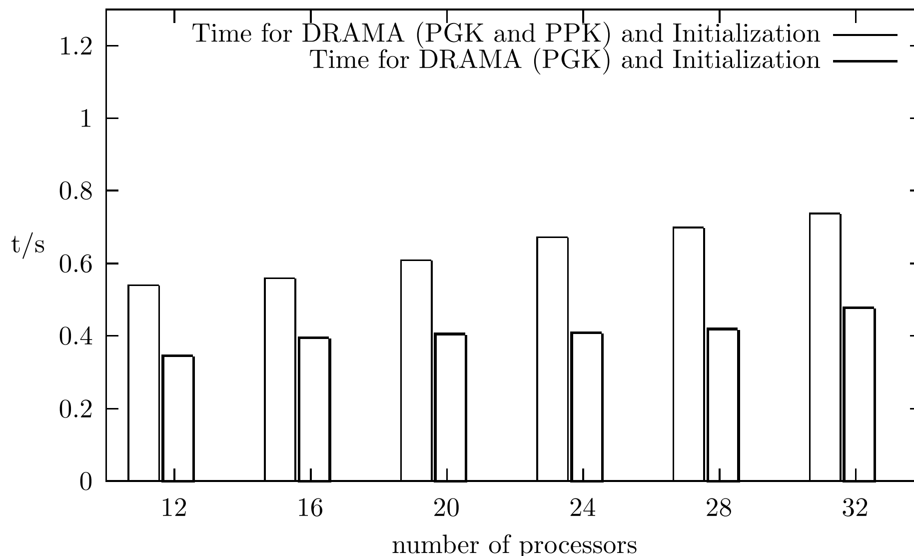


Figure 11. Comparison between the wall clock times for a two step distribution (PGK and PPK) and a one step distribution (only PGK) for a mesh of 10800 elements

4 Summary and Outlook

The DRAMA library was successfully installed on the ZAMpano cluster after some changes in the source code were made. The DRAMA library offers an interface for different partitioning algorithms linked with a cost function in order to solve mesh based problems on parallel computers. One big advantage of the library is that it provides a uniform interface for the supported partitioning algorithms.

The variety of algorithms provides a powerful tool to obtain work balance between the processors for a large bandwidth of parallel problems: There are quite fast algorithms, which produce distributions that are not optimal, algorithms that are not parallel but able to distribute the mesh from only one processor to the others and more expensive algorithms producing high quality distributions.

It was possible to utilize the cost function in order to simulate an inhomogeneous computing environment. For more insight into the functionality of the cost function a better understanding and tests with extended applications, i.e. more than one element type, more complex structures and larger problem sizes, are necessary.

Moreover an FEM code was implemented that made it possible to combine DRAMA with

an application. Thus the quality of the distributions produced with different partitioners could be estimated. Here it will be likewise advisable to test the behaviour of the partitioners together with more complex applications. Further on it might be interesting to investigate the performance of DRAMA in an adaptive refinement regime, i.e. in a situation where elements and/or nodes are added or expunged between two computation steps. This is of importance for simulations which modify the structure by itself during the computation, e.g. crash test simulations.

Acknowledgments

I want to thank my host at the ZAM, Dr. B. Körfgen, and D. Koschmieder for much helpful support and productive discussions.

5 Results visualized with RAPS

In order to visualize the geometry of the problem and the distributions between the processors an interface for RAPS was implemented. RAPS (Räumliches Aska Plot System) is a structure analysis plot system which allows to visualize FEM problems and their results. It was written by D. Koschmieder and provides many powerful tools for graphical output, e.g. 3D presentation of the structure and representation of the corresponding calculation results with shading, isolines, arrows etc..

References

1. The DRAMA Consortium
   "Updated library interface definition version 1.0 – Deliverable D1.2c, 12.11.99"
   Available via WWW under:
   `http://www.ccrl-nece.technopark.gmd.de/~drama/drama.html`
2. The DRAMA Consortium
   "Final DRAMA Cost Model version 2.0 – Deliverable D1.1b, 12.11.99"
   Available via WWW under:
   `http://www.ccrl-nece.technopark.gmd.de/~drama/drama.html`
3. T.Coupez, H.Digonnet, B.Maerten, D.Rose, A.Basermann, J.Fingberg, G.Lonsdale
   "Report on repartitioning algorithms and the DRAMA library"
   Available via WWW under:
   `http://www.ccrl-nece.technopark.gmd.de/~drama/drama.html`
4. G. Karypis, K.Schloegel, V. Kurnar
   "ParMETIS, parallel graph partitioning and sparse matrix ordering library"
   Available via WWW under:
   `http://www-users.cs.umn.edu/∼karypis/metis/parmetis/download.shtml`
5. H.R. Schwarz
   "Methode der finiten Elemente"
   Teubner Studienbücher, 1991

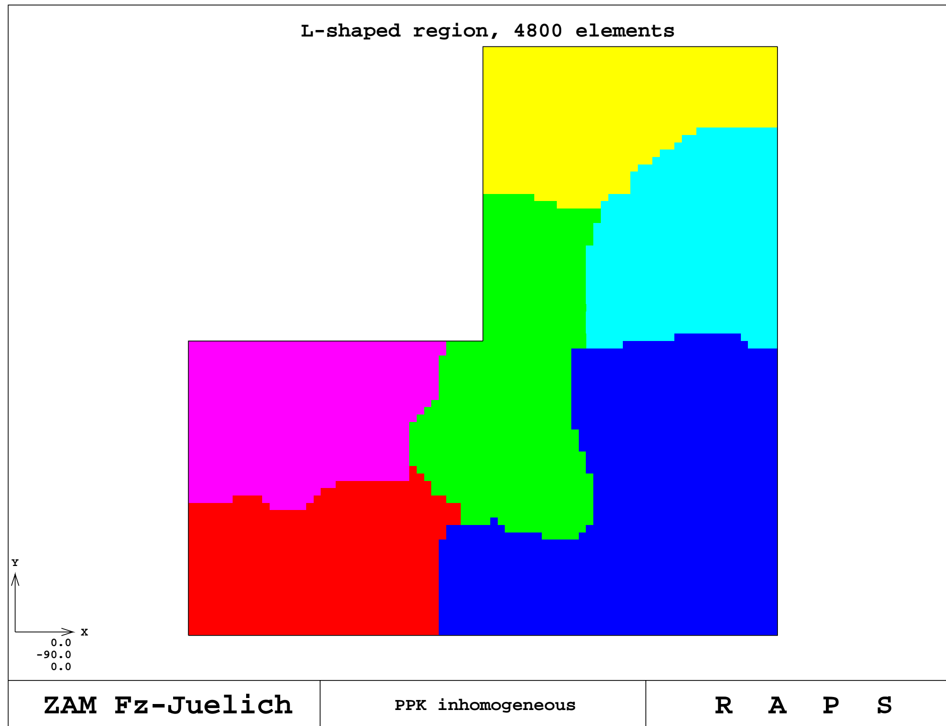


Figure 12. Distribution from PPK for an inhomogeneous computing environment with 6 processors; the processor shown in black is faster than the others

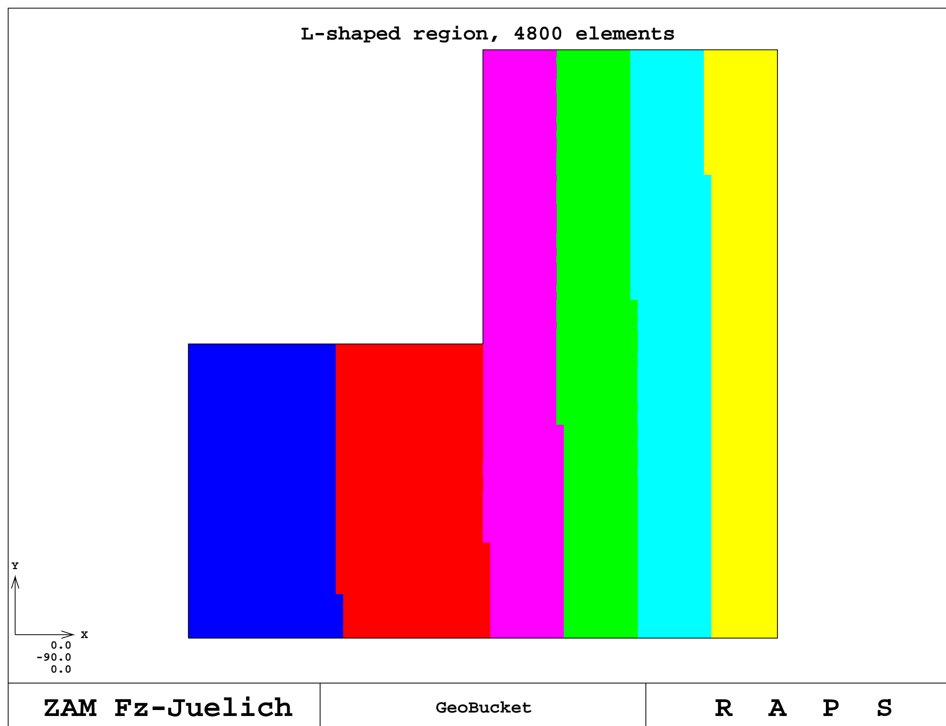


Figure 13. Distribution from the GeoBucket partitioner with 6 processors

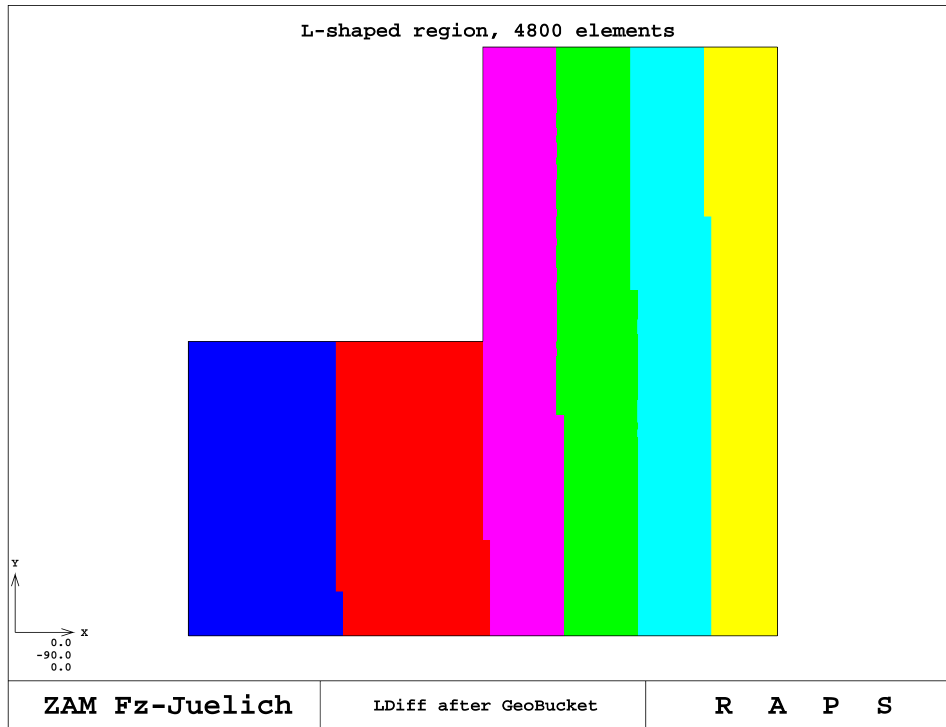


Figure 14. Distribution from the LDiff partitioner (GeoBucket pre-distributed, 6 processors)

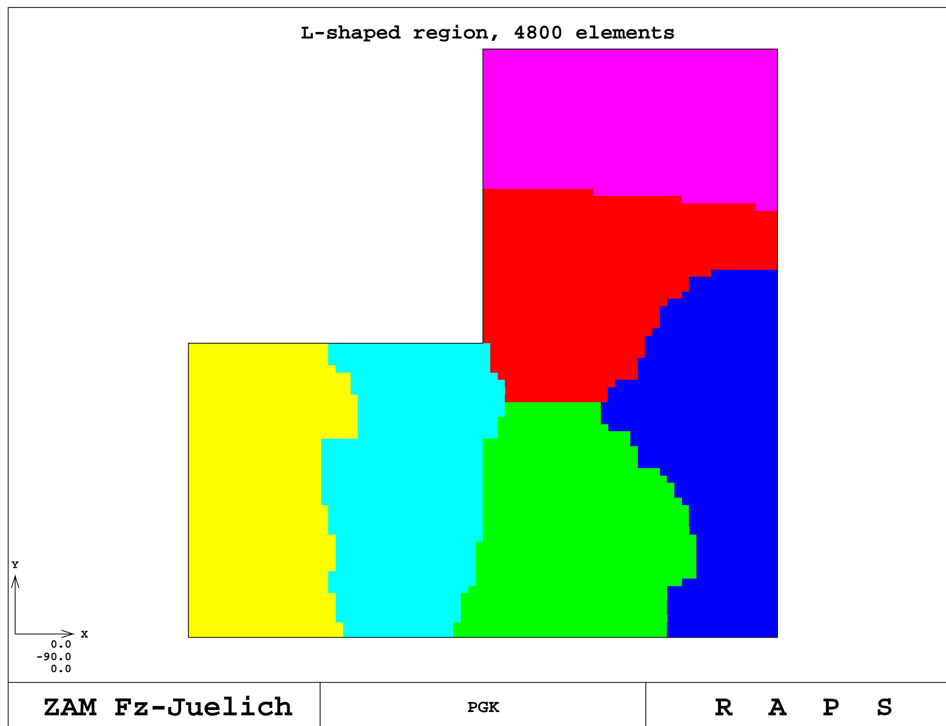


Figure 15. Distribution from the PGK partitioner with 6 processors

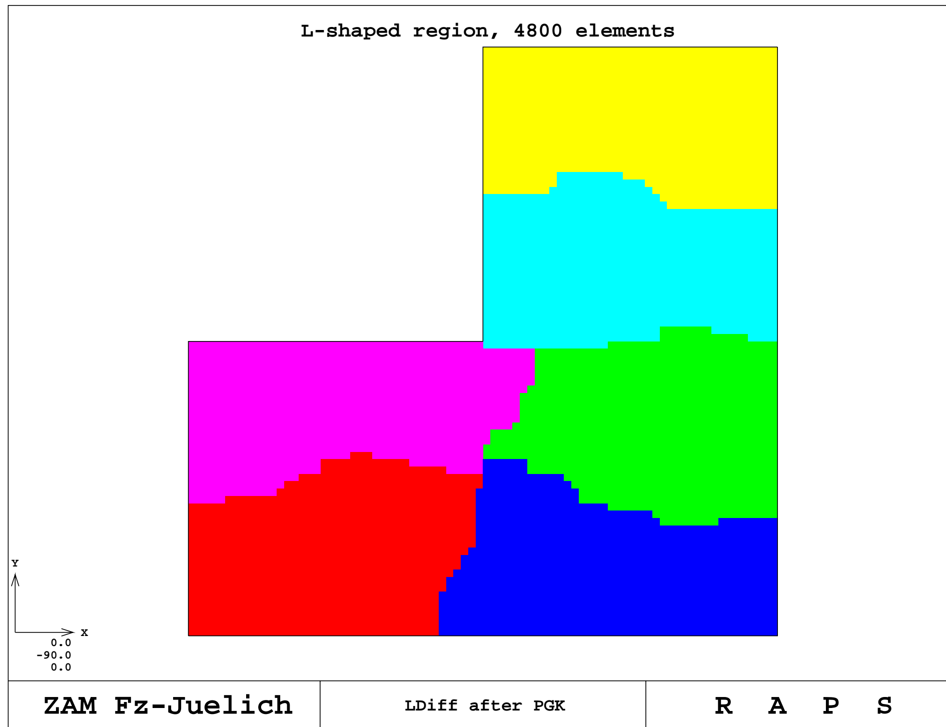


Figure 16. Distribution from the LDiff partitioner (PGK pre-distributed, 6 processors)

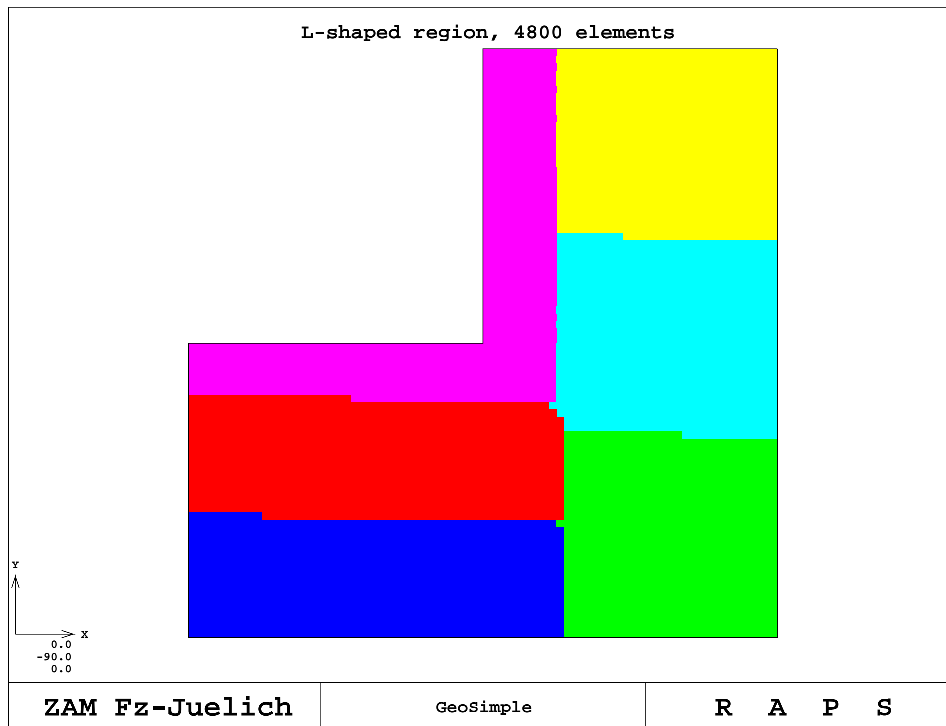


Figure 17. Distribution from the GeoSimple partitioner, 6 processors

Simulation of colloidal systems in aqueous solution

Oliver Vormoor

Georg-August-Universität Göttingen
Institut für Röntgenphysik
Email: ovormoo@gwdg.de

Abstract: A software for simulating the dynamical behavior of a colloidal system in aqueous solution is presented. Target application of this software is the precipitation process of iron in water when adding an electrolyte to the suspension. While the transport process of the electrolyte is described by a continuous density function solving the diffusion equation, molecular dynamics techniques are used for modeling the colloidal particles. Both processes are coupled by electrostatic interaction, which results in a large computational effort. We discuss the simulation model in detail with the numerical approximations and introduce some parallelization strategies for the software. Finally some benchmark results are presented, where it is also shown that for small densities of colloidal particles load balancing between the processing elements is necessary.

1 Introduction

Colloidal systems are important for many practical applications. We are speaking of a system of colloidal particles, if a characteristical length of the particles is in the range from $\approx 10nm$ up to $\approx 1\mu m$. For particles with these dimensions the fraction of the surface atoms plays a more important role than in solid physics. This results in many interesting effects, which can be found e. g. in [1].

Target application are precipitation and coagulation processes of hematite in aqueous media. Hematite is essentially colloidal ferric oxide, the particles to be simulated have a diameter of $\approx 100nm$. Due to protonization the hematite particles have a small electric surface charge and a repulsive electrostatic interaction. In addition the particles interact via van der Waals-forces and a hard-core repulsion, which is weaker than the electrostatic interaction except for very small distances.

When adding a salt to a colloidal suspension interaction between the colloidal particles is changed. The ions do not distribute uniformly over the complete volume as expected for a system of pure water, they interact with the electrostatic potential built up by the colloidal particles. Correspondingly the electrostatic potential of the colloids is screened by the ions.

Derjaguin, Landau, Verwey and Overbeek (see e. g. [1]) developed a theory for the equilibrium state describing the interaction of colloidal particles for different ion concentrations. It can be shown that above a certain ion concentration, the *critical coagulation concentration (c.c.c.)*, the electrostatic potential is screened below the van der Waals interaction of the colloids, which results in a pure attractive interaction between the colloids (see Fig. 1). Thus coagulation occurs in the dispersion, i. e. clusters of colloidal particles

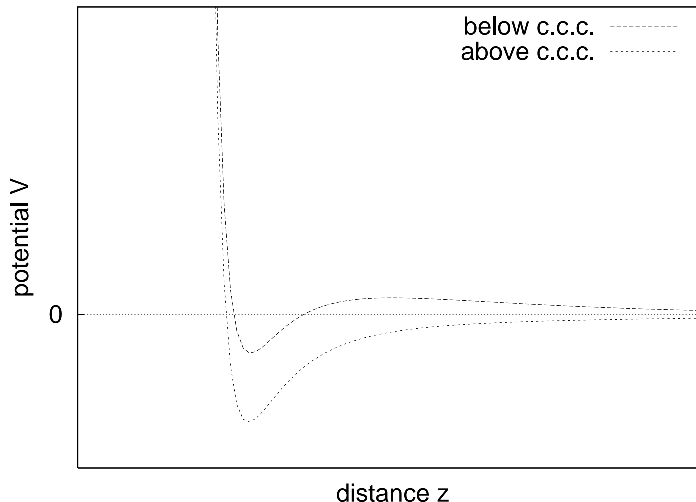


Figure 1. Interaction between colloidal particles depending on the concentrations of ions in their surrounding

are formed. This theory has been verified experimentally by a large number of experiments.

Main interest lays on the structure of the clusters near the critical coagulation concentration. The concept of fractal dimension is used to characterize the system. Clusters with a high fractal dimension d are rather compact, while clusters with a lower fractal dimension have a more open structure. Scattering experiments suggest a decrease of the fractal dimension when the ion concentration c_{ion} crosses the *c.c.c.*, i. e. clusters at $c_{\text{ion}} < c.c.c.$ have a rather dense structure, while clusters at $c_{\text{ion}}D > c.c.c.$ have a more open structure. Direct imaging using X-ray microscopy techniques shows a completely different process: just below *c.c.c.* the clusters have a smaller fractal dimension than above *c.c.c.* Both methods have several difficulties to measure the fractal dimension, that can introduce some errors during the measurement. Various aggregation models have been introduced to describe these phenomena, but they all lack different details mostly including reorganization of clusters.

Main target of this project is to simulate a complete system of colloidal particles in aqueous solution including the ion transport process, which is in a first approach modeled as a diffusion process. The electrostatic interaction coupling the dynamics of the colloidal particles and the diffusion process requires a large computational effort. Thus a parallel implementation of the simulation software is recommended, which is the topic of the current article.

The software itself is implemented using ANSI-C. No design aspects or implementation details will be given here, they will lead beyond this article and will be made available in the near future in a separate technical report.

2 Physics and Simulation model

2.1 Dynamics of colloidal particles

Starting with the description of the dynamics of the colloidal particles we summarize the forces acting on one colloidal particle. There are several types of interactions. The most

important of them are the following:

1. electrostatic interaction and van der Waals-force between all colloidal particles,
2. electrostatic interaction between the colloids and the ions of the salt and
3. interaction with the solvent.

Due to protonization in aqueous media the colloidal particles have an electric charge. Since the complete system in non electric, this electric charged is already screened, even if no salt has been added to the suspension. Thus we apply a Debey-Hückel type potential by multiplying the bare Coulomb potential with an exponential factor [2]:

$$V_{\mathrm{el}}(r) = \frac{1}{\epsilon}\frac{q_i q_j}{r} e^{-r/r_0} \tag{1}$$

where $r = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between the particles i and j. ϵ is the dielectric constant including the solvent and depends on the chosen units. In general, ϵ will include nonlocal properties of the medium an will be a function of the distance r. However, in a first approximation we keep ϵ as a fixed parameter. Finally r_0 is a screening radius. Particles with a distance $r \gg r_0$ do not give any contribution to the electrostatic potential, and one can truncate the interaction beyond a certain distance R.

When evaluating the interaction between colloidal particles and their surrounding ion environment, q_j has to be replaced by a charge density ρ, which is integrated over the interaction volume V

$$V_{\mathrm{ion}}(\mathbf{r}_i) = \frac{q_i}{\epsilon} \int_V \frac{\rho(\mathbf{r}')}{|\mathbf{r}_i - \mathbf{r}'|} e^{-r/r_0} d^3 r' \tag{2}$$

In addition to the electrostatic interaction we have van der Waals interaction between the colloidal particles. The colloidal particles consist of many atoms, so we have to distinguish three different approximations for distance r between the surfaces of two colloids with radius R

1. For very small distances $r \ll R$ the potential can be approximated by [1]

$$V_{\mathrm{vdW}}(r) \sim -\frac{1}{r} \tag{3}$$

2. If the surface distance is of the order of magnitude of R, i. e. $r = O(R)$, we get the following power law:

$$V_{\mathrm{vdW}}(r) \sim -\frac{1}{r^3} \tag{4}$$

3. For very large distances $r \gg R$, the distance between all induced dipoles is the same and we get the normal van der Waals-law

$$V_{\mathrm{vdW}}(r) \sim \frac{1}{r^6} \tag{5}$$

Due to simplicity the third approximation is used for all distances keeping in mind, that for small distances corrections may be necessary. This allows us to use the Lennard-Jones potential [3]:

$$V_{\mathrm{LJ}}(r) = 4\epsilon' \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{6}$$

ϵ' is the depth of the Lennard-Jones well and σ is called the Lennard-Jones radius, i. e.

$$\min_{r>0} V_{\mathrm{LJ}}(r) = -\epsilon' \quad \text{and} \quad V_{\mathrm{LJ}}(\sigma) = 0. \tag{7}$$

The potential energy of one particle is summarized in the following expression ($\mathbf{r}_i$ is the position of the particle i and $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ the distance between particles i and j):

$$U_i(\mathbf{r}_i) = \underbrace{\sum_j \left\{ 4\epsilon' \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] + \frac{1}{\epsilon} \frac{q_i q_j}{r_{ij}} e^{-r_{\mathrm{ij}}/r_0} \right\}}_{\text{interaction between colloids}} + \underbrace{\frac{q_i}{\epsilon} \int_V \frac{\rho(\mathbf{r}')}{|\mathbf{r}_i - \mathbf{r}'|} e^{-|\mathbf{r}_i - \mathbf{r}'|/r_0} d^3 r'}_{\text{colloid-ion-interaction}} \tag{8}$$

The interaction with the solvent is not modeled explicitly, a continuum description with a friction force $\mathbf{F}_d = -\xi \mathbf{v}$ and stochastic dynamics are used. A random force $\mathbf{A}(t)$ is added to the equation of motion and we get the *Langevin-equation* for the total force acting on a particle [2]:

$$m\ddot{\mathbf{r}}_i = -\mathrm{grad} U_i(\mathbf{r}_i) - \xi \dot{\mathbf{r}}_i + \mathbf{A}_i(t) \tag{9}$$

$\mathbf{A}_i(t)$ is assumed to have a Gaussian distribution with average = 0 and δ-correlation:

$$\overline{\mathbf{A}_i(t)} = \mathbf{0} \tag{10}$$

$$\overline{\langle A_i(t), \mathbf{A}_j(t') \rangle} = -6\gamma k_{\mathrm{B}} T \delta(t - t') \delta_{ij}, \tag{11}$$

where $\gamma = \frac{\xi}{m}$ is the friction constant ($\langle \cdot, \cdot \rangle$ denotes the inner product, $\overline{X}$ the average of X).

The modeled system of colloidal particles is a canonical ensemble, i. e. a system with constant temperature. The aqueous media is a heat reservoir, to which our colloidal system is adapted. Since do not consider the solvent molecules explicitly, one has to take care that the temperature remains constant during the simulation process [3].

With the summarized energy from Eq. (8) and the Langevin equation (Eq. (9)) we are now able to select our integrator. The *Velocity-Verlet-Algorithm* [3] is used to propagate the system:

$$\mathbf{r}(t + \delta t) = \mathbf{r}(t) + \mathbf{v}(t)\delta t + \frac{\delta t^2}{2m} \mathbf{F}(t) \tag{12}$$

$$\mathbf{v}(t + \delta t) = \mathbf{v}(t) + \frac{\delta t}{2m} [\mathbf{F}(t) + \mathbf{F}(t + \delta t)] \tag{13}$$

with $\mathbf{F}(t) = -\mathrm{grad} U_i(\mathbf{r}_i) - \xi \dot{\mathbf{r}}_i + \mathbf{A}_i(t)$ from Eq. (9).

Van der Waals and electrostatic interaction are short ranged forces in this simulation model, so the *linked-cell method* can be used to increase simulation speed significantly. Section 3 will show that the cells are also very useful for load balancing strategies on parallel computers.

No periodic boundary conditions are used, because the simulated volume is of the approximate size of a realistic volume in a X-ray microscopy chamber.

While the motion of the colloidal particles is modeled explicitly using molecular dynamics techniques, the diffusion processes is described in a continuum approach, i. e. every point in space is assigned an ion density $c_\pm$. The subscript $\pm$ is necessary, because an salt always has two types of ions, positive and negative. Diffusion in empty space is usually described by the diffusion equation

$$\frac{\partial c_\pm}{\partial t} = D_\pm \Delta c_\pm, \tag{14}$$

where $D_\pm$ is the diffusion constant. If an external force $\mathbf{F}$ acts on the ions, the right hand side of Eq. (14) gets another term, which transforms Eq. (14) to the *Smoluchowski equation* [4]:

$$\frac{\partial c_\pm}{\partial t} = D_\pm \Delta c_\pm - \mu_\pm \mathrm{div}(c_\pm \mathbf{F}) \tag{15}$$

where $\mu_\pm$ is the mobility of the ions.

In our case the force $\mathbf{F}$ is the electrostatic interaction of colloids and ions. The colloids build up a screened electrostatic potential Φ (electro dynamical aspects are completely neglected here):

$$\Phi(\mathbf{r}) = \sum_i \frac{q_i}{|\mathbf{r} - \mathbf{r}_i|} e^{-|\mathbf{r}-\mathbf{r}_i|/r_0} \tag{16}$$

Using Eq. (16) in Eq. (15) with $\mathbf{F} = -q_\pm \mathrm{grad}\Phi$ (q is the charge of the ions) yields:

$$\frac{\partial c_\pm}{\partial t} = D_\pm \Delta c_\pm + \mu_\pm q_\pm (\langle \mathrm{grad} c_\pm, \mathrm{grad}\Phi \rangle + c_\pm \Delta \Phi) \tag{17}$$

This partial differential equation has to be transformed into a finite difference equation.

Using a finite difference grid with a fixed grid constant g^{-1} for the ion densities $c_\pm$ and the electrostatic potentialΦ one obtains the following finite difference equation:

$$\begin{aligned} c_\pm(\mathbf{x}, t+\delta t) &= s_\pm \cdot c_\pm(\mathbf{x}, t) + r_\pm \sum_{\mathbf{j}\in\mathrm{NN}} c_\pm(\mathbf{j}, t) \\ &+ u_\pm \sum_{i=x,y,z} [c_\pm(\mathbf{x}+g\mathbf{e}_i) - c_\pm(\mathbf{x}-g\mathbf{e}_i)] \cdot [\Phi(\mathbf{x}+g\mathbf{e}_i) - \Phi(\mathbf{x}-g\mathbf{e}_i)] \\ &- \frac{\mu_\pm}{\epsilon} c_\pm [q_+ c_+ + q_- c_-] \end{aligned} \tag{18}$$

Derivation of Eq. (19) is shown in Appendix A. The first summation over $\mathbf{j}$ includes all nearest neighbours (NN), while the second summation iterates over the different directions in space with the canonical unit vectors $\mathbf{e}_x$, $\mathbf{e}_y$, $\mathbf{e}_z$, respectively. The parameters $s_\pm$, $r_\pm$ and $u_\pm$ are derived from the diffusion constant $D_\pm$, length δt of one time step, size of one grid cell g and mobility $\mu_\pm$ as:

$$s_\pm = 1 - 6r_\pm, \qquad r_\pm = \frac{D_\pm \delta t}{g^2} \quad \text{and} \quad u = \frac{\mu_\pm q_\pm}{4g^2} \tag{19}$$

Care has to be taken for the boundary conditions. Due to the colloids in the system, several grid points are occupied by colloidal particles and the density $c_\pm$ must vanish at

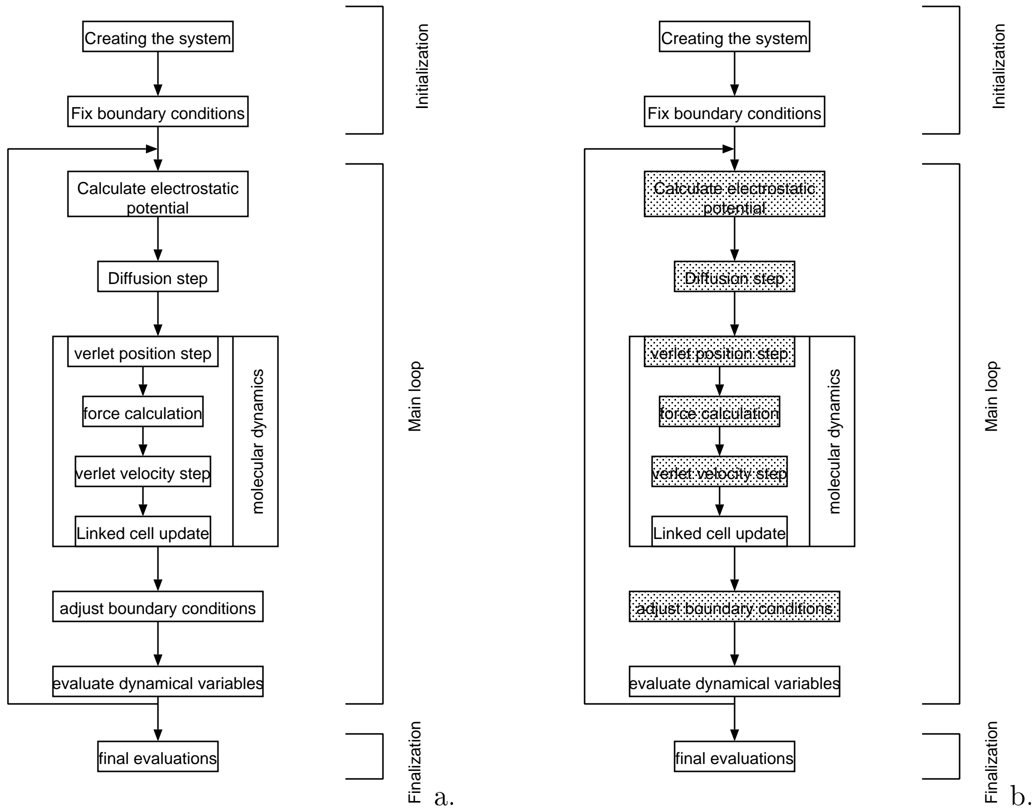


Figure 2. Flow diagram of the simulation process, a. sequential version, b. parallelized version using OpenMP, where underlayed steps are work sharing steps.

these points. So we have a system with rather complex boundary conditions. Furthermore the colloids are moving and the boundary conditions are changing during the simulation. Therefore after each molecular dynamics step boundary conditions for the diffusion have to be adjusted.

The decision for a continuum approach for ion transport process is based on the computer effort for the simulation. The colloidal particles simulated here have a diameter of $\approx 100nm$, pure molecular dynamical simulations are able to simulate a cube with a side length of about $10nm$, which is several orders of magnitude too small. This approach should reach a simulation volume of about $5-6\mu m$ side length.

2.3 Simulation process

After having outlined the basic principles we construct the complete simulation process as shown in Fig. 2a. The initialization and finalization processes are not very interesting and shown just for the figure to be complete. Each time step starts with calculation of the electrostatic potential Φ from Eq. (16) followed by the diffusion step, Eq. (19). Then the molecular dynamical step is performed using Eqs. (8) and (9) for force calculation and Verlet step Eq. (12). If preferred evaluation of dynamical variables may be included after every step and after the complete simulation.

All steps of the main-loop in Fig. 2a. are loops themselfs. For the potential calculation an iteration over all colloidal particles is performed, while the diffusion iterates over all grid points in the volume. Force calculation, Verlet steps and linked-cell update iterate over

all colloidal particles.

Performance tests show, that most CPU time is needed for the interaction between colloids and ions, i.e. for calculation of electrostatic potential and force acting from the ions on the colloids. Customary cut off-radii require an iteration over $\approx 10^5$ grid points for force / electrostatic potential calculation of one colloidal particle.

Currently simple diffusion is used for the ion transport through the system. One can think of other transport processes, e. g. hydrodynamics coupled with diffusion. This means solving the Navier-Stokes equation.

The software is designed in a modular way so that the selected transport process or the interaction between colloidal particles may be easily exchanged.

3 Parallelization strategies

With knowledge of the previous section we are able to develop some parallelization strategies for this simulation software. We start with a short outline of a shared-memory parallelization using the OpenMP standard. This is followed by a discussion of a general parallelization strategy for a message-passing system. The last parts of this section introduce a uniform domain decomposition and two load balancing strategies.

For finite difference grids normally a spatial decomposition of simulation data for parallelization on distributed-memory systems is used with the need of exchanging boundaries among the processing elements (PE's). Since we use finite difference grids for representation of $c_\pm$ and Φ a spatial decomposition of these grids is recommended. Thus data of the colloidal particles must distributed among the processing elements (PE's) using a spatial decomposition, too, because the colloid interact with their environment.

A new problem arises using a spatial decomposition. The needed CPU-amount during one time step shows the following properties:

- The CPU amount needed for calculation of the electrostatic potential scales with $T \sim N$ where N is the local number of colloids staying in the domain of one PE; for each colloidal particle the electrostatic potential in the environment has to be calculated.
- For the diffusion step, CPU load is equal for the total volume V and scales directly with $T \sim V$
- Force calculation and molecular dynamics step need a CPU time that grows also directly with N: $T \sim N$.

For low densities and large number of PE's heavy load imbalance can occur if the number of colloids manages be one PE varies. Therefore two load balancing strategies are introduced in the last part of this section.

3.1 Sharing the work using OpenMP

The OpenMP-standard enables an easy and fast way of parallelization on shared-memory systems and does not force the programmer to change his programming style. Fig. 2b

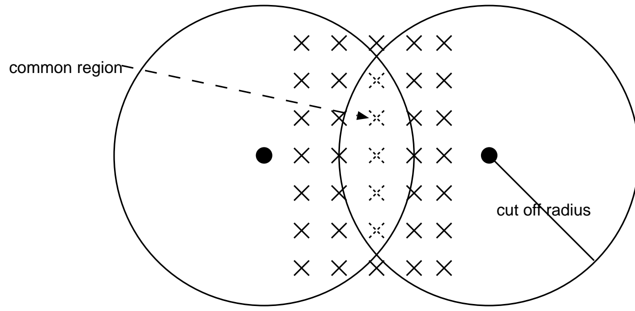


Figure 3. Common region where two colloidal particles build up an electrostatic potential

shows the flow diagram of the OpenMP-version of the simulation program. Nearly all steps in the main loop share the work completely among available processors and are underlayed in the figure except the linked-cell update. The update is not very CPU intensive. Locking and unlocking the cells would need more CPU time than the update itself.

At force calculation, Verlet-steps and adjustment of boundary conditions work sharing is performed in the loop over all colloidal particles, the diffusion step splits up the loop iterating over all grid points of the volume. A little bit more effort has to be spent for the calculation of the electrostatic potential. It can occur that two colloids have a common region, where the electrostatic potential has to be calculated - see Fig. 3. So work sharing has to be done in the loop for calculating the electrostatic potential of one colloidal particle within the cut off-radius.

3.2 First approach with a uniform spatial decomposition

In a first approach a strategy using a uniform spatial decomposition has been implemented and tested. It will be discussed just shortly.

The volume is split up to n_p blocks (n_p is the number of processing elements) of equal size. Since the molecular dynamics uses the linked-cell method, our finest granularity for spatial decomposition is strongly recommended to be the sizes of the cells. The n_p blocks therefore have sizes that are integer multiples of the used cut off radius, since size of the cells is chosen as the cut off radius.

The molecular dynamics implemented here is very similar to the SD-approach of the DMMD software [5]. Each PE manages the colloids in its local cells and needs MD-data of the neighbour cells that have to be updated after every time step. A lot of book-keeping work has to be done, since the number of colloids per cell are varying during the simulation. Details about this book-keeping will lead beyond of this summary.

In contrast to the rather complex molecular dynamics the distribution of finite difference grids for electrostatic potentials and ion densities are quite simple. The derivation of electrostatic potential and diffusion step is a nearest neighbour-problem (see Eq. (19)), i. e. every local grid object needs a hull of one grid point thickness. This hull has to be updated after every potential calculation and diffusion step in to order keep the consistency of the simulation data. This is a very well known problem in parallel computing, further discussion can be found e. g. in [6].

Since the colloids are interacting with a larger environment, i. a. a ball with a radius e. g. 32 grid points, some care has to be taken, if a neighbour particle interacts with the local ion

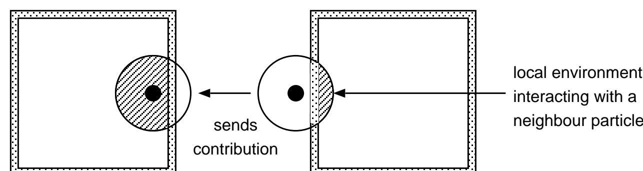


Figure 4. Calculating the interaction of the local ion environment with a neighbour particle

environment. The potential calculation is not a problem: Keeping in mind that we already have the data of all neighbour particles on the local PE from the molecular dynamics the iteration for calculating the electrostatic potential for a single colloid is performed over local and neighbour particles. The calculation just has to take care about the boundaries, i.e. calculation is canceled beyond the boundaries of the local grid. The situation is a little bit more complex during the force calculation - see Fig. 4: The environment interacting with the colloid is distributed among two or more PE's. Especially the local ions of the PE on the right hand side are acting with a neighbour particle. The PE on the left hand side has to be notified about the force acting its local particle. The implemented solution of this problem calculates the contribution of the local environment to the force acting on the neighbour particle and sends this contribution to the PE that manages this particle. That PE receives this neighbour contribution (from its point of view) and adds it to its force acting from its local environment.

3.3 Load balancing with Unbalanced Recursive Bisectioning

The implementation of a uniform spatial decomposition gives rather good efficiencies for colloid densities with a volume fraction $\rho \geq 0.05$ as it is discussed later. For lower colloid densities (and these are preferred experimental densities) the efficiency decreases rapidly, because the CPU load is not partitioned uniformly among all PE's as mentioned before. In this part two dynamic load balancing strategies are introduced, which are both based on the *Unbalanced Recursive Bisectioning* (URB) [7].

The URB-algorithm splits up a volume into blocks trying to reach an equal load for every block. The load is usually calculated by a cost function. As mentioned before the finest granularity for our problem is a cell of the linked-cells, the algorithm should try to partition these cells among the PE's with equal load - see Fig. 5 for the two dimensional case. The three dimensional case works analogous. Cost function is simply the number of colloids in one cell, which allows us to reach an equal load for potential and force calculation as well as for the Verlet-step.

Fig. 5 shows the partition achieved for the system configuration at a certain time. Colloids are moving through the volume and therefore local cell load changes during simulation. An adjustment of the decomposition scheme is necessary. Criteria for this update are discussed later.

With this adjusted spatial decomposition we encounter the problem how to handle the finite difference grids. Because the decomposition scheme changes during the simulation, the domain managed by a certain PE changes during the simulation, too. Additionally load imbalance occurs now during a diffusion step, if the domains have different sizes as shown in Fig. 5.

For a good load balance we are forced to handle two different decompositions of our

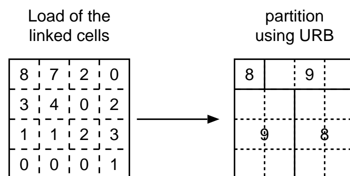


Figure 5. Domain decomposition using URB algorithm

simulation data:

- A uniform domain decomposition (*UDD*) for the diffusion step and
- a spatial decomposition for the particles dynamically, which is adjusted to the local particle density (*ASD* = adjusted spatial decomposition) for potential and force calculation.

So we are in the need of switching between two decompositions during a simulation step. A time step contains now the following components:

1. Calculate the electrostatic potential using the adjusted spatial decomposition.
2. Copy parts of the finite difference grid for electrostatic potential if necessary (ASD $\rightarrow$ UDD - see Fig. 6).
3. Perform the diffusion step.
4. Copy the parts of the ion grids between PE's (UDD $\rightarrow$ ASD - see Fig. 6).
5. Calculate the force acting on the colloids and perform a molecular dynamics step.

The discussion below will show that the copying process does not effect efficiency, because it can be performed in background during calculation.

Our finest spatial granularity is the size of the linked-cells, so blocks of these size are exchanged between the different PE's. Due to this copying process and dynamical adjustment of the spatial decomposition a simple decomposition similar to the uniform domain decomposition would be rather inefficient. This forces us to change our domain decomposition of the finite difference grids completely. Since only cells are moved between PE's during an adjustment the finite difference grids are stored block wise, i. e. our complete finite difference grid is constructed by a block grid, which contains small grid objects as its entries. All calculations concerning finite difference grids are performed using these grid blocks. This allows us to exchange blocks and adjust the local part of the finite difference grid very efficiently.

In a second approach one can think of not copying blocks between various PE's but trying to speed up solving the diffusion equation (Eq. (19)). This can be done by using a coarse grid constant in domains free of colloidal particles. By using a grid constant that is just half of the fine grid constant we win a factor of eight in computation time. Again we use the linked-cells as the smallest domain with an equal grid constant. If one cell is free of colloidal particles, the diffusion equation is solved with a coarse grid constant otherwise with a fine grid constant - see Fig. 7. Again we need a block wise storage of the grid objects, but copying blocks between PE's is only necessary if the spatial decomposition is

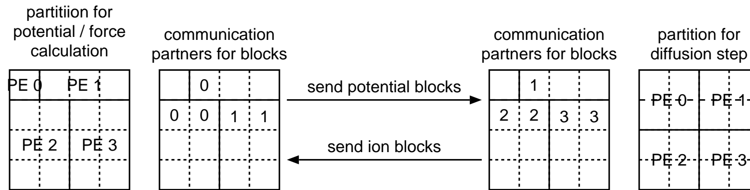


Figure 6. Copying grid blocks between the PE's

adjusted. Using this approach some more work has to be done, if a colloid is moving into an empty cell or leaving as last particle one cell. Then we have to switch from coarse to fine grid constant and vice versa. This means nothing else than the density data have to be copied (and interpolated when switching from a coarse to a fine grid).

Earlier in this section the question arised when to adjust the spatial decomposition. Due to the movement of the colloids it is clear that the local particle density changes within a simulation. A first simple solution would be to update the spatial decomposition after a fixed number n of time steps. This is very easy to implement but disturbs our dynamic load balancing. The resulting efficiency varies very sensitively with n. If n is chosen too small, an adjustment is performed too often, if n is too large, heavy load imbalances can occur, which become even worse than for a uniform domain decomposition. A better strategy includes a dynamical adjustment. As a criterium for the need of new a decomposition scheme can refer the variance of the average load for each PE, i. e. the number of local particles managed by one PE. If the variance crosses a certain threshold value, adjustment of the spatial decomposition scheme is performed.

Finally the flow diagram of our simulation process using dynamic load balancing is shown in Fig. 8. Comparison with Fig. 2 shows that diffusion step and molecular dynamics step have been exchanged. This allows us to copy the blocks of the finite difference grids from one PE to another in the background and improve efficiency. The ion grids are not needed for calculation of the electrostatic potential, while the potential grid is not needed during the force calculation. Exchanging colloid data in background is not possible, because the new particle positions are needed for adjusting the boundary conditions. So this exchanging is automatically done during the linked-cell update.

Within the approach of a coarse and a fine grid constant one just needs to copy blocks after an adjustment of the decomposition scheme. Therefore much less communication is required. It is expected that this approach gets its best efficiency for very low densities of colloidal particles.

4 Benchmark results

Currently an OpenMP parallelization and a uniform domain decomposition approach of the simulation software are implemented. It is planned to compare these approaches with the two introduced load balancing techniques. In this section results of some benchmark tests are shown and discussed.

4.1 Speedup achieved with OpenMP

The following architectures were available for testing the OpenMP-version of the software:

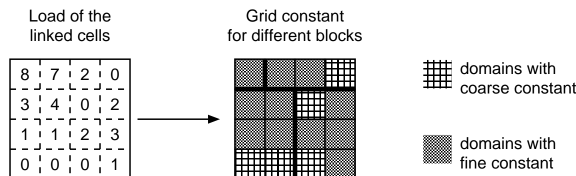


Figure 7. Choosing coarse or fine grid constants

- a SMP node with four Pentium III Xeon processors (one node of ZAMpano at ZAM) running Linux,
- an IBM RS/6000 SP with four Power 3 Winterhawk II processors running AIX 4.3 and
- a SGI Origin 3000 with 128 Processors running Irix 6.5.

The measurements were performed with ≈ 400 colloidal particles in a $16 \times 16 \times 16$ volume (volume size in diameters of colloids) and a grid constant of $g = \frac{1}{8}$, i.e. the size of the finite difference grids was $128 \times 128 \times 128$ points. A larger system was used for the Origin 3000 in order to keep the number of particles per processor in a reasonable range also for the maximum number of processors. The obtained speedup was scaled appropriately to smaller systems for comparison.

Fig. 9 shows the speedup for the parallel version of the simulation software using OpenMP. As expected the simulation software shows a very good speedup on all three systems, on the Origin even a super-linear speedup. Measurements on ZAMpano show the lowest efficiency. Probably the bandwidth of the data bus of the computer becomes the bottleneck in this case. This has been tested with two different Compilers (PGI and GuideC), the results were the same. The same problem occurs for large numbers of processors on the Origin machine.

4.2 Efficiency for a uniform domain decomposition

The simple OpenMP approach gives good results for small numbers of PE's, however, a parallelization on massively parallel systems requires a MPI-implementation of our simulation software. As the discussion above shows, this introduces a lot of overhead due to book-keeping especially for molecular dynamics.

Two architectures were available for benchmarks:

- a Cray T3E with 512 Alpha 21164 processors (T3E-1200 at ZAM) and
- a SMP-Cluster with eight nodes, each with four Pentium III Xeon processors running Linux (ZAMpano at ZAM).

Currently only the parallelized simulation software using a uniform domain decomposition is implemented and was used for the benchmarks. Our test system consisted of ≈ 3300 colloids in a $32 \times 32 \times 32$ volume (again in diameters of colloids), grid constant was $g = \frac{1}{8}$ leading to a resulting grid size of $256 \times 256 \times 256$ points.

Fig. 10 shows results of the performance tests. The scalability is good, but efficiency is decreasing for large number of PE's (on Cray to $\approx 78\%$ for 64 PE's). It is expected that

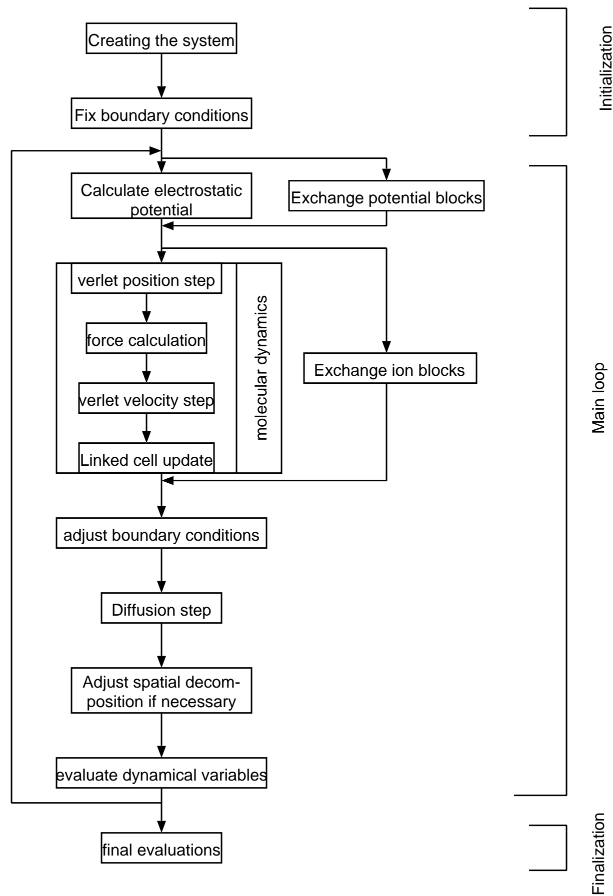


Figure 8. Flow diagram of the simulation process with load balancing

load imbalances bound the speedup here. Because of the required memory of $\approx$ 600MB no test-run for a single PE has been performed on Cray. Correspondingly the speedup has been normalized to the use of 2 PE's.

Profiling was done using 8 PE's for two different densities of colloidal particles, one for a volume fraction of ≈ 0.05 and a second for a volume fraction of $\approx 6.5 \cdot 10^{-3}$. Volume size was increased in second case accordingly so that the number of colloids remained constant. Measurements show an efficiency of about 80% for high and just 50% for low particle density. Measurements of the local number n of particles per PE showed that for low colloid densities n differs by up to a factor of two on different PE's. This results indicates the strong need for load balancing techniques, since low particle densities are the mean target in future simulations.

5 Conclusions and Outlook

As expected the parallel version of the simulation on shared memory-systems gives a very good efficiency, since nearly the total simulation program consists of loops that allow ideally work sharing. The OpenMP-standard allows the programmer to add some directives to his code and achieve a good speedup for small number of processors. This is a fast and easy way for parallelization.

On distributed memory sytems, the parallelization of the software is more complicated. The uniform domain decomposition currently implemented in the program lacks efficiency

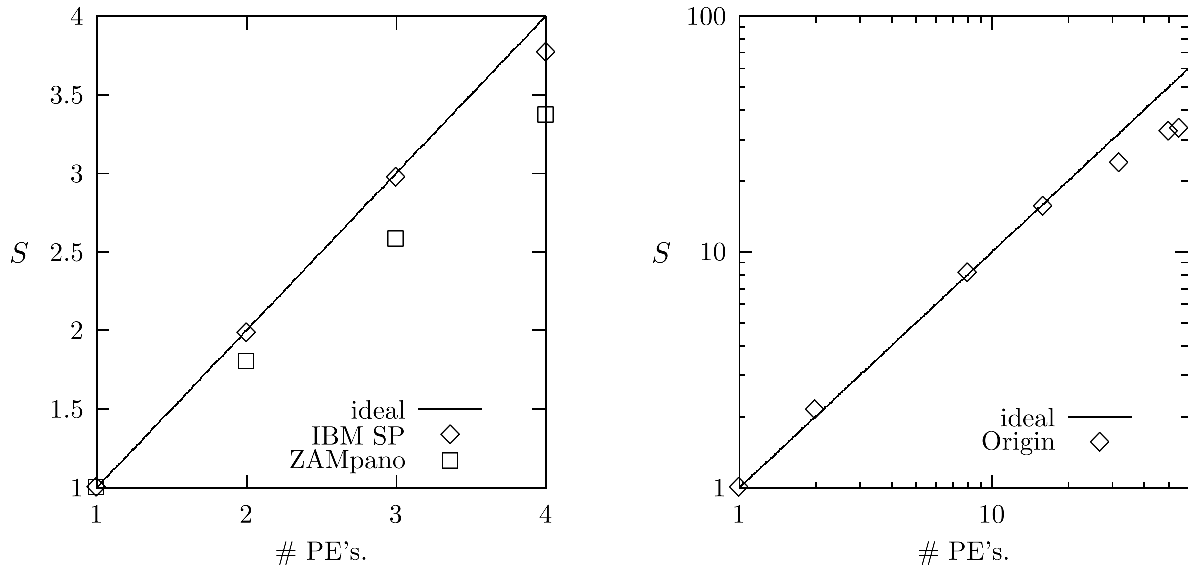


Figure 9. Speedup S measured on various shared-memory architectures

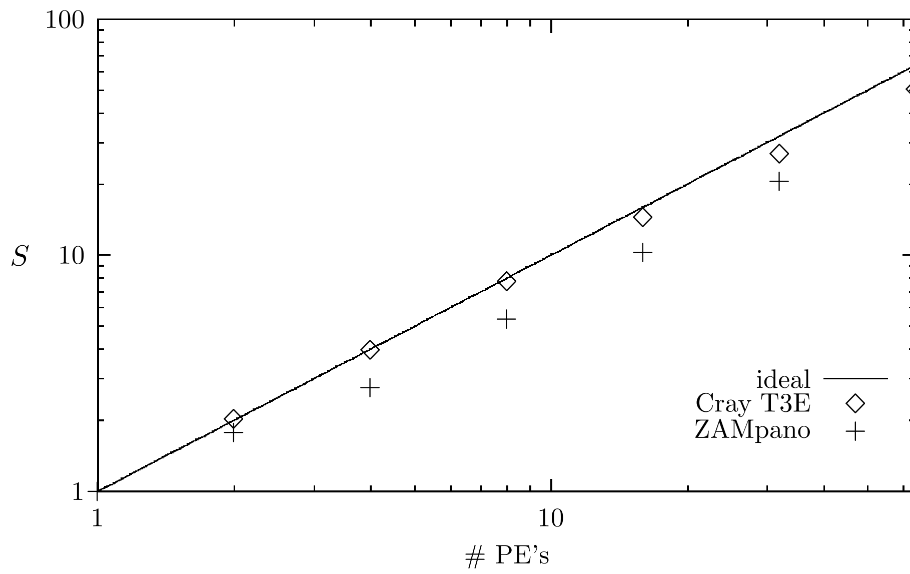


Figure 10. Speedup S for uniform domain decomposition on message-passing systems

in the case of low particle densities, since the number of particles per PE differs by up to a factor of two on the system. Here load balancing strategies introduced are necessary to achieve a reasonable performance on MPP-systems and a large number of PE's.

It is planned to implement both load balancing strategies discussed in section three as well as a hybrid implementation of the software, i. e. a combination of OpenMP and MPI on SMP-clusters like ZAMpano or the IBM SP introduced above.

Due to lack of time it is not possible to present some physical results here, like the change of the fractal dimension. The physical problem is governed by a good number of adjustable parameters, which first need to be fixed in an appropriate way before starting production runs. The current work is focused on this problem and we hope to present first results in the near future.

Acknowledgements

I want to thank my hosts at ZAM, Dr. R. Berrendorf and Dr. G. Sutmann. I am also grateful to Dr. B. Steffen and D. Koschmieder for many helpful discussions and Dr. R. Vogelsang, who performed the benchmarks on the SGI Origin machine.

References

1. H. J. Mögel, G. Breszinski, *Grenzflächen und Kolloide*, Spektrum Akademischer Verlag, 1993
2. P. E. Smith, B. M. Pettitt, *J. Phys. Chem* 98, 9700-9711 (1994)
3. R. Haberland et al., *Molekulardynamik*, Vieweg, 1995
4. J. A. MacLennan, *Introduction to none equilibrium statistical mechanics*, Prentice Hall, 1989
5. N. Attig et al., *Molecular dynamics algorithms for massively parallel computers* in R.Esser et al. (*eds.*), *Molecular dynamics on parallel computers*, p. 46., World Scientific 2000
6. W. H. Press et al., *Numerical Recipes in C* , 2nd edition, Cambridge Univ. Press, 1994
7. B. Hendrikson, K. Devine *Comp. Meth. Appl. Mech. & Eng.* to be published, currently available at: `ftp://ftp.cs.sandia.gov/pub/papers/bahendr/dyn_survey.ps.gz`

A Derivation of Eq. (19)

This section shows the derivation of the finite difference equation for simulating a diffusion step. The partial differential equation to solve during the simulation was Eq. (17), which is shown again as starting basis:

$$\frac{\partial c}{\partial t} = D\Delta c + \mu q(\langle \text{grad} c, \text{grad}\Phi\rangle + c\Delta\Phi) \tag{20}$$

Due to simplicy the subscript $\pm$ used before is left out here. c is the density of the ions at the point $\mathbf{x}$ at the time t and Φ the electrostatic potential. The ions are assumed to have the charge q and mobility μ. This partial differential equation has to be transformed into a finite difference equation. Approximating the partial spatial derivation

$$\frac{\partial c(\mathbf{x})}{\partial x} \approx \frac{c(\mathbf{x} + \zeta\mathbf{e}_x) - c(\mathbf{x} - \zeta\mathbf{e}_x)}{2\zeta} \tag{21}$$

with the canonical unit vector $\mathbf{e}_x$ and a small difference ξ lets Δc be

$$\begin{aligned} \Delta c &\approx \text{div grad } c & (22) \\ &= \sum_{i=x,y,z} \frac{\partial}{\partial i} \frac{c(\mathbf{x}+\zeta\mathbf{e}_i) - c(\mathbf{x}-\zeta\mathbf{e}_i)}{2\zeta} & (23) \end{aligned}$$

with $\frac{\partial}{\partial i}$, $i = x, y, z$ the partial derivations $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial z}$, $\frac{\partial}{\partial z}$ are denoted, respectively.

Application of Eq. (21) on $c(\mathbf{x} \pm \zeta\mathbf{e}_i)$ gives

$$\begin{aligned} \Delta c &\approx \sum_{i=x,y,z} \frac{1}{2\zeta} \left[\frac{c(\mathbf{x}+2\zeta\mathbf{e}_i) - c(\mathbf{x})}{2\zeta} - \frac{c(\mathbf{x}) - c(\mathbf{x}-2\zeta\mathbf{e}_i)}{2\zeta} \right] & (24) \\ &= \underbrace{\left[\sum_{i=x,y,z} \frac{c(\mathbf{x}+2\zeta\mathbf{e}_i) + c(\mathbf{x}-2\zeta\mathbf{e}_i)}{4\zeta^2} \right]}_{\substack{\text{summation over all points} \\ \text{with distance } 2\zeta \text{ from } \mathbf{x}}} - \frac{6}{4\zeta^2} c(\mathbf{x}) & (25) \end{aligned}$$

Now we turn to the second term on the right hand side of Eq. (20). Both gradients are approximated using Eq. (21)

$$\langle \text{grad} c(\mathbf{x}), \text{grad} \Phi(\mathbf{x}) \rangle = \frac{1}{4\zeta'^2} \sum_{i=x,y,z} [c(\mathbf{x}+\zeta'\mathbf{e}_i) - c(\mathbf{x}-\zeta'\mathbf{e}_i)] \cdot [\Phi(\mathbf{x}+\zeta'\mathbf{e}_i) - \Phi(\mathbf{x}-\zeta'\mathbf{e}_i)]$$

The last term in the right hand side of Eq. (20), i. e. $\Delta\Phi$, can be simplified using the Maxwell equation

$$\Delta\Phi = \frac{1}{\epsilon}\rho = \frac{1}{\epsilon}(q_+ c_+ + q_- c_-) \tag{27}$$

The time derivation for c is approximated by discrete time steps of size δt:

$$\frac{\partial c}{\partial t} \approx \frac{c(t+\delta t) - c(t)}{\delta t} \tag{28}$$

Now we use a finite difference grid with the size of one grid cell $g = 2\zeta = \zeta'$ to represent the densities c and the electrostatic potential Φ. Then the summation in Eq. (25) iterates over all nearest neighbours (NN). Inserting now Eqs. (25), (26), (27) and (28) into Eq. (20) gives

$$\begin{aligned} c(\mathbf{x}, t+\delta t) &= \underbrace{s \cdot c(\mathbf{x}, t) + r \sum_{\mathbf{j} \in \text{NN}} c(\mathbf{j}, t)}_{\hat{=} c + D\Delta c} \\ &+ \underbrace{u \sum_{i=x,y,z} [c(\mathbf{x}+g\mathbf{e}_i) - c(\mathbf{x}-g\mathbf{e}_i)] \cdot [\Phi(\mathbf{x}+g\mathbf{e}_i) - \Phi(\mathbf{x}-g\mathbf{e}_i)]}_{\hat{=} \mu q \langle \text{grad} c, \text{grad} \Phi \rangle} \\ &+ \underbrace{\left(-\frac{\mu}{\epsilon} c[q_+ c_+ + q_- c_-] \right)}_{\hat{=} c\Delta\Phi} & (29) \end{aligned}$$

The parameters s, r and u are derived from the diffusion constant D, length δt of one time step, size of one grid cell g and mobility μ as:

$$s = 1 - 6r, \qquad r = \frac{D\delta t}{g^2} \quad \text{and} \quad u = \frac{\mu q}{4g^2} \tag{30}$$

Einfluss elektrischer Ladungen auf Stabilität und Agglomerationsvorgänge in Suspensionen

Jochen Werth

Gerhard-Mercator-Universität Duisburg
Fachbereich 10 - Physik
e-Mail: werth@comphys.uni-duisburg.de

Zusammenfassung: Mit dem vorgestellten Computerprogramm werden Suspensionen elektrisch geladener Teilchen simuliert und die Agglomeration der Partikel untersucht. Durch Darstellung der hydrodynamischen Wechselwirkungen in der Stokeslet-Näherung kann auf eine explizite Simulation der umgebenden Flüssigkeit verzichtet werden. Die Teilchentrajektorien werden mittels paralleler Molkulardynamiktechniken verfolgt. Der zur Kraftberechnung verwendete hypersystolische Algorithmus wird ausführlich dargestellt.

A Einleitung

Granulare Materialien haben in der heutigen Verfahrenstechnik eine große Bedeutung. In vielen industriellen Prozessen, die Feststoffe verarbeiten, werden diese in Form von Granulaten oder Pulvern eingesetzt. Ein großer Teil der damit verbundenen Probleme ist heute noch nicht gelöst. Insbesondere der Einfluß der elektrischen Aufladung von Granulaten ebenso wie der eigentliche Aufladungsprozess isolierender Materialien ist noch weitgehend unverstanden.

Ein spezielles Problem in diesem Zusammenhang ist das Verklumpen feiner Pulver: Durch van der Waals-Kräfte zwischen den Partikeln kommt es zur Bildung größerer Klumpen. Dies kann z.B. zum Verstopfen von Rohren und Trichtern führen oder aber für die weitere Verwendung des Pulvers hinderlich sein.

Eine Methode, die van der Waals-Anziehung der Teilchen in einem Pulver A zu verringern, ist die Bedeckung der Partikel mit einem feineren Pulver B [1]. Die B-Teilchen dienen dabei als Abstandshalter zwischen den A-Teilchen und reduzieren somit deren van der Waals-Wechselwirkung. Technisch ergibt sich hier jedoch das Problem, einen geeigneten Beschichtungsprozess zu finden, da die anziehende Wechselwirkung in Pulvern mit sinkendem Teilchendurchmesser stark ansteigt und deshalb das B-Pulver vor der Deposition im allgemeinen noch stärker zum Verklumpen neigt.

Um diese Verklumpung zu vermeiden, bietet sich als mögliches Verfahren an, beide Pulver entgegennamig elektrostatisch zu laden und in einer geeigneten Flüssigkeit in Suspension zu halten. Die abstoßende Wechselwirkung gleichnamig geladener Teilchen unterdrückt dabei Kontakte A-A bzw. B-B und fördert Kontakte A-B. Durch die umgebende Flüssigkeit wird ein vorzeitiges Verklumpen der Pulver verhindert. Dabei hat sich flüssiger Stickstoff als geeignetes Medium herausgestellt, da diese Flüssigkeit stark unpolar ist und

damit die Ladungen der Partikel nicht abschirmt. Zudem kann das Pulver nach dem Beschichtungsvorgang leicht zurückgewonnen werden.

Ziel des im folgenden beschriebenen Projektes ist es, ein Verständnis für die Vorgänge in Suspensionen geladener Partikel zu gewinnen. Dabei soll insbesondere die Stabilität der Suspension in Hinsicht auf das Clustern der Partikel untersucht werden. Zu untersuchende Parameter sind hier das Verhältnis der Ladungen, Radien und Konzentrationen der Partikel. Ziel ist es, diejenigen Werte der Parameter zu finden, bei denen die Beschichtung des Pulvers A optimal ist. Dadurch soll der Aufwand verfahrenstechnischer Versuchsreihen reduziert werden, da bisher eine Bestimmung "optimaler" Parameter nur durch empirische Untersuchungen möglich ist.

Die Untersuchung soll mit Methoden der Molekulardynamiksimulation durchgeführt werden. Um die in der Simulation notwendigen großen Teilchenzahlen erreichen zu können, soll auf eine explizite Simulation der umgebenden Flüssigkeit verzichtet werden. Statt dessen wird die langreichweitige hydrodynamische Wechselwirkung der Partikel durch Kräfte in der Stokeslet-Approximation angegeben. Die jeweiligen Kräfte werden im folgenden Abschnitt beschrieben.

Im zweiten Teil dieses Berichts wird dann genauer auf das verwendete Simulationsverfahren eingegangen. Insbesondere wird dabei die im Rahmen des Gaststudentenprogramms geschriebene parallele Version des Simulationsprogramms vorgestellt. Schließlich folgt eine kurze Zusammenstellung erster Ergebnisse.

B Physikalische Grundlagen

Auf die suspendierten Partikel wirken elektrostatische und hydrodynamische Kräfte. Im einzelnen sind dies:

- van der Waals-Kräfte,
- die Coulomb-Kraft,
- die Stokes'sche Reibung und
- hydrodynamische Wechselwirkungskräfte.

Ziel ist es, den jeweiligen Beitrag dieser Kräfte zur Clusterbildung in der Suspension zu untersuchen.

B.1 Van der Waals-Wechselwirkung

Die van der Waals-Kräfte werden durch fluktuierende elektrische Dipole verursacht und stellen eine wesentliche Wechselwirkung zwischen einzelnen Molekülen dar. Das Potential nimmt dabei mit wachsendem Abstand r zwischen den Molekülen wie r^{-6} ab.

Da es sich bei den Partikeln in der Suspension um makroskopische Teilchen handelt (Durchmesser der Partikel ca. 50-500 μm (Pulver A) bzw. 0.1-100 μm (Pulver B)), muß über die Wechselwirkung aller Moleküle zweier Partikel integriert werden. Mit der Annahme, daß es sich bei den Teilchen um Kugeln mit Radius a_1 bzw. a_2 handelt und h der

Abstand der Kugeloberflächen ist, erhält man [2]:

$$F = -\frac{Aa_1a_2}{6h^2(a_1+a_2)} f(p) \tag{1}$$

mit

$$f(p) = \begin{cases} \frac{1+3.54p}{1+1.77p}, & p < 1 \\ \frac{0.98}{p} - \frac{0.434}{p^2} + \frac{0.067}{p^3}, & p > 1 \end{cases}$$

wobei

$$p = \frac{2\pi h}{\lambda_L}$$

und $\lambda_L \approx 100$ nm die Londonsche Retardationswellenlänge ist. A bezeichnet die Hamaker-Konstante, die sowohl von den Eigenschaften des Partikels als auch von denen des Mediums abhängt. Typischerweise liegt A in der Größenordnung $10^{-19} \ldots 10^{-21}$ J.

B.2 Coulomb-Wechselwirkung

Während die van der Waals-Kraft immer bei allen Partikeln zu finden ist, wird die Coulomb-Kraft durch die aufgebrachte elektrische Ladung der Partikel verursacht und kann deshalb beeinflußt werden. Die Ladungen befinden sich auf der Oberfläche der Partikel. Die aufgebrachten Oberflächenladungen liegen dabei in der Größenordnung von 10^{-7} C/m^2 [1]. Zur Vereinfachung wird jedoch davon ausgegangen, daß sich die gesamte Ladung eines Partikels in dessen Schwerpunkt befindet.

Zusätzlich soll angenommen werden, daß die Flüssigkeit zwischen den Partikeln ideal unpolar ist und auch keine Ionen enthält. Die Ladungen der Teilchen werden also nicht durch die Flüssigkeit abgeschirmt. Damit ergibt sich bei einem Abstand r der Teilchenzentren eine Kraft von

$$\mathbf{F} = \frac{1}{4\,\pi\,\varepsilon\,\varepsilon_0}\,\frac{q_1\,q_2}{r^2}\,\frac{\mathbf{r}}{r} \quad . \tag{2}$$

B.3 Stokes'sche Reibung

Die Stokes'sche Reibung beschreibt die Kraft auf ein durch die umgebende Flüssigkeit bewegtes Teilchen. Der Kraftterm lautet

$$\mathbf{F} = -6\pi\eta a\mathbf{v} \tag{3}$$

wobei η die Viskosität der Flüssigkeit und $\mathbf{v}$ die Geschwindigkeit des Teilchens ist.

B.4 Hydrodynamische Wechselwirkungskräfte

Wie bereits in der Einleitung gesagt, soll die Flüssigkeit nicht explizit mitsimuliert werden. Stattdessen werden die Kräfte zwischen den Partikeln in der sogenannten Stokeslet-Approximation berechnet [3]. Diese wird im folgenden kurz hergeleitet.

Nimmt man zunächst an, daß sich die Partikel ungestört durch hydrodynamische Wechselwirkungen in dem von den anderen Ladungen erzeugten elektrischen Feld bewegen, so

ergibt sich ihre Geschwindigkeit $\mathbf{v_0}$ aus dem Gleichgewicht zwischen der Coulomb-Kraft $\mathbf{F}_{\mathrm{el}}$ und der Stokes'schen Reibung:

$$\mathbf{F}_{\mathrm{el}} = 6\pi\eta a \mathbf{v}_0$$

Durch seine Bewegung erzeugt jeder Partikel in der umliegenden Flüssigkeit ein Geschwindigkeitsfeld. Für ein kugelförmiges Teilchen mit Radius a, dessen Schwerpunkt sich im Koordinatenursprung befindet und das die konstante Geschwindigkeit $\mathbf{v_0}$ hat, ergibt sich die Geschwindigkeit $\mathbf{u}(\mathbf{r})$ der Flüssigkeit am Ort $\mathbf{r}$ zu

$$\mathbf{u}(\mathbf{r}) = \frac{3}{4}\, a \left(\frac{\mathbf{v}_0}{r} + \frac{(\mathbf{v}_0 \cdot \mathbf{r})\mathbf{r}}{r^3} \right) .$$

In erster Näherung kann die Wechselwirkung der Partikel bestimmt werden, indem in der Stokes'schen Reibung (3) nicht die 'absolute' Geschwindigkeit der Partikel, sondern ihre Relativgeschwindigkeit gegenüber dem von den anderen Partikeln erzeugten Geschwindigkeitsfeld der Flüssigkeit berücksichtigt wird. Damit ergibt sich als modifizierte Stokes'sche Reibung für einen Partikel i in der Stokeslet-Näherung:

$$\mathbf{F}_{\mathrm{i}} = -6\pi\eta a \left(\mathbf{v}_{\mathrm{i}} - \sum_{\mathrm{j}\neq\mathrm{i}}^{n} \mathbf{u}_{\mathrm{j}}(\mathbf{r}_{\mathrm{i}} - \mathbf{r}_{\mathrm{j}}) \right) \tag{4}$$

Bei Partikeln, die nur einen sehr geringen Abstand zueinander haben, muß noch ein weiterer Kraftterm, die Schmierwechselwirkung, berücksichtigt werden:

$$\mathbf{F} = -6\pi\eta a_{\mathrm{i}} a_{\mathrm{j}} \frac{\Delta \mathbf{v}}{h} \tag{5}$$

Dabei sind a_{i} und a_{j} die Radien der Teilchen und $\Delta\mathbf{v}$ die Relativgeschwindigkeit in Richtung der Verbindungslinie ihrer Schwerpunkte.

C Computersimulation

C.1 Molekulardynamik-Simulation

Bei der Molekulardynamik handelt es sich um eine Simulationsmethode, in der mittels Integration der Newtonschen Bewegungsgleichung Ort und Geschwindigkeit aller N Systemteilchen im Zeitverlauf nachvollzogen werden. Dies ermöglicht die Simulation von Systemen, die sich fernab vom Gleichgewicht befinden. Die Bewegungsgleichung der Partikel wird dabei in diskreten Zeitschritten integriert. Je nach Art des verwendeten Integrationsalgorithmus sind dabei pro Zeitschritt eine oder mehrere Berechnungen aller Kräfte notwendig. Daher fällt im allgemeinen auch fast der gesamte Rechenaufwand einer Molekulardynamiksimulation in der Kraftberechnungsroutine an.

Für das hier betrachtete Problem ergibt sich der folgende Ablauf der Kraftberechnung:

- Berechnung der Coulomb-Wechselwirkung aller Teilchen
- Bestimmung der langreichweitigen hydrodynamischen Wechselwirkung in Stokeslet-Näherung mit Hilfe der vorher bestimmten Coulomb-Kräfte.

- Berechnung der kurzreichweitigen Wechselwirkung (Schmierkräfte) und der Stokes'schen Reibungskraft.

Zu beachten ist, daß für die ersten beiden Punkte bei N Systemteilchen $\frac{3}{2}N(N-1)$ Kraftberechnungen notwendig sind (da nur für die Coulomb-Wechselwirkung *actio = reactio* gilt). Der Berechnungsaufwand wächst also quadratisch in der Teilchenzahl.

C.2 Systolische und hypersystolische Algorithmen

Grundsätzlich eignen sich Systeme mit ausschließlich kurzreichweitigen Wechselwirkungskräften besser für eine Parallelisierung auf Computersystemen mit verteiltem Speicher als Systeme mit langreichweitigen Kräften: Daten benachbarter Teilchen können nach Möglichkeit jeweils im Speicher einer CPU gehalten werden, so daß nur für die Teilchen an den Rändern des jeweiligen Gebietes Daten von anderen Prozessoren benötigt werden (und damit zwischen den Speicherbereichen kopiert werden müssen). Bei dem hier vorgestellten Problem muß jeweils die Wechselwirkung jedes Teilchens mit jedem anderen berücksichtigt werden, da ja gerade der Einfluß der langreichweitigen Kräfte auf das System untersucht werden soll.

Eine Möglichkeit zur Behandlung langreichweitiger Kräfte stellen die systolischen Algorithmen dar. Von diesen soll zunächst die einfachste Version vorgestellt werden.

Angenommen, es sollen die kompletten Coulombkräfte zwischen N Teilchen auf einem Computersystem mit 6 Prozessoren und verteiltem Speicher (NORMA - no remote memory access) berechnet werden. Die einfachste Möglichkeit ist, die N Teilchen gleichmäßig auf die CPUs zu verteilen und auf jedem Prozessor zunächst die Kräfte zwischen den lokalen Teilchen zu ermitteln. Danach werden die Prozessoren logisch zu einem Ring "zusammengeschaltet": Die Teilchendaten jeder CPU werden im Kreis an den jeweiligen Nachbarn gesendet. CPU 1 erhält also die Daten von 2, 2 von 3, usw. Prozessor 6 bekommt die Teilchendaten von Prozessor 1 geliefert. Dann bestimmt jeder Prozessor, welche Kräfte die "neuen" Teilchen auf die bisherigen "alten" Teilchen ausüben und addiert sie zu den bisher berechneten Kräften hinzu. Dann werden die eben empfangenen Daten wieder an den nächsten Nachbarn weitergesendet: Die ursprünglichen Daten der CPU 3 gelangen also zu CPU 1, die von 4 zu 2, usw. Tabelle 1 zeigt die Position der Datenpakete im Verlauf des Updates.

CPU		1	2	3	4	5	6
Schritt	0	1	2	3	4	5	6
	1	2	3	4	5	6	1
	2	3	4	5	6	1	2
	3	4	5	6	1	2	3
	4	5	6	1	2	3	4
	5	6	1	2	3	4	5

Tabelle 1. Position der Datenpakete während eines systolischen Updates. Zu Beginn (Schritt 0) befindet sich Datenpaket 1 auf der CPU 1, Paket 2 auf CPU 2, usw. Danach werden die Daten zyklisch versendet, bis jedes Datenpaket einmal bei jedem Prozessor war.

Betrachtet man den Algorithmus näher, fallen zwei Kritikpunkte sofort auf: Zum einen wird die Symmetrie der Kraftberechnung (*actio = reactio*) nicht ausgenützt. Eine sol-

che Symmetrie ist aber ohnehin nicht immer zu finden (sie gilt z.B. nicht in den oben betrachteten hydrodynamischen Wechselwirkungen).

Der zweite Punkt betrifft den Kommunikationsaufwand: Wie man in Tabelle 1 sehen kann, befindet sich jedes Datenpaket während des Updates irgendwann einmal im lokalen Speicher jedes Prozessors. Würden die kopierten Daten (im Gegensatz zum eben beschriebenen Algorithmus) auch an jedem Prozessor dauerhaft gespeichert werden, so hätte nach allen Kopiervorgängen jeder Prozessor die erforderlichen Daten, um alle Wechselwirkungen im gesamten System zu berechnen. Eigentlich ist es aber völlig ausreichend, wenn die Daten zur Kraftberechnung zwischen einem beliebigen Paar der Datenpakete nur auf einem der sechs Prozessoren zu finden sind. Es muß also nach einem Algorithmus gesucht werden, der die Datenpakete so verteilt, daß

1. die Wechselwirkung jedes möglichen Paares der Datenpakete auf mindestens einem Prozessor berechnet werden kann.
2. möglichst wenige Paare der Datenpakete auf mehreren Prozessoren liegen, damit möglichst wenig Kommunikation anfällt.

Der Algorithmus soll zusätzlich auf allen Prozessoren symmetrisch arbeiten, d.h. auf jeder CPU soll eine Anweisung der Art "schicke Deine Daten an Deinen übernächsten Nachbarn" ausgeführt werden.

Diese Anforderungen werden von den "hypersystolischen" Algorithmen erfüllt [4]. Tabelle 2 zeigt einen solchen Algorithmus für unser Problem. Die Daten werden in nur 2 Kommunikationsschritten an die Nachbarn kopiert, dann kann jedes mögliche Wechselwirkungspaar auf einem der Prozessoren gefunden werden. Als Beispiel sind die Datenpakete, die für die Kraftberechnung der Teilchen aus Datenpaket 1 benötigt werden, in Tabelle 2 hervorgehoben. CPU 1 berechnet die Wechselwirkungen $1 \leftrightarrow 2$ und $1 \leftrightarrow 4$, CPU 6 die Wechselwirkungen $1 \leftrightarrow 3$ und $1 \leftrightarrow 6$, und CPU 4 schließlich $1 \leftrightarrow 5$. Symmetrisch dazu findet man auch alle Wechselwirkungspaare für die anderen Datenpakete. Zu beachten ist, daß der geringere Kommunikationsaufwand der hypersystolischen Algorithmen durch einen höheren Speicherverbrauch erkauft wird.

CPU		1	2	3	4	5	6
Schritt	0	*1*	2	3	4	5	**6**
	1	**2**	3	4	**5**	6	*1*
	2	**4**	5	6	*1*	2	**3**

Tabelle 2. Position der Datenpakete während eines hypersystolischen Updates.

Im Gegensatz zum einfachen systolischen Algorithmus ist es nun noch notwendig, daß die Kräfte nach der Berechnung noch gesammelt und addiert werden. In dem Algorithmus gemäß Tabelle 2 berechnet z.B. CPU 1 (wie auch schon im systolischen Algorithmus) die Kräfte $2 \rightarrow 1$ und $4 \rightarrow 1$. Zusätzlich werden auch die Kräfte $1 \rightarrow 2$ und $4 \rightarrow 2$ sowie $2 \rightarrow 4$ berechnet und die Ergebnisse an die Prozessoren 2 bzw. 4 versendet. Die Berechnung der Kräfte $1 \rightarrow 4$ ist auf CPU 1 nicht notwendig, da dies schon auf CPU 4 geschieht. Die anderen Prozessoren arbeiten analog.

Das in Tabelle 2 dargestellte Kopierschema der Daten bezeichnet man auch als "hypersystolische Matrix". Diese Matrix hängt von der Anzahl der verwendeten Prozessoren ab. Leider ist bisher noch kein Verfahren bekannt, mit dem für eine beliebige Anzahl von

Prozessoren eine optimale (bezüglich des Kommunikations- und Rechenaufwands) hypersystolische Matrix angegeben werden kann. Im folgenden Abschnitt werden die von mir verwendeten hypersystolischen Matrizen für 8 und 12 Prozessoren dargestellt.

C.3 Das Simulationsverfahren

In meiner Simulation wird der vorgestellte hypersystolische Algorithmus für eine parallele Berechnung der langreichweitigen Kräfte verwendet. Bisher ist der Algorithmus für 8 und 12 Prozessoren implementiert, die zugehörigen hypersystolischen Matrizen sind in den Tabellen 3 und 4 dargestellt.

CPU		1	2	3	4	5	6	7	8
Schritt	0	*1*	2	3	4	5	**6**	7	**8**
	1	**2**	3	4	5	6	**7**	8	*1*
	2	**4**	5	6	7	8	*1*	2	**3**
	3	**5**	6	7	8	1	2	3	4

Tabelle 3. Hypersystolische Matrix bei Verwendung von 8 Prozessoren.

Der Update erfolgt so, daß zunächst die Coulombkräfte aller Teilchen berechnet werden. In einem folgenden zweiten Durchlauf des hypersystolischen Schemas werden dann die hydrodynamischen Stokeslet-Kräfte bestimmt. Dabei ist es für die Effizienz der Kraftberechnung unerheblich, ob die Daten im lokalen Speicher eines Prozessors zu benachbarten oder weit entfernten Teilchen gehören.

Die Schmierwechselwirkung der Teilchen und die van der Waals-Kräfte können vereinfacht behandelt werden: Die van der Waals-Kraft zwischen den Teilchen erreicht bei den simulierten Teilchengrößen erst bei sehr geringen Teilchenabständen die Größenordnung der anderen Kräfte und wird dann bei noch geringeren Abständen dominant. Sie wird daher als eine reine "Klebekraft" zwischen den Teilchen simuliert: Sobald es zu einem Zusammenstoß zweier Teilchen kommt, werden diese an der Kontaktstelle miteinander verklebt und können sich im weiteren Verlauf der Simulation nicht mehr voneinander trennen.

Auch die Schmierwechselwirkung zwischen den Partikeln führt nur bei geringen Teilchenabständen zu einer wesentlichen Abweichung von der Stokeslet-Kraft. Hinzu kommt, daß diese Kraft in der angegebenen Form bei einem Teilchenkontakt divergiert und damit numerisch schwierig zu behandeln ist. Daher soll diese Kraft zunächst nicht mitsimuliert werden.

Nachdem die Kräfte für jeden einzelnen Partikel bestimmt wurden, wird bei den zu einem Cluster verklebten Teilchen die Gesamtkraft auf den Cluster aus den Einzelkräften bestimmt. Hierzu sollten die Daten aller zu einem Cluster gehörenden Teilchen im lokalen Speicher eines einzelnen Prozessors liegen. Dies wird erreicht, indem eine Gebietsaufteilung des simulierten Volumens durchgeführt wird. Jedem Prozessor werden die Daten eines bestimmten Teilgebietes zugeordnet. Dadurch wird auch weniger Kommunikation notwendig, um Kollisionen (und damit Clusterbildung und -wachstum) zwischen den Teilchen zu entdecken.

Ein Updateschritt der Simulation hat also den folgenden Ablauf:

- Überprüfung des Systems auf Teilchenkollisionen und Aktualisierung der Clusterli-

CPU		1	2	3	4	5	6	7	8	9	10	11	12
Schritt	0	*1*	2	3	4	5	**6**	7	8	9	**10**	11	**12**
	1	**2**	3	4	5	6	**7**	8	9	10	**11**	12	*1*
	2	**4**	5	6	7	8	**9**	10	11	12	*1*	2	**3**
	3	**8**	9	10	11	12	*1*	2	3	4	**5**	6	7

Tabelle 4. Hypersystolische Matrix bei Verwendung von 12 Prozessoren.

ste.

- Berechnung der Coulombkräfte.
- Berechnung der Stokeslet-Kräfte.
- Berechnung der Gesamtkraft auf Teilchenagglomerate.
- Berechnung der neuen Teilchenpositionen und -geschwindigkeiten.
- Auswertung/Speicherung der Daten.

D Stand der Arbeiten und Ausblick

Die beschriebene parallelisierte Version der Simulation ist auf dem ZAMpano-Cluster des Forschungszentrums Jülich implementiert und wird zur Zeit getestet. Momentan liegen noch nicht genügend Informationen vor, um Aussagen zur Skalierung der Performance mit der Prozessorzahl zu machen. Erste Simulationsläufe zeigen hier jedoch ein positives Bild. Es muß jedoch noch geklärt werden, ob die Verteilung der Teilchendaten mittels Gebietszerlegung durch die Teilchenagglomeration zu ungleichen Lastverteilungen zwischen den Prozessoren führt. Dies ist mit der Bildung immer größerer Cluster zu erwarten, wenn die Daten aller Teilchen eines Clusters immer auf nur einer CPU gehalten werden sollen.

Aus physikalischer Sicht ist noch zu beachten, daß die Brown'sche Molekularbewegung der Partikel in der Simulation bislang noch nicht berücksichtigt wird. Diese kann aber wahrscheinlich bei der Größe der untersuchten Partikel nicht vernachläßigt werden. Es ist zu erwarten, daß durch die ungeordnete Teilchenbewegung eine Agglomeration gleichnamig geladener Teilchen erleichtert wird.

Nach einem extensiven Test des Programms soll zunächst ein grundlegendes Verständnis für den Einfluß der einzelnen Kräfte auf die Agglomerationsvorgänge in der Suspension gewonnen werden. Mit diesem Wissen kann dann auch untersucht werden, ob mit den in der Einführung genannten Parametern eine gezielte Steuerung der Clusterbildung möglich ist. Weiter soll untersucht werden, welches Verhalten eine Suspension elektrischer oder magnetischer Dipole zeigt.

Zusätzlich ist nach einer Untersuchung der Skalierbarkeit der Simulation mit der Prozessorzahl eine Erweiterung der Kraftroutinen auf eine größere Anzahl von Prozessoren und eine Portierung auf die Cray T3E geplant.

Danksagung

Die in diesem Bericht vorgestellte parallelisierte Version des Simulationsprogrammes wurde während der Teilnahme am Gaststudentenprogramm "Ausbildung zum wissenschaftli-

chen Rechnen" des Forschungszentrums Jülich erstellt. Ich möchte mich bei allen Mitarbeitern des NIC/ZAM für die freundliche Aufnahme und großartige Unterstützung bedanken. Ich konnte während meines Aufenthalts in Jülich sehr viel lernen. Für die finanzielle Unterstützung danke ich dem Forschungszentrum Jülich und der Firma Silicon Graphics GmbH.

Literatur

1. Prof. K.E. Wirth, Universität Erlangen-Nürnberg, private Kommunikation.
2. T. van de Ven, *Colloidal Hydrodynamics*, Academic Press, London (1989).
3. I.M. Janosi, T. T'el, D.E. Wolf, J.A.C. Gallas, Phys.Rev. E **56**, 2858 (1997).
4. Th. Lippert, H. Hoeber, G. Ritzenhöfer, K. Schilling, *Hyper-Systolic Processing on APE100/Quadrics N^2-Loop Computations*, HLRZ 95-45, WUB 95-21 (1995).

Clock-Synchronisation auf dem ZAMpano Linux SMP-Cluster

Rouslan Zinetoulline

Gerhard-Mercator-Universität Duisburg
Fachbereich 10 - Physik

rouslan@comphys.uni-duisburg.de

Zusammenfassung: Das ZAMpano Linux SMP-Cluster im ZAM ist weitgehend aus Standard PC Hardware-Komponenten aufgebaut. Dies hat zur Folge, dass keine spezielle Hardware zur Synchronisation der Uhren des Clusters existiert. Für viele Anwendungen aus dem Bereich Performance-Tools ist es jedoch unabdingbar, eine synchronisierte globale Uhr auf allen Knoten des Clusters zur Verfügung zu haben; ein Beispiel für solche Anwendungen ist das Event-Tracing-Tool VAMPIR. In der vorliegenden Arbeit wird gezeigt, dass es möglich ist, eine Synchronisation der lokalen Uhren der Cluster-Knoten mit Software-Methoden herbeizuführen, die eine ausreichend hohe Auflösung und Genauigkeit erreicht.

Bei Parallelrechnern ist es in vielen Fällen wichtig, eine globale Uhr im System zu haben. Anwendungen hierfür sind z.B. die sogenannten Event Tracing Tools wie VAMPIR. Dabei müssen verschiedene Ereignisse aufgezeichnet werden. Um die Ereignisse aufzeichnen zu können, braucht man eine gemeinsame Zeit auf allen Prozessoren. Es gibt zwei Möglichkeiten, die gemeinsame (oder globale) Zeit auf allen Prozessoren zu bekommen:

- Jeder Prozessor merkt Ereignisse mit Hilfe der lokalen Uhr, und anschließend werden die Ereignisse von allen Prozessoren zusammengefasst, die Zeitstempel werden korrigiert und in die richtige Reihenfolge gestellt.
- Man benutzt eine Routine, die die globale Zeit liefert. In diesem Fall muß beim Starten des Programms eine Initialisierung durchgeführt werden.

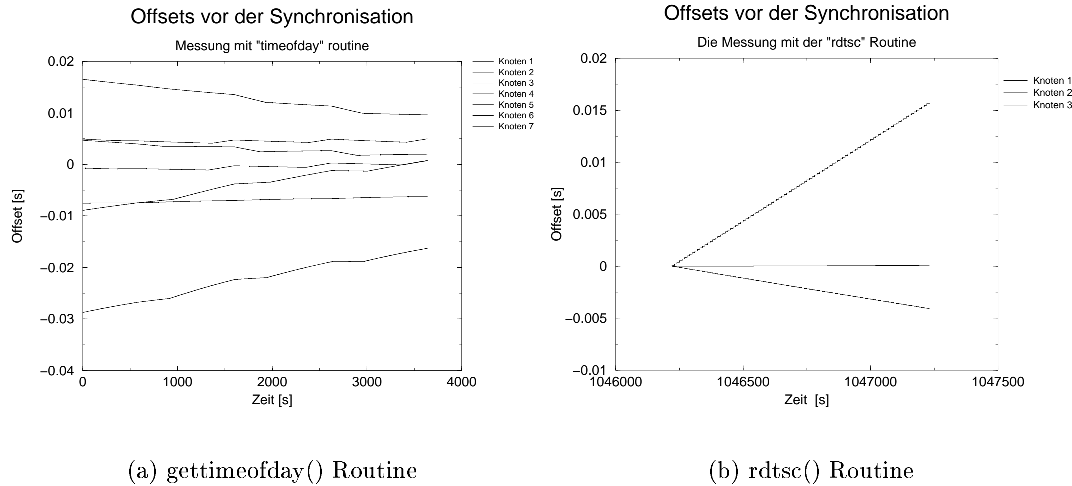
In meiner Arbeit habe ich ein Programm geschrieben, das die Uhren auf dem ZAMpano SMP-Cluster [1] synchronisiert. Es gibt schon Programme, die die Uhren synchronisieren, die Genauigkeit der Synchronisation ist aber zu klein. Ein Beispiel dafür ist das Network Time Protokoll [2]. Dieses Protokoll synchronisiert die Uhren in einem System und ist auf Ethernet-Kommunikation basiert. Die Ethernet-Kommunikation ist relativ langsam und hat große Latenzzeiten, und deswegen ist die Genauigkeit nicht sehr groß.

Die Uhren laufen auf verschiedenen Prozessoren unterschiedlich schnell und haben verschiedene Startzeiten. Die Aufgabe ist, mit Hilfe der Kommunikation die Uhren im System zu synchronisieren.

Man kann auf verschiedene Weise die Zeit auf dem Rechner messen. Die Möglichkeiten sind:

- Systemuhr (gettimeofday()-Funktion)
- Zyklenzähler (rdtsc()-Funktion)
- Hardwareuhr

Die Nachteile der Systemclock sind zu kleine Messgenauigkeit und die Korrektur durch ntp (Network Time Protocol). Die Hardwareuhr ist schwer zu behandeln. Deswegen wurde der Zyklenzähler für die Zeitmessungen benutzt.



(a) gettimeofday() Routine

(b) rdtsc() Routine

Abbildung 1. Messungen der Offsets

Der Zyklenzähler wird durch die rdtsc()-Routine aufgerufen. Er zeigt die Anzahl der Prozessorticks seit einen bestimmten Moment. Er liefert eine Zahl in long long - Format. Man kann es in 'normale' Zeit umwandeln, indem man diese Zahl durch den Prozessortakt dividiert. Diese Routine benutzt Assembler-Befehle und ist deswegen sehr schnell und genau. Theoretisch kann man mit Hilfe dieser Uhr die Zeit mit der Genauigkeit $\frac{1}{Prozessortakt}$ messen.

Im allgemeinen müssen wir eine globale Uhr auf allen Prozessoren haben. Dafür wird ein Prozessor als 'Time-Server' benutzt. Ich nenne diesen Prozessor auch 'Root'. Das heißt, daß alle anderen Prozessoren mit diesem Prozessor synchronisiert werden. Alle anderen Prozessoren simulieren die 'globale Zeit' des Time-Servers. Meine Arbeit basiert auf dem modifizierten Christian-Algorithmus [3]

Die Begriffe:

- korrigierte Uhr: Die Uhr, die auf dem lokalen (nicht Time-Server) Prozessor die globale Zeit liefert.
- Offset: der Unterschied der Uhranzeigen zwischen zwei Prozessoren zu einem bestimmten Moment.
- Drift: der Unterschied in den Geschwindigkeiten der Uhren. Er zeigt, wie schnell die Uhren auseinanderlaufen.

Der Christian-Algorithmus beruht auf einem einfachen Ping-Pong zwischen den Prozessoren. Der Time-Server kommuniziert mit anderen Prozessoren.

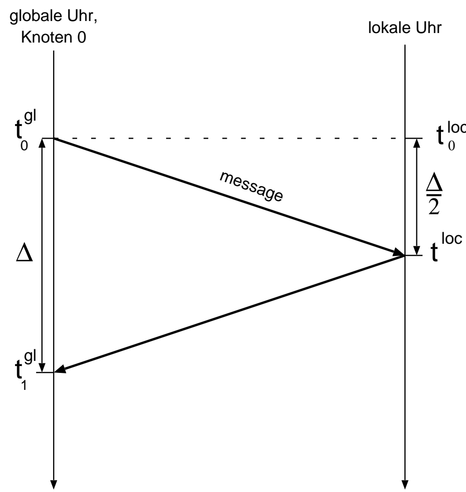


Abbildung 2. Ping-Pong zwischen Prozessoren

In Moment t_0^{gl} schickt der Time-Server eine Nachricht zum anderen Prozessor. Dieser Prozessor misst die Zeit t^{loc}, wenn die Nachricht ankommt und schickt diese Zeit als Nachricht zurück. Der Time-Server empfängt diese Nachricht im Moment t_1^{gl}. Nach diesem Ping-Pong hat der Time-Server die nötige Information, um den Offset zu berechnen. Es ist wichtig, daß dieser Ping-Pong n-mal wiederholt wird; die Nachricht mit der minimalen Dauer wird als 'wahre' angenommen. Das macht man, um mögliche Verzerrungen und Störungen zu vermeiden. Dieser Algorithmus basiert auf der Annahme, daß die Nachricht in Moment $t_0^{gl} + \frac{\Delta}{2}$ beim zweiten Prozessor ankommt. Das stimmt nur dann, wenn der Hinweg genau so lang ist wie der Rückweg. Man kann sagen, daß die Nachricht mit der minimalen Dauer am wenigstens verzögert ist. In der Praxis wurde eine Nachrichtendauer von etwa 30 μs mit Hilfe der MPI-Kommunikation erreicht. Die Dauer der einzelnen Nachrichten beträgt 14-16 μs. Es ist sehr unwahrscheinlich, daß die Nachricht viel schneller in der einen Richtung läuft als in der anderen Richtung. Deswegen ist die Genauigkeit der Messung groß.

Wie man auf der Abbildung $1^{b)}$ sehen kann, ist der Offset eine lineare Funktion der Root-Prozessor Zeit, wenn man die rdtsc()-Routine benutzt. Also, wir simulieren die globale Zeit auf dem lokalen Prozessor als eine lineare Funktion der lokalen Zeit. Dazu dient eine Routine cor_clock(). Sie benutzt die Werte von Offset und Drift, die bei der Initialisierung berechnet werden und liefert eine 'korrigierte' Zeit.

$$t^{gl} = t^{loc} + \text{Offset}(t^{loc})$$
$$\text{Offset}(t) = \text{Offset}_0 + \text{Drift} \cdot t$$

Dabei ist Offset_0 eine Konstante, Offset des Prozessors zum Zeitpunkt t_0. Wir haben hier zwei unbekannte Größen - Offset_0 und Drift.Um sie zu bestimmen, werden zwei Messungen des Offsets mit einem Intervall T durchgeführt. Das Intervall T muß so gewählt werden, daß die Drift genug genau bestimmt wird. In der Praxis ist ein Intervall von etwa 15-20 Sekunden ausreichend.

$$\text{Offset}(t_0) = t_0^{glob} - t_0^{loc}$$
$$\text{Offset}(t_1) = t_1^{glob} - t_1^{loc}$$

mit $t_1 = t_0 + T$. Danach kann man die Drifts berechnen.

$$\text{Drift} = \frac{\text{Offset}(t_1) - \text{Offset}(t_0)}{t_1^{loc} - t_0^{loc}}$$

Die zweite Konstante ist dann

$$\text{Offset}_0 = \text{Offset}(t_0) - \text{Drift} \cdot t_0^{loc}$$

Jetzt haben wir alles, um die korrigierte Uhr zu bekommen.

$$t^{gl} = (1 + \text{Drift}) \cdot (t^{loc} - t_0^{loc}) + \text{Offset}(t_0) + t_0$$

Dabei ist t^{gl} die globale Zeit, die wir auf dem lokalen Prozessor simulieren.

Beim Starten des Programms wird zuerst die Initialisierung durchgeführt. Dabei werden die Offsets von allen Prozessoren bestimmt und anschließend die Drifts berechnet (auf dem Root-Prozessor). Danach werden die Offsets und Drifts auf die Prozessoren verteilt.

Als globale Uhr dient die cor_clock() - Routine. Sie benutzt den 64-Bit rdtsc()-Zyklenzähler, Offset und Drift und liefert eine korrigierte Zeit. Es wurden Messungen des Offset nach der Synchronisation gemacht. Die Abbildung 3 stellt die Messung über 5000 Sekunden dar. Wie man sieht, sind die Offsets nicht linear in der Zeit. Die Offsets in der Abbildung 1[b)]

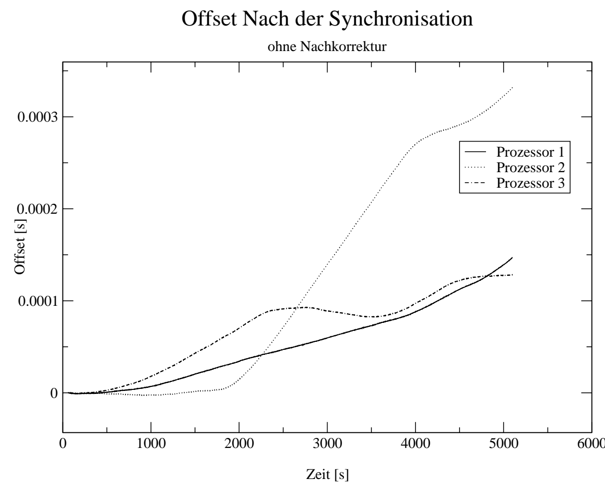


Abbildung 3. Offset nach der Korrektur

sehen dagegen linear aus. In Wirklichkeit sind sie aber nicht linear. Die Größenordnungen sind allerdings sehr unterschiedlich.

Die Offsets, die man mit Hilfe der korrigierten Uhr berechnet, bleiben für etwa 500 Sekunden unter 10 μs. Nach einer Stunde sind die Drifts aber schon bei 250 μs. Das passiert wegen der Schwankungen der Drift. Abbildung 4 zeigt die Drift von einem Prozessor, gemessen über 3000 Sekunden. Dabei wurde die Drift über 60 Sekunden gemittelt. Die Werte der Drift liegen im Bereich von Nanosekunden. Man sieht, daß die Drift über lange Zeit das gleiche Vorzeichen hat. Deswegen wird der Offset immer größer. Auf der Abbildung 3 sieht man, daß im Bereich von 0 bis etwa 2500 Sekunden die Drifts aller Prozessoren das

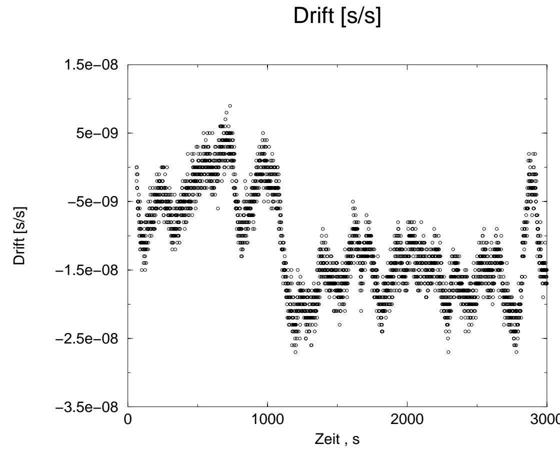


Abbildung 4. Schwankungen des Drifts

gleiche Vorzeichen haben und die Offsets wachsen. Dann wechselt die Drift des 3. Prozessors das Vorzeichen und der Offset verkleinert sich. Die Ursachen dieses Driftverhaltens sind noch nicht klar. Wir benutzen den Zyklenzähler, es müssen sich also die Taktraten der Prozessoren ändern. Die Abweichung der Drift um 25 *ns* entspricht einer Änderung des Takts um 20 Zyklen. Man vermutet, daß die Schwankungen durch Temperaturänderungen verursacht werden.

Wenn man mit dieser Genauigkeit nicht zufrieden ist, kann man die Uhren während der Zeit nachkorrigieren. Dafür wird bei jedem cor_clock()-Aufruf geprüft, wie lange die Uhren schon nicht mehr korrigiert wurden. Wenn dieses Interval größer als ein bestimmter Wert ist, wird die Korrektur aufgerufen.

Mit Hilfe dieser Prozedur kommuniziert der Prozessor mit dem Time-Server und addiert einen kleinen Term zur Drift, um die Schwankungen zu dämpfen. Sie schickt eine Nachricht zum Root-Prozessor, um ihn zu 'wecken'; danach kommunizieren diese Prozessoren. Der Time-Server bestimmt den Offset des Prozessors. Dieser Offset ist ein Fehler, der durch ungenaue Bestimmung oder Schwankung der Drift verursacht ist. Wir wollen, daß die Uhren stetig sind und daß nach X Sekunden der Offset wieder 0 wird. Wir haben: $\text{Offset}_0, \text{Drift}, t_0^{loc}$ und Fehler. Wir machen eine kleine Änderung der Formel für die korrigierte Uhr.

$$t^{gl} = (1 + \text{Drift} + \delta) \cdot (t^{loc} - t_0^{loc}) + \text{Offset} + t_0^{loc}$$

Am Anfang bei der Synchronisation wird δ auf Null gesetzt. Später bei dem Korrekturaufruf wird δ entsprechend geändert.

Zum Moment der Offsetmessung haben wir

$$t_{real}^{gl} = t_{sim}^{gl} + \text{Fehler}$$

wobei t_{real}^{gl} die Zeit am Root-Prozessor ist und t^{gl} eine simulierte Zeit. Fehler muß nach X Sekunden 0 sein.

$$\delta = \frac{\text{Fehler}}{X}$$

Die Uhr muß stetig sein. Um die Änderung der Drift zu kompensieren, ändern wir auch den Offset:

$$(1 + \text{Drift} + \delta_1)(t^{loc} - t_0^{loc}) + \text{Offset}_1 + t_0^{loc} = (1 + \text{Drift} + \delta_2)(t^{loc} - t_0^{loc}) + \text{Offset}_2 + t_0^{loc}$$

Dabei sind:

- δ_1 - alter Wert von δ
- δ_2 - neuer Wert von δ
- t^{loc} - Zeitpunkt, zu dem wir den Offset(Fehler) gemessen haben.
- Offset_1 - der alte Wert und Offset_2 - der neue Wert der Offsetkonstante.

Wir haben also 2 unbekannte Größen - Offset_2 und δ_2. Alle anderen Größen sind bekannt. Wenn man die Gleichungen löst, bekommt man

$$\begin{aligned} \delta_2 &= \frac{\text{Fehler}}{X} \\ \text{Offset}_2 = \text{Offset}_1 &+ (\delta_1 - \delta_2) \cdot (t^{loc} - t_0^{loc}) \end{aligned}$$

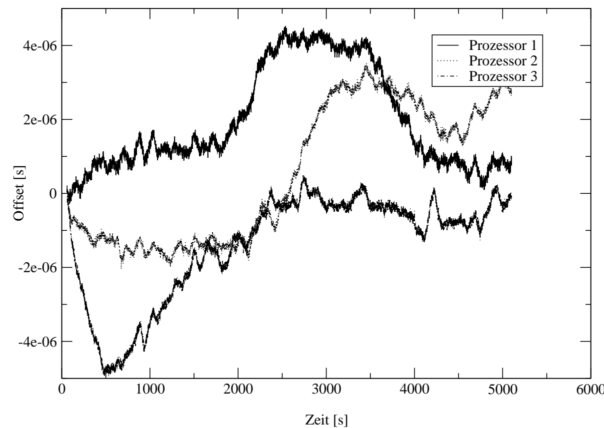


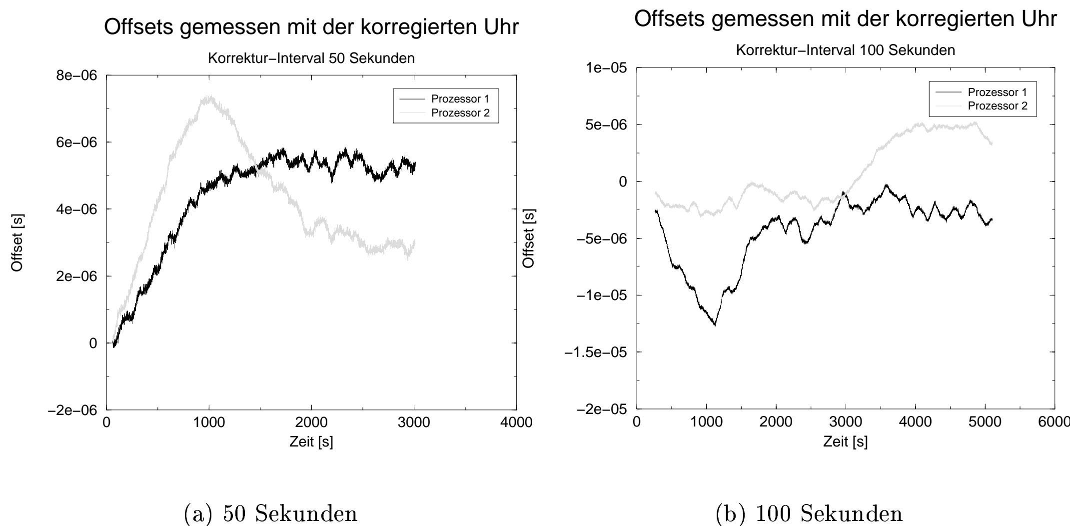
Abbildung 5. Messung des Offsets nach der Synchronisation mit der Korrektur

Mit dieser Methode ist es möglich, Offsets unter 5-10 μs über lange Zeit zu halten. Abbildung 5 zeigt das Verhalten des Offsets über 5000 Sekunden.

Wie groß kann man Intervall X wählen, ohne die Genauigkeit zu verlieren? Es wurden die Offsets gemessen mit dem Intervall 50, 100 und 200 Sekunden. Die Messergebnisse sind in der Abbildungen 6 und 7 dargestellt.

Wie man sieht, gibt es fast keinen Unterschied zwischen dem 50- und dem 100-Sekunden-Bild. In beiden Fällen bleiben die Offsets unter 10 μs über 5000 Sekunden. Bei dem Intervall von 200 Sekunden sind die Offsets schon bis zu 20 μs gewachsen. Man muß also selber entscheiden, welche Genauigkeit man braucht und damit ein entsprechendes Intervall wählen.

Es gibt aber ein Problem bei der Nachkorrektur: wenn ein Prozessor seine Uhr synchronisieren will, muß er irgendwie den Time-Server benachrichtigen. Das heißt, daß der Time-Server bereit sein muß, diese Nachricht zu bekommen. Eine normale Anwendung, die die synchronisierte Uhr benutzen will, muß sich aber nicht darum kümmern. Es gibt zwei Möglichkeiten, dieses Problem zu behandeln:



(a) 50 Sekunden

(b) 100 Sekunden

Abbildung 6. Korrektur mit verschiedenen Intervallen

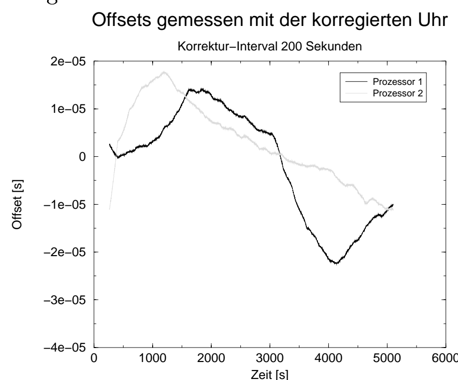


Abbildung 7. Messung mit dem Korrekturintervall 200 Sekunden

- Es gibt noch freie Myrinet Ports, die man für die Kommunikation benutzen kann. Man könnte auf der untersten Ebene mit Hilfe der GM - Bibliothek (spezielle Myrinet-Kommunikation) die Offsetbestimmung durchführen

- Ein Programm, das die Offsets bestimmt und die korrigierte Zeit liefert, im Hintergrund laufen lassen.

Die Ergebnisse:

Ich habe ein fertiges Programm, das man leicht in eine Bibliothek umwandeln kann. Sie hat zwei Funktionen: syncInit() und cor_clock(). Die syncInit() - Routine führt die Synchronisation durch. Danach kann man die cor_clock() - Routine benutzen, um die korrigierte Zeit zu bekommen. Die Nachkorrektur ist wegen Zeitmangels nicht implementiert.

Man kann die Uhren mit sehr hohen Genauigkeit synchronisieren. Die Offsets bleiben unter 10 μs über 500 Sekunden ohne Nachkorrektur. Das ist ausreichend für kleine Anwendungen.

Mit der Nachkorrektur kann man die Offsets beliebig lang unter 10 μs halten.

Literatur

1. ZAMpano - ZAM Parallel Nodes, http://zampano.zam.kfa-juelich.de
2. Network Time Protokoll, Network Working Group, http://www.eecis.udel.edu/~ntp
3. Andrew S. Tanenbaum, Distributed Operating Systems, Prentice Hall, 1994