Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Interner Bericht

Beiträge zum Wissenschaftlichen Rechnen Ergebnisse des Gaststudentenprogramms 2002 des John von Neumann-Instituts für Computing

Rüdiger Esser (Hrsg.)

FZJ-ZAM-IB-2002-12

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

Interner Bericht

Beiträge zum Wissenschaftlichen Rechnen Ergebnisse des Gaststudentenprogramms 2002 des John von Neumann-Instituts für Computing

Rüdiger Esser (Hrsg.)

FZJ-ZAM-IB-2002-12

November 2002

(letzte Änderung: 1. November 2002)

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, führte das ZAM in diesem Jahr zum dritten Mal 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 kamen für zehn Wochen, vom 5. August bis 11. Oktober 2002, ins Forschungszentrum. Sie beteiligten sich hier an den Forschungs- und Entwicklungsarbeiten des ZAM 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:

Katharina Benkert	Bernd Körfgen
Stefan Borowski	Paul Gibbon
Martin Fengler	Inge Gutheil, Thomas Müller
Ekkehard Petzold	Jörg Striegnitz
Jens Rühmkorf	Bernd Mohr
Eduard Schreiner	Dieter Bartel, Stefan Birmanns
Christoph Spanke	Karsten Scholtyssik
Vladimir Stegailov	Godehard Sutmann
Jürgen Wieferink	Holger Dachsel

Zu Beginn ihres Aufenthalts erhielten die Gaststudenten eine viertägige Einführung in die Programmierung und Nutzung der Parallelrechner im ZAM. Um den Erfahrungsaustausch untereinander zu fördern, präsentierten die Gaststudenten 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 aktuelle Forschungsarbeiten weiterzubringen - und den Betreuern, die tatkräftige Unterstützung dabei geleistet haben.

Ebenso danken wir allen, die im ZAM und der Verwaltung des Forschungszentrums bei Organisation und Durchführung des diesjährigen Gaststudentenprogramms mitgewirkt haben. Besonders hervorzuheben ist die finanzielle Unterstützung durch den Verein der Freunde und Förderer der KFA und die Firma CRAY. Es ist beabsichtigt, das erfolgreiche Programm künftig zu wiederholen, schließlich ist die Förderung des wissenschaftlichen Nachwuchses dem Forschungszentrum ein besonderes Anliegen.

Weitere Informationen über das Gaststudentenprogramm, auch die Ankündigung für das kommende Jahr, findet man unter http://www.fz-juelich.de/zam/gaststudenten.

Jülich, November 2002 Rüdiger Esser

Inhalt

Implementation of a parallel direct solver for sparse matrices on ZAMpano

Katharina Benkert

Department of Mathematics
University of Technology, Chemnitz

Email: benke@mathematik.tu-chemnitz.de

Abstract: The goal of this report is to investigate the efficiency of the parallel direct solver PSPASES on the SMP cluster ZAMpano with mixed MPI/SMP parallelism. The results obtained are compared with those on the massively parallel computer CRAY T3E. Further on, the library has been integrated in the software package FINEART (FINite element analysis using Adaptive Refinement Techniques) and the performance is compared to the existent sequential solver of FINEART.

Introduction

Various physical and technical problems like heat conduction problems, problems from solid structure mechanics, electrostatics and magnetostatics are modeled with partial differential equations. The finite element method (FEM) is one method of spatial discretization [1] which leads to a system of linear or nonlinear equations. A solver subsequently computes a discrete solution.

Mathematically, the structure to be analyzed is subdivided into a mesh of finite-sized elements of simple shape. Within each element a simple (usually linear, quadratic or cubic) ansatz is made for the physical quantity to be calculated. The degrees of freedom (DOFs) in the nodal points of the finite elements are the parameters of the ansatz (shape) functions; thus the original problem of finding the solution of partial differential equations is reduced to the discrete problem to determine the DOFs in the nodal points. In this report, we focus on static elastomechanical problems, i.e. the relevant DOFs are the nodal displacements.
Using this discretization, the equations of equilibrium are assembled in matrices which can be solved using standard numerical tools, here a solver for systems of linear equations. After applying the appropriate boundary conditions, the nodal displacements are found by solving the system of linear equations given by the global stiffness matrix and the physical loads, e.g. forces. Once the nodal displacements are known, element stresses and strains can be calculated.

Since FEM problems tend to lead to high-dimensional systems of equations, one might think of various techniques to speed up the solution process in order to enable an accurate resolution of the underlying physical phenomena. The use of multigrid, adaptive mesh refinement, or parallel computer environments are just some examples. This is, in short, the goal of the collaboration 'Finite Element Analysis of Structures on coupled SMP-Computers' of the Central Institute for Applied Mathematics (Forschungszentrum Jülich GmbH, Germany), INTES GmbH (Stuttgart, Germany), the Structural Engineering Research Centre (Chennai, India) and the NIIT (New Delhi, India) [2].

As a part of this project, the sequential solver in the FEM software FINEART has been replaced by the parallel library PSPASES. This modified version of FINEART was tested on the coupled SMP computer ZAMpano.

Computing environment

In the following, a detailed description of the hardware and software discussed in this report is given.

ZAMpano

ZAMpano [3], the abbreviation of '*ZAM pa*rallel *no*des', is a research project of the Central Institute for Applied Mathematics (ZAM) to investigate aspects of architecture, programming models and languages, performance tools and resource management in the context of SMP clusters.

It consists of eight compute nodes and one service node, each with four Intel Pentium III processors and 2 GB main memory. The nodes are connected by Fast-Ethernet as well as Myrinet (see figure 1). The latter is used as a low-latency high-transfer-rate connection for user-level message passing via GM (the communication layer for Myrinet) between the compute nodes.

The transfer rate achieved for message passing via GM is up to 120 MB/s between nodes and up to 200 MB/s for shared-memory intra-node communications at a latency of approximately 15 μsec.

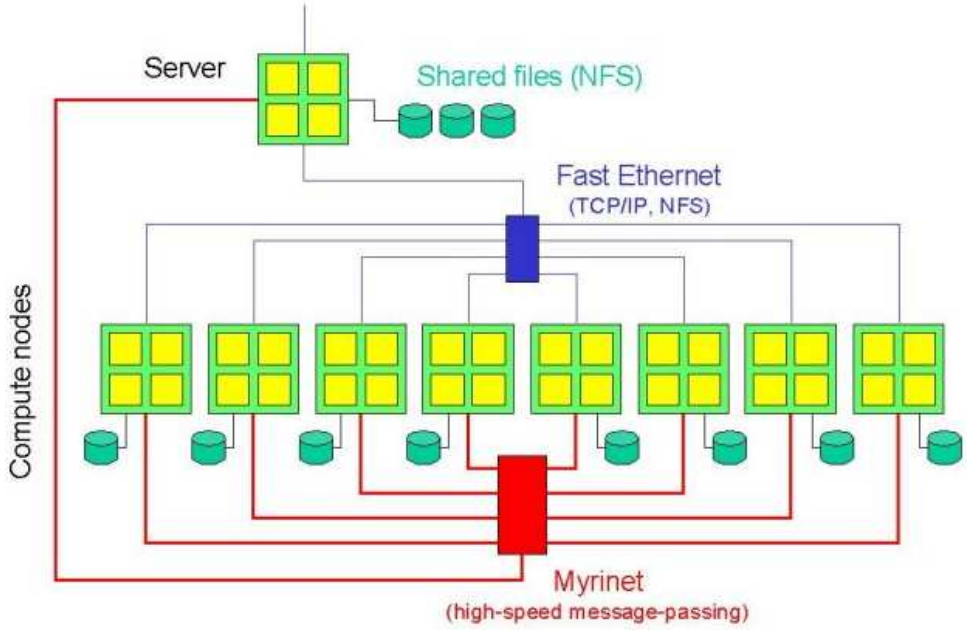


Figure 1: ZAMpano hardware confi guration

CRAY T3E

The massively parallel computer CRAY T3E-1200 is equipped with 512 processors (600 MHz) and a storage of 512 MB per processor. It uses mainly the message passing programming model. The transfer rate for internodal communication is up to 300 MB/s.

FINEART

The FEM program FINEART (FINite Element analysis using Adaptive Refinement Techniques) was developed by the Structural Engineering Research Centre (SERC), Chennai, India.

It offers an iterative calculation of FEM problems with h-adaptive mesh refinement until a given (heuristical) accuracy is reached. Each iteration consists of five steps:

- calculation of the element stiffness matrices
- assembling of the global stiffness matrix
- solution of the linear system of equations
- error estimation
- if necessary, adaptive mesh refinement

At present, the public domain solver of LASPack [4] is used which is an iterative solver for sparse symmetric matrices.

PSPASES

PSPASES (Parallel SPArse Symmetric dirEct Solver) is a MPI-based parallel library intended to solve a system of linear equations, $AX = B$, where A is a sparse symmetric positive definite matrix and B a multiple right-hand side.

PSPASES uses a direct method of solution, which consists of four consecutive stages of processing: ordering, symbolic factorization, Cholesky factorization, and triangular systems solution:

- ordering
  During the ordering phase, a permutation matrix P is computed, such that PAP^T will incur a minimal fill during the factorization phase.
- symbolic factorization
  The symbolic factorization phase determines the non-zero structure of the triangular Cholesky factor L.
- numerical factorization
  The numerical factorization step computes the Cholesky factor L that satisfies
  $$PAP^T = LL^T$$
- triangular systems solution
  The triangular system solution of $Ax = b$ is realized by solving two triangular systems viz. $Ly = b'$ followed by $L^T x' = y$, where $b' = Pb$ and $x' = Px$.

The final solution x is obtained using $x = P^T x'$ [5].

Each of these phases is implemented using scalable and high-performance algorithms developed by Mahesh Joshi, George Karypis, and Vipin Kumar [6].

The library is public domain and can be obtained from the authors' web page [7]. PSPASES can be used on any parallel computer or network of workstations equipped with MPI and BLAS libraries as well as Fortran-90 and C language compilers. Nevertheless, the algorithms used in PSPASES restrict the number of processors p to be a power of 2 satisfying $p \geq 2$.

Benchmarks of PSPASES

Installation

The installation on CRAY T3E was smooth, whereas the installation of PSPASES on ZAMpano required some changes to the source code of PSPASES due to interlanguage calling problems between C and Fortran-90 routines. They were caused by the PGI compilers which added underscores to the function names of interlanguage calls.

For Fortran-90 routines calling C routines, the problem was solved using the C() pragma directive. It takes the names of external functions as arguments and specifies that these functions are written in C language, so the Fortran Compiler does not append an underscore as it ordinarily does with external names. The C() directive for a particular function must appear before the first reference to that function. It must also appear in each subprogram that contains such a reference [8]. The conventional usage is:

```
EXTERNAL ABC, XYZ
!$PRAGMA C(ABC, XYZ)
```

For C routines calling Fortran-90 routines, a header file named underscore.h was added to each C function possessing Fortran-90 calls. It contains a number of #define statements, for instance:

```
#define porder porder_
#define emovea emovea_
...
```

Testing

Three test programs for binary, fcc, Harwell-Boeing and Rutherford-Boeing matrix formats (cf. [9, 10]) are included in the distribution of PSPASES. Each test program reads the matrix input file and converts the matrix from symmetric storage mode, which takes advantage of the symmetry, to full storage mode. Afterwards, the PSPASES functions with a randomly generated right-hand side are called.

The performance was tested with sparse symmetric positive definite matrices from 'Matrix Market', an Internet site with test data from a variety of applications [11]. Mainly, the following matrices were used:

name	size	entries	diagonal NNZE	NNZE below diagonal
bcsstk16	4884	147631	4884	142747
bcsstk18	11948	80519	11948	68571
s3dkt3m2	90449	1921955	90449	1831506
s3dkq4m2	90449	2455670	90449	2455670

where entries refers to the number of non-zero elements (NNZE) on the diagonal and in the lower triangular part of the matrix.

Optimization

To optimize the runtime performance of PSPASES, installation and runtime options have to be taken into consideration.

Installation options

During installation, one has to decide which BLAS routines, LAPACK routines and compilers to use.

On ZAMpano, the optimized ATLAS-BLAS routines performed much better than the BLAS libraries distributed with the compiler. For the bcsstk25-matrix, the use of the ATLAS-BLAS routines gave a speedup of 20% on two processors.

PSPASES uses one LAPACK routine DPOTRF (version 2.0) which computes the Cholesky factorization of a real symmetric positive definite matrix and which is distributed with the library. Linking the LAPACK 3.0 library instead did not make any difference.

The compilers used are the GNU C compiler and the PGI compilers with the default optimization level O2 and O1.

Runtime options

The runtime behavior of PSPASES can be influenced by the distribution of processes on ZAMpano (see below) as well as by the internal parameters block size, ordering method, and run option.

Figure 2 shows the dependence of the solver time, this means the computational time used by PSPASES functions, on the *process distribution* on ZAMpano. Four processes are distributed on n nodes where p processors are utilized. If multiple processes run on the same node, the runtime with and without shared memory (SM) is measured.

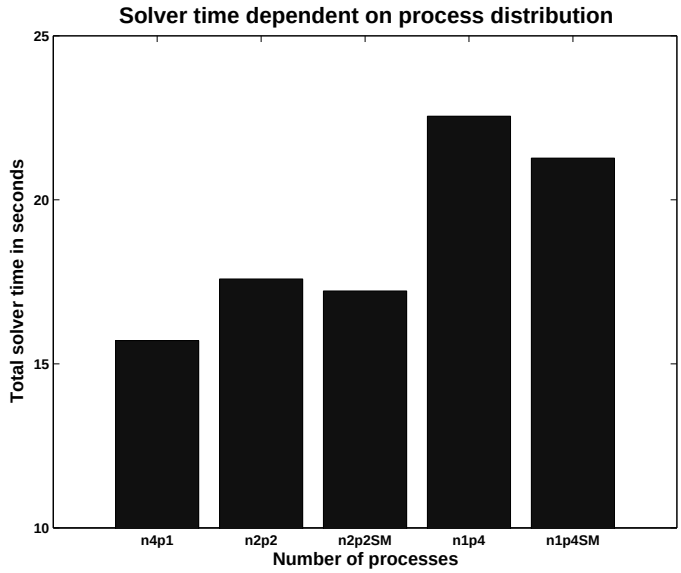


Figure 2: Solver time depending on process distribution for matrix s3dkt3m2 on ZAMpano with parallel ordering

The distribution of processes on ZAMpano favors a wide-spread distribution. As shown, the use of shared memory is accelerating the solution process, but since the calculations are memory-intensive, cache effects are slowing down the whole process compared with the computation on different nodes. For smaller matrices, the differences are less profound. The transfer rates up to 200 MB/s via GM mentioned above are only valid for two processes exchanging small data packages.

The *block size* is a parameter which determines the internal matrix distribution in PSPASES. Tests show (cf. figure 3), that the default value of 64 is a good choice in most cases. Alternatively, a block size of 32 could be applied.

The *ordering* step of PSPASES uses the libraries METIS [12] for serial or ParMETIS [13] for parallel ordering. Figure 4 shows the solver time depending on the ordering method, as for all four matrix market

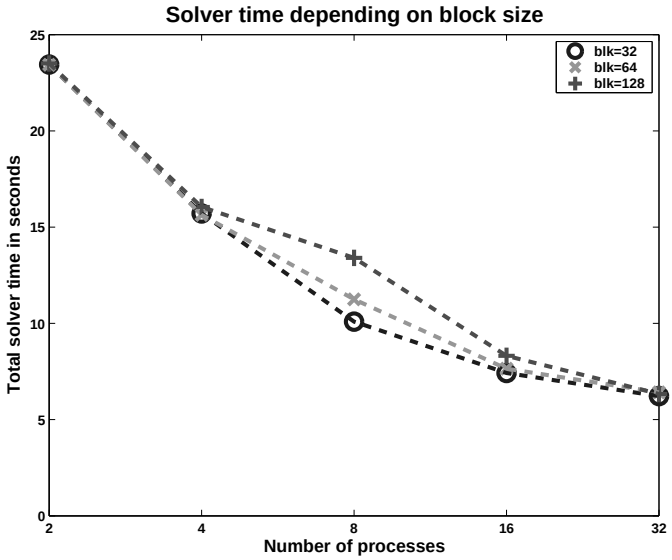


Figure 3: Solver time depending on block size for matrix s3dkt3m2 on ZAMpano with parallel ordering

matrices (that means up to a size of 100000) serial ordering was faster. Parallel ordering becomes only faster for really large problems.

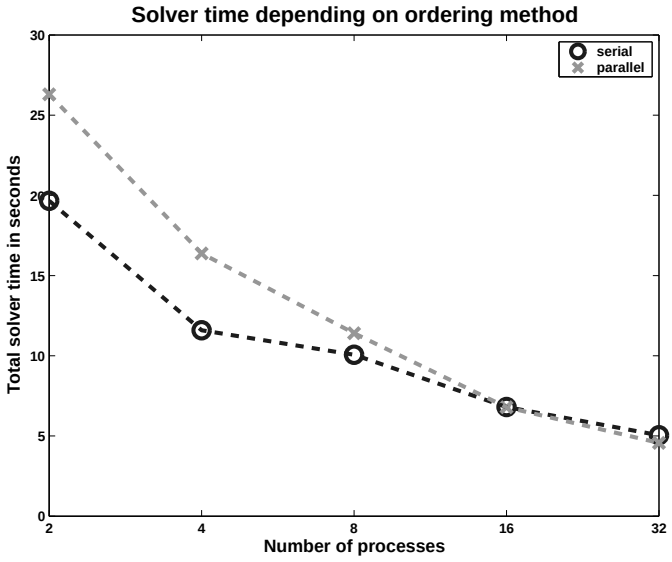


Figure 4: Solver time depending on ordering method for matrix s3dkq4m2 on T3E with block size 32

The calling sequence of the PSPASES function can be controlled by the parameter *run option*. One has the choice to call all stages of PSPASES one by one, or to use a bundle function which includes ordering, as well as symbolic and numerical factorization. The explicit call of every stage is advantageous if sparse matrices with the same non-zero structure are available, but for the presented measurements, the faster bundle function is used.

Comparison with CRAY T3E

The calculations on CRAY T3E showed the same qualitative behavior as on ZAMpano, as regards the parameters block size, ordering method and runtime option. The quantitative behavior indicated somewhat shorter solver times on the T3E.

A qualitative difference (cf. figure 5) was a better speedup on T3E originating from faster network communications.

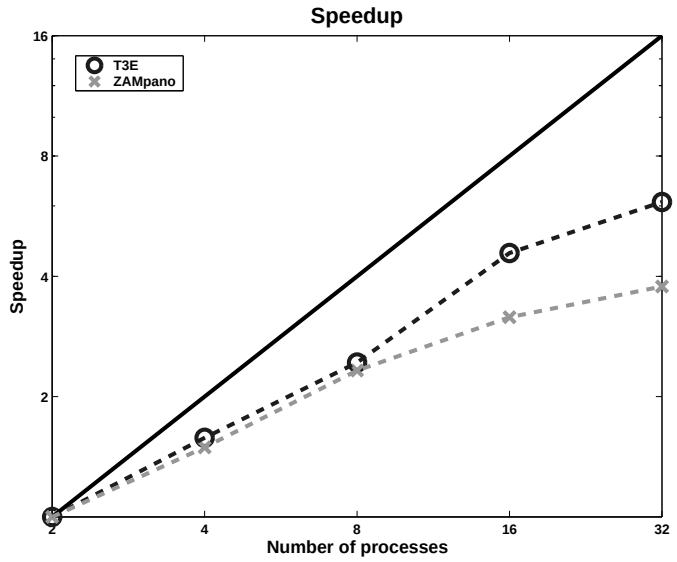


Figure 5: Speedup for matrix s3dkt3m2 with parallel ordering and block size 32

Integration in FINEART

To use PSPASES from FINEART a MPI program, similar to the test programs distributed with the library, named PARSOLV had to be implemented. As the different FINEART modules communicate through files, the input routines had to be adapted to the FINEART files. Furthermore, the gathering of the solution vector and output routines had to be added. The main program was written in Fortran 90, whereas the input routine for the global stiffness matrix and the output routines were written in C.

The implementation was tested with models of the stresses caused by forces on a wrench and a L-domain with different mesh sizes (cf. figures 6, 7). For visualization, the tool RAPS (Räumliches Aska Plot System) [14] was used.

Figure 8 shows the runtime of the parallel program PARSOLV for the second level of refinement subject to different numbers of processes. The parallel program was bracketed by C functions to measure the 'outer' total runtime. The inner runtime was measured right after the MPI_INIT() and shortly before the MPI_FINALIZE(). The difference between outer and inner runtime is an approximation for the time spent setting up the parallel computing environment. The runtime without PSPASES is used for reading the input files, distributing the matrix and the right-hand side, converting the matrix and writing the output files.

One can observe an increase of the runtime with higher process numbers due to the growing amount of time for the library call as well as for MPI initialization and finalization. This indicates that the considered problems are too small to take full advantage of a high degree of parallization. The calculation of larger

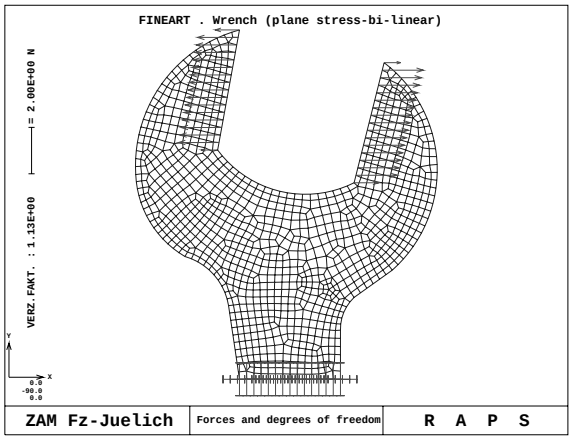


Figure 6: Forces and DOFs of wrench example

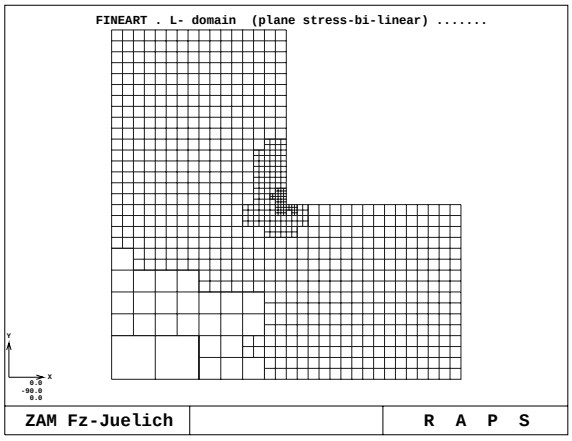


Figure 7: L-domain with adaptive mesh

problems is not possible at present because of problems in the FINEART assembly module.

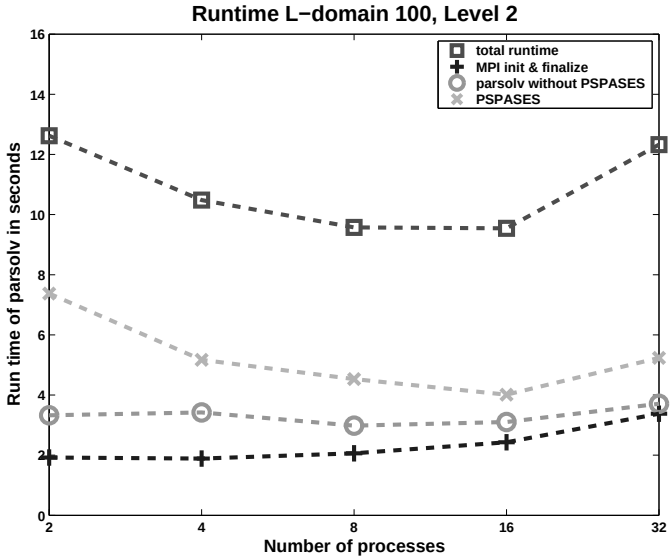


Figure 8: Solver time of the L-domain example in the second level of refi nement ($n = 68312$, $nnze = 643955$)

Nevertheless, the results compared to the sequential solver are quite encouraging. This is not due to parallization, but rather to the algorithms implemented in PSPASES. Figure 9 shows a noticeably improvement for only two processors.

Perspectives

As mentioned, the problem size of FINEART is currently restricted. To simulate the future occurrence of bigger problem sizes, symmetric positive definite sparse matrices up to a dimension of one million were generated. In this case, we observe a more convincing speedup as shown in figure 10.

To conclude, the use of PSPASES benefits at present from the algorithmic progress, but a high degree of parallization will be reserved to larger problem sizes in the future.

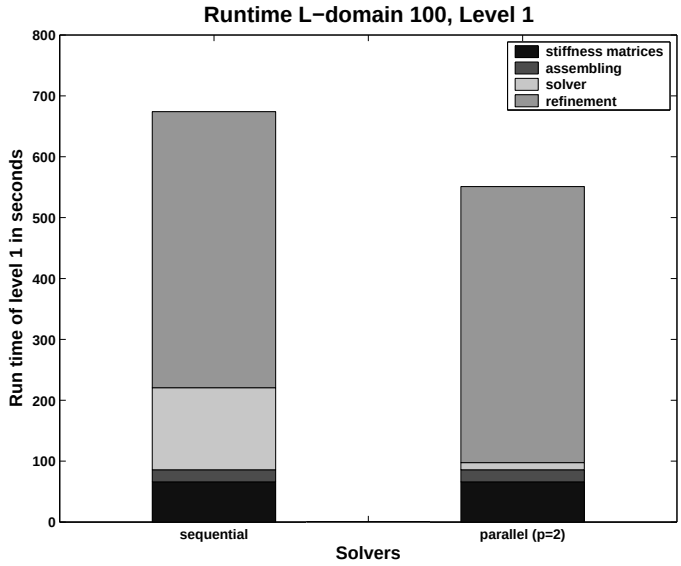


Figure 9: Runtime on L-domain example in the first iteration step ($n = 65842$, $nnze = 620422$)

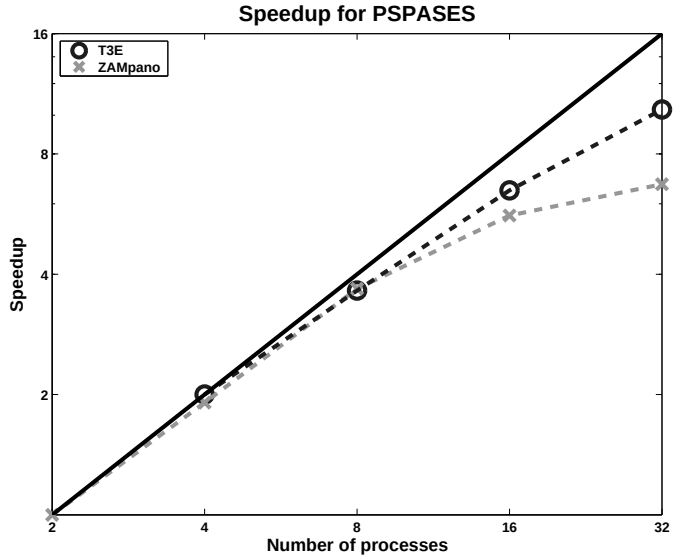


Figure 10: Sparse test matrix with $n = 1000000$, $nnze = 4,00 * 10^6$, calculated with parallel ordering and block size 64

References

1. H. R. SCHWARZ, *Methode der Finiten Elemente,* Teubner Studienbücher, 1991
2. *web page of the project 'Finite Element Analysis of Structures on coupled SMP-Computers',* http://www.fz-juelich.de/zam/RD/coop/feast-smp/index.html
3. *ZAMpano homepage,* http://zampano.zam.kfa-juelich.de/
4. *web page of LASPack,* http://casper.cs.yale.edu/mgnet/www/mgnet/Codes/laspack/
5. MAHESH JOSHI, GEORGE KARYPIS, VIPIN KUMAR, ANSHUL GUPTA, FRED GUSTAVSON, *PSPASES: An Efficient and Scalable Parallel Sparse Direct Solver*
6. MAHESH JOSHI, GEORGE KARYPIS, VIPIN KUMAR, *PSPASES: Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Linear Systems, User's Manual (version 1.0.3.),* 1999
7. *PSPASES homepage,* http://www-users.cs.umn.edu/~mjoshi/pspases/
8. *Fortran Programmer's Guide,*

http://www.ictp.trieste.it/~manuals/programming/sun/fortran/prog_guide/
9. MAHESH JOSHI, *README.USAGE (included in the PSPASES distribution)*
10. IAIN S. DUFF, ROGER G. GRIMES, JOHN G. LEWIS, *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*, 1992, available via ftp://ftp.cerfacs.fr/pub/harwell_boeing/userguide.ps.Z
11. *Matrix Market homepage*, http://math.nist.gov/MatrixMarket/
12. *METIS homepage*, http://www-users.cs.umn.edu/~karypis/metis/metis/
13. *ParMETIS homepage*, http://www-users.cs.umn.edu/~karypis/metis/parmetis/
14. *web page of RAPS*, http://www.fz-juelich.de/zam/CompServ/software/vislab/software/raps/raps_e.html
15. HENNING NIEHOFF, *Implementierung eines parallelen Gleichungssystemlösers in das Finite-Elemente-Programm FINEART*, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich, 2002

Appendix: FINEART files

As mentioned, the different FINEART modules communicate by file exchange. The following list should give a short overview over the different files involved in the solver calls [15].

- project1
  This file contains the name of the project and the refinement level. This information is used to create specific names which consist of the project name *proj*, an underscore, the letter 'l' followed by the level number *m* and the appropriate ending: .dat, .im, .solv or .out.

- *proj*_l*m*.dat
  Primarily, the number of nodes and the dimension are specified in the file which gives the total number of possible DOFs. Then, the node numbers are linked to the DOFs of the linear system of equations. Zeros in the sequence of DOFs indicate boundary conditions which diminish the number of possible DOFs. Afterwards, the values for the right-hand side are given.

- *proj*_l*m*.solv
  The first line contains the row pointers and the second the column indices to the values of the global stiffness matrix in the file gstif.

- *proj*_l*m*.out
  After two header lines, all node numbers together with the nodal displacement for each degree of freedom are listed.

- gstif
  The file contains the values of the global stiffness matrix of the current level in binary format.

Porting a parallel implementation of multi-dimensional quantum dynamics from SHMEM to MPI

Stefan Borowski

Fritz-Haber-Institut der Max-Planck-Gesellschaft, Abteilung Chemische Physik
Faradayweg 4-6, 14195 Berlin

E-mail: borowski@fhi-berlin.mpg.de

Abstract:

Starting from a recently developed SHMEM parallelization of a multi-dimensional quantum dynamics code [1], an alternative parallel implementation utilizing MPI is proposed. The applied quantum dynamical method using a pseudospectral algorithm is introduced for a general 4D Hamiltonian. The proven data decomposition in all dimensions is only briefly reviewed, since it is adopted to the MPI implementation without changes. An efficient communication scheme avoiding strided data communication on multi-dimensional arrays is presented. Specific details on both the MPI and the SHMEM communication are discussed. In a speedup analysis, the new MPI and the original SHMEM implementation are compared with respect to their performance outcome. A perspective to further optimization and extension of the MPI implementation is given.

Introduction

A rigorous treatment of quantum dynamical phenomena requires the solution of the time-dependent Schrödinger equation. While low-dimensional quantum dynamics can be performed routinely on workstations, high-dimensional calculations still remain a challenge. This is due to the exponential scaling behavior with the number of degrees of freedom f taken into account. For a general basis representation of the extent N, the overall problem size amounts to N^f. The processing of such huge data objects results in an exhausting numerical effort even for semi-linearly scaling algorithms [2]. Therefore, accurate quantum dynamical studies are restricted to few dimensions only.

Computing power and memory requirements of high-dimensional studies exceed the capabilities of single workstations by far. Consequently, a great need for efficient parallel implementations has developed over the past years. Characterized by highly scalable algorithms and a large range of problem size, quantum dynamical studies are well suited for exploitation of parallel computing power. This was recently demonstrated by a successful parallelization of a multi-dimensional quantum dynamics code using SHMEM message passing [1]. The present report introduces a MPI implementation of this parallelization concept. The report is organized as follows: In the first section, we give a characterization of the general quantum dynamical problem and an introduction to pseudospectral algorithms. The second section discusses the parallelization strategy and compares its MPI and SHMEM realization. The third section makes a comparison of the parallel performance of the MPI and the SHMEM implementation in a speedup analysis. Finally, an outlook to future developments is shown in the last section.

Theoretical Fundamentals

The quantum dynamical problem

The dynamics of a quantum mechanical system is described by the time-dependent Schödinger equation

$$\mathrm{i}\frac{\partial}{\partial t}\Psi(t) = \hat{H}\Psi(t), \tag{1}$$

which can formally be solved

$$\Psi(t+dt) = \hat{U}(dt)\Psi(t) \quad \text{with} \quad \hat{U}(t') = \exp(-i\hat{H}t') \tag{2}$$

by introducing the time evolution operator $\hat{U}(t)$. For time-dependent systems, the time evolution operator is restricted to short time steps in order to sample the time dependence of the Hamiltonian. However, the dynamics of a time-independent Hamiltonian can be characterized by both short-time and global propagation methods because of its trivial time dependence. In literature, a variety of propagation schemes [3] is found that enables a problem-adapted modelling of the time evolution operator. For the implementation on hand, the Split propagator [4] and the Chebychev propagator [5] are realized as short-time and global propagation schemes, respectively.

The essential computationally relevant operation in all propagation methods is the action of the Hamiltonian $\hat{H}$ upon the wavefunction $\Psi(t)$ [3]. Therefore, the representation of the Hamiltonian and its efficient operation upon the wavefunction will be the focus of the following consideration.

Pseudospectral algorithms

A general 4D Hamiltonian including two radial coordinates $\{X, Y\}$ and two rotational coordinates $\{\theta, \phi\}$ is given by

$$\hat{H} = \hat{T}^{\mathrm{rad}}(\hat{K}_X, \hat{K}_Y) + \hat{T}^{\mathrm{rot}}(\hat{\mathbf{J}}, X, Y) + \hat{V}(X, Y, \theta, \phi) \tag{3}$$

$$\hat{T}^{\mathrm{rad}} = \frac{\hat{K}_X^2}{2M_1} + \frac{\hat{K}_Y^2}{2M_2} \qquad \hat{T}^{\mathrm{rot}} = \frac{\hat{J}^2}{2I(M_1, M_2, \mu, X, Y)}$$

with the radial momentum operators $\{\hat{K}_X, \hat{K}_Y\}$ and the angular momentum operator $\hat{\mathbf{J}}$. The radial kinetic energy operator $\hat{T}^{\mathrm{rad}}$ describes the motion of the masses M_1 and M_2 along the coordinates X and Y, respectively. The rotational kinetic operator $\hat{T}^{\mathrm{rot}}$ characterizes a rotation with the moment of inertia I that can depend on the masses M_1 and M_2, the reduced mass μ and the radial coordinates $\{X, Y\}$. As a concrete instance of such a 4D Hamiltonian, we invoke the dynamics of a diatomic molecule on a surface or the dynamics of a triatomic system in the gas phase with vanishing total angular momentum.

For the representation of the wavefunction and the Hamiltonian, a common basis expansion is not desirable. The action of the Hamiltonian upon the wavefunction would scale quadratically with the overall system size according to a matrix-vector multiplication. In multi-dimensional studies, this prodigal numerical effort could hardly be managed even by supercomputers. Therefore, we favour pseudospectral algorithms [2, 6] that approach linear scaling with the overall system size. This can be achieved by using two conjugated representations of the wavefunction

$$\Psi(X,Y,\theta,\phi,t) = \begin{cases} \sum_{ikjm} \Psi^{\mathrm{FBR}}_{ikjm}(t)\Phi_{ik}(X,Y)Y_{jm}(\theta,\phi) \\ \\ \sum_{\alpha\beta\gamma\delta} \Psi^{\mathrm{DVR}}_{\alpha\beta\gamma\delta}(t)\delta_{\alpha\beta}(X,Y)\delta_{\gamma\delta}(\theta,\phi) \end{cases} \tag{5}$$

that diagonalize several operators of the Hamiltonian. The *Finite Basis Representation (FBR)* is an expansion in plane waves $\{\Phi_{ik}\}$ for the radial coordinates and in spherical harmonics $\{Y_{jm}\}$ for the rotational coordinates. The *Discrete Variable Representation (DVR)* consists of spatial grid representations in the different coordinates. For the radial coordinates a common equidistant Fourier grid $\{\delta_{\alpha\beta}\}$ is employed. For the rotational coordinates the grid representation $\{\delta_{\gamma\delta}\}$ is composed of a Gauss-Legendre grid in θ and an equidistant Fourier grid in ϕ [6]. In a pure FBR picture, the radial kinetic energy operator $\hat{T}^{\mathrm{rad}}$ is diagonal. If we choose the DVR for the radial coordinates and the FBR for rotational coordinates, the rotational kinetic energy operator $\hat{T}^{\mathrm{rot}}$ becomes diagonal. Finally, the potential energy operator is diagonal in the pure DVR picture. Consequently, in the corresponding representations the action of the different operators upon the wavefunction can be performed as a linearly scaling vector-vector multiplication. The link between the FBR and the DVR in the different dimensions is given by fast transforms:

$$\begin{array}{lccc} T^{\mathrm{rad}}_{ikjm} \cdot \backslash & \Psi^{\mathrm{FBR}} & & \Psi^{\mathrm{DVR}} \quad / \cdot V_{\alpha\beta\gamma\delta} \\ & i & \leftarrow \mathrm{FFT} \rightarrow & \alpha \\ & k & \leftarrow \mathrm{FFT} \rightarrow & \beta \\ T^{\mathrm{rot}}_{\alpha\beta jm} \cdot \backslash & j & \leftarrow \mathrm{GLT} \rightarrow & \gamma \\ & m & \leftarrow \mathrm{FFT} \rightarrow & \delta \end{array} \tag{6}$$

The Fast Fourier Transform (FFT) in $\{X,Y\}$ and ϕ scales semi-linearly $N \log N$ with the dimension extent whereas the Gauss-Legendre Transform (GLT) in θ shows the quadratic scaling relation of a matrix-vector multiplication. In practical applications, the semi-linear scaling of the FFTs dominates the quadaratic scaling of the Gauss-Legendre Transform due to the relatively small extent of the rotational basis in θ (shown in the third section). Finally, the action of the Hamiltonian upon the wavefunction shows an overall semi-linear scaling.

Parallelization

The multi-dimensional quantum dynamics method introduced in the previous section requires an enormous amount of CPU time and memory. In order to master these demands, a massively parallel implementation utilizing the SHMEM message passing library has recently been proposed [1]. However, the SHMEM communication interface is restricted to the manufacturers Cray and SGI. To get access to all manufacturers, an implementation utilizing the standard MPI communication interface has been developed. In the following section, the proven parallelization strategy is reviewed and a comparision of the MPI and the SHMEM communication is presented.

Multi-dimensional data decomposition

For the action of the Hamiltonian upon the wavefunction, the two main operations are the kinetic and potential energy operations in the corresponding representation and the FBR-DVR transforms to change between the different representations of the wavefunction. The kinetic and potential energy operations are local operations on any decomposed data structure because of the element-by-element character of the

vector-vector multiplication. However, the FBR-DVR transforms necessitate the whole global vector to be transformed. Due to this global character, communication on the data distribution has to be performed. Hence, first of all the desired data decomposition should provide a perfect load balancing for kinetic and potential energy operations. Furthermore, it should raise only minimal communication overhead for the FBR-DVR transforms.

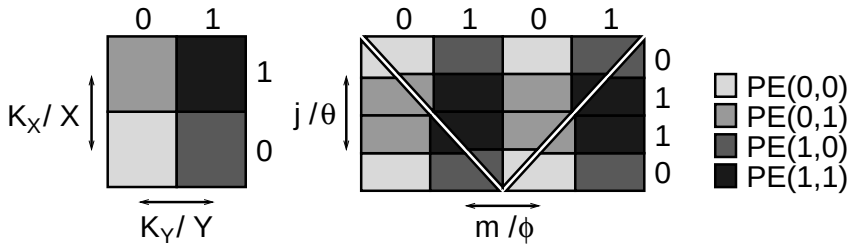


Figure 1: Perfect load balancing for computational effort of locally acting kinetic/potential energy operations: Example of a 2×2 PE grid for the radial coordinates $\{X, Y\}$ (left panel) and the rotational coordinates $\{\theta, \phi\}$ (right panel), respectively. Data structures located on a certain PE are labelled by colored areas. The FBR data $\{K_X, K_Y, j, m\}$ is limited to the central $\{j, m\}$ triangle only whereas the DVR data $\{X, Y, \theta, \phi\}$ includes the whole $\{\theta, \phi\}$ rectangle. The color scheme indicates that all PEs process the same amount of data in both the FBR and the DVR.

To obtain a flexible implementation for a variety of applications, the data is distributed among the processing elements (PEs) in all degrees of freedom as shown in Fig. 1. This data decomposition is used for both the MPI and the SHMEM implementation. Since the cartesian coordinates have the same data structure within FBR and DVR, a simple block distribution already provides an optimal load balancing (see left panel of Fig. 1). For the rotational coordinates, a more sophisticated data decomposition is neccessary. The data shape of the rotational FBR basis is the central $\{j, m\}$ triangle in the right panel of Fig. 1, which is characterized by the multiplicity of the j states with respect to their projection m. However, the data object of the rotational DVR grid is the whole $\{\theta, \phi\}$ rectangle as the spatial grid in ϕ has to resolve the largest projection of the angular momentum. Hence, a data distribution is developed that ensures optimal load balancing in both the FBR and the DVR data (see right panel of Fig. 1).

This data decomposition optimally meets the above mentioned requirements: A perfect load balancing for the kinetic and potential energy operation is achieved for both radial and rotational coordinates as shown by the color scheme in Fig. 1. Furthermore, the FBR-DVR transforms on the data decomposed structure need only few communication. For a more detailed description of the data desomposition we refer to [1].

Communication scheme

The main communication on the data distribution amounts to the preperation and redistribution of the global data before and after the FBR-DVR transforms. To transform a certain coordinate, the global dimension is collected on all involved PEs while the lost parallelism is recovered by a work decomposition in another coordinate (gathering). Then, the transformation of the global dimension is performed on the work decomposed structure. By redistributing the transformed global dimension among the involved PEs, the work decomposition is finished (scattering). An illustration of this procedure is given in Fig. 2 for a 3D slice of the whole 4D wavefunction array $\psi(X, Y, \phi, \theta)$[1] in the case of SHMEM communication.

[1]The order of the FORTRAN array dimensions is due to the following guideline: rapidly transforming dimensions of large extent first to guarantee optimal data access.

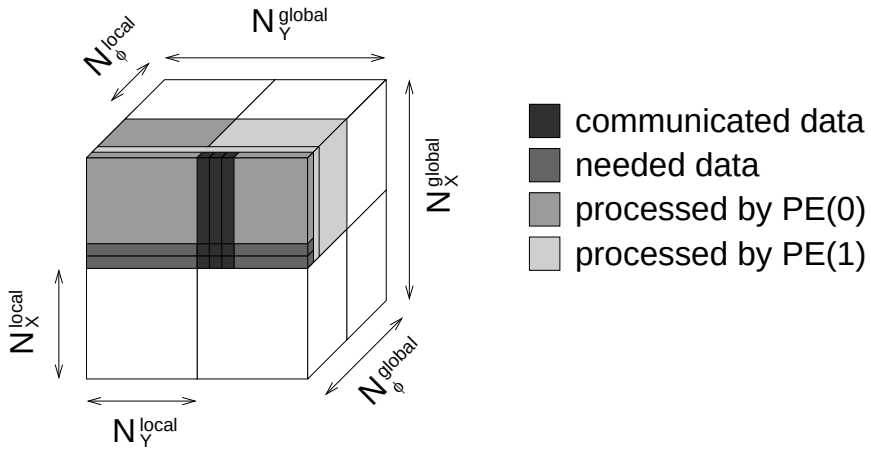


Figure 2: Perfect load balancing for communication overhead and computational effort of globally acting FBR-DVR transforms: Example of a 3D wavefunction $\Psi(X, Y, \phi)$ that is distributed on a $2 \times 2 \times 2$ PE cube. For the FFT in Y, communicated and needed data as well as PEs the data is processed by are indicated by the color scheme. A detailed description is given in the text below.

The global wavefunction of the size $N_X^{\mathrm{global}} \times N_Y^{\mathrm{global}} \times N_\phi^{\mathrm{global}}$ is distributed on a $2 \times 2 \times 2$ PE cube resulting in local wavefunctions of the size $N_X^{\mathrm{local}} \times N_Y^{\mathrm{local}} \times N_\phi^{\mathrm{local}}$. To perform a FFT in Y, a direct communication to build global vectors in Y is not desirable because the second dimension Y possesses the stride of the first dimension X. Therefore, a 2D array, which is local in X and global in Y, is collected by communicating local vectors in X. In order to decompose the work, every PE collects and transforms another 2D array with respect to the index of the third dimension ϕ. After the FFT on the 2D array is completed, the original data distribution is restored by the corresponding inverse communication.

This communication scheme avoids strided data communication even if global data of the back dimensions is needed. All communication is done in the first dimension, the global data is built up in the dimension of interest and the work is decomposed in another dimension. Since a nearly maximal communication bandwidth is already reached with quite small messages in the case of SHMEM communication, the vectors of the first dimension are sufficient as communicated data structure. However, for MPI communicaton arrays of the first two dimensions are necessary as communicated data objects to reach a reasonable communication bandwidth. This communication of arrays introduces a small local reordering effort for the processing of the first dimension. The gather and scatter communication is realized by MPI_GATHERV/MPI_SCATTERV and SHMEM_GET/SHMEM_PUT for the MPI and the SHMEM implementation, respectively.

Speedup Analysis

In this section, the parallel performance of the MPI and the SHMEM implementation are compared. To this purpose, a 4D propagation of varying system size is performed on a varying number of PEs of the Cray T3E 600. Then, the dependence of the run time $t_{\mathrm{run}}(N, P)$ on both the system size N and the number of PEs P is characterized in a speedup analysis. All timing calculations are performed by using the Split propagator. Since the FBR-DVR transforms are the essential communication-intensive part of the Split propagator as well as of the Chebychev propagator, the performance results are representative

of both propagation schemes.

Fixed size speedup

First of all, we fix the 4D system $X \times Y \times \phi \times \theta$ to the medium size $128 \times 128 \times 64 \times 32$ in order to study the behavior as the number of PEs varies. The resulting fixed size speedup $t_{\rm run}(N,1)/t_{\rm run}(N,P)$ [7] shown in Fig. 3 is almost linear for both implementations, which indicates a very high computation to communication ratio. As parallel efficiency the slope of the curves, is determined. The resulting efficiency up to 512 PEs amounts to 73% for the MPI and to 89% for the SHMEM implementation, respectively. The efficiency difference of 16% between the two implementations is due to the additional overhead caused by the change from one-sided SHMEM to two-sided MPI communication: The SHMEM communication interface provides the feature that a calling PE can read from or write to the memory of a remote PE without participation of the remote PE. However, a MPI point-to-point communication requires the participation of both the source and the destination PE.

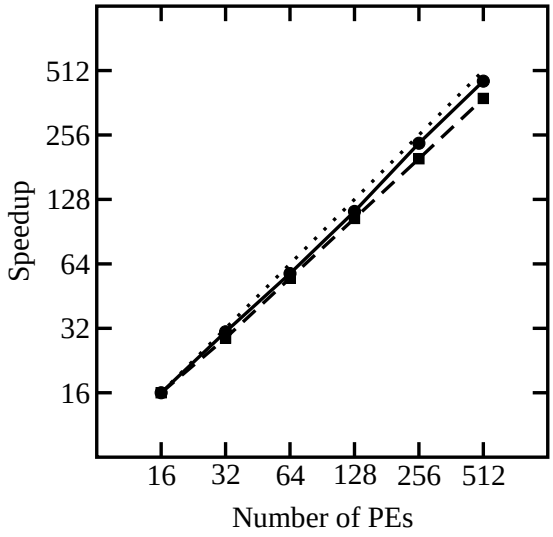


Figure 3: Fixed size speedup for a 4D system $X \times Y \times \phi \times \theta$ of the size $128 \times 128 \times 64 \times 32$. The performance of the implementations using SHMEM (solid line) and MPI (dashed line) is compared to the linear speedup (dotted line).

Usual 4D production calculation are performed on 128 PEs of the Cray T3E 600. The calculation of a 1.5 ps quantum trajectory takes about 10 hours to propagate a 5 GByte wavefunction of the size $N = 3.4 \times 10^8$. Most of the run time, about 90%, is spent on the propagation scheme, 80% on the FBR-DVR transforms alone. The remaining time is spent initializing data, computing observables and writing data.

Scaled speedup

To investigate the scaling properties in a certain dimension, we successively double the extent and the number of PEs in the dimension of interest while keeping the overall size per PE fixed to $64 \times 64 \times 32 \times 16$. For the corresponding scaled speedup $Pt_{\rm run}(N,1)/t_{\rm run}(PN,P)$ [7] a linear scaling is assumed according to the anticipated scaling relation.

Fig. 4 shows the scaled speedup for the two radial coordinates involving FFTs. For all curves, a nearly linear behavior is observed corresponding to the semi-linear scaling relation of the FFT. The performance for the coordinate X (see right graph in Fig. 4) is very close to the linear speedup for both implementations because of the optimal data access in the first dimension. The slightly smaller slope for the MPI

implementation is caused by the additional local reordering effort even in the first dimension. For the coordinate Y (see right graph in Fig. 4), the slope of the scaled speedup ist smaller compared to the performance for the coordinate X. This is the consequence of the local reordering overhead of the strided data in the second dimension. Since this data reordering is similar for the MPI and the SHMEM implementation, the slope of the scaled speedup is nearly the same for both implementations.

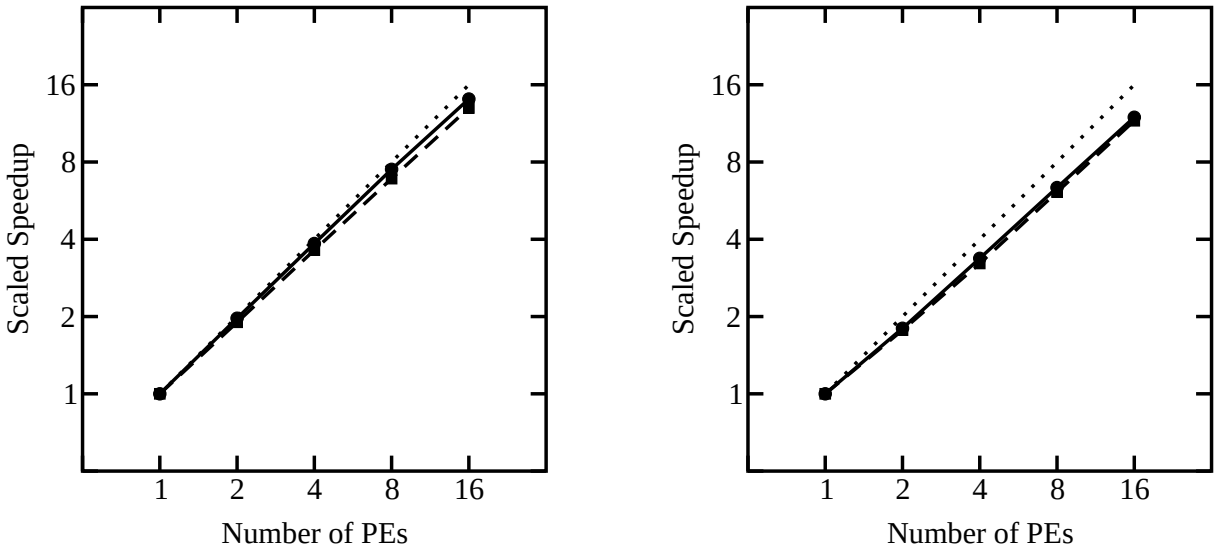


Figure 4: Scaled speedup for a 4D system $X \times Y \times \phi \times \theta$ of the size $64 \times 64 \times 32 \times 16$ per PE in the radial coordinates X (left panel) and Y (right panel). The performance of the implementations using SHMEM (solid line) and MPI (dashed line) is compared to the linear speedup (dotted line).

For the closely connected rotational coordinates, a joint scaled speedup is shown in Fig. 5. For the two implementations, a very similar performance behavior is observed. The initial performance is governed by the semi-linear scaling of the FFT in ϕ. With increasing extent of the two rotational dimensions, the quadratic scaling of the Gauss Legendre Transform in θ starts to compete. The resulting speedup drops slowly behind the supposed linear behavior more and more, due to the quadratic scaling of the Gauss Legendre Transform. However, the asymptotic quadratic scaling is irrelevant in practical applications since the computational effort of the Gauss Legendre Transform in θ only equals that of the FFT in ϕ. Hence, the quadratic scaling does not even dominate the semi-linear scaling within the transform of the rotational coordinates. This proves the anticipated semi-linear total scaling of the Hamiltonian.

In the case of two and eight PEs, the scaled speedup for the MPI implementation is slightly smaller compared to the behavior for the SHMEM implementation. This is caused by a difference between the two implementations concerning the communication for the Gauss-Legendre Transform in θ. In the SHMEM implementation, the communication of FBR data is restricted to the central $\{j, m\}$ triangle (see right panel of Fig. 1), which can readily be managed by SHMEM_GET and SHMEM_PUT. However, in the MPI implementation the whole $\{j, m\}$ rectangle including dummy data outside the central triangle is communicated by MPI_GATHERV and MPI_SCATTERV. For this collective communication, a distinction between the relevant and the dummy data is rejected since it would cause a strong asymmetry in the communication. A solution for this problem is proposed in the next section that addresses future improvements of the MPI implementation.

Further optimization and extension

Of course, the development of the presented MPI implementation is not yet completed. To proceed the MPI implementation further we will:

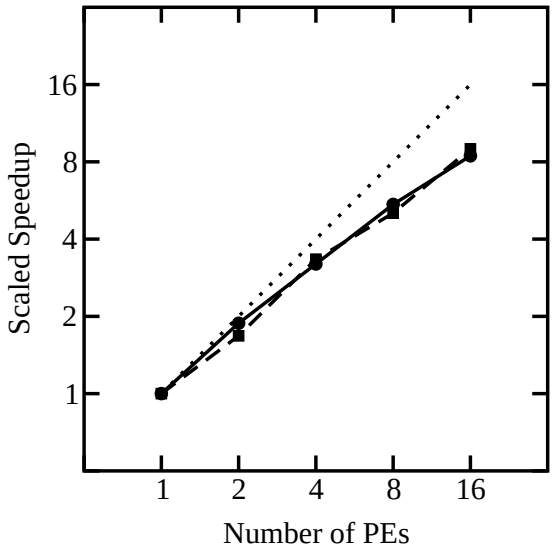


Figure 5: Scaled speedup for a 4D system $X \times Y \times \phi \times \theta$ of the size $64 \times 64 \times 32 \times 16$ per PE in the rotational coordinates $\{\theta, \phi\}$. The performance of the implementations using SHMEM (solid line) and MPI (dashed line) is compared to the linear speedup (dotted line).

- scrutinize the use of MPI_ALLTOALLV instead of MPI_GATHERV/MPI_SCATTERV,
- replace the collective communication MPI_GATHERV/MPI_SCATTERV by point-to-point communication MPI_SEND/MPI_RECV for the Gauss-Legendre Transform in θ to restrict the communication of FBR data to the central $\{j, m\}$ triangle (see right panel of Fig. 1),
- use BLAS routines for the matrix-vector multiplication of the Gauss-Legendre Transform, since in the MPI implementation this operation is not that closely connected to the communication any more,
- extend the MPI implementation to a 6D treatment by adding two more radial coordinates in order to describe for instance the dynamics of a diatomic molecule on a surface completely.

References

1. S. Borowski, S. Thiel, T. Klüner, H.-J. Freund, R. Tisma, H. Lederer, Comput. Phys. Commun. 143 (2002) 162.
2. D. Kosloff, R. Kosloff, J. Comput. Phys. 52 (1983) 35.
3. C. Leforestier, R. H. Bisseling, C. Cerjan, M. D. Feit, R. Friesner, A. Guldberg, A. Hammerich, G. Jolicard, W. Karrlein, H.-D. Meyer, N. Lipkin, O. Roncero, R. Kosloff, J. Comput. Phys. 94 (1991) 59.
4. M. D. Feit, J. A. Fleck, A. Steiger, J. Comput. Phys. 47 (1982) 412.
5. H. Tal-Ezer, R. Kosloff, J. Chem. Phys. 81 (1984) 3967.
6. G. C. Corey, D. Lemoine, J. Chem. Phys. 97 (1992) 4115.
7. J. L. Gustafson, G. R. Montry, R. E. Benner, SIAM J. Sci. Stat. Comput. 9 (1988) 609.

Ein neuer Ansatz zur effizienten Behandlung des Eigenwertproblems in SCF-Verfahren

(mit Anwendung im QUICKSTEP Programm)

Martin J. Fengler

Universität Kaiserslautern

E-mail: fengler@rhrk.uni-kl.de

Zusammenfassung: Theoretische Untersuchungen zur Dynamik molekularer Systeme sind ein heute nicht mehr wegzudenkender Bestandteil der Untersuchung komplexer Systeme in allen Bereichen der Chemie und helfen bei der Interpretation und Analyse experimenteller Untersuchungen. Weitverbreitet ist die Kopplung von (parametrisierten) Kraftfeldern (Molecular Mechanics) mit der klassischen Dynamik, d.h. der Lösung der Newton'schen Bewegungsgleichungen. Neuere Ansätze ersetzen die Kraftfeldmodelle durch die wesentlich genaueren DFT(Density Functional Theory)-Verfahren der ab-initio Quantenchemie.
Die Gewinnung dynamischer Information erfordert die Berechnung einer sehr großen Anzahl von Zeitschritten, so daß die DFT-Verfahren (hier das Quickstep-Modul von CP2K) äußerst effizient implementiert sein müssen. Hier wird in einem iterativen SCF (Self Consistent Field) Verfahren das DFT-Energiefunktional minimiert. Bislang war der zeitintensive Teil eines einzelnen SCF-Schrittes die Berechnung der Kohn-Sham-Matrix. Mit neuen DFT-Methoden, die eine Kombination aus ebenen Wellen- und Gaußfunktionen verwenden, vgl. [1, 2, 3], konnte der dazu notwendig Rechenaufwand stark reduziert werden.
Ein Großteil der noch verbleibenden Rechenzeit wird vom Eigenwertlöser, der im SCF-Verfahren zum Einsatz kommt, verbraucht. Ziel dieser Arbeit ist es, durch die Berücksichtigung der besonderen Matrixstruktur, die im SCF zu Tage tritt, eine Methode darzulegen, die klassischen Eigenwertlösern, wie sie beispielsweise in LAPACK/SCALAPACK zum Einsatz kommen, hinsichtlich algorithmischem und zeitlichem Aufwand signifikant überlegen ist. Zusätzlich wurde dabei auf die besondere Speichereffizienz und massive Parallelisierbarkeit geachtet. Den Grundstein bildet das klassische Jacobi-Verfahren, das durch den hohen Einsatz an BLAS-1 Routinen und Modifikationen nach Pulay in ein Verfahren mündet, das bereits einen sequentiellen Leistungsgewinn von bis zu Faktor 20 ergibt. Insbesondere zeigen Testrechnungen, daß die Konvergenz des *sehr empfindlichen* SCF-Verfahrens nicht beeinträchtigt wird.

Einleitung

Die Quantenchemie beschäftigt sich mit der Simulation von atomaren und molekularen Strukturen. Das Ziel ist es, molekulare zeitabhängige Phänomene zu untersuchen. Um dynamische Phänomene in hoher Detailgenauigkeit zu simulieren, sind sehr viele kleine Zeitschritte erforderlich. In der ab-initio Molekülдynamik werden hierzu die Energien und Kräfte auf der Basis einer ab initio Rechnung, üblicherweise DFT, in jedem Zeitschritt berechnet.

Der größte Aufwand bei der Berechnung eines solchen Zeitschrittes liegt primär in der Lösung der Schrödingergleichung, durch die die Elektronen-Dichte bestimmt ist. Die Dichte wird durch Energieminimierung mit einem iterativen SCF-Verfahren bestimmt:

Sei nun F die Kohn-Sham-Matrix. Sie setzt sich aus einem Ein-Elektronenanteil $F^{(1)}$ und Zwei-Elektronenanteil $F^{(2)}$ zusammen,

$$F_{\mu\nu} = h_{\mu\nu} + \sum_{\lambda\delta} D_{\lambda\delta}(\mu\nu|\lambda\delta) + F_{\mu\nu}^{\text{exch}}(D) = F_{\mu\nu}^{(1)} + F_{\mu\nu}^{(2)}.$$

Darin ist der erste Term h der Ein-Elektronen Hamiltonian, und der zweite Summand der Coulomb-Anteil, gefolgt von dem Austausch- und Korrelationsterm $F_{\mu\nu}^{\text{exch}}(D)$.

Die Dichtematrix $D \in \mathbb{R}^{n\times n}$ wird so bestimmt, daß die Gesamtenergie E minimiert wird. Die Dichte selbst setzt sich aus den mit n_i Elektronen besetzten Eigenvektoren $c_{\bullet i}$ der Kohn-Sham-Matrix zusammen:

$$D_{\mu\nu} = \sum_{i}^{\text{occ}} c_{\mu i} c_{\nu i} n_i,$$

Man startet im SCF mit einer *guten* Initial-Dichte, und durchläuft sukzessiv das folgende Schema, bis die Konvergenzkritierien erfüllt sind:

SCF-Verfahren:

Initialisiere die Eigenvektoren $c_{\bullet i}$

while (Konvergenzkriterium nicht erfüllt)
{

- $D_{\mu\nu} = \sum_i^{\text{occ}} c_{\mu i} c_{\nu i} n_{\text{occ}}$
- $F = F(D)$
- Berechne Gesamtenergie E
- Diagonalisiere $F \rightarrow$ neue Eigenvektoren c

}

Ein einzelner Durchlauf der While-Schleife wurde bislang durch die sehr aufwendige Berechnung der Kohn-Sham-Matrix dominiert, da sich deren Matrixeinträge aus $\frac{1}{8}n^4$ Integralen errechnen. Neuere DFT Verfahren, vgl. [1, 2, 3], konnten den Aufwand für die Berechnung des Coulomb-Anteils von $O(n^4)$ auf ca. $O(n \log n)$ reduzieren. Die verbleibende Auswertung des Austausch- und Korrelationsfunktionals erfordert ca. $O(n)$ bis $O(n^2)$, so daß der Rechenaufwand für den Aufbau der Kohn-Sham-Matrix

drastisch reduziert werden konnte. Damit wird ihre Diagonalisierung zum zeitbestimmenden Schritt. In vielen Fällen entfallen über 90% der Gesamtrechenzeit auf die Diagonalisierung.
Innerhalb des While-Schleifen-Körpers werden üblicherweise zusätzlich die Eigenwerte und Eigenvektoren der Kohn-Sham-Matrix berechnet. Der algorithmische Aufwand beträgt bei einem klassischen Verfahren, das auf dem QR-Algorithmus beruht, ungefähr $O(4n^3)$. Bei großen Systemen dominiert somit sehr schnell der Eigenwertlöser das SCF-Verfahren. Das Ziel dieser Arbeit ist es, einen neuen Löser darzustellen, der die Energieminimierung in ungefähr $O(\frac{2}{3}n^3)$ ausführen kann und im speziellen Fall schwach miteinander wechselwirkender Moleküle sogar einen quasi-quadratischen Aufwand zeigt.
Ein wesentlicher Gesichtspunkt ist dabei, daß die vorgestellte Methode nicht nur schneller als klassische Verfahren ist, sondern auch sehr speichereffizient und sehr gut parallelisierbar auf Grund minimaler Kommunikation ist. Ferner beeinträchtigt sie nicht die häufig eingesetzten Konvergenzbeschleunigungsverfahren, d.h. DIIS (direct inversion of iterative subspace)[4] und Dichte-Matrix-Mittelung. Für die Effizienz ist wesentlich, daß das eingesetzte Jacobi-Verfahren *intrinsisch orthogonale* Eigenvektoren erzeugt. Damit entfällt das besonders aufwendige (meist sequentielle) Nachorthogonalisieren der Eigenvektoren, das bei anderen Verfahren häufig notwendig ist, wie z.B. bei dem auf Bisektion und inverser Iteration beruhenden EVX-Löser[5], wenn große Cluster von Eigenwerten vorliegen.

Theoretische Grundlagen

Das Jacobi-Verfahren

Eigenwertlöser sind auf Grund des algebraischen Problems iterativer Natur. Das Jacobi-Verfahren bestimmt also wie andere Methoden die Eigenwerte und Eigenvektoren nur bis auf eine $|\cdot|-$Genauigkeit von $\epsilon \in \mathbb{R}^+$. Durch die schrittweise Anwendung von Rotationsmatrizen (orthogonalen Transformationen) wird durch Reduktion der Nichtdiagonal-Matrixeinträge die Matrix im Laufe des Jacobi-Verfahrens immer stärker diagonaldominant. Das Verfahren wird abgebrochen,wenn die Quadratsumme der Nichtdiagonal-Werte kleiner als eine Fehlerschranke $\varepsilon \in \mathbb{R}^+$ ist, oder die zu rotierenden Elemente unter einem vorher festgelegtem Schwellwert $\tau \in \mathbb{R}^+$ liegen. Die folgende Beschreibung des Jacobi-Verfahrens ist angelehnt an [6].

Sei im weiteren $A \in \mathbb{R}^{n\times n}$ symmetrisch. Von A sollen alle Eigenwerte und Eigenvektoren bestimmt werden, wobei letztere in der Matrix $Q \in \mathbb{R}^{n\times n}$ zusammengefaßt seien.

Definition:

Seien $p, q \in \{1, 2, \ldots, n\}, p \neq q, \theta \in \mathbb{R}$. Dann heißt

$$J(p,q,\theta) = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & c & & & s & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & -s & & & c & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow p\text{.te-Zeile} \\ \\ \\ \\ \leftarrow q\text{.te-Zeile} \\ \\ \\ \\ \end{matrix}$$

mit $c = \cos\theta$ und $s = \sin\theta$ *Rotationsmatrix* zum Winkel θ.

Wendet man nun die Matrix $J(p,q,\theta)$ auf A an, so wird der Matrixeintrag $A(p,q)$ annuliert. Das ergibt sich aus dem folgenden Satz:

Satz:
Sei $A = (a_{ij})_{i,j=1,\ldots,n} \in \mathbb{R}^{n\times n}$ symmetrisch und $p,q \in \{1,2,\ldots,n\}$ zwei feste beliebige Indizes mit $p \neq q$ und $a_{pq} \neq 0$. Es sei

$$\begin{aligned} \tau &:= \frac{a_{qq}-a_{pp}}{2a_{pq}}, \\ \tan\theta &:= \frac{\text{sign}\tau}{|\tau|+\sqrt{1+\tau^2}}, \quad \text{mit} \quad |\theta| \leq \frac{\pi}{4} \end{aligned}$$

Dann gilt für:

$$\widetilde{A} = J(p,q,\theta)^T A(p,q,\theta) J(p,q,\theta)$$

1. $$\widetilde{a}_{pq} = 0,$$

2. $$(\|\widetilde{A}\|'_F)^2 = (\|A\|'_F)^2 - 2a_{pq}^2$$

   mit

   $$\|A\|'_F := \sqrt{\sum_{i=1}^{n}\sum_{\substack{j=1\\ i\neq j}}^{n} a_{ij}^2}$$

Der letzte Punkt ist für die Formulierung des Jacobi-Verfahrens von immenser Bedeutung. Denn er beschreibt, daß durch wiederholte Anwendung der Rotationsmatrizen die Quadratsumme der Nichtdiagonal-Werte abnimmt. Als Konsequenz daraus nimmt die Matrix immer stärkere Diagonalstruktur an. Im Grenzübergang liegt somit eine Diagonalisierung D vor, d. h. die Eigenwerte stehen auf der Hauptdiagonalen. Wendet man parallel dazu die orthogonalen Transformationen $J(p,q,\theta)$ von rechts auf die zu Anfang als Einheitsmatrix initialisierte Matrix Q an, so erhält man aus (1) und einem ähnlichen Argument im Grenzübergang die Eigenvektoren.

$$\begin{aligned} Q^T AQ &= D \\ AQ &= QD \end{aligned} \tag{1}$$

Damit läßt sich das klassische spalten-/zeilenzyklische Jacobi-Verfahren wie folgt formulieren:

Algorithmus: (Jacobi-Verfahren, spaltenzyklisch)

```
Berechne qsum=Quadratsumme der Nichtdiagonal-Werte
Q=I
repeat
  do q=2,N
    do p=1,(q-1)
       If A(p,q) ≠ 0 then
        Berechne  tan θ für das Paar (p,q)
        Berechne  A := J(p,q,θ)' A J (p,q,θ)
        Berechne  Q := Q J(p,q,θ)
```

```
        Berechne  qsum  =  qsum − A(p,q) ∗ A(p,q)
       endif
     enddo
   enddo
until  qsum  <  ε
```

In der Praxis verwendet man üblicherweise die Variante, bei der man in der if-Bedingung zu einer Schwellwertbedingung übergeht:

```
...
If |A(p,q)|  >  θ then
...
```

Der dadurch gewonne Freiheitsgrad muß experimentell/heuristisch bestimmt werden. Alternativ verändert man wie im folgenden die Schwellwertstrategie.
Tatsächlich spielt es für die Konvergenz zumindest theoretisch ein entscheidene Rolle, in welcher Reihenfolge die Nichtdiagonal-Werte bearbeitet werden. Durch eine Schwellwertvariante kann die Konvergenz im Allgemeinen nicht mathematisch bewiesen werde. Zu bemerken ist hierbei, daß bereits zu Null rotierte - oder Quasi-Null-Elemente - durch die Bearbeitung benachbarter Matrixeinträge in der gleichen Zeile und Spalte wieder größer werden können.

Beschleunigung des Jacobi-Verfahrens

Bei der Implementation des Jacobi-Verfahrens ist zu beachten, daß eine effiziente Form der Anwendung der Rotationsmatrizen erfolgt. Die Rotationsmatrizen erzeugen von links und von rechts angewendet, ausschließlich eine Linearkombination von 2 Zeilen und 2 Spalten, vgl. Abb.1. Berücksichtigt man dies, so erhält man:

```
! Erste Implementierung von:
! J' * A * J

! Erst wird von rechts multipliziert
! A*J

  c_ip=A(:,p)*J_pp + A(:,q)*J_qp;
  c_iq=A(:,p)*J_pq + A(:,q)*J_qq;

  A(:,p)=c_ip;
  A(:,q)=c_iq;

! Dann von links:
! J' * A

  d_pj=J_pp*A(p,:) + J_qp*A(q,:);
  d_qj=J_pq*A(p,:) + J_qq*A(q,:);

  A(p,:)=d_pj;
  A(q,:)=d_qj;
```

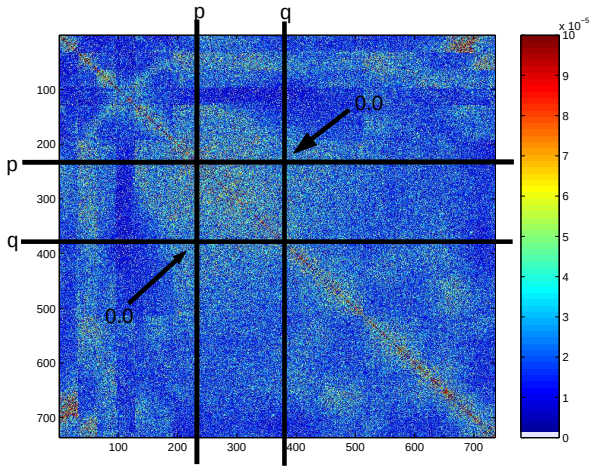


Abbildung 1: Linearkombination der p.ten/q.ten-Spalte bzw. Zeile.

Als nachteilig erweist sich sehr schnell, daß bei der Multiplikation von links (lesend) Zeilenzugriffe auftauchen. Dadurch treten bezogen auf das FORTRAN Speicherlayout Adressierungssprünge auf, die zu

Cache-Misses führen. Unter Ausnutzung der Symmetrie läßt sich jedoch die Multiplikation von links vorziehen und in Spaltenzugriffe ändern. Es bleibt lediglich, da das Produkt $J'AJ$ selbst wieder symmetrisch ist, die *Kreuzungspunkte* zu modifizieren, und die Matrix bzgl. Spalten p und q zu symmetrisieren. Allein das Beseitigen der Cache-Misses bringt einen Speedup von mehr als 30%.

```
! Stride-Optimierte Lösung:
! Erst Multiplikation von links:
! J'*A

  c_ip = J_pp*A(:,p) + J_qp*A(:,q);
  c_iq = J_pq*A(:,p) + J_qq*A(:,q);

  A(:,p)=c_ip;
  A(:,q)=c_iq;

! Dann Modifizieren der Kreuzungspunkte:
! A*J

  A(p,p) = A(p,p)*J_pp + A(q,p)*J_qp;
  A(q,q) = A(p,q)*J_pq + A(q,q)*J_qq;

  A(p,q) = 0.0_wp;
  A(q,p) = 0.0_wp;

! Anschließend A symmetrisieren bzgl. der Spalten p,q
! --> Zeilenzugriffe (schreibend)
```

Diese Modifikation erweist sich zwar in Bezug auf die benötigte Speichermenge als weniger elegant, weil im Speicherbedarf die Symmetrie nicht genutzt wird, dafür aber umso performanter.

Der algorithmische Aufwand liegt mit 6 Durchläufen bei Berechnung der Eigenwerte und Eigenvektoren bei ca. $48n^3$, vgl. [8]. Das ist sowohl im Vergleich zu den klassischen QR-Verfahren mit ca. $\frac{22}{3}n^3$ als auch zu den im EVX-Löser implementierten auf Bisektion und inverser Iteration beruhendem Verfahren mit ca. $\frac{10}{3}n^3$ deutlich ungünstiger [5]. Insbesondere ist zu beachten, daß die Konvergenz des Jacobi-Verfahrens im allgemeinen Fall zunächst nur linear ist. Mit fortschreitender Anzahl von Durchläufen nimmt jedoch die Matrix eine immer stärkere Diagonaldominanz an. Sind dann die Nichtdiagonal-Werte hinreichend klein, so ist die Konvergenz sogar quadratisch, vgl. [8].

Um das Jacobi-Verfahren konkurrenzfähig zu machen, müssen also Vereinfachungen, resp. Modifikationen in Kauf genommen werden, die sich erst mit Blick auf die spezielle Anwendung im SCF rechtfertigen lassen. Diese Veränderungen an dem Verfahren sind nur schwerlich als *stets erfolgreich* zu beweisen, sie sollen vielmehr erst eine Rechtfertigung durch den Erfolg der Methode in der Praxis bekommen. Das neue Modell bekommt daher den Charakter einer sehr guten Heuristik.

Beispielsysteme aus QUICKSTEP

Um einen besseren Überblick zu bekommen, zeigen Abb. 2 und 3 ein Beispiel einer Kohn-Sham-Matrix in MO-Basis, wie sie im 4. SCF-Schritt bei 32 simulierten H_20 Molekülen entstand.

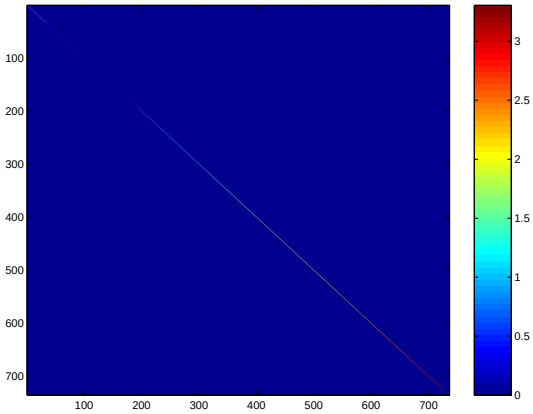


Abbildung 2: Betrag einer KS-Matrix (32 H_2O), Farbskala von $[0 \ldots 3.5]$.

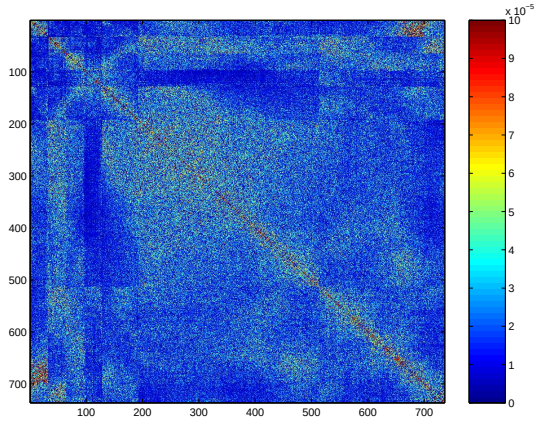


Abbildung 3: Betrag derselben KS-Matrix, Farbskala von $[0 \ldots 10^{-4}]$.

Wie zu sehen ist, liegen mit den Systemen in der MO-Basis stark diagonal dominante Matrizen vor, die jedoch dicht besetzt sind. Insbesondere zeigt die Erfahrung, daß genau in den kleinen Nichtdiagonal-Anteilen die entscheidende Information für die Konvergenz des SCF liegt. Sie ist weder vernachlässigbar noch gut komprimierbar, z.B. mit Wavelets. Wie Experimente mit Waveletkompressionstechniken zeigten, prägen sich unterliegende Strukturen stark auf die Waveletkoeffizienten auf und stören die üblicherweise entstehende dyadische Bandstruktur erheblich.

Da in den SCF-Iterationen die Energie nur von den besetzten Orbitalen abhängt, lassen sich mit einer Blockdiagonalisierung besetzte und unbesetzte Orbitale entkoppeln. Da das Energieminimum somit nur auf dem Untervektorraum, der durch die Eigenvektoren, die mit den besetzten Orbitalen korrespondieren, angenommen wird, genügt auch eine Blockdiagonalisierung. Anstatt alle Eigenvektoren des Unterraums zu berechnen, der von den Eigenvektoren der besetzten Orbitale aufgespannt wird, reicht es in jedem Schritt entsprechend viele (linear unabhängige) Vektoren zu bestimmen, die denselben Span besitzen.

Da die Blockung a priori bekannt ist, läßt sie sich völlig problemlos integrieren, vgl. Abb. 4.

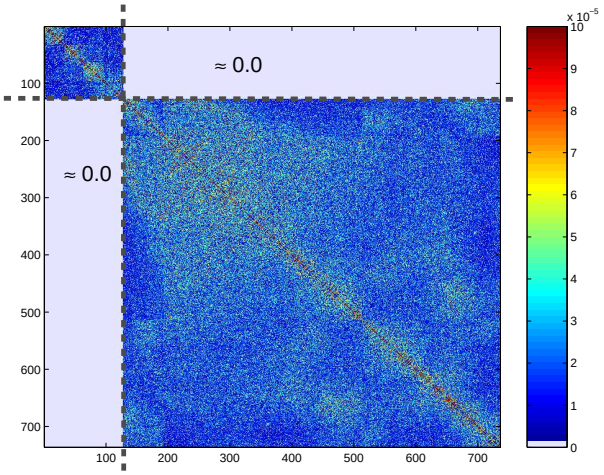


Abbildung 4: Block-Jacobi-Verfahren: Erwirkt die Trennung von besetzten und unbesetzten Orbitalen.

In unseren Fällen macht der Anteil der besetzten Orbitale etwa 20 % aus. Somit werden in einem Block-Diagonal-Durchlauf höchstens $0.2 \cdot n \cdot 0.8 \cdot n = 0.16 \cdot n^2$ Elemente bewegt. Da bei den Molekülen/Beispielen, die wir betrachtet haben, ohnehin nach der $4./5.$ SCF-Iteration der Nichtdiagonalanteil marginal ist, kann man auch erwarten, daß ein einziger Durchlauf bereits genügt, um eine neue Näherung für die Trennung der Eigenräume zu erreichen. Theoretisch ist das in dieser Situation mit der quadratischen Konvergenz des Jacobi-Verfahrens zu begründen. Diese Tatsache läßt sich *experimentell* bestätigen, und führt zu einer signifikanten Beschleunigung des gesamten Verfahrens.

Von weiterer Bedeutung für Beschleunigungen ist die Tatsache, daß der Eigenwertlöser als iteratives Verfahren selbst Teil eines iterativen Verfahrens ist, nämlich des SCF. Um die SCF-Konvergenz möglichst schnell zu erreichen, finden Konvergenzbeschleuniger, wie einfaches Mitteln von alter und neuer Dichte und DIIS ihren Einsatz. Wie unsere Rechnungen zeigten, ist die Konvergenz von SCF in der Kombination von Dichtemittelung und DIIS nicht oder nur marginal beeinflußt.

Eine weitere Optimierungsmöglichkeit stellt eine veränderte Schwellwertstrategie dar. Werden in der klassischen Variante nur Elemente rotiert, die über einem vorgegebenen Schwellwert liegen, so werden in einer Erweiterung nur die Elemente rotiert, die einen signifikanten Einfluß auf die Diagonalwerte, resp. Eigenwerte haben. Dazu skaliert man die Nichtdiagonal-Matrixeinträge A_{pq} mit dem reziproken Abstand der Eigenwerte λ_p un λ_q aus der letzten vollen Diagonalisierung:

$$\widehat{A}_{pq} \leftarrow \frac{1}{|\lambda_p - \lambda_q|} A_{pq}.$$

Führt man $k \in \mathbb{R}^+$ als *prozentualen* Einflußfaktor ein, so werden dann nur Elemente $\widehat{A}_{pq}$ rotiert, für die $\widehat{A}_{pq} > k \cdot \max_{\substack{p,q \\ p \neq q}} |\widehat{A}_{pq}|$ ist, mit $k \approx 0.01 \ldots 0.1$.

In unserem Fall würden exemplarisch in Abb. 5 nur die rötlich gefärbten Elemente in Frage kommen.

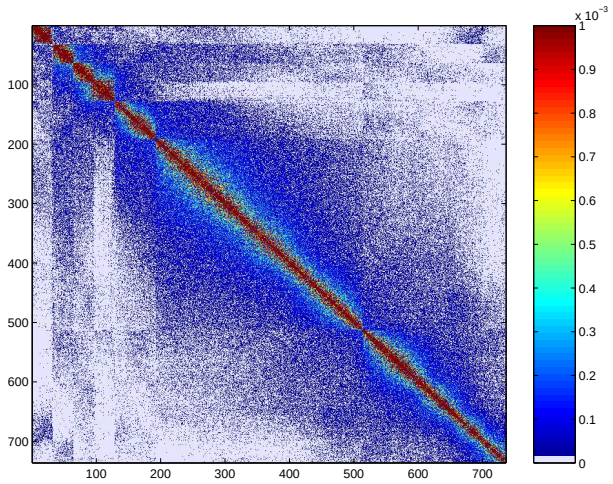


Abbildung 5: KS-Matrix nachdem mit $\frac{1}{|\lambda_i - \lambda_j|}$ skaliert wurde.

Idee von Pulay

Die bisher angesprochenen Modifikationen bringen das Block-Jacobi-Verfahren in Konkurrenz zu dem LAPACK-EVX-Löser. Der erreichte algorithmische Aufwand liegt in unseren Fällen jetzt, bei $0.16n^2$ maximal zu bewegenden Elementen und $8n$ Multiplikationen, ungefähr bei $1.2n^3$. Das wäre gegenüber einem klassischen *exakten* Löser immerhin ein Faktor von 3. Durch die oben erwähnten Speicherzugriffsprobleme ist davon in der Praxis nur ein Faktor 2 zu spüren. Das ist noch zu gering, und mit

Blick auf die bei einer Parallelisierung zu realisierenden Zeilenupdates unzureichend, nicht zuletzt da die SCALAPACK-Löser bei hinreichend großen Systemen einen Speedup von 25 mit 64 Prozessoren erreichen.

Nach einer Idee von P. Pulay, vgl. [7], die bereits in Zusammenhang mit semiempirischen Methoden 1981 veröffentlicht wurde, genügt es, auf das Aktualisieren der Kohn-Sham-Matrix vollständig zu verzichten. Es werden also keine Rotationen explizit angewendet, da die Eigenwerte im SCF verworfen werden können. Es werden ausschließlich die Eigenvektoren modifiziert. Das führt zu einem dramatischen Geschwindigkeitsgewinn, indem $4n$ Multiplikationen und die ungünstigen Zeilenzugriffe wegfallen.
Das Aktualisieren der Eigenvektoren wird mit BLAS-1 Routinen realisiert. Somit ergibt sich für das Block-Jabobi Verfahren mit einfacher Schwellwertstrategie:

```
do q=start_sec_block,N
  do p=1,(start_sec_block-1)

    if(abs(A(p,q))>thresh) then

      tau = (A(q,q)-A(p,p))/(2.0_wp*A(p,q));

      tan_theta = sign(1.0_wp,tau)/(abs(tau)+sqrt(1.0_wp+tau*tau));

      ! cos theta
      c = 1.0_wp/sqrt(1.0_wp+tan_theta*tan_theta);
      s = tan_theta*c;

      ! Und jetzt noch die Eigenvektoren produzieren:
      ! Q * J

      call dcopy(N,EV(1,q),1,c_ip,1)
      call dscal(N,-s,c_ip,1)
      call daxpy(N,c,EV(1,p),1,c_ip,1)

      call dscal(N,c,EV(1,q),1)
      call daxpy(N,s,EV(1,p),1,EV(1,q),1)

      call dcopy(N,c_ip,1,EV(1,p),1)

    endif

  enddo
enddo
```

Numerische Ergebnisse

Das gesamte Verfahren kann von den abnehmenden Nichtdiagonal-Werten während des fortschreitenden SCF-Verfahrens profitieren, so daß sich für die sequentielle Realisierung die folgenden Abbildungen ergeben.

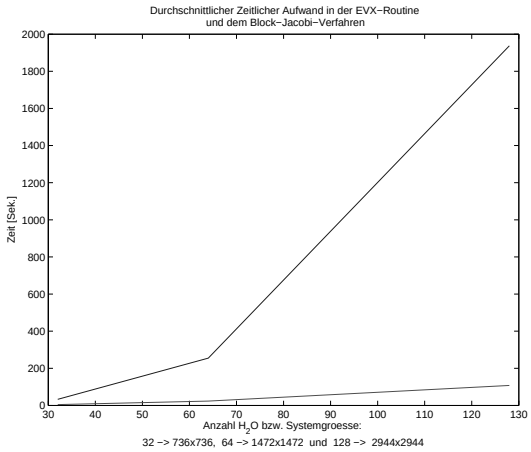


Abbildung 6: Durchschnittlicher Zeitaufwand mit wachsender Problemgröße (rot: Blockjacobi, blau: EVX).

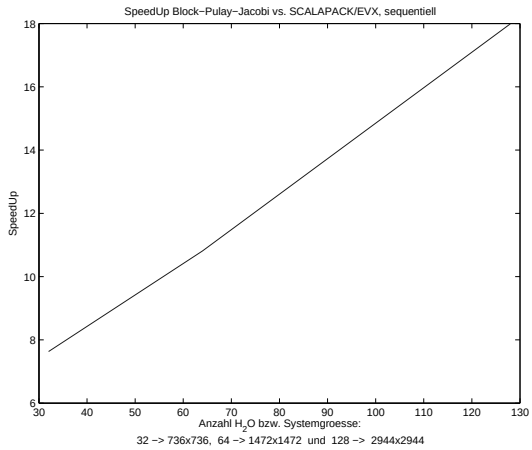


Abbildung 7: Durchschnittlicher Speedup mit wachsender Problemgröße.

Der theoretische algorithmische Aufwand beträgt maximal $4n * 0.16n^2 = 0.64n^3$. Mit dem Einsatz von BLAS-1 Routinen zeigt sich, dass der theoretische zu erwartende Faktor von $\frac{10}{3}/0.64 = 5.3$ gegenüber LAPACK-EVX tatsächlich bei kleinen Matrizen mit vollbesetztem Nichtdiagonal-Block angenommen wird.

Das spricht für eine sehr effiziente Implementierung. Bei großen Systemen spielt das extrem günstige Speicherlayout des Verfahrens eine immer stärkere Rolle. Hinzu kommt, daß dieses neue Verfahren sehr stark von der Schwellwertvariante profitiert und durch die Ersparnis von Rotationen einen signifikanten Performancezuwachs von mindestens einem Faktor 18 ergibt. Größere Systeme konnten bislang sequentiell nicht gerechnet werden, da der Speicherbedarf anderer Programmteile zu stark wuchs.

Bemerkenswert ist das nach Abb. 7 quasi-quadratische Laufzeitverhalten in Bezug zur Systemgröße, da der Speedup gegenüber dem n^3-skalierenden SCALAPACK-Löser linear mit der Systemgröße wächst. Dies ist wahrscheinlich auf die quasi-Entkopplung der vielen nur schwach miteinander wechselwirkenden Wassermoleküle zurückzuführen.

Da man erst nach einigen SCF-Schritten auf eine hinreichend ausgeprägte Diagonalstruktur der Kohn-Sham-Matrix stößt, wurde für die ersten 6 Schritte des SCF der SCALAPACK/LAPACK-EVX eingesetzt, und anschließend erst das modifizierte Jacobi-Block-Verfahren eingeschaltet. Die gesamte Programmlaufzeit des QUICKSTEP konnte durch die Verbesserung des Eigenwertlösers auf die Hälfte gesenkt werden, vgl. Abb. 8. Da bei den betracheten Beispielen immer insgesamt 12 SCF-Schritte durchgeführt wurden, liegt der zeitliche Aufwand immer noch zu einem Großteil in den vorgeschalteten 6 EVX Aufrufen. Die nachfolgenden 6 Aufrufe des Jacobi-Block-Lösers fallen nicht mehr ins Gewicht. Es dominieren nun Programmteile, die beispielsweise das allgemeine ins spezielle Eigenwertproblem transformieren.

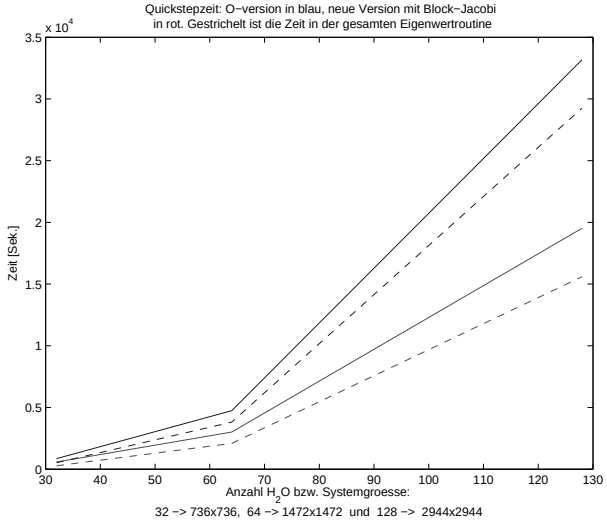


Abbildung 8: Gesamtzeit QUICKSTEP und Eigenwertlöser, bei verschiedenen Problemgrößen

Für die Zukunft wären hier Tests notwendig, ob man das neue Verfahren bereits nach noch weniger SCF-Iterationen einschalten kann. Möglicherweise könnte man es stets gemeinsam mit dem DIIS einsetzen.

Parallelisierung

Das nach Pulay modifizierte Verfahren läßt sich sehr einfach parallelisieren, da lediglich die Eigenvektoren zu modifizieren sind. Dazu muß jeder der beteiligten Prozessoren den *oberen Nichtdiagonal-Block* lokal verfügbar haben. Ferner operiert man auf einem $N \times 1$ Prozessgitter, so daß jeder Prozessor einen Anteil vollständiger Zeilen besitzt, vgl. Abb 9. Denn so kann er unabhängig von den anderer Prozessoren die notwendigen Linearkombination der p.ten/q.ten-Spalten durchführen. Es ist somit nur Kommunikation bei Beginn und Ende notwendig, um die erforderlichen Daten zu erhalten bzw. zu verteilen.

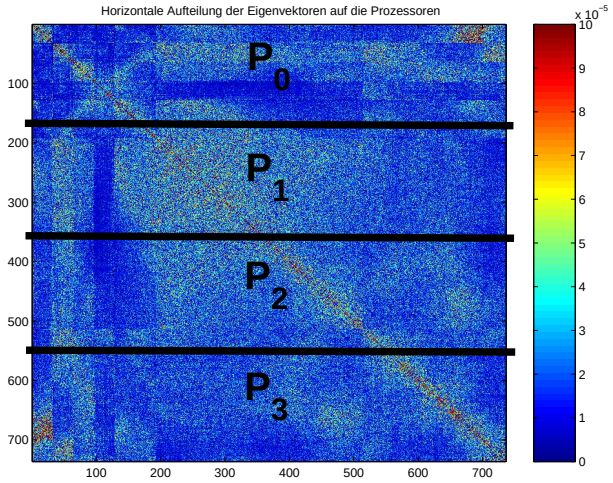


Abbildung 9: Datenverteilung der Eigenvektoren.

Es ist offensichtlich, daß diese Art von Parallelisierung sehr speichereffizient arbeitet. Die Skalierbarkeit des Algorithmus wurde am Beispiel von 64 H_2O getestet:

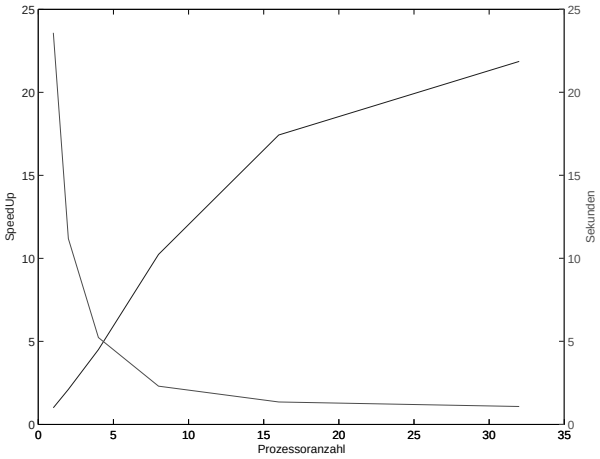


Abbildung 10: Speedup bei einem 1472×1472-System.

Bemerkenswert ist der zunächst superlineare Anstieg, der wohl auf Caching-Effekte zurückzuführen ist. Der verhältnismäßig langsame Anstieg im hinteren Teil der Kurve ist auf die kleine Problemgröße, die auf 32 CPUs berechnet werden soll, zurückzuführen. Damit ist der Kommunikationsaufwand zu Beginn und Ende des Eigenwertlösers dominierend, denn es wird ja effektiv nur 1 Sekunde gerechnet. Zum andern kommt es bei sovielen CPUs im Verhältnis zur Systemgröße zu Problemen in der Lastverteilung: jeder Prozessor bekommt hier blockweise Pakete von 32 Zeilen der Matrix. Verteilen sich 1472 Zeilen auf 32 CPUs, so besitzen die Hälfte der Prozessoren in etwa zwei Zeilenpakete und die anderen nur eines. Zu testen bleibt, ob sich der sehr gute Speedup für wesentliche größere Systeme auch auf größere Prozessorzahlen fortsetzt.

Ausblick

Es ist gelungen, ein Verfahren zu entwickeln und zu implementieren, das hinsichtlich Robustheit, Performance, Speedup und Speichereffizienz kaum Wünsche offen läßt. Es bleibt abzuwarten, wie sich das Verfahren bei sehr großen und komplizierteren Systemen verhält. Dies wird in Kürze am CSCS in LUGANO/MANNO getestet.
Erste Testrechnungen dieses neuen Ansatzes im Quantenchemieprogrammpaket TURBOMOLE mit anderen Molekülen zeigen darüberhinaus sehr vielversprechende Ergebnisse. Die Genauigkeit dieser Methode, die mittels des Schwellwerts zusätzlich gesteuert werden kann, demonstriert hier ausreichende Ergebnisse.

Literatur

1. M. KRACK, M. PARRINELLO *All electron ab initio molecular dynamics,* Phys. Chem. Chem. Phys. 2 (2000) 2105
2. G. LIPPERT, J. HUTTER, M. PARRINELLO *The gaussian augmented-plane-wave density functional method for ab initio molecular dynamics simulations,* Theor. Chem. Acc. 103 (1999) 124
3. G. LIPPERT, J. HUTTER, M. PARRINELLO *A hybrid gaussian and plane wave density functional scheme,* Mol. Phys. 92 (1997) 477
4. P. PULAY, *Improved SCF Convergence Acceleration* J. Comp. Chem. 3 (1982) 556
5. BLACKFORD,CHOI,CLEARY,D'AZEVEDO,DEMMEL,DHILLON,DONGARRA,HAMMARLING,HENRY,PETITET, STANLEY,WALKER,WHALEY, *SCALAPACK Users's Guide,* Siam, Philadelphia, 1997
6. G.ALEFELD,I.LENHARDT,H.OBERMAIER *Parallele numerische Verfahren,* Springer, Berlin, 2000
7. P.CSASZAR,P. PULAY *Fast Semiempirical Calculations,* Journal of Computational Chemistry, Vol.3, No. 2 227-228, 1982
8. H.R.SCHWARZ, *Numerische Mathematik,* B.G.Teubner, 1997

Parallelising a method for reconstructing evolutionary trees by the example of the software package 'Tree-Puzzle'

Ekkehard Petzold

University of Leipzig, Germany
Max Planck Institute for Evolutionary Anthropology
Leipzig, Germany

petzold@eva.mpg.de

Abstract: Methods for finding evolutionary trees based on the maximum likelihood principle involve high computational effort and therefore are particularly time-consuming. Hence, a lot of research is done to adjust existing routines to be computed parallely. The software package 'Tree-Puzzle', which implements the aforementioned methods, has already been partly parallelised, acceleration of computation time however still suffers from those parts still computed sequentially. This project is designed to parallelise the most perturbing sequential part in the programme.

Preface

When Charles Darwin founded his Theory of Evolution in 1859, along with it he introduced the evolutionary tree as a tool to visualise evolutionary relations between different species. Evolutionary trees are intended to show how species, represented by leaves, evolved from common ancestors, represented by parental nodes, and how long it took them to reach their evolutionary state relative to their ancestors, which is pictured by the branch lengths of the tree. Darwin, according to his theory, assumed that the data for the tree reconstruction was mainly based on physical properties and geographical extension of the species concerned.
However, scientific progress has added some important references to that list. Nowadays scientists also take into account the substantial information provided by genome and protein sequences, which could not be decoded in the nineteenth century. These new sources allow a more detailed view of evolutionary relations as happened recently when scientists discovered that the genome of Western African elephants is so different from the common African elephants' that they may possibly form a species of their own.
Now to analyse amounts of information as extensive as genome or protein data, the development and use of appropriate computer applications is easily motivated. 'Tree-Puzzle' represents an application concerning this purpose. Yet, the sequential approach offered by the programme primarily is not sufficient for large quantities of data, limiting those to certain numbers of species to be processed into one tree. To provide efficient scientific work with larger amounts involves making as much of the software as possible run parallely.
Throughout the last years, certain efforts have been taken to do so, but the computing time caused by the parts remained sequential still is restricting. Therefore, parallelisation must be enhanced, which forms the task of this specific project.

Processing Sequence Data Into a Maximum Likelihood Tree

Genome resp. protein data is provided in a list of sequences, each representing a species. These sequences are aligned with sites of certain functionalities. The number of of sequences is indicated by S, the number of sites by N.

	Site 0	Site 1	Site 2	Site 3	Site 4	Site 5	Site 6	Site 7	Site 8	...	Site N−2	Site N−1
Sequence 0	A	T	A	A	A	A	G	C	T	...	A	T
Sequence 1	C	G	A	G	G	C	G	C	C	...	T	G
Sequence 2	A	C	C	T	C	T	G	C	G	...	A	G
Sequence 3	A	C	G	G	G	T	T	C	A	...	A	T
...	...	...	...	...	...	...	...	...	...	...	...	...
Sequence S−1	T	C	G	A	G	T	A	C	T	...	A	C

Figure 1: Genome sequence alignment

Among the $\prod_{k=3}^{S}(2k-5)$ possible tree topologies with the S sequences provided by the alignment being the leaves, we want to find the most likely one. Therefore we need a likelihood function for tree topologies.

The Software Package 'Tree-Puzzle'

As an implementation of this problem, 'Tree-Puzzle' was written by biologist Korbinian Strimmer in 1997 and is available on [1]. It strongly refers to the basic approaches of "Programs for Molecular Phylogenetics Based on Maximum Likelihood" by Adachi and Hasegawa [2].

Basic Structure

'Tree-Puzzle' is devided into 3 main parts:

1. Estimation of likelihood function parameters The programme analyses the contents of the sequence alignment, such as frequency of occurrences of each base resp. protein or amounts of variety within each site. The parameters for the likelihood functions of tree topologies are calculated from these results.

2. Finding the most likely among all possible tree topologies The likelihood function determined in the first step is used to find the most likely tree topology.

3. Fixing the maximum likelihood branch length The parameters fixed in the first step also serve to estimate the branch lengths of the tree chosen in the second step.

Due to a faculty complexity, the second part is the most time-consuming. This part has already been parallelised. Now the first part has become the slowest, so parallelising efforts stressed on this part. The third part will be subject to further exertions in the future.

Existing Parallel Architecture

Thanks to the parallelisations done so far, a parallel architecture based on MPI has already been established in 'Tree-Puzzle'. This architecture was regarded suitable to host the new parallel features.

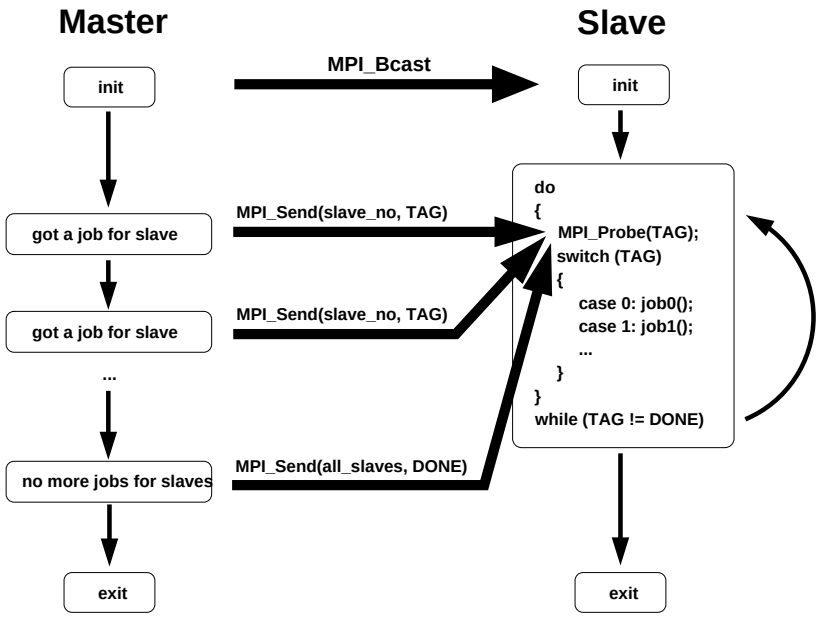


Figure 2: The existing parallel architecture

The strict master-slave structure implies that the master does the sequential work as originally implemented and calls a particular slave for a particular job, which is triggered by a single communication which carries a tag used to identify the kind of job. This entails a disadvantage of this architecture: Slaves cannot be called as a group from their stand-by state because MPI does not provide any kind of group communication having a tag that could be evaluated by the called slaves. However, that proved to be un-dramatic since this situation could be emulated by a line of asynchronous, thereby very fast single communications calling the slaves for a group communication which by the way acts as a synchroniser.

Parallelisation of the Parameter Estimation Part

The programme part of estimating the likelihood function parameters, which was to parallelised mostly, consists of nested conversion loops which excludes top level task parallelism as it was the case in the already parallelised part of the programme, which meant to scan lower levels in the programme structure for time-consuming functions.

First parallelisation step: Updating Pairwise Evolutionary Distances

Computing time analysis detected a function accounting for more than 90 per cent of time consumption in the parameter estimation part and inhering a high grade of task parallelism. The function updates a matrix containing estimated pairwise evolutionary distances between the sequences involved. Updating the estimated evolutionary distance between two sequences, besides the data of those two sequences and the old estimate, involves the latest update of likelihood function parameters. The distance matrix is a symmetric $S \times S$ matrix with the main diagonal vanishing, which leaves the update of the $S(S-1)/2$ elements of the upper triangular matrix that are independent from each other, thus parallelisation is applied here.

Calculation of Pairwise Evolutionary Distance

To understand the behaviour of a single distance update, we have to take in account how it is done. As aforementioned, the required parameters consist of the sequence data of both species, the old estimate and the latest likelihood function parameters, the latter defining a function that returns the likelihood of a given distance d between two bases resp. proteins at a single site. Multiplying the functions of all sites results in a likelihood function of d for the two sequences in whole.

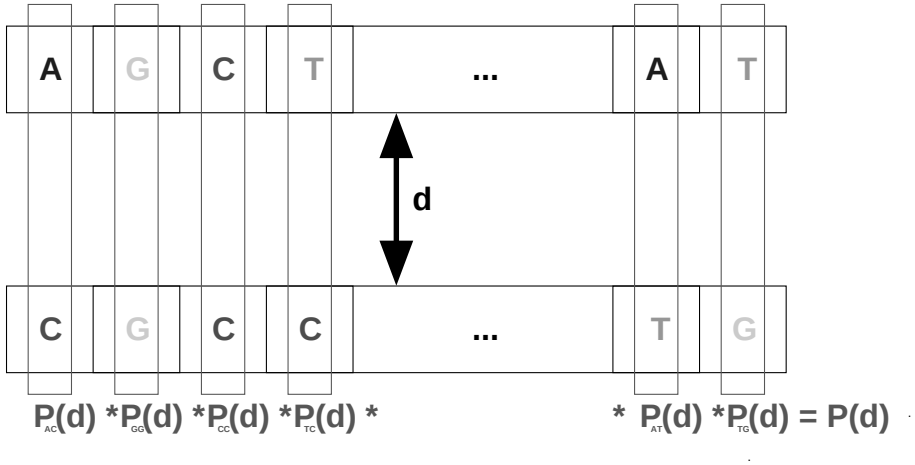


Figure 3: Evolutionary distance likelihood function

d has to be adjusted to maximise P. This constitutes an optimisation problem. 'Tree-Puzzle' therefore deploys a method called the Brent Algorithm, see figure 4, that has the advantage of avoiding tedious calculation of derivatives [3].

The computational effort to the calculation of one pairwise evolutionary distance is dependent on how fast the Brent algorithm converges.

Choosing a Scheduling Algorithm

Having $S(S-1)/2$ single jobs and P processors (one master and $P-1$ slaves), we are confronted with the task how to distribute the work. At this juncture, we have to find a happy medium between the paradigm of optimal load balancing and that of minimal communication.
To follow the latter, an intuitive approach would be to have each slave calculate the same number of matrix elements. This provides us with the least communication possible and a theoretically optimal load balancing. However, this is only optimal as long as the Brent algorithm converges robustly. Extensive testing showed this is mostly the case, but there is no reliability. With some data sets, the load balancing got out of equilibrium as can be seen in figure 5.

The opposite extreme is the so-called self-scheduling algorithm with chunk size set to one, the master giving each slave one matrix element to calculate, the slave returning the result and being handed a new element while there are any left. Load balancing is optimal, while communication is maximal. That means a communication overhead that becomes considerable with S being large and N being not too large. In these cases, an atomic job takes little time to calculate while the large number of atomic jobs causes communication jams.
This led to the consideration of the Guided self-scheduling algorithm, which is similar to the previous but attempts to avoid futile communication while keeping optimal weight balancing. It does so by starting with large chunk sizes and decreasing them continually. With L being the number of elements left to compute, chunk size C is calculated as follows:

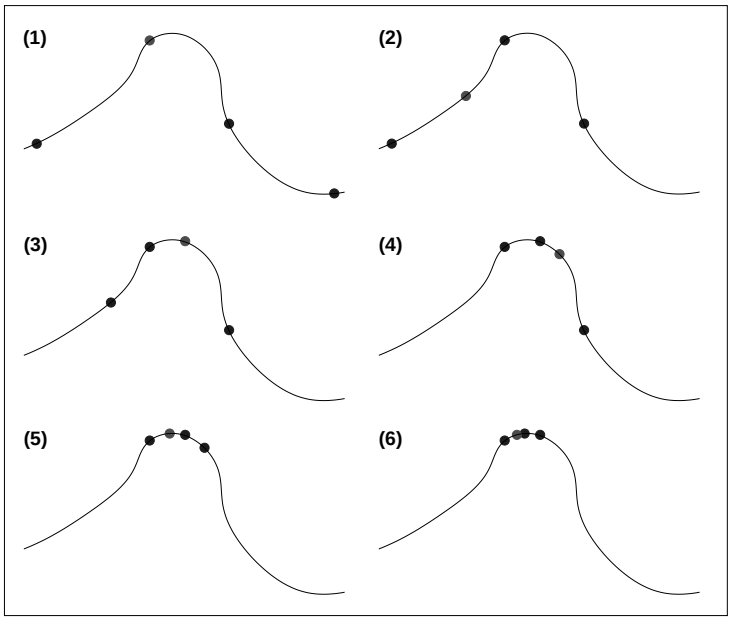


Figure 4: The Brent algorithm for finding a local extremum. Brent is initialised by three points, the medial point having a higher value than the outer points. This constellation provides a maximum bracketed by the outer points. Also, the proportion of distance a between the left and the medial point and distance b between medial and right point is such that a can be devided into $b + c = a$ with b and c having the same proportion again (1). At the beginning of each recursive step, the larger distance is devided this way, the smaller distance touching the medial point. Among these four points, that with the highest value is the new medial point with the two neighbours being the new outer points.

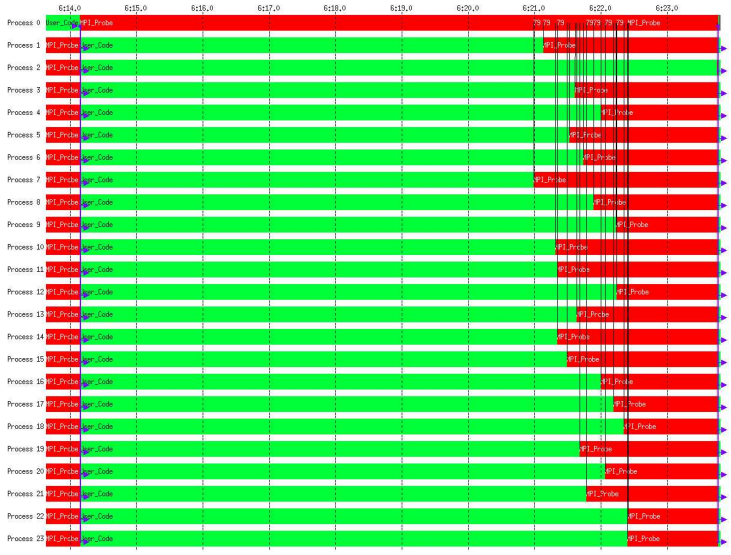


Figure 5: The scheduling algorithm of Static chunking. Due to non-optimal load balancing, process 2 causes a delay.

$$C = \frac{L}{2 \times P} \tag{1}$$

As L decreases, so does C. That has the effect of upholstering unexpectedly long computing time of single chunks. The new shape of communication is visualised in figure 6.

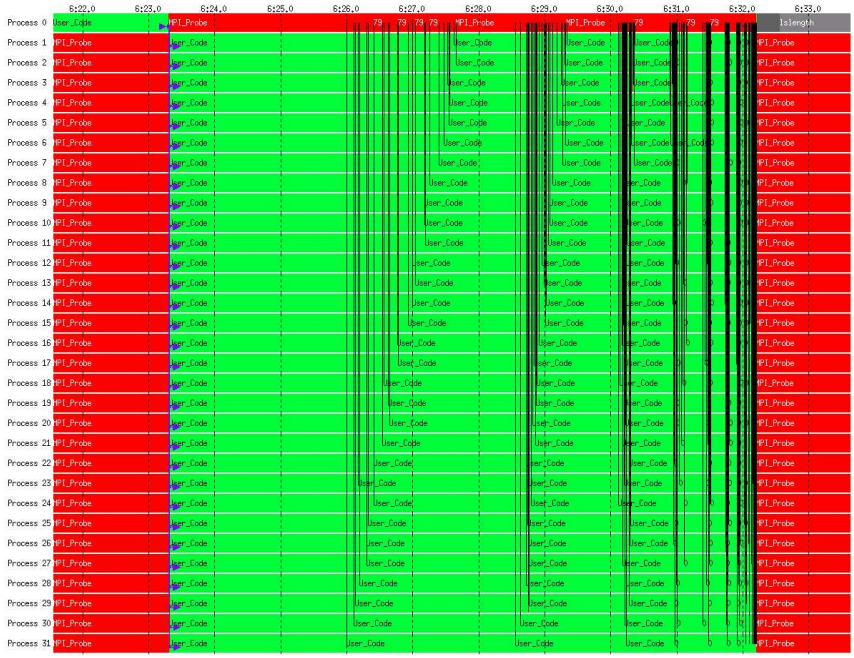


Figure 6: The Guided self-scheduling algorithm.

Using the guided self-scheduling algorithm, run times were never longer than those with static chunking, sometimes they were better. That is why the guided self-scheduling algorithm was eventually adopted in the parallelisation.
The speed-up charts in figure 7 show there is still a considerable sequential part extenuating the speed-up for large data sets, particularly for S being large.

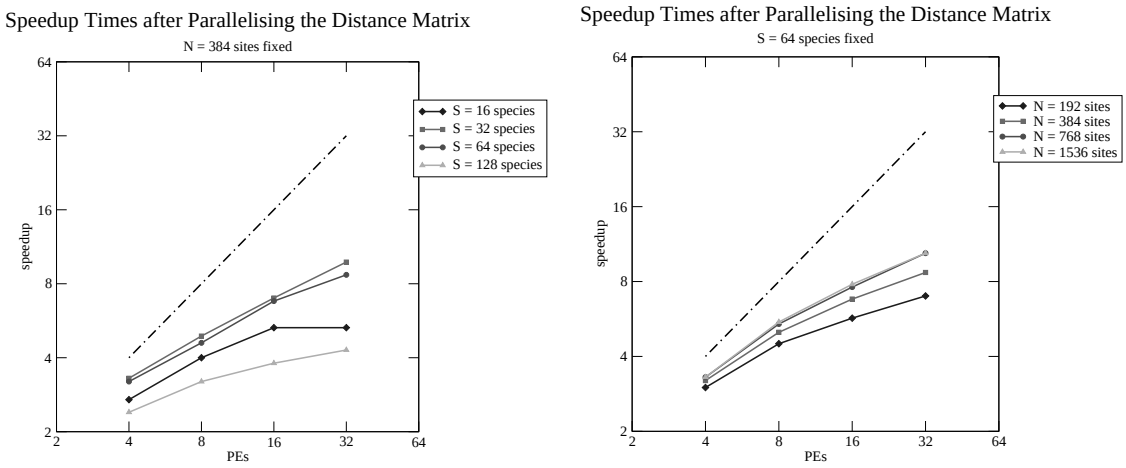


Figure 7: Speed-up chart for parameter estimation after fi rst parallelisation step.

Second Parallelisation Step: Initialising the Branch Relation Matrix

The modest results of the parallelisation done so far make further effort appropriate. The unacceptable lack of robustness against large numbers of sequences is a sign for a nested loop high-grade dependent on S. This loop could be found in the initialisation step of a $(2S-3) \times (2S-3)$ matrix. This matrix holds information about pairwise relations of the $(2S-3)$ branches of a fully resolved tree with the S sequences as leaves. It is initialised by counting the cases when both branches are elements of a path

from one leaf to another. Therefore a $(S(S-1)/2) \times (2S-3)$ Boolean matrix exists forming a path map, in which each column stands for one of the $S(S-1)/2$ pairs of sequences and contains `true` values for those of the $(2S-3)$ branches that are elements of the path between these two sequences, i.e. each row represents a branch and contains `true` values for those pairs of sequences connected by a path that this branch is part of. Calculating the $(2S-3)(S-1)$ elements of the upper triangular matrix plus main diagonal, each calculation consisting of $S(S-1)/2$ conditional additions, the resulting complexity is $O(S^4)$, see figure 8.

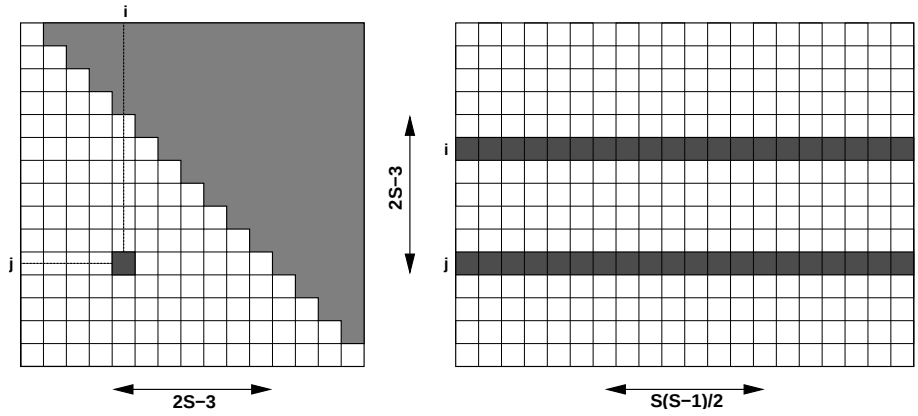


Figure 8: The branch relation matrix and the path map.

Sequential Optimisation

Despite parallelisation of this problem being relatively trivial, there is reason for optimising the sequential code first. The aforementioned nested loop hosts the calculation of one matrix element `BranchRelationMatrix[i][j]` by counting the number of occurrences of
`PathMap[i][k]` $=$ `true` $\wedge$ `PathMap[j][k]` $=$ `true`, $k = 0, \ldots, S(S-1)/2-1$.
In the conventional implementation this is technically done as follows.

```
for (i=0; i<2*S-3; i++) {
   for (j=0; j<=i; j++) {
      for (k=0, sum=0.0; k<S*(S-1); k++)
         sum += PathMap[i][k] * PathMap[j][k];
      BranchRelationMatrix[i][j] = sum;
      BranchRelationMatrix[j][i] = sum;
   }
}
```

The PathMap matrix is declared as `double` matrix although it exclusively contains Boolean values. Therefore the loop body contains a double precision multiplication and a double precision addition. This can easily be replaced by a PathMap matrix declared as `char` matrix to save memory, by changing the multiplication into a conditional and then nesting the sum, declared as an `int` sum, into the conditional with a `++` operation.

```
for (i=0; i<2*S-3; i++) {
   for (j=0; j<=i; j++) {
      for (k=0, int_sum=0; k<S*(S-1); k++)
         if (PathMap[i][k] && PathMap[j][k]) int_sum++;
```

```
      BranchRelationMatrix[i][j] = (double) int_sum;
      BranchRelationMatrix[j][i] = (double) int_sum;
   }
}
```

Approx. S^4 `double` multiplications and the same number of `double` additions are replaced by S^4 conditionals and a smaller number of `int` additions. This sequential optimisation provides a considerable speed-up as can be seen in figure 9.

Figure 9: Acceleration through sequential optimisation. The optimised loop is represented by the grey bar on process 0. The upper resp. lower charts display computing time before resp. after optimisation. The grey bar is clearly scaled down.

Parallelisation

The parallelisation of the initialisation loop for the branch relation matrix is straight forward. Since the number of basic computational operations is known it is no problem to distribute calculation a priori providing optimal load balancing. We use the static chunking scheduling algorithm from the first parallelisation step. After the path map matrix is broadcast to all slaves, each process calculates its part of the upper triangular branch relation matrix. Finally the results are gathered by the master.

```
MPI_Bcast(PathMap);
i = MyStarti; j = MyStartj;
for (l=0; l<MyJobLength; l++) {
   for (k=0, int_sum=0; k<S*(S-1); k++)
      if (PathMap[i][k] && PathMap[j][k]) int_sum++;
   MyResults[l] = (double) int_sum;
   NewCoords(&i, &j);
   MPI_Gather(GlobalResultVector, MyResults);
}
```

After this second parallelisation step speed-up values have improved. As can be seen from figure 10, speed-up is robust particularly with S being large.

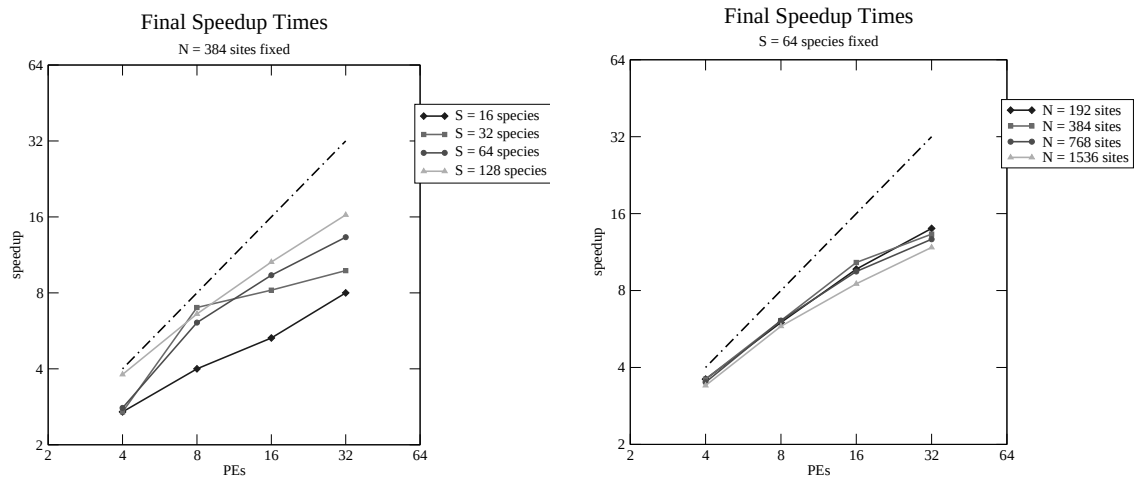


Figure 10: Final speed-up chart for parameter estimation after second parallelisation step.

Conclusion and Future Work

The improvements in computation speed achieved by the modifications of this project are beneficial. The practical value of 'Tree-Puzzle' could be increased since now it is possible to compute larger data sets without waiting hours or even days for the results of parameter estimation. A large data set, for example, containing 215 sequences with a length of 2100 took 2 hours 30 minutes for parameter estimation sequentially. After parallelisation, with the same sequence this part was finished after 8 minutes 38 seconds, using all 32 processors on the Zampano cluster.
Further parallelisation is recommended for the last part of 'Tree-Puzzle', which still runs absolutely sequentially and therefore takes longest at the moment.

References

1. HEIKO A. SCHMIDT, KORBINIAN STRIMMER, MARTIN VINGRON, ARNDT VON HAESELER, Tree-Puzzle 5.0, www.tree-puzzle.de
2. JUN ADACHI, MASAMI HASEGAWA, *Programs for Molecular Phylogenetics Based on Maximum Likelihood,* The Institute of Statistical Mathematics, Tokyo 1996
3. RICHARD P. BRENT, *Algorithms for Minimization without Derivatives,* Prentice-Hall, Englewood Cliffs, New Jersey, 1973

Design and implementation of a release framework for the KOJAK environment

Jens Rühmkorf

Lehrstuhl Prof. Speckenmeyer,
Institut für Informatik, Universität zu Köln

j.ruehmkorf@uni-koeln.de

Abstract: The KOJAK environment has been ported to Cray-T3E–UNICOS/mk and IA32–GNU/Linux. To address characteristics specific to the used computing platforms, a platform abstraction layer was designed and implemented. A release framework has been developed for the KOJAK environment to ease porting to further platforms in the future. The implementation of the framework relies on the GNU build system. Finally, the release framework has been adapted to the currently existing ports: Cray-T3E–UNICOS/mk, IA32–GNU/Linux, and SR8000–HI-UX/MPP.

KOJAK

Structural Overview of KOJAK

KOJAK stands for "Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks". It is a tool set for automatic performance analysis of MPI and OpenMP programs. KOJAK consists of four modules:

- The OPARI source-to-source translation tool (OpenMP Pragma And Region Instrumentor),
- the EPILOG tracing library (Event Processing, Investigating and LOGging),
- the EARL trace analysis language (Event Analysis and Recognition Language), and
- the EXPERT performance analysis tool (Extensible Performance Tool).

A KOJAK analysis process consists of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

First, users apply OPARI to the OpenMP C, C++ or Fortran source code which they want to analyse. By doing this, all OpenMP constructs and OpenMP library calls are redirected to appropriately instrumented wrapper functions at the source code level. For applications which do not use OpenMP, this step is omitted. For MPI programs, the standard PMPI profiling interface is used. All MPI, OpenMP and user-function instrumentations call functions of the EPILOG run-time library to generate event traces. These event traces are stored using the EPILOG binary trace data format.

Second, the resulting trace files from the first part serve as input for the EXPERT performance tool, which carries out the actual performance analysis and represents its results to the user. To make the analysis process simple and easy to extend, EXPERT uses EARL to map the event trace onto a higher level of abstraction.

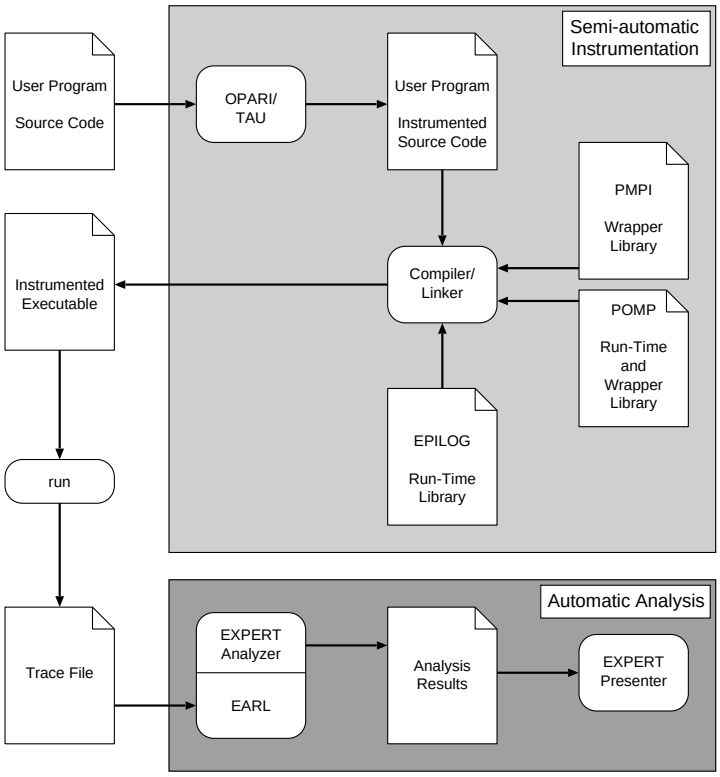


Figure 1: Structural overview of KOJAK

Technical Details of KOJAK's Modules

As the structure of KOJAK suggests, it makes sense to package OPARI and EPILOG together on the one hand and EARL and EXPERT on the other hand.

The software necessary to generate event traces, especially OPARI and EPILOG, has to be installed at the parallel computing facility the users run their application on:

- OPARI (when OpenMP is available)
  Language: mainly C++, some C and F90
  Provides: `bin/opari`, `pomp_lib.h`, etc.
- EPILOG
  Language: mainly C, some C++ and F90
  Build-Dependencies: MPI, → OPARI, OpenMP, optionally VTF3
  Provides: `libelg.a`, `libelg.mpi.a`, etc.

Generally all generated trace files will be transfered to a workstation where the user utilises EARL and EXPERT to analyse the event traces; the part of EPILOG which handles the processing of the trace files has to be installed onto the user's workstation as well:

- EARL
  Language: C++
  Maintainer-Dependencies: SWIG (for the Python bindings)
  Build-Dependencies: → EPILOG
  Provides: `libearlcmodule.so`, `python/site-packages/earl/earl.py`

- EXPERT
  Language: Python
  Build-Dependencies: → EARL

Platform Abstraction

For OPARI, all issues regarding different platforms could be addressed at source-code level in a portable way. Due to its nature, for EPILOG this could not be achieved in a similar way. Thus, a platform abstraction layer had to be introduced to overcome these obstacles.

For EARL and EXPERT the requirements were not as demanding in consequence of the nature of their runtime environment.

The Concept of EPILOG's Platform Abstraction Layer

Portability was achieved by supplying several C-functions that retrieve the necessary information in an appropriate manner for each platform.

(1) Platform identification:
- `char* elg_pform_name()`: The name of the platform EPILOG is installed on. Default is computed during install time of EPILOG and may be changed if not satisfying.

(2) Initialisation:
- `void elg_pform_init()`: Perform initialisation specific to this platform like identifying the clock rate.

(3) Event notification:
- `void elg_pform_mpi_init()`: To perform further initialisation after MPI has been initialised.
- `void elg_pform_mpi_finalize()`: To perform clean up before MPI is finalised.

(4) Clock:
- `int elg_pform_is_gclock()`: returns a value != 0 if the platform supports a global clock. Otherwise it returns 0.
- `double elg_pform_wtime()`: returns either the local or the global wall-clock time in seconds.

(5) File system:
- `char* pform_gdir()`: It is assumed that the platform provides a global file system and that the current working directory is part of it. For each platform a reasonable default is specified which may be changed during install time of EPILOG.
- `char* elg_pform_ldir()`: Returns the name of a local directory which can be used to store temporary trace files. Otherwise it returns `NULL`.

(6) SMP nodes:
- `int elg_pform_nodec()`: Number of SMP nodes installed.
- `int elg_pform_cpuc()`: Number of CPUs available at every SMP node.
- `long elg_pform_node_id()`: Returns a unique numeric identifier of the local SMP node.

- `char* elg_pform_node_name()`: Returns a unique string identifier of the local SMP node.

Handling of 32 Bit and 64 Bit Platforms

Compilers on 64 bit platforms often let the programmer choose between a 32 bit or 64 bit environment while defaulting to the 32 bit environment. In particular, the 32 bit environment usually sets int and pointer to 32 bits, while the 64 bit environment sets int and pointer to 64.

To avoid ambiguities, on 64 bit platforms EPILOG is compiled for both environments. For example, `libelg.32.a` and `libelg.64.a` are provided with `libelg.a` pointing to `libelg.32.a`. On 32 bit platforms `libelg.64.a` is simply omitted.

Porting EPILOG to Cray-T3E–UNICOS/mk

First `sysconf(_SC_CLK_TCK)` and `rtclock()` were used on CRAY-T3E for high resolution time-measurement; confer items (2) and (4):

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char** argv)
{
  int i;
  long time1, time2;
  double clockspeed, sec;

  /* only once at initialization time */
  clockspeed = 1.0/sysconf(_SC_CLK_TCK);
  printf ("clock rate = %.12f s\n", clockspeed);

  time1 = rtclock();
    work();
  time2 = rtclock();

  sec = (time2-time1) * clockspeed;
  printf ("work() took %.12f s\n", sec);

  exit(0);
}
```

Additionally `sysconf(_SC_CRAY_MAXPES)` and `sysconf(_SC_CRAY_LPE)` were used, which return the number of SMP nodes installed and the logical number of the local SMP node, respectively; confer item (6).

Porting EPILOG to IA32-GNU/Linux

Initially, EPILOG had been implemented on the ZAMpano cluster [13]. To make EPILOG available to other, commonly available IA32-GNU/Linux computing platforms, the requirements of the platform

abstraction layer had to be fulfilled.

Unlike usual supercomputing platforms, GNU/Linux does not provide standardised methods for high resolution time-measurement. However, on most IA32-platforms, as will be shown, there are techniques to circumvent this; confer item (2) and (4).

This dissatisfying situation might change for GNU/Linux in the near future: there is a project which works on providing "High Resolution POSIX Counters for GNU/Linux"[3]. The feature freeze for the Linux 2.6 kernel has been set by Linus Torvalds to October 31, 2002; at the time of writing this it is not sure whether this project's patches will make it into kernel 2.6.

The IA32 Time-stamp Counter

The RDTSC (read time-stamp counter) had been introduced by Intel for the Pentium processor in 1993. The time-stamp counter is a 64-bit model specific register (MSR) that is incremented every clock cycle.

The usage of the RDTSC instruction is not difficult. Corresponding inline assembly code for measuring time with a clock rate resolution would look like this:

```
unsigned __int32 time, time_low, time_high;
unsigned __int32 hz = 550000000;  // 550 Mhz CPU

__asm {
  RDTSC                     ; read time stamp
  mov    time_low,  eax     ; read low 32bit
  mov    time_high, edx     ; read high 32bit
}

work();

__asm {
  RDTSC                     ; read time stamp
  sub    eax, time_low      ; compute the difference
  sub    edx, time_high     ;
  div    hz                 ; unsigned divide EDX:EAX by hz
  mov    time, eax          ; compute ellapsed time
}
```

On IA32–GNU/Linux systems it can be easily verified whether the underlying platform supports time-stamp counters using the `/proc`-filesystem by looking for the `tsc` flag:

```
zam008[~]> cat /proc/cpuinfo
processor      : 0
vendor_id      : GenuineIntel
model name     : Pentium III (Katmai)
cpu MHz        : 549.087
cache size     : 512 KB
...
flags          : fpu vme de pse tsc msr ...
zam008[~]>
```

Most modern IA32 processors support the time-stamp counter. For GNU/Linux the following processor types provide this counter:

`Pentium-Classic`	Intel Pentium
`Pentium-MMX`	Intel Pentium MMX
`Pentium-Pro`	Intel Pentium Pro/Celeron/Pentium II
`Pentium-III`	Intel Pentium III and Celerons based on the Coppermine core
`Pentium-4`	Intel Pentium 4
`K6`	AMD K6, K6-II and K6-III (aka K6-3D)
`Athlon`	AMD K7 family (Athlon/Duron/Thunderbird)
`Winchip-2`	IDT Winchip 2
`Winchip-2A`	IDT Winchips with 3dNow! capabilities
`CyrixIII`	VIA Cyrix III or VIA C3

Issues Affecting the Cycle Count

Processors of the Pentium-family support out-of-order execution, where instructions are not necessarily performed in the order they appear in the source code. This can be a serious issue when using the RDTSC instruction, because it could potentially be executed before or after its location in the source code, giving a misleading cycle count.

In order to keep the RDTSC instruction from being performed out-of-order, a *serialising instruction* is required. A serialising instruction will force every preceding instruction in the code to complete before allowing the program to continue. One such instruction is the CPUID instruction, which is normally used to identify the processor on which the program is being run.

In addition to the need for serialising instructions, cache effects have also to be taken into consideration. Using RDTSC on the same section of code can often produce very different results: the first time a line of data or code is brought into cache, it takes a large number of cycles to transfer it into memory.

Cache effects can only be removed for small sections of code. The technique basically requires moving the instructions and data involved in the L1 cache, which is the cache closest to the processor. Thus all effects of transactions between both memory to data cache and memory to instruction cache are removed. This technique of storing memory into a cache before it is actually used is known as "cache warming".

Since cache effects can only be removed for small sections of code and this technique would usually require more involvement by the user than desired, this method has not been taken into consideration yet. For repeated code sections there is also the option of taking a time average for such sections. Due to the unknown structure of the measured code this is also no option.

The GNU Build System

When developing software in UNIX environments, portability of code is important. Although several organisations have developed standards such as POSIX.1 or ISO/IEC 9899:1999, developing software which is portable is no easy task. From the maintainer’s point of view, it is desirable not to be too concerned with system specific details while still attaining portability.

The Free Software Foundation provides several tools which aid the user in the build process. For this project, GNU autoconf, automake and libtool were used.

GNU autoconf is essentially a shell script compiler, which reliably discovers build and runtime informa-

tion. It takes a file `configure.ac` (or `configure.in`) as its input and converts all occurring GNU M4 macros within that file into portable shell script code, the `configure`-script.

GNU automake is a tool that lets the user specify the build needs in a `Makefile.am` with a vastly simpler syntax than that of a plain `Makefile`, and then generates a portable `Makefile.in` for use with autoconf.

Producing shared libraries portably is no easy task – each system has its own incompatible tools, compiler flags, etc. GNU libtool addresses this situation and tries to handle all the requirements of building shared libraries. Not all platforms are supported yet, at the time of writing this, e.g. Cray-T3E–UNICOS/mk and SR8000–HI-UX/MPP are not supported. Most common "workstation"platforms are supported though.

The tools briefly described above provide several programs which closely interact. For hints regarding the appropriate usage of these programs and important implementation details we advise the reader to consult the file `README.maintainer` which is part of the distribution of the KOJAK environment.

Usage of the Release Framework

When porting the software to further platforms, a file `elg_pform_NEWPFORM.c` which corresponds to `elg_pform.h` has to be implemented. Additionally, support for the new platform has to be added to `configure.ac`, which is used by `autoconf`. Only minor changes have to be done to achieve this, as this excerpt from `configure.ac` shows:

```
# ${host} is determined through GNU autoconf
case ${host} in
  hppa*-hitachi-*)
    AC_CONFIG_LINKS(src/elg_pform.c:src/elg_pform_hitachi.c)
    AC_SUBST([ELG_PFORM], [hitachi])
  ;;
  *newplatform*)
     AC_CONFIG_LINKS(src/elg_pform.c:src/elg_pform_newplatform.c)
     AC_SUBST([ELG_PFORM], [newplatform])
  ;;
esac
```

With the current implementation of KOJAK's release framework in place, installing one of KOJAK's modules does not differ much from the usual procedure: `./configure`, `make`, `make install`.

The following options are specific to EPILOG:

`--enable-omp`	Compile for use with OpenMP (default is no).
`--enable-user-instrument=VAL`	Enable user instrumentation (one of 'generic', 'hitachi', 'pgi'; defaults to 'generic').
`--with-pform-name=STRING`	Specify the name of the platform EPILOG is installed on (is determined through `./configure`, e.g. 'sparc-solaris').
`--with-pform-gdir=DIR`	Specify a directory which is part of the systems global filesystem (defaults to '`.`').
`--with-pform-ldir=DIR`	Specify a local directory which can be used to store temporary files (defaults to '`/work`').

`--with-pform-nodec=INT`	Specify number of SMP nodes installed (depending on the platform this can be determined during runtime).
`--with-pform-cpuc=INT`	Specify number of CPUs available at every SMP node (depending on the platform this can be determined during runtime).
`--with-mpi-ldflags=MPI-LDFLAGS`	Specify which LDFLAGS are required for linking MPI programs (defaults is none)
`--with-omp-flags=OMP-FLAGS`	Specify which flags the C, C++ and F90 compiler require for OpenMP programs (defaults is none).
`--with-cc-omp-flags=CC-OMP-FLAGS`	specify which flags the C compiler requires for OpenMP programs (if different from C++, F90; default is none).
`--with-cxx-omp-flags=CXX-OMP-FLAGS`	Specify which flags the C++ compiler requires for OpenMP programs (if different from C, Fortran; default is none).
`--with-f90-omp-flags=F90-OMP-FLAGS`	specify which flags the F90 compiler requires for OpenMP programs (if different from C, C++; default is none)

The following options are common to other modules as well:

`--with-ldflags=LDFLAGS`	Specify which LDFLAGS to use (default is none)
`--with-cc=CC`	Specify which C compiler to use (default is `mpicc`).
`--with-cflags=CFLAGS`	Specify which CFLAGS to use (defaults to '`-g -O2`' for the GNU C compiler or '`-g`' for other compilers).
`--with-cxx=CXX`	Specify which C++ compiler to use (default is `g++`).
`--with-cxxflags=CXXFLAGS`	Specify which CXXFLAGS to use (defaults to '`-g -O2`' for the GNU C++ compiler or '`-g`' for other compilers).
`--with-f90=F90`	Specify which F90 compiler to use (necessary for OpenMP-support, default is none)
`--with-f90flags=F90FLAGS`	Specify which F90FLAGS to use (default is none).

References

1. **F. Wolf, B. Mohr,** *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*, accepted for PDP 2003.
2. **T. Bowden, B. Bauer, J. Nerin,** *The* `/proc` *filesystem*, Linux Kernel Documentation for version 2.4.19, 2002.
3. **J. Mehaffey,** *High Resolution POSIX Timers for Linux*, The Real-time and Embedded Systems Forum, Anaheim, 2002.
4. **D. MacKenzie, B. Elliston, A. Demaille,** *Autoconf*, Free Software Foundation, Edition for Autoconf version 2.53, 2002.
5. **D. MacKenzie, T. Tromey,** *Automake*, Free Software Foundation, Edition for Automake version 1.6.1, 2002.
6. **G. Matzigkeit, A. Oliva, T. Tanner, G. V. Vaughan,** *Libtool*, Free Software Foundation, Edition for Libtool version 1.4.2, 2002.
7. **B. Mohr, A. D. Malony, S. Shende, F. Wolf,** *Design and Prototype of a Performance Tool Interface for OpenMP*, Interner Bericht FZJ-ZAM-IB-2001-09, Forschungszentrum Jülich, 2001.
8. **T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, J. L. Träff,** *Knowledge Specification for Automatic Performance Analysis, APART Technical Report*, Interner Bericht FZJ-ZAM-IB-2001-08, Forschungszentrum Jülich, 2001.
9. **B. Mohr, F. Wolf,** *Knowledge Specification Analysis for SMP Cluster Applications*, Interner Bericht FZJ-ZAM-IB-2001-05, Forschungszentrum Jülich, 2001.
10. *UNICOS/mk System Calls Reference Manual*, Cray, Inc., 2001.
11. *HI-UX/MPP for SR8000 Programmer's Reference*, Hitachi, Ltd., manual number 6A30-3-024-08(E), 2001.
12. Leibniz Rechenzentrum der Bayerischen Akademie der Wissenschaften. Hitachi SR8000-F1. http://www.lrz-muenchen.de/. Setup in 2000.
13. Forschungszentrum Jülich. ZAMpano (ZAM parallel nodes). http://zampano.zam.kfa-juelich.de/. Setup in 2000.
14. *Using the RDTSC Instruction for Performance Monitoring*, Intel Corporation, 1997.
15. Forschungszentrum Jülich. Cray T3E-1200. http://www.fz-juelich.de/zam/CompServ/services/sco.html. Setup in 1997.
16. **T. Mathisen,** *Pentium Secrets*, BYTE 07/1994.

Interaktive Realzeit-Visualisierung des Verhaltens komplexer chemischer Systeme

Eduard Schreiner

Lehrstuhl für Theoretische Chemie
Ruhr-Universität Bochum

E-mail: eduard.schreiner@theochem.ruhr-uni-bochum.de

Zusammenfassung: Die Bibliothek SVT (Scientific Visualization Toolkit) wurde um die Funktionalität erweitert, aus MD-Simulationen stammende Trajektorien in Form einer interaktiven Animation darzustellen. Die Implementierung ist erweiterbar gestaltet und beinhaltet Funktionen zur Manipulation des Ablaufs der Animation und der animierten Objekte. Zusätzlich wurde ein GUI zur Bedienung der neuen Features programmiert.

Einleitung

Die Molekulardynamik (MD) ist eine Modellierungsmethode, die die Berechnung der zeitlichen Evolution eines chemischen Systems aus gegebenen Anfangsbedingungen erlaubt [1]. Zum Beispiel wird ein Proteinmolekül bei bestimmten Anfangsbedingungen wie Druck, pH-Wert, Ionenstärke und Temperatur in einem Wassertropfen platziert und im Laufe der Simulation wird seine Faltung unter diesen Bedingungen untersucht werden. Zur Zeit existieren zwei Hauptrichtungen in der MD. Bei der Molekularmechanik (MM) werden die klassischen Bewegungsgleichungen für jedes einzelne Molekül im Kraftfeld aller Nachbarmoleküle gelöst. Die Hybridmethode QM/MM vereinigt die Molekularmechanik mit der Quantenmechanik in der Weise, daß ein Teil des Systems klassisch mit MM behandelt wird, der andere mit quantenmechanischen Methoden. Zur Untersuchung von chemischen Reaktionen eignet sich die MM nicht, da bei dieser Methode Bindungen weder geknüpft noch gebrochen werden können. Die Quantenmechanik unterliegt nicht diesen Restriktionen und wird daher für eben diesen Zweck eingesetzt.

Das Resultat einer Simulation ist eine Trajektorie, die die Dynamik eines Systems beschreibt und sich aus Schnappschüssen in regelmäßigen Abständen simulierter Zeit zusammensetzt. Es sind typischerweise bis zu 100000 Einzelkonfigurationen, die insgesamt den Verlauf einer chemischen Reaktion, Scherverhalten von Werkstoffen oder Faltung von Proteinen in der Zeit beschreiben, ähnlich wie Messungen während eines laufenden Experiments.

Da die heute simulierten Systeme groß sind, typischerweise $10^3 - 10^6$ Atome, und die räumliche Anordnung der Systemteile zu einander eines der wichtigsten Ergebnisse der Rechnung darstellt, ist die Visualisierung des Verlaufs der Simulation eine unverzichtbare Methode, um sich schnell einen Eindruck von den simulierten Vorgängen zu verschaffen.

Als Ausgangspunkt für die vorliegende Arbeit diente die VR-Visualisierungs-Bibliothek SVT (Scientific Visualization Toolkit)[2]. Sie wird im ZAM (Zentralinstitut für angewandte Mathematik, FZ Jülich) von Stefan Birmanns, Herwig Zilken und Frank Delonge entwickelt und erlaubt die Darstellung statischer dreidimensionaler Szenen, wie der Anordnung von Atomen in einem Molekül. Die Bibliothek basiert auf OpenGL und ist plattformunabhängig, d. h. sie läßt sich auf einem PC oder einer SGI-Workstation, unter Windows oder unter Linux, an einer Holobench oder an einem normalen Bildschirm

verwenden. SVT unterstützt Stereo-Darstellung mittels einer *Shutter-Brille*, Kraftrückkopplungsgeräte, 3D-Eingabegeräte und Tracking-Systeme, mit deren Hilfe die Position des Beobachters bestimmt und die Darstellung dementsprechend angepaßt werden kann. Genauere Ausführungen zu SVT sind [2] zu entnehmen.

Vorüberlegungen

Die SVT-Bibliothek diente bisher unter anderem zur Visualisierung statischer molekularer Szenarien, die aus nur einem Einzelbild bestehen. Da es sich bei den Ergebnissen einer MD-Simulation jedoch um dynamische Prozesse handelt, sollte die Darstellung ebenfalls dynamisch sein. Für eine dynamische zeitabhängige Visualisierung stehen prinzipiell zwei Alternativen zur Verfügung. Die eine ist ein Film, die andere ist eine Realzeit-Animation. Der Film besteht aus vielen Einzelbildern und kann in einer sehr hohen Auflösung auf einem leistungsfähigen Rechner erstellt und auf einem langsameren ohne Schwierigkeiten dargestellt werden. Man kann jedes einzelne Bild als Standbild betrachten, den Film vorwärts oder rückwärts abzuspielen und er kann, einmal angefertigt, beliebig oft betrachtet werden. Der Nachteil liegt jedoch im festen Ablauf: die Darstellungsweise, die Ausrichtung, die Größe und die Position des dargestellten Objekts können zum Zeitpunkt des Betrachtens nicht mehr variiert werden. Für die Untersuchung eines molekularen Systems sind diese Features jedoch essentiell. Die Realzeit-Animation bietet all diese Optionen. Sie entsteht zum Zeitpunkt des Betrachtens und erlaubt daher Transformationen wie die Rotation für jedes einzelne Objekt individuell.

Die Manipulation einer Animation kann auf zweifache Weise erfolgen: im Raum (Rotation, Translation, Skalierung oder Selektion von Systemteilen) oder in der Zeit (Springen zu beliebigen Zeitpunkten der Trajektorie, sie vorwärts oder rückwärts in der Zeit betrachten oder anhalten). Diese Manipulationen dürfen sich nicht gegenseitig stören, sondern müssen entweder vollkommen unabhängig von einander sein oder auf definierte Weise miteinander interagieren.

In Anbetracht der Komplexität der heute simulierten Systeme ($10^3 - 10^6$ Atome) bedeutet die Realzeit-Animation jedoch einen enormen Aufwand bei der Visualisierung, da alle Objekte in Echtzeit gerendert werden müssen. Pro Sekunde sollten mindestens 20 Bilder aufgebaut werden. Bei eine Repräsentation von Atomen als Kugeln und von Bindungen als Zylinder müssen in jedem Bild ca. 256 Dreiecke pro Atom bzw. Bindung gerendert werden. Insgesamt ergeben sich größenordnungsmäßig $10^7 - 10^{10}$ zu rendernde Dreiecke pro Sekunde. Man sollte sich daher im Vorfeld überlegen, welche Systemteile man untersuchen will, und die Trajektorie dementsprechend auf diese Teile beschränken.

Die Länge der Trajektorie stellt den Betrachter vor das Problem, daß es zu viel Zeit in Anspruch nimmt, sich die Trajektorie komplett durchzuschauen. Bei 20 Bildern pro Sekunde und einer Trajektorienlänge von 100000 Einzelkonfigurationen würde das reine Abspielen über eine Stunde dauern. Es sollte daher einen Zeitraffer-Animationsmodus geben, d. h. eine Möglichkeit nur jede n-te Konfiguration darzustellen. Dies ermöglicht auch die Darstellung größerer Systeme in einer akzeptablen Zeit. Zusätzlich würde eine mögliche Auswahl eines Zeitfensters aus der Trajektorie und eine Unterbrechung der Animation z.B. während einer Rotation des animierten Objekts die Arbeit weiterhin vereinfachen.

Ein Molekül besteht nicht nur aus Atomen sondern auch aus Bindungen. Der Aufwand bei der Berechnung der Bindungen während der Visualisierung skaliert bestenfalls mit $O(NlogN)$. Diese Arbeit muß bei einer Animation für jedes Einzelbild durchgeführt werden. In MM-Trajektorien sind die Bindungen jedoch über die gesamte Trajektorienlänge fix. Das Gleiche trifft auf den größten Teil einer QM/MM-Trajektorie zu. Es ist daher angebracht nach einer Methode zu suchen, den mit der Berechnung der Bindungen verbundenen Overhead zu minimieren. Die Methode sollte jedoch die Visualisierung sowohl von MM- als auch von QM/MM-Trajektorien ermöglichen.

Es existiert eine Fülle von Formaten, in denen eine Trajektorie vorliegen kann, z. B. XYZ, PDB, DCD,

CRD, NAMDBIN ... Bei dem vorliegenden Projekt entschied man sich für das Paar PSF (protein structure file) und DCD (dynamical coordinates data). Die PSF-Datei definiert die Zusammensetzung des Systems und stellt gleichzeitig eine Art Schablone für die DCD-Datei dar, die die Koordinaten jedes Teilchens zu jedem Zeitpunkt enthält. Der Vorteil dieser Formate ist, daß sie sehr einfach und sehr standardkonform sind. Das PSF-Format ist ein ASCII-Format und existiert in lediglich zwei leicht unterschiedlichen Subtypen, CHARMm und X-PLOR, die sich leicht in einander überführen lassen. Das Gleiche gilt für das binäre DCD-Format.

Das Konzept

In diesem Abschnitt soll das Lösungskonzept zu den oben dargestellten Problemen, das den neu implementierten Teilen zugrunde liegt, erläutert werden.

Die Selektion des darzustellenden Teils eines molekularen Systems erfolgte im vorliegenden Fall durch ein Programm, das von mir gegenwärtig am Lehrstuhl für theoretische Chemie in Bochum entwickelt wird und eigentlich der numerischen Auswertung von Trajektorien dient.

Die gesamte Trajektorie wird aus den Eingabedateien eingelesen und in Form eines SVT-Objekts im Speicher gehalten. Es setzt sich aus Atomen und Bindungen zusammen, die ebenfalls SVT-Objekte darstellen. Diese sind bereits in der Form, wie sie für die Darstellung statischer Moleküle benötigt wird, implementiert. Um die Zeitabhängigkeit zu berücksichtigen, soll nicht jede Konfiguration in einer eigenen Szene dargestellt werden, sondern der Konfigurationswechsel in ein und derselben Szene durch eine räumliche Transformation der darin enthaltenen Objekte stattfinden. Dies würde den Speicheraufwand, der gerade bei großen Systemen ein Poblem darstellt, vermindern und die Animation beschleunigen.

Der Aufwand für die Berechnung der Bindungen während der Visualisierung minimiert sich durch die Formatwahl: die PSF-Datei enthält Informationen über Konnektivitäten. Für eine MM-Trajektorie ist dies hinreichend, da die Bindungen sich im Verlauf der Trajektorie nicht ändern. Für eine QM/MM-Trajektorie ist die Strategie, die Trajektorie in zwei Teile zu zerlegen. Für den einen werden Bindungen aus der PSF-Datei übernommen, für den anderen liest man die Bindungen nicht ein, sondern berechnet sie zur Laufzeit der Animation. Der mit der Berechnung der Konnektivitäten verbundene Aufwand ist in diesem Fall vernachlässigbar, da die QM-Teile nur einen geringen Bruchteil des Gesamtsystems darstellen.

Während der Visualisierung wird in SVT eine Szene aufgebaut, die aus verschiedenen hierarchisch angeordneten Objekten besteht. Objekte können Kameras, Lichtquellen, Moleküle oder Knoten, die ebenfalls aus Objekten aufgebaut sind, sein. Alle Objekte eines Knotens werden bei einer Transformation des Knotens wie Rotation oder Translation gleich behandelt. Dies bietet die Möglichkeit, daß eine räumliche Transformation an einer Konfiguration sich auf alle übrigen auswirkt, indem man die Objekte, die die Trajektorie repräsentieren, an einen Knoten koppelt. Rotationen, Translationen oder Skalierungen betreffen nun alle Objekte, die diesem Knoten zugehören. Das bedeutet, daß wenn eine Konfiguration rotiert wird, alle nachfolgenden Konfigurationen es ebenfalls sind. Gleichzeitig entkoppelt dieses Konzept die räumliche Transformation von der zeitlichen. Diese spielt sich in einem Objekt des Knotens ab und hat keine unerwünschten Effekte auf andere Objekte. Eine erwünschte Wechselwirkung zwischen den beiden Arten von Transformationen ist das Anhalten der zeitlichen Transformation, während eine räumliche durchgeführt wird. Dies ermöglicht es dem Betrachter, das Objekt in der Konfiguration zu untersuchen, in der er es sehen möchte, ohne die Animation abbrechen zu müssen.

Insgesamt müssen folgende Funktionalitäten für die Manipulation des zeitlichen Ablaufs implementiert werden:

1. Abspielen der Trajektorie vorwärts und rückwärts in der Zeit

2. Abspielen mit einem Inkrement von ≥ 1, d. h. Überspringen einer beliebigen anzugebenden Anzahl von Konfigurationen während des Abspielens (Zeitrafferdarstellung)

3. Auswahl eines Zeitfensters aus der Gesamtlänge der Trajektorie

4. Einzelschritte um nur ein Inkrement vorwärts oder rückwärts in der Zeit

5. Springen zu einer beliebigen Konfiguration in der Trajektorie

6. Anwählen der ersten bzw. letzen Konfiguration

7. Abbruch und gleichzeitige Rückehr zur Ausgangskonfiguration

8. Pause, bei der die Animation angehalten, aber nicht auf Ausgangskonfiguration gesetzt wird, so daß sie anschließend von der aktuellen Position fortgeführt werden kann

Die Animation und allgemein ein Wechsel zu einem bestimmten Zeitschritt der Trajektorie kommt durch synchronen Zugriff auf die korrespondierenden Koordinaten aller Atome zustande. Die Repräsentationen der Bindungen werden dabei neu berechnet. Der Ablauf der Animation wird durch eine Endlosschleife gesteuert, die unter Umständen unterbrochen oder abgebrochen werden kann.

Es ergibt sich eine Fülle von Funktionen, die am einfachsten durch ein GUI gesteuert werden können. Die Entwicklungsumgebung Qt Designer 2.0 von TROLLTECH bietet sehr gute Möglichkeiten, ein solches Bedienungselement zu programmieren. Dieses sollte intuitiv und einfach zu bedienen sein. Qt Designer ermöglicht die Benutzung der Qt-Objekte und gleichzeitig die Definition eigener Slots für jedes Objekt [3]. Dadurch können Funktionen von nicht Qt-Objekten angesteuert werden.

Implementierung

Da SVT in der Programmiersprache C++ geschrieben ist, wurden die Erweiterungen ebenfalls in C++ durchgeführt.

Um die gestellten Anforderungen zu erfüllen, mußten einige bestehende Klassen der SVT-Bibliothek verändert und erweitert werden. Die Klassen svt_pdb, svt_atom und svt_bond wurden bisher zur Visualisierung statischer Systeme verwendet. Es bot sich daher an, an dieser Stelle anzusetzen.

Jedes dieser Objekte wurde von der Klasse svt_node abgeleitet, die einen Knoten repräsentiert, und verfügt über eine Funktion, die es dazu befähigt sich selbst zu zeichnen, `draw_GL()`. Die `draw_GL()`-Funktion der Klasse svt_pdb steht dabei hierarchisch über den beiden anderen, da diese Klassen Bestandteile von svt_pdb sind. Das bedeutet, daß die Implementierung der Animation an dieser Stelle geschehen sollte.

svt_pdb(1)

In dieser Klasse gab es die umfangreichsten Modifikationen und Erweiterungen. Es wurden ein neuer Konstruktor und 25 neue Member-Funktionen geschrieben.

In erster Linie mußten Lesefunktionen für die neuen Formate implementiert werden. Um eine Erweiterung auf zusätzliche Formate zu erleichtern, wurde eine Funktion (`loadFiles(svt_str)`) erstellt, die aus dem Format der Dateien entscheidet, welche Lesefunktion aufgerufen werden soll. Ein weiterer Vorteil dieser Vorgehensweise ist, daß man eine Trajektorie in Form einer PSF-Datei und mehreren zugehörigen DCD-Dateien als Input liefern kann.

Das Objekt, das die Dateien darstellt, wurde als ein STL-Vektor von STL-maps implementiert. Die map setzt sich aus einem integer als key und einem string als value zusammen. Der key beschreibt das Format, der string den Dateinamen. Dieses Konstrukt wird an den Konstruktor von svt_pdb übergeben, der mit ihm als Argument `loadFiles` aufruft. Mit Hilfe von Iteratoren wird der gesamte Vektor durchlaufen, anhand des keys das Format bestimmt und mit dem Dateinamen als Argument die entsprechende Lesefunktion aufgerufen.

svt_atom

Diese Klasse beschreibt ein Atom samt seinen Eigenschaften wie Ladung oder Masse. Jedes Atom hat zusätzlich seine eigenen Koordinaten. Bisher wurden die Positionen durch einen svt_vector4f beschrieben. Dies wurde in der Weise geändert, daß sie nun in einem svt_array von SVT-Vektoren gehalten werden, um die Position jedes Atoms für jede Konfiguration zu speichern. Der Vorteil liegt auf der Hand: jedes Atom ”kennt” seine Koordinaten zu jedem Zeitpunkt, und man muß nicht für alle Zeitpunkte komplette Atom-Objekte im Speicher halten. Jedes Atom enthält somit seine eigene Dynamik. Alle Funktionen, die bisher auf die einzelnen Koordinaten zugegriffen haben, mußten nun auf das neue Koordinatenobjekt angepaßt werden. Zusätzlich wurden zwei Funktionen hinzugefügt, um den Zugriff auf die Koordinaten eines bestimmten Zeitpunkts zu ermöglichen.

svt_bond

Die Klasse svt_bond repräsentiert eine Bindung in Form von Zeigern auf zwei Atome. Die Implementierung als Zeiger machte es recht einfach, die Klasse auf die aktuelle Situation auszudehnen. Es wurde lediglich ein Funktionsaufruf in die `drawGL()`-Funktion hinzugefügt: die Zylinder, die die Bindungen in bestimmten Darstellungsmodi bilden, müssen für jede Konfiguration neu gerendert werden.

svt_pdb(2)

Aus `loadFiles` heraus wird eine PSF-Lesefunktion aufgerufen (`readPSF(const svt_str &)`), die ihrerseits Funktionen zum Lesen der einzelnen Einträge der PSF-Datei aufruft. An dieser Stelle werden alle Atome initialisiert und ihre Koordinaten zunächst auf den Ursprung gesetzt. Ferner werden die Bindungen eingelesen. Danach wird wiederum aus `loadFiles` eine Funktion zu Einlesen der DCD-Datei gestartet (`readDCD(const svt_str &)`), die ihrerseits Funktionen zum Lesen der einzelnen Records der DCD-Datei aufruft. Dabei werden die korrekten Koordinaten den Atomen zugewiesen (s. Abb. 1). Nach erfolgreichem Einlesen ergibt sich der in Abb. 2 dargestellte Objektaufbau.

Die Funktion `drawGL()` ist für die Zeichnung des svt_pdb-Objektes zuständig, indem sie die `drawGL()`-Funktionen der beiden anderen Klassen aufruft. Um eine Animation zu realisieren, wurde um diese Funktion eine Endlosschleife errichtet. Nach einer Abfrage einiger Flags, die die Möglichkeit einer Animation bestätigen, kann sie durch `playForward()` bzw. `playBackward()` in der jeweiligen zeitlichen Richtung gestartet werden. Beim Durchlaufen der Schleife wird die Konfigurationsnummer um das aktuelle Inkrement erhöht und in allen Atomen aktualisiert. Die `drawGL()`-Funktionen für Bindungen und Atome werden aufgerufen und die nächste Schleifeniteration beginnt. Der gesamte Vorgang ist schematisch in Abbildung 3 dargestellt. Der Szenengraph behält während der gesamten Animation seine feste Struktur, lediglich die Elemente in ihm werden mit der Zeit selbst intern transformiert.

Das Bedienungselement

Das Bedienungselement wurde mit dem Qt Designer 2.0 von TROLLTECH erstellt. Diese Entwicklungsumgebung ermöglicht die Benutzung von Qt-Objekten und gleichzeitig die Definition eigener Slots

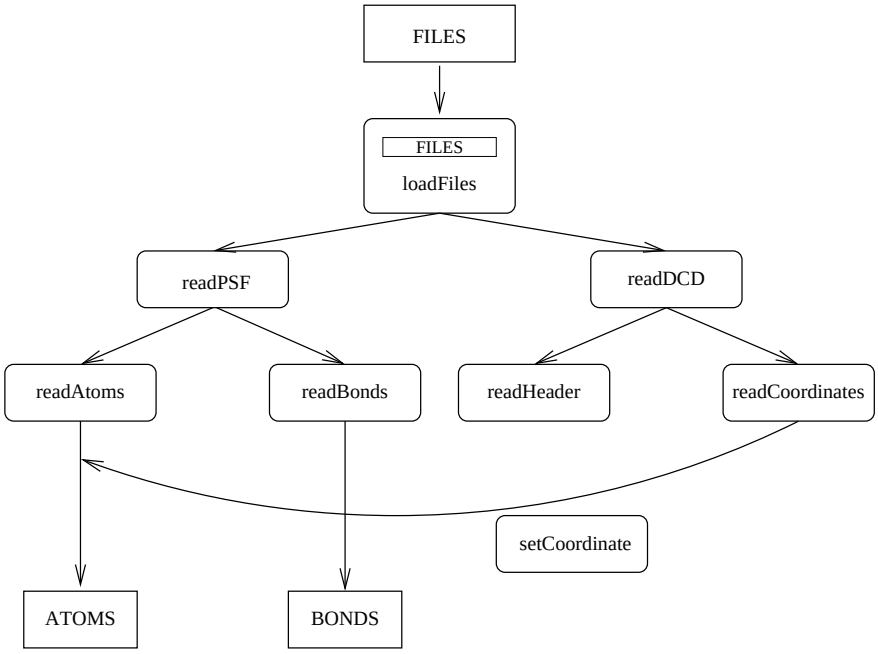


Abbildung 1: Aufbau der Datei-Leselogik von svt_pdb. Die scharfkantigen Rechtecke stellen SVT-Objekte dar, die abgerundeten Funktionen. Die zeitliche Abfolge der Leseoperationen ist von rechts nach links und von oben nach unten angeordnet

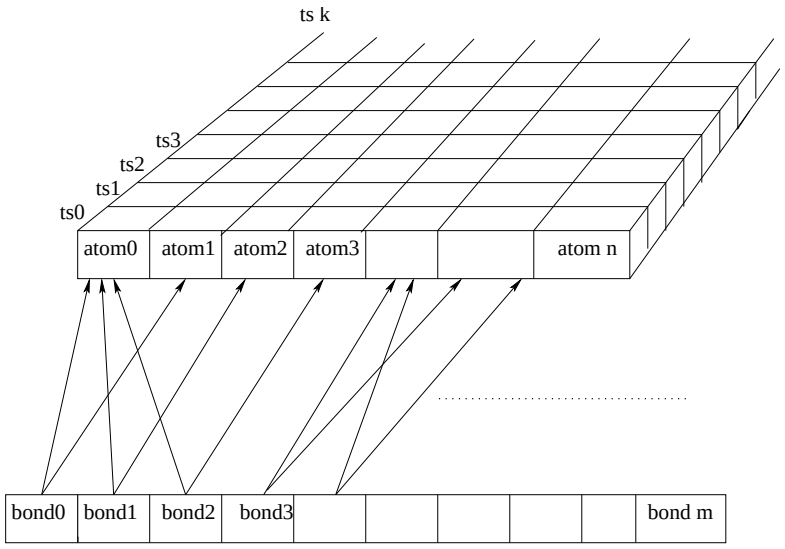


Abbildung 2: Schematische Darstellung der relevanten Objekte in svt_pdb. Atome sind in einem Feld der Größe n enthalten, die der Anzahl der Atome im System entspricht. Jedes Atom besitzt ein Feld der Größe k, in dem die Koordinaten zu jedem Zeitpunkt (ts) gehalten werden. Die Bindungen sind in einem Feld der Größe m gespeichert und stellen Zeiger auf die zugehörigen Atome dar.

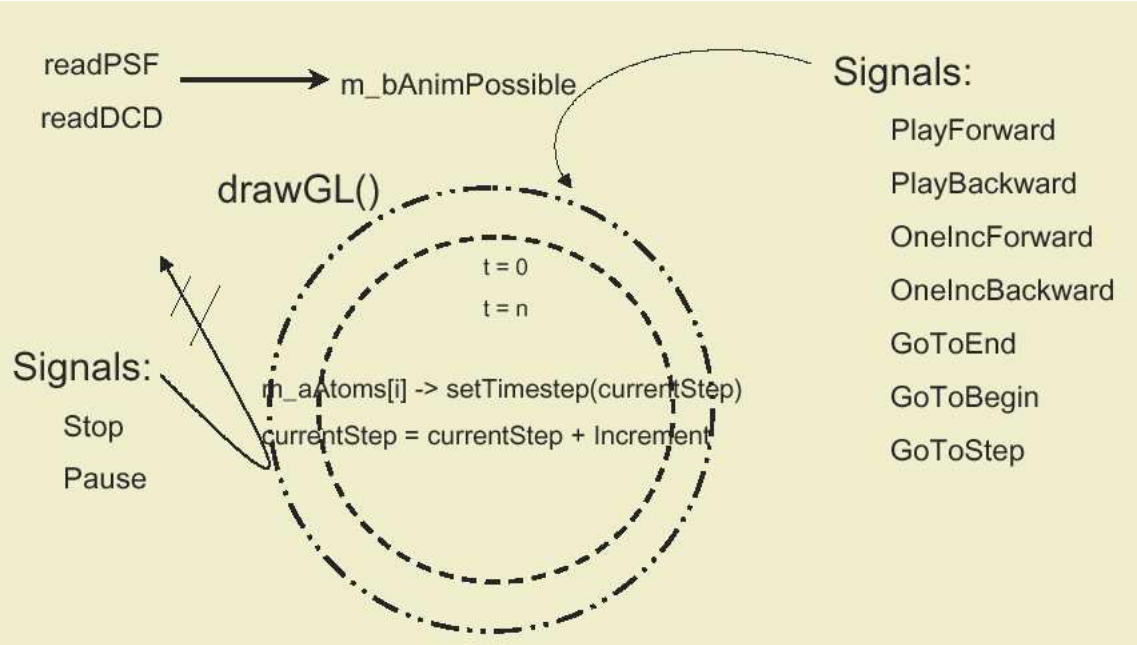


Abbildung 3: Schematische Darstellung des Ablaufs der Animation

für jedes Objekt [3]. Durch diese Slots können außerhalb von Qt implementierte Objekte angesteuert werden. Für jede der Funktionen, die den Animationsmodus verändern, wurde ein Slot in dem Applet implementiert und mit den entsprechenden Signalen der das GUI aufbauenden Qt-Objekte verbunden.

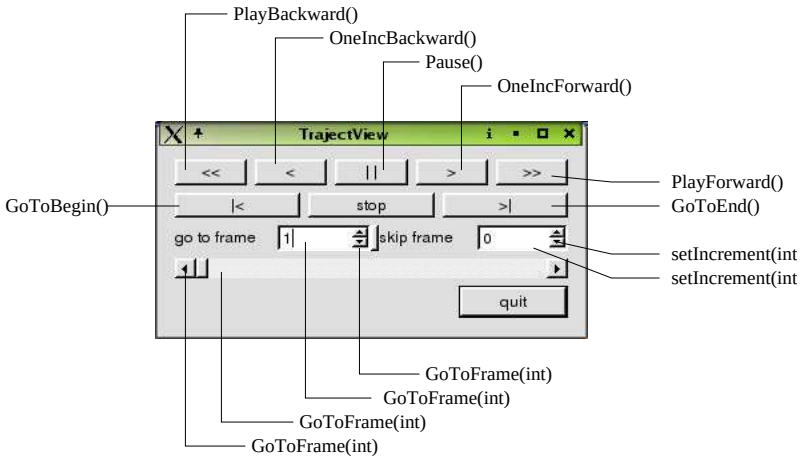


Abbildung 4: Das Bedienungselement für die Steuerung der Animationsmodi

Die Spinboxen *”go to frame”* und *”skip frame”* können durch direkte Eingabe einer Zahl oder durch die Betätigung der Pfeiltasten bedient werden. Im letzten Fall verändert sich die Eingabe um 1. Die Bildlaufleiste kann in analoger Weise bedient werden: entweder man bedient den Schieber, wobei die Trajektorie um eine von der Länge der Bildlaufleiste abhängigen Anzahl von Schritten fortschreitet, oder man bedient die Pfeile an den Enden der Leiste, um eine Veränderung der Konfigurationsnummer um 1 zu erreichen. Alle Funktionalitäten der Bildlaufleiste und der Spinbox *”go to frame”* sprechen die selbe Funktion in svt_pdb an.

Tests

Die neue Funktionalität wurde an drei jeweils 1000 Konfigurationen langen Trajektorien getestet. Die zu animierenden Systeme setzten sich wie folgt zusammen:

- 102 Atome, 100 Bindungen
- 162 Atome, 163 Bindungen
- 222 Atome, 224 Bindungen

Getestet wurde an einem PIII 750MHz mit 512 MB Hauptspeicher und einer Gloria III Graphikkarte mit NVIDIA QuadroII Chipsatz und 64 MB RAM. Als Darstellungsmodi wurden *line* und *cpk* gewählt (s. Abbildung 5).

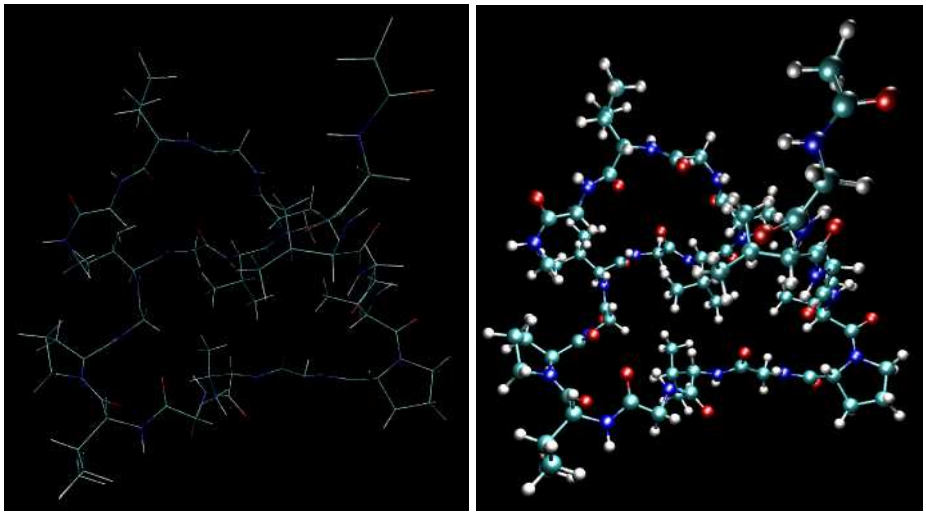


Abbildung 5: Das dritte System in *line*- bzw. *cpk*-Darstellung

Im line-Modus konnten die Systeme bei laufender Animation mit 200, 127 bzw. 90 fps dargestellt werden. Man sieht deutlich, daß die Anzahl der dargestellten Bilder pro Sekunde sich umgekehrt proportional zur Systemgröße verhält. Im cpk-Modus konnten 60, 40 bzw. 67 fps erreicht werden. Die Abweichung in diesem Fall könnte mit der geringen Menge der Messwerte zusammenhängen.

Ein vierter Test wurde mit einem System bestehend aus 27000 Atomen und 18000 Bindungen durchgeführt. Die erreichten Frequenzen waren 2 bzw 1 fps. Dies zeigt die Schwierigkeiten, die sich bei sehr großen Systemen ergeben, und bestätigt die Notwendigkeit der Vorselektion der interessierenden Teile des zu untersuchenden Systems. Momentan ist die gängige Hardware diesem enormen Rechenaufwand noch nicht gewachsen.

Ausblick

Neben den beschriebenen implementierten Funktionen muß die Animationsfähigkeit auf Darstellungmodi ausgedehnt werden, die Splines benutzen. Dazu gehören die tube-Darstellung und die noch zu implementierende ribbon-Darstellung. Es sollten verschiedene Animationsgeschwindigkeiten ermöglicht werden, damit ein Einzelbild länger betrachtet werden kann (Zeitlupenfunktion). Zusätzlich soll die Auswahl eines bestimmten Zeitfensters es dem Benutzer erlauben, diskrete Abschnitte der Trajektorie näher zu untersuchen.

Manche Trajektorien liefern neben der Koordinateninformationen auch solche über die Entwicklung der Geschwindigkeiten der Teilchen oder des elektrostatischen Potentials. Der Visualisierung dieser Daten muß in Zukunft ebenfalls Sorge getragen werden.

Zur Zeit existiert keine Rückopplung von svt_pdb auf das Bedienungselement. Das bedeutet, daß mit dem Fortschreiten der Trajektorie sich weder der Schieber der Bildlaufleiste mitbewegt, noch die Konfigurationsnummer in der *"go to frame"* Spinbox ändert. Diese Funktionen sind sehr wichtig, da sie dem Betrachter zum einen eine schnelle Abschätzung der momentanen Position in der Trajektorie anhand der Bildlaufleiste und zum anderen eine genaue Bestimmung der aktuellen Konfigurationsnummer anhand der Spinbox erlauben.

Die erweiterte Bibliothek muß im nächsten Schritt in SenSitus integriert werden. SenSitus ist ein am ZAM von Stefan Birmanns entwickeltes Programm zur Visualisierung von statischen molekularen Systemen, das auf SVT basiert.

Literatur

1. A. R. Leach, Molecular Modelling, Prentice Hall, 2001
2. SVT, http://www.fz-juelich.de/vislab/virtual/svt/
3. Qt Designer Manual, http://doc.trolltech.com/3.0/designer-manual.html

Eine Testumgebung für das parallele I/O-Paket GIO

Christoph Spanke

FB 17, AG Monien, Universität Paderborn

E-mail: spanke@uni-paderborn.de

Einleitung

Bei Parallelrechnern gibt es zwei grundsätzlich verschiedene Konzepte zum Austausch von Daten zwischen den verschiedenen Prozessen. Eines davon ist das „shared memory“-Konzept [1], das allen Prozessen in einem Rechnerverbund Zugriff auf einen gemeinsamen Speicher erlaubt und damit für den Programmierer relativ leicht zu handhaben ist. Allerdings ist ein Programm dieser Art schwer portierbar, da eine „shared memory“-Funktionalität als solche nicht in einem durchschnittlichen Computernetz angeboten wird; die verschiedenen Prozesse haben dort nur die Möglichkeit zu kommunizieren, aber sie haben keinen gemeinsamen, für alle adressierbaren Speicher.
Die andere Möglichkeit zum Datenaustausch bei Parallelrechnern ist das „message passing“-Modell, das eine große Verbreitung erfahren hat und aufgrund solcher Standards wie MPI [2] relativ leicht portierbar ist. Zusätzlich erlaubt dieses Modell eine genaue Kontrolle über die stattfindende Kommunikation, was folglich viel Optimierungspotential birgt, ein Vorteil, den ein gemeinsamer Speicher so nicht bietet.
Angenehm für den Programmierer wäre es nun, ein Konzept nutzen zu können, das die Vorteile beider Modelle weitgehend miteinander vereinigt, also intern Nachrichtenkommunikation verwendet und damit auf praktisch jedem Rechnerverbund lauffähig ist, aber für den Programmierer die Benutzung von verteiltem Speicher simuliert. Genau das hat die Entwicklung der GIO-Bibliothek (Global Input/Output) motiviert. GIO [3] realisiert auf der Dateiebene, was GA (Global Arrays) [4] schon auf der Variablenebene anbietet; es ermöglicht also die Nutzung verteilter globaler Dateien. Zur Kommunikation wird die Bibliothek ARMCI (Advanced Remote Memory Copy Interface) [5] verwendet, mit einigen Veränderungen zur Anpassung an GIO.

Funktionsweise von GIO

Dateien

Die mit GIO erzeugbaren globalen Dateien enthalten beliebig viele, vom Benutzer zu definierende Records. Die Records enthalten die eigentlichen Daten und können als globale Arrays betrachtet werden; sie haben eine fest definierte Länge und können Elemente eines bei der Erzeugung festzulegenden Datentyps aufnehmen.

Records

Ein Record wird nun unter den an der Programmausführung beteiligten Prozessen aufgeteilt, wobei es für den Benutzer zwei Funktionen gibt, die er nutzen kann, um die Aufteilung des für den Record benötigten Speichers auf die Prozesse zu bestimmen; entweder er benutzt die Funktion „gio_create_record“, welche den Speicher gleichmässig über alle Prozesse verteilt, oder er spezifiziert mit „gio_create_record_irreg“

genau, welcher Prozess einen wie großen Teil des Records erhalten soll. Diese beiden, und einige wesentliche andere Funktionen, sollen im folgenden kurz beschrieben werden:

Funktionen

error=GIO_OPEN(„/work“, file_handle_1)

Legt globale Datei im „work“-Verzeichnis an, die auf die nach diesem Funktionsaufruf dann mit der in „file_handle_1“ gespeicherten ID zugegriffen werden kann.

error=GIO_CREATE_RECORD(file_handle_1, record_handle_1, n, MT_INT)

Legt einen Record für n Integers in der zu „file_handle_1“ gehörigen Datei an; dieser bekommt dann die nach dem Funktionsaufruf in „record_handle_1“ stehende ID. Der Record wird gleichmässig auf alle an der Berechnung teilnehmenden Prozesse aufgeteilt.

error=GIO_CREATE_RECORD_IRREG(file_handle_1, record_handle_1, distribution_array, n, MT_INT)

Legt einen Record für n Prozesse an. Dabei gibt der Programmierer in distribution_array[i], i=1,...,n, jeweils an, wie viele Elemente des im letzten Parameter spezifizierten Datentyps Prozess i speichern soll.

error=GIO_WRITE(file_handle_1, record_handle_1, offset, length, variable_1, stride, MT_INT)

Schreibt ab der Startadresse von „variable_1“ den als Integer interpretierten Inhalt an die Stelle „offset“ des spezifizierten Records. Dabei kann man noch eine Länge angeben, wenn man z.B. mehrere Elemente eines Arrays schreiben will, und einen „stride“, der die Schrittweite zum jeweils nächsten zu schreibenden Element nach jedem geschriebenen Element angibt.

error=GIO_READ(file_handle_1, record_handle_1, offset, length, variable_1, stride, MT_INT)

Liest den Inhalt des Records, als Integer interpretiert, von der Stelle „offset“ an in die Variable „variable_1“ aus. Auch dabei lassen sich wieder Länge und Schrittweite angeben.

Zielsetzung der Testumgebung

Mit der Testumgebung zu GIO sollen zwei Ziele erreicht werden: es soll möglich werden, Erweiterungen zu der Bibliothek standardisiert zu testen, und außerdem sollen Anwendungsprogrammierer ihre Programme mit Hilfe der Testumgebung auf Effizienz überprüfen können.

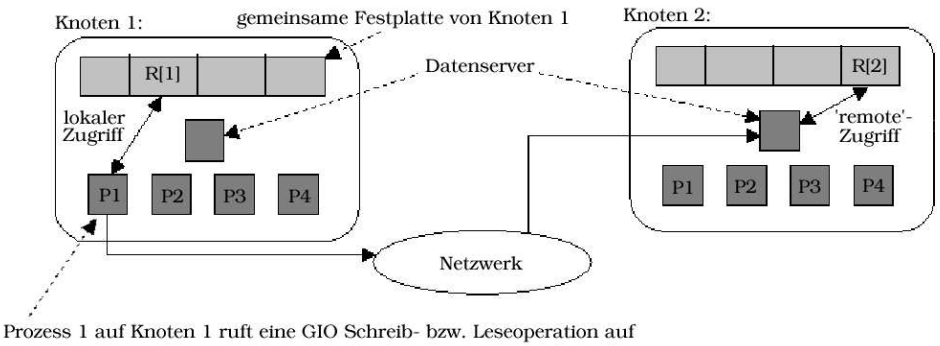
GIO-Erweiterungen

Um Erweiterungen vergleichen zu können, muss man sich auf einen festen Satz von Benchmarks festlegen, die dann mit der Erweiterung getestet werden. Vergleicht man nun die Testergebnisse von GIO mit und ohne Erweiterung, kann man leicht feststellen, ob und bei welchen Zugriffsmustern die Erweiterung Verbesserungen erreicht hat. In dem Zusammenhang ist es wichtig, die Benchmarks sinnvoll festzulegen. Die in dieser Arbeit verwendeten Testprogramme sind durchaus gut geeignet, aber die Testumgebung ist leicht anzupassen und Änderungen sind so bewusst offen gehalten.

GIO-Anwendungen

Ein Anwendungsprogrammierer kann die Testumgebung verwenden, um den Datendurchsatz seines Programms oder bestimmter Programmteile zu ermitteln, aber sein Programm muss dafür gewisse Konventionen erfüllen. Die Zeitmessungen müssen vom Programmierer in das Programm eingefügt werden und sein Programm muss eine Ausgabedatei erstellen, die dem von GNUplot geforderten Format entspricht (siehe dazu die Anleitung der Testumgebung).
Beides ist anders auch nicht zu lösen, da schließlich nur der Programmierer selbst weiß, welche Zeiträume er eigentlich messen will. Die Tests können nun folgendermaßen verwendet werden, um die zeitkritischen Bereiche in einem Programm zu bestimmen:

Lese- bzw. Schreiboperationen haben unter GIO immer die gleiche Form. Als Beispiel betrachte man folgende von Knoten 1, Prozess 1 ausgehende Operation; R bezeichnet dabei einen Record, R[i] den Teil des Records, der bei Prozess i liegt:



Hier sind zwei Knoten mit jeweils vier Prozessen dargestellt. Innerhalb eines Knotens kommunizieren die Prozesse nicht über Nachrichten, der Datenaustausch wird auf dieser Ebene über den verteilten Speicher auf den Knoten abgewickelt. Für die Kommunikation mit Prozessen außerhalb des jeweiligen Knotens ist ein Datenserver-Prozess verantwortlich. Von diesen gibt es genau einen pro Knoten, über den sämtliche Kommunikation abwickelt wird. Nun wird bei einer Lese-/Schreiboperation, die auf einen Record zugreift, ein Teil der Operation im lokalen Speicher abgewickelt (auf dem Teil des Records, den der aufrufende Prozess selbst hält, durch das „memcpy“ dargestellt), und der Rest muss über das Netzwerk mit Nachrichtenkommunikation abgewickelt werden; auf dem Empfängerknoten wird dann auf den lokalen Speicher zugegriffen und die geforderte Operation erledigt.
Wie man sieht, gibt es nun im Verlauf einer solchen Operation mehrere Möglichkeiten für den „Flaschenhals“: entweder er besteht in der Netzwerkkommunikation (wohl der Normalfall, da in der Regel Netzwerkkommunikation langsamer ist als eine Speicheroperation), den Festplattenzugriffen (falls viele Prozesse gleichzeitig auf den selben Platten arbeiten, kann das ein Programm stärker bremsen als die Netzwerkkommunikation), ungeschickter Programmierung (ungünstigen in dem Programm verwendeten Datenstrukturen) oder in Software-Randbedingungen (z.B. maximale Nachrichtengröße unter ARMCI; wird diese erreicht, werden große Nachrichten in mehrere kleine aufgespaltet, was für zusätzlichen Kommunikationsaufwand sorgt). Vergleicht der Anwendungsprogrammierer nun die Ergebnisse seines Programms mit denen der Testmuster, ist es evtl. möglich, die geschwindigkeitshemmenden Elemente in seinem Programm zu finden und es an diesen Stellen zu verbessern.

Funktionsweise der Tests

GIO ist eigentlich eine C-Bibliothek, lässt sich aber mit einem bereits implementierten Interface auch in Fortran-Programmen nutzen. Um dessen Funktionsweise ebenfalls garantieren zu können, wurden alle Testprogramme für GIO in Fortran programmiert. Die Tests sind in drei Gruppen aufgeteilt: Funktionalitätstests, Stabilitätstests und Performance-/Benchmark-Tests.

Testrechner

Die Tests wurden im Zentralinstitut für Angewandte Mathematik im Forschungszentrum Jülich auf dem „ZAMpano“-Parallelrechner [6] durchgeführt. Dieser Parallelrechner bietet acht Knoten mit jeweils vier 550 MHz-Prozessoren an. Jeder Prozessor verfügt jeweils über 512 KB Cache und 2 GB Arbeitsspeicher. Für GIO sind insbesondere die Festplattendaten von Bedeutung; jeder Knoten verfügt über eine 9 GB-Platte; der Ultra-160 SCSI Bus hat eine theoretische Bandbreite von 2 x 80 MB/s (2 Kanäle parallel), pro Disk kann man ca. 40 MB/s erreichen. Das hängt in der Praxis aber natürlich sehr von der Art und Größe der Zugriffe ab. Die Knoten sind untereinander durch ein Myrinet- und ein TCP/IP-Netzwerk verbunden, die Tests wurden mit dem Myrinet durchgeführt.

Funktionalitätstests

Die Funktionalitätstests prüfen die korrekte Funktionsweise der GIO-Funktionen. Sie sind sehr kurz und übersichtlich gehalten und damit als erste Anwendungsbeispiele und Tutorial für einen Benutzer interessant.

Stabilitätstests

Die Stabilitätstest rufen die wesentlichen GIO-Funktionen mit sehr großen Datenmengen auf. Sie sollten die Obergrenze für die maximal erlaubte Zahl von erzeugbaren Dateien/Records aufzeigen; wünschenswert wäre an dieser Stelle, dass lediglich der auf dem Parallelrechner zur Verfügung stehende Festplattenspeicher die Verwendung von GIO einschränkt.

Benchmarks

Schließlich wurden Performance- bzw. Benchmarktests vorgenommen, welche den meisten Zeitraum des Projektes in Anspruch genommen haben. Diese sind eben darauf ausgelegt, Zeitmessungen für die verschiedenen GIO-Operationen vorzunehmen und auch unterschiedliche Lese-/Schreibfolgen für ein bestimmtes Problem in Bezug auf Effizienz zu bewerten, z.B. die Frage: Ist es mit GIO sinnvoller, eine Matrix elementweise oder als Block auszulesen, und unter welchen Bedingungen? Es ist klar, dass im Regelfall die Zugriffe auf große Speicherblöcke die bessere Wahl sind, und mit der Testumgebung konnte dies auch gezeigt werden. Einige besonders wichtige Lese-/Schreibfolgen werden in dem Testprogramm hintereinander ausgeführt und selbstständig ausgewertet, so dass sich ein Programmierer, der an der GIO-Bibliothek selbst arbeitet, sofort über die Auswirkungen seiner Änderungen auf die in den Testprogrammen implementierten Zugriffsmuster Klarheit verschaffen kann.

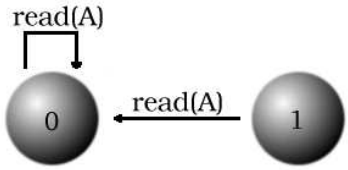
Testergebnisse

Zuerst wurde die GIO-Bibliothek einem ausführlichen Funktionalitäts- und Stabilitätstest unterzogen, wobei Records bis zur Gesamtgrösse von ca. 50 GB auf dem „ZAMpano“-Parallelrechner angelegt wurden. Unter diesen Bedingungen läuft das Programm stabil, mehr oder größere Records konnten aufgrund

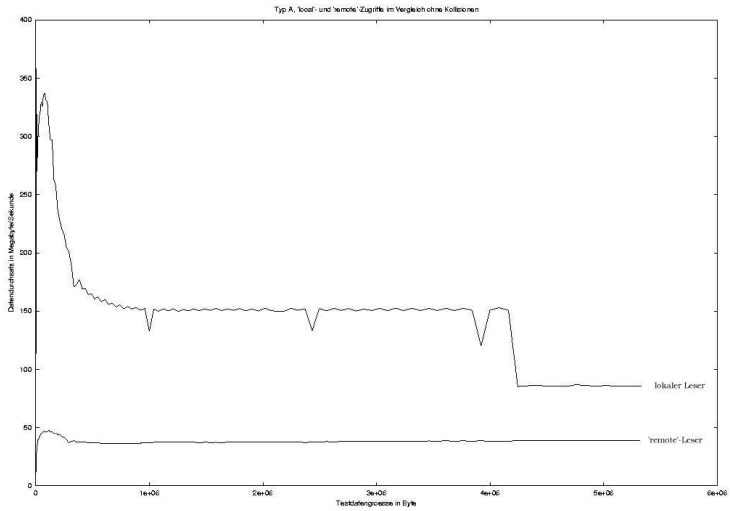
der Speicherplatzbeschränkungen der Festplatten auf dem Parallelrechner nicht erfolgreich sein. Das Dokument, welches Sie gerade lesen, ist eine Beispielanwendung für die Testumgebung; die nun folgenden Zugriffsmuster wurden mit Hilfe der Testumgebung ausgewertet und in diesem Dokument zusammengefasst. Als Benchmarks wurden nun folgende Zugriffsmuster implementiert:

Typ A

Zuerst ist natürlich interessant, wie sehr die Zugriffsgeschwindigkeit davon abhängt, ob „remote“, also über das Netzwerk, oder lokal auf einen Record zugegriffen wird. Dabei sollten erst einmal Interferenzen mit anderen Prozessen ausgeschlossen werden, um zu sehen, welche Geschwindigkeiten überhaupt erreichbar sind. Deshalb arbeitet das erste Benchmark-Programm nur mit zwei Prozessen (das Programm kann zwar mit beliebig vielen Prozessen gestartet werden, aber nur zwei verrichten wirklich Arbeit); einer liest lokal, der andere „remote“. Das sich diese beiden Prozesse nicht behindern, ist durch eine Synchronisation sichergestellt. A bezeichnet den zu lesenden Record:



Das nun folgende Schaubild wurde von der Testumgebung automatisch mit Hilfe von GNUplot erstellt, allerdings zum besseren Verständnis noch mit dem Grafikprogramm „GIMP“ nachbearbeitet und um einige Erläuterrungen erweitert. Das gilt auch für die anderen hier vorgestellten Schaubilder, die Form der Beschriftung ist bei Resultaten aus der Testumgebung etwas anders. Die Ergebnisse zu Testschema A sehen folgendermaßen aus, wobei die obere Kurve den Datendurchsatz des lokalen Lesers und die untere Kurve den des „remote“-Lesers repräsentiert:

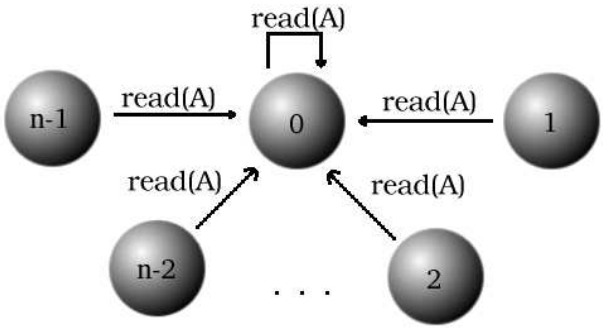


Die Kurven steigen erst einmal an, da kleine Datenmengen immer vom Datendurchsatz her gesehen teuer sind; bei diesen fällt der Funktionsaufruf selbst und die damit einhergehende Kommunikation stark ins Gewicht. Der starke Abfall der Kurven danach ist auf Cache-Effekte zurückzuführen, da der Cache bei wachsenden Datenmengen natürlich immer geringere Bedeutung hat. Dann pendelt sich der Verlauf der

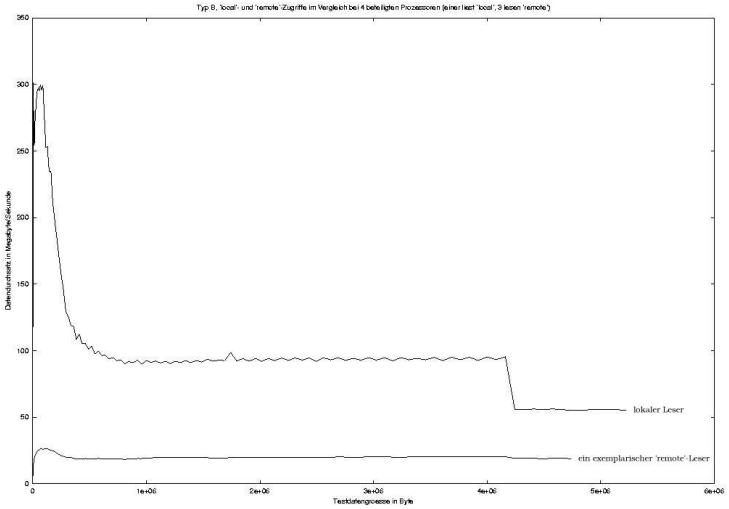
Kurve ein, nur bei dem lokalen Leser gibt es noch einen Abfall bei ca. 4 MB, wohl auf den Festplattencache zurückzuführen, der auf den ZAMpano-Maschinen eine Größe von 4 MB hat. Tests wurden bis weit über diese Datengröße hinaus gemacht, aber der Datendurchsatz hat sich im wesentlichen nicht mehr verändert.

Typ B

Nun können in einem Anwendungsbeispiel natürlich Interferenzen auftreten, d.h. mehrere Prozesse versuchen, auf den gleichen Speicherbereich zuzugreifen. Der schlechteste Fall wäre dabei, wenn alle Prozesse zum selben Zeitpunkt auf den gleichen Recordteil zugreifen müssten. Dieser Fall wird im folgenden untersucht:



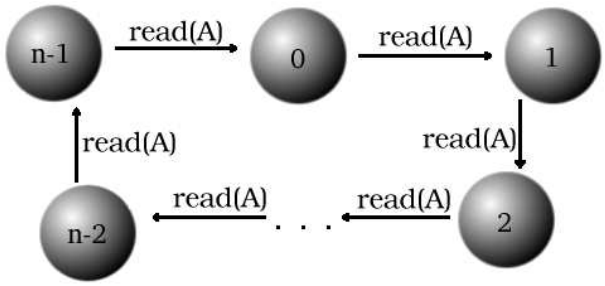
Die obere Kurve zeigt wieder den einzigen lokalen Leser, die andere Kurve einen exemplarisch ausgewählten „remote“-Leser. Die Kurvenverlauf entspricht dem von Typ A, man sieht allerdings, dass der Datendurchsatz insgesamt geringer geworden ist im Vergleich zum ersten Test:



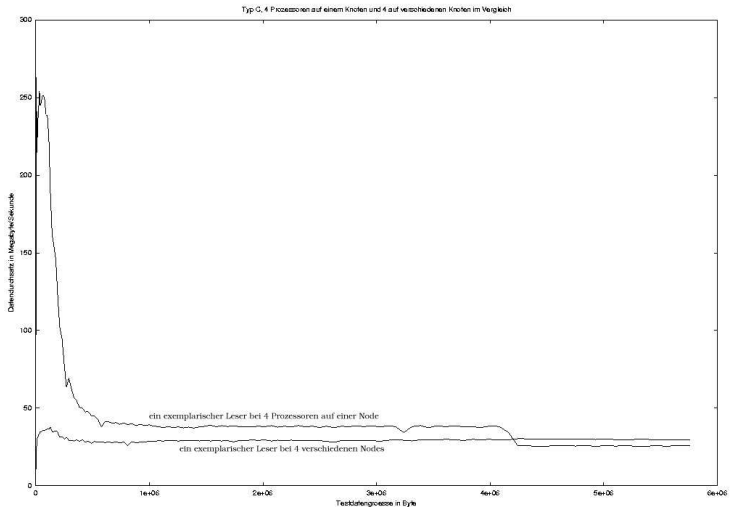
Sowohl bei der lokalen als auch bei den Netzwerk-Leseoperationen hat sich der Datendurchsatz in etwa halbiert. Das ist insofern ein erfreuliches Ergebnis, als dass der Datendurchsatz offensichtlich besser als linear in der Anzahl der Prozesse skaliert.

Typ C

Interferenzen treten allerdings auch schon dann auf, wenn Prozesse auf verschiedene Speicherbereiche zugreifen. Im nächsten Test greifen alle Knoten über das Netzwerk auf ihren Nachbarknoten zu, ein ringartiges Kommunikationsmuster. Also Prozess 0 liest Daten von Prozess 1, Prozess 1 liest von Prozess 2,..., Prozess n-1 liest von Prozess 0.
Eigentlich sollten sich die Prozesse dabei nur gegenseitig behindern, falls sie auf dem selben Knoten liegen (dann bildet die Festplatte einen Flaschenhals, da alle von der selben Platte lesen). Falls sie nicht auf dem selben Knoten liegen, sollte in etwa der gleiche Datendurchsatz erreicht werden wie bei Kommunikationstyp A, denn es gibt ja dann keine gleichzeitigen Zugriffe auf eine Festplatte und damit auch keine Interferenzen:

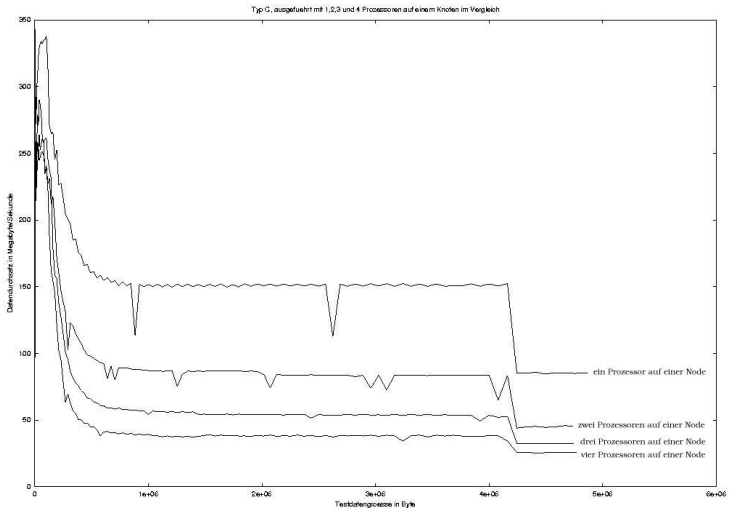


Bisher liefen lokale Zugriffe grundsätzlich schneller ab als „remote“-Zugriffe; an dieser Auswertung lässt sich sehen, dass dies nicht immer der Fall ist. Die obere Kurve zeigt den Datendurchsatz bei vier Prozessen auf einem Knoten, die untere Kurve den von 4 Prozessen auf verschiedenen Knoten:



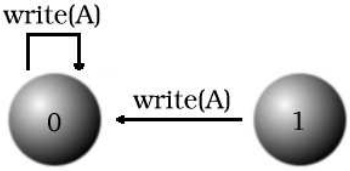
Man sieht, dass nur bei den kleinern Datenmengen wie erwartet die Prozesse auf dem gleichen Knoten schneller sind (sie können ja den Cache nutzen), aber bei größeren Datenmengen erkennt man kaum noch einen Unterschied; sobald der Festplattencache nicht mehr genutzt werden kann, sind die Zugriffe über das Netzwerk sogar schneller als die über den gemeinsamen Speicher. Das ist schon bemerkenswert. Dieser Überholeffekt tritt zuerst bei vier Prozessen im Vergleich auf, mit höheren Prozesszahlen lässt sich das Experiment leider nicht durchführen, da die Obergrenze des „ZAMpano“ bei vier Prozessen pro Knoten liegt (Dass nur vier Prozesse auf einem Knoten platziert wurden, wird natürlich wiederum an genau diesem Effekt liegen; warum mehr Prozessoren auf einen Knoten platzieren, wenn dadurch Zugriffe

langsamer werden?).
Da bei Netzwerkkommunikation selbst keine Interferenzen auftreten, ist der Datendurchsatz unter dieser Bedingung unabhängig von der Anzahl der verwendeten Prozesse und pendelt sich bei ca. 30 MB/Sekunde ein. Bei Ringkommunikation auf einem Knoten wird die Skalierung des Datendurchsatzes an folgendem Schaubild deutlich:

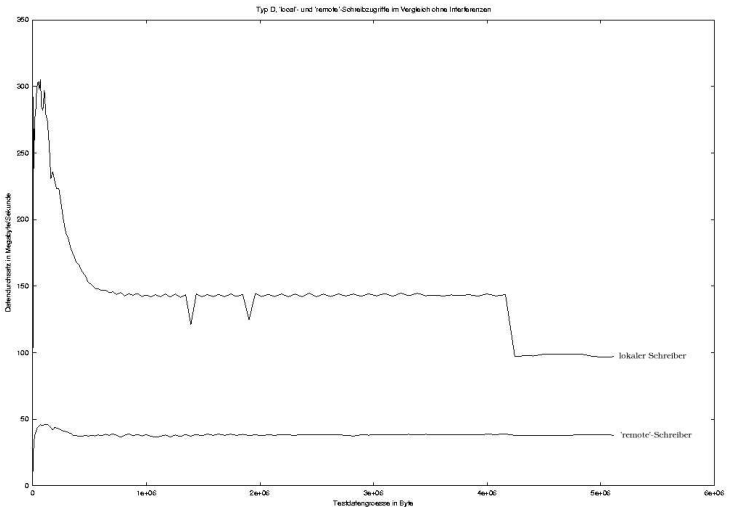


Typ D

Jetzt folgen einige Schreibtests. Zuerst soll wieder, wie bei den Lesezugriffen auch, eine Messung unter Ausschluss von Kollisionen stattfinden: Prozess 0 verwaltet einen Record, in den er selber schreibt, dann wird synchronisiert, und Prozess 1 schreibt über das Netzwerk in den Record. Also wie Typ A, nur Schreibe- anstelle von Leseoperationen:

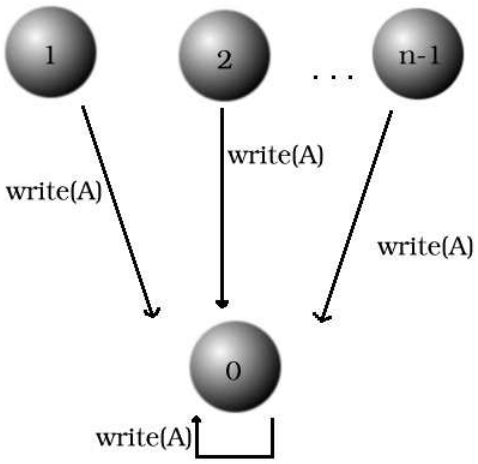


Der Kurvenverlauf ist in folgendem Bild dargestellt und unterscheidet sich, wie erwartet, nicht von dem zu Typ A gesehenen:

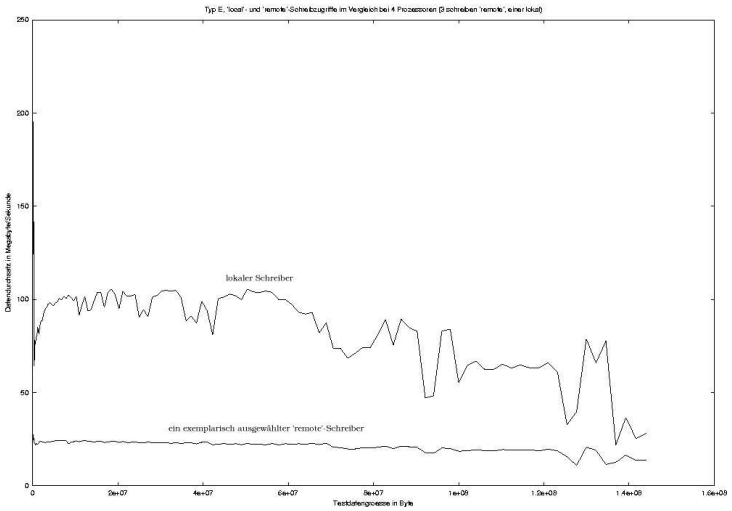


Typ E

Ein weiterer Schreibtest lässt nun vier Prozesse gleichzeitig schreiben, einer lokal, die anderen „remote“. Also Typ B mit Schreibe- anstelle von Leseoperationen:

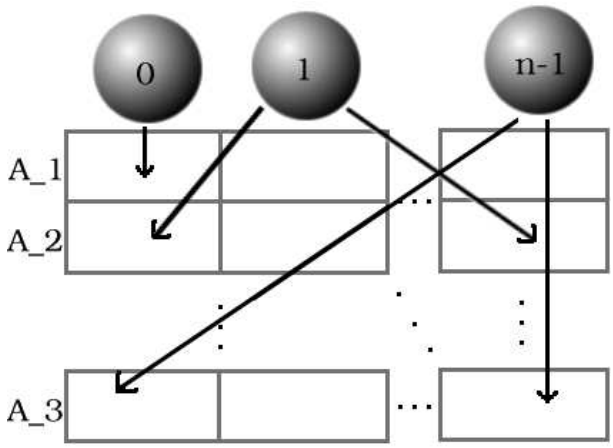


Die Auswirkungen bei Schreibzugriffen sind etwas anders als bei Lesezugriffen, was an der unterschiedlichen Verarbeitung der Zugriffsarten liegt; ist ein Prozess einmal mit seiner Schreiboperation an der Reihe, wird er nicht mehr unterbrochen, anders als bei einer Leseoperation, so dass die enormen Schwankungen entstehen; kommt der Prozess schnell an die Reihe, erreicht er einen sehr hohen Datendurchsatz; aber das Argument gilt natürlich auch in die Gegenrichtung. Die Auswirkungen dieses „Schedulings“ sind sehr stark, die Ausschläge sind nur deshalb nicht noch größer, weil die Experimente recht häufig wiederholt werden. Trotz der Schwankungen ist die Tendenz der Kurven zu erkennen; sie ähneln dem Kurvenverlauf bei der Auswertung von Zugriffstyp B:

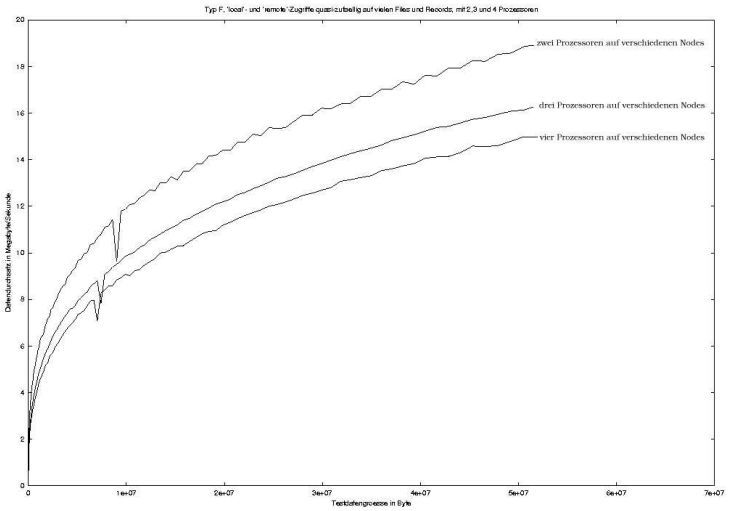


Typ F

Im letzten Test wurden dann sehr viele Records erzeugt, fair verteilt auf die teilnehmenden Prozesse. Auf die Records wurde dann quasi-zufällig zugegriffen, schreibend und lesend. Dieses Szenario simuliert eine recht realistische Anwendung, in der es unmöglich sein kann, das Programm durch eine günstigere Verteilung der Records auf die Prozesse effizienter zu machen. Hier bezeichnen die A_i's nicht Records, sondern globale Dateien, und die Zellen sollen jeweils einen Record darstellen, der vollständig von dem Prozess gehalten wird, in dessen Spalte er hier dargestellt wird:

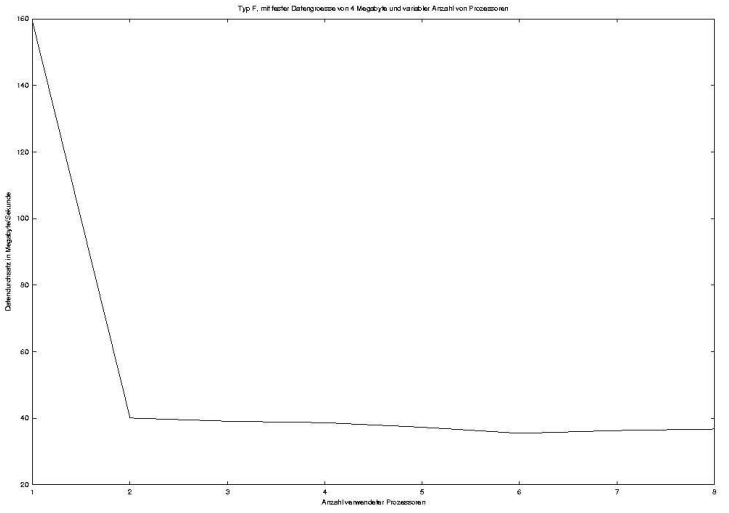


Die Ergebnisse sind interessant; oben die Kurve für zwei Prozesse, darunter die für drei und vier Prozesse:



Man erkennt, dass der Datendurchsatz nicht zu stark bei einer höheren Anzahl verwendeter Prozesse abfällt. Das wäre natürlich auch ein fatales Ergebnis. Dass es nicht eintritt, liegt an dem geringen Anteil der lokalen Zugriffe, die für eine starke Skalierung, aus hinreichend erläuterten Gründen, verantwortlich sind. Und da die Zugriffe quasi-zufällig sind, ist die Wahrscheinlichkeit für einen lokalen Zugriff 1 / #Prozesse; folglich verringert sich der Datendurchsatz bei einer noch größeren Anzahl von verwendeten Prozessen immer langsamer.

Da bei den anderen Tests die Unterscheidung lokal gegenüber „remote“ im Vordergrund stand, ist es nicht sinnvoll gewesen, Schaubilder zu zeigen, die den Datendurchsatz abhängig von der Anzahl der verwendeten Prozesse zeigen; die maximale Anzahl der Prozesse pro Knoten liegt ja nur bei vier. In dem letzten Testbeispiel allerdings sieht das anders aus, also macht hier eine Auswertung in der Anzahl der verwendeten Prozesse durchaus Sinn (die verwendete Gesamtdatengröße beträgt 4.000.000 Byte = 4 MByte). Acht Knoten erlauben immerhin acht Messungen:



Der Abfall am Anfang ist natürlich nur logisch, da bei einem Prozess noch alle Operationen lokal abgewickelt werden. Man kann erkennen, dass der Datendurchsatz danach nur noch sehr langsam abfällt und sich am Ende einpendelt, ein durchaus erfreuliches Ergebnis.

Zusammenfassung:

GIO (Global Input/Output) ist eine Erweiterung der GlobalArray-Bibliothek. GA gestattet dem Benutzer, globale Felder in seinen Programmen zu benutzen. GIO erweitert GA in der Hinsicht auf Dateien und macht es möglich, auch Dateien global anzulegen und sie auf einem Parallelrechner so zu benutzen, als wären es lokale Dateien auf einem einzelnen Rechner. Dabei wurden die Kommunikationsfunktionen der Bibliothek ARMCI (Advanced Remote Memory Copy Interface) verwendet bzw. im Bedarfsfall den Anforderungen von GIO angepasst. Die im Rahmen des Projektes entwickelte Testumgebung mit den beschriebenen Tests hat die Funktionalität und Effizienz von GIO gezeigt und wird sowohl die Programmierung von Anwendungen als auch die von Erweiterungen zu dieser Bibliothek vereinfachen und beschleunigen.

Literatur

1. R. BARRIUSO, A. KNIES, *SHMEM Users Guide,* CRAY Research, SN-2516,1994
2. W. GROPP, E. LUSK, A. SKJELLUM, *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* MIT Press, 1999
3. K. SCHOLTYSSIK, *GIO: Ein System für parallele Ein-/Ausgabe auf Rechnerverbundsystemen,* Paper für die PASA 2002
4. J. NIEPLOCHA, J. JU, M. K. KRISHNAN, B. PALMER, V. TIPPARAJU, *The Global Arrays User's Manual,* URL: http://www.emsl.pnl.gov:2080/docs/global/user.html
5. J. NIEPLOCHA B. CARPENTER, *ARMCI: A Portable Remote Memory Copy Library for distributed Array libraries and Compiler Run-time Systems,* In Proc. RTSSP IPPS/SPDP 1999, San Juan
6. *ZAMpano - ZAM Parallel Nodes,* URL: http://zampano.zam.kfa-juelich.de/

Optimization of neighbour list techniques and analysis of effects of round-off errors in molecular dynamics calculations

V. V. Stegailov

Moscow Institute of Physics and Technology, Institutskij per. 9, Dolgoprudnij, Moscow Region, Russia

vlad_st@yahoo.com

Abstract: The first part of this report is devoted to the methods of acceleration of the force evaluation routine in molecular dynamics (MD) calculations. Different approaches are considered: simple brute force algorithm, Verlet list and linked list methods, combination of Verlet and linked lists. Several recipes are given in order to reach the best perfomance. Consequences of round-off errors, integration scheme precision and Lyapunov instability on results of MD calculation are discussed in the second part of the report.

Introduction

Molecular-dynamics simulation is widely used to explore different phenomena in material science, chemistry, molecular biology and other fields where it is important to know how a physical system evolves at the microscopic level of individual atoms. Nowadays modern computers allow to perform such simulations on a large scale [1] that was hard to imagine even ten years ago, but in spite of this there are still a lot of possible tasks which would still take too much time to be solved. That's why it is very desirable to optimize computational algorithms to make them faster and if possible not to lose the accuracy either.

General Scheme of an MD Calculation

Let us consider the main features of a MD model that will be necessary for the following exposition. A physical system is represented as a set of point particles – atoms, and evolves according to the classical equations of motion

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = \vec{F}_i(\vec{r}_1, \ldots, \vec{r}_N), \quad i = 1, .., N, \tag{1}$$

where N is the total number of particles in the model, m_i and $\vec{r}_i = (x_i, y_i, z_i)$ are mass and position of the i-th particle, $\vec{F}_i$ is the force acting on the i-th particle that is usually determined as a derivative of the total potential energy:

$$\vec{F}_i(\vec{r}_1, \ldots, \vec{r}_N) = -\frac{\partial U(\vec{r}_1, \ldots, \vec{r}_N)}{\partial \vec{r}_i}. \tag{2}$$

In practice the potential energy is represented as a sum like

$$U(\vec{r}_1, \ldots, \vec{r}_N) = \sum_{i<j}^{N} u(\vec{r}_i, \vec{r}_j) + \sum_{i<j<k}^{N} u(\vec{r}_i, \vec{r}_j, \vec{r}_k) + \ldots . \tag{3}$$

Simple models with a *pair potential* (when only the first sum in eq.(3) is left) can describe various (mainly bulk) properties of real systems (e.g. Lennard-Jones potential for liquid Ar). However to improve precision and to be able to treat more complex cases (e.g. interfaces etc.) it is necessary to use multibody potentials (e.g. embedded atom potential for Si).

It is also possible to classify pair and multibody potentials as *short ranged* and *long ranged* potentials (Fig. 1, left). In the case of short ranged potentials it is possible to introduce a distance R_{cut} – so called *cut-off radius* – and to neglect all the contributions to the force acting on the given particle from other particles that are more distant than R_{cut}. For the calculation of long ranged potentials (that vanish with distance slower than r^{-d}, where d is dimensionality of the system) several special approaches based on Ewald summation have been developed. In general, it is always possible to split a given potential into short ranged and long ranged parts. The further exposition will be essentially based on the assumption of a short range potential in the system.

One should also mention the role of boundary conditions. Only if we were interested in simulation of small clusters or drops we could be satisfied with 10^5-10^6 particles in the MD model, which are realistic values used in production runs nowadays. To simulate properties of bulk media or interfaces one should apply periodic boundary conditions in 1, 2 or 3 dimensions.

Brute Force Algorithm

The straight-forward approach to calculate forces is sometimes called *brute force* algorithm and consists in the consideration of contributions of *all* particles in the system to the force acting on a given particle. In the case of a pair potential it is implemented as a double loop over particles and hence consumes time proportional to N^2 (more precisely, taking into account Newton's 3rd law of action-couteraction one can express time per force routine call as $T_{b.f.} = \cdot N(N-1)/2)$ [1] .

In the case of three- and multibody potentials this method consumes time proportional to N^3 and so on. Such a dependence on the system size is unsatisfactory, because it becomes impossible to simulate large systems in this way. At the same moment a lot of time during force evaluations is wasted for consideration of pairs that are certainly separated farther than R_{cut}. In an ideal case one should consider only particles in the cut-off sphere of the given particle that contain $(4\pi/3)\rho R_{cut}^3$ neighbours in the average (ρ is the number density). For most cases it is valid that $R_{cut} < L$, where L is a characteristic length of the system (e.g. length of a simulation box edge) and the overhead of the brute force method is of the order L^3/R_{cut}^3.

Verlet List Algorithm

An obvious improvment is the following (see [2]). Since trajectories of the particles are continuous there is always a certain set of particles – neighbours to the given one – that does not change for some time. This time depends on the properties of the system: in a solid or a viscous liquid such a list of neighbours is constant much longer than for gases, but nevertheless it is constant for some time in each case.

One can introduce a skin (buffer) layer of thickness of R_{skin} and store all the indices of neighbour particles (that are closer than $R_{list} = R_{cut} + R_{skin}$ to the given one) in the list at a certain moment t^* (Fig. 1, right). In the next step during the force evaluation it is possible to use this list instead of

[1] Proportionality factors, arising from load/store and fbating point operations are ignored here.

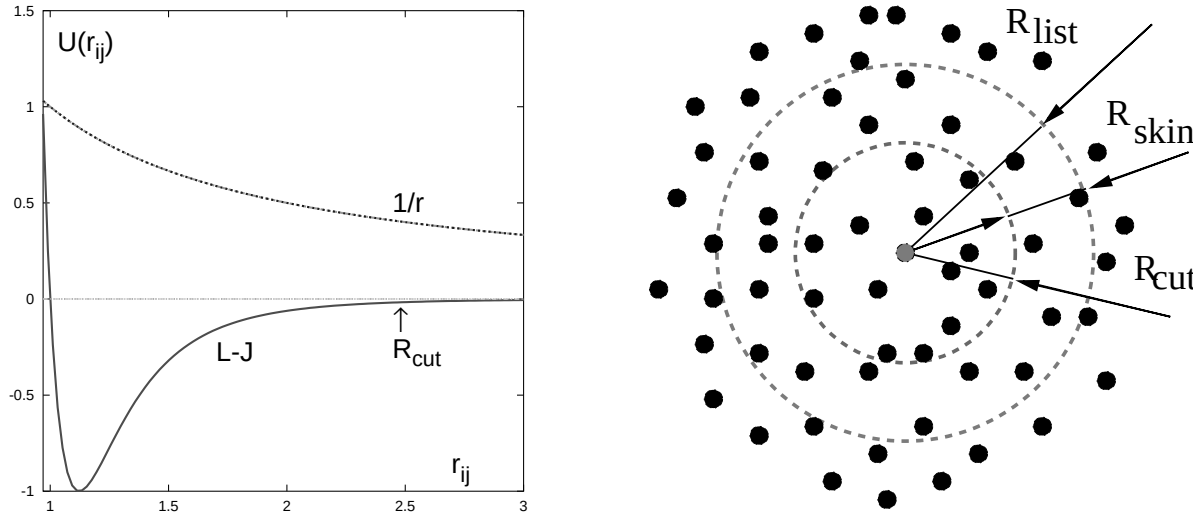


Figure 1: (left) Lennard Jones 12-6 potential $U(r_{ij}) = 4\epsilon((\sigma/r_{ij})^{12} - (\sigma/r_{ij})^6)$ (short ranged, cut-off radius 2.5σ is shown) and Coloumb repulsive potential (long ranged). (right) Illustration of the Verlet neighbour list.

performing a double loop over all particles. The list should be updated just after the moment t when particle displacements with respect to t^* become larger than the skin-radius R_s: for example, one can check the following rather conservative condition:

$$3 \cdot \max_{\substack{1 \le i \le N \\ \alpha \in \{x,y,z\}}} \left(\alpha_i(t + t_*) - \alpha_i(t_*)\right)^2 \ge \left(\frac{R_{skin}}{2}\right)^2 \tag{4}$$

Speedup estimation

In the case of the Verlet neighbour list the total time which is spent for the force routine may be expressed as a sum of two terms[2]:

$$T_{v.l.} = \frac{1}{n+1}\frac{N(N-1)}{2} + \frac{n}{n+1}\frac{N}{2}\left(4\pi\rho \int_0^{R_{cut}+R_{skin}} g(r) r^2 dr\right) \tag{5}$$

where n is an average number of steps between list updates, $g(r)$ is the radial distribution function. The first term, proportional to N^2, corresponds to the time spent for list updates. The second one stands for force evaluations while the neighbour list is known. The expression in big brackets is the averaged number of neighbours for a given particle.

In accordance with eq.(4) one can write a relation between the list update frequency n and the skin radius R_{skin}:

$$R_{skin} = 2 \cdot n \cdot \Delta t \cdot 3\sqrt{\frac{3k_BT}{m}} \tag{6}$$

where Δt is a time step of numerical integration, $v_{th} = \sqrt{\frac{3k_BT}{m}}$ is an average thermal velocity of particles (k_B is Bolzmann constant and T is temperature). The factor of 3 is approximate and accounts for the fact that only "hot" particles from the tail of the Maxwell distribution determine a moment t when

[2]See the remark in the section .

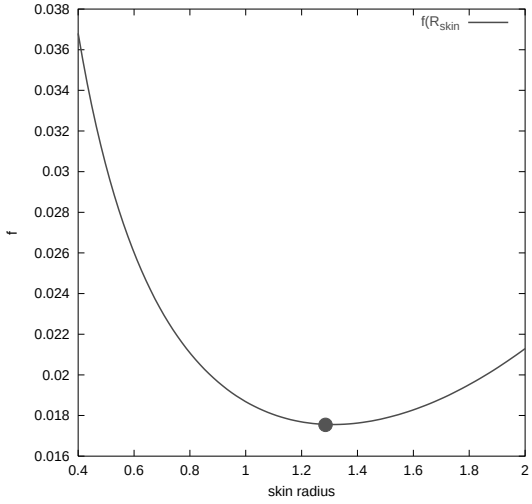


Figure 2: Small values of R_{skin} lead to frequent list updates that consume time as N^2, large value result in big numbers of neighbours N_c that brings us from N closer to the N^2 case.

(4) becomes valid. The factor 2 means that in the extreme situation particles can move towards each other.

One can rearrange eq.(6) as a function of R_{skin}

$$n(R_{skin}) = \frac{R_{skin}}{2 \cdot \Delta t \cdot 3 v_{th}}, \tag{7}$$

and then estimate a speedup $T_{v.l.}/T_{b.f.}$ as a function of R_{skin} and fixed parameters of the simulation:

$$f(R_{skin}; \rho, T, \Delta t, N, R_{cut}) = \frac{T_{v.l.}}{T_{b.f.}} = \frac{1}{n+1} + \frac{n}{n+1}\frac{1}{N-1}\left(4\pi\rho \int_0^{R_{cut}+R_{skin}} g(r) r^2 dr\right). \tag{8}$$

Assuming that the MD model corresponds to either a gaseous or at least to a liquid phase one can approximate the number of neighbours (the expression in big brackets) as

$$4\pi\rho \int_0^{R_{cut}+R_{skin}} g(r) r^2 dr \cong \frac{4\pi}{3}\rho(R_{cut} + R_{skin})^3. \tag{9}$$

In this case f does not depend on an *a priori* unknown radial distribution function $g(r)$. Therefore it is possible to estimate an optimum value of R_{skin} from a the condition $\partial f / \partial R_{skin} = 0$ *before* a simulation run (Fig. 2).

Numerical results

To check the validity of the estimation of R_{skin}^{opt} benchmarks for different cases were performed [3]. Results are presented in the figures 3, 4 and 5. R_{skin}^{opt} can be predicted very accurately by the expression eq.(8) in approximation eq.(9) for various temperatures T, numbers of particles N, integration time steps Δt when density $\rho < 1$, i.e. up to solid phase because in this case eq.(9) becomes not valid and one should take

[3] Model system consisted of N particles interacting via repulsive part of Lennard-Jones 12-6 potential ($R_{cut} = 2^{1/6}\sigma$). Half-step leap-frog integration scheme was used. Periodic boundary conditions in 3 dimensions were applied. Initial configurations corresponded to equilibrium states for the given density ρ and temperature T. Reduced units are used: σ^{-3} for the density, ϵ for the temperature and $(m\sigma^2/\epsilon)^{1/2}$ for the time step. A computer with CPU Pentium III Xeon 550 MHz, 512 KB on-chip cache was used. Routines for time measurement written by Jens Rühmkorf are a part of *epilog* library.

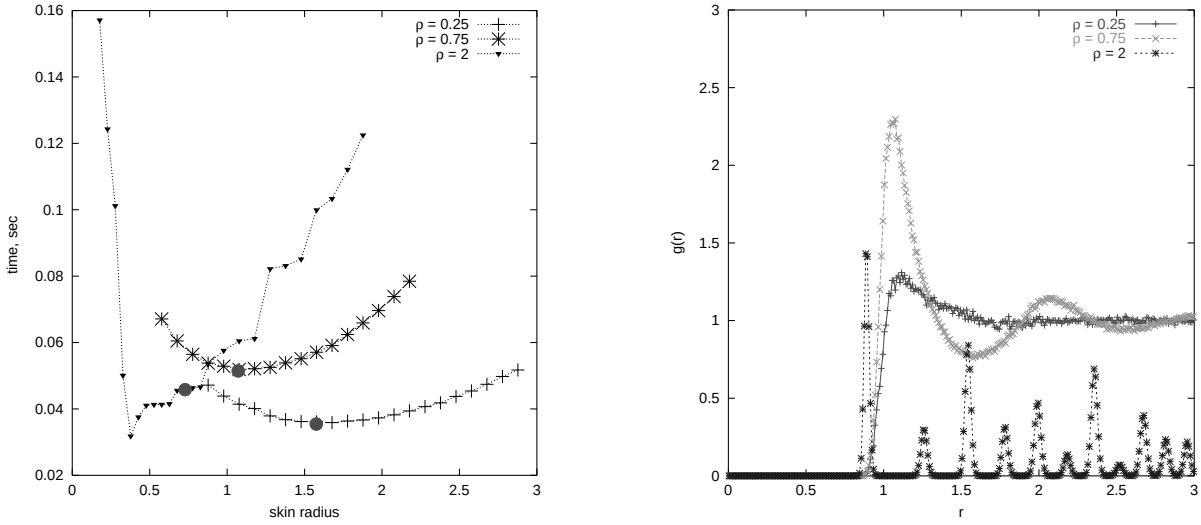


Figure 3: (left) Average time per force routine call as a function of R_{skin} for different values of density ρ (T=2, N=4000, Δt=0.001). Filled red circles correspond to the values of R_{skin} obtained as minimum of f from eq.(8) in approximation eq.(9). (right) Radial distribution functions $g(r)$ for the same cases as presented on the left fi gure.

into account $g(r)$ (see Fig. 3, right) explicitly. Moreover as the density of the system increases particle motion becomes more and more diffusive and eq.(6) based on a free-flight assumption becomes less and less valid as well. Nevertheless the results of calculations showed that the extrapolation of eq.(8) in approximation eq.(9) for dense systems gives reasonable values of R_{skin}^{opt} that can be used as preliminary estimations.

Linked List Algorithm

In spite of the fact that the Verlet list method gives an essential speedup in comparison with the brute force approach it also consumes time proportional to N^2 because one can not eliminate list updates at all and hence it becomes too slow for large systems. A further improvement is possible with so called *linked list* or *linked-cell list* method that allows to reduce the number of operations to $O(N)$ [2] (see Fig. 9 in the Appendix).

Conventional method

The conventional linked list technique consists in the following. One subdivides the volume of the simulation box into such subcells that the side of the subcell $l \geq R_c$. Once in the beginning of the program the map of neighbour subcells for each given subcell is created. The first part of the force routine is the sorting of all particles into the subcells, that can be performed fast enough (number of operation is proportional to N). Then the force acting on a given particle is calculated via consideration of particles in the neighbour subcells. Since their indices are known this part of the algorithm also takes $O(N)$ operations.

However there is a certain overhead in this case as well. During force calculation for each particle we consider about $27l^3/(4\pi/3)R_c^3 \geq 6.5$ times more particles than is necessary.

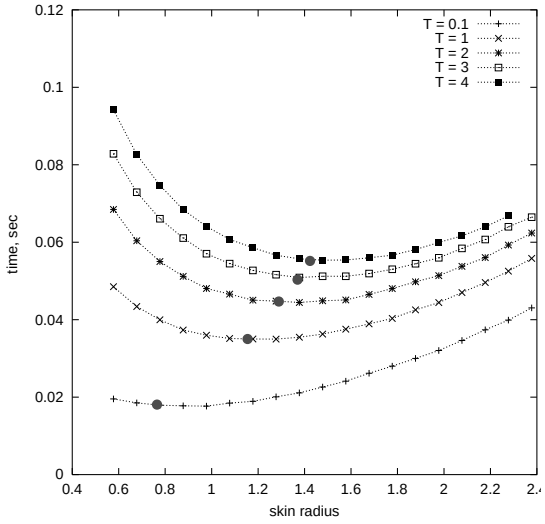


Figure 4: Average time per force routine call as a function of R_{skin} for different values of temperature T (ρ=0.45, N=4000, Δt=0.001). Filled red circles correspond to the values of R_{skin} obtained as minimum of f from eq.(8) in approximation eq.(9).

Modified method

A possible way to reduce this overhead is to use smaller subcells. The idea is that the smaller is a subcell side l the better approximation of a cut-off sphere for a given particle we have (Fig. 6) [3]. In the limit of $l \to 0$ there will be no overhead at all, i.e. during force calculation for each particle only particles from its cut-off sphere will be considered. However this situation in impossible in practice because: i) time to look through neighbour subcells increases as $O(l^{-3})$, ii) there is always a certain memory limit.

A modified linked list algorithm can be implemented in two different ways: i) with *off-set list* (see Fig. 10 and Fig. 12 in the Appendix), ii) with *subcell neighbour list* that is created only once in the beginning of the calculation (see Fig. 13 in the Appendix).

Speedup estimation

Let us estimate the speedup which may be obtained using the modified version of the linked list. The necessary parameters are presented in the following table:

	Fixed parameters:
N	number of particles in the simulation box
R_c	cut-off radius
ρ	particle density
	Variable parameter:
M	number of subcells per simulation box edge
	Derived parameters:
$L = (N/\rho)^{1/3}$	simulation box edge length
$l = L/M$	subcell edge length
$m = R_c/l$	number of subcells per cut-off radius
$N_n = N_n(m)$	number of neighbour subcells for a given subcell

An average execution time $T_{l.l.}$ of one force-routine loop consists of the following terms

1st: $T_i \cdot M^3$ – time to look through each cell searching for i-th particle

2nd: $(\frac{1}{2} \cdot \frac{N}{M^3} \cdot T_f) \cdot N$ – calculation of the force which acts on the given particle from particles in the same subcell

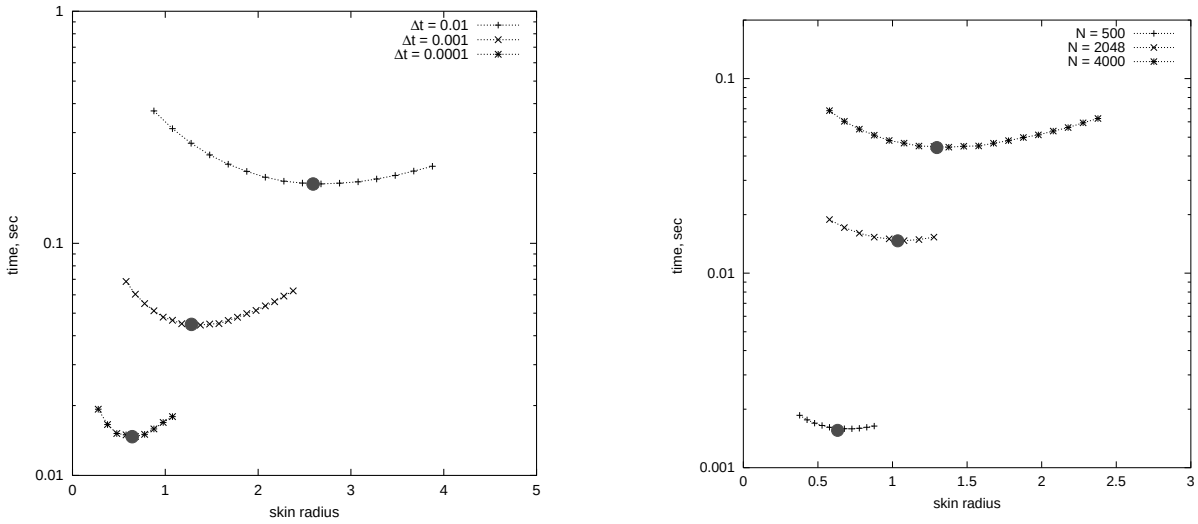


Figure 5: Average time per force routine call as a function of R_l for different values of time step Δt (left, N=4000) and number of particles N (right, Δt=0.001) (for both figures T=2, ρ=0.45). Filled red circles correspond to the values of R_{skin} obtained as minimum of f from eq.(8) in approximation eq.(9).

3rd: $(N_n \cdot T_j) \cdot N$ – time for looking through each of the neighbour subcells

4th: $(N_n l^3 \rho \cdot T_f) \cdot N = N_n \frac{N}{M^3} T_f \cdot N$ – force calculation from particles in neighbour subcells

$$T_{l.l.} = T_i M^3 + \frac{0.5 N^2 T_f}{M^3} + N T_j N_n(m(M)) + N^2 T_f \frac{N_n(m(M))}{M^3} \tag{10}$$

This expression leads to the following remarks:

- the gain from using large M comes from the fact that it is possible to decrease the 4th term keeping the 1st and 3rd ones relatively small,
- when $M \to \infty$: 1st $= O(M^3)$, 2nd=$O(M^{-3})$, 3rd $= O(M^3)$, 4rd $= O(1)$,
- the advantage of the modified method – small volume overhead $\frac{N_n \cdot l^3}{\frac{4\pi}{3} R_{cut}^3}$ – may be really useful only if times T_i and T_j are small enough.

Numerical results

Benchmarks for different densities and numbers of subcells were performed[4]. Results are presented in Fig. 7. It is seen that:

- the modified method is better than the conventional one for high densities, more precisely – when there are many particles in the cut-off sphere ($\frac{4\pi}{3}\rho R_{cut}^3 \gg 1$),
- as it was expected from the preliminary speedup estimation the modified linked list algorithm implemented with off-set list performs slower and gives less speedup for large M than the algorithm implemented with subcell neighbour list, because T_j in the former case is much larger.

[4]See remark in section . For this benchmarks $R_{cut} = 2.5\sigma$.

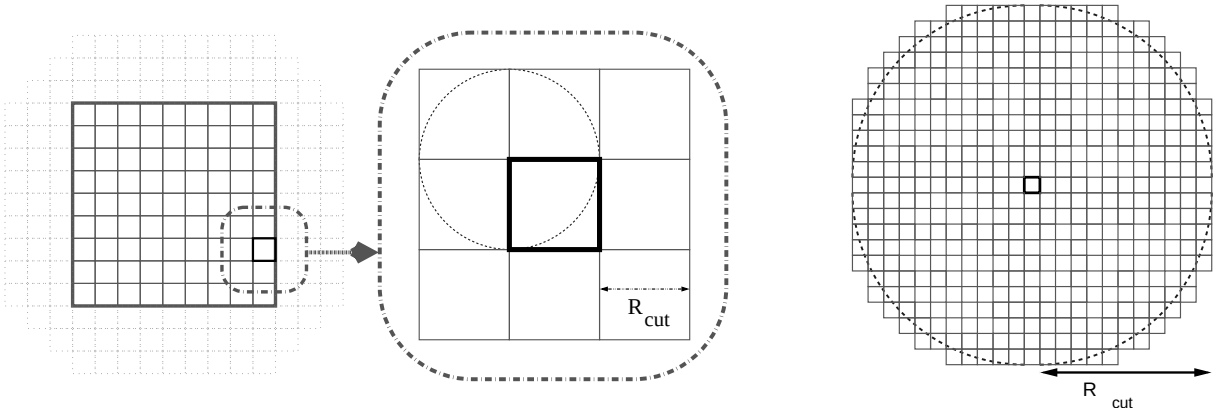


Figure 6: Illustration of the conventional (left) and modified (right) linked list methods.

Combined Verlet-Linked List Algorithm

It is possible to combine Verlet list and linked lists methods. It gives improvements to both techniques because the Verlet list update can be performed by only $O(N)$ operations using subcell sorting. This is the aim of further work.

About precision of the MD method: how much "dynamics" is in MD ?

As it was mentioned in the introduction an MD simulation consists in numerical integration of the set of the classical equations of motion for a many particle system. The result of each MD simulation is a phase-space trajectory of the system. It is possible to derive all nessesary thermodynamic, structural, correlational and other properties from the known MD trajectory. However any numerical solution contains some errors and a MD simulation is not an exception. Let's try to pose the following question: what is the real precision of the MD method, i.e. how is the numerical MD trajectory related to the hypothetical exact trajectory.

There are two reasons that separate us from the ideal case: i) the finite representation of real numbers in the computer and round-off errors, ii) the finite-difference character of the numerical scheme.

Hierarchy of approximations

Cauchy problem for equations of motion

As it was mentioned in the introduction a physical model beyond MD simulation is simple and corresponds to the set of ODE (1) and a certain initial condition $(\vec{r}^{\,o}, \vec{v}^{\,o})$ (point in the $6N$-dimensional phase space), where $\vec{r}^{\,o} = (\vec{r}^{\,o}_1, \ldots, \vec{r}^{\,o}_N)$ and $\vec{v}^{\,o} = (\vec{v}^{\,o}_1, \ldots, \vec{v}^{\,o}_N)$. Since forces practically always are smooth functions of coordinates the theorem of existence and uniqueness of the solution of the Cauchy problem is valid in this case. This means that there exists an exact (hypothetical) solution (trajectory) $(\vec{r}^{\,d}(t), \vec{v}^{\,d}(t))$ (d stands for "dynamical") for the given set (1) and for the given initial conditions (initial configuration $(\vec{r}^{\,0}, \vec{v}^{\,0})$).

Finite-difference approximation

Poincaré showed that it is not possible to obtain a solution for the N-body problem with $N \geq 3$. Hence finite difference approximations should be used for a numerical solution. There is a broad spectrum of

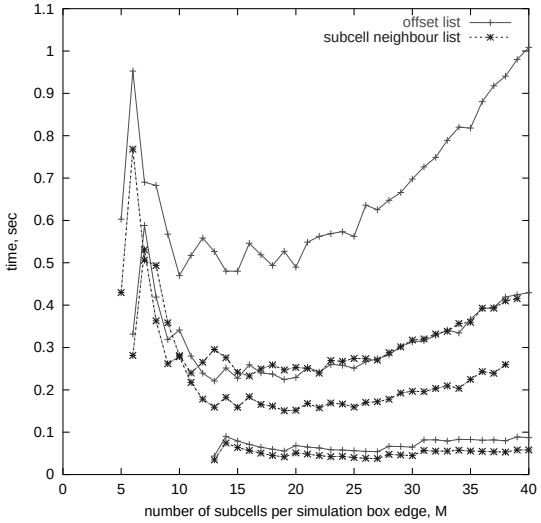


Figure 7: Average time per force routine call as a function of M for the linked list algorithm implemented with off-set list (red) and subcell-neighbour-list (blue). Different values of density are presented: $\rho = 0.1$ (lowest points), $\rho = 0.75$ (middle points), $\rho = 2$ (highest points) (T=2, N=4000, Δt=0.001). The point with smallest M in each case corresponds to the conventional method.

possible schemes, e.g. the so-called half step leap-frog:

$$\vec{v}_i(t + 1/2\Delta t) = \vec{v}_i(t - 1/2\Delta t) + \Delta t\vec{F}_i/m_i, \tag{11}$$

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{v}_i(t + 1/2\Delta t). \tag{12}$$

A particular scheme and appropriate initial conditions determine a function $\{\vec{r}^n(\Delta tk), \vec{v}^n(\Delta tk), k = 0, 1, \ldots\}$ (n stands for "numerical").

Round-off errors

For the case of pair potentials, the force acting on a tagged particle i is a sum of contributions of all its n_i neighbours

$$\vec{F}_i = \sum_{k=1}^{n_i} \vec{F}_{ij_k}, \quad j_k \in G_i. \tag{13}$$

where G_i is an ordered sequence of indices.

The structure of G_i depends on the method of force calculation: if a Verlet list is used the order of elements will be different from the case of the linked list algorithm. In the digital representation with finite accuracy, different orders of summation in eq.(13) cause different results in $\vec{F}_i$.

Let's denote the real MD trajectory obtained from calculation as $\{\vec{r}^c(\Delta tk), \vec{v}^c(\Delta tk), k = 0, 1, \ldots\}$ (c stands for "calculated").

Lyapunov instability

Two MD trajectories calculated from initially close but not coinciding configurations exponentially diverge with time. This property reflecting the chaotic nature of many particle systems is called *Lyapunov instabilty*.

Let $(\vec{r}_i(t), \vec{v}_i(t))$ be the 1-st trajectory and $(\vec{r}'_i(t), \vec{v}'_i(t))$ be the 2-nd trajectory calculated from a slightly perturbed initial configuration. Lypunov instability means that quantities

$$< \Delta r^2(t) >= \frac{1}{N} \sum_{i=1}^{N} (\vec{r}_i(t) - \vec{r}'_i(t))^2, \tag{14}$$

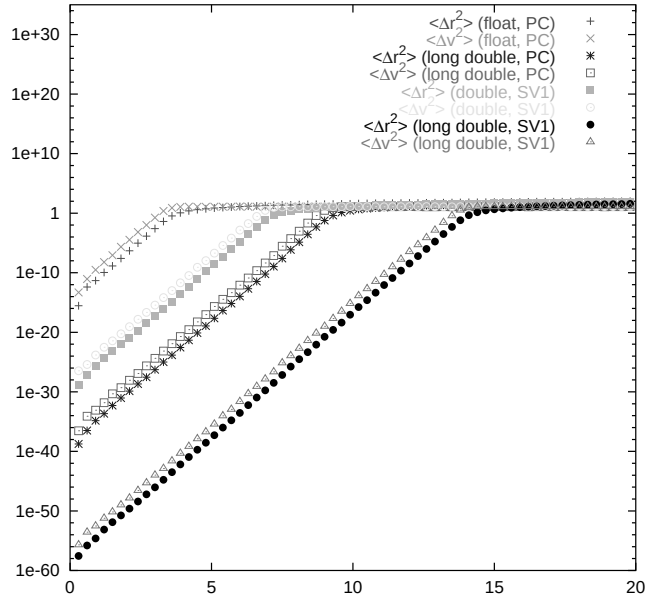


Figure 8: Divergences with time t of coordinates $< \Delta r^2(t) >$ and velocities $< \Delta v^2(t) >$ of the two trajectories calculated from an identical initial cofi guration with different sequence of particle indices. Different precisions of real number representation were used: fbat, PC – 4 bytes; long double, PC – 8 bytes; double, Cray SV1 – 8 bytes; long double, Cray SV1 – 16 bytes.

$$< \Delta v^2(t) >= \frac{1}{N}\sum_{i=1}^{N}(\vec{v}_i(t) - \vec{v}_i'(t))^2, \tag{15}$$

will increase exponentially with time:

$$< \Delta r^2(t) >= A\exp(Kt), \quad < \Delta v^2(t) >= B\exp(Kt), \quad t_l < t < t_m \tag{16}$$

where t_l is some transient time (of the order of the inverse collision frequency), A,B and t_l are functions of the perturbation ($< \Delta r^2(0) >$, $< \Delta v^2(0) >$) of the initial configuration, K is the averaged over the whole phase space maximum positive Lyaponov exponent (this quantity depends only on the *physical* properties of the system like density and temperature and does not depend on the value of the initial perturbance).

After a certain period, t_m, differences between two trajectories become of the order of the characteristic system size, i.e. $< \Delta r^2(t) > \sim \rho^{-1/3}$ and $< \Delta v^2(t) > \sim v_{th}^2$. And the behaviour of divergence changes: $< \Delta r^2(t) >$ enters into the diffusion regime and $< \Delta v^2(t) >$ remains at a constant value. The time t_m may be considered as is a *memory time*, i.e. how long an MD system remembers that initial conditions for both trajectories were close at $t = 0$. Like A and B, the memory time t_m depends on the value of an initial perturbation.

Numerical results

Round-off errors are the weakest perturbation that is possible in computer simulations. In oder to check the effect of round-off errors they were explicitly enforced in the force routine. *Artifi cial index rearrangements* were used (see Fig. 11 in Appendix). A list of indices was used to identify neighbour particles. By randomization of the list sequence it was possible to change the sequence of force summation which thereby resulted in the desired round-off errors.

Two trajectories $\{\vec{r}^{c1}(\Delta tk), \vec{v}^{c1}(\Delta tk), k = 0, 1, \ldots\}$ and $\{\vec{r}^{c2}(\Delta tk), \vec{v}^{c2}(\Delta tk), k = 0, 1, \ldots\}$ started from identical initial configuration but particles had different indices in each case. Trajectories diverge exponentially because round-off errors act as an initial perturbation (tiny differences in the values of forces). Let us denote the memory time for this case as *computational memory time* t_m^c. This time depends on the accuracy of real number representation, e.g. in C++ float, double, long double (Fig. 8).

The value of t_m^c gives the decorrelation time between the calculated trajectory $\{\vec{r}^c(\Delta tk), \vec{v}^c(\Delta tk)\}$ and the exact solution of the particular finite-difference approximation scheme $\{\vec{r}^n(\Delta tk), \vec{v}^n(\Delta tk)\}$ for the same initial configuration.

It is possible to determine a *dynamical memory time* t_m^d which is the decorrelation time between the numerical trajectory $\{\vec{r}^n(\Delta tk), \vec{v}^n(\Delta tk)\}$ and the exact solution of the set of differential equations eq.(1) $\{\vec{r}^d(t), \vec{v}^d(t)\}$ for the same initial configuration [4]. The value of t_m^d depends on the time step and logarithmically increases like $t_m^d = C_1 - C_2 ln(\Delta t)$, where C_1 and C_2 are determined by the integration scheme. For usual values of time steps $t_m^d < t_m^c$. From the condition $t_m^d = t_m^c$ one can determine the time step for the longest correlational predictions at a given accuracy of the real number representation and for a given numerical scheme. The investigation of the possible effects of precision on the behaviour of correlation functions are the aim of future work.

Summary

The results of this work are the following:

- An analytical expression for the estimation of time comsumption for Verlet list algorithm was developed. A way to determine the optimum skin-radius for a given set of parameters N, ρ, T, R_c, Δt was proposed.
- The performed calculations showed that this expression is valid for gases and liquids. An extrapolation for dense (solid) phase gives reasonable and useful estimates for the optimum as well.
- The analytical expression for the estimation of time consumption for the linked list algorithm was developed.
- The linked list algorithm was realized for arbitrary numbers of subcells in two different variants: i) by offset lists, ii) by neighbour-subcell-list.
- The effect of round-off errors on a MD trajectory was investigated for different accuracies of the real number representation. The notion of *computational memory time* was introduced and its relation to *dynamical memory time* was discussed.
- It was shown how to determine a time step which would give the best precision of the MD trajectory for a given accuracy of the real number representation.

Acknowledgments

I would like to thank the Central Institute for Applied Mathematics and personally Dr. Rüdiger Esser for the possibility to take part in the guest student program this year. I was very glad to work under the guidance of Dr. Godehard Sutmann who explained me a lot of new things in MD and whose remarks helped me to get a clear understanding of the algorithms.

References

1. N. Attig, M. Lewerenz, G. Sutmann, R. Vogelsang, Molecular Dynamics Algorithms for Massively Parallel Computers in *Proceedings of the Workshop on Molecular Dynamics On Parallel Computers*, World Scientifi c, 2000.
2. M. P. Allen and D. J. Tildesley. *Computer simulation of liquids.* Oxford Science Publications, Oxford, 1987.
3. W. Mattson, B. M. Rice. *Comp.Phys.Comm.*, 119:135-148, 1999.
4. G. Norman, V. Stegailov. *Comp.Phys.Comm.*, 147:678-683, 2002.

```
                //linked list filling
                celli = 1/cell;
                for(int u=0; u<N; u++)
                {
                        icell = (long)(1 + floor( (z[u]+EL_Z2)*celli )  +
                                           floor( (y[u]+EL_Y2)*celli )*M        +
                                           floor( (x[u]+EL_X2)*celli )*M*M      );
                        list[u] = head[icell];
                        head[icell] = u;
                }
```

Figure 9: Sorting for linked list algorithm.

```
void offset_list_creation(double clen)
{
        long p=0;
        double  xlen, ylen, zlen;
        double rs2 = rcut2;

        xlen = floor(sqrt(rs2)/clen) + 1;
        for( double i=-xlen; i<=xlen; i++)
        {
                if(i!=0)
                ylen = floor(sqrt(rs2-sqr((fabs(i)-1)*clen))/clen) + 1;
                else
                ylen = floor(sqrt(rs2)/clen) + 1;

                for( double j=-ylen; j<=ylen; j++)
                {
                        if(i!=0 && j!=0)
                        zlen = floor( sqrt( rs2-sqr((fabs(i)-1)*clen)-sqr((fabs(j)-1)*clen)) / clen ) + 1;
                        else
                        {
                                if(j==0)
                                zlen = floor( sqrt( rs2-sqr((fabs(i)-1)*clen)) / clen ) + 1;
                                if(i==0)
                                zlen = floor( sqrt( rs2-sqr((fabs(j)-1)*clen)) / clen ) + 1;
                                if(i==0 && j==0)
                                zlen = floor( sqrt( rs2-sqr((i)*clen)-sqr((j)*clen)) / clen ) + 1;
                        }

                        for( double k=-zlen; k<=zlen; k++)
                        if ( (i>0) || (i==0 && j>0) || (i==0 && j==0 && k>0) )
                        {
                                offset_x[p] = (long)i;
                                offset_y[p] = (long)j;
                                offset_z[p] = (long)k;
                                p++;
                        }
                }
        }
}
```

Figure 10: Creation of the offset list in 3 dimensions.

```
   for(int i = 0;i<N;i++)
   {
           int tmp,j;
           tmp = iindex[i];
           j = (int)((double)N*rand()/(RAND_MAX+1.0));
           iindex[i] = iindex[j];
           iindex[j] = tmp;
   }
     ...
   if (update == 1)
   {
           //LIST UPDATE
   }
   else
   for ( int ii=0; ii<N; ii++)
   if (iindex[ii]!=N-1)
   {
           unsigned long int i;
           i = iindex[ii];
           //CALCULATION OF THE FORCE ACTING ON i-TH PARTICLE
   }
```

Figure 11: Principle method of artifi cial index rearrangement.

```
for (icell_x=1; icell_x <= M; icell_x++)
    for (icell_y=1; icell_y <= M; icell_y++)
        for (icell_z=1; icell_z <= M; icell_z++)
        {
                i = head[ icell_z + (icell_y-1)*M + (icell_x-1)*M*M ];

                while (i!=-1)
                {
                        xi = x[i];
                        yi = y[i];
                        zi = z[i];
                        fxi = fx[i];
                        fyi = fy[i];
                        fzi = fz[i];

                        //loop over all molecules below i-th in the current cell
                        j = list[i];
                        while (j!=-1)
                        {
                                xij = xi-x[j];
                                        if ( xij > EL_X2 )      xij = xij-EL_X;
                                        if ( xij < -EL_X2 )     xij = xij+EL_X;
                                yij = yi-y[j];
                                        if ( yij > EL_Y2 )      yij = yij-EL_Y;
                                        if ( yij < -EL_Y2 )     yij = yij+EL_Y;
                                zij = zi-z[j];
                                        if ( zij > EL_Z2 )      zij = zij-EL_Z;
                                        if ( zij < -EL_Z2 )     zij = zij+EL_Z;

                                rij2 = xij*xij + yij*yij + zij*zij;

                                if ( rij2 <= rcut2 )
                                {
                                        rij6 = rij2*rij2*rij2;
                                        rij12 = rij6*rij6;
                                        fxij = (24*xij/rij2)*(2/rij12-1/rij6);
                                        fyij = (24*yij/rij2)*(2/rij12-1/rij6);
                                        fzij = (24*zij/rij2)*(2/rij12-1/rij6);
                                        fxi += fxij;
                                        fyi += fyij;
                                        fzi += fzij;
                                        fx[j] -= fxij;
                                        fy[j] -= fyij;
                                        fz[j] -= fzij;
                                        U += 4*(1/rij12-1/rij6) + 1;
                                }

                                j = list[j];
                        }

                        //loop over neighbour cells
                        for (long p=0; p<NN; p++)
                        {
                                incell_x = icell_x + offset_x[p];
                                incell_y = icell_y + offset_y[p];
                                incell_z = icell_z + offset_z[p];

                                if (incell_x > M) incell_x -= M;
                                if (incell_x < 1) incell_x += M;
                                if (incell_y > M) incell_y -= M;
                                if (incell_y < 1) incell_y += M;
                                if (incell_z > M) incell_z -= M;
                                if (incell_z < 1) incell_z += M;

                                j = head[ incell_z + (incell_y-1)*M + (incell_x-1)*M*M ];
                                while (j!=-1)
                                {
                                        xij = xi-x[j];
                                                if ( xij > EL_X2 )      xij = xij-EL_X;
                                                if ( xij < -EL_X2 )     xij = xij+EL_X;
                                        ...

                                        rij2 = xij*xij + yij*yij + zij*zij;

                                        if ( rij2 <= rcut2 )
                                        {
                                                rij6 = rij2*rij2*rij2;
                                                ...
                                        }

                                        j = list[j];
                                }
                        }

                        fx[i] = fxi;
                        fy[i] = fyi;
                        fz[i] = fzi;
                        i = list[i];
                }
        }
```


Figure 12: Modifi ed linked list algorithm implemented with offset list.

```
for (icell_x=1; icell_x <= M; icell_x++)
for (icell_y=1; icell_y <= M; icell_y++)
for (icell_z=1; icell_z <= M; icell_z++)
{
        unsigned long cn = icell_z+(icell_y-1)*M+(icell_x-1)*M*M;
        for(int p=0;p<NN;p++)
        {
                unsigned long cn = icell_z+(icell_y-1)*M+(icell_x-1)*M*M;

                incell_x = icell_x + offset_x[p];
                incell_y = icell_y + offset_y[p];
                incell_z = icell_z + offset_z[p];

                if (incell_x > M) incell_x -= M;
                if (incell_x < 1) incell_x += M;
                if (incell_y > M) incell_y -= M;
                if (incell_y < 1) incell_y += M;
                if (incell_z > M) incell_z -= M;
                if (incell_z < 1) incell_z += M;

                cneighbours[(cn-1)*NN+p] = incell_z+(incell_y-1)*M+(incell_x-1)*M*M;
        }
}

...

//force calculation
for (icell=1; icell <= M3; icell++)
{
        i = head[icell];
        while (i!=-1)
        {
                xi = x[i];
                yi = y[i];
                zi = z[i];
                fxi = fx[i];
                fyi = fy[i];
                fzi = fz[i];

                //loop over all molecules below i-th in the current cell
                j = list[i];
                while (j!=-1)
                {
                        xij = xi-x[j];
                                if ( xij > EL_X2 )      xij = xij-EL_X;
                        ...
                        j = list[j];
                }

                ncell_first = (icell-1)*NN;
                ncell_last = ncell_first + NN;

                //loop over neighbour cells
                for (long k = ncell_first; k < ncell_last; k++)
                {
                        j = head[cneighbours[k]];
                        while (j!=-1)
                        {
                                xij = xi-x[j];
                                        if ( xij > EL_X2 )      xij = xij-EL_X;
                                ...
                                j = list[j];
                        }
                }

                fx[i] = fxi;
                fy[i] = fyi;
                fz[i] = fzi;
                i = list[i];
        }
}
```

Figure 13: Modifi ed linked list algorithm implemented with subcell neighbour list.

Implementation of a parallel Fast Multipole Method

Jürgen Wieferink

University of Münster

wieferink@uni-muenster.de

Abstract: The Fast Multipole Method (FMM) is an effective way for calculating Coulomb energies. To make calculations even faster this method is ported to parallel computers. The "Global Arrays" package is used to accomplish this. The work and data distribution used is meant to run on parallel computers with about eight processors. On machines of this size the scaling is almost ideal, and even for 16 or 32 processors there is a considerable speed up.

Introduction

Nowadays simulation calculations are indispensable for understanding complex physical and chemical systems and processes. Performing such simulations in the context of Molecular Dynamics (MD) or Density Functional Theory (DFT), there is always the need to calculate the Coulomb field.

Direct methods for calculating these fields take very long for larger systems because of the far range of the Coulomb forces. They cannot be truncated at a certain distance. N being the number of particles or the size of the system for continuous charge patterns, the direct methods lead to $\mathcal{O}(N^2)$ scaling.

The Fast Multipole Method (FMM) is a more effective way for calculating Coulomb fields, forces or energies because it obeys linear scaling. Therefore, this method provides the opportunity to cope with systems which were previously not workable.

This work begins with an introduction to the mathematical basics like multipole and Taylor expansion of a Coulomb field and their behaviour under coordinate transformation. Then the algorithm and the actual implementation and parallelism is presented. At last, benchmarks are shown.

Mathematical basics

The spherical harmonics

At the beginning a few standard functions are discussed. The definitions and other properties are taken from [4] and [5]. There are the ordinary Legendre polynomials, given explicitly by

$$P_l(x) := \frac{1}{2^l l!}\frac{d^l}{dx^l}(x^2-1)^l \qquad l = 0, 1, 2, \ldots$$

or implicitly by

$$\frac{1}{\sqrt{1+\varrho^2-2\varrho x}} = \sum_{l=0}^{\infty} P_l(x)\varrho^l. \tag{1}$$

The Legendre polynomials are limited ($|P_l(x)| \leq 1$) for $|x| \leq 1$, the only range of interest. The associated Legendre polynomials are defined by

$$P_{lm}(x) := (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x) \qquad m = -l, \ldots, l.$$

With these functions, the spherical harmonics

$$Y_{lm}(\vartheta, \varphi) = \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_{lm}(\cos\vartheta) e^{im\varphi} \tag{2}$$

are constructed, which obey the addition theorem [5]

$$P_l(\cos\gamma) = \frac{4\pi}{2l+1} \sum_{m=-l}^{+l} Y_{lm}^*(\vartheta_0, \varphi_0) Y_{lm}(\vartheta, \varphi), \tag{3}$$

where γ is the angle between (ϑ_0, φ_0) and (ϑ, φ): $\cos\gamma = \cos\vartheta\cos\vartheta_0 + \sin\vartheta\sin\vartheta_0\cos(\varphi - \varphi_0)$.

The task

The FMM is applied to the calculation of the total Coulomb energy[1]

$$E_c = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{q_i q_j}{|\mathbf{r}_i - \mathbf{r}_j|} \tag{4}$$

of a system and the Coulomb forces $\nabla_{\mathbf{r}_i} E_c$ acting on individual particles. $\mathbf{r}_i$ and q_i are position and charge of the ith particle and N is the overall number of particles. For a continuous charge pattern the Coulomb energy is written as

$$E_c = \frac{1}{2} \int\limits_{R^3} d^3 r_0 \int\limits_{R^3} d^3 r \, \frac{\rho(\mathbf{r})\rho(\mathbf{r}_0)}{|\mathbf{r} - \mathbf{r}_0|}$$

instead. These two notations are equivalent as long as $\rho(\mathbf{r}) = \sum_{i=1}^{N} q_i \delta(\mathbf{r} - \mathbf{r}_i)$. Direct solution methods require an effort of the order $\mathcal{O}(N^2)$.

The idea

Since the direct method is not feasible for current problems, there is need to do some optimisation. The naive solution would be to simply truncate all long range interactions, thus to neglect all terms with $|\mathbf{r}_i - \mathbf{r}_j| > r_0$. But it turns out that this leads to wrong results. Obviously long range interactions are essential to the Coulomb field.

It has to be examined how two separated charge patterns as shown in figure 1 interact with each other. Of particular interest is the effect of the external charge pattern (ρ_{ex}) on the one located at point of origin (ρ_{loc}). The Coulomb energy

$$\begin{aligned} E_c &= \frac{1}{2} \int d^3 r \int d^3 r' \frac{(\rho_{\mathrm{ex}}(\mathbf{r}) + \rho_{\mathrm{loc}}(\mathbf{r}))\,(\rho_{\mathrm{ex}}(\mathbf{r}') + \rho_{\mathrm{loc}}(\mathbf{r}'))}{|\mathbf{r} - \mathbf{r}'|} \\ &= \frac{1}{2} \iint d^3 r_0 d^3 r_0' \frac{\rho_{\mathrm{ex}}(\mathbf{r}_0)\rho_{\mathrm{ex}}(\mathbf{r}_0')}{|\mathbf{r}_0 - \mathbf{r}_0'|} + \frac{1}{2} \iint d^3 r d^3 r' \frac{\rho_{\mathrm{loc}}(\mathbf{r})\rho_{\mathrm{loc}}(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} + \iint d^3 r d^3 r_0 \frac{\rho_{\mathrm{loc}}(\mathbf{r})\rho_{\mathrm{ex}}(\mathbf{r}_0)}{|\mathbf{r} - \mathbf{r}_0|} \\ &=: E_{\mathrm{ex}} + E_{\mathrm{loc}} + 2E_{\mathrm{ia}} \end{aligned}$$

[1] using cgs or Gauss system

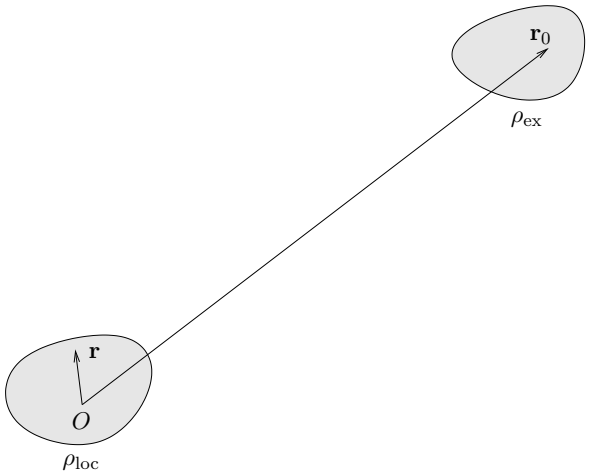


Figure 1: Two spatially separated charge patterns

splits into the internal parts E_{ex} and E_{loc}, and the interaction part E_{ia}. The former ones still have to be calculated directly. So the latter part

$$E_{\mathrm{ia}} = \frac{1}{2}\int d^3r \rho_{\mathrm{loc}}(\mathbf{r})\Phi_{\mathrm{ex}\to\mathrm{loc}}(\mathbf{r}) \qquad \text{with the potential} \qquad \Phi_{\mathrm{ex}\to\mathrm{loc}}(\mathbf{r}) := \int d^3r_0 \frac{\rho_{\mathrm{ex}}}{|\mathbf{r}-\mathbf{r}_0|} \tag{5}$$

is the interesting one. To deal with it the implicit definition of the Legendre polynomials (1) and the addition theorem (3) are used:

$$\begin{aligned}\frac{1}{|\mathbf{r}-\mathbf{r}_0|} &= \frac{1}{\sqrt{r^2+r_0^2-2rr_0\cos\gamma_{\mathbf{r},\mathbf{r}_0}}} \overset{(1)}{=} \sum_{l=0}^{\infty}\frac{r^l}{r_0^{l+1}}P_l(\cos\gamma_{\mathbf{r},\mathbf{r}_0}) \\ &\overset{(3)}{=} \sum_{l=0}^{\infty}\sum_{m=-l}^{+l}\underbrace{r^l Y_{lm}^*(\vartheta,\varphi)}_{=:Q_{lm}(\mathbf{r})}\underbrace{\frac{4\pi}{2l+1}\frac{1}{r_0^{l+1}}Y_{lm}(\vartheta_0,\varphi_0)}_{=:N_{lm}(\mathbf{r}_0)} = \sum_{l,m}Q_{lm}(\mathbf{r})N_{lm}(\mathbf{r}_0)\end{aligned} \tag{6}$$

The bottom line is that the coordinates factorise. Thus E_{ia} can be written

$$\begin{aligned}E_{\mathrm{ia}} &= \frac{1}{2}\int d^3r\int d^3r_0\frac{\rho_{\mathrm{loc}}(\mathbf{r})\rho_{\mathrm{ex}}(\mathbf{r}_0)}{|\mathbf{r}-\mathbf{r}_0|} \\ &= \frac{1}{2}\sum_{l,m}\underbrace{\left\{\int d^3r\rho_{\mathrm{loc}}(\mathbf{r})Q_{lm}(\mathbf{r})\right\}}_{=:q_{lm}}\cdot\underbrace{\left\{\int d^3r_0\rho_{\mathrm{ex}}(\mathbf{r}_0)N_{lm}(\mathbf{r}_0)\right\}}_{=:p_{lm}} = \frac{1}{2}\sum_{l,m}q_{lm}p_{lm}.\end{aligned} \tag{7}$$

The coefficients q_{lm} and p_{lm} depend only on the appropriate charge pattern.

The infinite sum in (6) has to be approximated by a finite one up to $l = p$. The error made is smaller than

$$\begin{aligned}\left|\frac{1}{|\mathbf{r}-\mathbf{r}_0|}-\sum_{l=0}^{p}\frac{r^l}{r_0^{l+1}}P_l(\cos\gamma_{\mathbf{r},\mathbf{r}_0})\right| &= \left|\sum_{l=p+1}^{\infty}P_l(\cos\gamma_{\mathbf{r},\mathbf{r}_0})\frac{r^l}{r_0^{l+1}}\right| \\ &\le \frac{r^{p+1}}{r_0^{p+2}}\sum_{l=0}^{\infty}\left(\frac{r}{r_0}\right)^l = \frac{1}{r_0-r}\left(\frac{r}{r_0}\right)^{p+1}.\end{aligned} \tag{8}$$

This shows that $r < r_0$ is required to obtain convergence. Thus, in (7) it must be possible to construct a sphere around the origin so that ρ_{ex} is completely outside and ρ_{loc} completely inside. The approximation is getting better with higher multipole moments (larger p) and a better separation (smaller r/r_0).

Interpretation of the introduced quantities

The q_{lm} are the l-pole moments of the charge pattern ρ_{loc}. They are the coefficients of an expansion of the Coulomb field far away from the origin.

$$\begin{aligned}\Phi_{\mathrm{loc}\to\mathrm{ex}}(\mathbf{r}_0) &= \int d^3r \frac{\rho_{\mathrm{loc}}(\mathbf{r})}{|\mathbf{r}-\mathbf{r}_0|} \overset{(6)}{=} \int d^3r \sum_{l,m} \rho_{\mathrm{loc}}(\mathbf{r}) \frac{r^l}{r_0^{l+1}} Q_{lm}(\mathbf{r}) N_{lm}(\mathbf{r}_0) \\ &\overset{(7)}{=} \sum_{l,m} \frac{4\pi}{2l+1}\, q_{lm}\, \frac{Y_{lm}(\vartheta_0,\varphi_0)}{r_0^{l+1}}\end{aligned} \tag{9}$$

To get to know what the p_{lm} mean, (7) has to be rearranged.

$$E_{\mathrm{ia}} = \frac{1}{2}\sum_{l,m} q_{lm}p_{lm} = \frac{1}{2}\int d^3r \rho_{\mathrm{loc}}(\mathbf{r}) \sum_{l,m} Q_{lm}(\mathbf{r}) p_{lm}$$

A comparison to equation (5) leads to

$$\Phi_{\mathrm{ex}\to\mathrm{loc}}(\mathbf{r}) = \sum_{l,m} Q_{lm}(\mathbf{r}) p_{lm} = \sum_{l,m}\, p_{lm}\, r^l Y_{lm}^*(\vartheta,\varphi) \tag{10}$$

as a Taylor expansion of the external generated Coulomb field around the origin with p_{lm} as coefficients. For further information [5, ch. 2.3.8] may be recommended.

As can be seen in (7), q_{lm} and p_{lm} depend linearly on ρ_{loc} and ρ_{ex}. Thus for expansions around the same origin the principle of superposition is effective.

Used conventions

White and Head-Gordon [2] have introduced some slightly different notation. They use scaled associated Legendre polynomials defined by:

$$\begin{aligned}\tilde{P}_{l|m|}(x) &:= \frac{1}{(l+|m|)!} P_{l|m|} & \tilde{P}_{l,-|m|}(x) &:= (-1)^m \tilde{P}_{l|m|} \\ \tilde{\tilde{P}}_{l|m|}(x) &:= (l-|m|)! P_{l|m|} & \tilde{\tilde{P}}_{l,-|m|}(x) &:= (-1)^m \tilde{\tilde{P}}_{l|m|}\end{aligned}$$

Thus the expansion (6) changes to

$$\frac{1}{|\mathbf{r}-\mathbf{r}_0|} = \sum_{l=0}^{\infty}\sum_{m=-l}^{+l} \underbrace{r^l \tilde{P}_{lm}(\cos\vartheta) e^{-im\varphi}}_{=:O_{lm}(\mathbf{r})} \cdot \underbrace{\frac{1}{r_0^{l+1}} \tilde{\tilde{P}}_{lm}(\cos\vartheta_0) e^{im\varphi_0}}_{=:M_{lm}(\mathbf{r}_0)}. \tag{11}$$

The multipole and Taylor coefficients scale to

$$\begin{aligned}\omega_{lm} &:= \int d^3r \rho_{\mathrm{loc}}(\mathbf{r}) O_{lm}(\mathbf{r}) = \int d^3r \rho_{\mathrm{loc}}(\mathbf{r}) r^l \tilde{P}(\cos\vartheta) e^{-im\varphi} \\ \mu_{lm} &:= \int d^3r_0 \rho_{\mathrm{ex}}(\mathbf{r}_0) M_{lm}(\mathbf{r}_0) = \int d^3r_0 \rho_{\mathrm{ex}}(\mathbf{r}_0) \frac{1}{r_0^{l+1}} \tilde{\tilde{P}}(\cos\vartheta_0) e^{im\varphi_0}.\end{aligned} \tag{12}$$

These equations show that $\omega_{l,-m} = \omega_{lm}^*$ and $\mu_{l,-m} = \mu_{lm}^*$ for real-valued ρ_{ex} and ρ_{loc}. Thus only the coefficients with $m \geq 0$ have to be computed and stored.

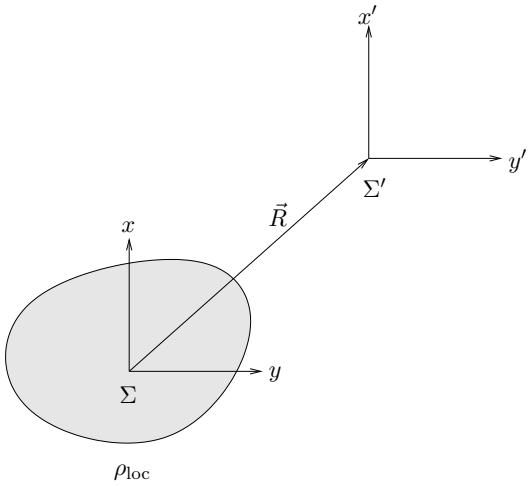


Figure 2: Translation of ω_{lm} into a new coordinate system. *This means a conversion of a multipole expansion* (9) *into a new one about another origin.*

Coordinate transformations

It is of particular interest how the introduced coefficients behave under coordinate transformation.

Translation

The most important transformation is the translation. Starting point is

$$O_{lm}(\mathbf{r}_1 + \mathbf{r}_2) = \sum_{j=0}^{l} \sum_{k=-j}^{j} O_{l-j,m-k}(\mathbf{r}_1) O_{jk}(\mathbf{r}_2), \tag{13}$$

which is taken from [2]. A hint to the proof is given at [1]. Let a multipole expansion in the coordinate system Σ be given and one in a coordinate system Σ' shifted with $\vec{R}$ ($\mathbf{r} = \mathbf{r}' + \vec{R}$) be desired:

$$\begin{aligned} \omega'_{lm} &= \int d^3r' \rho'_{\mathrm{loc}}(\mathbf{r}') O_{lm}(\mathbf{r}') = \int d^3r \rho_{\mathrm{loc}}(\mathbf{r}) O_{lm}(\mathbf{r} - \vec{R}) \\ &= \sum_{j=0}^{l} \sum_{k=-j}^{j} \int d^3r \rho_{\mathrm{loc}}(\mathbf{r}) O_{l-j,m-k}(-\vec{R}) O_{jk}(\mathbf{r}) = \sum_{j=0}^{l} \sum_{k=-j}^{j} O_{l-j,m-k}(-\vec{R}) \omega_{jk} \\ &=: \sum_{j=0}^{l} \sum_{k=-j}^{j} A_{jk}^{lm}(-\vec{R}) \omega_{jk} \end{aligned} \tag{14}$$

Hence the shift of the expansion corresponds to a matrix multiplication of the coefficients.

The next formula needed should transform ω_{lm} into μ'_{lm}. This is a transformation of a multipole expan-

sion of a far field (9) to a local Taylor expansion (10) about a distant point. The formula

$$\begin{aligned}\sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{r})M_{lm}(\mathbf{r}_0') &\overset{(11)}{=} \frac{1}{|\mathbf{r}-\mathbf{r}_0'|} = \frac{1}{|\mathbf{r}-\mathbf{r}_0+\vec{R}|} \overset{(11)}{=} \sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{r}-\mathbf{r}_0)M_{lm}(-\vec{R}) \\ &\overset{(13)}{=} \sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{r}) \left[\sum_{j=0}^{\infty}\sum_{k=-j}^{j} M_{j+l,k+m}(-\vec{R})O_{jk}(-\mathbf{r}_0)\right] \\ \Rightarrow M_{lm}(\mathbf{r}_0) &= \sum_{j=0}^{\infty}\sum_{k=-j}^{j} (-1)^{j+k} M_{j+l,k+m}(-\vec{R})O_{jk}(\mathbf{r}_0)\end{aligned} \tag{15}$$

can be used to obtain

$$\mu_{lm}' = \sum_{j=0}^{\infty}\sum_{m=-l}^{l} (-1)^{j+k} M_{j+l,k+m}(-\vec{R})\omega_{kj}. \tag{16}$$

analogue to the transformation to ω_{lm}'. In the last step of (15)

$$O_{jk}(-\mathbf{r}) = r^j \cdot \tilde{P}_{jk}(-\cos\vartheta) \cdot e^{-ik(\varphi+\pi)} = r^j \cdot (-1)^j \tilde{P}_{jk}(\cos\vartheta) \cdot (-1)^k e^{-ik\varphi} = (-1)^{j+k} O_{jk}(\mathbf{r})$$

is used. As showed by White and Head-Gordon [2] ignoring the terms with $j > p$ or $l > p$ leads to errors which behave qualitatively like the ones in (8).

At last a transformation formula from μ_{lm} to μ_{lm}' is required.

$$\begin{aligned}\sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{r})M_{lm}(\mathbf{r}_0+\vec{R}) &\overset{(11)}{=} \frac{1}{|\mathbf{r}-(\mathbf{r}_0+\vec{R})|} = \frac{1}{|(\mathbf{r}-\vec{R})-\mathbf{r}_0|} \\ &\overset{(11)}{=} \sum_{l=0}^{\infty}\sum_{m=-l}^{l} O_{lm}(\mathbf{r}-\vec{R})M_{lm}(\mathbf{r}_0) \\ &\overset{(13)}{=} \sum_{j=0}^{\infty}\sum_{k=-j}^{j} O_{jk}(\mathbf{r}) \left[\sum_{l=j}^{\infty}\sum_{m=-l}^{l} O_{l-j,m-k}(-\vec{R})M_{lm}(\mathbf{r}_0)\right]\end{aligned}$$

after changing summation indices leads to

$$\mu_{lm}' = \sum_{j=l}^{\infty}\sum_{k=-j}^{j} O_{j-l,k-m}(-\vec{R})\mu_{jk} =: \sum_{j=l}^{\infty}\sum_{k=-j}^{j} C_{jk}^{lm}(-\vec{R})\mu_{jk}. \tag{17}$$

Rotation

As one can see in (14), (16) and (17), each of these transformations needs $\mathcal{O}(p^4)$ floating point operations (FLOPS). According to White and Head-Gordon [3] this can be reduced to $\mathcal{O}(p^3)$ FLOPS using rotations. This is accomplished by rotating the coordinate system to achieve that translation can be done along the z axis ($\theta = 0$). Because of $P_{lm}(\cos 0) = P_{lm}(1) = \delta_{m0}$ all non-zero m terms vanish, and the translation only needs $\mathcal{O}(p^3)$ FLOPS. The complexity of a rotation also scales with $\mathcal{O}(p^3)$, thus the whole transformation behaves this way, too.

Assuming $\vec{R} = R\mathbf{e}_z$ to be parallel to the z or quantisation axis, the translations simplify to

$$\omega'_{lm} = \sum_{j=m}^{l} (-1)^{l-j} \frac{R^{l-j}}{(l-j)!} \omega_{jm}$$
$$\mu'_{lm} = \sum_{j=m}^{\infty} (-1)^{l-m} \frac{(j+l)!}{R^{j+l+1}} \omega_{j,-m}$$
$$\mu'_{lm} = \sum_{j=l}^{\infty} (-1)^{j-l} \frac{R^{j-l}}{(j-l)!} \mu_{jm}$$

For rotation the Wigner D matrices like in

$$Y_{lm}(\vartheta + \theta, \varphi + \phi) = \sum_{k=-l}^{l} D^l_{km}(\theta, \phi) Y_{lk}(\vartheta, \varphi)$$

are used. For O_{lm} and M_{lm} this means:

$$O_{lm}(r, \vartheta + \theta, \varphi + \phi) = \sum_{k=-l}^{l} \sqrt{\frac{(l-k)!(l+k)!}{(l-|m|)!(l+|m|)!}} D^l_{km}(\theta, \phi) O_{lk}(r, \vartheta, \varphi)$$
$$M_{lm}(r, \vartheta + \theta, \varphi + \phi) = \sum_{k=-l}^{l} \sqrt{\frac{(l-|m|)!(l+|m|)!}{(l-k)!(l+k)!}} D^l_{km}(\theta, \phi) M_{lk}(r, \vartheta, \varphi)$$

According to White and Head-Gordon [3][2] the rotation matrices are analytically given by $D^l_{km}(\theta, \phi) = e^{-ik\phi} d^l_{km}(\theta)$ with

$$\begin{aligned} d^l_{km}(\theta) =& \frac{1}{2^l} \sqrt{\frac{(l-m)!(l+m)!}{(l-k)!(l+k)!}} (1 + \operatorname{sgn}(k) \cos\theta)^{|k|} (\sin\theta)^{m-|k|} \cdot \\ & \cdot \sum_{n=0}^{l-m} (-1)^{l-m-n} \binom{l-k}{n} \binom{l+k}{l-m-n} (1+\cos\theta)^n (1-\cos\theta)^{l-m-n}. \end{aligned}$$

Because of

$$d^l_{mk}(\theta) = (-1)^{k+m} d^l_{km}(\theta) \qquad \text{and} \qquad d^l_{km}(\theta) = (-1)^{k+m} d^l_{-k,-m}(\theta)$$

these matrices have to be calculated only for

$$0 \le l \le p \qquad |k| \le l \qquad |k| \le m \le l$$

This is handled by the recursion relation

$$d^l_{k+1,m}(\theta) = \frac{k+m}{\sqrt{l(l+1) - k(k+1)}} \frac{\sin\theta}{1+\cos\theta} d^l_{km}(\theta) + \sqrt{\frac{l(l+1) - m(m-1)}{l(l+1) - k(k+1)}} d^l_{k,m-1}(\theta)$$

with the starting point

$$d^l_{0m}(\theta) = \begin{cases} \sqrt{\frac{(l-m)!}{(l+m)!}} P_{lm}(\cos\theta) & \text{for } m \ge 0 \\ (-1)^m \sqrt{\frac{(l-|m|)!}{(l+|m|)!}} P_{l|m|}(\cos\theta) & \text{for } m < 0 \end{cases}$$

[2] who reference Edmonds [6]

and for stability reasons also by

$$d^l_{k,m-1}(\theta) = \sqrt{\frac{l(l+1)-k(k+1)}{l(l+1)-m(m-1)}} d^l_{k+1,m}(\theta) - \frac{k+m}{\sqrt{l(l+1)-m(m-1)}} \frac{\sin\theta}{1+\cos\theta} d^l_{km}(\theta)$$

with starting point

$$d^l_{kl}(\theta) = \frac{1}{2^l}\sqrt{\frac{(2l)!}{(l-k)!(l+k)!}}(\sin\theta)^{l-k}(1+\cos\theta)^k.$$

The computation of the rotation matrices produces some overhead. So this method is reasonable only if each matrix is used many times after being calculated. Because of

$$d^l_{km}(\pi-\theta) = (-1)^{l+k} d^l_{mk}(\theta) = (-1)^{l-m} d^l_{km}(\theta) \tag{18}$$

only rotation matrices for angles $0 \leq \theta \leq \pi/2$ have to be determined.

The algorithm

In this section the algorithm of the FMM is described mainly following White and Head-Gordon [2].

Preparations

The program starts with the Cartesian coordinates x_i, y_i, z_i and the charge q_i of each particle. The ranges are scaled in a way that all particles reside in a cube with side length "one".

Then this cube is subdivided into eight (2^3) child boxes, each with half the side length. This subdivision is performed s times, each step dividing a parent box into eight children. The dth subdivision leads to depth d and 8^d child boxes.

There is also the term "level" where the level of a subdivision is given by "*level* = *depth* + 1". The maximum depth $d_{\max}$ is called the "lowest(!) level". This subdivision is shown one dimensionally in figure 3. Of course there are only 2^d boxes in depth d in this case.

Two boxes in the same level are called "well separated" if there are at least *ws* boxes between them. If the lowest level boxes of two particles are well separated, the interaction between them is handled as far field interaction. The minimum depth (highest level) where there are still boxes well separated is called $d_{\min}$. In figure 3 *ws* is "one" and $d_{\min} = 2$.

The parameters of the FMM are the depth $s = d_{\max}$ of the subdivision, the definition of well separated *ws* and the length p of the multipole expansion.

Pass 1: The multipole moments

In pass 1 the multipole moments are calculated. First this is done in the lowest level using (12)[3] with the centre of the box as point of origin. This has to be done for each particle, and thus this part scales linearly with the number N of particles.

In the second part the multipole moments are translated to the centre of the parent boxes. Then all multipole moments belonging to one and the same box are accumulated. This is done up to depth $d_{\min}$. With M being the number of boxes in depths $d_{\max}$ to $d_{\min}$, this part obeys $\mathcal{O}(M)$ scaling. To be exactly there

[3] or the discrete counterpart $\omega_{lm} = \sum_i q_i O_{lm}(\mathbf{r}_i)$ respectively

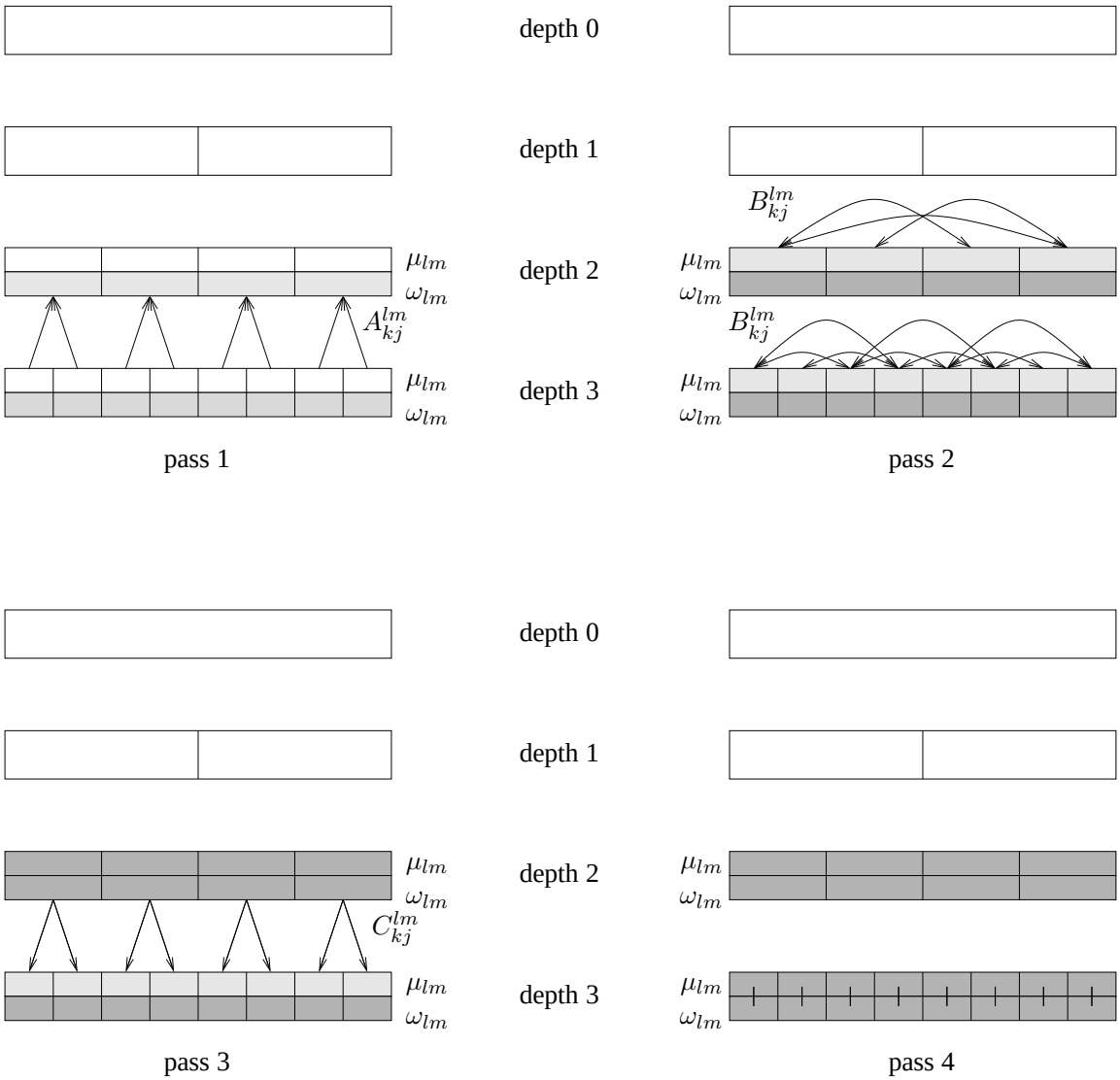


Figure 3: Schematic view on one dimensional FMM with parameters ws=1 and $d_{\max} = 3$. *Data known before a specifi c pass are drawn darker than the ones to be calculated.* After [2].

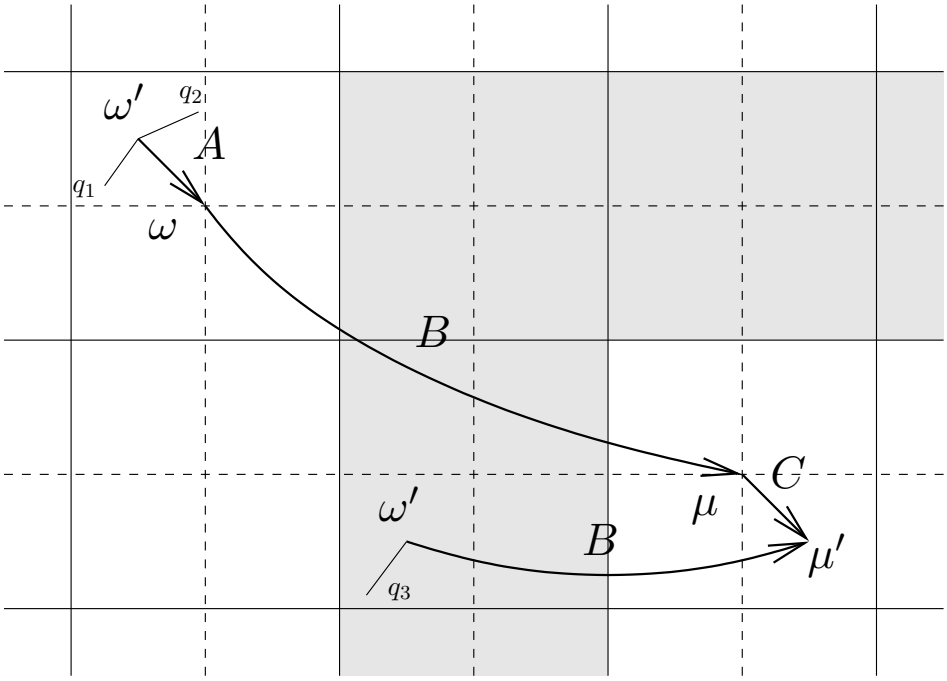


Figure 4: The first three passes of the FMM with *ws*=1. *It is shown how the information of the charges q_1, q_2 and q_3 gets to the Taylor expansion μ' in the child box in the lower right corner. Parent boxes are separated by solid lines, child boxes by dashed ones.*

are no calculations needed at depth $d_{\min}$, but this slightly more pessimistic value is used for simplicity reasons.

After pass 1 there are multipole expansions of the local charge patterns in all relevant boxes.

Pass 2: Calculation of Taylor coefficients

In the second pass the multipole expansions around the centre of one box are used to calculate the Taylor expansion around the centre of a special set of other boxes using (16). This special set consists of all other boxes which are well separated but whose parents are not. This is visualised in figure 4 where all boxes having these properties relative to the child box in the lower right corner are drawn dark (assuming *ws*=1). Again, all Taylor coefficients computed around one centre are added up.

In doing so the number of boxes for which one has to calculate the μ_{lm} from a certain multipole expansion depends only on *ws* and especially not on the system size. For *ws*=1 there are $8 \cdot 3^3 - 3^3 = 205$ target boxes for each source. Thus, this pass is of order $\mathcal{O}(M)$.

Pass 3: Translation of Taylor coefficients

Pass 3 is some kind of counterpart to pass 1. While in pass 1 the multipole expansions are shifted "up the tree", here the Taylor expansions are shifted "down the tree". This is done by translating the Taylor expansion to the centre of each child box using (17) and adding it to the Taylor expansion already calculated for this box in pass 2. The calculations have to be done for each child box, so this pass is of order $\mathcal{O}(M)$.

At the end of pass 3 for each lowest level box there is a Taylor expansion of the complete far field.

Pass 4: Calculation of far field interaction

This pass corresponds to the calculation of E_{ia}. If only the complete energy is needed the box energies $\sum_{l,m} \omega_{lm}\mu_{lm}$ are added over all boxes in lowest level giving the far field energy. If one wants to have the force acting on each particle, too, the Taylor expansions have to be evaluated for each particle.

In the first case this pass has the complexity $\mathcal{O}(M')$, M' being the number of boxes in the lowest level, and in the second case $\mathcal{O}(N)$.

Pass 5: Calculation of near field interaction

In pass 5 all interactions that have not yet been taken into account are calculated directly. It might be advantageous to separately calculate the interactions inside one box and between two boxes because then the order of magnitude will not differ that much.

In this pass, as in pass 2, we have to keep in mind that the number of boxes not well separated to a specific one is limited and depends only on *ws*. If the number of particles in the lowest level box is $z = N/M'$, this pass is of order $\mathcal{O}(zN) = \mathcal{O}(N^2/M')$.

Total complexity

If it is possible to keep the number of particles per lowest level box $z = N/M'$ constant, the number of lowest level boxes is proportional to the number of particles ($M' = N/z \propto N$). Thus the FMM scales $\mathcal{O}(N) + \mathcal{O}(M)$.

$$M = \sum_{d=d_{\min}}^{d_{\max}} 8^d = 8^{d_{\min}} \sum_{d=0}^{d_{\max}-d_{\min}} 8^d = 8^{d_{\min}} \frac{1-8^{d_{\max}-d_{\min}+1}}{1-8} = \frac{8^{d_{\max}+1}-8^{d_{\min}}}{7}$$

and $8^{d_{\max}} = M' = N/z$ show that M grows linearly with N:

$$M = \frac{8^{d_{\max}+1}-8^{d_{\min}}}{7} \lesssim \frac{8M'}{7} = \frac{8}{7}\frac{N}{z} \propto N.$$

Thus, for constant z and large N the FMM altogether obeys linear scaling for ideal homogenous charge patterns. In other cases it might be delicate to obtain a constant z. But even in this case Greengard [1] offers possibilities to obtain linear scaling on arbitrary charge patterns at least for two dimensional problems.

Fractional depth

The above considerations do not take into account that d and $d_{\max}$ have to be integer values. Thus the given complexity is only true as long as zN is a power of eight. Therefore, even with an ideal charge pattern, there generally is no ideal $d_{\max}$, so that the calculations take longer than linear scaling might suggest for disadvantageous system sizes N.

To handle this there is the concept of fractional depth. With the above formulas the ideal depth d_{frac} is calculated, and $d_{\max}$ is set to the next larger integer. Then the charge pattern is scaled by the factor $1 - 2^{d_{\mathrm{frac}}-d_{\max}}$. If all empty boxes are ignored this leads to clear speed up for accordant N.

The implementation

The task at the "Guest Student Program" is it to parallelise a given implementation of the FMM.

The given sequential implementation

The given implementation computes the Coulomb energy E_c of a particle pattern. The parameter *ws* is set constantly to "one", d_{frac} and p are determined in a way that the number of floating point operations for a given demand of accuracy is minimised.

Empty boxes do not lead to useless computations or memory usage. This is necessary because fractional depth is used, but it also increases effectivity for granular charge patterns with large charge free areas.

The rotation matrices are only needed in the first three passes. In pass 1 and pass 3 only those for $\cos\theta = 1/\sqrt{3}$ have to be calculated because the angles θ between the centres of child boxes and their parents are all equivalent in the sense of (18). Pass 2 needs some special care. The given program first examines which translations have to be done, and buffers them in the correct order. The actual translations are executed not until the buffer is full or all translations have been taken into account.

The memory usage splits to diverse buffers, $4N$ `REAL` numbers for particle positions and charges, N `INTEGER` numbers indicating which box a particle is in, and $4M \cdot (p+1)(p+2)/2 \sim 2Mp^2$ `REAL` numbers for the real and imaginary part of the ω_{lm} and μ_{lm}, where M is the number of boxes in depths $d_{\min}$ to $d_{\max}$.

The parallel implementation

One possibility of parallising is to put the initial data into a "global array", hence to distribute it to all processes, and then to compute locally as much as possible, and to exchange data only if inevitable. This needs some substantial modifications to the actual code and cannot be done in the ten weeks of the "Guest Student Program". Instead all processes compute on the same complete data set, and only some time expensive steps are executed parallely. In these steps either coefficients (ω_{lm}, μ_{lm}) or energies are calculated. For these quantities there is the principle of superposition so that a global sum can be performed to obtain the final result of the parallel section. If the program spends enough time in the parallel parts, there is hope that this leads to acceptable parallel performance at least if there are not too many processes.

The actual parallel parts are in detail:

- The calculation of the multipole moments in lowest level in pass 1.
- The translation of the multipole moments to the centre of the parent box. This has be done consecutively for each level because the results of one level are needed to calculate the multipole moments for the parent level.
- The execution of the buffered transformations in pass 2. The global sum is performed only once at the end of pass 2, even if the buffer is emptied multiple times.
- The calculation of the near field interaction within one box in pass 5.
- The calculation of the near field interaction between two boxes in pass 5. The global sum is performed on the total energy at the end of pass 5.

There are two major sequential parts left that still disturb the parallel performance. At first, the preparations before the actual FMM take some time. This is in fact an independent problem which has to be worried about separately. Thus this part will be ignored in the subsequent time measurements.

The other major sequential part is pass 3. When carrying out the FMM sequentially this pass takes less than a percent of the time, so the parallel performance does not suffer too much if nothing is done. The

problem is, that in this pass it is not possible to adopt the strategy used in the second part of pass 1 because there is already data in the μ_{lm} in all levels, so that a global sum would lead to wrong results. Direct methods to deal with this problem would consume either too much memory or too much time for communication. Instead pass 3 is left out, and pass 4 is performed on all levels. This can be done if the total energy is needed exclusively. If also the forces are of interest, pass 3 is substantial and has to be reintroduced.

The disadvantage of the chosen kind of parallel programming is the memory consumption. Each process has to memorise the complete data. This restricts the size of systems to the memory size of one process, and not to the sum of all.

The program is tested mainly on three homogenous charge patterns with $8^5 = 32768$, $8^6 = 262144$ and $8^7 = 2097152$ particles. The used platforms are the `CRAY T3E-1200` and a linux workstation cluster. On both platforms the 8^7 charge pattern is some kind of upper limit, which cannot be exceeded clearly due to memory limitations.

The results

Figure 5 shows the speed up on the `CRAY`. For 8 PEs (processing elements) there is almost no deviation from the ideal case. 32 PEs still speed up the calculations by a factor larger than 24.

In Figure 6 the time consumed for sequential parts and global sums is left away. Thus this plot only depends on the performance of the parallel calculations. Before each of these parts the processes are synchronised, so all processes start at the same time. As expected, the average times scale exactly linear. Anything else would have meant the number of calculations being changed. But also the times for the slowest PE are almost linear scaling which means that we do not have major granularity problems. Thus the deviation to linear scaling of the whole program in figure 5 must be generated by sequential parts and global sums.

To examine, which of these two factors causes more problems, one might take a look at figure 7. This figure shows that the predominant part of the speed up reduction is originated by the global sums.

All mentioned plots are created based on the results shown in table 1. The stated degrees of parallelisation, which measure the granularity problems, are all above 98%.

On the linux workstation cluster the parallel program is acceptable when running on up to eight processors, as can be seen in figure 8. The clearly worse scaling compared to the `CRAY` is due to the slower network.

Conclusion

The parallelism shown performs very well on parallel computers with eight or sixteen processors, as long as the treated system is small enough, so that memory consumption is no severe problem. For larger systems a new approach for work and data distribution would be needed, which almost inevitably would lead to more granularity problems.

Thus, the chosen strategy with replicated data is very effective, as long as there is enough memory.

Acknowledgements

I would like to thank anyone who helped to make my participation at the “Guest Student Program” possible. Especially I want to thank Dr. Esser for the management of the Guest Student Program and Dr.

Dachsel for his advice referring to FMM and the actual code.

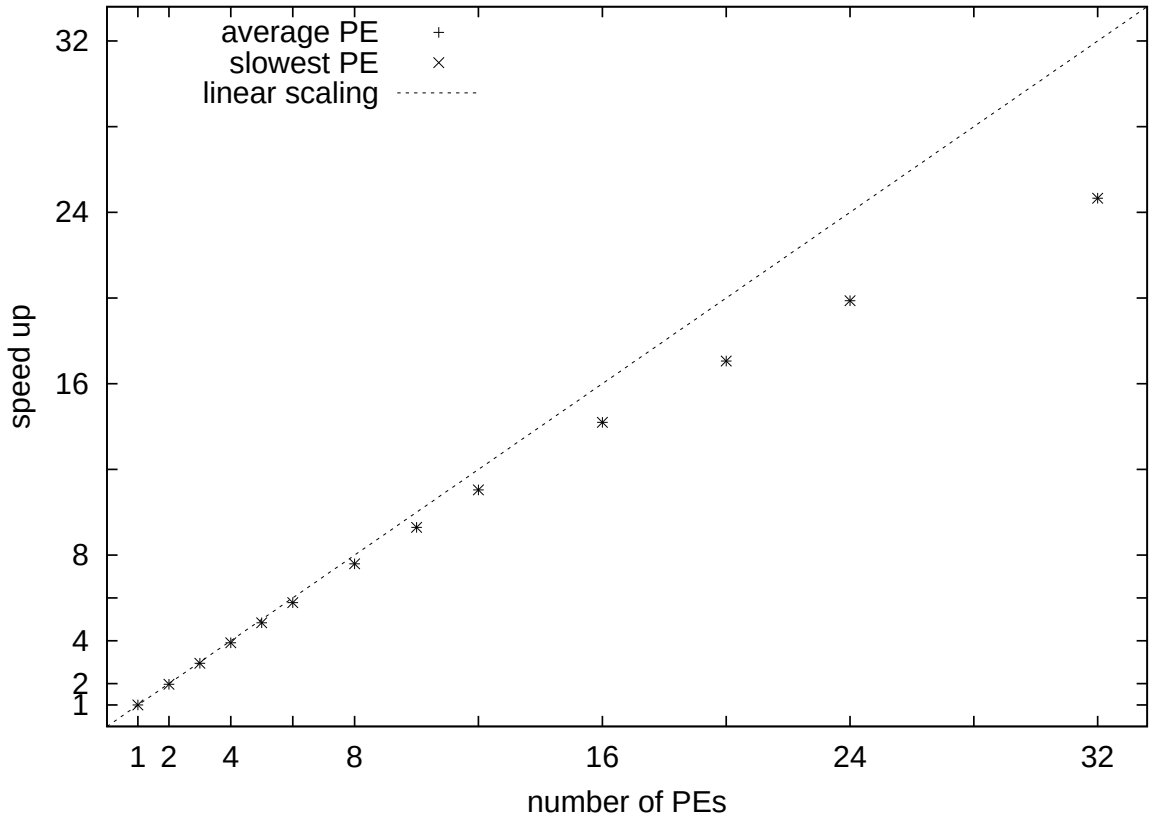


Figure 5: Total speed up on CRAY with 8^7 particles and length $p = 32$ of multipole expansion.

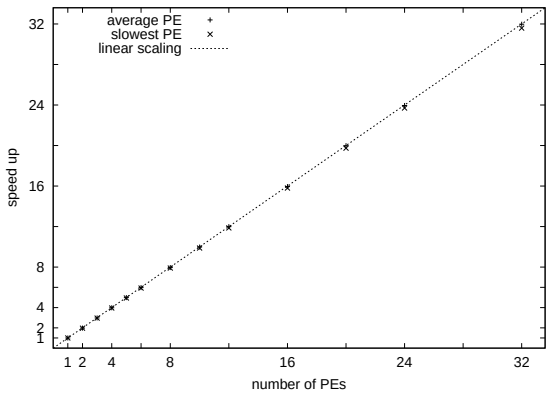


Figure 6: Speed up of only the parallel calculations in figure 5. *Neither sequential parts nor communication overhead through global sums are regarded.*

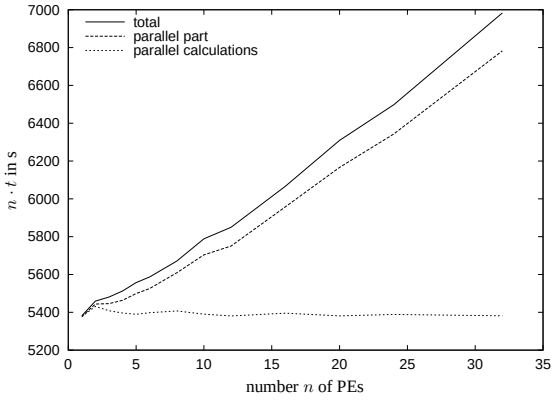


Figure 7: CPU time accumulated over all PEs. *If the program would scale ideally with the number of PEs, all lines had to be horizontal. The "total" curve corresponds to figure 5, the "parallel calculations" to figure 6. The "parallel part" curve shows the time spend in parallel parts including the global sums.*

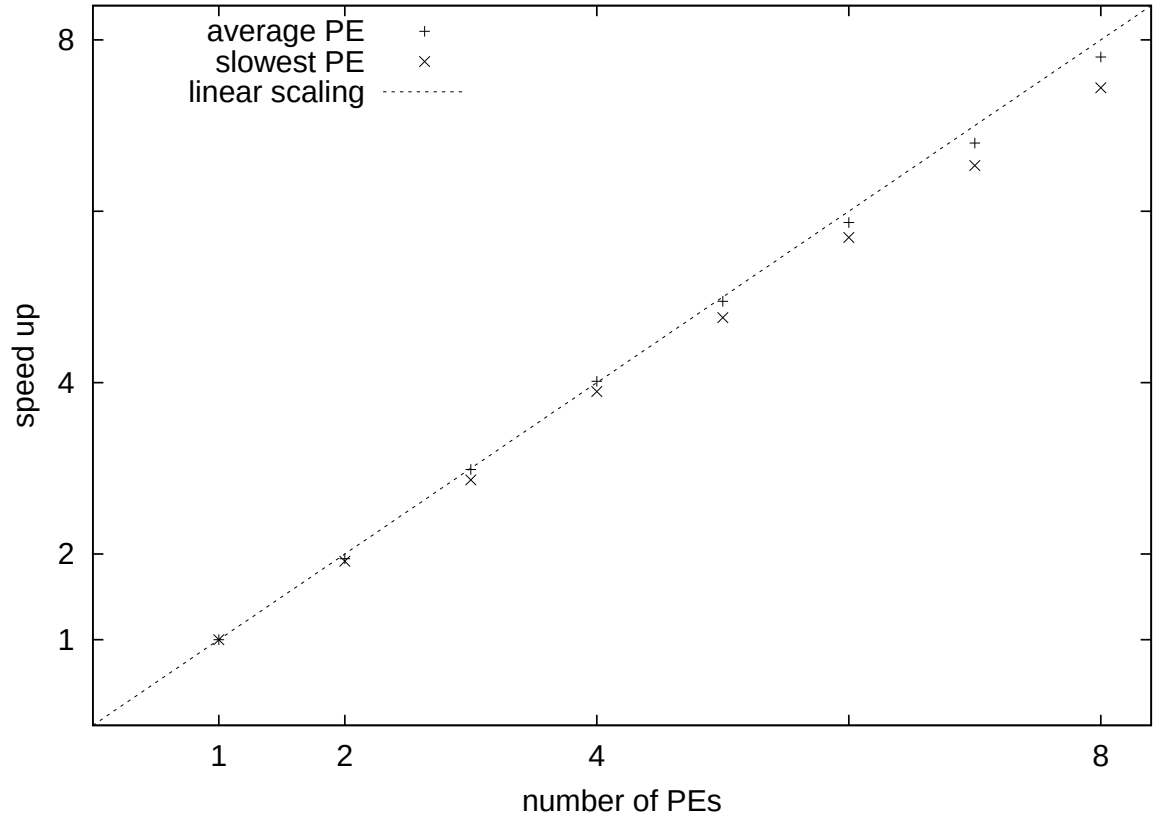


Figure 8: Speed up for cluster of linux workstations. $N = 8^7$, $p = 32$.

No.	parallel calculations					global sum			
	1a	1b	2	5a	5b	1a	1b	2	5
1	747	73	12067	156	3687	1877	278	2190	0
2	747	73	12071	156	3688	1877	278	2191	0
3	747	73	12072	156	3690	1877	278	2194	0
4	747	73	12076	156	3690	1877	278	2195	0
5	747	73	12077	156	3694	1877	278	2196	0
6	747	73	12079	157	3695	1877	278	2196	0
7	747	73	12082	157	3697	1878	278	2198	0
8	747	73	12083	157	3698	1878	278	2226	0
9	747	73	12085	157	3699	1878	278	2226	0
10	747	73	12085	157	3699	1878	278	2226	0
11	747	73	12086	157	3699	1878	278	2227	0
12	747	73	12089	157	3700	1878	278	2228	0
13	747	73	12089	157	3702	1878	278	2228	0
14	747	73	12089	157	3703	1878	278	2229	0
15	747	73	12090	157	3704	1879	278	2229	0
16	747	73	12090	158	3707	1879	278	2229	0
17	747	73	12090	158	3707	1879	278	2230	0
18	747	73	12091	158	3707	1879	278	2230	0
19	748	73	12091	158	3708	1879	278	2230	0
20	748	73	12092	158	3708	1879	278	2230	0
21	748	73	12093	158	3709	1879	278	2230	0
22	748	73	12094	158	3710	1879	278	2231	0
23	748	73	12096	158	3712	1879	278	2231	0
24	748	73	12097	158	3713	1879	278	2232	0
25	748	73	12103	159	3714	1879	278	2232	0
26	748	73	12279	159	3723	1879	278	2232	0
27	749	73	12282	159	3726	1879	278	2232	0
28	749	73	12283	159	3729	1879	278	2232	0
29	749	73	12287	159	3734	1879	278	2232	0
30	749	73	12288	159	3735	1879	278	2233	0
31	749	73	12294	159	3742	1879	278	2233	0
32	749	73	12295	159	3746	1879	278	2233	0
ave	748	73	12130	158	3709	1878	278	2222	0
err	0.8	0.0	83.3	1.0	15.3	0.8	0.0	15.0	0.0
par	99.8	100.0	98.7	99.1	99.0				

Table 1: CPU times for each PE and each parallel part in $\mathrm{cs}(= 0.01\,\mathrm{s})$. Measured with $N = 8^7$ particles and $p = 32$ length of multipole expansion.*All times are sorted in each category separately. Below are the average "*`ave`*"and the root-mean-square deviation "*`err`*". The last number is the degree of parallising* $\eta_{\mathrm{par}} = 100 \cdot \frac{\sum t_i}{n t_{\max}} = 100 \cdot (1 - \frac{\sum (t_{\max} - t_i)}{n t_{\max}})$*. This is a number showing granularity problems.*

References

1. L. GREENGARD, *The Rapid Evaluation of Potential Fields in Particle Systems*, MIT, Cambridge, 1988
2. C. A. WHITE AND M. HEAD-GORDON, *Derivation and efficient implementation of the fast multipole method*, J. Chem. Phys. **101**, p. 6593, 1994
3. C. A. WHITE AND M. HEAD-GORDON, *Rotating around the quartic angular momentum barrier in fast multipole method calculations*, J. Chem. Phys. **105**, p. 5061, 1996
4. I. N. BRONSTEIN, K. A. SEMENDJAJEW, G. MUSIOL, H. MÜHLIG, *Taschenbuch der Mathematik, 4. Auflage*, Verlag Harry Deutsch, Frankfurt am Main, 1999
5. W. NOLTING, *Grundkurs Theoretische Physik; 3 Elektrodynamik, 5. Auflage* Viehweg, 1997
6. A. R. EDMONDS, *Angular Momentum in Quantum Mechanics*, Princeton University Press, Princeton, New Jersey, 1957
7. R. N. ZARE, *Angular Momentum, Understanding Spatial Aspects in Chemistry and Physics*, Department of Chemistry, Stanford University, Stanford, California, 1988