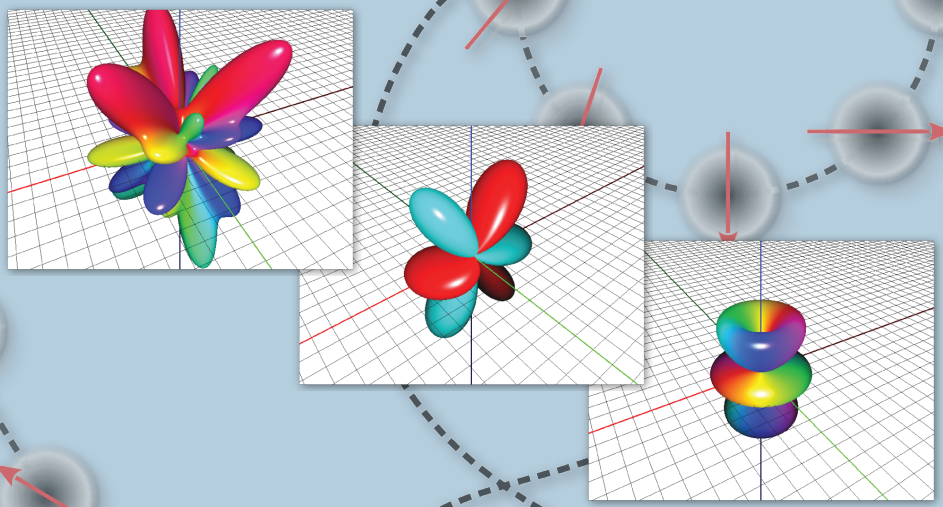


JSC Guest Student Programme Proceedings 2012

Edited by Mathias Winkel

FZJ-JSC-IB-2012-01

JÜLICH FORSCHUNGSZENTRUM

Member of the Helmholtz Association

Proceedings 2012

JSC Guest Student Programme on Scientific Computing

Editor
Mathias Winkel

November 2012

FZJ-JSC-IB-2012-01

Jülich Supercomputing Centre
Forschungszentrum Jülich GmbH
D-52425 Jülich

Tel: +49 2461 61-6402
Fax: +49 2461 61-6656
Email: jsc@fz-juelich.de
WWW: http://www.fz-juelich.de/ias/jsc/

FZJ-JSC-IB-2012-01

JSC Guest Student Programme on Scientific Computing
Proceedings 2012

The complete volume as well as reports from earlier Guest Student Programmes are freely available on http://www.fz-juelich.de/ias/jsc/gsp

Editorial

As one of the leading HPC centres in Europe, Jülich Supercomputing Centre provides supercomputer resources, IT tools and HPC expertise for computational scientists at German and European universities, research institutions, and in industry. One important part of this mission is to introduce young academics to HPC and its role in scientific research. To this end, JSC hosts regular support activities and educational programmes.

The *JSC Guest Student Programme on Scientific Computing* has already been successfully running for 13 years. Since the first programme in 2000, a total of 133 students have seized the opportunity to join research teams from JSC and other institutes on the Forschungszentrum Jülich campus for ten weeks. Working on challenging topical scientific projects, they gained experience with up-to-date hardware and software as well as HPC-related methods and algorithms. For many students, the programme has been the foundation for a career in HPC and the basis for fruitful continuing cooperations.

The JSC Guest Student Programme 2012 took place from 06 August to 12 October. Once again it was run under the CECAM framework (Centre Européen de Calcul Atomique et Moléculaire) and organized in cooperation with the German Research School for Simulation Sciences (GRS). Support by IBM Deutschland through a sponsorship within the IBM University Relations programme is also gratefully acknowledged.

This year's announcement yielded a record response of applications from 13 countries, comprising applicants from a particularly wide range of disciplines, including mathematics, physics, chemistry, computer science and engineering, biomedicine and earth sciences. Competition for the available places was especially strong, and after the final selection process, 13 students were invited to Jülich.

The guest students and their local advisers were:

Daniel Arndt	(Göttingen)	Mike Nicolai (JSC)
Kieran Austin	(Leipzig)	Thomas Neuhaus (JSC)
Mario Berljafa	(Zagreb)	Edoardo Di Napoli (JSC)
Khaldoon Ghanem	(Aachen)	Erik Koch (GRS)
David Haensel	(Dresden)	Ulrich Kemloh (JSC)
Christian Jost	(Bonn)	Stefan Krieg (JSC)
David Martín Rodríguez	(Salamanca)	Anupam Karmakar (JSC)
Sebastian Rupprecht	(Augsburg)	Martin Müser (NIC), Wolf Dapp (NIC)
Barbara Schlögl	(Würzburg)	Martin Müser (NIC), Mykola Prodanov (NIC)
Anne Springer	(Bonn)	Lars Hoffmann (JSC)
Artur Strebel	(Wuppertal)	Oliver Bücker (JSC), Wolfgang Meyer (JSC)
Felix Uhl	(Bochum)	Godehard Sutmann (JSC), Viorel Chihaia (JSC)
Tommaso Zanca	(Ferrara)	Dirk Pleiter (JSC)

During their stay, the participants worked on research projects in a broad area of fields, ranging from applications in atmospheric sciences, fluid and molecular dynamics, particle-in-cell methods, and safety research, to fundamental research in elementary particle physics and mathematical algorithms. Besides using the multi-purpose cluster *JUROPA* and the GPU system *JUDGE*, one of the key topics was the utilization of the newly installed IBM Blue Gene/Q system *JUQUEEN* in Jülich – number 5 on the November 2012 Top500 list and currently the most powerful supercomputer in Europe.

During a concluding colloquium, the participants presented and discussed their work with their fellow students, their supervisors, and other local scientists. Finally, more detailed reports have been prepared and were compiled into this JSC publication. It underlines the students' ability for self-contained, focused, and cooperative work as young scientists on up-to-date topics.

Of course, success of the programme is not only due to single persons but the result of the hard work of many contributors. I would like to thank the guest students for their work and dedication, contributing to challenging and exciting scientific topics as well as the advisers for their cooperation and patient help, not only regarding the students' work. Thanks go to the lecturers Florian Janetzko, Alexander Schnurpfeil, Jan Meinke, Peter Philippen, and Willi Homberg from JSC, Suraj Prabhakaran from GRS, and Jiri Kraus from NVIDIA who organised and held the training courses.

Additionally, I would like to thank Ivo Kabadshow who has been doing a great job as supporting organizer and Ria Schmitz for her tireless work on and against bureaucracy. Special thanks go to IBM Deutschland, in particular Andreas Pflieger and Martin Mähler, for significant financial support and a great contribution to the colloquium.

The upcoming JSC Guest Student Programme will start on August 05, 2013. It will be officially announced in January 2013 and is open to students from the natural sciences, engineering, computer science and mathematics after their Bachelor and before reaching their Master's degree. Further information, reviews, former results, and the announcement of the upcoming programme are available online at http://www.fz-juelich.de/ias/jsc/gsp.

Jülich, November 2012 *Mathias Winkel*

Contents

Embedding Plasticity into a Parallel Green's Function Molecular Dynamics Code

Barbara Schlögl

Universität Würzburg
Fakulät für Physik und Astronomie
Theoretische Physik III
Am Hubland
97074 Würzburg

E-mail: bschloegl@physik.uni-wuerzburg.de

Abstract:
Numerical simulation of plastic deformation of solid bodies is still a challenging field in contact mechanics. In this work we describe four different approaches to implement plastic deformation into a parallel Green's function molecular dynamics (GFMD) code originally developed for handling only the elastic case. The first approach of local deformation in real space suffers from the fact that its optimization is not universal over a range of pressures, as well as from discontinuities in the displacement field. The second approach of pressure-dependent deformation taking into account the position of nearest neighbours produces ringing artifacts from the transformation to Fourier space. While the last two propositions seem promising, they need some further investigation to ensure or refute their feasibility.

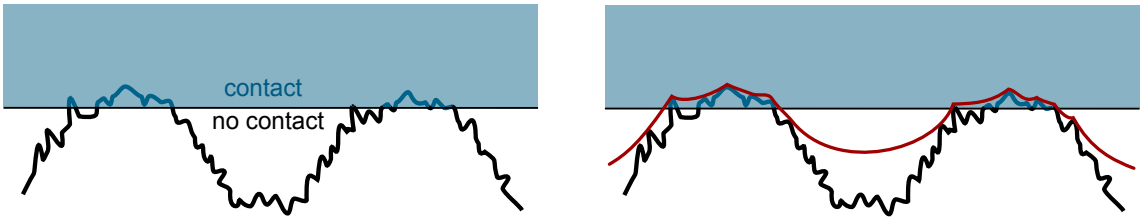
1 Introduction

Computer simulations and especially molecular dynamics play an important role in materials science. In particular contact mechanics is dealing with phenomena occuring during the interaction of two or more surfaces [1]. For this purpose the roughness and plasticity of real surfaces also have to be taken into account. Using a coarse grained approach we are able to simulate surfaces over a wide range of lengths, ranging from the atomic level to mesoscopic scales. We in particular are interested in the deformation of two bodies where one is pushed down on the other. Both are assumed to possess roughness as well as elasticity. The latter makes the traditional assumption to the contact area ineligible for our purposes, as will be explained in section 2. The section will cover the Green's function molecular dynamics (GFMD) code which enables us to examine the effects of elastic deformation. Still in many cases the influence of plastic deformation may not be negligible. Here we aim to implement modifications to the existing code to be able to take plasticity into account. We will consider four different approaches and examine their respective advantages and drawbacks.

2 GFMD code for Elastic Deformation

2.1 Set up

Consider two surfaces with roughness on different length scales ranging from the atomic to the macroscopic scale. As long as they move perpendicular to the interface we can neglect shear movement and stress. This allows us to summarize their characteristics i. e. roughness and elasticity into either one of the surface. The result may be an ideally flat surface which contains all of the elasticity in the system while the other is perfectly rigid but carries the information about the combined roughness.
Now if the surfaces are to be brought in contact we are interested in finding the contact area and accordingly the gap distribution. Traditional approaches as summarized for example in [1] and described in [2], assume the highest asperity to be in contact first, then the second, the third and so on. This effectively corresponds to cutting through the height distribution of the undeformed surface at a certain height. Everything above would constitute as a point in contact while every point below would correspond to a gap (see Fig. 1a). Even though this may be sufficient for some applications or as a first estimate, it does not take into account the elasticity of the surface. It will deform and may be pushed down further than assumed, resulting in points of additional contact below the cut line. On the other hand the particles exert forces on each other ensuring a continuous surface which may result in gaps appearing above the cut (see Fig. 1b).
This behaviour necessitates a more sophisticated approach on the task.



(a) Traditional picture of contact area

(b) Contact area of a rough and an elastic surface

Figure 1: Comparison of contact areas

2.2 Green's Function Molecular Dynamics

Molecular Dynamics (MD) allows us to compute the physical movement of interacting atoms or molecules numerically. We solve Newton's equation of motion through Verlet integration taking into account that the movement of every particle will have an influence on its neighbours. Green's function molecular dynamics in particular is concerned with the treatment of semi-infinite elastic solids. The full elastic response can be modelled though only surface atoms are taken into account. This is why the method is considerably faster than an all-atom simulation while still taking into account the long range interactions. [3]
Since we are not interested in the time evolution of the process but rather in its outcome, i. e. the static response of one surface being pushed at another with a certain pressure, we aim to damp the motion as fast as possible. For this the damping should be chosen so that the centre-of-mass mode is critically damped or slightly underdamped. One GFMD step in our implementation consists of:

- Transforming the surface topology from real to Fourier space
- Calculating the elastic restoring forces
- Adding the external pressure
- Adding the damping forces
- Computing the Verlet integration
- Transforming the surface topology back to real space
- Applying boundary conditions
- Checking for termination (whether the equilibrium is reached)

2.3 Closer Look: Boundary Conditions

In order to understand in detail how the plastic deformation was implemented in the GFMD code it's worthwhile to have a closer look at the boundary condition. All of the proposed approaches for plasticity modify them to some extent. After setting up the two surfaces, the flat, elastic displacement field and the rough, rigid counterbody, we move the particles of the displacement field according to the external pressure as predicted by the Verlet integration. Now the fundamental principle is that the two bodies may not overlap, because this would correspond to the unrealistic situation of interpenetration of the surfaces. Instead, every point violating this condition is placed on the surface of the counterbody and may subsequently be altered further in accordance with the molecular dynamics. Note that our implementation of GFMD can be used to simulate static contacts in systems of different length scales. Thus the grid points may not correspond to 'real' atoms in contrast to conventional MD. For symplicity we will refer to the grid points as 'particles' or 'points'.
The boundary condition can be expressed as:
`displacement = max(displacement, wallPos - effSurf)`
where the initial displacement is determined by the molecular dynamics, `effSurf` describes the height distribution of the rigid counterbody (effective surface) and `wallPos` is originally a fixed field of reference, which may be chosen to zero or the initial displacement (see Fig. 2).

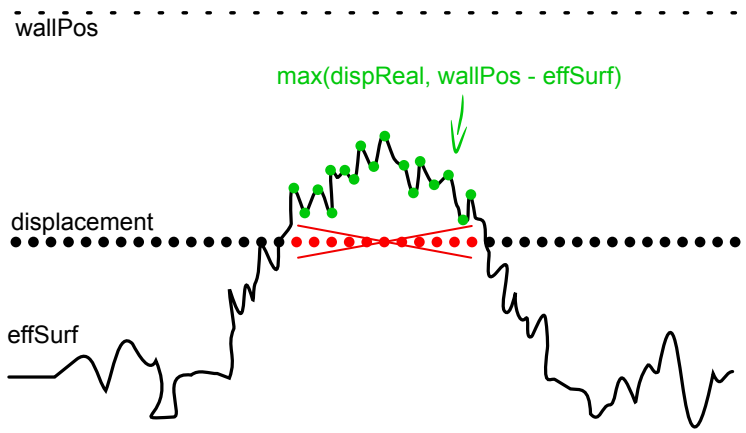


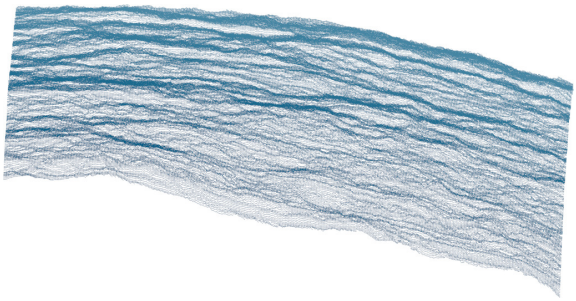
Figure 2: Boundary condition: the two surfaces are not allowed to overlap

3 Plastic Deformation

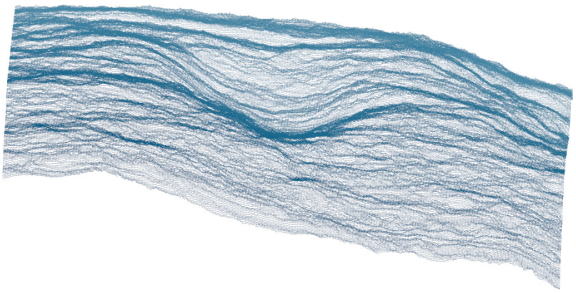
The ultimate goal is to develop an empirical method to calculate plastic deformation using the GFMD approach and to apply it to data obtained from real surfaces. In principle all of the four suggested approaches are based on a subtle change in the boundary condition. While keeping the boundary condition for the displacement field as described in section 2.3, we assume the reference points of `wallPos` no longer to be fixed. This effectively corresponds to deforming the effective surface, where raising the `wallPos` is equivalent to a dent in the effective surface. Consequently this can be related to the plastic deformation of the real substrate if we assume the hardness of the indenter to be sufficiently big. The experimental data provided to us by the group of Dr. Martin Dienwiebel consisted of scans of a copper OFHC surface of dimensions 650 by 650 pixels, where one pixel corresponds to $3.698 \cdot 10^{-7}\, m$, bringing the total size to a square of sidelength $240.41\ \mu m$ (see Fig. 3). Additionally we got scans of the rubin indenter which was of the same dimension. Then the indenter was pushed on the surface with a certain force and subsequently removed to allow for scanning of the deformed surface. Afterwards the indenter was restored to its position and pressed down with a higher pressure. The procedure was repeated for forces of $0.8\ N$, $1.6\ N$, $3.2\ N$, $6.4\ N$ and $12.8\ N$ (see Fig. 3 for some characteristic examples).

3.1 Fixed Local Change

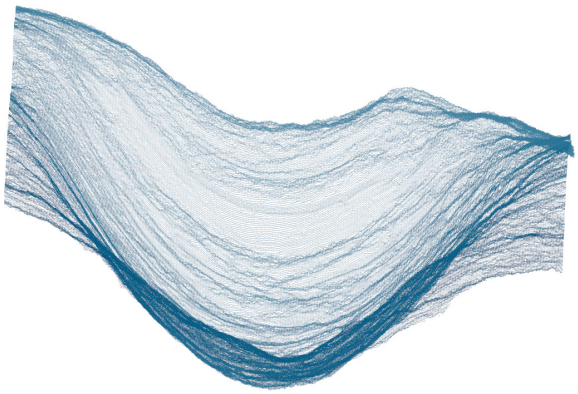
As a first attempt to include plastic deformation we used a fairly simple and straightforward approach. After each MD timestep (see sec. 2.2) the forces are transformed from Fourier to real space. Then the pressure on each point is evaluated and compared to a threshold value. For our purpose we chose the Vicker hardness of the experimental surface which was OFHC Copper. If the pressure on a point exceeded the threshold, its position would be plastically deformed. To find a reasonable length scale by which to distort the corresponding `wallPos`, we deemed a fraction of the absolute height of the original, undeformed substrate a good starting point. It was necessary to optimize this proportionality factor in order to reproduce the experimental data. If the value was chosen too small, the effect of plastic deformation would be negligible and also the system would take a very long time to converge. On the other hand choosing the proportionality factor too big led to an even graver problem. In this case the points would tend to overshoot, being pushed away from the contact area too far, thus losing contact and resulting in false 'holes' in the substrate (see Fig. 4). This issue will be further addressed in section 3.4. First thing to note about this method is that the system was found to be very sensitive to this parameter. If the surface is rough and contains some high asperities, the local force acting on them can be considerably big and will lead to overshooting. This is relevant especially at the beginning of the simulation when only a few points are in contact. One solution may be to choose the value depending on time or contact area, i. e. smaller in the beginning with few contact points and bigger after. But this would be on the one hand quite complex to Figure out and on the other be only slightly motivated by actual physics. Furthermore we find that any optimization of this value is hardly transferable to other values of pressure. It is quite intuitive that a higher pressure would lead to a bigger indentation, as can clearly be observed in the experimental data (see Fig. 3). Since this approach did not take into account the magnitude of the pressure, an optimization of parameters for one pressure will yield too small deformation for higher pressures and overshoot for smaller pressures. In the next section we will take this insight into account with a slightly more sophisticated method.



(a) Original undeformed copper surface



(b) Deformation with parabolic rubin indenter, force 1.6N



(c) Deformation with parabolic rubin indenter, force 12.8N

Figure 3: Experimental data

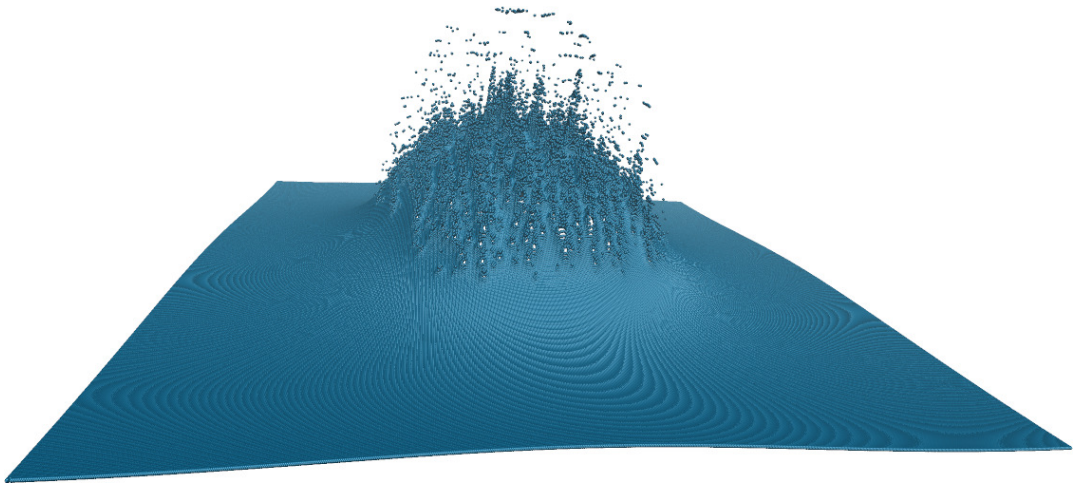


Figure 4: Application of the local approach to the experimental data. Note the detached points

3.2 Nearest Neighbours

3.2.1 Method

In principle the next approach is quite similar to the previous one except for two important refinements. For one the change in `wallPos` will be pressure-dependent and secondly an attempt at preventing overshooting is implemented.
Intuitively from everyday experience and supported by the experimental data (Fig. 3) it is to be expected that greater pressure on a surface will lead to a bigger deformation. As before, we pick all points on which the pressure exceeds a certain threshold, but here we will change their respective `wallPos` proportionally to the pressure they are exposed to. To be precise, it will be a certain fraction of the amount by which the pressure exceeds the threshold. As before the system will be quite sensitive to this proportionality factor.
Also we may still encounter the problem of overshooting. To account for this, we introduce the additional condition that in any given timestep a point may not be lowered further than its nearest neighbours. Again being more precise it will be moved proportionally to pressure but at most to the mean height of its nearest neighbours. This will on the one hand diminish the problem of points losing contact, while at the same time ensuring that it is still always possible for points to move. The reason for choosing the mean may not be obvious at first glance. Consider the condition that a point may not be moved further than the highest nearest neighbour. This would not merely diminish the risk of points losing contact but get rid of the problem altogether by being a much more conservative estimate. But it would introduce a new problem where points forming a plateau with their surrounding neighbours can not be moved further even though the pressure might still exceed the threshold. This necessitates a compromise hence the choice of mean height of nearest neighbours.
Although these measures take care of the most basic issues in realizing the plastic deformation, the implementation revealed a new serious problem. When changing the `wallPos` the displacement field is changed accordingly. Since the change is only local on each point separately, discontinuities may appear in these fields. If subsequently the displacement field is transformed to Fourier space as necessary for the MD loop (see sec. 2.2), these discontinuities will introduce what is know as ringing artifacts. [4] Due to the implementation peculiarities of the Fast Fourier transform the edges of any step function will be subject to overshoot and oscillations. A quite pronounced illustration of this problem can be observed in Fig. 5. It shows the change in `wallPos` and therefore the plastic deformation of an ideal parabolic indenter. Obviously it is not the expected response of the substrate, because the ringing artifact completely obscures the actual parabolic deformation.
To alleviate this issue we attempted to smoothen discontinuities in the displacement field. Whenever the height difference between two neighbouring points was found to be too steep, the lower point was lifted by half the distance. When this did not improve the ringing artifacts sufficiently we looped over the condition several times until all steps were smaller than a certain threshold. Unfortunately it was not possible to find a threshold which was small enough to prevent ringing while at the same time ensuring that the surface was not distorted too much.

3.2.2 Some remarks on parallel implementation

Although the code was already MPI-parallelized, accessing the height of the nearest neighbours required some additional communication, so the topic will be briefly addressed at this point.

In principle we use a two-dimensional array for storing the topography of our surface and displacement fields, where the index of the array is the x- and y-coordinate of the points respectively. Using the fftw-library entails that this array will be partitioned in slices along the x-direction, rather than the more common boxes. Since we apply periodic boundary conditions this means that in y-direction we encounter no problems in accessing information about nearest neighbours to the left and right (see Fig. 6). However the communication becomes necessary when the local topmost row on a processor tries to access its top neighbour or the bottommost its bottom neighbour respectively. To deal with this issue we introduce two arrays of 'virtual neighbours'. Each processor sends an array containing the height of all points in the top row to the virtual bottom array of the previous processor as well as the information about all points in the bottom row to the virtual top array of the next processor. Again we suppose periodic boundaries. The arrays are kept updated automatically, because each point of the top or bottom row will update the corresponding information on its own processor whenever it is modified. Then after each MD timestep the whole arrays will be sent to the neighbouring processors to ensure that the information about nearest neighbours is ever up to date.

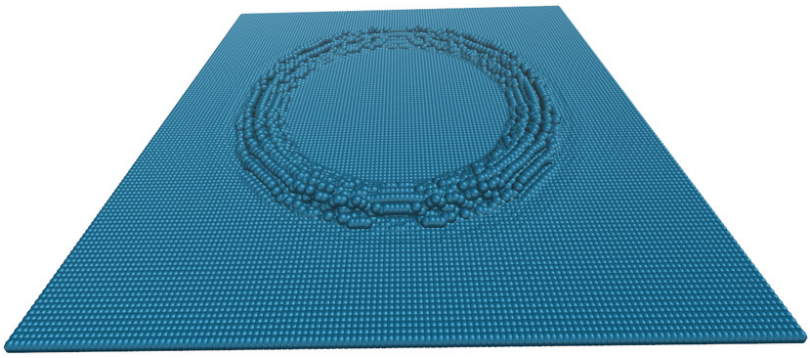


Figure 5: Ringing artifacts in the `wallPos` for an ideal parabolic indenter

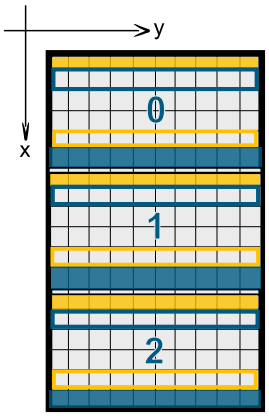


Figure 6: Division of the fields on several processors. Framed fields are not able to access their top/bottom neighbours. Coloured fields indicate arrays of virtual neighbours.

3.3 Fourier Space method

In an attempt to circumvent the issue of ringing artifacts as described in the previous section, we tested an approach avoiding unnecessary Fourier transformations.
The main idea is to eliminate the need for transforming the `wallPos` array from real to Fourier space and the subsequent ringing artifacts from its discontinuities by introducing a `wallPos` already in Fourier space. This field would then be modified in reciprocal space and the boundary conditions applied to the displacement field before transforming the latter back to real space and check for termination (see sec. 2.2). Although this seems like an elegant way to avoid the ringing artifacts altogether,

this approach turned out to be more difficult than this short outline reveals.
The first problem to present itself was finding a reasonable threshold criterion. Operating in Fourier space implies the use of complex numbers. It is necessary to interpret them correctly and the threshold criterion we used before in sections 3.1 and 3.2 can no longer be applied. As we now compare complex numbers, it is not as straightforward to tell when the pressure in Fourier space exceeds a maximum permissible value. So we would need to identify how the local force on a point in real space is expressed in Fourier space and what would then be a reasonable criterion to decide whether and how to change the boundary condition or its prerequisites.
The same underlying issue also sprouts difficulties when applied in reverse direction. Modifying the displacement field on a certain frequency translates to changing several points on the real surface. This is the reason why we cannot assume a simple response as before. Assume we solve the problem of finding an appropriate threshold criterion, which may be frequency-dependent or otherwise modified so that it holds for all frequencies in our domain. Furthermore assume the procedure to be implemented analogous to the real space approach. If the pressure on one frequency exceeds the threshold, the `wallPos` for this particular frequency will be modified. This implies changing the position of several points all over the surface in real space. One may argue that this could reflect some long range interactions but let us consider the case of some high asperities in contact. In that case the deformation is expected to occur locally only on these points and would decrease the height of these spikes while a response of a totally different location on the surface would not only be counterintuitive but most probably also wrong. Still this is what would be implied, were we to change the `wallPos` only locally on one frequency as proposed until now.
Nevertheless it is not impossible to find a function to change the `wallPos` accordingly in Fourier space in order to obtain a local response in real space restricted to only a certain area to be deformed. This response function can be expected to be quite complex and most certainly requires change in `wallPos` over a range of frequencies. This in turn will probably necessitate communication over different processors which may slow down the parallelized code considerably.
In view of these of yet unresolved question and due to the end of the program drawing nearer, we abandoned this method for the time being in favour of another promising approach which will be described in the next section. Nevertheless, further investigations in this direction may prove worthwhile using our preliminary findings and are necessary to ultimately ascertain its extent of utility.

3.4 Bidirectional Local Change

The final approach at embedding plasticity revisits the idea of changing the `wallPos` in real space again. However with a more successful take on the problem of overshooting.
This method is a combination of the ones introduced in sections 3.1 and 3.2. As before, the change in `wallPos` is performed in real space and only locally on those points on which the pressure exceeds the hardness. Again the amount of change is proportional to the difference of hardness threshold and pressure. The important difference is that all points which have been deformed will be revisited. Now each point will be examined if either the local pressure on it exceeds the threshold and modified accordingly or if it had already been deformed once, there are two possibilities. If the point has not been moved far enough, the pressure will still exceed the threshold and the point will be moved further. If the point has been moved too far, then the pressure will be smaller than the hardness and the point has to be moved back. For this purpose we look again at the difference between pressure and hardness which now will be of the opposite sign and modify the point's position proportionally. This implies

that a particle which overshot by a big amount will be moved further back than one which only differs by a small amount. Effectively we hope to ensure finding an equilibrium as fast as possible, where every deformed point comes to rest in a position where the pressure approximates the hardness, i. e. at the threshold for plastic deformation.
We found that for small system sizes it was possible to find parameters which look very promising and meet the expectations well, as can be seen in Fig. 7 for a ideal parabolic indenter. Still since the parameters are extremely sensible to system size and not in an obvious straightforward dependency it will take some further investigations to enable this method to being applied generally to other system sizes and to the experimental data.
At the moment this last method seems to yield the most promising results given that a matching set of parameters can be found.

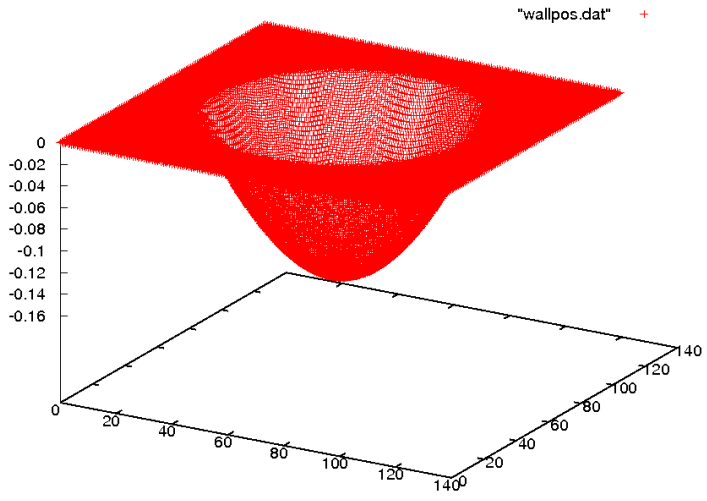


Figure 7: Plastic deformation for a small parabolic indenter (128 x 128 grid points, height in arbitrary units)

4 Conclusion

In conclusion, we tried four different approaches to embedding plasticity into a parallel Green's function molecular dynamics code. We described their properties and discovered their potential drawbacks. The first two suggestions of fixed height local deformation and pressure-dependent deformation with respect to nearest neighbours suffered from serious issues, respectively overshoot and ringing artifacts. The Fourier space method needs some further investigation before we are able to draw definite conclusions of its utility.
Meanwhile the last approach of changing the deformed points until they reach a point of equilibrium at the threshold pressure yields the most promising results but needs some refinement to ensure generalisation.

5 Acknowledgements

I would like to thank Prof. Dr. Martin Müser and Dr. Mykola Prodanov for supervising my project, the experimental group of Dr. Martin Dienwiebel for providing the experimental data, Mathias Winkel and Ivo Kabadshow for organizing the program and most of all my fellow gueststudents for moral and hands-on support and a lot of fun.

References

1. B.N.J. Persson *Contact mechanics for randomly rough surfaces*. Surface Science Reports 61; 2006: 201-2
2. J.A. Greenwood and J.B.P. Williamson, *Contact of Nominally Flat Surfaces*. Proc. Roy. Soc. London, Ser. A 295; 1966: 300
3. Carlos Campañá and Martin H. Müser, *Practical Green's function approach to the simulation of elastic semi-infinite solids*. PHYSICAL REVIEW B 74; 2006: 075420
4. George Em Karniadakis and Robert M. Kirby II, *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press; 2003

Multigrid, conjugate gradient solver for Reynolds thin film equation

Sebastian Rupprecht

Universität Augsburg
86135 Augsburg
Germany

E-mail: sebastian.rupprecht@outlook.com

Abstract:
In this report, we study the flow between two solids with rough surfaces, as described by Reynolds thin film equation. After deriving the equation's weak formulation by variational calculus, we calculate the current with a conjugate gradient method. To obtain better runtime performances we implement sparse matrices and compare various iterative solvers and preconditioners. We show that the conjugate gradient method is clearly superior to a local solver. This holds for both approaches to model the contact area between the two surfaces, the traditional cutoff model and the elastic model.

1 Introduction

Failure of seals have caused some dramatic catastrophes in the last 30 years, for example the Challenger disaster and the explosion and sinking of the oil platform 'Deepwater Horizon'. While in 2010 a seal failed to restrain gas from the borehole which finally was one of the reasons for the oil spill [1], in 1986 a seal in one of Challengers's rocket boosters failed so that emanating gas could reach the fuel tank which caused the explosion [2]. But seal failures can also have consequences in our daily life, like energy loss or environmental pollution. Therefore, the design of seals should play a more important role in research than in the last years. Because experiments are expensive and time-consuming, we want to simulate the performance of seals to gain a better understanding of them. The fluid flow between two surfaces can be approximated by a partial differential equation (PDE), Reynolds thin film equation, in two dimensions. After discretising the PDE with Finite Differences to obtain a linear system, we store the resulting system with sparse matrices. Since the interface and the contact area between seal and substrate largely depend on the surface roughness of the latter, we have to deal with the gap topography [3]. Following an introduction into the theory we consider various iterative methods to solve the linear system and give a brief summary of different preconditioners. Finally, we compare our results with those of the local method which was used in previous work [3].

2 Reynolds thin film equation

In this work we use Reynolds thin film equation to model the thin film lubricant flow in 2D between a rigid and a deformable surface with fractal contact. In [4] it is shown that it is reasonable to add all the roughness to the rigid surface which leads to the model in Fig. 1.

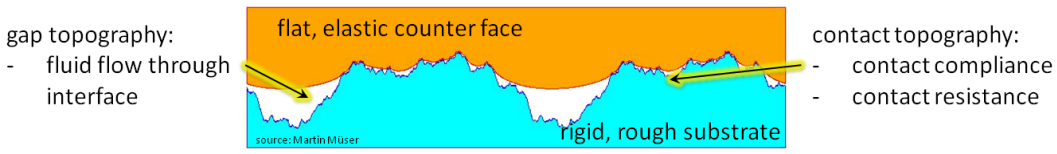


Figure 1: elastic contact between rough surfaces

2.1 Derivation of Reynolds thin film equation

Reynolds thin film equation can be derived by the *Continuity Equation* and *Navier-Stokes Equations* integrated over the third dimension like in [5] which leads us to

$$\nabla \cdot \mathrm{J} = \nabla \cdot \left(\underbrace{\frac{u(x,y)^3}{12\eta}}_{\sigma(x,y)} \cdot \nabla \mathrm{p}(x,y) \right) = 0 \qquad \text{in } \Omega, \tag{1}$$

where the domain Ω is the unit square $(0,1) \times (0,1)$, J the current within the domain Ω, p the pressure, $u(x,y)$ the gap height between the two surfaces at position (x,y), and η the viscosity. In the following, we will aggregate the term $\frac{u(x,y)^3}{12\eta}$ into the conductivity $\sigma(x,y)$, which consequently scales with the third power of the gap height and inversely with viscosity. The fluid flow will decrease with growing viscosity η which is very intuitive since viscosity is the measure of a fluid's internal resistance to flow. From the derivation of Reynolds thin film equation by the *Continuity Equation* for incompressible fluids we know that the current does not have sources or sinks within the domain so that the divergence of J has to be zero. We can see from Eqn. (1) that we need a pressure difference to obtain a current. Setting the pressure to $\mathrm{p} = 1$ on $x = 0$ and $\mathrm{p} = 0$ on $x = 1$ gives us a flow in positive x-direction. To simulate a large system in y-direction we use periodic boundary conditions for $y = 0$ and $y = 1$. This gives us the following boundary conditions in addition to Reynolds thin film equation:

$$\begin{aligned}
\nabla \cdot \mathrm{J} = \nabla \left[\sigma(x,y) \cdot \nabla \mathrm{p}(x,y) \right] &= 0 && \text{in} && (0,1) \times (0,1) && \\
\mathrm{p} &= 1 && \text{on} && \{0\} \times [0,1) && (2) \\
\mathrm{p} &= 0 && \text{on} && \{1\} \times [0,1) && (3) \\
\mathrm{p}(x,0) &= \mathrm{p}(x,1) && \text{on} && (0,1) \times \{0,1\}. && (4)
\end{aligned}$$

A rendering of the setup is shown in Fig. 2.

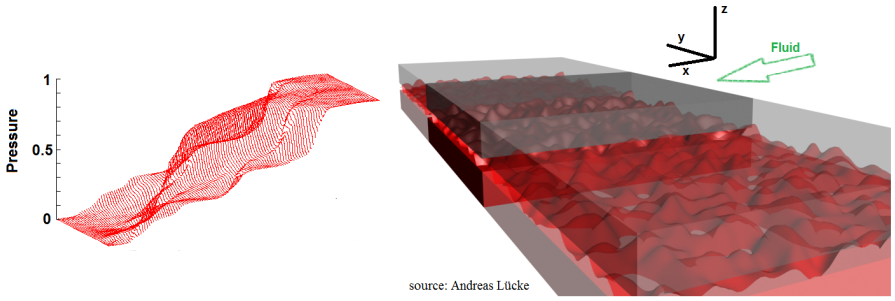


Figure 2: visualisation of the setup with boundary conditions

Expanding Eqn. (1) leads to the following equivalent equation

$$\triangle \mathrm{p} = -\frac{12\eta}{u(x,y)^3}\left(\nabla\frac{u(x,y)^3}{12\eta}, \nabla \mathrm{p}\right), \tag{5}$$

where the round brackets denote an inner product. We can see that the right-hand-side depends on the solution p such that we cannot use most of the solvers for the standard Poisson equation anymore but we will need to derive a modified solver using finite differences that can be used for the generalized Poisson Eqn. (1).

2.2 Derivation of variational form

In the following we want to derive the weak form of Reynolds thin film Eqn. (1) in a similar way like it is done in [6]. To deduce a more accurate method to solve Eqn. (1) than directly applying numerical derivatives, we should notice that $\sigma(x,y)$ does not have to be sampled at identical grid points as $\mathrm{p}(x,y)$. We define the conductivities as sectionally constant within the regions of 4 neighbouring grid points like in Fig. 3.

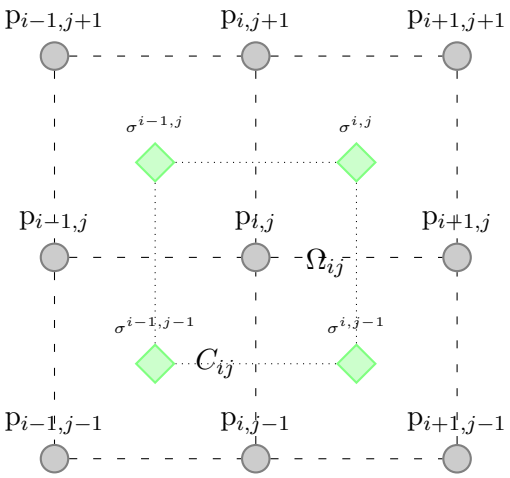


Figure 3: uniform finite difference mesh for Reynolds thin film equation

With $\mathrm{p}_{i,j} \equiv \mathrm{p}(x_i, y_j)$ being defined along the original grid points on a uniform mesh, we obtain

$$\sigma_{i,j} \equiv \sigma(i,j) = \sigma\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}\right), \tag{6}$$

where $h = \frac{1}{\mathcal{L}+1}$ is the constant grid length in our equidistant mesh of $\mathcal{L} \times \mathcal{L}$ grid points. This approach allows us to express Eqn. (1) in its *weak form* by using variational calculus. So we can remove all the 2^{nd}-order derivatives to leave only the 1^{st}-order derivatives which we then will approximate numerically.

We define Ω_{ij} as the square region around a single pressure sample point $\mathrm{p}_{i,j}$ as shown in Fig. 3. Since Reynolds thin film equation shall be fulfilled in the whole domain Ω, we take the surface integral over Ω_{ij} and obtain

$$\iint_{\Omega_{ij}} \nabla \cdot \left[\sigma(x,y) \cdot \nabla \mathrm{p}(x,y)\right] \mathrm{d}\Omega = 0, \tag{7}$$

where $d\Omega = \mathrm{d}x\mathrm{d}y$ is the differential surface area. By applying the divergence theorem we can convert the surface integral over Ω_{ij} into a contour integral around the enclosing boundary C_{ij} and obtain the weak form of Eqn. (1):

$$\iint_{\Omega_{ij}} \nabla \left[\sigma(x,y) \cdot \nabla \mathrm{p}(x,y)\right] \mathrm{d}x\mathrm{d}y = \oint_{C_{ij}} \sigma(x,y) \cdot \nabla \mathrm{p}(x,y)\, \mathrm{dn} = 0, \tag{8}$$

where $d\mathrm{n}$ is the differential unit normal vector. Since we are working in two dimensions, C_{ij} is a square contour which can be split into four sub-integrals around each side of the square called S_1, S_2, S_3 and S_4:

$$\oint_{C_{ij}} \sigma(x,y) \left[\frac{\partial}{\partial x}\mathrm{p}(x,y)\hat{\mathrm{x}} + \frac{\partial}{\partial y}\mathrm{p}(x,y)\hat{\mathrm{y}}\right] \mathrm{dn} = \int_{S_1} + \int_{S_2} + \int_{S_3} + \int_{S_4}. \tag{9}$$

In the following we want to simplify the four sub-integrals. Let us start with the integral along the right edge of the enclosing contour S_1 with $\mathrm{dn} = \hat{\mathrm{x}}\mathrm{d}y$:

$$\int_{S_1} = \int_{-h/2}^{h/2} \sigma(x,y) \left[\frac{\partial}{\partial x}\mathrm{p}(x,y)\hat{\mathrm{x}} + \frac{\partial}{\partial y}\mathrm{p}(x,y)\hat{\mathrm{y}}\right] \overbrace{\hat{\mathrm{x}}\,\mathrm{d}y}^{\mathrm{dn}} = \int_{-h/2}^{h/2} \sigma(x,y) \frac{\partial}{\partial x}\mathrm{p}(x,y)\mathrm{d}y. \tag{10}$$

Since S_1 is the entire border between the regions Ω_{ij} and $\Omega_{i+1,j}$ which are defined by $\mathrm{p}_{i,j}$ and its right neighbour $\mathrm{p}_{i+1,j}$, we approximate the partial derivative by a difference quotient between the two samples. We also assume the conductivity $\sigma(x,y)$ to be constant along the entire border with the value given by the arithmetic average of both conductivities limiting the right boundary. Calculating the integral gives us

$$\int_{S_1} \approx h \left[\frac{\sigma_{i,j} + \sigma_{i,j-1}}{2}\right] \left[\frac{\mathrm{p}_{i+1,j} - \mathrm{p}_{i,j}}{h}\right] = \frac{1}{2}\left[\sigma_{i,j} + \sigma_{i,j-1}\right]\left[\mathrm{p}_{i+1,j} - \mathrm{p}_{i,j}\right]. \tag{11}$$

The analogous calculations on the other three borders result in

$$\int_{S_2} \approx \frac{1}{2}\left[\sigma_{i-1,j} + \sigma_{i,j}\right]\left[\mathrm{p}_{i,j+1} - \mathrm{p}_{i,j}\right], \tag{12}$$

$$\int_{S_3} \approx \frac{1}{2}\left[\sigma_{i-1,j-1} + \sigma_{i-1,j}\right]\left[\mathrm{p}_{i-1,j} - \mathrm{p}_{i,j}\right], \tag{13}$$

$$\int_{S_4} \approx \frac{1}{2}\left[\sigma_{i,j-1} + \sigma_{i-1,j-1}\right]\left[\mathrm{p}_{i,j-1} - \mathrm{p}_{i,j}\right]. \tag{14}$$

With the following notation

$$a_0 = \sigma_{i,j} + \sigma_{i-1,j} + \sigma_{i,j-1} + \sigma_{i-1,j-1}, \tag{15}$$

$$a_1 = \frac{1}{2}\left[\sigma_{i,j} + \sigma_{i,j-1}\right], \tag{16}$$

$$a_2 = \frac{1}{2}\left[\sigma_{i-1,j} + \sigma_{i,j}\right], \tag{17}$$

$$a_3 = \frac{1}{2}\left[\sigma_{i-1,j-1} + \sigma_{i-1,j}\right], \tag{18}$$

$$a_4 = \frac{1}{2}\left[\sigma_{i,j-1} + \sigma_{i-1,j-1}\right], \tag{19}$$

we can express the contour integral in Eqn. (9) as

$$\oint_{C_{ij}} \approx -a_0\mathrm{p}_{i,j} + a_1\mathrm{p}_{i+1,j} + a_2\mathrm{p}_{i,j+1} + a_3\mathrm{p}_{i-1,j} + a_4\mathrm{p}_{i,j-1} = 0. \tag{20}$$

Equation (20) represents a five-point stencil for Reynolds thin film Eqn. (1). From here it is straightforward to generate a linear system which has to be solved.

3 Solution of Linear System

With the theoretical background from Section 2.2 it is possible to generate a system of linear equations which we want to solve. But before we look onto different iterative solvers for this linear system, we need to store our system using sparse matrices.

3.1 Sparse Matrices

The system matrix we obtain with the five-point stencil approach for Reynolds thin film equation is very large for large grid sizes. With a grid of size $\mathcal{L} \times \mathcal{L}$ (including both fixed left and right boundary) the system matrix A has dimensions $(\mathcal{L}^2 - 2\mathcal{L})$-by-$(\mathcal{L}^2 - 2\mathcal{L})$. But in each row of A we have only at most 5 non-zero entries, namely $a_0, a_1, ..., a_4$ which can be also zero in areas with no conductivity. Notice that the $2\mathcal{L}$ rows associated with the grid points neighbouring the left and right boundary have at most 4 non-zero entries since the left respectively right neighbour is fixed and therefore implemented

in the right-hand-side vector b of the linear system. To avoid finite size effects we want to simulate the current in a grid of size 4096×4096. Thus in this case at most approximately 8.4×10^7 ($5\mathcal{L}^2 - 12\mathcal{L}$) out of 2.8×10^{14} elements are non-zero. To save memory and especially to accelerate matrix-vector multiplications in our calculations we implement sparse matrices in C++ with the help of the library *SparseLib++*. In the following we describe the *compressed column storage* format which stores all non-zero values in each column and their associated row indices in the arrays `val()` respectively `row()`. Furthermore, pointers to the first element in each column have to be stored in the array `col()` to ensure a unique assignment [7]. We want to illustrate the procedure with the following matrix which represents an extract of our system matrix (without dependency on upper and lower neighbours a_2 and a_4 which are stored in a far away off-diagonal):

$$\begin{pmatrix} a_0^1 & -a_1^1 & 0 & \dots & \dots & 0 \\ -a_3^2 & a_0^2 & -a_1^2 & 0 & & \vdots \\ 0 & -a_3^3 & a_0^3 & -a_1^3 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & -a_3^{N-3} & a_0^{N-3} & -a_1^{N-3} \\ 0 & \dots & \dots & 0 & -a_3^{N-2} & a_0^{N-2} \end{pmatrix}$$

Keeping in mind that C++ is based on 0-based indexing, this matrix can be stored with the help of the following arrays:

$$\begin{aligned} \texttt{val} &= [a_0^1, -a_3^2, -a_1^1, a_0^2, -a_3^3, ..., -a_1^{\mathcal{L}-3}, a_0^{\mathcal{L}-2}], \\ \texttt{row} &= [0, 1, 0, 1, 2, 1, 2, ...], \\ \texttt{col} &= [2, 5, 8, ...]. \end{aligned}$$

Notice that for symmetric matrices (which, in general, our system matrix is not) it would be sufficient to store only either the upper or the lower triangular portion of the matrix.

3.2 Method of Conjugate Gradient

We now want to solve a linear system of the form

$$A\, x = b. \tag{21}$$

Let us assume that $A \in \mathbb{R}^{n \times n}$ is a symmetric, positive-definite matrix and $b \in \mathbb{R}^n$. We can reformulate Eqn. (21) as a minimisation problem. So the solution of Eqn. (21) can be identified with the solution of

$$\min_{x \in \mathbb{R}^n} f(x) = \frac{1}{2}\,(Ax, x) - (b, x)\,. \tag{22}$$

We know that $x \in \mathbb{R}^n$ is solution of Eqn. (21) if and only if it solves the minimisation problem Eqn. (22).
For the solution we deal with the CG method which is an iterative method and can – in contrast to direct methods – handle large systems. It generates a sequence of conjugate vectors which are the gradients of the quadratic functional Eqn. (22). So the idea of conjugate gradients is:

1. Minimise $f(x)$ in the k^{th} iteration step in $span\{d^0, d^1, d^2, ..., d^{k-1}\}$ where d^i denotes the minimising direction in the i^{th} step.
2. Vectors d^k shall be linear independent. It follows that $span\{d^0, d^1, d^2, ..., d^{n-1}\} = \mathbb{R}^n$.

Consequently the CG method converges for polynomials of order 2 in exact arithmetic after at most n iterations where n is the number of degrees of freedom. This result is rather theoretical and not directly useful for us since we are only calculating with machine accuracy. On the other hand, the convergence rate plays an important role because we are interested in an approximation in reasonable time rather than an exact solution. Therefore, we hope that the algorithm converges in significantly fewer steps than the number of degrees of freedom of our system $(\mathcal{L}^2 - 2\mathcal{L})^2$.
The CG method is extremely effective since in every iteration only one matrix-vector product is needed which costs not more than $\mathcal{O}(n)$ for sparse matrices. The problem is that it only works with symmetric, positive-definite system matrices, otherwise it will diverge in general. Consequently, we will have a look to more general iterative solvers in the next section since we have seen that our system matrix is normally not symmetric.

3.3 Alternative iterative solvers

We have seen in the previous sections that the system matrix that we obtained by discretising Eqn. (1) is generally non-symmetric and so the CG method is not applicable. Thus we will deal with three non-stationary methods which are all based on the CG method and can handle with non-symmetric coefficient matrices:

- BiConjugate Gradient (BiCG)
  The BiConjugate Gradient method generates sequences of minimising vectors like the CG method for both the original system matrix A and its transpose A^T. Not each of those sequences is orthogonalized, but they are made 'bi-orthogonal', i.e. mutually orthogonal. In each iteration the BiCG requires a matrix-vector multiplication with the system matrix and its transpose. The two matrix-vector products are independent such that they can be done in parallel. Nevertheless, the convergence of the BiCG method may be irregular and it is possible that the algorithm breaks down [8].
- Conjugate Gradient Squared (CGS)
  The Conjugate Gradient Squared method is a variant of the BiConjugate Gradient method which does not need the matrix-vector multiplications with A^T because it applies the updating operations for both the A- and A^T-sequences to the same vectors. Although the transpose of the coefficient matrix is not required, the computational costs per iteration are similar to the previous method. However, the two matrix-vector products are not independent such that the number of synchronisation points in a parallel environment has to be larger. Although the CGS method converges typically about twice as fast as the BiCG method, the convergence behaviour is in practice much more irregular. The method tends to diverge if the starting guess is close to the solution [8].
- BiConjugate Gradient Stabilized (Bi-CGSTAB)

The BiConjugate Gradient Stabilized method is also based on BiCG but uses different updates for the A^T-sequence. The computational costs per iteration are similar to BiCG and CGS but the transpose matrix is not required. The Bi-CGSTAB method avoids the irregular convergence patterns which can be observed in the CGS method while maintaining the same speed of convergence. With the smoother convergence less accuracy of the updated residual is lost [8].

In the following we want to use Bi-CGSTAB for our simulations because it combines both advantages of BiCG and CGS: a high convergence rate and smooth convergence.

3.4 Preconditioners

We know that the convergence rate of iterative methods depends to a large extent on the condition number of the system matrix. Now, we want to transform Eqn. (21) into a linear system with the same solution but better spectral properties, i.e. the condition number of the latter is closer to 1 than $\mathtt{cond}_2(A)$ is. Hence, our goal is to find a preconditioning matrix M that approximates the system matrix A such that the transformed system

$$M^{-1}Ax = M^{-1}b \tag{23}$$

has the same solution as the original system and the spectral properties of $M^{-1}A$ are more favourable ($\mathtt{cond}_2(M^{-1}A) < \mathtt{cond}_2(A)$).
Note that there is also a class of preconditioning matrices M that approximate A^{-1} so that only multiplication of A by M is needed. Since it is hard to find those preconditioning matrices, we will concentrate on the first idea of approximating A. In order to work with the most efficient preconditioner we want to compare the following three:

1. *Jacobi Preconditioning*
   The Jacobi preconditioner is the simplest of those three because it only consists of the diagonal of the system matrix A:
   $$m_{i,j} = \begin{cases} a_{i,j} & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$
   For using Jacobi preconditioner we do not need extra storage since we can access the elements in the system matrix instead of storing another vector [8].

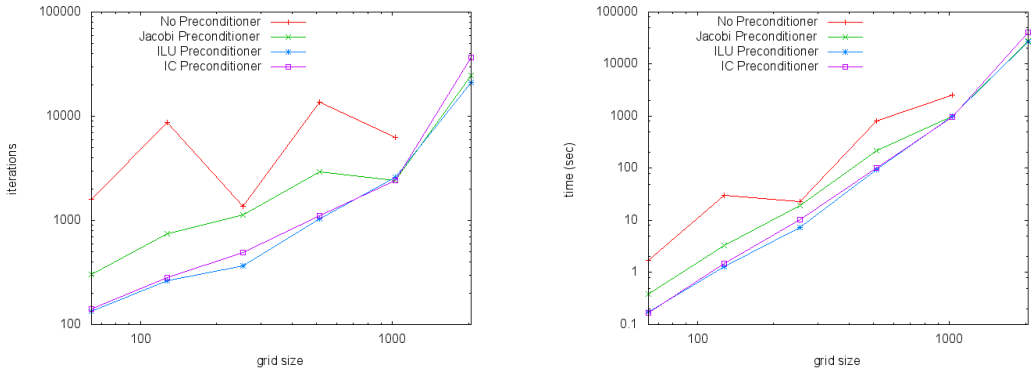
2. *Incomplete LU (ILU) Preconditioning*[1]
   Incomplete LU factorisation gives us the ILU preconditioner in the factored form $M = LU \approx A$ with L lower and U upper non-singular triangular matrix. Since the matrix $(LU)^{-1}A$ is closer to the unit matrix, the iterative method converges faster. Furthermore the calculation of $y = Ax$ with sparse A needs few computing time and the solution of $c = (LU)^{-1}y$ can be calculated efficiently by solving the equivalent linear system $(LU)c = y$ by forward-backward-substitution and making use of the sparse storage of L and U [8].

3. *Incomplete Cholesky (IC) Preconditioning*
   For the case that we have a symmetric, positive-definite system matrix, we can use the IC pre-

[1] Factorisation is called incomplete if during the factorisation process non-zero elements in positions, where the original matrix had a zero, are ignored.

conditioner that is given in the form $M = LL^*$ where L is a lower triangular matrix and L^* its conjugate transpose. The IC preconditioner, which was derived from ILU factorisation, is nearly twice as efficient as the ILU preconditioner if it is applicable [8].

To avoid breakdowns of the Jacobi and incomplete preconditioners because of zero elements, we use a safety query in the algorithm which ensures that all conductivities smaller than 10^{-40} are set to 10^{-40}. We have to consider that preconditioners cause extra costs because of the initial setup and its application in each iteration. Consequently, we need a certain trade-off between the cost of constructing respectively applying the preconditioner and the gain in convergence speed.

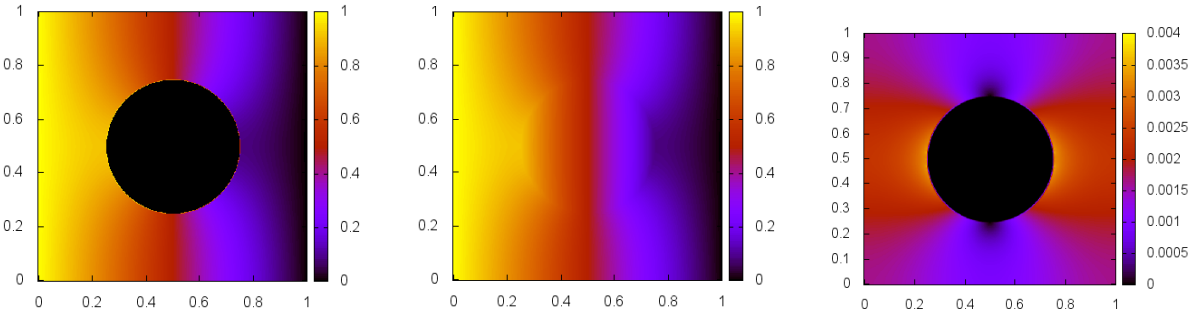


(a) iterations (b) runtime

Figure 4: comparison of no preconditioning, Jacobi preconditioning, ILU preconditioning and IC preconditioning for 15% relative contact area (elastic model); Bi-CGSTAB did not converge for grid of size 2048×2048 without preconditioner

We can see in Fig. 4 that the incomplete factorisation preconditioners are superior to Jacobi preconditiong and especially to no preconditioning since they do not only need less iterations (what was expected) but also need less time to converge. For grids of size 2048×2048 the Bi-CGSTAB method without preconditioning does not converge in a reasonable time anymore. This shows how important preconditioning is for solving large systems with iterative solvers. We can also observe that the IC preconditioner has the same convergence rate like the ILU preconditioner until grids of size 1024×1024 even though this preconditioner only works with non-symmetric matrices as mentioned above. It is indicated by the values for grids of size 2048×2048 that for higher grid sizes the IC preconditioner is more time-consuming than ILU so that we will use the ILU preconditioner by default.

But using preconditioners can lead to a different solution of the linear system. Let us for example consider a domain where the conductivity has a circle defect, i.e. the conductivity is zero in a circle within the domain. Let the conductivity be constantly one in the rest of the domain for better illustrations. We can see in Fig. 5 that the obtained pressure when solving this system without preconditioning is zero within the 'defect' circle whereas the pressure using preconditioning with any of the three above mentioned preconditioners is smoothed in this circle area. Looking at the current $\mathrm{J} = \sigma(x, y)\nabla \mathrm{p}(x, y)$ reveals that preconditioning has no consequences on the obtained current in which we are actually interested in. Preconditioning does not result in a different current since in this case a circle structure can also be observed in that sense that the pressure along the circle boundary is constant to the surrounding area.

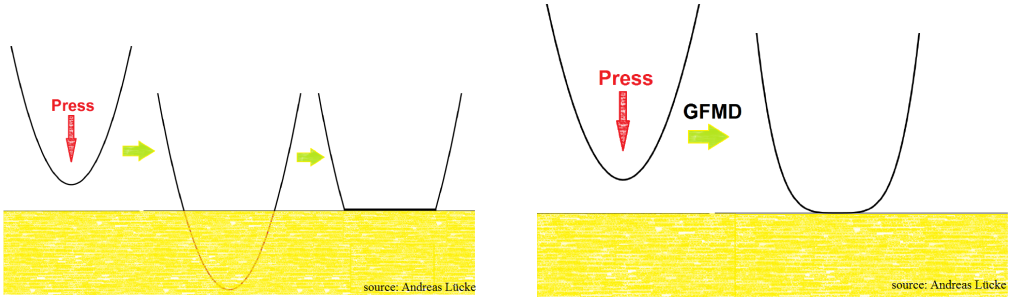


(a) pressure for no preconditioning (b) pressure for preconditioning (c) current for preconditioning/no preconditioning

Figure 5: effect of preconditioning on 512×512 grid with circle defect of conductivity

4 Simulations

We have seen in Fig. 1 that we add up all the roughness of both surfaces to the rigid one. The resulting contact when an external force presses the deformable against the rigid surface can be measured in two different ways [9]. By way of illustration we can consider in the case of contact the soft material to be the initially parabolic body like in Fig. 6. In the (conventional) *overlap model* all the overlapping components are cut and the positions of these points is set to the surface position. In the *elastic model* the soft material gets deformed and the displacement of a grid point has an influence on all the other grid points [9]. This model is more realistic since it describes the elastic response of materials better than the overlap model.



(a) overlap model (b) elastic model

Figure 6: comparison of mechanical contact models

We can see in Fig. 7 that the non-contact area is much more fractal in the elastic model and especially that the percolation threshold decreases from 50% (overlap model) to approximately 42% (elastic model) relative contact area [3].

Next, we want to have a look on how the Bi-CGSTAB method works on a system setup with 15% relative contact area on a grid of size 512×512 with a zero initial guess, i.e. zero pressure within the whole inner domain. We can see in Fig. 8 that in this case the algorithm works from the left to the right which is reasonable since it detects the jump from $\mathrm{p} = 1$ on the left boundary to $\mathrm{p} = 0$ for all the right neighbours and this information spreads in each iteration.

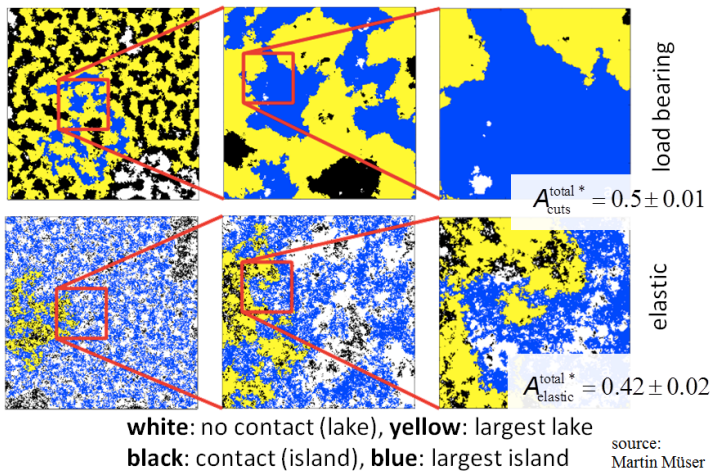


Figure 7: comparison of contact area for overlap (upper) and elastic (lower pictures) model at 46% contact

We now want to study the influence of the initial guess on the runtime performance of our Bi-CGSTAB method. For this, we compare the zero initial guess with a mean-field pressure distribution initial guess, i.e. a starting vector which describes the pressure decreasing linearly from the left ($p \equiv 1$) to the right boundary ($p \equiv 0$).
We can see in Fig. 9 that Bi-CGSTAB with a mean-field initial guess only needs 300 instead of 500 iteration steps compared to Bi-CGSTAB with zero initial guess. Furthermore, we can observe that Bi-CGSTAB needs far fewer iterations than the local solver on a grid of size 512×512 with 15% relative contact area for the elastic model.

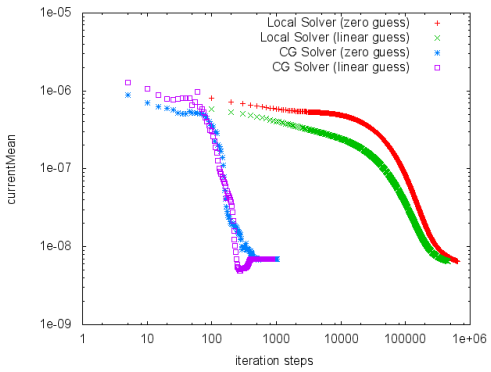


Figure 9: comparison of local and CG solver for zero and mean-field initial guess on 512×512 grid with 15% relative contact area (elastic model)

Because of the drastic runtime improvement with the better (mean-field) initial guess, we implement a multigrid method for a better performance for large grid sizes, i.e. we use a hierarchy of discretisations to obtain an optimal initial guess. For this, we average the given data to get a coarser grid, in our case we start with a grid of size 32×32. The solution is then interpolated and used as an initial guess for the next finer grid of size 64×64. Applying this procedure iteratively until arriving at the original grid, which should be square and the total number of grid points an even power-of-two, gives us an initial guess near the solution, which should shorten the runtime significantly.

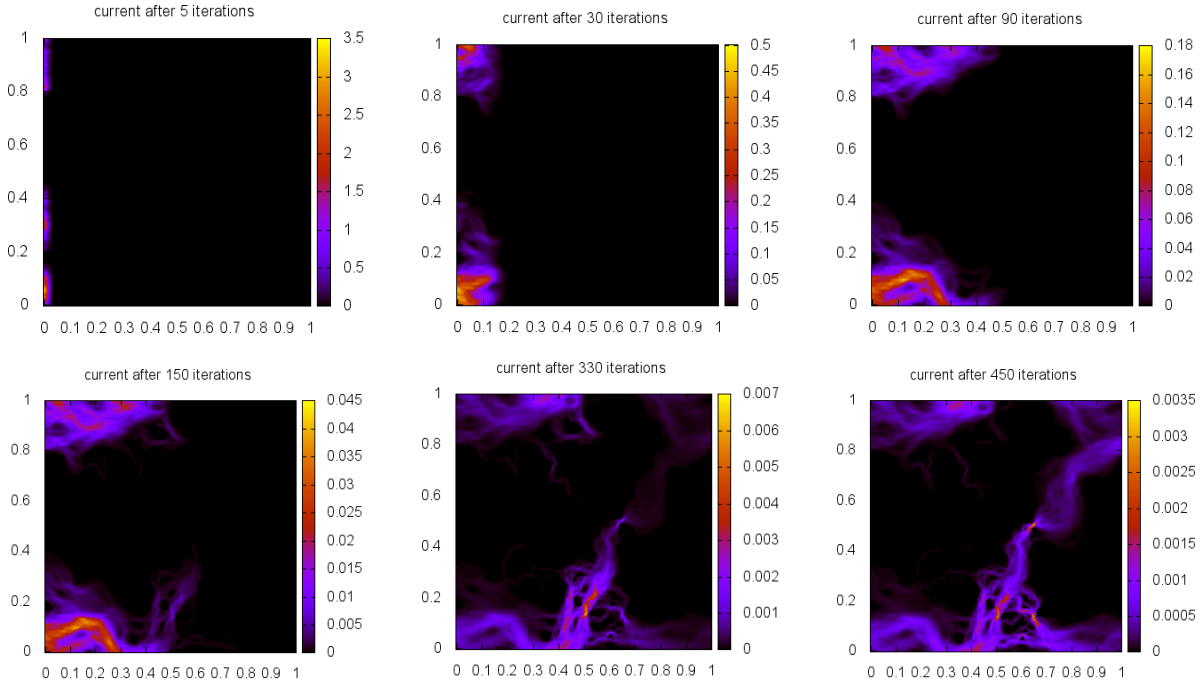
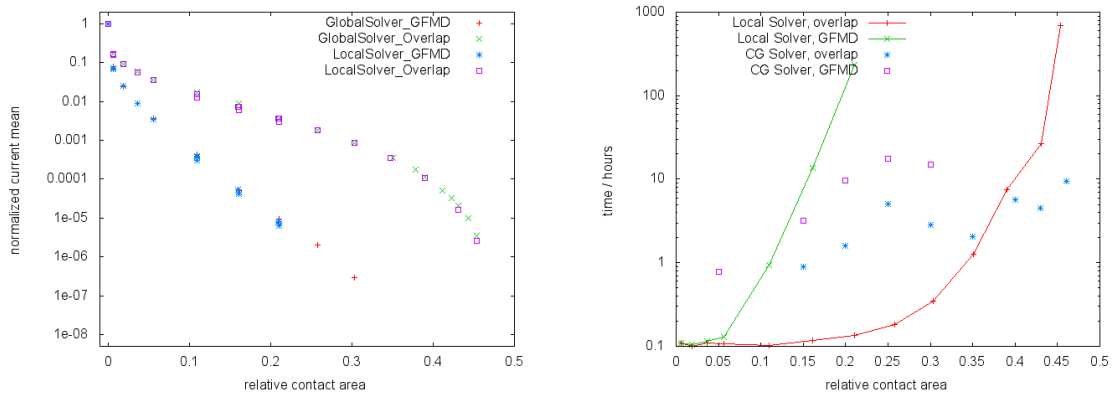


Figure 8: simulation with 15% relative contact area on 512×512 grid with zero initial guess

We now want to compare the multigrid, conjugate gradient solver ("global") for Reynolds thin film equation with the solver which was used in a previous work on this topic [3]. The latter solves the problem locally since each grid point only depends on the four nearest neighbours. The pressure for each grid point is varied in order to minimise the local and consequently the global error which is the sum of all local errors. Because iterations over all grid points are necessary, 'information' diffuses very slowly through the system ($\mathcal{O}(n^2)$ with n total amount of grid points). We can see in Fig. 10a that the global solver produces exactly the same results like the local solver: The normalised current mean[2] decreases much faster for the elastic model and for both models the percolation threshold is approached. After we have ensured that our global solver works properly, we now want to compare the runtimes of local and global solver.

In Fig. 10b we can observe the different runtimes of Bi-CGSTAB and the local solver for both the overlap and the elastic model on grids of size 4096×4096. In the overlap simulations, the local solver is more efficient for small contact area since in Bi-CGSTAB each iteration costs much more and the number of iterations, which the local solver needs to converge, is still acceptable. At about 40% relative contact area this turns and Bi-CGSTAB is significantly superior. Using the more realistic elastic model (where the local solver needs much more time to converge than in the overlap model) we recognise that Bi-CGSTAB is already from about 10% relative contact area on faster than the local solver. While the local solver needs about 10 days for the simulation, Bi-CGSTAB is converged after less than 10 hours. This discrepancy continues to mount with the runtimes of the local solver increasing exponentially while the conjugate gradient method seems to have an upper limit of iteration steps until converging.

[2]normalised by the fluid current for zero contact



(a) comparison of fluid current

(b) comparison of performance

Figure 10: comparison of local and global solver on 4096×4096 grids

Since the goal of our work was to calculate flow through elastic contacts with a relative contact area near the percolation threshold (which lies at about 42% for this model [3]) on grids of size 4096×4096, we are very confident that this new implementation of a conjugate gradient solver makes it possible to do these simulations in a reasonable time. This will enable us to describe and study the critical behaviour near the percolation threshold.

5 Acknowledgements

I want to thank Mathias Winkel and Ivo Kabadshow for the great organisation of this year's JSC guest student programme. Furthermore I would like to thank my advisers Prof. Dr. Martin Müser and especially Dr. Wolf Dapp for his help and advice in those 10 weeks. But most of all I would like to thank all the other guest students who made this guest student programme a unique experience for me.

References

1. BP. Deepwater Horizon: Accident Investigation Report (2010)
2. U.S. Government Printing Office. Report of the Presidential Commission on the Space Shuttle Challenger Accident (1986)
3. W. Dapp, A. Lücke, B. Persson, M. Müser. Self-Affine Elastic Contacts: Percolation and Leakage. Phys. Rev. Lett. 108 (2012) 244301
4. B. Schlögl. Embedding plasticity into a parallel Green's function molecular dynamics code. FZJ-JSC-IB-2012-01 (2012)
5. J. Egolf, S. Swaminathan, K. Spence. Air Bearing Optimization. Available from: http://www.math.udel.edu/ pelesko/Teaching/Math512_ Fall_ 2005/Milestone5(final).pdf
6. J. Nagel. Solving the Generalized Poisson Equation Using the Finite-Difference Method (2011)
7. R. Pozo, K. Remington, A. Lumsdaine. SparseLib++ v1.5 User's Guide (1996)
8. R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. 2nd ed. Philadelphia, PA: SIAM (1994)
9. A. Lücke. Viscous flow through fractal contacts. FZJ-JSC-IB-2011-06 (2011)

Visualizing Complex Functions Using GPUs

Khaldoon Ghanem

RWTH Aachen University
German Research School for Simulation Sciences
Wilhelm-Johnen-Straße
52428 Jülich

E-mail: k.ghanem@grs-sim.de

Abstract:
This document explains some common methods of visualizing complex functions and how to implement them on the GPU. Using the fragment shader, we visualize complex functions in the complex plane with the domain coloring method. Then using the vertex shader, we visualize complex functions defined on a unit sphere like spherical harmonics. Finally, we redesign the marching tetrahedra algorithm to work on the GPGPU frameworks and use it for visualizing complex scalar fields in 3D space.

---

1 Introduction

GPUs are becoming more and more attractive for solving computationally demanding problems. This is because they are cheaper than CPUs in two senses. First, they provide more performance for less cost i. e. cheaper GFLOPs. Second, they are more energy efficient i. e. cheaper running times. This reduced cost comes at the expense of less general purpose architecture and hence a different programming model.

There are two main ways of programming GPUs. The first one is used in the graphics community using shading languages like HLSL, GLSL and Cg. In these languages, the programmer deals with vertices and fragments and processes them with the so called vertex and fragment shaders, respectively. Actually, this has been the only way of programming GPUs for a while. Fortunately, in the recent years, frameworks for general programming have been developed like CUDA and OpenCL. They are much more suited for expressing the problem more abstractly in terms of threads. These threads are then processed with the so called kernels.

Visualizing complex functions makes an ideal problem to be solved on the GPU because the function needs to be evaluated at different points of the domain and these points are processed independently. We deal in this document with three types of complex functions; each one requires a different visualization

method and each method is most appropriately implemented on the GPU in a different way. The complex functions, we are addressing, are functions in complex plane, functions on unit sphere and functions in 3D space.

2 Complex Functions in Complex Plane

To visualize complex functions of a single complex variable, we would need a four-dimensional space! However, by encoding the values in the complex plane in colors, we are able to visualize these functions in two dimensions. This method is called **Domain Coloring** [1].

2.1 Domain Coloring

For visualizing the function $f : \mathbb{C} \to \mathbb{C} : w = f(z)$:

- Cover the range complex plane with some color map or scheme. i.e give each point w a color.
- For each point of the domain complex plane z, compute $w = f(z)$ and then color z with the color of w.

The result of the above procedure is a colored image of the domain (see Figure 1).

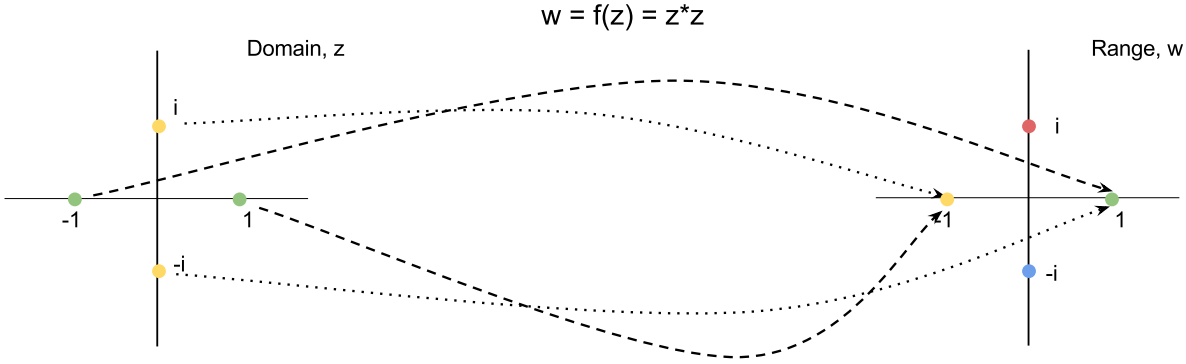


Figure 1: As an illustrating simple example, let us assume we want to visualize the square function z^2 at only four points $1, -1, i, -i$. We give each of these points in **Range** plane a unique color. Then we compute the function values at these points and color them according to the result: $f(1) = f(-1) = 1$ (green), and $f(i) = f(-i) = -1$ (yellow). The image to the right is called *color map* and the one to the left is *plot of* z^2

2.2 Choosing Color Map

The goal here is to give each complex number a color. In principle, you can use any picture to cover the complex plane. However, the resulting function plots are usually not easy to interpret. Also, the picture cannot cover the whole infinite plane, and thus we need a more systematic way.
For a complex number w, choose color hue according to the argument $\arg(w)$ from a smooth color sequence (color gradient). Then, choose color brightness according to the fractional part of the $\log_2 |w|$ (see Figure 2). For a thorough explanation on how to read the plots of such color maps, see [2].

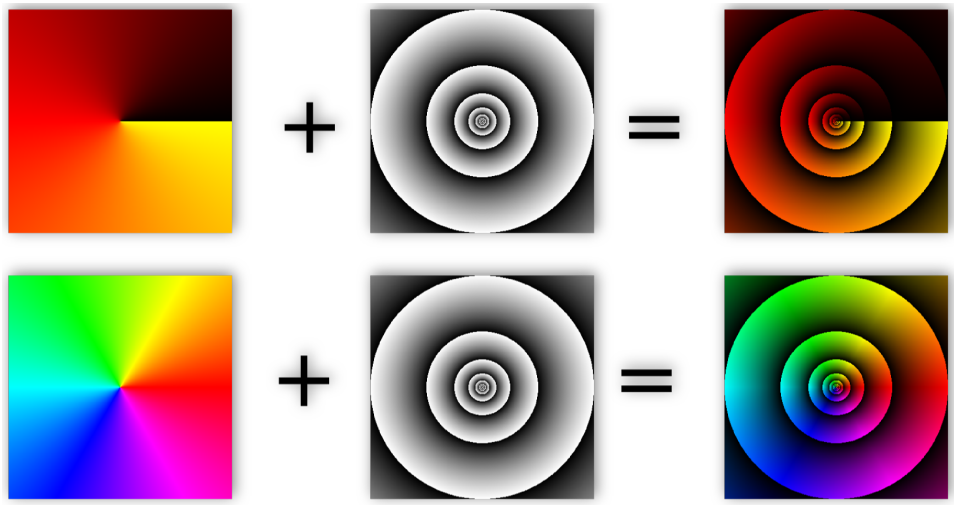


Figure 2: Two color maps differ by argument encoding. The first map interpolate between three colors; black, red and yellow. The second map spans the whole spectrum.

2.3 Using the GPU

The coloring of the domain complex plane is embarrassingly parallel as each point of the plane is processed independently. To utilize the GPU in this problem, we map complex points to screen pixels and color them with a fragment shader written in some shading language. The procedure outline is as following:

- Make a rectangle covering the whole screen using the graphics API , OpenGL or DirectX.
- The rectangle's fragments will be created by the fixed functionality of the graphics pipeline. These fragments are the final pixels of the screen, because there is only one primitive and it is not clipped.
- Fragment shader will be executed for each fragment. Inside fragment shader:
  - Transform the fragment screen coordinates to a complex point z using a transformation matrix passed from main program.
  - Compute the function at that point $f(z) = w$.
  - Determine the fragment color according to the color map at the resulting point w.

2.4 Implementation

We developed an application that visualize complex functions of a single variable. The main program is written using OpenGL graphics API and GLUT library while the fragment shader is written in GLSL shading language. The application takes as input a list of complex functions expressed with common mathematical symbols. Function expressions are parsed and translated into appropriate shader calls and operations using a lexical analyzer and a parser generated by lex and yacc tools.

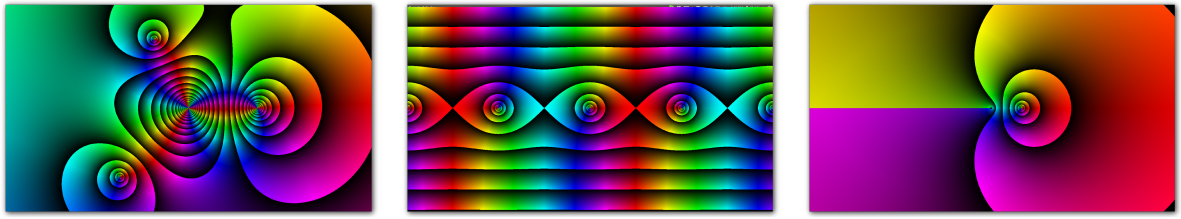


Figure 3: Plots of some complex functions generated using our domain coloring program: $(z-2)^2(z+1-2i)(z+2+2i)/z^3$ (left), $\sin(z)$ (center), $\log(z)$ (right).

3 Complex Functions on Unit Sphere

The second class of complex functions we address is functions defined on a unit sphere $f : [0,\pi] \times [0,2\pi) \to \mathbb{C} : f(\theta,\varphi) = w$. Although we could apply the domain coloring method to the surface of unit sphere, we are already visualizing in three dimensions and it is convenient to use the extra dimension we have at our disposal.

- Start with a unit sphere in 3D space.
- Each point on the surface has the spherical coordinates (r,θ,φ).
- Deform the sphere such that for each point $r = |f(\theta,\varphi)|$.
- Color the surface according to $\arg(f(\theta,\varphi))$ using some smooth color sequence as we did in domain coloring method 2.1.

3.1 Calculating the normals

Since we are now working in 3D, normals at surface points should be provided for appropriate lightening. To calculate them, we express the deformed unit sphere as an isosurface of a scalar field: $F(r,\theta,\varphi) = \sqrt{f(\theta,\varphi)\overline{f}(\theta,\varphi)} - r$ with an isovalue equals zero. Then the gradient of the field ∇F is normal to the isosurface.
The gradient in spherical coordinates is calculated using

$$\nabla F = \frac{\partial F}{\partial r}\hat{\boldsymbol{r}} + \frac{1}{r}\frac{\partial F}{\partial \theta}\hat{\boldsymbol{\theta}} + \frac{1}{r\sin\theta}\frac{\partial F}{\partial \varphi}\hat{\boldsymbol{\varphi}}$$

where

$$\frac{\partial F}{\partial r} = -1 \qquad \frac{\partial F}{\partial \theta} = \frac{\frac{\partial \overline{f}}{\partial \theta} f + \frac{\partial f}{\partial \theta}\overline{f}}{2\sqrt{f\overline{f}}} = \frac{Re[\frac{\partial \overline{f}}{\partial \theta} f]}{|f|} \qquad \frac{\partial F}{\partial \varphi} = \frac{Re[\frac{\partial \overline{f}}{\partial \varphi} f]}{|f|}$$

So we do not only need to calculate the function value but also the partial derivatives of its complex conjugate.

3.2 Using the GPU

We generate a unit sphere (vertices and triangles). Then, using a vertex shader, each vertex of the sphere is modified independently. Inside the vertex shader:

- Retrieve vertex's angles (θ, φ)
- Compute function value $f(\theta, \varphi)$
- Modify vertex coordinates such that $r = |f(\theta, \varphi)|$
- Modify vertex color according to $\arg(f(\theta, \varphi))$ using some smooth color sequence.
- Compute partial derivatives of the function and use them to compute the gradient vector.
- Modify vertex normal such that it points in the direction of the gradient.

3.3 Implementation

Spherical harmonics are the most well known complex functions on a unit sphere. They form a complete set of orthonormal functions thus any square-integrable complex function on a unit sphere can be expanded as linear combination of them. Due to their importance, we developed an application for visualizing linear combinations of spherical harmonics. The main program is written using OpenGL graphics API and GLUT library, while the vertex shader is written in GLSL shading language. For a stable method of evaluating spherical harmonics see [3].

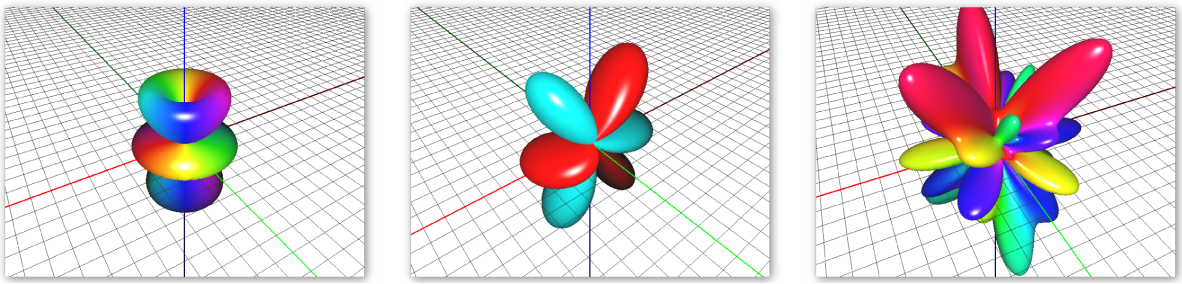


Figure 4: Plots of spherical harmonics Y_3^1(left), its real part (center), and the linear combination $\frac{1}{2}Y_1^0 + iY_5^3 + (\frac{1}{2} + i\frac{1}{2})Y_7^{-3}$ (right)

4 Complex Functions in 3D

We often need to visualize discrete complex functions defined in 3D space $f : \mathbb{R}^3 \to \mathbb{C}$ like wave functions resulting form quantum mechanics calculations. First, we use the trick of encoding the argument of the complex function in color as in the domain coloring method 2.1. What is left then, is visualising the absolute value which can be considered as a scalar field in 3D space. This suggests the following procedure:

- Specify one absolute value to visualize.
- Calculate an isosurface of the absolute value using marching tetrahedra.
- Color the isosurface according to the argument using some smooth color sequence.
- If necessary, change the isovalue and repeat process to gain more info.

4.1 Marching Tetrahedra

Marching Tetrahedra is an isosurface extraction algorithm. Given a 3D scalar field, it finds the surface on which the field has a constant value, the isovalue. The key idea of marching tetrahedra is noting that isosurface of a volume is the union of the isosurfaces of its components.

The field values are given at the points of a mesh. Any mesh is naturally divided into mesh cells. In this document, we consider structured meshes with *parallelepiped* cells (usually cubes). We could take the mesh cell as our building block and find the isosurface of each mesh cell independently and then collect them to form the final isosurface. This would be called **Marching Cubes Algorithm**[4, 5]. Marching cubes suffer from some ambiguities in finding the isosurface of a mesh cell. These ambiguities are not present in marching tetrahedra.

In **Marching Tetrahedra Algorithm**[6, 7], we go one step further beyond marching cubes and divide each mesh cell into six tetrahedra. Then our building block is the tetrahedron. There are several ways of splitting a hexhedron (usually a cube) into six tetrahedra. The way we do it is illustrated in (figure 5).

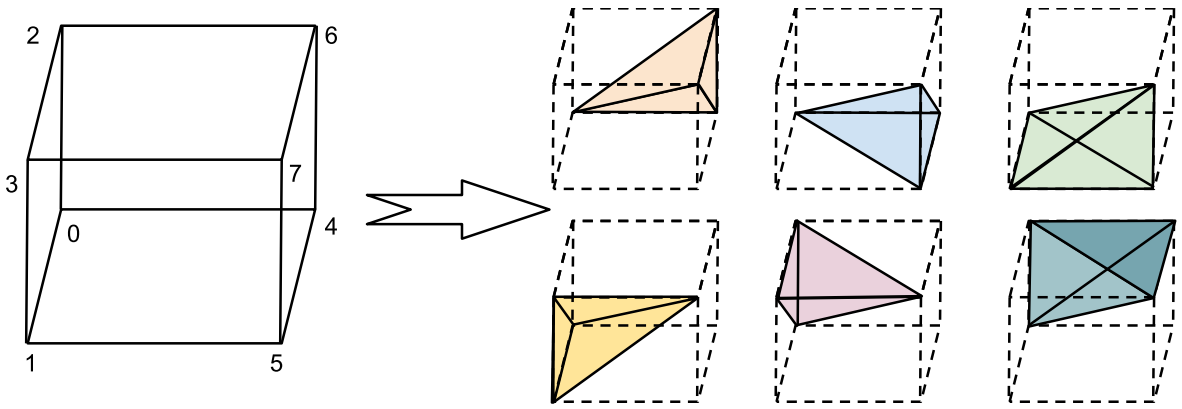


Figure 5: Splitting a cube (or a parallelepiped in general) into six tetrahedra. They are listed from right to left, top to bottom: (0,6,4,7), (0,4,5,7), (0,5,1,7), (0,5,1,7), (0,1,3,7), (0,3,2,7), (0,2,6,7).

To find the isosurface of one tetrahedron, we check whether each of its four corners is above or below the isovalue (let's denote them as a plus or minus, respectively). The isosurface clearly passes only through edges connecting corners of opposite signs. Since we only know field values at corners, we linearly interpolate them along the edges to get the intersection points (see Figure 6).

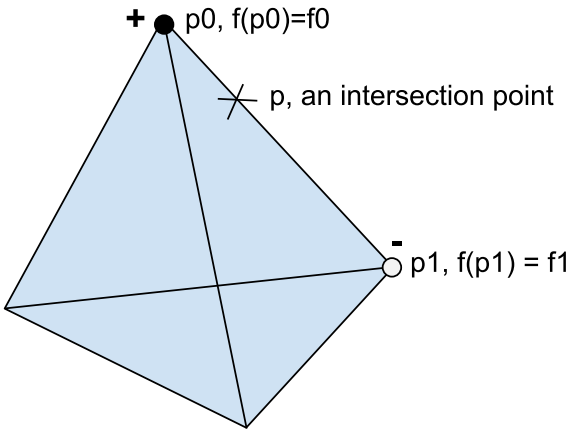


Figure 6: The intersection of isosurface with an edge can be computed by considering a linear interpolation along the edge. If two corners p0 and p1 are of opposite signs then the intersection point on their edge p is computed as $p = \alpha p_1 + (1 - \alpha)p_0$ where $\alpha = \frac{c - f_0}{f_1 - f_0}$ and c is the isovalue.

Besides getting the intersection points (isosurface vertices), we want to connect them. i.e form triangles. First, note that the triangulation depends only on the signs of corners, not their exact field values. Since we have four corners with two possible signs each, we have in total $2^4 = 16$ different possible patterns of a tetrahedron.

We enumerate the corners and represent the pattern of a tetrahedron by a 4-bits number, where each bit indicates whether the corresponding corner is above or below the isovalue. Then we enumerate the edges and use a lookup table with 16 entries. Given the tetrahedron's pattern, the table should return isosurface triangles in terms of the edges they connect (see Figure 7).

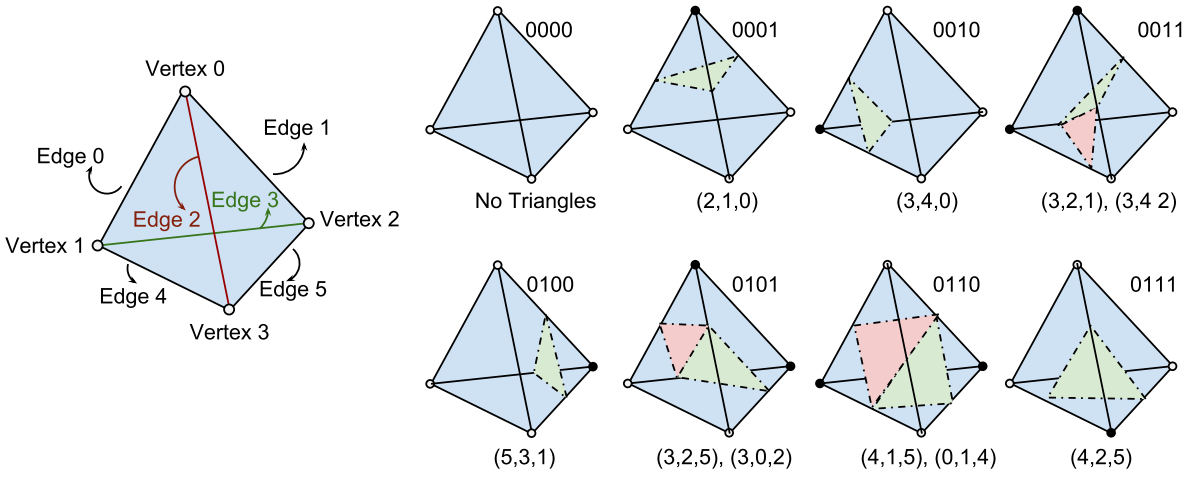


Figure 7: To the left is our choice for enumerating corners and edges. To right is the triangulation for eight different tetrahedron's pattern. The other eight are obtained by flipping patterns' bits and listing edges in reverse order.

4.2 Avoiding Duplicate Isosurface Vertices

We note in the original marching tetrahedra, that isosurface vertices (intersection points) lying on edges shared between adjacent tetrahedra are duplicated. This is a disadvantage for two reasons. First, it leads to unnecessary storage of repeated vertices e.g. a vertex on the diagonal of a mesh cell would be repeated six times. Second, if the generated surface is to be processed later or used for some calculations, then this could produce artifacts due to round-off errors. To avoid this duplication, we split the algorithm into two stages: Generating Vertices and Generating Triangles.

In the **Generating Vertices** stage, we loop over all edges. For each edge, check whether it is cut by the isosurface (by checking whether the two end mesh points are of opposite signs). If so, calculate the intersection point (isosurface vertex). Then store the vertex in a hash table indexed by some edge id.

In the **Generating Triangles** stage, we loop over all mesh cells. For each mesh cell, process all six tetrahedra. For each tetrahedron, calculate its pattern. Using the tetrahedron's pattern, get its triangulation. Generate triangles using pointers to vertices (not vertices directly). Get vertices' pointers by looking up edge-vertex hash table generated in the previous stage.

For this to work, we need to identify the edges globally and we do it by associating each mesh point with seven edges (see Figure 8). This association is unique and covers all possible edges. Then the id of an edge connecting mesh points (i_1, j_1, k_1) and (i_2, j_2, k_2), where $0 \leq i_2 - i_1 \leq 1$ and $0 \leq j_2 - j_1 \leq 1$ and $0 \leq k_2 - k_1 \leq 1$, is a tuple with two entries. The first is the minimum of the two mesh point ids (i_1, j_1, k_1). The second is a number between 1 and 7 identifying the edge within the mesh point and calculated as $(i_2 - i_1) * 4 + (j_2 - j_1) * 2 + (k_2 - k_1)$.

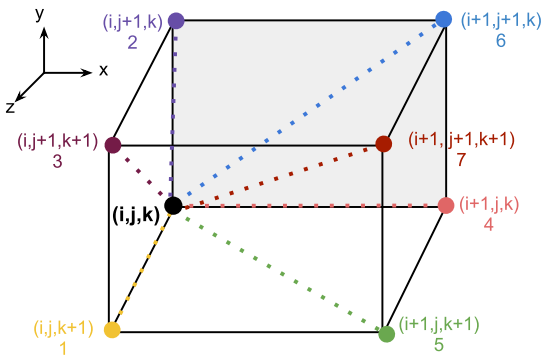


Figure 8: Each mesh point (i,j,k) can be associated uniquely with seven edges. 1: (i,j,k)-(i, j, k+1) ; 2: (i,j,k)-(i, j+1, k); ; 3: (i,j,k)-(i, j+1, k+1) ; 4: (i,j,k)-(i+1, j, k) ; 5: (i,j,k)-(i+1, j, k+1) ; 6: (i,j,k)-(i+1, j+1, k) ; 7: (i,j,k)-(i+1, j+1, k+1).

Note that using this association, the loop over edges in first stage is actually a double loop where the outer is over mesh points and the inner is over edges associated with each mesh point. Also note that the triangulation of a tetrahedron is, as before, expressed in terms of our enumeration of edges inside

the tetrahedra (see Figure 7). So a mapping from the local edge enumeration to global edge ids should be done before retrieving vertices' pointers from the hash table.

4.3 Using the GPU

Unlike the previous two visualization methods, domain coloring and sphere deformation, we won't use a shading language but rather a GPGPU language like OpenCL or CUDA. Although it is possible to implement the marching tetrahedra on vertex or fragment shaders (see [8, 9, 10]), it is more convenient to express the problem more abstractly using GPGPU. There are implementations of marching cubes on OpenCL and CUDA (see [11, 12]), but to our knowledge, there is no public implementation of marching tetrahedra using GPGPU yet.

We go along the same line of thought as in the last algorithm and separate generating vertices from triangles. First a collection of threads, one for each mesh point, is created. Each of these threads runs a kernel that would generate vertices lying on edges associated with the corresponding mesh point. Then another collection of threads is created, one for each mesh cell. Each of these threads would generate the triangles of tetrahedra associated with the corresponding mesh cell.

There are two main complications on the GPU. First, there is no dynamic memory allocation. So all memory allocation and deallocation should be done before or after the computation but not during it. One solution is to allocate memory for every potential vertex. However, this is impractical, as it would need (7 vertices per mesh point $\times$ 3 coordinates per vertex) = 21 times the storage needed for the scalar field. A similar argument goes for the triangles.
Second, the threads should work independently and write data to distinct memory locations. So each thread should know where to write its results before starting the computation.
Circumventing these problems is done by splitting each stage into yet another three stages. First, we count the number of vertices that would be generated per mesh point. Second, a prefix-scan is done to get the number of generated vertices and a list of addresses for storing the vertices associated with each mesh point. Finally, the necessary memory is allocated and vertices are generated and stored in their appropriate locations. Triangle generation is also split into three stages similarly. This method is highly inspired by [11, 12].

Generating Vertices

1. Allocate necessary memory on GPU for array `mpVertsNum`.
   `mpVertsNum`: An array of bytes. Its size equals the total number of mesh points. Element `mpVertsNum[mp]` contains the number of vertices associated with mesh point `mp`.
2. Run kernel `processMP` for each mesh point to fill `mpVertsNum`.
   Elements of `mpVertsNum` are calculated by counting the number of intersected edges associated with each mesh point. An edge is intersected if its two ends are of different signs (one above the isovalue and the other below).(see Figure 6)
3. Run kernel `preScan` on `mpVertsNum` and store the result in `mpVertsBaseAddress`.

`mpVertsBaseAddress`: An array of integers. Its size equals the total number of mesh points. Element `mpVertsBaseAddress[mp]` indicates where vertices, associated with mesh point `mp`, should be stored in the vertex array. It is computed as a prefix-scan of array `mpVertsNum`. So element `mpVertsBaseAddress[mp]` contains the sum of `mpVertsNum` elements up to and excluding `mp`. There is an efficient implementation of prefix-scan algorithm on GPU (see [13]).

4. Allocate memory on GPU for array `verts` of size `vertsNum`.
   `vertsNum`: An integer containing the number of generated vertices.
   It is computed as the sum of the last entries of `mpVertsNum` and its prefix-scanned version `mpVertsBaseAddress`.
   `verts`: An array of floats. Its size equals three times `vertsNum`. It stores the coordinates of the generated vertices. The three coordinates of each vertex are stored consecutively. The vertices associated with each mesh point are stored consecutively from `3*mpVertsBaseAddress[mp]` till `3*mpVertsBaseAddress[mp+1]`. Vertices of a mesh point are ordered according to the local ordering of the edges they lay on (see Figure 8).
5. Run kernel `generateVerts` for each mesh point to fill `verts`.
   Vertex coordinates are computed as linear interpolation (see Figure 6).

Generating Triangles

1. Allocate necessary memory on GPU for array `mcTriangsNum`.
2. Run kernel `processMC` for each mesh point to fill array `mcTriangsNum`.
3. Run the kernel `preScan` on array `mcTriangsNum` and store the result in array `mcTriangsBaseAddress`.
4. Allocate memory on GPU for array `triangs`.
5. Run kernel `generateTriangs` for each mesh cell to fill array `triangs`.

The semantics are very similar to **Generating Vertices**; Just replace mesh points with mesh cells, edges with tetrahedra and vertices with triangles.
One complication in the final step is getting the addresses of vertices which will be used in forming triangles. Triangles are expressed in terms of the edges on which their vertices lie. So the problem reduces to knowing where the vertex of a certain edge is stored.
If `mp` is lowest numbered mesh point of an edge then the vertex of that edge will be located at `3*mpVertsBaseAddress[mp]` in array `verts` with some additional shift that depends on other vertices associated with `mp`. To get this shift easily, we build an additional array `mpVertsEdgeIndex` where the bit number `i` of element `mpVertsEdgeIndex[mp]` indicates whether edge number `i` associated with mesh point `mp` is intersected by the isosurface. This way, all the information needed for determining shifts of vertices associated with mesh point `mp` are contained in `mpVertsEdgeIndex[mp]`. The computation of this array is most conveniently done inside kernel `processMP`.

4.4 More Details and Optimization

In the previous description, we omitted several aspects of the method to make the presentation more clear. They are explained here:

- Many subtask like splitting the mesh cell into tetrahedra, getting triangles from tetrahedra pattern, getting shifts from `mpEdgeIndex`, etc. can be done using look-up tables. These look-up tables should be stored in the constant memory of GPU which is basically a small fast read-only memory.
- Considerable speed-up can be obtained by considering only mesh points that actually have a non-zero number of associated vertices and only mesh cells that actually generate triangles. This requires building yet another array for knowing which mesh points are 'active'. Note also that by associating each mesh cell with its lowest numbered mesh point, a mesh cell can be skipped if that mesh point is not active.
- Kernels `processMP` and `processMC` can be combined for speed-up. This is because every mesh cell can be associated with its lowest numbered mesh point and then both kernels need to read the same memory locations. Thus, combining them saves repeated memory accesses.
- For getting the normals, the central difference formula can be used to get normals at mesh points. This could be done outside the marching tetrahedra algorithm. The normals at isosurface vertices are then computed in the same way as the coordinates i. e. as a linear interpolation and this computation can be incorporated inside kernel `generateVers`.

4.5 Implementation

Due to time constrains, we developed, as a proof of concept, an application running on CPU but following the GPU-tailored algorithm. An OpenCL application is still under development. The application takes as input GAUSSIAN CUBE format files for reading in the mesh and the scalar data. For full account of complex functions, this file should be accompanied with another file specifying the argument. After loading the data, the user can interactively change the isovalue to see different isosurfaces.

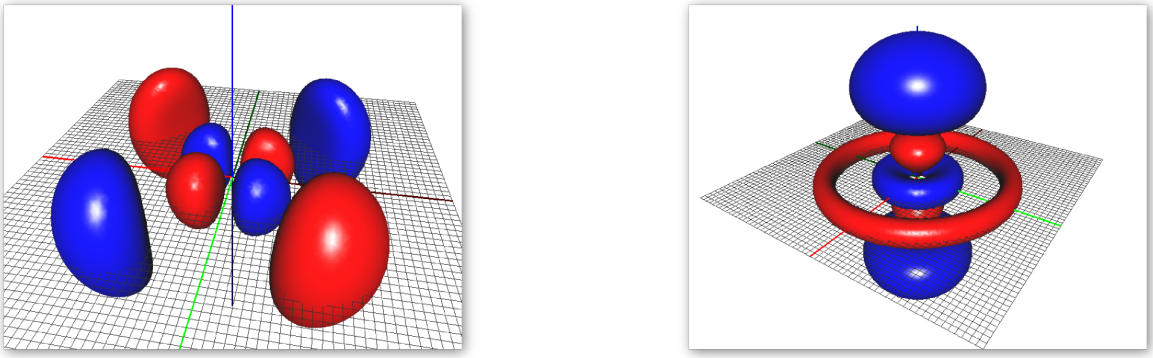


Figure 9: Isosurface plots of hydrogen wave functions $4d_{xy}$(left) and $4d_{3z^2-r^2}$ (right)

5 Summary

We described how to visualize three different classes of complex functions using the GPU. First, we explained how to visualize complex functions of a single complex variable using the domain coloring method on the fragment shader. Then, we explained how to visualize complex function defined on unit sphere by deforming that sphere and coloring it. This is done on the vertex shader. Finally, we explained how to visualize complex functions in 3D space by extracting the isosurfaces of the absolute value using marching tetrahedra method and then coloring that surface. This was designed to work on a GPGPU framework.

6 Acknowledgements

I would like to thank my supervisor Prof. Erik Koch for bringing this interesting program to my attention to and for providing the necessary knowledge and advice to finish the project successfully. I would like also to thank German Research School for Simulation Sciences for sponsoring my project. Last but not least, I would like to thank Jülich Supercomputing Center staff and in particular Mr. Mathias Winkel for organising the program and making it a unique experience.

References

1. Wikipedia contributors. Domain Coloring [Internet]. Wikipedia, The Free Encyclopedia [updated 2012 September 20; cited 2012 Oct 07]. Available from: http://en.wikipedia.org/wiki/Domain_coloring
2. Lundmark M. Visualizing complex analytic functions using domain coloring [internet]. 2004 May [cited 2012 Oct 07]. Available from: www.mai.liu.se/ halun/complex/domain_coloring-unicode.html
3. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical Recipes, The Art of Scientific Computing. 3rd ed. New York: Cambridge University Press;2007. Chapter 6, Special Functions; p.292-295.
4. Lorensen WE, Cline HE. Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH Comput. Graph. 1987 Aug;21(4):163-169.
5. Bourke P. Polygonising a scalar field [Internet]. 1994 May [cited 2012 Oct 04]. Available from: http://paulbourke.net/geometry/polygonise/
6. Doi A, Koide A. An Efficient Method of Triangulating Equivalued Surfaces by using Tetrahedral Cells. IEICE Trans Inf Syst. 1991 Jan;E74(1):214-224.
7. Bourke P. Polygonising a Scalar Field Using Tetrahedrons [Internet]. 1997 Jun [cited 2012 Oct 04]. Available from: http://paulbourke.net/geometry/polygonise/
8. Reck F, Dachsbacher C, Grosso R, Greiner G, Stamminger M. Realtime isosurface extraction with graphics hardware. Eurographics 2004 Short Presentations; 2004.
9. Pascucci V. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Proceedings of IEEE TCVG Symposium on Visualization; 2004. p. 293–300.
10. Klein T, Stegmaier S, Ertl T. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. Proceedings of Pacific Graphics '04; 2004. p.186–195.
11. NVIDIA CUDA SDK - Physically-Based Simulation - Marching Cubes Isosurfaces [Internet]. 2008 [updated 2009 June 15; cited 2012 Oct 05]. Available from: http://www.nvidia.com/content/cudazone/cuda_sdk/Physically-Based_Simulation.html#marchingCubes
12. NVIDIA OpenCL SDK Code Samples -OpenCL Marching Cubes Isosurfaces [Internet]. [cited 2012 Oct 05]. Available from: http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/website/OpenCL/html/samples.html
13. Harris M, Sengupta S, Owens JD. GPU Gems 3. Addison-Wesley Professional; 2007. Chapter 39. Parallel Prefix Sum (Scan) with CUDA.

Porting and optimization of EPOCH (a Particle-In-Cell code) to Blue Gene/Q

David Martín Rodríguez

Universidad de Salamanca
Facultad de Ciencias
Plaza de la Merced
37008 Salamanca (Spain)

E-mail: davidmr@usal.es

Abstract:
Laser-plasma interaction can be simulated with Particle-In-Cell (PIC) codes. We had a MPI parallelized PIC code, called EPOCH, running on a BlueGene/P architecture in the Jülich Supercomputing Centre (JSC). Since the JSC was upgrading the hardware to a BlueGene/Q, the code needed to be ported to the new architecture. Here we analyze the parallelization of the code and discuss the possible need for hybrid parallelization to adapt the code to the multicore nature of BlueGene/Q. Therefore we implement several hybrid parallelization strategies and analyze the impact on the code performance.

1 Introduction

Due to the advances in laser technology we can nowadays experiment on high intensity light-matter interaction. Either for economic, security or technological reasons it is not always possible to experiment directly with these high intensity sources. Therefore we need a method to study and predict laser-matter interactions. Nowadays the most reliable tools we can use for this purpose are the Particle-In-Cell (PIC) codes. The PIC code used in this project, EPOCH, is an open source Fortran PIC code for high intensity laser-plasma interactions. It is currently fully MPI parallelized using domain decomposition. This decomposition can even vary dynamically due to a load balancing option already implemented in the code. The installation of a new BlueGene/Q cluster in the JSC triggered the need to port many codes to this new architecture. In the porting process of our code we discovered some architecture specifics that can be exploited to obtain a better performance since the BlueGene/Q system provides multicore compute nodes.

In order to improve performance we ought to implement a hybrid parallelization mechanism. Meaning not only use MPI, but rather combine it with several threads per MPI rank. An analysis of the code performance has been conducted in order to reveal where we should focus our efforts to improve the code. Finally we have designed, implemented and compared several hybrid parallelization

strategies. Comparisons show that hybrid parallelization offers a better performance under specific conditions.

2 Particle-In-Cell codes

Since the introduction of the Chirped-Pulse-Amplification (CPA) technique, laser intensities are strong enough to produce a massive ionization on matter, which turns into a plasma. For simulation purposes it can be approximated as a collection of free charged particles.

To study the the laser-plasma interaction we can make use of some simulation tools called PIC codes. In a PIC code the plasma is represented as a collection of *macroparticles*. Each of this macroparticles is a representation of several real particles of a particular sort (protons, electrons, ...). Numerically it can be treated as a distribution of charges. The laser is not considered as a collection of photons, but as an electromagnetic field. The domain is discretized in cells in which the electromagnetic field is stored.

2.1 Basic principles of a PIC code

A PIC code, using a a Finite Difference Time Domain (**FDTD**) scheme, solves the evolution of an electromagnetic field described by the Maxwell equations and the particle motion using the Lorentz force.

2.1.1 Electromagnetic field propagation

The Maxwell equations solved numerically by the PIC codes describe the behavior of the electromagnetic field:

$$\vec{\nabla} \times \vec{E} = -\frac{1}{c}\frac{\partial \vec{B}}{\partial t} \tag{1}$$

$$\vec{\nabla} \times \vec{B} = \frac{1}{c}\frac{\partial \vec{E}}{\partial t} + \frac{4\pi}{c}\vec{J} \tag{2}$$

$$\vec{\nabla} \cdot \vec{E} = 4\pi\rho \tag{3}$$

$$\vec{\nabla} \cdot \vec{B} = 0 \tag{4}$$

where $\vec{J}$ is the current, E is the electric field, B is the magnetic field, and ρ the charge density. The last two equations are actually set as initial conditions.

2.1.2 Particle Motion

A point charge experiences a force (called Lorentz force) due to electromagnetic fields

$$F = q[E + (v \times B)] \,, \tag{5}$$

where F is the force (N), q is the electrical charge of the particle (cu) and v is the current velocity of the particle (m/s).

2.2 PIC algorithm

Essentially, a PIC code consists of a three step loop (Figure 1). Each iteration of the loop resolves one *time step*.

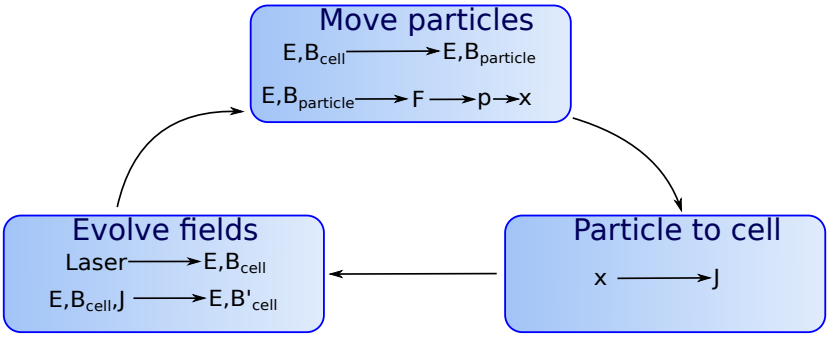


Figure 1: PIC loop

- **Move particles**.

  We have the electromagnetic field of each cell and we need to calculate the movement of the particles due to this electromagnetic field. Hence we first need for each particle to interpolate the fields stored in the cells nearby. After doing this we will know the field that affects each particle. The next step is to calculate the force generated on each particle and then move the particle. For doing this we apply the Boris algorithm that resolves the Lorentz force.

- **Particle to cell**.

  Due to the movement of the particles there is some amount of charge going through the boundaries of the cells. In other words, some currents are occurring at the boundaries of the cells. For each boundary that a particle crosses, we deposit some current in the corresponding cell.

  These two logical steps are usually joined in the real codes into a single step called *push particles*.

- **Evolve fields**.

  This step, also called *solver*, resolves the first two Maxwell equations, calculating the evolution of electromagnetic field for a *time step* using the currents and the previous electromagnetic field.

  The introduction of the laser electromagnetic field in the system depends on the code. On some PIC codes it is introduced at every iteration from one side of the simulation box. In other codes the whole laser pulse is introduced in the simulation box at the beginning.

3 EPOCH

EPOCH is an open source Particle-In-Cell code. Basically it is a loop over three major functions:

- `update_eb_fields_half`
- `push_particles`
- `update_eb_fields_full`

`push_particles` correspond to the steps *Move particles* and *Particle to cell* of section 2.2. The reason for merging this two steps is computational because some calculations (the movement of each particle) and some modifications (the currents of the cells) must be done for each particle. That means, if the two steps are joined together in one single function, we can avoid looping twice over all the particles.

Concerning the step *Evolve fields*, here it is split in two halves: `update_eb_fields_half` and `update_eb_fields_full`.

3.1 Parallelization

As discussed in the introduction, EPOCH was originally parallelized using pure MPI. The domain is decomposed in several parts and distributed among all the MPI ranks (Figure 2). That means that every rank deals with some cells and the particles in this cells. The distribution of cells and particles among the ranks can change even during runtime due to the dynamic load balancing implemented in the code. Nevertheless the idea persists: one rank does the calculations for one set of cells and the particles inside.

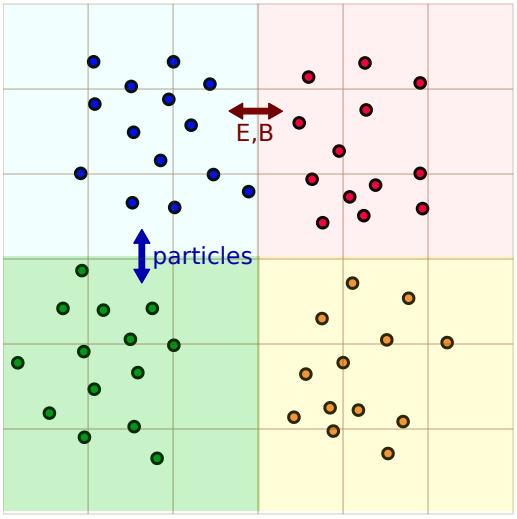


Figure 2: EPOCH parallelization scheme. In the example the domain is decomposed in four parts for four MPI ranks. The rank deals with the particles of his area of influence

The ranks have to communicate for each and every iteration of the loop. On one hand there is an exchange of particles between ranks and on the other, an exchange of fields.

4 Hybrid parallelization of EPOCH on JUQUEEN

JUQUEEN is the BlueGene/Q cluster we wanted our code to work on. BlueGene/Q is based on compute nodes with 16 GB of Memory and 16 cores that are able to execute up to 64 threads simultaneously by hardware hyper-threading. That joined with the fact that each rank has to deal with a large amount of particles opened the door for **hybrid parallelization** to take advantage of the multicore architecture. If we use more than one thread per rank, these threads have access to the data of the rank because they share memory. Therefore the computational load can be distributed among these threads.

4.1 EPOCH analysis

In order to know where to apply the hybrid parallelization we need to understand the behavior of the code.

We have tested the code for hard scaling using different optimization options in the compiler (Figure 3), concluding that:

- EPOCH scaling behavior is impressive. That means that the code is perfectly suitable for a supercomputer like JUQUEEN.
- The best optimization tested for the code is O3. O5 optimization does not show any improvements.

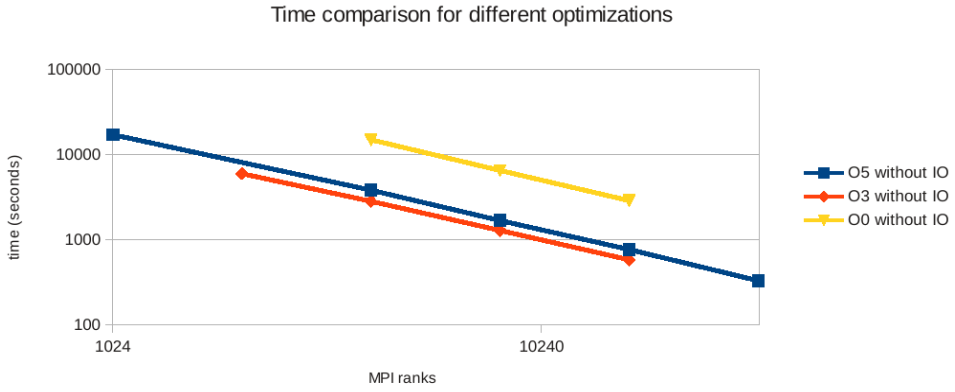


Figure 3: EPOCH time comparison for different optimizations of the compiler.

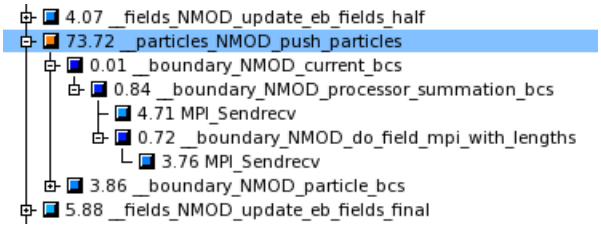


Figure 4: Time distribution.

If we look at how the time is distributed in the execution of the code, we can see that most of the time is spent in the *particle pusher*. 73% of the time is spent in the calculations and around 8,4% in communications inside the pusher (Figure 4).

If we now focus on the time used in communications in the whole system (Figure 5) we notice that most of the time is spent in the communications of the pusher. In contrast if we take a look at the number of communications and the bytes transferred in each function (Figures 6 and 7) we can assure that only a third of the bytes communicated are transferred during the pusher. In conclusion we can remark that the communications are more efficient in the solver than in the pusher.

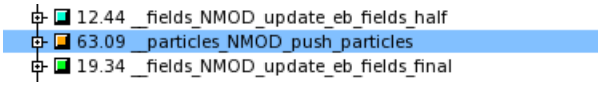


Figure 5: Comunication time distribution.

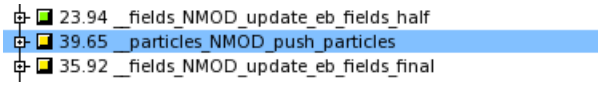


Figure 6: Comunications distribution.

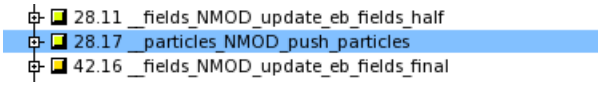


Figure 7: Bytes transferred distribution.

4.2 Hybrid parallelization strategies

We considered two technologies: **OpenMP** and **IBM Speculative execution**.

The first one is commonly known and explaining it is far from the purpose of this document.

Speculative execution can be considered a step further in compiler-directed code parallelization. In order to accomplish this kind of parallelization, the compiler must analyze the code and will only be able to parallelize it in certain cases if there are no problems with dependencies. Since some languages are really versatile, there may be some hidden dependencies that lead to errors in execution. In the new IBM Blue Gene/Q this problem is solved by the compiler which is able to detect conflicts in execution time and roll back those threads that were involved in the conflict.

4.2.1 Data Hazard problem

As we said, our goal was applying hybrid parallelization in the pusher algorithm. Thus, we first need to understand the algorithm. In the next lines we can see a simplified pseudocode:

```
foreach part in particles {
  dx = calculateMovement(cellFields)
  part->xNew = part->xOld + dx
  currents = calculateCurrent(part->xOld, part->xNew)
  indexes = calculateCellsToBeUpdated(part->xNew)
  foreach index in indexes{
```

```
        cellCurrent[index] += currents[localIndex]
      }
    }
```

For each particle we need to calculate the movement it describes due to electromagnetic field stored in the cells. In other words, we need to calculate the variation in its position (`dx`). After moving the particle we calculate the current that its movement generates in the boundaries of the cells (`currents`) and also we need the addresses of the cells that are affected by that particle (`indexes`). Once we know what currents must be deposited in which cells, we just loop over the affected cells and update their currents (`cellCurrent`).

The immediate parallelization of this algorithm is using one thread for a subset of particles. The problem is that there is a chance that two or more threads try to update the same cell at the same moment. This concurrency problem of two threads updating the same value at the same time is commonly known as **data hazard**. We have implemented several solutions to this problem and they will be discussed in the next sections.

4.2.2 OpenMP unsafe version

This is not really a solution rather the version of the code in which we do not really take care of the problem. Therefore, the execution of this version can lead to incorrect results. But it can be considered a tool for testing purposes, since it give us an idea of the best performance we can expect with hybrid parallelization. The loop is parallelized with OpenMP using a `PARALLEL DO` as we can see in the next pseudocode.

```
!$OMP PARALLEL ...
  // declaration of private variables

  !$OMP DO
  foreach part in particles {
    dx = calculateMovement(cellFields)
    part->xNew = part->xOld + dx
    currents = calculateCurrent(part->xOld, part->xNew)
    indexes = calculateCellsToBeUpdated(part->xNew)
    foreach index in indexes{
      cellCurrent[index] += currents[localIndex]
    }
  }
```

4.2.3 OpenMP with atomic operations

In order to avoid the data hazard problem, we can just tell OpenMP that the problematic updates must be done as atomic operations. This way when one thread is updating the current of one cell, there is no way another thread can burst into the operation. Therefore, the final results should be correct. This time the updates are marked as atomic as we can see in the next pseudocode:

```
...
  !$OMP DO
  foreach part in particles {
    ...
    foreach index in indexes{
      !$OMP ATOMIC
```

```
        cellCurrent[index] += currents[localIndex]
      }
    }
```

4.2.4 OpenMP with reduction

OpenMP provides a clause to perform a reduction on one or more variables. A private copy for each list variable is created for each thread. At the end of the reduction, the operation on the variable is applied to all private copies of the shared variable and the final result is written to the global shared variable. The loop is parallelized with OpenMP using a `PARALLEL DO`. In this case we add the reduction clause in the OpenMP directive. In our code, the reduction is done over the array of currents (`cellCurrent`). Reductions on arrays are implemented in Fortran, in contrast to other languages like C.

non

```
    !$OMP PARALLEL ... REDUCTION(+:cellCurrent)
      ...
      !$OMP DO
      foreach part in particles {
        ...
        foreach index in indexes{
          cellCurrent[index] += currents[localIndex]
        }
      }
```

4.2.5 Speculative execution

In this case we told the compiler to execute the loop with speculative threads, as can be seen in the next code:

```
    !SEP$ SPECULATIVE DO ... REDUCTION(+:cellCurrent)
      foreach part in particles {
        ...
        foreach index in indexes{
          cellCurrent[index] += currents[localIndex]
        }
      }
```

4.2.6 OpenMP storing currents

Here the strategy is to loop twice over the particles, making all the calculations in the first loop which will be parallelized with OpenMP. In this first loop the calculated currents must be stored for each particle, so that in the second loop we only need to update the values of the cells:

```
      !$OMP DO
      foreach part in particles {
        dx = calculateMovement(cellFields)
        part->xNew = part->xOld + dx
        part->currents = calculateCurrent(part->xOld, part->xNew)
        part->indexes = calculateCellsToBeUpdated(part->xNew)
      }
```

```
foreach part in particles {
  foreach index in part->indexes{
    cellCurrent[index] += part->currents[localIndex]
  }
}
```

4.3 Results

4.3.1 A small problem

In order to know if a hybrid parallelization will work or not, we used a small simulation with only one MPI rank. The question was whether we could beat the execution time of the version with pure MPI. The results (Figure 8) show that:

- Speculative execution does not seem to be a valid alternative since it needs more time than the pure MPI version probably because there are so many conflicts in execution time that the threads roll back wasting runtime.
- The version storing currents does not give many benefits. Probably because we need to iterate through all the particles twice.
- The version with atomic operation scales better than the rest, although it starts of with some overhead.
- The performance of version with reductions is close to the unsafe version since it does not need to spend so much time in synchronization mechanisms.

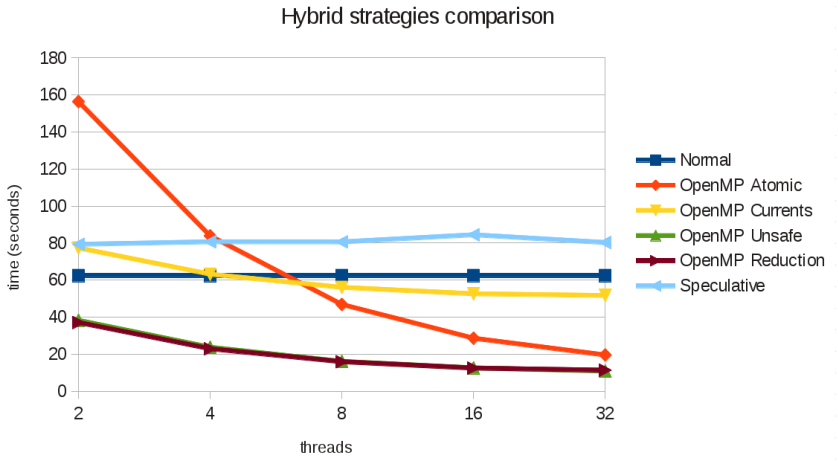


Figure 8: Hybrid parallelization comparison for a small job with only one MPI rank. The normal version (dark blue with squares) refers to the pure MPI version that actually uses only one thread.

4.3.2 A real case

We need to analyze the results on a larger scale with a real problem. We chose a fixed number of nodes (512) and made some tests varying the number of MPI ranks per node and then the number

of threads per rank. If we use only one rank per node and increase the number of threads per rank (Figure 9, left) we can conclude that several of the hybrid parallelization strategies beat the peformance of a pure MPI version. But if we increase the number of MPI ranks per node (Figure 9, right) the number of available threads decreases, and performance is better with a pure MPI version.

The best option (OpenMP with atomic operations) scales really well, but for 8 ranks per node the number of available threads is not enough to compensate the time spent in the creation of the threads. The reader might find one option missing in this real case scenario: the OpenMP reduction. This version fails in execution time while creating the threads because, for each reduction variable, OpenMP creates a local copy for each thread and we presume that the size of the variables in this case scenario was too large.

We have also found the best parameters (ranks per node and threads per rank) for each strategy comparing the best time for each strategy (Figure 10).

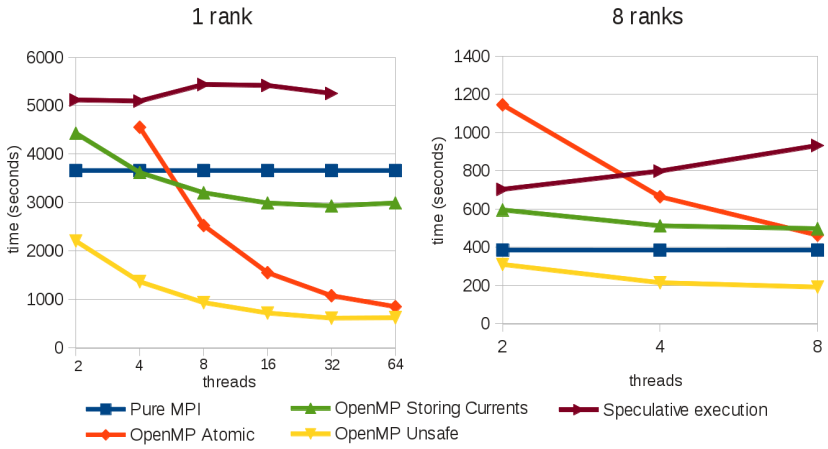


Figure 9: Hybrid parallelization comparison for a more realistic simulation with only 1 MPI rank on the left and with 8 on the right. The pure MPI version (dark blue with squares) uses only one thread.

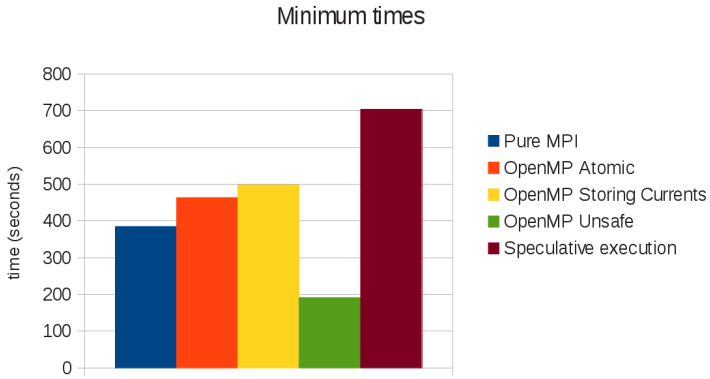


Figure 10: Best times for each strategy. We have obtained the best time for each strategy changing the number of ranks per node (from 1 to 8) and the number of threads per rank

5 Conclusions

We have analyzed the performance of the MPI parallelized PIC code (EPOCH) on a BlueGene/Q cluster concluding that there was a chance to improve the performance using hybrid parallelization.

In order to implement a hybrid parallelization we have to deal with a data hazard problem. We have implemented several strategies to deal with this problem and we have analyzed the performance of these strategies.

Although we had promising results in a test case scenario, a pure MPI version is still better for a real case scenario. Nevertheless, an unsafe hybrid parallelized version of the code beats the pure MPI version in every studied case. Therefore we conclude that we could go deeper in this direction in further studies. If we had the particles sorted by their position in one coordinate, we could just avoid the data hazard just by selecting which threads deal with which particles.

References

1. K. Bennet, C. Brady, H. Schmitz, C. Ridgers, Developers Manual for the EPOCH PIC codes. University of Warwick

Optimization of Lattice QCD kernels for Blue Gene/Q

Christian Jost

Helmholtz-Institut für Strahlen- und Kernphysik
Nussallee 14-16
53115 Bonn

E-mail: jost@hiskp.uni-bonn.de

Abstract:
In this project the QDP++ library of the USQCD software package was optimized for the Blue Gene/Q supercomputer. The sublibrary `libintrin` was recoded and works correctly. Due to compiler problems with the used templates, the C++ was recoded but could not be compiled. While the new library works correctly the full integration of the library into QDP++ could not be completed within this project. Due to the limitation of the tests not every aspect of the code could be tested thoroughly, but a problem with the data alignment in memory was found and fixed. The amount of data used was very small and it can be assumed that the performance of the code will increase with larger problem sizes due to better usage of the resources.

---

1 Introduction

1.1 Quantum Chromodynamics

The Standard Model of Particle Physics describes quarks as the basic building blocks of hadrons. Of the four known interactions, the strong interaction is the strongest within hadrons. The field theory describing the strong force at this level is called quantum chromodynamics or QCD. Each strong interacting particle has a color charge which has to be conserved in the interaction. In contrast to quantum electrodynamics, the field theory of electromagnetic interactions, QCD has three charges, red, blue and green. The hadrons observed in nature are always colorless, which means they consist of color and anti-color pairs or have an equal amount of all colors or anti-colors. The former are called mesons, the latter baryons or anti-baryons, depending on their content. This phenomenon is called color confinement. Another feature of QCD, that other field theories do not have, is asymptotic freedom. That is that the strong interaction becomes stronger as the energy of the interacting particles decrease [1]. This phenomenon makes some usual approaches for computing, e.g. physical observables, unsuitable. To calculate basic properties of particles, such as the mass, the calculations are normally done with the particle being at rest. Then the interactions that need to be calculated are small and can be calculated with perturbation theory. In case of QCD this is not possible since the interaction becomes stronger as the energy decreases. One ansatz to solve this problem is to discretize QCD and solve it numerically.

This ansatz is called Lattice QCD or LQCD. The calculation of the lattice still yields a number of problems. The lattice has to be on the one hand fine grained to make precise calculations, on the other hand the volume of the lattice has to be large enough to neglect finite size effects. This leads to the need of high performance computing to be able to do the calculations within a reasonable amount of time. Although with new supercomputers at the petascale there is a lot of computational power available, it is usually easier to calculate the same problems for different higher quark masses and extrapolate the results to the physical mass scale.

1.2 Blue Gene/Q

The IBM Blue Gene/Q is a modern supercomputer that was released in 2011. It is the third generation of IBM's Blue Gene Series and as of June 2012 the fastest supercomputer of the world[1]. At the same time it is a very energy efficient system and also tops the Green500[2]. At the Jülich Supercomputing Center 8 racks of a Blue Gene/Q machine, called JUQUEEN, were set up in May 2012 and another 20 racks should be available in October 2012. Each rack consists of 2 midplanes having 16 node cards each. Each node card includes 32 compute cards which in turn consist each of one chip module and 16 GB DDR3 memory. The chip is a 64-bit PowerPC A2 chip with 18 cores at 1.6 GHz. Only 16 of the cores can be addressed directly, a seventeenth manages threading and I/O operations. The last core is a backup for not working cores. The chip integrates also a chip-to-chip network. The network setup is a 5D-torus so that each chip has 10 neighbors for fast access. This makes the system very scalable and well suited for LQCD calculations [2, 3].

Each core is capable to run 4 hardware threads at the same time adding up to a total number of 64 threads. Furthermore each core has a quad floating point unit capable of operations on 4 double precision values simultaneously. The chip is designed to work with atomic operations and allows transactional memory and speculative execution, but these features were not used in this project.

For the quad floating point unit a special instruction set is used, the Quad Processing eXtension (QPX). These instructions cover load and store operations, arithmetic operations, logical operations and permutation operations. A subset of the commands interprets the four doubles as two complex numbers and allows complex multiplication operations. To show the power of QPX instructions, a simple add function written with normal C code and with C code enhanced with QPX instructions was written. Table 1 shows the number of cycles each function needs to add two vectors of different sizes. The starting size of the vector was four doubles and was doubled in each step. The measurement was done with 100 iterations and with 10,000 iterations to be able to extrapolate the overhead if needed. The results show that it is very efficient for large data sets to make use of QPX instructions. With a vector size of 512 doubles, the QPX instruction code is more than twice as fast, at 4096 doubles almost more than 2.7 times as fast.

[1] Top500, `http://top500.org/lists/2012/06`
[2] Green500, `http://www.green500.org/?q=lists/green201206`

iterations	size	no QPX	QPX	ratio
100	4	575572	540391	1.07
	8	594192	548448	1.08
	16	635658	571353	1.11
	32	720082	601528	1.20
	64	890426	661476	1.35
	128	1245970	781679	1.59
	256	1915483	1022301	1.87
	512	3300038	1502493	2.20
	1024	6274492	2563134	2.45
	2048	11850690	4557018	2.60
	4096	23154936	8502389	2.72
10000	4	57362000	53969231	1.06
	8	59398608	54800448	1.08
	16	63480054	57079741	1.11
	32	71940682	60080728	1.20
	64	88960826	66050116	1.35
	128	124500368	78029515	1.60
	256	191342083	102080445	1.87
	512	329797494	150080509	2.20
	1024	627412686	256014010	2.45
	2048	1185176947	455377872	2.60
	4096	2315082450	850065514	2.72

Table 1: Comparison of the number of cycles used by a simple add function coded without using QPX instructions and coded with QPX instructions. The measurement was done using IBM's hpm library. At a vector size of 512 doubles the code with QPX instructions is more than twice as fast as the code without QPX instructions, at 4096 doubles it is more than 2.7 times as fast.

2 QDP++

2.1 Implementation

A whole package of software related to LQCD has been developed by the USQCD community[3]. The package is divided into three layers. The bottom layer consists of libraries concerned with message passing between processes (QMP), threading (QMT) and linear algebra (QLA). All other software of the package is built on top of this layer. The middle layer implements mainly all mathematical structures and QCD objects. QCD Data Parallel (QCD in C and QCD++ in C++ [4]) belongs to this layer. The top layer is an assembly of applications, for example the software chroma.

In this project QDP++ should be optimized for Blue Gene/Q. The software is coded in C++ and relies on template metaprogramming. Through the template structure a strong hierarchy is imposed on objects. This structure ensures the correct behavior of spin and color components of objects and the application of complex arithmetic operations where needed. The structure of a lattice object is imple-

[3] `http://usqcd.jlab.org/usqcd-software/`

mented for example in the following way:
`typedef OLattice<PScalar<PColorMatrix<RComplex<float>,NC> > > LCM`
This `LCM` is a lattice object, which transforms as a scalar under spin transformations, as a `NCxNC` matrix under color transformations, where `NC` is the number of colors, and has complex entries. This allows the code to take care of how to do operations on two objects without needing to loop over entries or lattice points. Therefore the user does not need to know how to implement the operation. The addition of two `LCM a` and `b` can then be written as `LCM c = a+b`, for example.

In order for the code to be efficient on a wide range of architectures template specializations are used. Apart from generic templates, there are, for example, templates that use SSE instructions to speed up the code. The templates are specialized in such a way that both SSE2 and SSE3 can be used, depending on the architecture. To have a reference, QDP++ was build on a local workstation using both SSE2 and SSE3 instructions and running all supplied tests to make sure the built worked correct. To further test QDP++ chroma was built on top of QDP++ and the chroma tests were executed. All of the tests tried passed.

2.2 Adaption

JUQUEEN provides two compilers, the GCC compiler, version 4.4.6, and the IBM compiler XL C/C++, version 12.1. The XL compiler is the only compiler that is able to handle QPX instructions, but there is no specialization using these instructions. A first try to compile QDP++ with XL failed because it seems XL cannot handle the template structure of QPD++. The compilation with GCC worked, but the test failed to build because SSE instructions are used in the tests.

QDP++ uses a sublibrary called `libintrin` where basic, optimized functions are stored. This library is coded completely in C code. Therefore in a first step only this library was adapted to JUQUEEN. For the compilation of QDP++ GCC is used for the C++ part and XL for the C part. To make QDP++ aware of the adapted library, now called `libqpx`, a new configure flag was introduced and the makefiles were changed accordingly.

The complete list of functions can be seen in table 2. They all operate on the following predefined data types:

- `su3_vector`, a complex vector with three entries,
- `su3_matrix`, a complex (3×3) matrix,
- `wilson_vector`, a vector of `su3_vector`s with dimension four and
- `half_wilson_vector`, a vector of `su3_vector`s with dimension two.

The functions cover addition of vectors, multiplications between vectors and matrices, vectors and adjacent matrices, matrix-matrix operations, projection of a vector onto another and the simultaneous subtraction of four vectors from a fifth. To ensure that the ported functions produce the anticipated results the functions were called with sample data. The results were then compared with the results from the local, unchanged build. This test was later extended to enable the benchmarking of the library.

During the benchmarking another problem showed up. The QPX load instruction looks for aligned

name	description
add_su3_vector	sum of two vectors
mult_adj_su3_mat_4vec	multiplication of a vector with 4 adjoint matrices and storage of the results in 4 different vectors
mult_adj_su3_mat_hwvec	multiplication of a half wilson vector with an adjoint matrix
mult_adj_su3_mat_vec_4dir	multiplication of a vector with 4 adjoint matrices and storage of the results in 4 different vectors given as array
mult_adj_su3_mat_vec	multiplication of an adjoint matrix with a vector
mult_su3_an	multiplication of an adjoint matrix with another matrix
mult_su3_mat_hwvec	multiplication of a half wilson vector with a matrix
mult_su3_mat_vec	multiplication of a matrix with a vector
mult_su3_mat_vec_sum_4dir	sum of the multipliction of four matrices with four different vectors
mult_su3_na	multiplication of a matrix with an adjoint matrix
mult_su3_nn	multiplication of two matrices
scalar_mult_add_su3_matrix	sum of a scaled matrix and another matrix
scalar_mult_add_su3_vector	sum of a scaled vector and another vector
su3_projector	projection of one vector onto another
sub_four_su3_vecs	subtraction of four vectors from a fifth

Table 2: The functions contained in the library `libqpx` that were adapted to JUQUEEN and their descriptions. A self written test compared the results with the results of a unchanged build on a local workstation to ensure correctness.

data. Since `libqpx` works with single precision complex numbers, the alignment should is 16 bytes. The `su3_vectors` contain 3 complex numbers or 6 floats, they are 24 bytes long. The `su3_matrix` has 9 complex numbers and has a length of 72 bytes. The way the functions are called it cannot be guaranteed that the alignment of the input data is 16 bytes as required by the load instruction. There are several solutions to this problem that were discussed. First of all the function could be modified to have an aligned version and an unaligned version and checks which code will be executed at run time. Instead of modifying the whole code another option would be to assume that either one or two elements are not 16-byte aligned for an `su3_vector` and deal with these possibilities separately. Another solution could be that all data could be packed within a 16-byte alignment which would increase the data volume but works always. The data increase is 33% for the vector and approximately 10% for the matrix. Our solution to this was to load only two floats at a time which requires an alignment of 8 bytes. Since the alignments of `su3_vector` and `su3_matrix` are a multiple of eight, this seems to work and the tests indicate correct behavior without crashing. The disadvantage of this solution is the high number of QPX instructions used.

3 Benchmarking

3.1 Hardware Counters

On most modern chips hardware counters are implemented. These counters can count events that happen either in the core, the network or the memory. The number of counters depends on the specific architecture and not all events can be recorded at the same time. The recordable events can be divided into two subgroups. The general group of events can be recorded on nearly every chip. This could be events like the total number of completed instructions or the number of cycles needed for the completion of a task. The second group of events is specific to every architecture. On JUQUEEN the number of completed QPX instructions can be recorded, for example. These events can help to identify problems and bottlenecks in the code. For the benchmarking of the library, only core events and memory events were recorded.

To record the events the IBM library hardware performance monitor (hpm) was used [5]. The library needs to have MPI[4] initialized before the initialization of hpm and does the instrumentation automatically. This results in an overhead that is recorded as well. To extrapolate the overhead of a call to the `libqpx` functions they were wrapped in a for loop and called between 1 and 1,000,000 times. All measurements were done with only one thread per node. If the coding was effective the benchmark should show high performance. This is a result of a minimal number of instructions and low number of cycles. At the same time the performance could be lowered if the memory would not be used to its full extend. A high number of cache misses would indicate ineffective cache usage.

3.2 Results

The complete library `libqpx` was adapted to JUQUEEN and the test showed that everything works correctly. The benchmarking was done once without compiler optimization and once with level three compiler optimization. There are notable differences between these two cases. We show the results for the function `add_su3_vector` as an example. The code is shown in Figure 1.

The number of cycles needed for one run decreases with a factor of approximately 4.7 between no optimization, needing 230 cycles, and level three compiler optimization, needing 49 cycles, as it can be seen in Figure 2a. There is an overhead of approximately 9000 cycles, which is most probable due to the measurement process itself and the surrounding for loop. At the same time the rate of floating point operations per second (FLOPS) increases from about 60 MFLOPS to approximately 260 MFLOPS at maximum which is shown in Figure 2b. The floating point operations per second F are calculated with the following formula

$$F = \frac{\#\,\text{operations}}{\#\,\text{cycles}} \cdot f \tag{1}$$

where f is the chip's operation frequency, which is 1.6 GHz. The number of operations performed is six complex additions, the number of cycles per function call are shown in Figure 2a. At maximum efficiency eight floating point operations are performed per cycle, resulting in a theoretical maximal performance of $F_{\text{max}} = 12.8$ GFLOPS. In this test the performance is therefore only 2% of the theo-

[4]Message Passing Interface, `http://www.mcs.anl.gov/research/projects/mpi/`

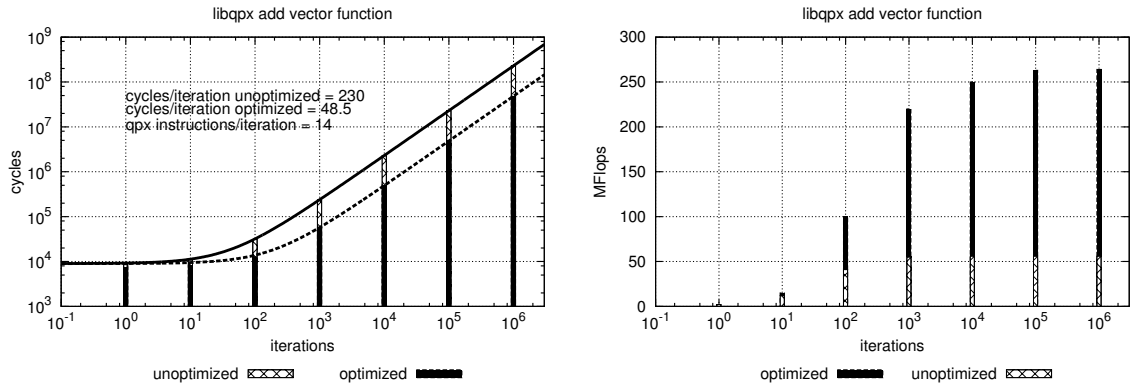
```
void
qpx_add_su3_vector(su3_vectorf *aa, su3_vectorf *bb, su3_vectorf *cc)
{
  /* QPX Variables */
  vector4double xmm2, xmm3, xmm0, xmm1;

  xmm0 = vec_ld2a(0L, (float*) &( (aa)->c[0] ) );
  xmm1 = vec_ld2a(0L, (float*) &( (aa)->c[1] ) );
  xmm0 = vec_sldw(xmm0, xmm1, 2);
  xmm1 = vec_ld2a(0L, (float*) &( (aa)->c[2] ) );
  xmm2 = vec_ld2a(0L, (float*) &( (bb)->c[0] ) );
  xmm3 = vec_ld2a(0L, (float*) &( (bb)->c[1] ) );
  xmm2 = vec_sldw(xmm2, xmm3, 2);
  xmm3 = vec_ld2a(0L, (float*) &( (bb)->c[2] ) );
  xmm0 = vec_add( xmm0, xmm2 );
  xmm1 = vec_add( xmm1, xmm3 );
  vec_st2a(xmm0, 0L, (float*) &( (cc)->c[0] ) );
  vec_sldw(xmm0, xmm0, 2);
  vec_st2a(xmm0, 0L, (float*) &( (cc)->c[1] ) );
  vec_st2a(xmm1, 0L, (float*) &( (cc)->c[2] ) );
}
```

Figure 1: The source code of the function `add_su3_vector` of the library `libqpx` that was adapted for JUQUEEN.

retical maximal performance if the code is optimized. One reason for this low performance is that only every second cycle a QPX instruction can be scheduled per hardware thread. Therefore with at least two threads per core this problem would vanish. There are 14 QPX instructions in the code and therefore when operating with one thread the core spends 21 cycles or roughly 42% waiting or doing other instructions. Since only very little data is used and it fits into the L1 cache, these numbers might change quite significantly with other configurations. Due to the small amount of data there are no big amounts of cache misses recorded, which can change for larger problem sizes. Since the test was only supposed to test for correctness of data, testing with more data has not been done. Since all data is stored consecutively in memory, the data prefetch should work good enough that the run time increase due to the larger problem size should not be too great. The total number of QPX instructions does not change with the optimization levels and is the same number as in the code.

Although no explicit integer arithmetic operations were made in the code, the data shows that integer operations are executed. This is shown in Figure 3. There is some overhead of approximately 350 operations, most probable due to the measurement process and the for loop that envelops the function call. On average the unoptimized code has 13 integer operations while the optimized code has only 6. In comparison to the 14 QPX instructions done in the function there is a lot of integer arithmetic done. This is probably due to the measuring process and the for loop wrapped around the function call. The number of conditional and unconditional branching, as shown in Figure 4, is within the expectation. Since the function is called within another function, there should be at least one conditional branching operation and one unconditional branching operation per function call. The reason for the slightly higher number of unconditional branching operations in the optimized code is not known.

An overview over the number of used QPX instructions in the different functions is shown in table 3.



(a) cycles (b) MFLOPS

Figure 2: The performance of the function `add_su3_vector` compiled without compiler optimization and with level three compiler optimization. The performance is significantly better with compiler optimization. The time a function call needs was measured within a for loop. The number of cycles as a function of the iterations is shown in (a). The performance as a function of the iterations is shown in (b). The number of cycles needed decreases from 230 to approximately 49 per call and the performance rises from 60 MFLOPS to 260 MFLOPS. The data was produced using one thread on a JUQUEEN node with very small data sets and might change significantly for other setups. From (a) it can be seen that there is an overhead of approximately 9000 cycles, including the cycles due to the measurement process and a for loop wrapped around the function call.

The table shows the total number of QPX instructions, the number of load and store instructions, and the number of instructions that does the actual arithmetic operations, including the shuffle operations that are due to the alignment workaround.

4 Summary

The adaption of QDP++ code to the Blue Gene/Q architecture and instruction set proved to be trickier than anticipated. One of the major problems was the inability to compile C++ templates with QPX instructions. Until the end of this project no working solution was found. The second major problem had to do with the alignment of data in memory. The data is 24-byte or 72-byte aligned, depending on the data type, while the QPX instructions needed the data to be aligned as multiples of 16. The issue could be worked around with loading and storing only two floats at a time, requiring and 8-byte alignment and shuffling the data into the correct registers. This leads to a greater overhead and a decreased performance. Nevertheless a self-written test showed that the functions of the basic sublibrary worked as expected and the sublibrary was integrated into the package.

5 Acknowledgement

I would like to thank Dr. Stefan Krieg and Prof. Dr. Dirk Pleiter for their support and advice during this project as well as Mathias Winkel and Ivo Kabadshow for the organization of the program. A big

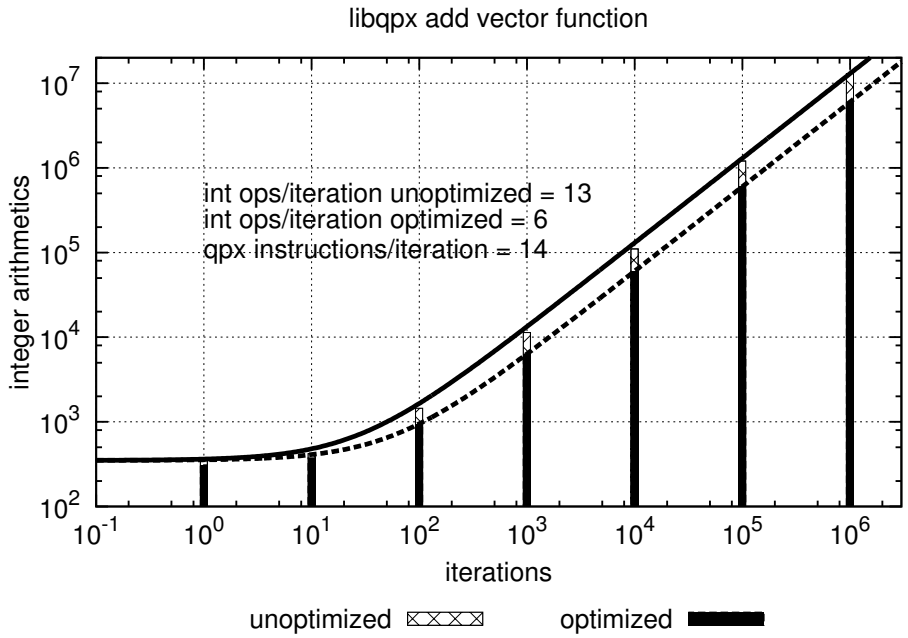
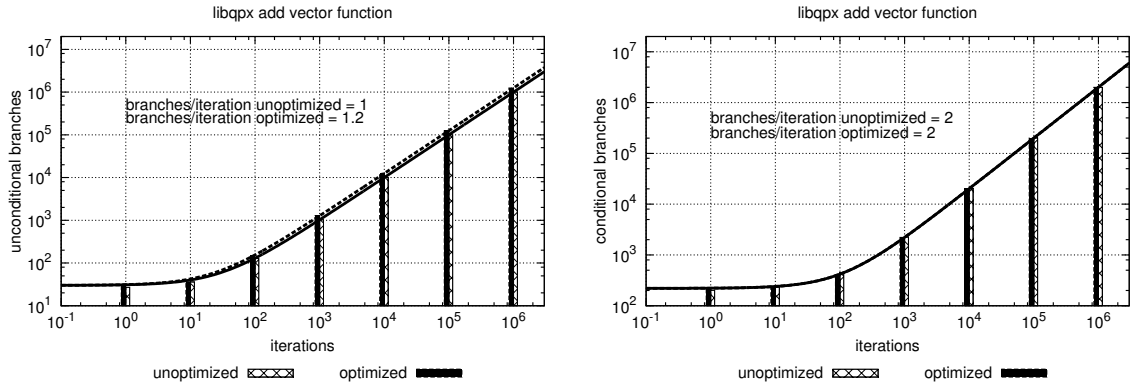


Figure 3: The number of integer arithmetic operations executed in function `add_su3_vector`. The plot shows the measurement done without compiler optimization and with level three compiler optimization. In the optimized version the number of integer arithmetic operations is halved. There is some overhead probably due to the measurement itself and a for loop that envelops the function call.

thank you to Michael Knobloch for his help with PAPI and Scalasca, to Thorsten Hater for his help with Scalasca and to Andrea Nobile for his help with hpm and providing the tools to extract interesting data from the logs. This all would have been very boring without the other students and their part in this exciting program.

References

1. F. Tekin, R. Sommer, U. Wolff and Alpha Collaboration. *The running coupling of QCD with four flavours*. Nucl. Phys. B, 2010 Nov; 840:114-128, [arXiv:1006.0672v1].
2. P. Vezolle. *IBM Blue Gene/Q architecture and system software overview* [Internet]. Presented at: Workshop "Introduction to Blue Gene/Q" 2012 May, [cited 11.10.2012]. 54 p. Available from: `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Documentation/Documention_node.html`.
3. R. Haring et al. *The IBM Blue Gene/Q compute chip*. IEEE Micro, 2012 Mar/Apr; 32(2):48-60.
4. R. Edwards, B. Joo. *The Chroma Software System for Lattice QCD*. Nucl. Phys. Proc. Suppl., 2005 Mar; 140:832-834, [arXiv:hep-lat/0409003].
5. M. Gilge et al. *IBM System Blue Gene solution: Blue Gene/Q application development* [Internet]. IBM Redbook; 2012 Aug [cited 11.10.2012]. 160 p. Available from: `http://www.redbooks.ibm.com/abstracts/sg247948.html`.



(a) unconditional (b) conditional

Figure 4: The branching of the function `add_su3_vector` compiled without compiler optimization and with level three compiler optimization. Since the function is called within another function there should be two conditional branching operations, in which both the optimized and the unoptimized code agree. Where the unconditional branching operation is coming from is not clear, neither why the unoptimized code is better.

function	number of instructions			
	total	arithmetic	load	store
qpx_add_su3_vector	13	4	6	3
qpx_mult_adj_su3_mat_4vec	160	130	18	12
qpx_mult_adj_su3_mat_hwvec	44	31	7	6
qpx_mult_adj_su3_mat_vec_4dir	199	138	49	12
qpx_mult_adj_su3_mat_vec	46	37	6	3
qpx_mult_su3_an	91	70	12	9
qpx_mult_su3_mat_hwvec	44	31	7	6
qpx_mult_su3_mat_vec	172	136	24	12
qpx_mult_su3_mat_vec_sum_4dir	46	37	6	3
qpx_mult_su3_na	92	71	12	9
qpx_mult_su3_nn	92	71	12	9
qpx_scalar_mult_add_su3_matrix	44	17	18	9
qpx_scalar_mult_add_su3_vector	14	5	6	3
qpx_su3_projector	48	31	8	9
qpx_sub_four_su3_vecs	32	14	15	3

Table 3: The table shows the overview over the different functions of the library `libqpx` and the number of QPX instructions they use respectively. The second column shows the total number of instructions, the third column the number of arithmetic operations including the shuffling operations to assemble the data into the correct registers. The last two columns show the number of load and store instructions, respectively.

Graph 500 benchmarking using flash memory cards

Tommaso Zanca

University of Ferrara
Faculty of Science, Department of Physics
Via Saragat 1, 44122 Ferrara, Italy

E-mail: tommaso.zanca@student.unife.it

Abstract:
The performance of supercomputers is measured using benchmarks, where a large amount of data are computed by the machine. Graph 500 is a benchmark designed for testing the access speed to data that present highly irregular patterns, typical of graph structures. Volatile memory is easily exceeded for this kind of applications, so additional memory from external devices is necessary. Our analysis focused on flash memory cards, which present larger access speed to data than hard disks, and memory dimension of several hundreds of GBytes. The performance measurements have been performed on the computer cluster JUNIORS installed at Forschungszentrum Jülich.

1 Introduction

Using the words of Lord Kelvin: "If you cannot measure it, you cannot improve it". For this purpose benchmarks are used to measure the supercomputer performances. Nowadays the ranking of the most powerful supercomputers is built using the High-Performance Linpack Benchmark. This consists in solving a set of linear equations $A \cdot x = b$ with A being a large, densely populated matrix. The performance metrics is the number of floating-point operations per second (FLOPS), used to construct the Top500 List [1]. The Linpack benchmark deals with a very regular problem, so no conclusions can be drawn with respect to the performance of applications with highly irregular data access, such as graph problems.
Graph 500 is a different type of benchmark that aims on providing kernels which reflect the requirements of applications which deal with graph problems, i. e. with random access to the memory due to the highly irregular data pattern. In this case we have a different performance metrics, namely the number of traversed edges per second (TEPS), that is used to build the Graph 500 List [2]. The huge amount of data is a typical feature for graph problems, and the way how to store all these data is an important aspect to take into account. The volatile memory is the best solution if we want a fast access to data, but its capacity is severely limited. On the other hand, hard disks can offer very large memory dimension, but this time the slow access speed is the limiting factor. Non-volatile flash memory cards are a new kind of memory storage that offers new opportunities, allowing storage dimensions of several hundreds of GBytes, with high I/O operations per second rate (IOPS).

2 Background

We introduce some basic notions about graph theory and real networks, in order to better understand the Graph 500 algorithm.

2.1 Graph theory definitions

First we need to define the mathematical elements that we will use in the next paragraphs:

Definition 1 (Graph)**.** *A **graph** is an abstract representation of a set of objects, called **nodes** (or **vertices**), where some pairs of the nodes are connected by links, called **edges**.*

Definition 2 (Distance)**.** *The **distance** is the number of edges in a shortest path connecting two vertices.*

Definition 3 (Diameter)**.** *The **diameter** is the greatest distance between any pair of vertices.*

Definition 4 (Degree)**.** *The **degree** of a node is the number of edges the node has to other nodes.*

2.2 Real networks

Graphs are a useful mathematical tool to study the features and the behaviour in time of real networks, like the World Wide Web, social networks, protein interactions, etc. Real networks across a wide range of domains present surprising regularities, such as power laws, small diameters, communities, and so on. Here we present the main properties [3].

1. *Degree distribution*
   The degree-distribution of a graph is a power law if the number of nodes N_d with degree d is given by $N_d \propto d^{-\gamma}$ ($\gamma > 0$) where γ is called the power law exponent. Power laws have been empirically found, for example, in the Internet [4], the Web [5, 6], citation graphs [7] and online social networks [8].

2. *Densification power law*
   The relation between the number of edges *E(t)* and the number of nodes *N(t)* in evolving network at time *t* obeys the *densification power law* (DPL), which states that $E(t) \propto N(t)^a$. The *densification exponent a* is typically greater than 1, implying that the average degree of a node in the network is *increasing* over time (as the network gains more nodes and edges). This means that real networks tend to sprout more edges than nodes, and thus densify as they grow [9, 10].

3. *Shrinking diameter*
   The diameter of graphs tends to shrink or stabilize as the number of nodes in a network grows over time [9, 10]. This is somewhat counterintuitive since from common experience as one would expect that as the volume of the object (a graph) grows, the size (i. e. the diameter) would also grow. But for real networks this does not hold as the diameter shrinks and then seems to stabilize as the network grows.

2.3 Kronecker Graph Model

In this section we present a model used to construct a network with the properties mentioned above. A good realistic network generation model is important for extrapolations, hypothesis testing, "what if" scenarios and simulations, when real graphs are difficult or impossible to collect.

The model that we present is called *Kronecker Graph Model* [3], and it is based on a recursive construction. Matrices are the mathematical tool used by this model. At each graph we can associate an *adjacency matrix* defined in this way:

Definition 5 (Adjacency matrix)**.** *An **adjacency matrix** of a finite graph on n vertices is the $n \times n$ matrix where the entry $a_{i,j}$ is 1 if vertex i and vertex j are connected, otherwise $a_{i,j}$ is 0*

We begin with an *initiator* graph K_1, with N_1 nodes and E_1 edges, and by recursion we produce successively larger graphs K_2,K_3,... such that the k^{th} graph K_k has $N_k = N_1^k$ nodes. If we want these graphs to exhibit the densification power law, then K_k should have $E_k = E_1^k$ edges. The procedure used to produce densifying graphs with constant or shrinking diameter, and thereby to match the qualitative behaviour of real networks, is described in terms of the *Kronecker product* of two matrices. The Kronecker product is defined in this way:

Definition 6 (Kronecker product of matrices)**.** *Given two matrices $\mathbf{A} = [a_{i,j}]$ and $\mathbf{B}$ of sizes $n \times m$ and $n' \times m'$ respectively, the **Kronecker product matrix** $\mathbf{C}$ of dimensions $(n \cdot n') \times (m \cdot m')$ is given by*

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \doteq \begin{pmatrix} a_{1,1}\mathbf{B} & a_{1,2}\mathbf{B} & \cdots & a_{1,m}\mathbf{B} \\ a_{2,1}\mathbf{B} & a_{2,2}\mathbf{B} & \cdots & a_{2,m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\mathbf{B} & a_{n,2}\mathbf{B} & \cdots & a_{n,m}\mathbf{B} \end{pmatrix}$$

We then define the Kronecker product of two graphs simply as the Kronecker product of their corresponding adjacency matrices.

Definition 7 (Kronecker product of graphs)**.** *If G and H are graphs with adjacency matrices $A(G)$ and $A(H)$ respectively, then the **Kronecker product $G \otimes H$** is defined as the graph with adjacency matrix $A(G) \otimes A(H)$.*

Iterating the Kronecker product we can produce a growing sequence of matrices:

Definition 8 (Kronecker power)**.** *The **k^{th} power** of K_1 is defined as the matrix $K_1^{[k]}$ (abbreviated to K_k), such that:*

$$K_1^{[k]} = K_k = \underbrace{K_1 \otimes K_1 \otimes \ldots K_1}_{k\ times} = K_{k-1} \otimes K_1$$

Finally we define the *Kronecker graph*:

Definition 9 (Kronecker graph)**.** ***Kronecker graph** of order k is defined by the adjacency matrix $K_1^{[k]}$, where K_1 is the Kronecker initiator adjacency matrix.*

2.4 Graph problems

One of the most common problems in graph theory is the *graph traversal*. It consists in visiting all the nodes in a graph in a particular manner, for example listing all the nodes in an ascending order with respect to their distance from a root node. In the next paragraph we focus on a specific technique that outputs such a list.

2.5 Breadth-first search

The *breadth-first search* (BFS) is a strategy for searching in a graph that aims to expand and examine all nodes by systematically searching through every solution. As mentioned above, the node list (parent array) generated by this algorithm as output is ascending ordered with respect to the distance of the nodes from a starting node, called root node.

This is the algorithm pseudocode:

```
1 initialize an empty queue Q
2 insert the root node in Q
3 while Q is not empty
4   u = top of Q
5   remove u from Q
6   for each neighbour v of u
7     if v has not been visited yet and it is not in Q
8       insert v in Q from below
9   mark u as visited
```

3 Graph 500 benchmark

The intent of *Graph 500 benchmark* is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a graph. In addition to a kernel to construct the graph from the input tuple list, there is one additional computational kernel to operate on the graph. The input values required to describe the graph are:

- *Scale (S)*: the logarithm base two of the number of vertices (N), so we can write $N = 2^S$.
- *Edgefactor*: the ratio between the number of edges of the graph and the number of its vertices.

3.1 Overall benchmark

The benchmark performs the following steps:

1. Generate the edge list.
2. Construct a graph from the edge list (**timed**, kernel 1).
3. Randomly sample 64 unique search keys with degree at least one, not counting self-loops.

4. For each search key:
   a) Compute the parent array using the BFS algorithm (**timed**, kernel 2).
   b) Validate that the parent array is a correct BFS search tree for the given search tree.
5. Compute and output performance information.

3.2 Performance metrics

The performance metrics computed by the benchmark is the *Traversed Edges Per Second* (TEPS). This value is an index of the computational power of the machine on which the benchmark runs. Let $time_{k2}$ be the measured execution time for kernel 2. Let m be the number of edges traversed by the search, counting any multiple edges and self-loops. The performance rate (number of edge traversals per second) is defined as

$$\text{TEPS} = \frac{m}{time_{k2}} \ .$$

4 Performance measurements

4.1 JUNIORS

The Graph 500 benchmark has been performed on *JUNIORS* (JUelich Novel IO Research System). This is a computer cluster installed at Forschungszentrum Jülich. It is based on a x86 architecture and it consists in one *Login Node* and 10 *Compute Nodes* (juniors1 to juniors10). Each node is composed of 2 CPUs, each of which consists of 6 cores, for a total of 12 cores and 48 GBytes of shared memory per node. The JUNIORS cluster is equipped with:

- Hard disk drives
- 2 kinds of flash memory cards:
  - Texas Memory Systems Ramsan 450 GBytes
  - Fusion-IO Duo 320 GBytes

4.2 Benchmark configurations

The Graph 500 benchmark has been executed on juniors5 and juniors6 using the *OpenMP Application Program Interface* parallelized version. The main intent of the measurements is to compare the time execution and the performance metrics TEPS between three different storage memory types:

- Volatile memory
- Hard disk
- Texas Memory Systems Ramsan cards

For the flash memory cards case, the flash memory has been mapped into the process' virtual memory space using the *mmap()* function inside the *xmalloc_large_ext()* routine. The flash cards memory capacity is 450 GBytes.

4.3 Problem size

Figure 1 shows the dimension of generated data (problem size) as a function of the scale parameter. Enabling the external memory devices (hard disk or flash memory card), from scale 25 a large amount of data starts to be written on the external drive. For scale larger than 26, the amount of data exceeds the volatile memory capacity, and the benchmark can be performed only with external memory storage devices. This happens for data dimensions lower than the 48 Gbytes available from the juniors node because the data writing is not well optimized, and more memory space is required than necessary.

Scale	Data size (GBytes)
14	$4.19 \cdot 10^{-3}$
15	$8.39 \cdot 10^{-3}$
16	$1.68 \cdot 10^{-2}$
17	$3.35 \cdot 10^{-2}$
18	$6.71 \cdot 10^{-2}$
19	$1.34 \cdot 10^{-1}$
20	$2.68 \cdot 10^{-1}$
21	$5.37 \cdot 10^{-1}$
22	$1.07 \cdot 10^{0}$
23	$2.15 \cdot 10^{0}$
24	$4.29 \cdot 10^{0}$
25	$8.59 \cdot 10^{0}$
26	$1.72 \cdot 10^{1}$
27	$3.44 \cdot 10^{1}$

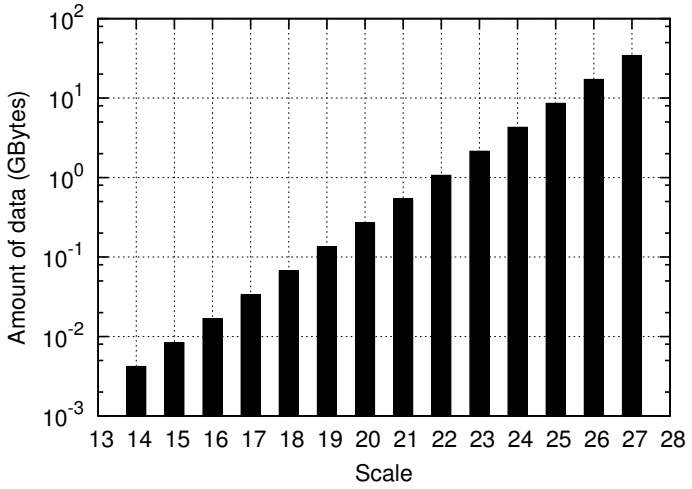


Figure 1: Problem size for different scale values.

4.4 Performance for volatile memory application

4.4.1 Construction time, BFS median time and TEPS measurements

Table 1 shows the execution time for the *construction* and *BFS* routines, and the performance metrics TEPS using the volatile memory. The measurements have been performed for different threads numbers, spanning from 1 to 12. The input value for the scale is 20, generating 268 MBytes of data.

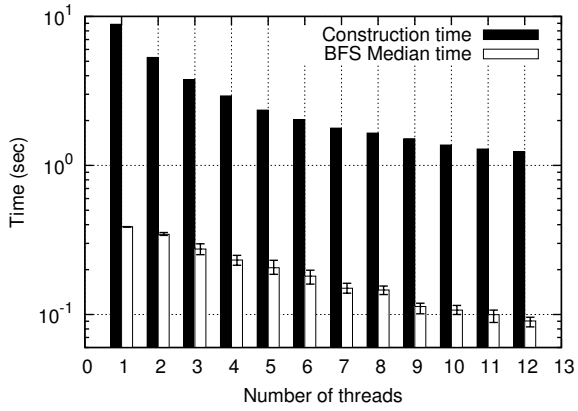
The same measurements have been performed for different scale values, keeping the number of threads equal to 12 (Table 2, Figure 3a and Figure 3b).

Threads number	Construction time (s)	BFS median time (s)	Median TEPS
1	8.86	$3.88 \cdot 10^{-1}{}^{+7.5 \cdot 10^{-4}}_{-9.9 \cdot 10^{-4}}$	$4.32 \cdot 10^{7}{}^{+1.5 \cdot 10^{5}}_{-8.4 \cdot 10^{4}}$
2	5.31	$3.45 \cdot 10^{-1}{}^{+9.4 \cdot 10^{-3}}_{-6.8 \cdot 10^{-3}}$	$4.88 \cdot 10^{7}{}^{+8.7 \cdot 10^{5}}_{-1.4 \cdot 10^{6}}$
3	3.77	$2.75 \cdot 10^{-1}{}^{+2.3 \cdot 10^{-2}}_{-2.4 \cdot 10^{-2}}$	$6.12 \cdot 10^{7}{}^{+5.8 \cdot 10^{6}}_{-4.1 \cdot 10^{6}}$
4	2.93	$2.32 \cdot 10^{-1}{}^{+1.7 \cdot 10^{-2}}_{-1.8 \cdot 10^{-2}}$	$7.30 \cdot 10^{7}{}^{+7.5 \cdot 10^{6}}_{-5.5 \cdot 10^{6}}$
5	2.35	$2.06 \cdot 10^{-1}{}^{+2.5 \cdot 10^{-2}}_{-1.9 \cdot 10^{-2}}$	$8.21 \cdot 10^{7}{}^{+1.1 \cdot 10^{7}}_{-9.0 \cdot 10^{6}}$
6	2.03	$1.81 \cdot 10^{-1}{}^{+1.7 \cdot 10^{-2}}_{-2.1 \cdot 10^{-2}}$	$9.35 \cdot 10^{7}{}^{+1.5 \cdot 10^{7}}_{-8.4 \cdot 10^{6}}$
7	1.78	$1.51 \cdot 10^{-1}{}^{+1.1 \cdot 10^{-2}}_{-1.1 \cdot 10^{-2}}$	$1.13 \cdot 10^{8}{}^{+9.0 \cdot 10^{6}}_{-8.2 \cdot 10^{6}}$
8	1.65	$1.46 \cdot 10^{-1}{}^{+8.5 \cdot 10^{-3}}_{-1.0 \cdot 10^{-2}}$	$1.38 \cdot 10^{8}{}^{+1.0 \cdot 10^{7}}_{-5.9 \cdot 10^{6}}$
9	1.51	$1.13 \cdot 10^{-1}{}^{+5.9 \cdot 10^{-3}}_{-1.2 \cdot 10^{-2}}$	$1.51 \cdot 10^{8}{}^{+1.8 \cdot 10^{7}}_{-8.7 \cdot 10^{6}}$
10	1.37	$1.07 \cdot 10^{-1}{}^{+8.2 \cdot 10^{-3}}_{-6.5 \cdot 10^{-3}}$	$1.59 \cdot 10^{8}{}^{+2.2 \cdot 10^{7}}_{-1.2 \cdot 10^{7}}$
11	1.29	$9.97 \cdot 10^{-2}{}^{+6.8 \cdot 10^{-3}}_{-1.1 \cdot 10^{-2}}$	$1.70 \cdot 10^{8}{}^{+2.6 \cdot 10^{7}}_{-1.1 \cdot 10^{7}}$
12	1.24	$9.04 \cdot 10^{-2}{}^{+5.4 \cdot 10^{-3}}_{-7.8 \cdot 10^{-3}}$	$1.88 \cdot 10^{8}{}^{+1.7 \cdot 10^{7}}_{-1.2 \cdot 10^{7}}$

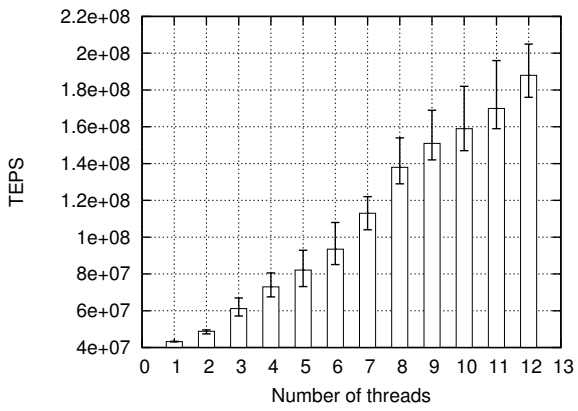
Table 1: Construction time, BFS median time and median TEPS measurements for different threads numbers.

Scale	Construction time (s)	BFS median time (s)	Median TEPS
14	$2.39 \cdot 10^{-2}$	$7.21 \cdot 10^{-4}{}^{+1.1 \cdot 10^{-4}}_{-5.3 \cdot 10^{-5}}$	$3.65 \cdot 10^{8}{}^{+3.1 \cdot 10^{7}}_{-4.6 \cdot 10^{7}}$
15	$3.71 \cdot 10^{-2}$	$1.63 \cdot 10^{-3}{}^{+1.5 \cdot 10^{-4}}_{-1.2 \cdot 10^{-4}}$	$3.25 \cdot 10^{8}{}^{+3.0 \cdot 10^{7}}_{-2.7 \cdot 10^{7}}$
16	$6.73 \cdot 10^{-2}$	$3.49 \cdot 10^{-3}{}^{+3.8 \cdot 10^{-4}}_{-2.3 \cdot 10^{-4}}$	$3.03 \cdot 10^{8}{}^{+1.9 \cdot 10^{7}}_{-2.9 \cdot 10^{7}}$
17	$1.28 \cdot 10^{-1}$	$7.30 \cdot 10^{-3}{}^{+6.5 \cdot 10^{-4}}_{-6.1 \cdot 10^{-4}}$	$2.92 \cdot 10^{8}{}^{+2.5 \cdot 10^{7}}_{-2.3 \cdot 10^{7}}$
18	$2.44 \cdot 10^{-1}$	$1.42 \cdot 10^{-2}{}^{+1.9 \cdot 10^{-3}}_{-9.2 \cdot 10^{-4}}$	$2.99 \cdot 10^{8}{}^{+2.3 \cdot 10^{7}}_{-3.7 \cdot 10^{7}}$
19	$5.03 \cdot 10^{-1}$	$3.18 \cdot 10^{-2}{}^{+2.7 \cdot 10^{-3}}_{-3.3 \cdot 10^{-3}}$	$2.65 \cdot 10^{8}{}^{+3.1 \cdot 10^{7}}_{-2.1 \cdot 10^{7}}$
20	$1.04 \cdot 10^{0}$	$8.07 \cdot 10^{-2}{}^{+5.1 \cdot 10^{-3}}_{-8.5 \cdot 10^{-3}}$	$2.09 \cdot 10^{8}{}^{+2.8 \cdot 10^{7}}_{-1.2 \cdot 10^{7}}$
21	$2.28 \cdot 10^{0}$	$2.52 \cdot 10^{-1}{}^{+2.0 \cdot 10^{-2}}_{-2.5 \cdot 10^{-2}}$	$1.35 \cdot 10^{8}{}^{+2.3 \cdot 10^{7}}_{-1.1 \cdot 10^{7}}$
22	$4.29 \cdot 10^{0}$	$7.38 \cdot 10^{-1}{}^{+8.1 \cdot 10^{-2}}_{-1.2 \cdot 10^{-1}}$	$9.19 \cdot 10^{7}{}^{+1.8 \cdot 10^{7}}_{-9.2 \cdot 10^{6}}$
23	$9.04 \cdot 10^{0}$	$2.13 \cdot 10^{0}{}^{+3.5 \cdot 10^{-1}}_{-5.2 \cdot 10^{-1}}$	$6.46 \cdot 10^{7}{}^{+2.1 \cdot 10^{7}}_{-1.0 \cdot 10^{7}}$
24	$1.87 \cdot 10^{1}$	$5.07 \cdot 10^{0}{}^{+5.4 \cdot 10^{-1}}_{-1.1 \cdot 10^{0}}$	$5.39 \cdot 10^{7}{}^{+1.5 \cdot 10^{7}}_{-5.8 \cdot 10^{6}}$
25	$3.83 \cdot 10^{1}$	$1.06 \cdot 10^{1}{}^{+1.5 \cdot 10^{0}}_{-2.2 \cdot 10^{0}}$	$5.09 \cdot 10^{7}{}^{+1.5 \cdot 10^{7}}_{-6.1 \cdot 10^{6}}$
26	$7.67 \cdot 10^{1}$	$2.17 \cdot 10^{1}{}^{+3.7 \cdot 10^{0}}_{-2.3 \cdot 10^{0}}$	$5.03 \cdot 10^{7}{}^{+5.8 \cdot 10^{6}}_{-5.9 \cdot 10^{6}}$

Table 2: Construction time, BFS median time and median TEPS measurements for different scale values.

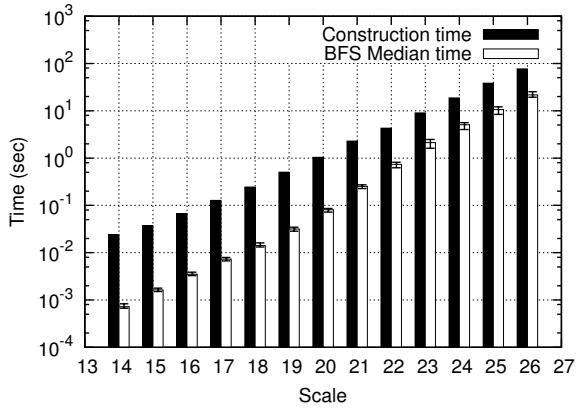


(a) Construction time and BFS Median time

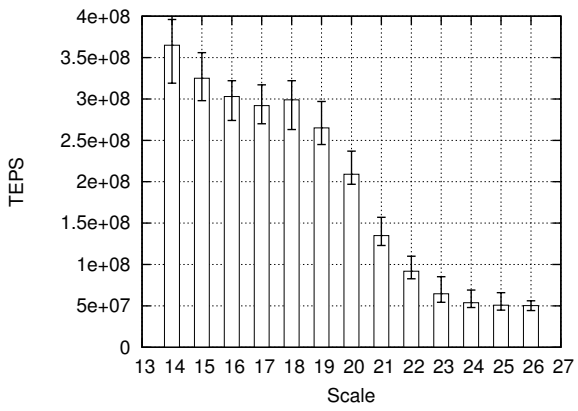


(b) Performance metrics TEPS

Figure 2: Measurements for different threads numbers.



(a) Construction time and BFS Median time



(b) Performance metrics TEPS

Figure 3: Measurements for different scale values.

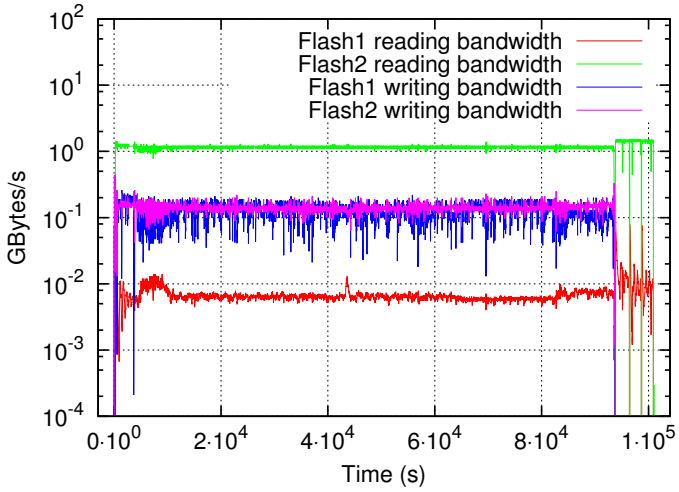


Figure 4: Flash cards bandwidth measurements.

4.5 Scalasca & PAPI

The Graph 500 benchmark has been analyzed using two performance analysis tools: Scalasca and PAPI. Scalasca is a software used to measure and analyze the runtime behaviour of parallel programs, while PAPI is a library that is used in Scalasca to collect *low level* performance metrics of computer systems. We focused on two types of measurements:

- Clock cycles (PAPI_TOT_CYC)
- Level 1 and level 2 cache misses (PAPI_L1_DCM and PAPI_L2_TCM)

Tables 3, 4 and 5 show the measurements for the generation routine (*generate_kronecker_range()*), the creation graph routine (*create_graph_from_edgelist()*) and the BFS routine (*make_bfs_tree()*). We can notice that in the BFS routine the ratio between cache misses and total cycles is bigger than in the other two routines, which is likely due to the increased amount of random access to the memory for the BFS.

generate_kronecker_range()			
SCALE	**PAPI_TOT_CYC**	**PAPI_L1_DCM**	**PAPI_L2_TCM**
16	$2.99 \cdot 10^{11}$	$3.35 \cdot 10^{8}$	$2.55 \cdot 10^{6}$
18	$1.28 \cdot 10^{12}$	$2.58 \cdot 10^{9}$	$9.91 \cdot 10^{6}$
20	$5.47 \cdot 10^{12}$	$8.37 \cdot 10^{9}$	$5.78 \cdot 10^{7}$
22	$2.49 \cdot 10^{13}$	$3.72 \cdot 10^{10}$	$3.51 \cdot 10^{8}$

Table 3: *generate_kronecker_range()* routine.

create_graph_from_edgelist()			
SCALE	**PAPI_TOT_CYC**	**PAPI_L1_DCM**	**PAPI_L2_TCM**
16	$4.34 \cdot 10^{9}$	$2.96 \cdot 10^{6}$	$1.21 \cdot 10^{6}$
18	$1.42 \cdot 10^{10}$	$2.23 \cdot 10^{7}$	$4.87 \cdot 10^{6}$
20	$6.05 \cdot 10^{10}$	$1.03 \cdot 10^{8}$	$2.37 \cdot 10^{7}$
22	$2.54 \cdot 10^{11}$	$3.81 \cdot 10^{8}$	$3.41 \cdot 10^{8}$

Table 4: *create_graph_from_edgelist()* routine.

make_bfs_tree()			
SCALE	**PAPI_TOT_CYC**	**PAPI_L1_DCM**	**PAPI_L2_TCM**
16	$4.57 \cdot 10^{9}$	$1.51 \cdot 10^{7}$	$1.07 \cdot 10^{7}$
18	$1.49 \cdot 10^{10}$	$6.14 \cdot 10^{7}$	$4.56 \cdot 10^{7}$
20	$3.76 \cdot 10^{10}$	$3.64 \cdot 10^{8}$	$2.99 \cdot 10^{8}$
22	$1.62 \cdot 10^{11}$	$2.85 \cdot 10^{9}$	$2.49 \cdot 10^{9}$

Table 5: *make_bfs_tree()* routine.

4.6 Performance for different storage devices

After testing the Graph 500 benchmark using only the volatile memory, we enabled hard disk and flash memory cards as external storage memory devices, in order to compare the benchmark performances.

Table 6 shows these measurements for scale 25 and 26. For these scale values it is possible to see execution time differences between the different storage memories, because the data are partially written on the external devices, slowing down the performance. For scales lower than 24, no significant difference appears because data are almost entirely written and read in the volatile memory, while for scales larger than 26 it is not possible to perform the benchmark with only the volatile memory, because data dimensions exceed its capacity.

	Scale	Construction time (s)	BFS Median time (s)	Median TEPS
Volatile memory	25	$3.83 \cdot 10^1$	$1.06 \cdot 10^{1^{+1.5 \cdot 10^0}_{-2.2 \cdot 10^0}}$	$5.09 \cdot 10^{7^{+1.5 \cdot 10^7}_{-6.1 \cdot 10^6}}$
Hard disk	25	$4.71 \cdot 10^2$	$9.42 \cdot 10^{0^{+1.1 \cdot 10^0}_{-1.4 \cdot 10^0}}$	$5.76 \cdot 10^{7^{+9.5 \cdot 10^6}_{-6.1 \cdot 10^6}}$
Flash memory card	25	$1.02 \cdot 10^2$	$9.67 \cdot 10^{0^{+8.8 \cdot 10^{-1}}_{-1.5 \cdot 10^0}}$	$5.62 \cdot 10^{7^{+1.3 \cdot 10^7}_{-4.8 \cdot 10^6}}$
Volatile memory	26	$7.67 \cdot 10^1$	$2.17 \cdot 10^{1^{+3.7 \cdot 10^0}_{-2.3 \cdot 10^0}}$	$5.03 \cdot 10^{7^{+5.8 \cdot 10^6}_{-5.9 \cdot 10^6}}$
Hard disk	26	$3.66 \cdot 10^4$	$2.43 \cdot 10^{3^{+2.5 \cdot 10^3}_{-2.4 \cdot 10^3}}$	$3.34 \cdot 10^{7^{+1.7 \cdot 10^7}_{-3.3 \cdot 10^7}}$
Flash memory card	26	$3.91 \cdot 10^3$	$3.77 \cdot 10^{3^{+3.2 \cdot 10^1}_{-3.7 \cdot 10^3}}$	$1.79 \cdot 10^{7^{+3.2 \cdot 10^7}_{-1.7 \cdot 10^7}}$

Table 6: Different storage devices.

In Table 7 we can analyze in more detail the execution time between the different benchmark routines for scale 26. In this case the comparison is done with 2 flash memory cards enabled simultaneously. The unique very large difference is in the graph creation routine, where a large amount of data are written on the flash cards, causing a significant increase of execution time. For the *BFS* routine no flash card access is observed, which explains the *BFS* timings being the same for the volatile and flash memory version.

Kernel	Execution time for volatile memory (s)	Execution time for 2 flash cards (s)
Graph generation	244	243
Graph construction	78	2620
BFS	24.5 ± 2.5	24.5 ± 5.5
BFS validation	17 ± 2	19.5 ± 0.5

Table 7: Execution time for different routines (scale 26).

4.7 Flash cards bandwidth

The flash cards bandwidth measurements have been obtained using the script *tmsmon.pl*, a simple tool implemented within this project used to count the number of blocks read and written on the flash cards using the card's hardware counters as well as Linux block layer counters. The size of each block on the flash device is 4096 Bytes, and the reading and writing operations are performed always on full blocks, which may result in much more data being read and written than absolutely necessary. The bandwidth values, shown in Table 8, have been calculated taking the ratio between the total amount of data read or written on 2 flash cards and the execution time spent during the *construction* routine. Table 9 shows measurements for different kernels for scale 27. For this scale only three BFSs have been performed, because of the large execution time required. The writing operations are performed only during the *generation* and *construction* routines, while the reading operations are performed during *construction*, *BFS* and *BFS validation* routines. The amount of read and written data is much larger than expected,

which means that the writing on the flash cards blocks is not efficient. In Figure 4 we can see the reading and writing bandwidth for single cards. While the writing operations look similar, the reading ones are significantly different.

Scale	Bandwidth (Bytes/s)	
	read	write
24	0	$3.2 \cdot 10^3$
25	$2 \cdot 10^3$	$8.9 \cdot 10^8$
26	$1.4 \cdot 10^2$	$1.3 \cdot 10^9$
27	$1.2 \cdot 10^9$	$2.7 \cdot 10^8$

Table 8: Aggregate average bandwidth measurements for 2 flash cards.

Kernel	Execution time (s)	Data written (Bytes)	Writing bandwidth (Bytes/s)	Data read (Bytes)	Reading Bandwidth (Bytes/s)
Graph generation	$4.96 \cdot 10^2$	$9.66 \cdot 10^3$	$1.9 \cdot 10^1$	0	0
Graph construction	$9.37 \cdot 10^4$	$2.54 \cdot 10^{13}$	$2.7 \cdot 10^8$	$1.10 \cdot 10^{14}$	$1.2 \cdot 10^9$
BFS	$2.33 \cdot 10^3 \pm 2.5 \cdot 10^2$	0	0	$3.25 \cdot 10^{13} \pm 3.6 \cdot 10^{11}$	$1.4 \cdot 10^9$
BFS validation	$6.07 \cdot 10^1 \pm 1.2 \cdot 10^1$	0	0	$3.43 \cdot 10^9 \pm 1.3 \cdot 10^9$	$5.13 \cdot 10^7 \pm 1.7 \cdot 10^7$

Table 9: Scale 27 performance for 2 flash cards.

4.7.1 FIO

FIO is a benchmark used for IO performance operations. With this software we obtained information about the bandwidth and IOPS for flash cards and hard disk in sequential and random reading and writing, as we can see in Table 10 (Values in brackets are the vendor specifications of the flash memory card). The bandwidth measurements for reading operations obtained from Graph 500 (Figure 4) match very well the FIO values for one of the two flash cards, while for the other flash card the reading bandwidth is two orders of magnitude lower. The writing bandwidths instead are similar for the two flash cards, but they are one order of magnitude lower than the FIO measurements.

	Bandwidth (Bytes/s)				IOPS			
	read		write		read		write	
	seq	rand	seq	rand	seq	rand	seq	rand
Hard disk	$1.12 \cdot 10^8$	$1.13 \cdot 10^8$	$2.56 \cdot 10^6$	$4.23 \cdot 10^5$	$2.81 \cdot 10^4$	$2.81 \cdot 10^4$	$6.30 \cdot 10^2$	$1.08 \cdot 10^2$
Flash card	$1.30 \cdot 10^9$	$1.03 \cdot 10^9$	$8.92 \cdot 10^8$	$8.82 \cdot 10^8$	$3.25 \cdot 10^5$	$2.57 \cdot 10^5$	$2.23 \cdot 10^5$	$2.21 \cdot 10^5$
	$(1.25 \cdot 10^9)$		$(9.00 \cdot 10^8)$			$(3.00 \cdot 10^5)$		$(2.20 \cdot 10^5)$

Table 10: FIO measurements.

5 Conclusions

From the Graph 500 benchmark executions we saw that for scales smaller than 25 the generated data dimension is significantly lower than the volatile memory capacity, and there is no difference between enabled and disabled external storage memory devices, because also in the second case the data are almost entirely read and written in the volatile memory. For scale 25 and 26 it is still possible to perform

the benchmark with only volatile memory. In case of enabled external storage memory, a considerably large amount of data are written on the external drive during the *construction* routine, causing a significant increase of execution time. In these cases the reading operations to the external memory performed during the *BFS* routine are negligible, which results in similar execution times for enabled and disabled external memory. From scale 27 the volatile memory is not sufficient to perform the benchmark, and the external memory becomes necessary. In this case, in addition to writing operations, also the reading operations are considerably large, causing an increase of execution time for the *BFS* routine too. The read and written volumes are much larger than the problem size, which means that the blocks counted in reading and writing operations are just partially full of data, leading to low efficiency. From the bandwidth analysis, the flash cards show larger access speed than hard disks, both in reading and writing operations. Compared to hard disks, flash memory cards represent the best solution when the volatile memory is not sufficient to manage large amounts of data.

6 Acknowledgements

I would like to thank my adviser Prof. Dr. Dirk Pleiter and Dr. Marcus Richter, for guiding me throughout this work experience. Thanks to Mathias Winkel, Ivo Kabadshow and all the JSC staff, for making possible this very interesting and instructive programme. Finally thanks to all the other guest students, for the great time spent together.

References

1. *TOP500 Supercomputer Sites*. Available from: http://www.top500.org
2. *The Graph 500 List*. Available from: http://www.graph500.org
3. J. Leskovec, D. Chakrabarti, J. Kleinberg, C Faloutsos, Z. Ghahramani. *Kronecker Graphs: An Approach to Modeling Networks.* Journal of Machine Learning Research 11 (2010) 985-1042
4. M. Faloutsos, P. Faloutsos, C. Faloutsos. *On power-law relationships of the internet topology.* Proceedings of the Conference on Applications, Technologies, Architectures and Protocols for Computer Communication, pages 251-262, 1999.
5. J. M. Kleinberg, S. R. Kumar, P. Raghavan, S. Rajagopalan and A. Tomkins. *The web as a graph: Measurements, models and methods.* Proceedings of the International Conference on Combinatorics and Computing, 1999.
6. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins and J. Wiener. *Graph structure in the web: experiments and models.* Proceedings of the 9th International Conference on World Wide Web, 2000.
7. S. Redner. *How popular is your paper? An empirical study of the citation distribution.* European Physical Journal B,4:131-134, 1998.
8. D. Chakrabarti, Y. Zhan and C. Faloutsos. *R-mat: A recursive model for graph mining.* SIAM Conference on Data Mining, 2004.
9. J. Leskovec, J. M. Kleinberg and C. Faloutsos. *Graphs over time: Densification laws, shrinking diameters and possible explanations.* Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pages 177-187, 2005.
10. J. Leskovec, J. M. Kleinberg and C. Faloutsos. *Graph evolution: Densification and shrinking diameters.* Transactions on Knowledge Discovery from Data (TKDD), 1(1):2, 2007.

Generating parallel random numbers: As easy as 1, 2, 3?

Artur Strebel

Bergische Universität Wuppertal
Gaußstraße 20
42119 Wuppertal

E-mail: astrebel@studs.math.uni-wuppertal.de

Abstract:
In this paper a general introduction into random number generation is given and some of the most common pseudo-random number generators (PRNGs) are described. Lately these PRNGs have encountered some problems with modern computer architectures. I also present a novel approach to random number generation proposed by John K. Salmon et al. in [1]: The Random123 library. It contains three PRNGs, the first two are based on cryptographic standards (AES and Threefish), the third one taking a new approach. All three PRNGs have excellent statistical properties and produce at least $2^{64} \times 2^{128}$ random numbers while maintaining good performance on both multi core and single core architectures. These PRNGs have been compared with the SPRNG library which is currently used at the Forschungszentrum Jülich.

1 Introduction

A **random number generator (RNG)** produces sequences of numbers which do not seem to follow any pattern thus being "randomly generated". Achieving this goal is a difficult task since a computer as a deterministic machine will not be able to produce real non-deterministic numbers. Therefore **pseudo-random number generators (PRNGs)** were developed. These algorithms have become an important part of many applications such as molecular dynamics or diverse Monte Carlo simulations. Since these applications have been developed to be run on modern multi core architectures it has become crucial for PRNGs to be able to run in parallel as well.

In this paper I present a novel approach to parallel random number generation. First I will provide a basic insight into random number generation by giving basic definitions and quality criteria in section 2. It also covers some currently used pseudo-random number generators. Since they are designed to run on a single core it is necessary to provide schemes to parallelize these PRNGs. This will be described in section 3 revealing also general problems with these PRNGs considering parallelization which lead to the new Random123 PRNGs from Salmon et al. They will prove to have good statistical properties while being on par with the SPRNG library considering CPU time. The big advantage of these novel PRNGs is their intuitive setup and minimalistic memory usage. This library is discussed in section 4 along with the SPRNG library which implements parallelized versions of some conventional PRNGs.

In section 5 I present results from statistical tests and compare the SPRNG and the Random123 library regarding these statistical properties and run time on multi core architectures. In the end I give recommendations on which PRNG to use depending on the application.

2 Random number generation

The first part of this section focuses on giving a general insight on random number generation and theoretical measurements evaluating the quality of the produced numbers. The second part will provide examples for some established pseudo-random number generators.

2.1 Basic Definitions

There are two different approaches on random number generation. The first approach are *physical* random number generators which generate real random numbers by measuring random sources from physical processes. These processes may be radioactive decay or atmospheric noise. Sequences produced by a physical random number generator are non-reproducible and generated fairly slow.
Since it is extremely important for scientific simulations to have reproducible data, physical random number generators are not suited for scientific computing. For this purpose *pseudo*-random number generators (short: PRNGs) have been designed. They can be defined as a 5-tuple $(\mathcal{S}, \mu, f, \mathcal{U}, g)$, where

- $\mathcal{S}$ is a finite set of *states* s called the *state space*,
- μ is a probability distribution on $\mathcal{S}$ used to select the initial state (*seed*) s_0,
- $f\colon \mathcal{S} \to \mathcal{S}$ is the *transition function*,
- $\mathcal{U}$ is the *output space*,
- $g\colon \mathcal{S} \to \mathcal{U}$ is the *output function*.

It is very natural to set $\mathcal{S} = \mathbb{Z}/n\mathbb{Z}$ with $n = 2^W$ where W is the size (in bit) of the used data type (`float, double,...`). This is mainly for computational purpose since every set can be reduced to its internal bit-representation. Then, with f and g being well defined, it is obvious that the period of a PRNG is bounded by the cardinality of $\mathcal{S}$. Every good PRNG is designed such that it exhausts (nearly) the complete state space. The output space is typically chosen to be $\mathcal{U} = (0, 1)$. This enables the invoking application to scale the random numbers to any interval needed.
This kind of random number generators allow reproducible streams of numbers by invoking the same PRNG with the same seed. The downside of PRNGs is the fact that they are by construction deterministic thus not really “random”. So the question following this observation is how to decide whether a sequence of numbers is "random" or if it follows some obvious pattern. In the following, two criteria for measuring "randomness" are given. The first one focuses on checking whether the random numbers follow an uniform distribution while the second one checks if the generated numbers are in some sense "unpredictable".

Consider n successively generated numbers and interpret them as coordinates in a n-dimensional unit square (assuming $\mathcal{U} = (0, 1)$). Dividing this unit square into k subsquares with equal size and counting

the points per subsquare should lead to a multinomial distribution with an expected value of $\frac{\#points}{k}$. So taking 100 points and assuming $k = 4$ would mean that each subsquare should hold about 25 points in order to pass this test. It can be seen as a generalization of a Bernoulli distribution with success probability $p = \frac{1}{k}$ for every subsquare.
The *spectral test* is a very powerful test which analyzes the lattice structure of certain classes of PRNGs thus giving a good a priori measurement of its quality. This test and the according PRNGs are discussed in the next part.
A more technical criterion is commonly used by cryptographers and is referred to as the *(strict) avalanche criterion*. Given an input s_i and the according output u_i after applying the PRNG the following implication has to hold:

$$\text{Change one single bit in } s_i \Rightarrow \text{Each bit in } u_i \text{ has a } 50\% \text{ probability to change.}$$

This test gives an indicator on how the output is linked to the input. So to put this criterion in other words, every output bit should be affected by every input bit. While this property is far more important for security of cryptographic algorithms it will prove useful in setting up and generating high quality random numbers as seen later on.
So in general a good PRNG should produce numbers which are both uniformly distributed and unpredictable. Another important criterion for randomness is the mutual independence of the generated numbers. Looking at the first class of PRNGs which will be discussed in the next section this criterion will not be met leading to problems when trying to parallelize them.

2.2 Conventional PRNGs

Most of the currently used PRNGs have one general property in common. They use a complex transition function f and a trivial output function g to generate random numbers. Either g is simply the identity making the output space equal to the state space or g is a linear mapping into $(0, 1)$. Both ways, the output function is not fundamentally responsible for the quality of these PRNGs hence we will concentrate on analyzing the transition function f. This function generates the current number s_i by applying a complex function to the last state s_{i-1}. One of the simplest PRNGs is a *Linear Congruential Generator (LCG)*. Because of the simple transition function this generator is fairly fast. It is given by:

$$f(s) = a \times s + b \mod m$$

The parameters a, b, m can be chosen arbitrarily but heavily influence the quality of the generated random numbers. Applying the spectral test to the produced sequence results in a qualifiable lattice structure which can be described with the theorem of Marsaglia:

Marsaglias theorem: Given a LCG with $s_{i+1} = a \times s_i + b \mod m$ and $u_i = \frac{s_i}{m}$, the points defined by a k-tuple $(u_l, \ldots, u_{l+k-1})$ of consecutive random numbers are distributed on a **maximum** of $\sqrt[k]{m \times k!}$ parallel hyperplanes in $\mathbb{R}^k$.

This theorem implies the more or less undesirable fact that LCGs **always** have this hyperplane structure. So keeping this in mind the goal of a good LCG is to achieve this upper bound as good as possible thus

approximating an uniform distribution by choosing appropriate parameters. An example for poorly chosen parameters is the LCG "RANDU" with $a = 65539$, $b = 0$, $m = 2^{31}$. Using Marsaglias theorem with $k = 3$ yields a maximum of 2344 possible hyperplanes while RANDU only reaches 15 hyperplanes. So the distances between these hyperplanes is too big to approximate a uniform distribution.
A more general form of a LCG is the *Multiple Recursive Generator (MRG)*. MRGs use more than one previously generated number to compute the next random number:

$$f(s_i) = \sum_{j=1}^{n} (a_j \times s_{i-j}) + b \mod m$$

Depending on the parameter choice this PRNG tends to have better statistical properties and a longer period than a LCG. An often used special case of a MRG is a *Lagged Fibonacci Generator (LFG)* where each random number s_i depends on exactly two previous states s_{i-l}, s_{i-k} (l and k are called the *lags* of the generator and w.l.o.g. $l > k$), both having multiplier $a = 1$ and $b = 0$.
Presumably the most well known PRNG is the *Mersenne Twister*. Its name comes from the extremely long period of $2^{19937} - 1$ which is a Mersenne prime number. The computation of the random numbers has a couple of steps consisting of XOR and bitwise AND operations. The Mersenne Twister produces 624 numbers in one iteration which leads to equally distributed random number in hypercubes up to dimension 623. On the downside, these numbers have to be stored resulting in a high memory usage of 2,5 kB. So using this PRNG for applications where memory is limited this is not optimal.

3 Going parallel

Considering the development of high performance computing (HPC) which focuses on heavily parallel machines it is only natural to update PRNGs to reflect these changes. So the goal is to run a PRNG on a multi core architecture each producing an unique, independent stream of random numbers. This section presents two approaches to parallelize already existing PRNGs: the substream and the multistream approach. Both are relatively straightforward but both have their own issues when trying to apply to conventional PRNGs.

3.1 Substream approach

The first idea is to split up the state space into disjoint subspaces and assign each to one core. This is a fairly simple approach and requires only one set of good parameters. The big downside of this approach is that many PRNGs do not allow deterministic partitioning meaning it can not be assured that a certain partitioning leads to independent streams. Assuming the state space has been split into two subsets with arbitrary starting states s_k and s_l it may be possible that applying the PRNG on s_k already leads to the state s_l after a few steps thus making the streams dependent. It has also to be taken into account that the state space will of course be smaller on each core. So PRNGs with a rather small period are not optimal for this approach.

A good example for substream parallelization is the Mersenne Twister. Its state space is big enough and the partitioning can be done by executing a rather complicated jump ahead procedure.

3.2 Multistream approach

The other approach takes a look at the set of parameters. As already mentioned, nearly every PRNG has parameters which affect the output stream and the quality of the random numbers. So the idea of the multistream approach is to use different sets of parameters on each core. This approach is easy to set up since every PRNG can be invoked in the same way only using different parameters. This approach needs a PRNG with easy to compute or predefined sets of good parameters. For most PRNGs the set of good parameters is not only fairly limited but also it might be non-trivial to find good parameters or the quality of the numbers has to be examined afterward.
As for conventional PRNGs, LFGs are well suited for the multistream approach. It can be shown that for this PRNG the number of different streams is limited to maximal $2^{(W-1)*(l-1)-1}$. It should be noted that despite the big amount of distinct streams it might be tricky to setup different streams without getting stream dependencies. The Random123 library which will be presented later also favors this multistream parallelization.
If the PRNGs allow both kinds of parallelization it is of course possible to combine them to achieve a better rate of scalability. Again, the Random123 library allows this kind of hybrid parallelization in a very natural and convenient way.

3.3 Parallelizing conventional PRNGs

Now knowing two ways to parallelize a PRNG we can look back at the conventional PRNGs and try to apply these schemes. Starting with the LCGs it is obvious that using the multistream approach is in general more appropriate since the partitioning of the state space is rather difficult to achieve beforehand. Nevertheless, if $b = 0$ the substream approach may be feasible by calculating the k-th random number via $s_k = a^k \times s_0 \mod M$. Depending on the application both approaches can come in handy. If the state space is big enough for the application and a good parameter a is given the substream approach with $b = 0$ can be used. If only a few cores are available using predefined sets of good parameters with the multistream approach may be preferred.
Similar results naturally hold for MRGs and LFGs. A special case of a LFG should be highlighted in this context. Instead of adding the two lagged random numbers the *Multiplicative* LFG uses a multiplication to combine these numbers. This allows again the substream approach as described above.
The Mersenne Twister can be parallelized via the substream approach as pointed out in the respective section. But is has to be kept in mind that the Mersenne Twister requires a big amount of memory which can be even more critical on multi core architectures.
Even though some successful libraries implementing parallel random number generation are available there is one very central property which nearly all conventional PRNGs share. All PRNGs rely on using at least the previously generated random number to compute the next one thus making the PRNG sequential. This property is very undesirable when trying to parallelize any application.

4 Parallel PRNGs

As seen in the last part of the previous section the ultimate goal of parallel number generation is to be able to generate each number independently of all others in order to minimize memory requirement and maximize scalability. This section introduces two libraries which will prove to be well suited for parallel applications. The SPRNG library uses conventional PRNGs and parallelizes them via multistream approach. The Random123 library chooses a completely novel approach using modified cryptographic block ciphers to achieve high quality, easy to set up random numbers. These libraries are presented in this section with giving only a little to none qualification about quality and CPU time. These results will be given in the next section.

4.1 The SPRNG library [3]

The *Scalable Parallel Pseudo Random Number Generator* library (SPRNG) implements five different types of conventional PRNGs.
The first one is a combined generator consisting of a 64-bit LCG and a MRG. Its transition function is given by:

$$\begin{aligned} z(n) &= x(n) + y(n) * 2^{32} \mod 2^{64} \\ y(n) &= 107374182 * y(n-1) + 104480 * y(n-5) \mod 2147483647 \end{aligned}$$

$y(n)$ stays constant for every stream so the 64-bit LCG $x(n)$ defines the different streams. This leads to a period of 2^{219} which the possibility of over 10^8 different streams.

Generator two and three are both simple LCGs with a 48 respectively 64- bit modulus leading to according periods of 2^{48} and 2^{64}. The number of different streams available is in the order of 5×10^6 and 10^8. In order to assure stream independence large prime numbers are used for parameter b. Empirical tests have shown that this method improves the theoretical results presented in [4].

The fourth generator is a Modified LFG. The transition function is a XOR-conjunction of two slightly modified LFGs:

$$\begin{aligned} z(n) &= x(n) \,\mathrm{XOR}\, y(n) \\ X(n) &= X(n-k) + X(n-l) \mod M \\ Y(n) &= Y(n-k) + Y(n-l) \mod M \end{aligned}$$

Note that $x(n) = (X(n) >> 1) << 1$ meaning the last bit is set to zero while $y(n) = Y(n) >> 1$. This modification eliminates some stream correlations which have been noticed in the unmodified version. With default parameters ($k = 861$, $l = 1279$, $M = 2^{32}$) this PRNG has a period of around 2^{1310} with possibly 2^{39648} different streams. As mentioned above setting up different streams while retaining stream independency might be a tricky task. For further information about this problem see [5].

The fifth generator is a multiplicative LFG which has already been mentioned above. The transition function is given by:

$$x(n) = x(n-k) * x(n-l) \mod M$$

Using this PRNG with default parameters ($k = 5$, $l = 17$, $M = 2^{64}$) it achieves around 2^{1008} distinct streams each with a period of 2^{84}. It has the same concern about stream correlations as the previous PRNG.

4.2 Cryptographic block ciphers

Cryptographic block ciphers have already been successfully used for random number generation since their task of providing an output which seems to lack any measurable structure and is highly sensible to slight changes to the input is exactly what is needed for good random numbers. The main reason why they have not been able to come out on top is that they are significantly slower than conventional PRNGs. The Random123 library tackles this problem by trading security arrangements for speed. First of all I will introduce two cryptographic functions which are the basis for the block ciphers.

Definition: A *SP-network* is a diffusive bijection consisting of iterative rounds of blockwise bijective substitutions (*S-boxes*) and permutations (*P-boxes*).

So a SP-network is a bijective mapping providing all dependencies between input and output bits

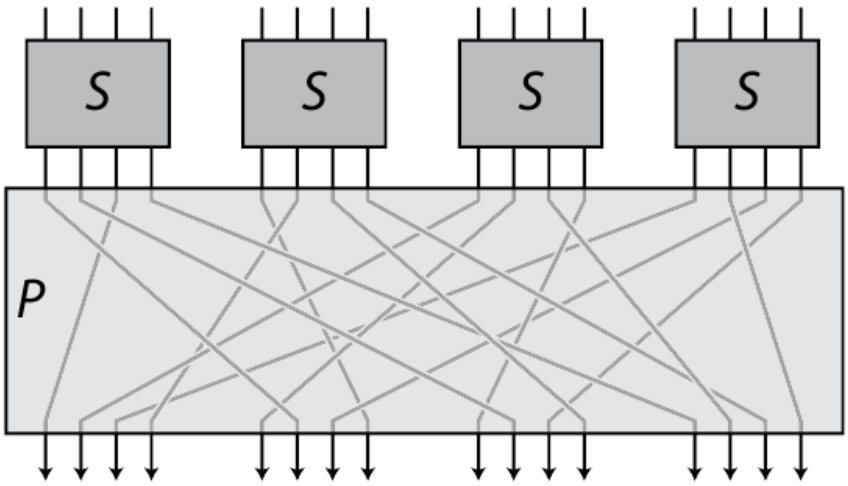


Figure 1: A SP box with four parallel S-boxes and a P-box afterwards. The P-box should distribute the output from each S-box as broad as possible to ensure good diffusion.

needed to meet the avalanche criterion. All block ciphers in this paper can be seen as a SP-network. They differ by using different bijections for the S-boxes. Depending on the SP-box several rounds are needed to provide enough *diffusion*. Diffusion gives a hint how well the output is linked to the input. So a function with good diffusion links every input bit with every output bit thus meeting the avalanche criterion.

Definition: The *Feistel function* is a bijective construction between $2p$-bit inputs from an arbitrary keyed p-bit function F_k and a bijective (keyed) p-bit function B_k and a group operation $\oplus$.

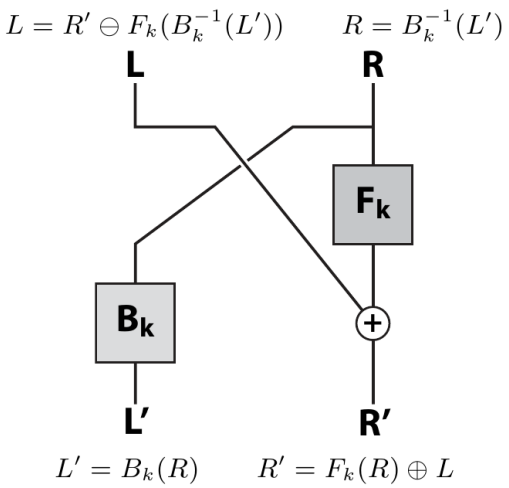


Figure 2: A schematic Feistel function. Given the output one can reconstruct the input by the functions given at the top. $\oplus$ is typically a bitwise AND or XOR.

Feistel functions are often used for S-boxes. It allows the use of non-bijective functions without losing the possibility to decode the output.

The first very well known cryptographic standard is the *Advanced Encryption Standard (AES)*. It is used in many applications for example SSH, WPA2 or PGP. AES is designed as a ten round SP-network using rather complicated SP-boxes. The input message will be encrypted in 128-bit blocks and additionally eleven precomputed 128-bit keys are XOR'ed with the input of each SP-box. The other algorithm is called *Threefish*. This standard is using a 72 rounds SP-network but with much easier and faster S-boxes by only using elementary operations like additions, rotations and XOR operations. As with AES Threefish uses an additional 128-bit key to encrypt its 128-bit input. Both algorithms can be expanded to work on 256-bit words and keys providing even better security.

4.3 The Random123 library [1]

As already mentioned, AES and Threefish are too slow to be used as an efficient PRNG. Therefore it is necessary to modify these algorithms such that the good quality of the random numbers is still given but run faster by a significant factor without taking too much or even any additional memory. This is mostly achieved by reducing the number of rounds. For AES only five rounds are needed to still pass all important statistical tests which will be discussed later. Also the round keys are computed by a variation of the *Weyl*-function. In this case it is a simple addition with a constant which leads to fast computation and no memory overhead. Compared to storing the eleven round keys separately in AES this is a gain of $11 \times \text{sizeof}(\texttt{datatype})$ bit with only a negligible extra CPU cost. This "new" algorithm is called *Advanced Randomization System (ARS)*. The efficiency of this algorithm can be drastically improved if the processor supports AES-NI, a new instruction set for AES encryption

which speeds up the algorithm by a factor of around eight. Having this instruction set is crucial for ARS since otherwise it will significantly fall behind in CPU time compared to other PRNGs. In a similar way, Threefish is sped up by only applying a maximum of 20 SP-boxes although even less are required to guarantee good statistical properties. This algorithm is natively able to split up the output to produce up to 4 random numbers at once which can come quiet handy in some applications. To differentiate this algorithm from Threefish this one is called *Threefry*. The third PRNG in the Random123 library is called *Philox* and uses a fairly simple Feistel function which is extremely well suited for GPUs. This Feistel-function is given by:

$$\begin{aligned} L' =& B_k(R) = (R \times M) \mod 2^W \\ R' =& F_k(R) \oplus L = \lfloor (R \times M)/2^W \rfloor \oplus k \oplus L \end{aligned}$$

It only needs seven rounds and can similar to Threefry produce up to four numbers at a time.

This kind of PRNGs are called *counter-based* because the transition function can be as simple as possible by just being a counter: $f(x) = x + 1$. Since the avalanche criterion is met for these PRNGs even a slight change of the state space will lead to a completely different and independent number. In this way, the generation of a random number is only dependent on the counter and the key eliminating all dependencies between the produced random numbers. We can therefore either use the substream approach to divide the state space in a very natural way by just assigning the first k numbers to the first core, the second k to the second core and so on or we use the multistream approach and run the same counter space on each core and just change the key. Again, changing the key, can be nothing more than just increasing its value by one. If an unique system constant (e.g. the MPI rank) is used as a key the memory for storing the key can be saved. So the parallelization of the Random123 PRNGs is very easy to set up and produces no memory overhead at all.

5 Numerical Results

So, having an overview on the different PRNGs the important question is how good these PRNGs are, considering statistical quality and CPU time. While there is literally an infinite amount of tests for randomness it is important to have a library providing the most important ones in a convenient way. One of these libraries is the TestU01 library which will be described in the next part. Some routines of this library are applied to the PRNGs from SPRNG and Random123. Finally, I produced 10^{12} random numbers with these random number generators and measured the CPU time.

5.1 TestU01 batteries

The TestU01 library [6] implements various tests to measure statistical patterns in the random number stream. There are three test batteries which apply a set of tests on a PRNG and report any suspicious results they find. The first one is *SmallCrush* and includes 15 tests. As the name implies this is a rather small test and checks for the most basic patterns. Failing tests from this battery is a clear sign for statistical flaws in this PRNG and should most times lead to dismissing this PRNG. The second battery is *Crush*. It consists of 96 tests and is a good indicator for the quality of a PRNG. Consequently, the third battery is *BigCrush* and has 106 tests with most of them also being

in Crush but with significantly bigger parameters. Only a very few PRNGs pass the stringent tests of BigCrush.

5.2 Crush results

Here I provide a comparison of the SPRNG and Random123 PRNGs. The numbers in the table indicate the number of tests the according generator failed in the according Crush test. The starred results are literature values.

Generator	SmallCrush	Crush	BigCrush
SPRNG-MRG	5	-	-
SPRNG-LCG48	0	5	8
SPRNG-LCG64	0	1	8
SPRNG-LFG1	0	0	0
SPRNG-LFG2	0	3	4
ARS	0*	0*	0*
Threefry4x32	0	0	0
Philox	0*	0*	0*

As seen from the table all Random123 PRNGs pass every test thus being *crush-resistant*. Except for the MRG from SPRNG all other generators also produce acceptable results.

5.3 CPU time measurements

Another important criterion for a good PRNG is the needed CPU time. In the following I give a direct comparison between the SPRNG and Random123 libraries by measuring the time taken to produce 10^{12} random numbers on up to 1024 cores. These tests have been run on JuRoPA, a computer cluster from the Research Centre Jülich with 2208 compute nodes each having two Xeon X5570 Nehalem-EP quad-core processors at 2.93 GHz[7]. Both libraries have shown to scale nearly perfect for up to 1024 cores. It has to be noted that these 10^{12} numbers have not been tested for statistical properties. Also, even if not in the table it is to expect that ARS will perform about twice as fast as Threefry.

Generator	2^0 cores	2^5 cores	2^{10} cores
SPRNG-MRG	4838	148	4.63
SPRNG-LCG48	4837	149	4.65
SPRNG-LCG64	7260	219	6.87
SPRNG-LFG1	8090	249	7.81
SPRNG-LFG2	8832	263	8.32
Threefry4x32	12875	384	12.03

Looking at the table and combine it with the statistical results we can see a tradeoff between statistical quality and CPU time.

6 Conclusion

In this paper I gave a basic insight into random number generation and its parallelization approaches. I then explained why the current generators have problems with HPC and presented a novel approach to random number generation by using modified cryptographic standards. They are very natural to parallelize and have both good performance and pass stringent statistical tests. Nevertheless, using the optimal PRNG highly depends on the demands of the invoking application and computational resources (memory usage and CPU time). The main criteria are statistical quality of the random numbers, CPU time, memory usage, scalability.
If low memory usage is critical the PRNGs from the Random123 library can be recommended since they produce no memory overhead despite from the space needed for the key and the counter. Also, since different streams are easy to set up and reliably independent the Random123 PRNGs are well suited for highly scalable applications. The SPRNG library is partially better when comparing CPU time but sacrifices some statistical quality for it. But if the invoking application is not susceptible to these statistical flaws using one of the first three PRNGs from SPRNG will result in fast number generation. If the application focuses on statistically flawless numbers then either a LFG from SPRNG or the Random123 library is recommended.

Looking into the future it is probable to see further research into counter-based random number generation since its usage is very intuitive and scales extremely well on multi core architectures.

References

1. John K. Salmon, Mark A. Moraes, Ron O. Dror, David E. Shaw. *Parallel Random Numbers: As Easy as 1, 2, 3*, D.E. Shaw Research, NewYork, NY 10036, USA
2. J.E. Gentle, W.Härdle, Y.Mori. *Handbook of Computational Statistics*, Springer Verlag, 2004
3. M. Mascagni and A. Srinivasan (2000), *Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation*, ACM Transactions on Mathematical Software, 26: 436-461
4. O.E. Percus and M.H. Kalos: *Random number generators for MIMD processors*, Journal of parallel and distributed computing, 6 (1989) 477-497
5. M. Mascagni, S.A. Cuccaro, D.V. Pryor, M.L. Robinson: *A fast, high quality, and reproducible parallel Lagged-Fibonacci pseudorandom number generator*
6. P. L'Ecuyer and R. Simard, *TestU01: A C Library for Empirical Testing of Random Number Generators*, ACM Transactions on Mathematical Software, Vol. 33, article 22, 2007.
7. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/JUROPA_node.html

Design and Implementation of an Experimental Finite Element Solver

Daniel Arndt

University of Göttingen
Institute for Numerical and Applied Mathematics
Lotzestr. 16-18
D-37083 Göttingen, Germany

E-mail: d.arndt@math.uni-goettingen.de

Abstract:
In Finite Element applications it is often desired to have a modular design in order to change different parts of the program easily. However, in the classical assembly approach, e.g., the element type and the physical problem are strongly coupled.
In this project a Visitor Pattern is used to overcome this coupling. It turns out that this is a promising approach for a modular design. All basic functionalities of an Finite Element code were implemented and no serious problem occured.

---

1 Introduction

The Finite Element Method (FEM) is one of the most used techniques to solve partial differential equations numerically, and was awarded to be one of the top 10 Computational Methods of the 20th Century [5]. Especially for domains with complicated geometries FEM can be easily applied, other methods may have some difficulties here.

Since the 1960s FEM is a constantly growing research area. Especially in the last few years some innovative concepts has been introduced, like:

- Extended Finite Element Method (XFEM)
- Discontinuous Galerkin Method (DG)
- Isogeometric Analysis (IGA)

To analyze these fields a flexible framework is beneficial. At the moment there are not many FEM codes that have enough flexibility. In a lot of FEM solvers, e.g., the element type is strongly attached to the problem to solve. Therefore, it is often hard to change the physical problem and keep the element type.

The aim of the project is to develop a code in which different elements and different physical problems can both easily be exchanged independently of each other. Furthermore, the code should offer the flexibility needed for modern finite element techniques. For doing so, the code must be designed as modular as possible. To gain this flexibility the Visitor Pattern is used. This pattern makes it possible to extend a given class with additional methods without modifying the class itself. In our case the basic FEM class holds different elements and is extended by a class that describes the physical problem. In this sense for every new physical problem one module has to be written.

2 Code Design

When developing software one often needs to solve commonly occurring design problems. Instead of solving these design problems every time again, one may want to use ideas of solutions to these problems that were thoroughly thought through. This is the concept of Design Patterns.
In 1994 Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, commonly known as the Gang of Four, published a book [2] in which they described 23 software design patterns which are nowadays the classic Design Patterns. They are separated these patterns in three categories

- **Creational Patterns:** Separate a system from how its objects are created, composed, and represented.
- **Structural Patterns:** Identify simple ways to realize relationships between entities.
- **Behavioral Patterns:** Identify common communication patterns between objects and realize these patterns.

2.1 Visitor Pattern

The Visitor Pattern is a behavioral pattern. It concerns the problem of a class which continuously has to be extended by functions. Instead of changing the base class every time, the idea is to separate new algorithms in a new class. The new class which realizes new methods is then called “visitor”.

This concept can be implemented as following (cf. Figure 1). The base class has to have an interface method that takes a derived visitor class as argument. In Figure 1 this method is called “accept”. Then the desired method can be used by calling the interface method with the respective visitor class as argument.
The executed function both depends on the derived class that provides the interface and on the derived visitor class.

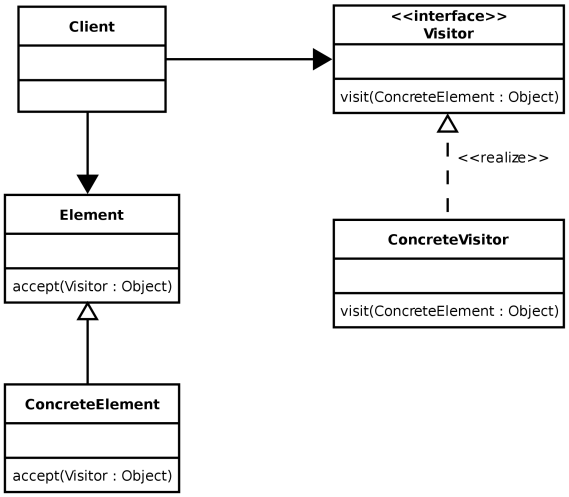


Figure 1: The class Element implements an "accept" method and the class Visitor a "visit" method. The functionality which has to be added to the Element class is then described in the class `ConcreteVisitor`. The operation can afterwards be used in the class `ConcreteElement` by calling the accept method with a `ConcreteVisitor` instance as argument.

2.2 Structure of the Finite Element Method

A Finite Element program can be divided in three parts:

1. **Preprocess:** First the problem to be solved has to be specified. Afterwards a mesh for solving it is generated.
2. **Solving process:** The finite dimensional (linear) problem is assembled and then solved by suitable direct or iterative linear solvers.
3. **Postprocess:** Quantities of interest have to be calculated out of the solution and may for example be displayed graphically.

The main focus in this work lies in the matrix assembly process. Here the different Finite Element Methods have the biggest differences when using different elements (like in XFEM, DG or IGA).

In the Finite Element approach the domain is partitioned into finitely many subdomains. On each of these so called elements the numerical solution u^h to the given partial differential equation is described by a linear combination of a finite number of basis functions $\{\phi_j\}_{j=1}^n$ with small support.

$$u^h = \sum_{j=1}^{n} c_j \phi_j \tag{1}$$

The differential equation can then equivalently be described in a variational formulation

$$a(u,v) = l(v) \qquad \forall v \tag{2}$$

where a is a bilinear form and l a linear form. Inserting the ansatz for u^h the following equations have to be solved

$$\sum_{j=1}^{n} a(\phi_i, \phi_j) c_j = l(\phi_i) \qquad i = 1 \ldots n. \tag{3}$$

Typically in the bilinear form an integral has to be computed. In the case of a Diffusion equation ($\Delta u = f$) the bilinear form a may be written as

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u \cdot \nabla v \, dx = \sum_k a_k(u,v), \\ a_k(u,v) &= \int_{\Omega_k} \nabla u \cdot \nabla v \, dx. \end{aligned} \tag{4}$$

This integral can be divided in integrals over the element domains Ω_k and (3) can be rewritten as

$$\sum_k \sum_{j=1}^{n} a_k(\phi_i, \phi_j) c_j = l(\phi_i) \qquad i = 1 \ldots n. \tag{5}$$

The integrals are now usually calculated by applying quadrature formulas with quadrature points x_{qk} and quadrature weights w_q. This means that (5) can be reformulated as

$$\sum_k \sum_q \sum_{j=1}^{n} \nabla\phi_i(x_{qk}) \cdot \nabla\phi_j(x_{qk}) w_q \, c_j = \sum_k \sum_q f(x_{qk}) \phi_i(x_{qk}) w_q \qquad i = 1 \ldots n. \tag{6}$$

With the notations

$$\begin{aligned} U &= \sum_k U^k \\ U^k_{ij} &= a_k(\phi_i, \phi_j) \\ b_i &= l(\phi_i) \end{aligned} \tag{7}$$

the discretized solution is then the solution to the linear system $U \cdot c = b$. In order to assemble this matrix for each element Ω_k the matrix U_k is calculated. In the example one has to determine

$$U^k_{ij} = \sum_q \nabla\phi_i(x_{qk}) \cdot \nabla\phi_j(x_{qk}) w_q. \tag{8}$$

The classical approach for programming this assembly process is shown in Listing 8.1 as realization of (8).

```
1 for (int k=element_begin;k<element_end;k++){
2   nlocnodes=elementlist[k]->nodes.size();
3   MatZeroEntries(element_matrix);
```

```
4   VecSet(element_rhs,0.0);
5   for(int i=0;i<nlocnodes;i++)
6     for(int j=0;j<nlocnodes;j++){
7       for(int q=0;q<nquad;q++){
8   qp=mesh->quadpoints[q];
9         loccontrib=sh11(i,qp,det)*sh11(j,qp,det);
10        loccontrib*=weights[q]*det;
11        MatSetValue(*element_matrix,i,j,&loccontrib,ADD_VALUES);
12      }
13    }
14  ...
15 }
```

Listing 8.1: Classical design

The idea is to calculate the contribution for each element separately. The first loop (line 1 in Listing 8.1) goes over all elements. For each element the contribution of all combinations of basis functions are calculated by using a quadrature formula. Here it is sufficient to take only basis functions into account that are not equal to zero on the considered element. That is the reason why the upper limit in the second and third loop are not equal to n (line 5 and 6 in Listing 8.1).

Here, the description of the physical problem is inside of four loops in the code (line 9 in Listing 8.1). Therefore, to change the physical problem one has to modify the code that is related to the geometry. In order to overcome this we use the Visitor Pattern. In the new approach the part that is responsible for the description of the physical problem is replaced by a call to an interface method which takes the physical problem as argument (line 5 in Listing 8.2). In this way the physical problem, which is implemented as derived class (Listing 8.3), visits the geometry class to assemble the linear system (line 12 in Listing 8.2).

```
1  for (int k=element_begin;k<element_end;k++){
2    nlocnodes=elementlist[k]->nodes.size();
3    MatZeroEntries(element_matrix);
4    VecSet(element_rhs,0.0);
5    this->accept_assemble(...);
6    ...
7  }
8
9  accept_assemble(visitFEM* problem,Element* element,
10                 int nlocnodes, Mat* element_matrix,
11                 Vec* element_rhs){
12   problem->assemble(this,element,nlocnodes,element_matrix,element_rhs);
13 }
```

Listing 8.2: New design with Visitor Pattern, FEMclass

```
1  int AdvDiff1D::assemble(fsQ1* mesh,Element* element, int nlocnodes, Mat* element_matrix){
2    for(int i=0;i<nlocnodes;i++)
3      for(int j=0;j<nlocnodes;j++){
4        for(int q=0;q<nquad;q++){
5    qp=mesh->quadpoints[q];
6          loccontrib=sh11(i,qp,det)*sh11(j,qp,det);
7          loccontrib*=weights[q]*det;
8          MatSetValue(*element_matrix,i,j,&loccontrib,ADD_VALUES);
9        }
10     }
11   return 0;
12 }
```

Listing 8.3: New design with Visitor Pattern, Diff1D

The associated class design is visualized in the Figures 2 and 3. The class `FEMclass` implements the main functions which can be called from outside the class to solve a predefined physical problem. The derived element classes specify which shape functions can be used and which mesh generators are implemented. On the other hand `visitFEM` just provides the assemble functions needed to be called from the `FEMclass`. The derived problem classes specify which parameters have to be set, whether initial valued have to be set and most important how the assembling is done.

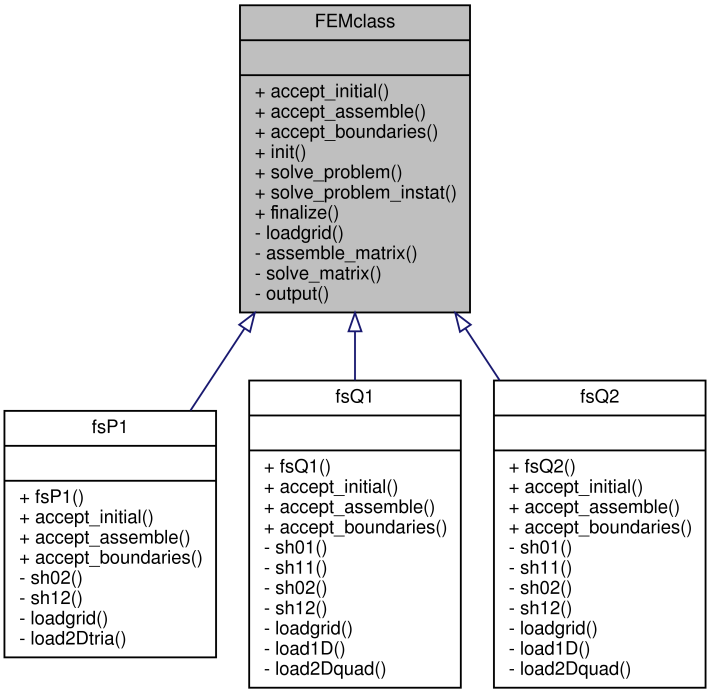


Figure 2: The UML diagram shows that the class `FEMclass` is responsible for the whole solving solving process. The subclasses implement the different kinds of elements.

For the underlying data structure PETSc [1] is used. In this way PETSc is also responsible for the parallelization during the assembling and solve process. During the project especially PETSc's manual [3] was very helpful and was often used.

3 Code Validation

The FEM solver was validated with different kinds of physical problems and element types. The test problems are taken from the book [4] where also an analysis for them can be found. In the one-dimensional case linear and quadratic elements were tested. In two dimensions Q1 and Q2 as well as P1 elements were considered.

All the following cases were solved on meshes with Q1 elements.

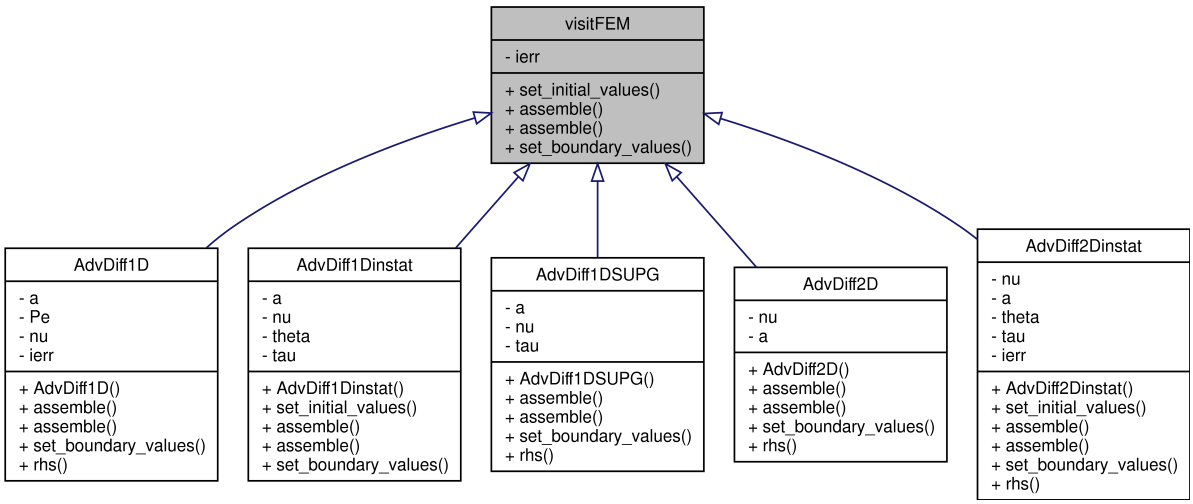


Figure 3: The class `visitFEM` has just the functionality to assemble an element. The subclasses describe the physical problems by implementing the assembling process accordingly.

1D Advection-Diffusion equation with and without SUPG stabilization

The first example is a simple Advection-Diffusion equation with either Dirichlet or homogeneous Neumann boundary conditions. It may be written in the form

$$au_x - \nu u_{xx} = s \qquad x \in \Omega = (0,1) \tag{9}$$

Here a is the parameter that describes advection and ν is responsible for the diffusion. The variational form which is solved for is then

$$a(w,u_x)_{L^2} + \nu(w_x,u_x)_{L^2} = (w,s)_{L^2} \qquad \forall w \tag{10}$$

The numerical solution is instable when a is to big as can be seen in Figure 4a.

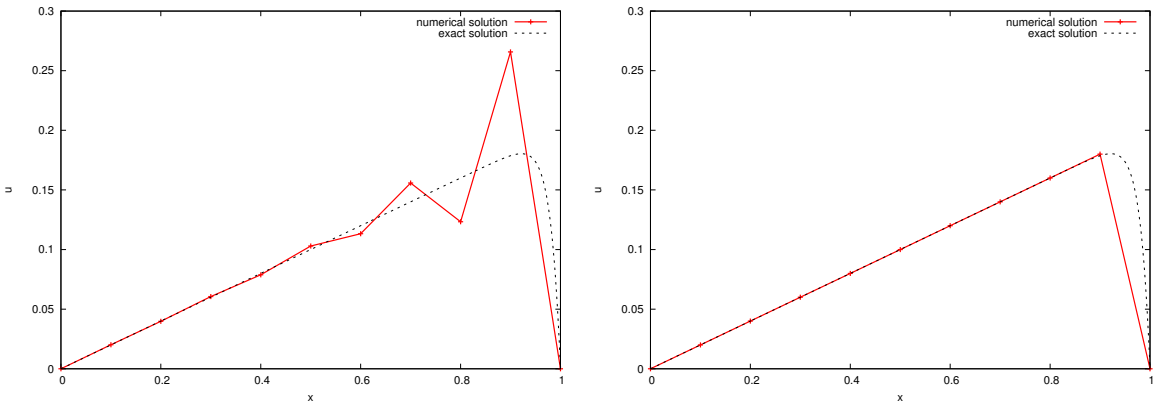
In order to stabilize the behavior observed in the former case in this example a SUPG stabilization is implemented. Therefore, the idea is to add the stabilization term

$$\int_\Omega a \cdot w_x \tau a \cdot (u_x - \nu u_{xx} - s)\, d\Omega \tag{11}$$

to the variational formulation. In the case of linear elements the second derivatives are equal to zero and the final variational formulation reads

$$a(w,u_x)_{L^2} + \nu(w_x,u_x)_{L^2} + a\tau(w_x, au_x - s)_{L^2} = (w,s)_{L^2} \qquad \forall w. \tag{12}$$

Now with the same parameters the numerical solution is no longer unstable. This can be seen in Figure 4b.



(a) unstabilized (b) SUPG stabilized

Figure 4: Numerical solution to the 1D Advection-Diffusion equation

1D instationary Advection-Diffusion equation

Finally for the one-dimensional case an instationary example with Dirichlet or homogeneous Neumann boundary conditions is considered. Based on the strong form:

$$u_t - \nu u_{xx} + a \cdot u_x = s \qquad x \in \Omega = (0,1) \tag{13}$$

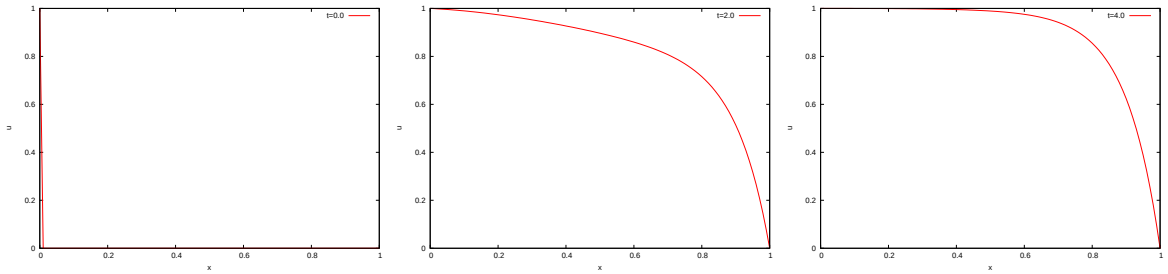
the weak form reads:

$$(w, u_t)_{L^2} + \nu(w_x, u_x)_{L^2} + a(w, u_x)_{L^2} = (w, x)_{L^2} \qquad \forall w. \tag{14}$$

For the time discretization the Θ-scheme is used. The fully discretized equation reads

$$\begin{aligned}&(w, u^{n+1})_{L^2} + \tau\Theta(\nu(w_x, u_x^{n+1})_{L^2} + a(w, u_x^{n+1})_{L^2}) \\ &= (w, u^n)_{L^2} + \tau(w, s)_{L^2} - \tau(1-\Theta)(\nu(w_x, u_x^n)_{L^2} + a(w, u_x^n)_{L^2}) \qquad \forall w.\end{aligned} \tag{15}$$

The following pictures (Figure 5) show a discontinuous initial condition that is transported in x-direction while diffusion is working on it.



(a) $t = 0.0$ (b) $t = 2.0$ (c) $t = 4.0$

Figure 5: Numerical solution to the instationary 1D Advection-Diffusion equation

2D Advection-Diffusion equation

For the two-dimensional part also a Advection-Diffusion equation with Dirichlet or homogeneous Neumann boundary conditions is implemented. Based on the strong formulation:

$$a \cdot \nabla u - \nu \Delta u = s \qquad x \in \Omega = (0,1) \times (0,1) \tag{16}$$

the weak formulation can be written as:

$$(w, a \cdot \nabla u)_{L^2} + \nu (\nabla w, \nabla u)_{L^2} = (w, s)_{L^2} \qquad \forall w. \tag{17}$$

In Figure 6 a solution with homogeneous Dirichlet boundary condition is shown.

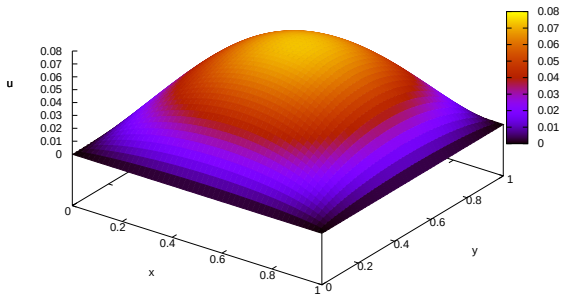


Figure 6: Numerical solution to the stationary 2D Advection-Diffusion equation

2D instationary Advection-Diffusion equation

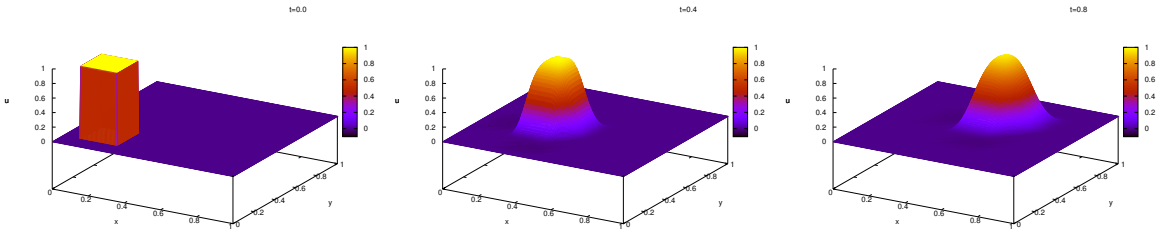
Finally there is also in two dimensions an instationary example. Analog to the one-dimensional case based on the strong form

$$u_t - \nu \Delta u + a \cdot \nabla u = s \qquad x \in \Omega = (0,1) \times (0,1) \tag{18}$$

the weak formulation that is discretized with the Θ-scheme can be written as

$$\begin{aligned} &(w, u^{n+1})_{L^2} + \tau\Theta(\nu(\nabla w, \nabla u^{n+1})_{L^2} + (w, a \cdot \nabla u^{n+1})_{L^2}) \\ &= (w, u^n)_{L^2} + \tau(w, s)_{L^2} - \tau(1-\Theta)(\nu(\nabla w, \nabla u^n)_{L^2} + (w, a \cdot \nabla u^n)_{L^2}) \qquad \forall w. \end{aligned} \tag{19}$$

In the following images (Figure 7) again a discontinuous initial solution on which advection and diffusion are working can be seen.

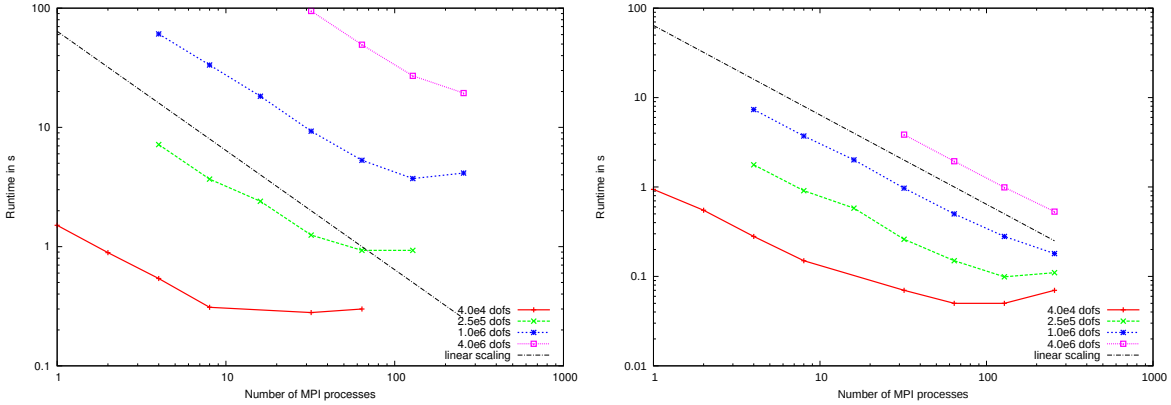


(a) $t = 0.0$ (b) $t = 0.4$ (c) $t = 0.8$

Figure 7: Numerical solution to the instationary 2D Advection-Diffusion equation

Parallel Performance

During the project scaling results on the supercomputer JUROPA were obtained. A two-dimensional Advection-Diffusion equation on a quadratic mesh is considered. As shown in Figure 8a the developed code scales with $4 \cdot 10^6$ nodes up to 256 processors well. This is clearly due to the effect that solving the linear system needs most of the run time. This part got no special attention during the project. In contrary the project focussed on the assembling process, which scales much better as shown in Figure 8b.



(a) Runtime assembly and solver (b) Runtime assembly

Figure 8: Scaling on JUROPA with 1 MPI process/core

4 Conclusion and Outlook

The project shows that the Visitor Pattern is a promising approach to overcome the strong coupling between geometry and physical problem. In the developed Finite Element solver the geometry and problems can easily and independently be changed. During implementing all the basic functionalities of an Finite Element code were implemented and no serious problem occured.

The code will be used in future work in the SimLab Highly Scalable Fluids & Solids Engineering. Especially Discontinuous Galerkin Methods and Isogeometric Analysis will be implemented.

5 Acknowledgments

I would like to thank my adviser Dr. Mike Nicolai for always having good answers to my many questions and for giving me suggestions on how to write Finite Element code. I had fun and learned a lot. Further thanks go to Mathias Winkel and Ivo Kabadshow for organizing the Guest Student Programme.

References

1. Balay S, Brown J, Buschelman K, Gropp WD, Kaushik D, Knepley MG, et al.. *PETSc Web page*; 2012. Available from: `http://www.mcs.anl.gov/petsc`.
2. Gamma E, Helm R, Johnson R, Vlissides JM. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional; 1994.
3. Balay S, Brown J, , Buschelman K, Eijkhout V, Gropp WD, et al. *PETSc Users Manual*. Argonne National Laboratory; 2012. ANL-95/11 - Revision 3.3.
4. Donea J, Huerta A. *Finite Element Methods for Flow Problems*. 1st ed. Chichester, West Sussex, New York: John Wiley & Sons; 2003.
5. Givoli D. *The Top 10 Computational Methods of the 20th Century*. IACM Expressions. 2001;11:5–9.

A parallel block iterative eigensolver optimized for sequences of correlated eigenproblems

Mario Berljafa

Faculty of Science, Department of Mathematics
University of Zagreb
Bijenička cesta 30
10000 Zagreb
Croatia

E-mail: mberljafa@gmail.com

Abstract:
In many materials science applications simulations are made of dozens of sequences; each sequence groups together eigenproblems with increasing self–consistent cycle outer–iteration index. Successive eigenproblems in a sequence possess a high degree of correlation. In particular, it was demonstrated that eigenvectors of adjacent eigenproblems become progressively more collinear to each other as the outer–iteration index increases. This result suggests one could use eigenvectors, computed at a certain outer–iteration, as approximate solutions to improve the performance of the eigensolver at the next one. In order to exploit this correlation we developed a block iterative eigensolver and showed the benefit of the usage of approximate versus random starting vectors. Moreover, we showed that the algorithm performs substantially better than the correspondent direct eigensolver, even for significant portion of the sought spectrum.

1 Introduction

In the last 30 years Density Functional Theory has been applied in a range of problems in physics, chemistry, biology and other. The problem that arises is the computation of the so called Kohn–Sham (KS) equations. Typically this equations are solved using an outer–iterative self consistent cycle: it starts from an initial guess for the charge density function, performs a few iterations and then converges to a final density. In practice, this outer–iterative cycle requires some form of discretization, and the effect of the discretization is the translation of the KS equations to a set of generalized eigenvalue problems for each outer–iteration cycle. It was proposed to look at this set as dozens of sequences of eigenvalue problems, where each sequence groups together problems with equal k–vectors and an increasing outer–iteration cycle index ℓ. We now concentrate on a sequence for a fixed k–vector.

Thus, we consider a sequence of N correlated generalized hermitian eigenvalue problems identified by a progressive index ℓ

$$\left\{P^{(\ell)}\right\} \equiv P^{(1)}, P^{(2)}, \ldots, P^{(N)} \quad ; \quad P^{(\ell)} : \; A^{(\ell)}x = \lambda B^{(\ell)}x. \tag{1}$$

The matrices $A^{(\ell)}$ are hermitian indefinite and the matrices $B^{(\ell)}$ are hermitian positive definite for all indices ℓ, which gives us a bounded discrete spectrum with real positive and negative eigenvalues

$$\lambda_{min} = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n = \lambda_{max}, \tag{2}$$

where n indicates the size of the matrices $A^{(\ell)}$ and $B^{(\ell)}$. The sequence is part of a process striving for convergence, which is achieved by consecutively solving all the problems in the sequence. Moreover, the solution of problem with index ℓ is necessary to initialize the eigensystem at iteration $(\ell + 1)$. Thus forcing us to solve one problem only after having solved the previous one.

Only a small part of the bottom end part of the spectrum is souhgt–after, no greater than $15\% - 20\%$ and usually quite smaller. The matrices $A^{(\ell)}$ and $B^{(\ell)}$ composing the eigenpencils $P^{(\ell)}$ are dense and the matrices $B^{(\ell)}$ tends to be quite ill-conditioned. The range of the problem we considered varies from $n \approx 2000$ to $n \approx 9000$. Each of the generalized eigenproblem is solved in two steps. First the matrix $B^{(\ell)}$ is factorized by a Cholesky decomposition which, in turn, is used to reduce the generalized into standard eigenproblem. In a second step we solve the standard eigenproblem.

The standard eigenproblem can be solved with a direct or an iterative method. A direct solver generally transforms the matrix into a tridiagonal one and then solves the eigensystem using standard algorithms like QR, Divide–and–Conquer or MRRR to name a few. Direct methods are the algorithms of choice when one is dealing with dense matrices, while iterative methods are usually applied to sparse matrices or matrices for which matrix–vector multiplication is the only convenient operation to perform.

Contrary to common practice we are going to use iterative methods, based on matrix–vector multiplication. This choice is motivated by a strong correlation between adjacent eigenproblems, as evidenced by the progressive collinearity of successive eigenproblems. Meaning, eigenvectors of the same eigenvalue from different eigenproblems are correlated. As the sequence index increases, the angle between those eigenvectors from adjacent eigenproblems decreases, reaching 10^{-8} or even 10^{-10} for the sequence index $\ell \approx 20$. Thus making the vectors more and more collinear [1].

While most standard iterative methods receive only one vector as input, it is desirable to work with a multiple number of vectors instead of a single one. Block iterative methods accept a variable set of multiple starting vectors. These methods have a faster convergence rate and avoid stalling when facing small clusters of eigenvalues. When augmented with polynomial accelerators their performance is further improved.

In our case we want to use a block of vectors with the specific objective of exploiting the correlation between successive eigenpairs. Let's assume we have already solved that the standard eigenproblem, $H^{(\ell)}Y^{(\ell)} = Y^{(\ell)}\Lambda^{(\ell)}$ related to the generalized eigenproblem $A^{(\ell)}X^{(\ell)} = B^{(\ell)}X^{(\ell)}\Lambda^{(\ell)}$ with index ℓ, where $B^{(\ell)} = L^{(\ell)}L^{(\ell)^H}$ is the Cholesky factorization of $B^{(\ell)}$.$L^{(\ell)}$ is lower triangular and $H^{(\ell)} =$

${L^{(\ell)}}^{-1} A^{(\ell)} {L^{(\ell)}}^{-H}$. Then the solution $\left(\Lambda^{(\ell)}, Y^{(\ell)}\right)$ is fed to our choice of eigensolver so as to speed up the solution of eigenproblem $H^{(\ell+1)} Y^{(\ell+1)} = Y^{(\ell+1)} \Lambda^{(\ell+1)}$.

The aim of this paper is to study how a specific iterative algorithm, Block Chebyshev Filtered Subspace Iteration (BChFSI) behaves when fed with approximate solutions, in comparison with random starting vectors. A parallel version is proposed and examined as well as a comparison between BChFSI and a corresponding direct eigensolver. In section 2 we present the algorithm, both sequential and parallel versions. In section 3 we show the numerical results, discuss the various approaches and theirs efficiency. With section 4 we conclude.

2 The algorithm

The chosen block iterative solver, BChFSI builds on top of the Subspace Iteration method. Its main characteristics are the acceptance of multiple starting vectors and the use of Chebyshev polynomials in order to highly accelerate the subspace iteration convergence. A schematic description of BChFSI is:

1: **start:**
2: Input an initial system of vectors $\hat{Y} := [\hat{y}_1, \ldots, \hat{y}_m]$.
3: **iterate:**
4: Filter the vectors by computing $\hat{Y} = C_m(\hat{Y})$.
5: Orthonormalize $\hat{Y}$.
6: Compute the Rayleigh quotient $G = \hat{Y}^H H \hat{Y}$ and solve the reduced standard problem $Gw = \lambda w$ giving $\left(\hat{\Lambda}, \hat{W}\right)$.
7: Compute new $\hat{Y} = \hat{Y}\hat{W}$.
8: Test for convergence.

In the following we describe the theoretical background needed for the algorithm and conclude this section with a detailed pseudo–code of the implemented BChFSI. In subsection 2.1 we define the Chebyshev polynomials and point out its main properties. In subsection 2.2 we show why in order to use the Chebyshev filtering we need to perform a few Lanczos steps. In subsection 2.3 we discuss the details regarding the check for the convergence, and in subsection 2.4 we present the detailed pseudo–code. Subsection 2.5 ilustrates details on the parallelization of the algorithm.

2.1 Chebyshev polynomials

Let $A \in \mathbb{C}^{n \times n}$ be a complex matrix, $\{(\lambda_i, v_i)\}_{i=1}^n$ it's eigenpairs, $w \in \mathbb{C}^{n \times 1}$ a complex vector and p an arbitrary polynomial. Expanding $w = \sum_{i=1}^n \gamma_i v_i$ in the eigenbasis, we obtain

$$p(A)w = \sum_{i=1}^{n} p(\lambda_i)\gamma_i v_i. \tag{3}$$

Our objective is to enhance eigenvectors corresponding to eigenvalues in a specific interval of the spectrum while, at the same time, be able to discard all the others. Equation 3 provides an effective tool for filtering out unwanted eigenvectors.

The problem of finding the optimal polynomials is set up as a min–max problem. It was shown (see [2]) that the optimal polynomial of degree m is the *scaled and translated* Chebyshev polynomial of the first kind of order m.

Definition 1. *The Chebyshev polynomial C_m of the first kind of order m, is defined as*

$$C_m(x) = \begin{cases} \cos\left(m \arccos(x)\right), & x \in [-1, 1], \\ \cosh\left(m \operatorname{arccosh}(x)\right), & |x| > 1. \end{cases} \tag{4}$$

One can easily verify that the Chebyshev polynomials satisfies the following three term recurrence

$$\begin{aligned} C_0(x) &= 1 \\ C_1(x) &= x \\ C_{m+1}(x) &= 2xC_m(x) - C_{m-1}(x), \quad m \in \mathbb{N}. \end{aligned} \tag{5}$$

which makes the computation efficient. Despite their definition, from the three term recurrence it can be shown that the $C_m(x)$ is a polynomial of degree m in x.

In practice, what makes the Chebyshev polynomials the most suitable is the rapid increase outside the interval $[-1, 1]$. In fact, the convergence of enhanced versus unwanted eigenvectors is controlled by the magnitude of the ratio $\frac{p(\lambda_{out})}{p(\lambda_{in})}$. This ratio can reach 10^{12} already for polynomials of degree $m = 20$, making the Chebyshev polynomials the optimal choice for our proposed target.

The scaled and translated Chebyshev polynomials $\hat{C}_m$ are defined as

$$\hat{C}_m(\lambda) = \frac{C_k\left[(\lambda - c)/e\right]}{C_k\left[(\lambda_1 - c)/e\right]}. \tag{6}$$

The rescaling is needed due to the blow up of the ratio $\frac{p(\lambda_{out})}{p(\lambda_{in})}$, to avoid overflow, and the parameters c, e are used to map any interval to the interval $[-1, 1]$. We will talk more about the mapping in the next subsection. In Algorithm 1 we present the pseudo–code of the filter.

Algorithm 1 Chebyshev filter.

Require: H, vectors Z_0 to be filtered, c, e, λ_1,DEG.
Ensure: Filtered vectors Z_{DEG}.

1: $\sigma_1 \leftarrow e/(\lambda_1 - c)$
2: $Z_1 \leftarrow \frac{\sigma_1}{e}(H - cI_n) Z_0$
3: **for** $i = 1 \rightarrow \text{DEG} - 1$ **do**
4: $\quad \sigma_{i+1} \leftarrow \frac{1}{(2/\sigma_1 - \sigma_i)}$
5: $\quad Z_{i+1} \leftarrow 2\frac{\sigma_{i+1}}{e}(H - cI_n) Z_i - \sigma_{i+1}\sigma_i Z_{i-1}$
6: **end for**

The algorithm is the straightforward implementation of the three term recurrence, Equation 5, applied to the scaled and translated Chebyshev polynomials.

2.2 The Lanczos algorithm

From Equation 3 it follows that $p(\lambda)$ needs to be small in magnitude for the unsought, and high for the desired eigenvalues. Since the desired eigenvalues are those from the bottom end part of the spectrum, the spectrum needs to be partitioned in two disjoint intervals. One containing the eigenvalues we are seeking for and the other containing the unwanted eigenvalues. Let the interval containing the unwanted eigenvalues be labeled I. The Chebyshev polynomial p which satisfies

$$p(x) \gg p(y) \quad \forall y \in I, \forall x \in \mathbb{R} \setminus I$$

will provide the effect we are looking for.

To benefit from the rapid increase of the Chebyshev polynomials outside the interval $[-1, 1]$, the interval $I = [\alpha, \beta]$ needs to be mapped into the interval $[-1, 1]$. The remaining problem is to determine the lower bound α and the upper bound β of the interval I.

The lower bound can be defined as the highest (current) approximation for the wanted eigenvalues, while the upper bound needs some more attention. The upper bound is actually the extreme right end of the whole spectrum of the matrix and can be computed by the k–step Lanczos iteration where k is in general very small, [3].

A pseudo-code of the k–step Lanczos iteration is given in Algorithm 2. For the Hermitian matrix A, the algorithm builds the matrices V_k and T_k such that $AV_k = V_kT_k + \beta_k v_{k+1}e_k^T$, where $V_k = [v_1 \dots v_k]$ is $n \times k$ and has orthonormal columns, T_k is $k \times k$ and is tridiagonal with α_js on the diagonal and β_js on the superdiagonal and the subdiagonal and e_k is the kth column of the $k \times k$ identity matrix.

Algorithm 2 k–step Lanczos.

Require: A Hermitian matrix A, initial vectors v_1 of norm unit.

1: $\beta_0 \leftarrow 0, v_0 \leftarrow 0$
2: **for** $j = 1 \rightarrow k$ **do**
3: $\quad w_j = Av_j - \beta_{j-1}v_{j-1}$
4: $\quad \alpha_j = \langle w_j | v_j \rangle$
5: $\quad w_j = w_j - \alpha_j v_j$
6: $\quad \beta_{j+1} = ||w_j||_2$
7: $\quad v_{j+1} = \dfrac{w_{j+1}}{\beta_{j+1}}$
8: **end for**

The upper bound β is then computed as

$$\beta = ||T_k|| + |e_k^T z_k||\beta_{k+1}|, \tag{7}$$

where $T_k z_k = \mu_k z_k$.

Now that we have identified the limits of the filtered interval, we can insert them in the unspecified

constants of Equation 6. We define $c = (\beta + \alpha)/2$ and $e = (\beta - \alpha)/2$ thus mapping the interval $I = [\alpha, \beta]$ into the interval $[-1, 1]$.

2.3 Locking the converged eigenvectors

A standard technique that manages the different rates of convergence of the eigenpairs is the so called locking. As soon as the first eigenpair has converged the corresponding eigenvector can be frozen and the computation can be carried on with the remaining vectors. Since eigenpairs typically converge in chunks and the main time consuming part of the subspace iteration is the matrix–vector multiplication, locking techniques supplie remarkable computation savings. From a mathematical point of view, what allows us to perform the locking is the Rayleigh–Ritz theorem.

Theorem 1 (Rayleigh–Ritz)**.** *Let $\mathcal{Y}$ be a subspace containing an eigenspace $\mathcal{X} < \mathcal{Y}$ of the standard eigenproblem $Hw = \lambda w$. Let Y be a basis of vectors for $\mathcal{Y} = \operatorname{span}(Y)$, Y^I a left inverse of Y, and $H_Y = Y^I H Y$, the so-called Rayleigh quotients for H. If (Λ, W) are primitive Ritz pairs of the reduced problem, i. e., $H_Y W = W\Lambda$, then (Λ, YW) are Ritz pairs for the original eigenproblem and* $\operatorname{span}(YW) = \mathcal{X}$.

The theorem allows us to split an initial eigen–subspace $\mathcal{Y}$ in successive subspaces $\mathcal{Y} = \mathcal{Y}_{\text{CONV}} + \mathcal{Y}^{\perp}_{\text{CONV}}$, with $\mathcal{Y}_{\text{CONV}}$ an eigen–subspace of $\mathcal{Y}$ and $\mathcal{Y}^{\perp}_{\text{CONV}}$ its orthogonal complement. At each iteration $\mathcal{Y}_{\text{CONV}}$ grows at the expense of $\mathcal{Y}^{\perp}_{\text{CONV}}$. Since the theorem can be applied at the same time to $\mathcal{Y}$ and any of its subspaces, it ensures that $\mathcal{Y}_{\text{CONV}}$ remains invariant while $\mathcal{Y}^{\perp}_{\text{CONV}}$ is scanned for additional eigen–subspaces, leading us to the following strategy.

1: **iterate:**
2: Set $j = 1$.
3: Receive as input an orthonormal basis $Z = \left[Q, \hat{Y}\right]$ for $\mathcal{Y}$, where $Q = [q_1, \ldots, q_{j-1}]$, $\operatorname{span}(Q) = \mathcal{Y}_{\text{CONV}}$ and $\hat{Y} = [\hat{y}_j, \ldots, \hat{y}_k]$, $\operatorname{span}\left(\hat{Y}\right) = \mathcal{Y}^{\perp}_{\text{CONV}}$.
4: Update $G = \hat{Y}^H H \hat{Y}$ and compute Ritz vectors $\hat{W} = [\hat{w}_j, \ldots, \hat{w}_k]$ and associated eigenvalues $\hat{\Lambda} = \left[\hat{\lambda}_j, \ldots, \hat{\lambda}_k\right]$.
5: Test convergence for $(\lambda_j, w_j), \ldots, (\lambda_k, w_k)$.
Set $i_{\text{CONV}} =$ number of converged eigenpairs.
6: Set $j = j + i_{\text{CONV}}$. Append converged vectors to Q.
7: Compute new $\hat{Y} = \hat{Y}\hat{W}$.

2.4 Block Chebyshev filtered subspace iteration

Finally, in Algorithm 3 we illustrate the detailed pseudo–code of the Block Chebyshev Filtered Subspace Iteration. Apart from the matrix describing the eigenproblem and the starting approximations for the eigenpairs the algorithm requires as input a stopping criteria TOL for checking the convergence, a degree DEG for the scaled Chebyshev polynomial and the number of wanted eigenvalues, NEV.

BChFSI was implemented using the Intel MKL LAPACK and BLAS libraries. LAPACK was used for solving the reduced problem with a direct solver – MRRR, and for the re–orthogonalization. BLAS

Algorithm 3 BChFSI

Require: H, approximate eigenpairs $\left(\hat{\Lambda}, \hat{Y}\right)$, TOL, DEG, NEV.
Ensure: Wanted eigenpairs (Λ, Y).

converged $\leftarrow 0$
$\Lambda \leftarrow []$
$Y \leftarrow []$
upper $\leftarrow$ LANCZOS$(H, \text{RANDN}(n, 1))$
repeat
  $\lambda_1 \leftarrow \min_i \hat{\Lambda}(i)$
  lower $\leftarrow \max_i \hat{\Lambda}(i)$
  $\hat{Y} \leftarrow$ CH_FILTER $\left(H, \hat{Y}, \text{lower}, \text{upper}, \lambda_1, \text{DEG}\right)$
  $\hat{Y} \leftarrow$ QR $\left(\left[Y\ \hat{Y}\right]\right)$
  $\hat{Y} \leftarrow \hat{Y}\,[:, \text{converged} : \text{NEV}]$
  $G \leftarrow \hat{Y}^H H \hat{Y}$ ▷ Compute the Rayleigh quotient.
  Solve $G\hat{W} = \hat{W}\hat{\Lambda}$. ▷ Compute the primitive Ritz pairs $\left(\hat{\Lambda}, \hat{W}\right)$.
  $\hat{Y} \leftarrow \hat{Y}\hat{W}$ ▷ Compute the approximate Ritz pairs $\left(\hat{\Lambda}, \hat{Y}\hat{W}\right)$.
  for $i = \text{converged} \rightarrow \text{NEV}$ **do** ▷ Check which among the Ritz vectors converged.
    if $\dfrac{||H\hat{Y}(:,i) - \hat{\Lambda}(i)\hat{Y}(:,i)||}{||\hat{Y}(:,i)||} < \text{TOL}$ **then**
      $\Lambda \leftarrow \left[\Lambda\ \hat{\Lambda}(i)\right]$ ▷ Lock converged eigenpairs.
      $Y \leftarrow \left[Y\ \hat{Y}(:,i)\right]$
    else
      break
    end if
  end for
  converged $\leftarrow i$ ▷ Lock converged eigenpairs.
  $\hat{\Lambda} \leftarrow \hat{\Lambda}(\text{converged} : \text{END})$
  $\hat{Y} \leftarrow \hat{Y}(:, \text{converged} : \text{END})$
until converged $<$ NEV

was used for matrix–vector multiplication in the k–step Lanczos iteration, Algorithm 2, for computing the norms during the convergence check and most important for the matrix–matrix multiplications. As we saw in Algorithm 1, the filter mainly consists of matrix–matrix multiplications. BLAS Level 3 subroutine _GEMM (general matrix–matrix multiplication) is the most optimized over all the routines in the library thus making it the best choice for our purposes.

2.5 Parallelization

After running the tests on the sequential implementation of BChFSI it was observed that the filtering takes around 91% of the total time needed for the computation. The re–orthogonalization takes $\approx 1\%$,

initializing and solving the reduced problem requires $\approx 4\%$ of the total time and the remaining 4% is mainly consumed by the convergence check. Therefore, our parallelization effort focuses on the Chebyshev filter.

Since Intel MKL provides multi–threaded versions of the libraries, LAPACK and especially BLAS, the easiest way of parallelization is just using these capabilities. Another approach was to build our own parallel version using OpenMP. In the second approach the distribution of the matrices is done in the following way. The rows of the matrix H are distributed among cores. For k cores, the first $\lfloor n/k \rfloor$ rows are distributed to the 1st core, the next $\lfloor n/k \rfloor$ rows are distributed to the 2nd and so on. The (eventual) extra $n \pmod k$ rows are distributed to the first $n \pmod k$ threads. Vectors to be filtered are shared among all the threads.

After testing both versions, the OpenMP one was performing slightly better due to better offloading of the work. This is justified by the small size of the vector matrix to be filtered.

Thus, the final parallel version is made of the OpenMP filter and multi–threaded LAPACK and BLAS in the rest of the algorithm.

3 Numerical results

Since our main task is to show the importance of feeding the approximate solutions to the eigensolver we have tested BChFSI also with starting random vectors. In other words, we show the benefit of using approximate solutions by comparing BChFSI when random vectors are used as input.

The tests were performed on JUROPA using one node with 8 cores. Each node consists of two Intel Nehalem quad–core processors, 2.93GHz. The theoretical peak performance is 11.71 Gigaflops per core.

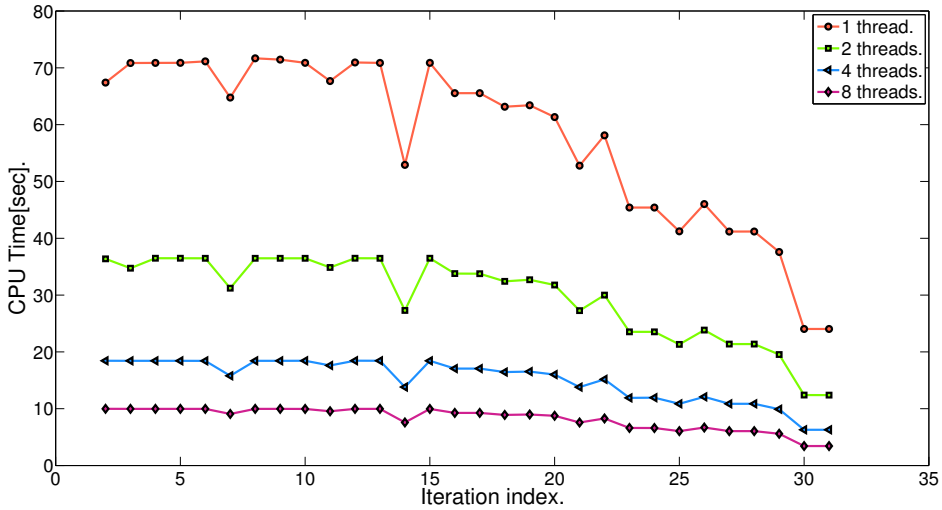


Figure 1: Problem size $n = 2628$, NEV $= 136$. Different number of threads.

In Figure 1 we can see the behavior of BChFSI fed with approximate solutions as the iteration index increases. The size of the problem is $n = 2628$, and we are looking for the bottom NEV $= 136$ (5.17% of the total spectrum) eigenvalues and corresponding eigenvectors. What we can notice immediately is that, as the iteration index increases, the time needed decreases. That's because the eigenvectors corresponding to the same eigenvalue of adjacent eigenproblems are becoming more and more collinear as the iteration index increases. In the same Figure we can see the benefit of the parallelization, using two cores the average speedup is 1.8 times, using four cores 3.5 times and with eight cores the parallel version is about 7 times faster than the sequential version. As we will see later, for bigger systems the speedup is even better.

Our algorithm was compared with a direct solver – MRRR implemented in LAPACK, using multi–threaded BLAS. In the Figures to come, this is labeled as *Direct*, while BChFSI with random starting vectors is labeled *Random*, and BChFSI fed with approximate solutions, from the previous iteration, is labeled *Approx*.

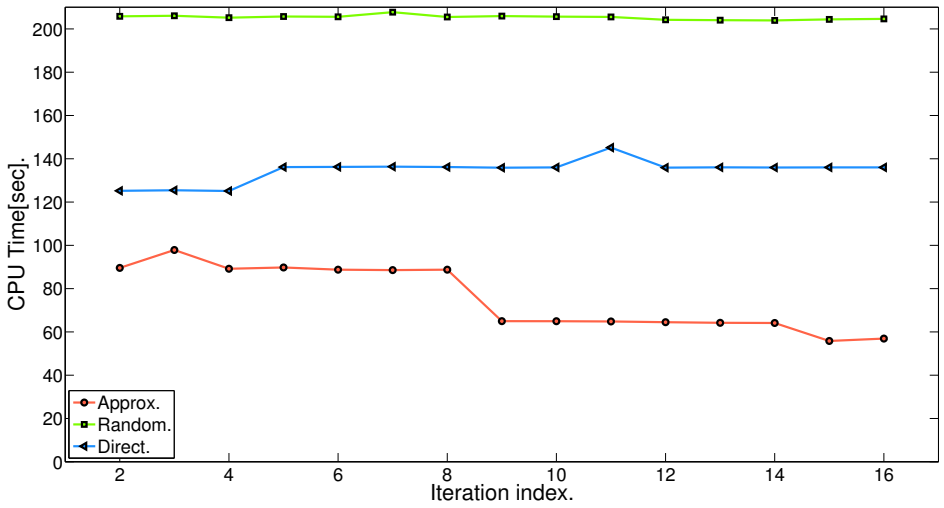


Figure 2: Problem size $n = 6217$, NEV $= 256$. Different approaches.

In Figure 2 and Figure 3 we compare the the three parallel algorithms, BChFSI with approximate solutions, BChFSI with random vectors and the MRRR, for different systems.

In Figure 2 we have a system of size $n = 6217$ and NEV $= 256$ eigenpairs are sought–after, which corresponds to 4.11% of the spectrum, and in Figure 3 we analyze the behavior of a system of size $n = 8970$ where NEV $= 972$ (10.83%) eigenpairs are needed.

We can once again notice the phenomena of computation time reduction for BChFSI fed with approximate solutions as the iteration index increases. The other two algorithms don't benefit from this property because they are solving each problem independently from the others. Thus, not exploiting the correlation.

For the system of size $n = 6217$, the gain of feeding approximate solution results in a starting speedup of 2.2, and grows as the iteration index increases to 3.1 after the first 8 iterations. This jump in speedup is due to the fact that at that point we needed only two iterations to converge with the *Approx*, while

with *Random* we still need 5 iterations. For the last two iterations the speedup of *Approx* against *Random* reaches 3.6.

Comparing *Approx* versus *Direct*, we can see that at the beginning *Approx* are 1.4 times faster than the *Direct* method, and as the iteration index increases this factor grows up to 2.4 making the BChFSI considerably faster.

Apart from the problem size, the main difference between the system on Figure 2 and Figure 3 is that for the system in Figure 3 a much bigger portion of the spectrum is desired, almost 11%. Despite this, we are still faster than a direct solver, and the benefit of using the approximate solution over just using random vectors results in a speedup of 5.6.

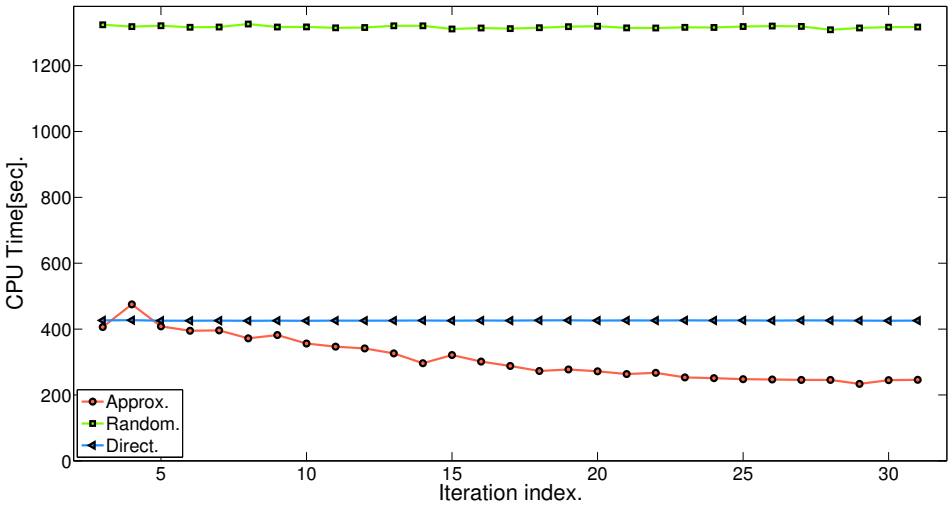


Figure 3: Problem size $n = 8970$, NEV $= 972$. Different approaches.

4 Conclusion

The aim of the project was to implement both the sequential and parallel version of BChFSI and show the benefit of using approximate solutions. Exploiting the approximate solutions by feeding them to the eigensolver allows a better performance when compared with direct solvers even for dense matrices. An important point in surpassing the direct solver was the almost perfect scaling of the parallel version. We conclude saying that the target of the project is completely fulfilled.

Acknowledgments

I want to thank my adviser Dr. Edoardo A. Di Napoli for guiding me through this project. For all the explanations and help given as well as for the understanding. A great thanks goes also to Prof. Dr. Sc. Sanja Singer for the Letter of Recommendation.

References

1. Di Napoli E, Blügel S, Bientinesi P. Correlations in sequences of generalized eigenproblems arising in Density Functional Theory. Computer Physics Communications 2012; 183:1674–1682.

2. Saad Y. Numerical Methods for Large Eigenvalue Problems. 2nd edition. `http://www-users.cs.umn.edu/~saad/eig_book_2ndEd.pdf`

3. Zhou Y, Li RC. Bounding the spectrum of large Hermitian matrices. Linear Algebra and its Applications 2011; 435:480–493.

Observation of a Universal Boltzmann Distribution in Dynamic Simulation Experiments of the 1D-Heisenberg Spin Model

Kieran Austin

University of Leipzig
Institute for Theoretical Physics
Brüderstraße 14-16
04103 Leipzig

E-mail: austin@itp.uni-leipzig.de

Abstract:
In this exploratory study, computer simulations of the classical Heisenberg spin model are carried out in the microcanonical ensemble. It is shown that the Boltzmann distribution can be generated in a dynamical simulation for various settings of the system parameters. The temperature was measured with the use of a novel expression and its correctness is verified. A short outlook is given for the use of this new tool.

---

1 Introduction

A physicists approach to numerical simulations is most often one where the underlying theoretical background comes from a probabilistic view, i. e. statistical physics. Moreover, the Boltzmann factor is one of the central tools used, e.g. to generate the canonical distribution for some system of interest. It is easy to define a temperature in this approach, as this parameter can be directly set to any desired value in the Boltzmann factor.

The classical approach to physics is, of course, a deterministic one, where the dynamics of the system is given by some equation of motion. The propagation in time is governed by a Hamiltonian $\mathcal{H}$, which value is the energy of this system. The equation of motion does not directly incorporate the temperature. Although the temperature can easily be measured in real-life experiments, how can this be done for a dynamical simulation?

A novel expression could be derived in [1] for the temperature, assuming ergodicity. This is a very useful tool for measuring the temperature in a dynamical way. An even more important aspect of this is that it is now possible to test these two approaches to physics and compare obtained results. Of course, these two approaches should not be contradictory, but rather complementary in understanding the behaviour of some system of interest.

This study is dedicated to exploring the behaviour of the Heisenberg Spin Model in one dimension in a dynamic computer simulation. Some work has been done in the past [2], though the temperature has not been measured directly with this novel expression. It is especially of interest here, how this magnetic system behaves in the process of equilibration and thermalisation. For this a microcanonical ensemble is prepared in which the subsystem is in an out-of-equilibrium state and connected to a much larger bath. Solving the equation of motion for this system and measuring energy and temperature should then show that the subsystem firstly thermalises and secondly follows the canonical distribution.

2 The Heisenberg Spin Model

In order to understand a magnetic system with the methods of computer simulations, one needs to find implementable models. These models have to describe the system of interest to some extent correctly, but are generally an abstraction of nature. One of the most famous models for a magnetic system is the Heisenberg Spin Model. It has been studied exhaustively in the past, and from statistical approaches some analytic expressions for its behaviour could be derived. Here it will be defined and throughout this study used in a classical sense.

The Heisenberg Spin Model is defined by its three-component spin vectors

$$\mathbf{s}_i = (s_i^x, s_i^y, s_i^z) \tag{1}$$

for every particle in a system of N particles. Each spin vector follows the constraint $\|\mathbf{s}_i\| = 1$, meaning it can freely rotate in three dimensions on a sphere with radius one. Furthermore, this constraint means that there are two degrees of freedom per spin. The third vector component can always be derived by knowledge of the other two. These particles with the attribute 'spin' are then arranged on a lattice, with equal spacing. Each spin then has a set of next neighbours. Here, these 3D-spins are arranged in one dimension, which can be thought of as a chain of spins.
In this investigation the Hamiltonian $\mathcal{H}$ of the system will the often used two-point correlation functional

$$\mathcal{H} = -\sum_{\langle i,j \rangle} J_{ij} \mathbf{s}_i \mathbf{s}_j \; . \tag{2}$$

The index of summation $\langle i, j \rangle$ implies a summation over all next neighbour spin pairs, i. e. only next neighbours directly interact with one another. The type and strength of coupling of these spin pairs is governed by the parameter J_{ij}. Generally one speaks of a ferromagnetic coupling if $J_{ij} > 0$, and an antiferromagnetic coupling if $J_{ij} < 0$. Usually, the couplings are set to $|J_{ij}| = 1$. For ferromagnetic and antiferromagnetic couplings this means that the spins energetically favour a parallel or antiparallel arrangement, respectively.
The dynamics of the system is described per spin by the following equation of motion:

$$\frac{d\mathbf{s}_i}{dt} = \frac{\partial \mathcal{H}}{\partial \mathbf{s}_i} \times \mathbf{s}_i \tag{3}$$

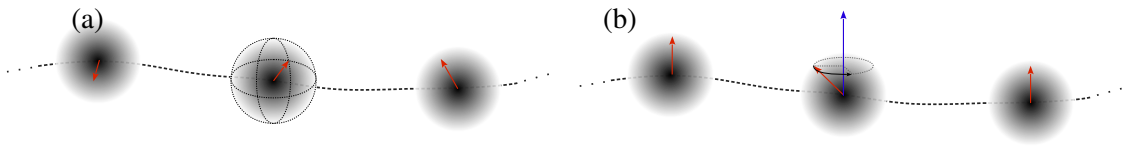


Figure 1: Two cutouts of the spin chain. (a) The spins, expressed as a three-dimensional vector, can move on a sphere with radius one. (b) The equation of motion for each spin can be visualized as a precession about its local effective field formed by its next neighbours.

This equation can be rewritten to $\dot{\mathbf{s}}_i = \mathbf{s}_i \times \mathbf{M}_{\text{eff}}$ and desribes a Larmor precession of the spin about an effective local field

$$\mathbf{M}_{\text{eff}} = \sum_{j=\text{nn}(i)} J_{ij}\mathbf{s}_j. \tag{4}$$

The index of summation denotes a summation over all next neighbour spins at position i. The precession is basically a rotation of the spin about the axis defined by the vector sum of its neighbours. This rotation does not change the energy locally, and thus conserves the energy of the whole system. In Figure 1 the spin chain is visualized.

An exact expression for the energy of the Heisenberg spin chain was derived in [3]. For a homogeneous ferromagnetic Heisenberg spin system ($J_{ij} \equiv 1$) the energy per spin is

$$e(\beta) = \beta^{-1} - \coth(\beta)\,, \tag{5}$$

where β is the inverse temperature $\beta = (k_B T)^{-1}$ and k_B set to unity. From this the specific heat can be derived:

$$c(\beta) = -\beta^2 \frac{de}{d\beta} = 1 + \beta^2(1 - \coth^2(\beta))\,. \tag{6}$$

These exact expressions enable a comparison of experimental data of a dynamic simulation to exact results derived from statistical physics.

The general expression of the temperature found in [1] is written as a limit of a time average.

$$\beta = \lim_{t\to\infty} \frac{1}{t} \int_0^t \nabla \left(\frac{\nabla \mathcal{H}}{\|\nabla \mathcal{H}\|^2} \right) d\tau\,. \tag{7}$$

Note that the integrand only incorporates the Hamiltonian $\mathcal{H}$ of the system, which can be calculated for every time step. The inverse temperature for spin systems, derived in [4], enables the calculation of an inverse temperature β_t

$$\beta_t = -\frac{4\mathcal{H}}{\sum_i \dot{\mathbf{s}}_i^2} \text{ and } \beta = \langle \beta_t \rangle\,. \tag{8}$$

The index t denotes the inverse temperature at one time step in the solution of the equation of motion for the whole system. The average $\beta = \langle \beta_t \rangle$ then yields the inverse temperature for the whole system, assuming long enough time series and that it is in equilibrium. Computationally, this is very convenient. The energy of the system and the time derivative $\dot{\mathbf{s}}_i$ is determined at every time step anyway for solving the equation of motion.

3 Methods

In this exploratory study it is of interest how a Heisenberg spin ring in the microcanonical ensemble behaves under a thermalisation process. For this, periodic boundary conditions are applied to the Heisenberg spin chain, meaning that the first spin has the last spin of the chain as one next neighbour. This ring has N particles with a spin, and is declared the subsystem. The subsystem is then connected to a much larger ring with B particles ($B \gg N$), which is declared to be the bath. The connection of the subsystem with the bath means that both rings can interact with each other, as illustrated in Figure 2. The Hamiltonian of the whole system $\mathcal{H}$ can be separated into a sum of a bath, interaction and subsystem Hamiltonian:

$$\mathcal{H} = \mathcal{H}_B + \mathcal{H}_I + \mathcal{H}_S \; . \tag{9}$$

Each of these can be measured individually. The interaction Hamiltonian $\mathcal{H}_I$ is a summation over the connecting bonds between bath and subsystem. They were chosen randomly, and a number of four interaction bonds was chosen throughout this study. Because a comparatively small number of interaction bonds was chosen, the interaction Hamiltonian $\mathcal{H}_I$ is presumed to be negligible.

It is necessary to be able to set the energy to a certain value for the whole system. To do so, the z-component of each spin vector is initially set to be in the range $s_i^z \in [h, 1]$, where $h \in [-1, 1]$. The other two spin vector components can then be chosen randomly to be on a circle with radius $r_i = \sqrt{1-(s_i^z)^2}$, such that the constraint $\|\mathbf{s}_i\| = 1$ is still fulfilled. The boundary values for the parameter h correspond to different values for the temperature, and thus to different values of energy. A limit $h \to 1$ corresponds to $\beta \to \infty$ (or $T \to 0$) and an energy per spin $e \to -1$, since all spins are then completely aligned. A limit $h \to -1$ corresponds to $\beta \gtrapprox 0$ (or $T \to \infty$) and an energy per spin $e \lessapprox 0$. The spins in this case are completely random.

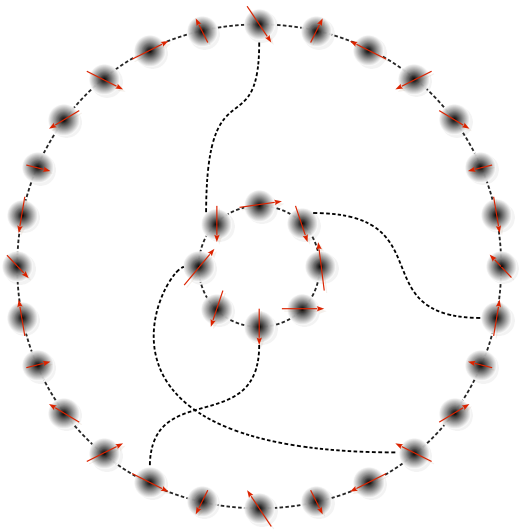


Figure 2: The system that was simulated in this study. The subsystem (inner ring) interacts with a large bath (outer ring). This microcanonical system follows energy conservation.

Another aspect is of importance. There is an ongoing discussion about the microcanonical ensemble in a molecular dynamics simulation. Usually the microcanonical ensemble is understood to have three globally constant parameters, namely the number of particles N, the volume V and the total energy E. One study [6] finds that the correct ensemble that must be associated with molecular dynamics is one where the total linear momentum and the constant of motion associated with the Galilean boosts must additionally be incorporated. Because it was not clear how this could affect a spin system, the total magnetization $M = \sum_i \mathbf{s}_i$ was set close to zero, $|M| \approx 0$. A method of over-relaxation was used. Having an initial start configuration of the system, randomly chosen spins were reflected along the axis of its local effective field M_{eff}. Because this is basically a rotation of $90°$, it conserves the total energy E of the system.

Two methods for solving the equation of motion have been used. These are the Predictor-Corrector and the Suzuki-Trotter decomposition method that have been proposed by [5] for the integration of the equation of motion of classical spin systems. For both methods one time iteration step means evolving the system by a time step δt.

3.1 The Predictor-Corrector Method

The basic idea behind the predictor-corrector method is to series expand the integration process of the equation of motion. For each spin the equation of motion (3) can be rewritten to a more symbolic form, $\dot{\mathbf{s}}_i = f(\mathbf{s}_i)$. First, an update is predicted by the explicit Adams-Bashforth four-step method:

$$\tilde{\mathbf{s}}_i(t+\delta t) = \mathbf{s}_i(t) + \frac{\delta t}{24}\left[55 f(\mathbf{s}_i(t)) - 59 f(\mathbf{s}_i(t-\delta t)) + 37 f(\mathbf{s}_i(t-2\delta t)) - 9 f(\mathbf{s}_i(t-3\delta t))\right] \ . \quad (10)$$

Second, this is corrected to a better approximation by applying one iteration of the Adams-Moulton three-step method:

$$\mathbf{s}_i(t+\delta t) = \mathbf{s}_i(t) + \frac{\delta t}{24}\left[9 f(\tilde{\mathbf{s}}_i(t+\delta t)) + 19 f(\mathbf{s}_i(t)) - 5 f(\mathbf{s}_i(t-\delta t)) + f(\mathbf{s}_i(t-2\delta t))\right] \ . \quad (11)$$

Both steps have a local truncation error of $\mathcal{O}(\delta t^5)$. A small downside of this method is that an additional algorithm has to be implemented. This method needs the four initial time steps $\mathbf{s}_i(0), \mathbf{s}_i(\delta t), \mathbf{s}_i(2\delta t)$ and $\mathbf{s}_i(3\delta t)$ to determine the next step in time $\mathbf{s}_i(4\delta t)$. For this, the Runge-Kutta method was implemented, which accumulates truncation errors too fast for it to be used for the whole simulation.

The magnetization is conserved within machine accuracy, if periodic boundary conditions are applied, as is the case here. A pitfall of this method can be, that the total energy of the system is not a conserved quantity. In fact, the total energy seems to increase linearly in time. This is illustrated for a rather small system in Figure 3.

3.2 The Suzuki-Trotter Decomposition Method

This method employs the fact that the time propagation of each spin is a precession about the local effective field. The chain, or more generally speaking the lattice, is decomposed into two sublattices $\mathcal{A}$ and $\mathcal{B}$ in a checkerboard style. Each spin is assigned a new attribute, expressed in colours. It can

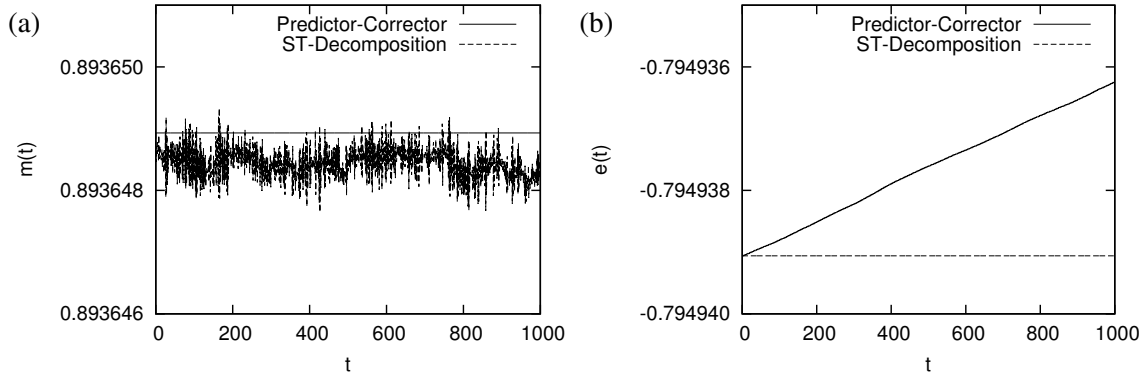


Figure 3: The two methods in comparison. (a) The magnetisation per spin versus time. The Predictor-Corrector method (PC) conserves the magnetisation, while for the Suzuki-Trotter decomposition method (STD) it seems to fluctuate around some mean value. (b) The total energy per spin versus time. While the STD method conserves the energy exact, the energy in the PC method seems to rise linearly in time.

either be 'white' or 'black'. Every 'white' spin then is set, such that it has only 'black' spins as next neighbours, and vice versa. All 'white' or 'black' spins belong to the sublattice $\mathcal{A}$ or $\mathcal{B}$, respectively. The sublattices are then alternately updated. Each spin is thus rotated about an angle $\mathbf{M}_i \delta t$ dependent only on spins on the other sublattice.

$$\mathbf{s}_i(t+\delta t) = \frac{\mathbf{M}_i(\mathbf{M}_i \cdot \mathbf{s}_i(t))}{\mathbf{M}_i^2} + \left[\mathbf{s}_i(t) - \frac{\mathbf{M}_i(\mathbf{M}_i \cdot \mathbf{s}_i(t))}{\mathbf{M}_i^2}\right] \cos(|\mathbf{M}_i|\delta t) + \frac{\mathbf{M}_i \times \mathbf{s}_i(t)}{|\mathbf{M}_i|} \sin(|\mathbf{M}_i|\delta t) \quad (12)$$

Here, M_i denotes the local effective field of the spin at position i. In order to obtain small errors for every iteration step, each iteration step is decomposed into several steps, updating the sublattice with some fraction of the time step δt. In this study, the spins of sublattice $\mathcal{A}$ are updated by a half time step $\delta t/2$, then the spins of sublattice $\mathcal{B}$ by a full time step δt, and finally the spins of sublattice $\mathcal{A}$ again updated by half a time step. All spins are then updated by one full time step δt. This gives a local truncation error of the order of $(\delta t)^3$.

The Suzuki-Trotter decomposition method exactly conserved the total energy of the system, because the updates correspond to a rotation about the local field. The magnetization though is not a conserved quantity. Again, see Figure 3 to compare both discussed methods for solving the equation of motion.

4 Results

To explore the behaviour of the spin system several experiments were carried out. Each experiment differs from the others in the settings of some system parameters or methods for solving the equation of motion. The parameters that were changed are the size of the bath B, the couplings of the bath Hamiltonian $\mathcal{H}_B$ and the couplings of the interaction Hamiltonian $\mathcal{H}_I$. The couplings of the subsystem Hamiltonian $\mathcal{H}_S$ were set to unity, i.e ferromagnetic. Also, for all experiments the size of the subsystem was kept at $N = 128$. Bath and Subsystem were always coupled by four interaction terms. For each experiment the initial setting of the spin in the bath was varied, resulting in different temperatures. The spins of the subsystem were initially set to be random, resulting in high values for the absolute temperature of the subsystem. After some time the two temperatures of the subsystem and bath should mix, although the temperature should mainly be driven by the much larger bath.

The process of thermalisation of the subsystem can be seen in figure 4 (a). After an initial value $e \lessapprox 0$, the energy settles at some mean value with some fluctuations. Also, for smaller couplings of the interaction Hamiltonian, the process takes longer. This is, of course, easily understandable: The energy transfer between bath and subsystem is hindered by weaker couplings.

In Figure 4 (b) the histogram $\Omega(e)$ of the subsystem energy is plotted. One can see that the distribution of the subsystem energies in fact follows a Gaußian distribution. In the following table values for the energy and specific heat of the subsystem are exemplarily displayed.

	exact	fit	measured
e	-0.291	-0.280	-0.281
c	0.24	0.22	0.21

This and the above discussed thermalisation show that the simulation does in fact show the expected physical behaviour, and also that the method used works correctly to within some error.

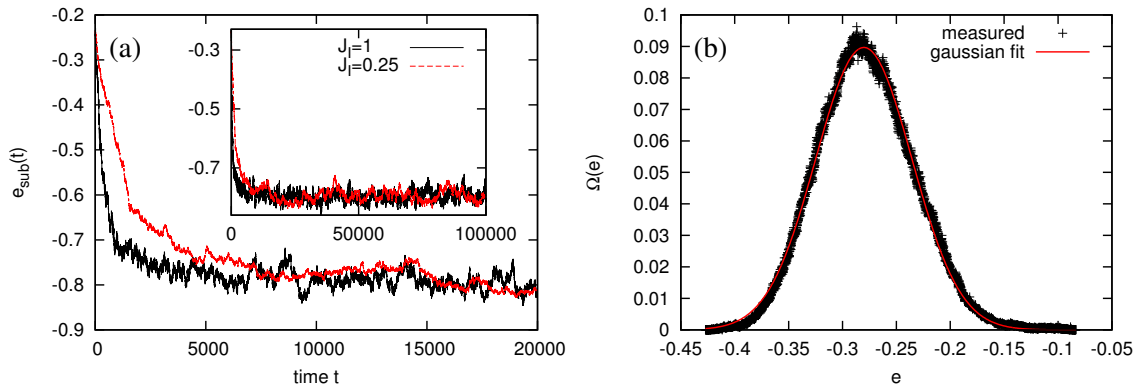


Figure 4: A simulation with the STD method with glassy interaction couplings. (a) The time series of the thermalisation process for two different couplings of the interaction Hamiltonian. The energy per spin of the subsystem settles and fluctuates around some mean value. (b) The histogram Ω of the subsystem energy at an inverse temperature $\beta \approx 0.92$.

In Figure 5 several plots of the measured subsystem energy (left) and specific heat (right) versus measured inverse temperature are shown for different experiments. The values were all measured as time averages after thermalisation of the subsystem, and compared to the exact expressions discussed above.

Comparing the Suzuki-Trotter decomposition (STD) and Predictor-Corrector (PC) method, one can see that the energy values for the PC-method are slightly higher than the exact curve. This is probably due to the earlier discussed linear increase of the energy over time. The STD-method generally delivers values closer to the exact curve.

To show the universality of the Boltzmann distribution, the width σ of the Gauß-distributed couplings of the bath were set to different values. Also the strength and type of the coupling have been systematically changed. These experiments were carried out with the use of the Suzuki-Trotter decomposition method. As one can see in Figure 5 all measured data points lie on the exact curves. The Gauß width was chosen to be quite narrow, i. e. one tenth of the mean value.

The specific heat plots show large errors, although the values are somewhat in the vicinity of the exact values. This can have several reasons. The statistical errors might be large, due to too short measured time series and the fact that the specific heat is a derivative of the measured energy values. The errors also seem to increase for $\beta \to \infty$, where the spins are in the state of absolute parallel alignment. The increasing magnetization might have an influence on this, though this is not fully understood.

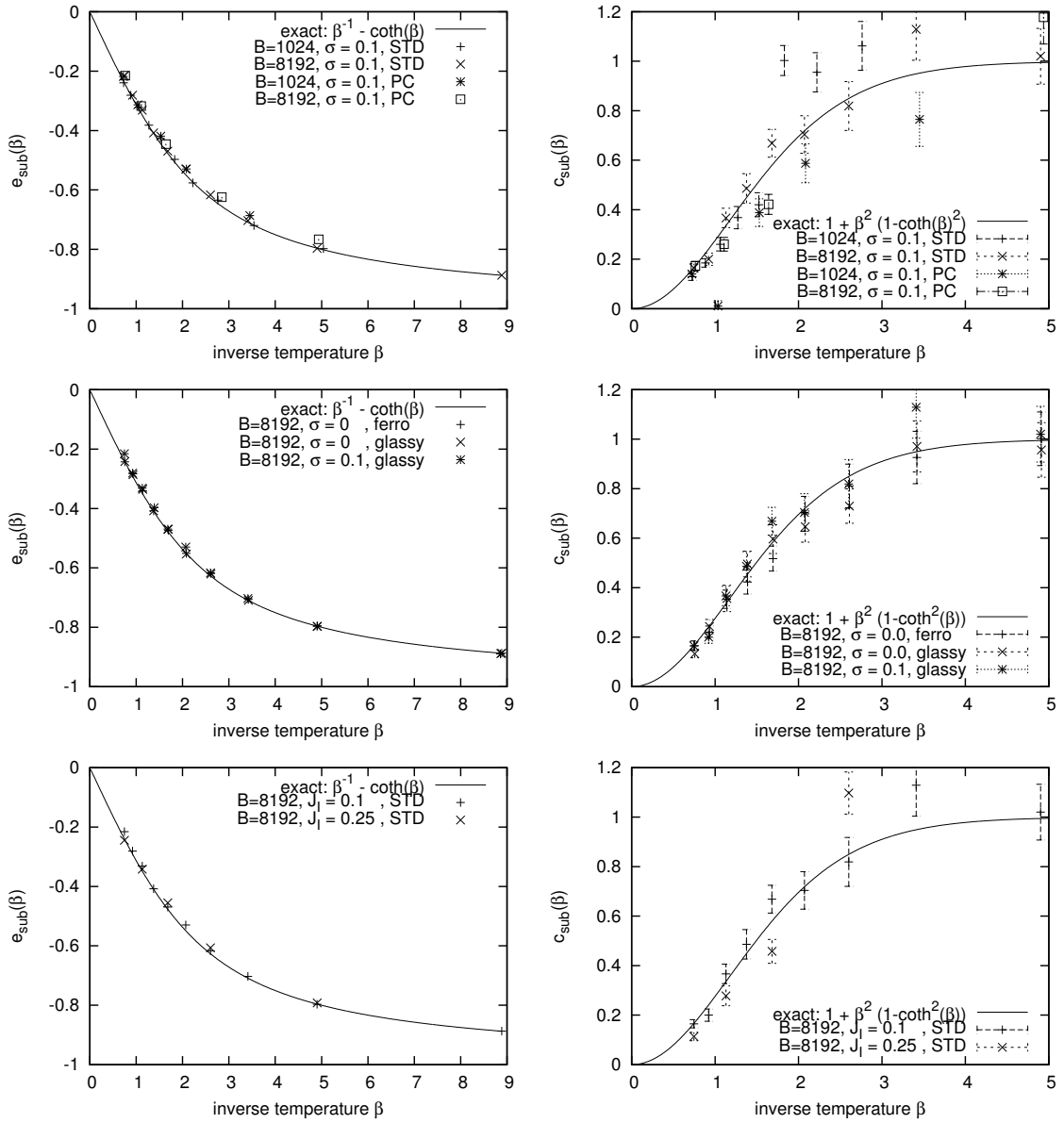


Figure 5: The energy (left) and specific heat (right) per spin of the subsystem for different settings of the system parameters and different methods for solving the equation of motion. The parameter B denotes the size of the bath, σ denotes the standard deviation of the Gauß-distributed J_{ij}s of the bath, and 'ferro' and 'glassy' denotes the type of coupling of the interaction Hamiltonian $\mathcal{H}_I$. Unless otherwise noted the absolute values of the latter are one. (top) Comparison of the two integration methods for different bath sizes B with glassy interaction couplings. (center) Comparison of the STD-method for different interaction and bath couplings. (bottom) Comparison of the STD-method for different glassy interaction couplings.

5 Conclusion and Outlook

In this exploratory study, the energy and specific heat of the subsystem were measured for the dynamic Heisenberg spin system in the microcanonical ensemble. More importantly a novel tool for measuring the temperature of the system was implemented and used. It was shown that the correct Boltzmann distribution for the subsystem could be generated in this dynamic simulation by solving the equation of motion with two different methods. Though there were some large, mainly statistical, errors in the measurement of the specific heat, it is assumed that the expression for the temperature yields correct values.

To stress this, this expression for the temperature is a very powerful tool. For the first time, it is now possible to directly compare statistical approaches to computational physics with the approaches of molecular dynamics. Furthermore, with this, the fundamentals of physics can be further explored and analysed by computer simulations in the future.

The first steps should be to further and thoroughly analyse Heisenberg spin systems. Explorations of how these systems behave for different interaction and bath couplings, for different system sizes and in higher dimensions should be considered. If this is understood, simulations of spin glass systems could be carried out, to understand their thermalisation processes. Also, simulations of the quantum-Heisenberg system could be carried out to check whether the Boltzmann distribution can be generated with these methods.

References

1. Rugh HH. *A dynamical Approach to Temperature.* Phys Rev Lett. 1997 Feb 3;78(5):772-4.
2. Jin F, Neuhaus T, Michielsen K, Miyashita S, Novotny M, Katsnelson MI, De Raedt H. *Equilibration and Thermalization of Classical Systems.* arXiv:1209.0995 (2012).
3. Fisher ME. *Magnetism in One-Dimensional Systems—The Heisenberg Model for Infinite Spin.* Am J Phys. 1964;32:343-6.
4. Nurdin WB and Schotte K. *Dynamical temperature of spin systems.* Phys Rev E 2000 Apr;61(4):3579-82.
5. Krech M, Bunker A and Landau DP. *Fast spin dynamics algorithms for classical spin systems.* Comput Phys Commun. 1998;111:1-13.
6. Ray JR and Zhang H. *Correct microcanonical ensemble in molecular dynamics.* Phys Rev E 1999 May;59(5):4781-5.

Efficient Communication Schemes for Stochastic Thermostats in parallel MD Simulations

Felix Uhl

Fakultät für Chemie und Biochemie
Lehrstuhl für Theoretische Chemie
Ruhr-Universität Bochum
Universitätsstraße 150, 44801 Bochum

E-mail: felix.uhl@theochem.ruhr-uni-bochum.de

Abstract:
A new algorithm for the parallelization of the Lowe-Andersen thermostat is presented. The implemented algorithm allows for a better control of the system's temperature compared to the original implementation in the IBIsCO code. This algorithm is more efficient than the original one, obtaining a speedup of a factor up to 14, depending on the chosen processor scheme.

---

1 Motivation

Molecular dynamics (MD) simulation techniques play an important role in the physical sciences. To control the temperature or to reach an equilibrium state in these simulations is neither a trivial nor an easy task. There are many approaches to introduce thermostats to MD simulations acting on the entire system or locally. In the present work the Lowe-Andersen (LA) thermostat [1] used in the IBIsCO code [2] is parallelized. The current implementation of this thermostat is mostly serial and produces slightly wrong results, which makes an equilibration of the system very difficult. Even slightly wrong temperatures could lead to unintended dynamics. Therefore it is important to detect the source of the temperature deviation and to correct it. Because the correct adjustment of the temperature with the LA thermostat is a very time consuming step, a parallelization improves the total performance of the program significantly.

2 Introduction

2.1 Molecular Dynamics

Under the condition that relativistic and quantum mechanical effects are neglected, the time evolution of a cartesian system is fully determined by the Newtonian equations of motion:

$$\underline{F}_i = m_i \underline{\ddot{R}}_i(t). \tag{1}$$

Here $\underline{F}_i$ and $\underline{\ddot{R}}_i(t)$ are the force acting on and the acceleration of the i-th point like particle of mass m_i, respectively.

A system consisting of N particles with the coordinates $\{\underline{R}_I\}$ is described by N equations of motion, coupled through the interaction potential $V(\{\underline{R}_I\})$:

$$\begin{aligned} M_1 \underline{\ddot{R}}_1 &= \underline{F}_1 = -\underline{\nabla}_1 V(\{\underline{R}_I\}) \\ M_2 \underline{\ddot{R}}_2 &= \underline{F}_2 = -\underline{\nabla}_2 V(\{\underline{R}_I\}) \\ &\vdots \\ M_N \underline{\ddot{R}}_N &= \underline{F}_N = -\underline{\nabla}_N V(\{\underline{R}_I\}). \end{aligned} \tag{2}$$

The equations of motion can be solved by time discretization using the velocity Verlet algorithm:

1. The new position of the particles are calculated:
   $\underline{R}_i(t + \Delta t) = \underline{R}_i(t) + \underline{v}_i(t)\Delta t + \frac{1}{2m_i}\underline{F}_i(t)\Delta t^2$
2. The new forces are calculated at the new positions:
   $\underline{F}_i(t + \Delta t) = -\nabla_i V(\{\underline{R}_I(t + \Delta t)\})$
3. Using the new and original forces, the new velocities are determined:
   $\underline{v}_i(t + \Delta t) = \underline{v}_i(t) + \frac{1}{2m_i}\left[\underline{F}_i(t) + \underline{F}_i(t + \Delta t)\right]\Delta t$
4. If $t < t_{\text{max}}$, then continue with step 1.

For an isolated system the total energy, the linear and the angular momentum must be conserved. In the case that additionally the volume and the number of particles are fixed the particles are evolved in the microcanonical or NVE ensemble. All other ensembles can formally be obtained from the microcanonical ensemble [4]. Systems that use temperature instead of energy as a control variable correspond to the canonical or NVT ensemble. In the case of an ideal gas the temperature is given by the equipartition theorem as

$$T = \frac{2}{g}\frac{\langle E_{\text{kin}}\rangle}{k_{\text{B}}}, \tag{3}$$

where g is the number of degrees of freedom of the system and k_{B} denotes the Boltzmann constant. The average kinetic energy ($\langle E_{\text{kin}}\rangle$) can be obtained from classical mechanics as:

$$E_{\text{kin}} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}m_i \underline{v}_i^2. \tag{4}$$

This relation links the temperature of a system directly to the velocities of the individual particles. To control the temperature of a system kinetic energy needs to be added or extracted from the particles using different kinds of thermostats. Although a system might have the right average kinetic energy

it might not be equilibrated. At the thermodynamic equilibrium the velocity distribution matches the Maxwell-Boltzmann distribution.

2.2 The Lowe-Andersen Thermostat

Thermostats can be classified as stochastic or deterministic depending on whether they use random numbers or not. The velocity scaling and the Berendsen thermostat [5] are two deterministic thermostats in which the velocities are rescaled. The rescaling is performed in a way that the kinetic energy gives the desired temperature. If the initial total momenta are zero, both scaling methods keep them invariant. However these two thermostats do not ensure a Boltzmann distribution of the velocities. Temperature gradients inside the system do not vanish after applying the thermostat. Another deterministic approach is the Nosé-Hoover thermostat. Trough a friction term in the equations of motion the particles are slowed down or accelerated. This approach does not perturb the dynamics of equilibrium systems significantly. However it does not satisfy Galilean invariance [6].

The Andersen thermostat [7] is based on a stochastic approach and couples the system to a heat bath. The physical idea is, that due to collisions between particles and bath particles, the simulated system adopts the bath temperature. Because a complete second temperature controlled dynamics is quite time consuming, the collisions are applied using random velocity corrections. Those corrections are performed with a predefined frequency of correction (Γ). New velocities are randomly chosen from the Boltzmann distribution and applied to the particles. The advantage of this thermostat is that it acts locally and thereby takes care of temperature gradients inside the system. Because the velocities are chosen randomly it does not conserve linear and angular momenta and is not Galilean invariant.

To overcome the disadvantages of the Andersen thermostat, Lowe modified the method using a pairwise correction to the particle velocities [1]. To reduce the computational effort, only the particles within a predefined cutoff radius r^{cutLA} (see Figure 1) are considered. The following steps are performed during a pairwise velocity correction of two particles i and j:

1. Determine the unit vector $\underline{\epsilon}_{ij}$ along the line connecting the centers of particle i and j.
2. Project the vector $(\underline{v}_i - \underline{v}_j)$ onto the vector $\underline{\epsilon}_{ij}$ to obtain $\underline{v}_{ij,\text{proj}} = \left[\left(\underline{v}_i - \underline{v}_j\right)\underline{\epsilon}_{ij}\right]\underline{\epsilon}_{ij}$.
3. Select a random velocity $v_{\text{rand}} = \left[\zeta' \sqrt{(k_\text{B}T)/(\mu_{ij})}\right]$ from the Boltzmann distribution corresponding to a desired temperature.
   ζ' is gaussian random number and $\mu_{ij} = {(m_i m_j)}/{(m_i+m_j)}$ denotes the reduced mass of the two particles.
4. Add $\left(v_{\text{rand}} \cdot \underline{\epsilon}_{ij} - \underline{v}_{ij,\text{proj}}\right)$ to $\underline{v}_i$ and subtract it from $\underline{v}_j$, to conserve the linear and the angular momenta.

The corrected velocities of two particles i and j are given by

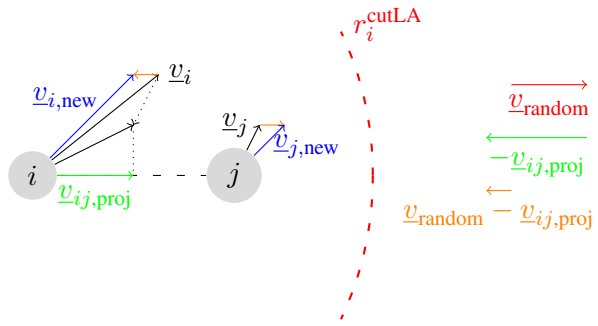


Figure 1: Pairwise correction to the velocities of the particles i and j using the Lowe-Andersen approach.

$$\underline{v}_i^{\text{new}} = \begin{cases} \underline{v}_i(t), \;\; \Gamma\Delta t < \zeta \\ \underline{v}_i(t) + \dfrac{\mu_{ij}}{m_i}\left(v_{\text{rand}} - \left(\underline{v}_i - \underline{v}_j\right)\cdot\underline{\epsilon}_{ij}\right)\cdot\underline{\epsilon}_{ij}, \;\; \Gamma\Delta t \geq \zeta\,, \end{cases}$$
$$\underline{v}_j^{\text{new}} = \begin{cases} \underline{v}_j(t), \;\; \Gamma\Delta t < \zeta \\ \underline{v}_j(t) - \dfrac{\mu_{ij}}{m_j}\left(v_{\text{rand}} - \left(\underline{v}_i - \underline{v}_j\right)\cdot\underline{\epsilon}_{ij}\right)\cdot\underline{\epsilon}_{ij}, \;\; \Gamma\Delta t \geq \zeta\,. \end{cases} \tag{5}$$

2.3 Two Algorithms for the Lowe-Andersen Thermostat

Because one particle usually has more than one neighbour the question arises how the velocity corrections should be performed within one timestep. One way would be to consider the values of the already modified velocities (LA1-method). The other method would use the initial velocities of the particles (LA2-method). In the latter case all the changes in the velocities are combined and added to the initial velocities of the particles.

LA1: Suppose that the velocity of a particle i has already been updated with $k-1$ neighbouring particles. Then the velocity of the particle i after the k-th correction is given by

$$\begin{aligned} \underline{v}_i^{\{k\}} &= \underline{v}_i^{\{k-1\}} + \frac{\mu_{ik}}{m_i}\left[v_{\text{rand},T}^{ik} - \left(\underline{v}_i^{\{k-1\}} - \underline{v}_k\right)\underline{\epsilon}_{ik}\right]\underline{\epsilon}_{ik} \\ &= \left(1 - \frac{\mu_{ik}}{m_i}\underline{\epsilon}_{ik}\underline{\epsilon}_{ik}\right)\underline{v}_i^{\{k-1\}} + \frac{\mu_{ik}}{m_i}\left(\underline{v}_k\underline{\epsilon}_{ik}\right)\underline{\epsilon}_{ik} + \frac{\mu_{ik}}{m_i}v_{\text{rand},T}^{ik}\underline{\epsilon}_{ik}. \end{aligned} \tag{6}$$

If we consider all k corrections for the LA1 method, it will result in

$$\underline{v}_i^{\{k\}} = \left[\prod_{j=1}^{k}\left(1 - \frac{\mu_{ij}}{m_i}\underline{\epsilon}_{ij}\underline{\epsilon}_{ij}\right)\right]\underline{v}_i^{\{0\}} + \sum_{j=1}^{k}\frac{\mu_{ij}}{m_i}\left(\underline{v}_j\underline{\epsilon}_{ij}\right)\underline{\epsilon}_{ij} + \sum_{j=1}^{k}\frac{\mu_{ij}}{m_i}v_{\text{rand},T}^{ij}\underline{\epsilon}_{ik}, \tag{7}$$

where $\underline{v}_i^{\{0\}}$ denotes the initial value of the velocity.

LA2: In the k-th velocity correction the initial velocity is used instead of $\underline{v}_i^{\{k-1\}}$. The corrected velocity of particle i is is given by

$$\underline{v}_i^{\{k\}} = \left(1 - \frac{\mu_{ik}}{m_i}\underline{\epsilon}_{ik}\underline{\epsilon}_{ik}\right)\underline{v}_i^{\{0\}} + \frac{\mu_{ik}}{m_i}\left(\underline{v}_k\underline{\epsilon}_{ik}\right)\underline{\epsilon}_{ik} + \frac{\mu_{ik}}{m_i}v_{\text{rand},T}^{ik}\underline{\epsilon}_{ik}. \tag{8}$$

The final velocity of particle i after k independent corrections is

$$\underline{v}_i^{\{k\}} = \left[\sum_{j=1}^{k}\left(1 - \frac{\mu_{ij}}{m_i}\underline{\epsilon}_{ij}\underline{\epsilon}_{ij}\right)\right]\underline{v}_i^{\{0\}} + \sum_{j=1}^{k}\frac{\mu_{ij}}{m_i}\left(\underline{v}_j\underline{\epsilon}_{ij}\right)\underline{\epsilon}_{ij} + \sum_{j=1}^{k}\frac{\mu_{ij}}{m_i}v_{\text{rand},T}^{ij}\underline{\epsilon}_{ik}. \tag{9}$$

Equations 7 and 9 only differ in the first term. In the first term in equation 7 a product is used whereas in equation 9 it is a sum. If every particle is corrected only once during a timestep, the two methods become identical. To study the consequence of the two approaches, both algorithms were implemented into a serial MD code [8]. 8000 particles interacting through a Lennard-Jones potential were arranged in a cubic geometry with periodic boundary conditions. The interparticle distance was chosen to be equal to the equilibrium distance. The calculations were performed using conventional reduced Lennard-Jones units [8].

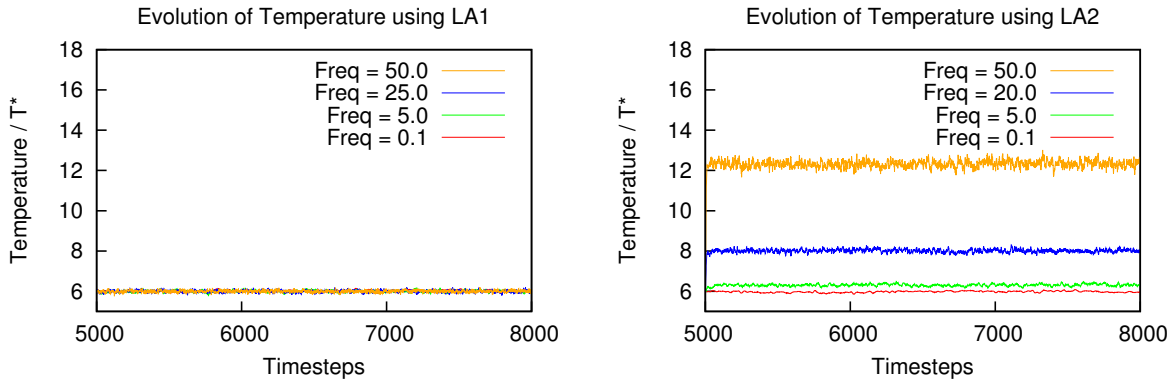


Figure 2: Time evolution of the temperature for the LA1 or LA2 method applying various correction frequencies.

For both methods a timestep of 0.001 time units (tu) was chosen. The system was equilibrated for 5 tu using velocity scaling at each timestep. Then the simulation was continued for additional 10 tu, applying the LA1 and LA2 thermostat in each timestep, respectively.

The temperature evolution during the simulation period is presented in Figure 2. In both cases the temperature is fluctuating around a constant value. The LA2 method only gives the right temperature if the frequency of correction is small enough. For bigger values of Γ the temperature is always higher than the desired temperature. In the case of $\Gamma = 50$ it is roughly twice the target temperature.

In the case of the LA1 method the current temperature is fluctuating around the target temperature for all frequencies.

2.4 The Difficulty of Parallelizing the Lowe-Andersen Thermostat

As shown above it is important to always use the current velocities to perform further corrections. This leads to problems in the parallelization, independent of the parallelization scheme (volume elements are assigned to processors; groups of particles are assigned to processors depending on their index; et cetera). Assume a pair of particles is selected for the Lowe-Andersen correction. As long as both particles are assigned to the same processor the current velocities of both particles are known to the processor. But if the particles are assigned to different processors, the correcting processor has to know the current velocity of both particles. That means that, if the velocity of a particle near the boundary gets corrected, the updated velocity has to be send to the neighbouring processors before the velocity of the same particle gets updated with the old velocity. Also both particles must not be involved in further correcting processes during a correction. This is necessary to ensure a pure LA1 correction of all particles.

3 The IBIsCO Code

The IBIsCO code [3] is an MD software developed for coarse-grained simulations in which the numerical potentials are derived by iterative Boltzmann inversion. IBIsCO is written in Fortran, and it is parallelized using MPI [2]. A periodic simulation box is decomposed, using an $NPX \times NPY \times NPZ$ scheme, into domains which are assigned to different processors. Certain properties of particles localized near the processor boundaries are exchanged with the neighbouring processors. For the interaction of the particles a boundary area is defined on the processors by the interaction cutoff radius.

3.1 Original Implementation of the Lowe-Andersen Thermostat

The original implementation of the LA thermostat in the IBIsCO code is performed as follows:

1. The components of the velocities and of the coordinates as well as the global indices are collected on the master processor using seven indipendent `MPI_GATHERV` calls.
2. All collected information is rearranged considering their global index.
3. The velocity correction is successively performed for each processor domain by the master processor.
4. The velocities are sent back to the host processors using three independent sending processes.

This implementation leads to several problems. A large amount of memory is allocated on the master processor for the positions, velocities and indices of the particles in the system, which might lead to a lack of memory for large simulations. A second problem is the serial character of this quite expensive method, which leads to a significant slowdown and an inefficient hardware usage.

4 Implementation of the modified Lowe-Andersen Thermostat

To eliminate the flaws and imperfections of the original Lowe-Andersen implementation, the IBIsCO code was modified. Besides a pure LA1 correction the modified algorithm aims to minimize the workload on the master processor, the data movement on the processors as well as to optimize the data movement between processors.

To guarantee a pure LA1 thermostat, the pair corrections are performed in every domain by the host processors. Only pairs of which both particles are located in the domain are taken into account. All remaining corrections need to be performed across the processor boundaries. They are carried out by the master processor. In this way no particle can be corrected by two different processors at the same time. Compared to the original implementation this version also reduces the amount of data that needs to be sent to the master processor. Because this algorithm is still partially serial, further optimizations have to be performed (see section 6).

The number of sending processes are minimized by copying the velocities, coordinates and the global index on the host processors to a single array and collect them with a single `MPI_GATHERV` call. To send back the corrected velocities to their host processors a single `MPI_SCATTERV` call is used.

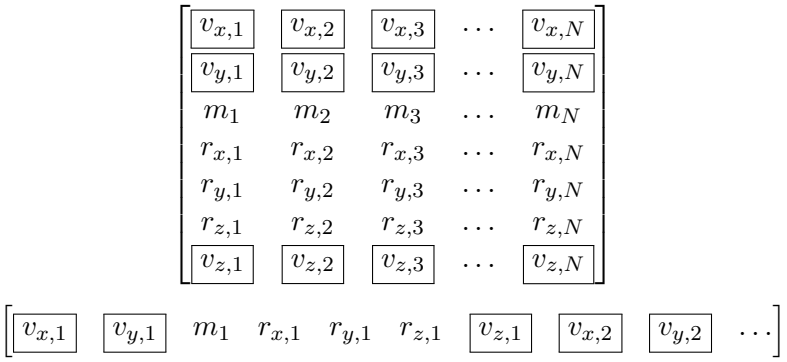


Figure 3: **Top:** Arrangement of the imported data on the master processor as a matrix. **Bottom:** Real arrangement of the data in the memory. Boxed entries are chosen by the `MPI_DATATYPE` mask for the back sending process. v are the velocities, m the masses and r the positions of the particle i; x, y and z denote the cartesian components.

The data movement on the master processor is reduced by not copying the received data to new arrays, but by accessing it then directly from the imported array (figure 3). To guarantee the velocities to be sent back efficiently, a handle (`MPI_DATATYPE`) was defined. It selects from the matrix in Figure 3 only the velocities. Because MPI can only handle gaps in the data, but not at the end of the `MPI_DATATYPE`, the z component of the velocities and the masses of each particles were exchanged. This leads to an unusual data arrangement, but can be handled more efficiently by MPI.

5 Analysis of the modified Lowe-Andersen Algorithm

All following calculations were performed with the IBIsCO code using the original and the modified Lowe-Andersen implementation, respectively.

In a first test the efficiency of the temperature adjustment of both implementations was investigated. Therefore a system consisting of 160000 coarse-grained particles contained in a cubic box with a side length of $30.30\ \mathrm{nm}$ was set up and was simulated for $10\ \mathrm{ps}$ using a timestep of $10\ \mathrm{fs}$. The target temperature was set to $450\ \mathrm{K}$ and the correction frequency was $\Gamma = 0.5$. A LA-cutoff radius of $1.6\ \mathrm{nm}$ was used.

The temperature evolution over time is presented in Figure 4. For both methods the instantaneous temperature fluctuates around a constant value. The new implementation matches the target temperature on average whereas the original implementation deviates by $20\ \mathrm{K}$. By analysing the code it was discovered that in the original implementation the velocities were sent back to the host processor directly after the corrections inside the corresponding domain were performed. Further corrections to those particles lead to a partial LA2 correction scheme and gives higher temperatures. Another advantage of the new implementation is the smaller temperature fluctuations compared to the original implementation. Those larger fluctuations also result from a partial LA2 correction (compare Figure 2).

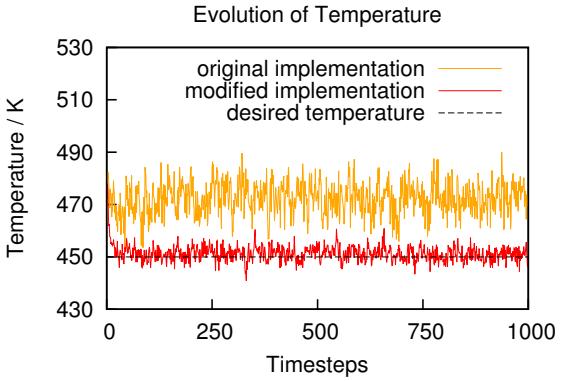


Figure 4: Time dependent temperature evolution for the original and modified implementation of the Lowe-Andersen thermostat.

To investigate the speedup of the new implementation, the above simulations were repeated for different sizes of the processor domains. Therefore the simulation box was divided equally in each direction into 2, 3 or 4 domains, which results in 8, 27 and 64 processors.

The CPU times are presented in Figure 5 and table 1. Figure 5 shows that the original implementation scales poorly with the number of processors. In addition to the better scaling behaviour, the modified method needs less total computation time. A comparison of CPU times reveals a speedup of a factor of six to fourteen, depending on the number of processors (compare table 1). The speedup seems to converge with a higher number of processors. The more processors are used, the shorter the time needed for the particle corrections on the processor. At the same time the total size of boundary regions increases. Therefore more particles need to be sent to the master processor, which results in a longer serial calculation.

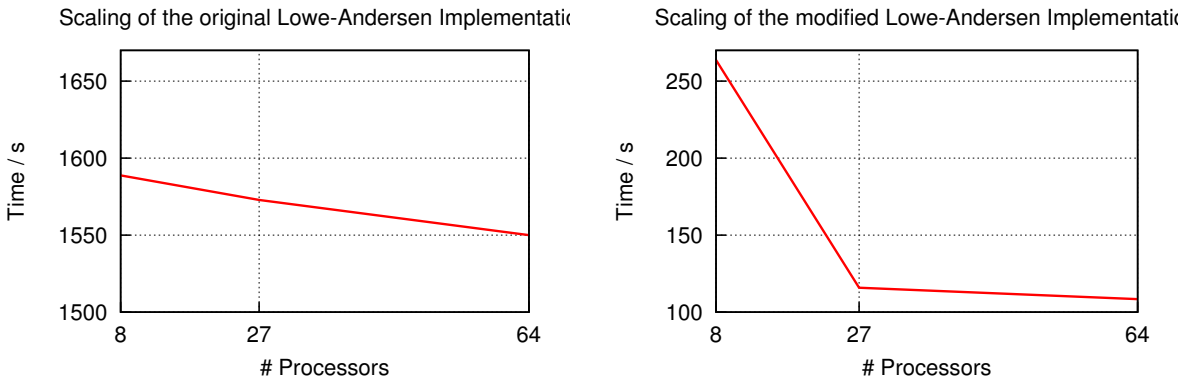


Figure 5: Comparison of the simulation times using the modified and original Lowe-Andersen implementation with a varying amount of processors.

Table 1: Comparison of the simulation times using the modified and original Lowe-Andersen implementation with a varying amount of processors.

# proc.	time(original)/s	time(modified)/s	$\frac{\text{time(original)}}{\text{time(modified)}}$
8	1588.8	263.7	6.03
27	1572.9	115.8	13.58
64	1550.0	108.4	14.30

To investigate this effect of the different size of the boundary area on the speedup, a larger system of 540000 particles was simulated using the same conditions as for the previous tests. A total number of 96 Processors was used. The simulation box was divided in different ways (compare table 2) into $NPX \times NPY \times NPZ$ processors. This results in different sizes for the boundary regions. In table 2 the ratio of the average number of sent particles ($\langle N_{\text{sent}} \rangle$) and the total number ($N_{\text{tot}}$) of particles are presented.

Table 2: Comparison of the simulation time using the original and modified Lowe-Andersen implementation with 96 processors. The speedup depends on the ratio of total number of particles and the average sent particles.

System division	$\frac{\langle N_{\text{sent}} \rangle}{N_{\text{tot}}}$	time(original)/s	time(modified)/s	$\frac{\text{time(original)}}{\text{time(modified)}}$
$NPX = 2, NPY = 6, NPZ = 8$	0.430	2565.8	356.0	7.208
$NPX = 4, NPY = 3, NPZ = 8$	0.405	2517.4	270.6	9.302
$NPX = 4, NPY = 6, NPZ = 4$	0.378	2865.3	284.4	10.076

The ratio of the simulation times needed for the simulation rises as less particles need to be sent to the master processor in the new implementation. This is due to more corrections being performed inside the processor domains. In both cases only the ratio between the CPU times are representative, because the rest of the simulation also depends on the division of the simulation box.

6 Conclusions and Outlook

In this project the Lowe-Andersen thermostat implemented in the IBIsCO code was successfully improved. The original implementation of the LA thermostat in the IBIsCO code gives higher instantaneous temperatures than demanded. This is due to a partially LA2 controlled velocity correction. The modified version of the parallelization gives the correct temperature with less fluctuations.

The modified version is faster up to a factor of 14 depending on the number of processors and the division scheme of the system. The LA thermostat algorithm can be further improved using a direct communication between the neighbouring processors. In such a case one processor will send the information from the boundary region to 13 of its 26 neighbouring processors ($6 \times$ side, $12 \times$ edge, $8 \times$ corner) and receive data from the other 13 processors. If implemented in the right way, this leads to a totally parallelized pure LA1 scheme.

7 Acknowledgement

I would like to thank Dr. Godehard Sutmann and Dr. Viorel Chihaia from JSC for their support and answering many questions. I would also like to thank Prof. Dr. Dominik Marx at Bochum University for his recommendation, which made my attendance possible. Special thanks to Mathias Winkel and Ivo Kabadshow for organizing the guest student programme. I would like to thank my fellow students for the great time in Jülich.

References

1. Lowe CP. *An alternative approach to dissipative particle dynamics*. Europhys Lett. 1999;47(2):145-151
2. Karimi-Varzaneh HA, Qian H-J, Chen X, Carbone P, Müller-Plathe F. *IBIsCO: A Molecular Dynamics Simulation Package for Coarse-Graind Simulation*. J Comput Chem. 2011;32(7):1475-1487
3. IBIsCO [homepage on the Internet]. Theoretical Physical Chemistry TU Darmstadt: Florian Müller-Plathe; [updated 2008; cited 2012 Oct 18]. Available from: `http://www.theo.chemie.tu-darmstadt.de/ibisco/IBISCO.html`
4. Marx D, Hutter J. *Ab Initio Molecular Dynamics*. Cambridge University Press; 2010
5. Berendsen HJC, Postma JPM, van Gunsteren WF, DiNola A, Haak JR. *Molecular dynamics with coupling to an external bath*. J Chem Phys. 1984;81:3684-3690
6. Koopman EA,Lowe CP. *Advantages of a Lowe-Andersen thermostat in molecular dynamics simulations*. J Chem Phys.2006;124:204103
7. Andersen HC. *Molecular dynamics simulations at constant pressure and/or temperature*. J Chem Phys.1980;72:2384
8. Frenkel D, Smit Berend. *Understanding Molecular Simulations: From Algorithms to Applications*. Academic Press; 2008

Identification of Gravity Waves in AIRS Brightness Temperatures

Anne Springer

University of Bonn
Institute for Geodesy and Geoinformation
Nußallee 17
53115 Bonn

E-mail: annespr@uni-bonn.de

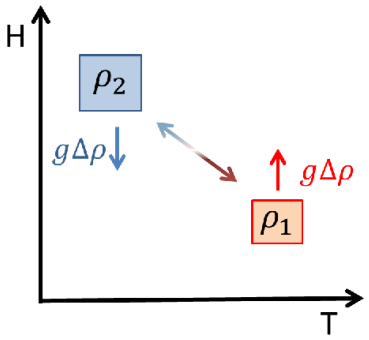
Abstract:
The Atmospheric Infrared Sounder (AIRS) provides infrared radiance data, which are used to calculate brightness temperatures. Stratospheric gravity waves can be found in temperature perturbation data. A toolbox is developed to identify gravity waves in the AIRS data and to analyse their properties. This information can be used to gain a better understanding of gravity wave sources and their propagation in the stratosphere. The output of our toolbox is a statistic of amplitudes and wavevectors. Case studies give information about the functionality of the toolbox and reveal the dependency of the results on certain control parameters. Gravity waves are detected with success and the wavevectors and corresponding amplitudes are determined with good accuracies.

1 Introduction

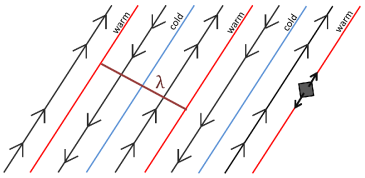
Gravity waves play an important role in the circulation of the middle atmosphere. The current public debate about climate related topics, such as global warming, evidence the need for reliable global climate and circulation models. The stratospheric and mesospheric circulation can only be explained by considering the momentum and the energy which is transported by gravity waves. Hence, a deeper understanding of the origin and propagation of gravity waves will permit a better modelling of the circulation in the atmosphere. A review about the effects and the role of gravity waves in the middle atmosphere is given by [5].
In this work we identify gravity waves in Atmospheric Infrared Sounder (AIRS) brightness temperature data. Different analysis methods, which determine the properties of gravity waves, are assembled in a gravity-wave-toolbox as described in section 4. On the one hand we use the Fast Fourier Transformation (FFT) and on the other hand a sine fit.
In section 2 we introduce the phenomena of gravity waves and briefly respond to the sources of gravity waves. We present some background information about the AIRS data in section 3. In section 5 this report concludes with a demonstration of the results of our work by means of a concrete case study.

2 Atmospheric Gravity Waves

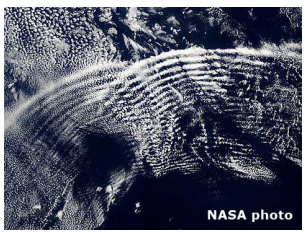
Gravity waves are observed in temperature, wind and density. They transport momentum from lower altitudes into the mid and upper atmosphere. Their existence is pictured by considering the stable stratification of the atmosphere [2]. A thought experiment (figure 1a) illustrates the essential mechanism. An air parcel of density ρ_1 is shifted upwards by some event. In a next step it cools down and extends. As a consequence the density of the air parcel decreases. But the density of the background atmosphere also decreases with a certain amount. At some moment the air parcel is heavier than the background atmosphere. Consequently gravity provokes a downward shift. All in all, the air parcel oscillates around its equilibrium position. This is the mechanism which occurs in gravity waves.
There are different sources of gravity waves. Well known initiators are airflow over mountains and convection. Mountains provoke an initial upward shift of the air parcels. Convection transports heat from lower to higher altitudes, which produces a warm front and activates gravity waves. More detailed information to these and other sources are given in [5]. In Figure 1b the propagation of a gravity wave is depicted. Again we examine the behaviour of one air parcel. Starting with a warm front, in a next step our air parcel is shifted upward along this warm front. Then it will cool down, become less dense and at a certain point it moves downwards again. Consequently, the air parcel is oscillating perpendicular to the propagation direction of the wave. In propagation direction we observe a change of warm and cold and a change of upward and downward wind. The changes are related to the wavelength λ.
It is obvious that the wave can be detected by regarding either the temperature, or the up and downward winds or the changes in density. By the way, due to the changes in temperature, clouds unveil gravity waves. They condense at the cold fronts but not in warmer parts of the air. Figure 1c shows, seen from space, clouds which uncover gravity waves.



(a) The stable atmosphere: an air parcel with density ρ_1 is shifted upwards, cools down, expands and has a lower density. The background atmosphere also changes its density. At a certain point the air parcel is heavier than the background atmosphere and is shifted down again due to gravity.



(b) The gravity wave with wavelength λ propagates to the right. After a warm front, an upward wind follows, then a cold front, a downward wind and again a warm front. The air parcels oscillate perpendicular to the propagation direction.

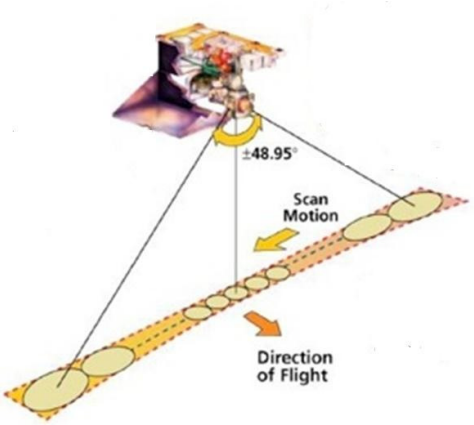


(c) Gravity waves are unveiled by clouds. In this Figure we show gravity waves as seen from space.

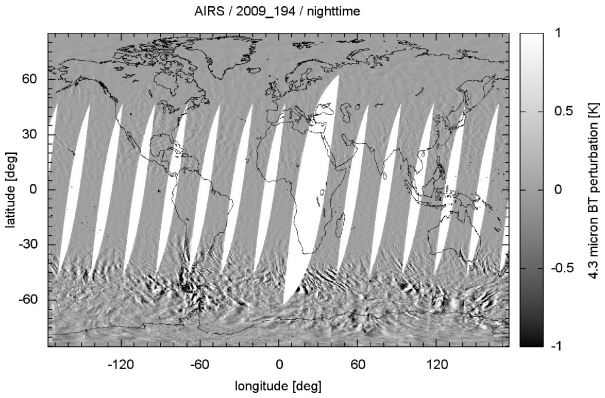
Figure 1: Gravity waves occur in a stable atmosphere and are unveiled by clouds.

3 AIRS Data

The Atmospheric Infrared Sounder (AIRS) instrument is one of six Earth observation instruments on board of the AQUA satellite. The mission was launched in 2002 in order to obtain information about the Earth's water cycle. The AQUA satellite orbits the Earth one times in 99 minutes, so about 14 times a day. The AIRS instrument is a cross-track scanning instrument. Each scan has an extension of 1600 km. The size of the 90 footprints is bigger at the extreme edges of the scan than in nadir direction. The measuring principle is pictured in Figure 2a.



(a) Measuring principle of the AIRS instrument: Infrared energy is measured in 2378 different channels for scans of 1600 km extension in across-track direction (source: [6]).



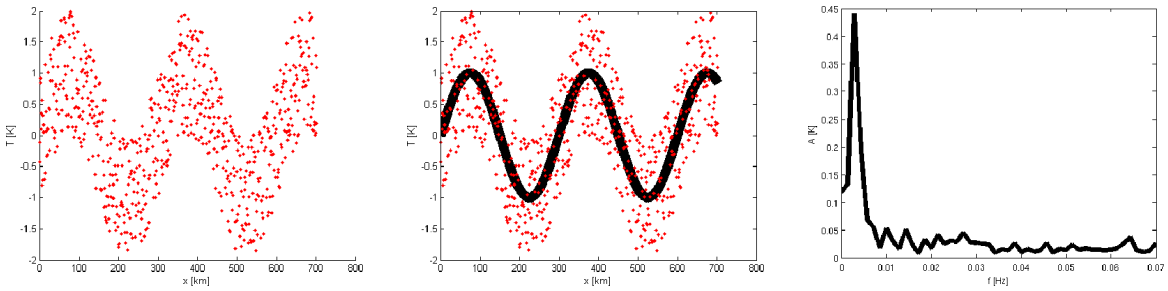
(b) For this Figure brightness temperature data from the channels which are sensitive to gravity wave were computed. The background temperature was subtracted and the remaining perturbation data contains the wave structures we want to analyse.

Figure 2: The AIRS instrument on board of the AQUA satellite provides infrared radiation data, from which brightness temperature data can be deduced.

The AIRS instrument measures infrared energy in 2378 channels [1]. Each wavelength is sensitive to the temperature over a certain range of height in the atmosphere. Gravity waves can be examined by using the channels which see the altitudes between 30 - 40 km. An example for 12 hours of AIRS data is shown in Figure 2b. The infrared radiation has been converted to brightness temperatures using the Planck function. In the Figure the background temperature has been subtracted in order to make smaller structures visible. This brightness temperature perturbation data is subject to our further investigations.
In the lower latitudes we can clearly distinguish the distinct descending paths. In the southern hemisphere different wave structures can be determined. To identify and model these waves we developed the gravity-wave-toolbox, which is presented in the next section.

4 The Gravity-Wave-Toolbox

The essential part of the gravity-wave-toolbox are two methods: the Fast Fourier Transformation (FFT) and the sine fit (figure 3). First of all we want to specify our problem and then develop a strategy to solve it.
The aim of this work is to develop a tool which calculates the amplitude and the wavevector at every data point. We begin with a view to the linear wave theory.



(a) This Figure shows a data set with one sine and superposed noise. The signal has a wavelength of about 300 km.

(b) One sine function is fitted to the data.

(c) The FFT reveals the most important wavenumber, which corresponds in this case to a wavelength of 300 km. But energy is also transferred to other wavenumbers.

Figure 3: We use two methods to analyse our data: FFT and sine fit

4.1 Linear Wave Theory

Gravity waves propagate three dimensional and the amplitude increases with the height exponentially if the wave propagates conservatively and if the background remains constant [5]. From the linear wave theory we get the description of a three dimensional wave as shown in equation 1. The wave is determined by the wavelengths in x-, y- and z-direction λ_x, λ_y and λ_z, the phase ϕ and the amplitude $T_0 exp(\frac{z-z_0}{2H})$. T_0 is the temperature perturbation at some reference height z_0 and $H \approx 7km$ the atmospheric scale height.

$$\Delta T = T_0 exp(\frac{z-z_0}{2H}) \sin(\frac{2\pi}{\lambda_x}x + \frac{2\pi}{\lambda_y}y + \frac{2\pi}{\lambda_z}z + \phi) \tag{1}$$

As we analyse extracts of wave structures we neglect the exponential growth of the amplitude. Furthermore we examine two dimensional data, so that the third component λ_z of the wavevector is omitted. Therefore we obtain the model presented in equation 2.

$$\Delta T = A \sin(\frac{2\pi}{\lambda_x}x + \frac{2\pi}{\lambda_y}y + \phi) \tag{2}$$

Based on the given brightness temperature perturbations ΔT we aim to determine the highlighted parameters.

4.2 Analysis by Fast Fourier Transformation

The Fast Fourier Transformation (FFT) is applied to calculate the values of the Discrete Fourier Transformation (DFT) in an efficient way. Equation 3 defines the DFT for a one dimensional data set of size N. Input are the temperature perturbations ΔT_j and we obtain the fourier coefficients c_k.

$$c_k = \sum_{j=0}^{N-1} \exp(-2\pi i \frac{jk}{N} \Delta T_j) \tag{3}$$

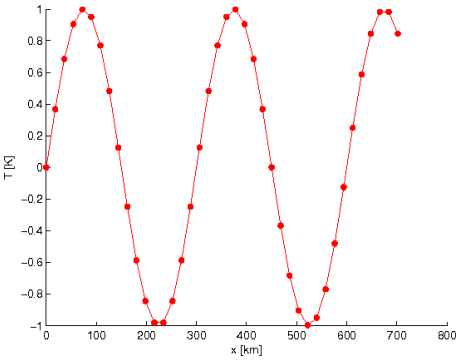
We extend the one dimensional FFT for two dimensional data by executing the FFT first on each row and then columnwise on the obtained results. From the coefficients c_k we deduce the amplitude A and the phase ϕ at each frequency. The wavenumbers respectively the wavelengths are obtained from the spacing of our data points. The resolution of the wavenumbers is limited as determined by equation 4. It depends on the sampling rate F_s of the input data and the number N of samples.

$$f_{res} = \frac{F_s}{N} \tag{4}$$

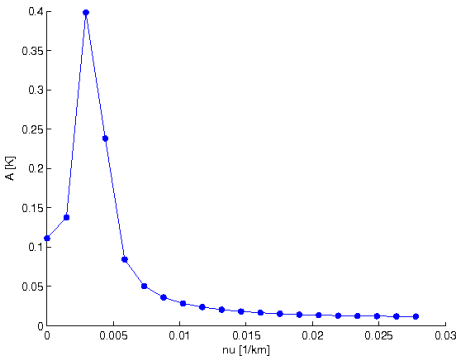
In our study the spacing of two data points is about 18 km. Consequently our sampling rate is $F_s = \frac{1}{18km} = 0.0556 \frac{1}{km}$. Let us now suppose a number of $N = 40$ data points. Then we obtain for the wavenumber resolution a value of $f_{res} = \frac{0.0556}{40} \frac{1}{km} = 0.0014 \frac{1}{km}$.

Figure 4a pictures the example of a sine with a wavelength of 300 km. From the FFT we obtain the amplitudes for each wavenumber. They are shown in Figure 4b. We select the wavenumber with the highest amplitude as result of our analysis. The wavelengths corresponding to the three maximum amplitudes are 684 km, 342 km and 228 km. Due to the limited resolution 342 km is the best approximation we can obtain for our signal wavelength. The analysis returns exactly the same value of 342 km for a signal with a wavelength of 400 km or 450 km. In summary the resolution of the wavenumber is worse for large wavelengths.

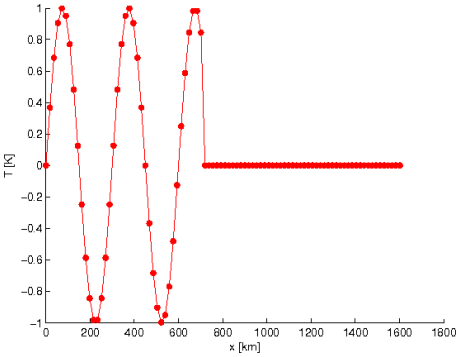
There are two possibilities to overcome the problem of the limited resolution. As equation 4 indicates we can either decrease our sampling frequency, which corresponds to a closer spacing of the data points, or increase the number of samples.
We can reach a closer spacing of our data points by interpolation. The number of samples can be increased by applying zero padding. That means, we add a certain number of zeros to our data (figure 4c). As we can see, the values of the corresponding spectrum in Figure 4d are spaced much closer than before in Figure 4b. The disadvantage of the zero padding is the appearance of side lobes. The extension of the data set with zeros corresponds to a multiplication with the boxcar-function. The fourier transformed of the boxcar function is the sinc-function, which causes the mentioned side lobes. Additionally, the energy is distributed to different wavenumbers with the effect, that the amplitudes decrease. We accept this to gain a better resolution of the wavenumbers. These effects are lowered by using a different window than the boxcar function. We use a Bartlett window.



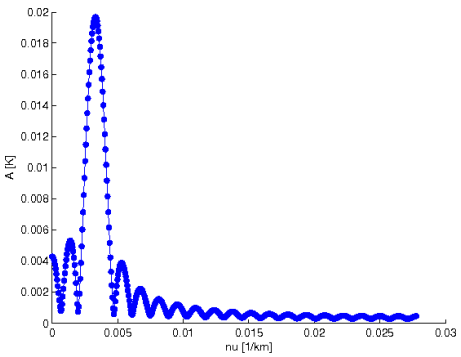
(a) A sine with a wavelength of 300 km.



(b) Result of the FFT applied on the data of Figure 4a. The resolution is rather low.



(c) A certain number of zeros is added to the sine of Figure 4a.



(d) The spectrum of the fourier transformed of the extended signal reveals side lobes, but has a better resolution than the result displayed in Figure 4b.

Figure 4: The resolution of the result of the FFT is limited. It can be improved by zero padding.

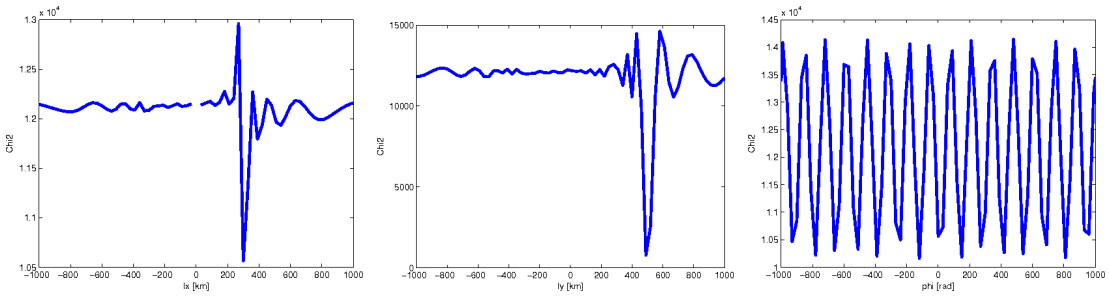
4.3 Analysis by Sine Fit

Regarding the AIRS data there is mostly exactly one wave which stands out of a certain region. Consequently we want to fit exactly one sine to our data. This is achieved by applying the least squares method. We minimize the cost-function, which is the sum of the squared residuals of every observation, as indicated by equation 5. The residuals are defined as the difference between the observations ΔT_j and the corresponding result from the model displayed in equation 2 using the estimated parameters A, λ_x, λ_y and ϕ. The model of equation 2 is two dimensional. We can modify the model to fit a one dimensional sine by omitting either the x-term or the y-term.

$$\chi^2 = \sum_{j=0}^{N-1} r_j^2 \rightarrow min \tag{5}$$

$$r_j = \Delta T_j - A \sin(\frac{2\pi}{\lambda_x} x_j + \frac{2\pi}{\lambda_y} y_j + \phi) \tag{6}$$

We are dealing with a non-linear model, which is solved by the Gauß-Newton-Algorithm. Therefore we need starting parameters, which can be taken from the results of the FFT. Convergence is not guaranteed and depends strongly on the quality of the starting parameters. Figure 5 shows three dimensions of the cost function of a wave with wavelength $\lambda_x = 300$ km and $\lambda_y = 500$ km. Figure 5a pictures the cost-function in λ_x-direction, Figure 5b in λ_y-direction and 5c in direction of the phase. For the amplitude the minimization problem is linear and the cost-function quadratic. As we can deduce from the Figures, bad starting parameters will lead to a local minimum and not necessarily to the required global one.



(a) Cost function for the parameter λ_x (b) Cost function for the parameter λ_y (c) Cost function for the parameter ϕ

Figure 5: Cost function plotted along the axis of different parameters for a data set which contains a wave with $\lambda_x = 300$ km and $\lambda_y = 500$ km.

To overcome this problem we automatically test a certain number of starting parameter sets. Then we select the result which leads to the smallest value of the cost function.

4.4 Technical Aspects

The two methods are applied for extracts of our data set. That means, we take a box of size NxN and shift it over our data. The perturbation data which is contained in this box, is analysed by the FFT and the sine fit. The parameters are saved for the point in the centre of the box.
Alternatively we can also apply FFT and sine fit only in one dimension: in x- or in y-direction. In this case, instead of the box we work with a stripe in x-or y-direction and shift it over the data set.
Before applying our analysis methods some other steps have to be accomplished. The proceeding of our work is visualized in Figure 6.

First of all we have to subtract the background atmosphere to reveal the gravity wave structures. Therefore we fit a polynomial of degree four to the data for each across-track scan and subtract it from the data. We obtain the brightness temperature perturbation data. As we already mentioned in section 3 the footprints at the extreme borders of each scan are bigger than those in nadir direction. As the FFT requires equally spaced data we interpolate the data in across-track direction. Furthermore we can use the interpolation step to increase the sampling rate in order to obtain a better resolution of the FFT. Optionally the data can be smoothed with a median or a moving average filter. Furthermore we calculate the variance of the data to decide if the area manifests enough variation to contain a gravity wave. The thus prepared data is now analysed by our gravity-wave-toolbox. We choose a box size NxN. For example 40x40 data points, which correspond to about 720x720 km, have proved suited. Now we accomplish the FFT with the option of zero padding. We have the possibilities to achieve the analyses in

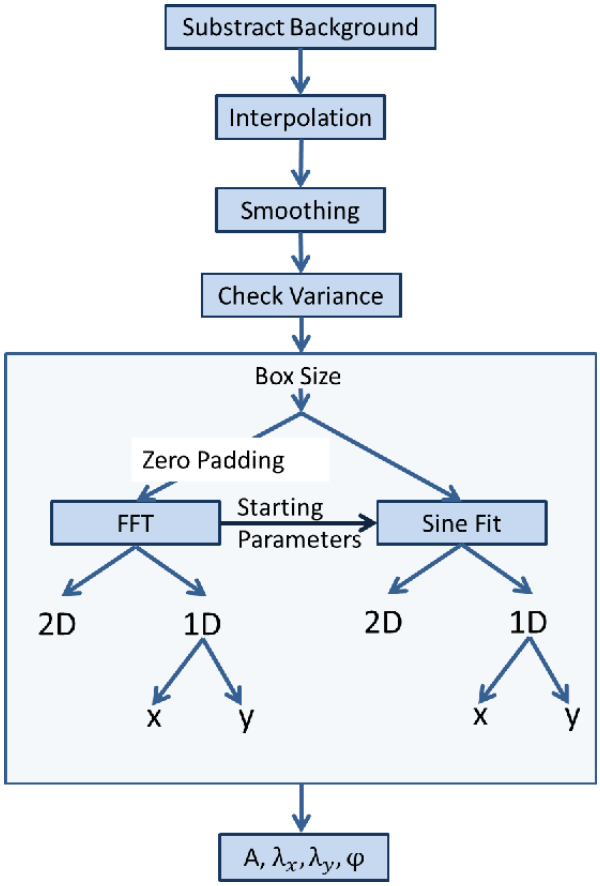


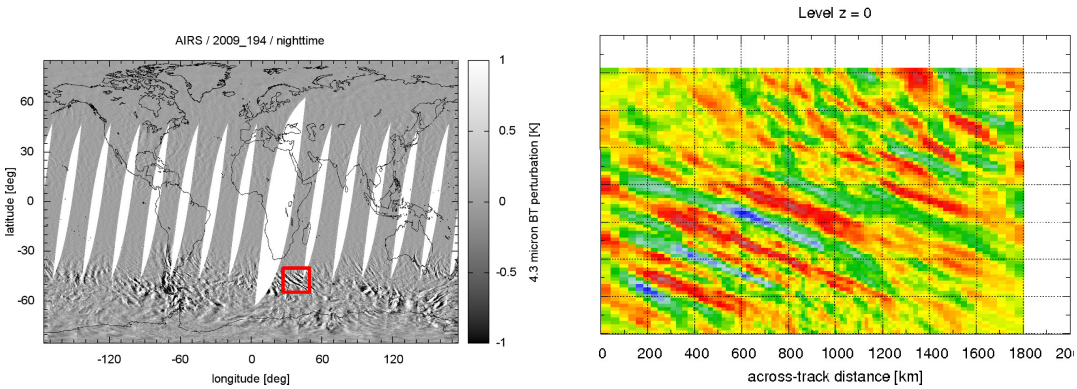
Figure 6: The proceeding to identify gravity waves is as following: First we subtract the background. Then we interpolate the data, smooth it if necessary and check the variance. Afterwards the gravity-wave-toolbox is applied and gives the wave parameters.

two dimensions or in one dimension, in x- or y-direction. The results from the FFT are used as source for determining suited starting parameters for the sine fit. This analysis also can be accomplished in the different dimensions and directions.
As a result we obtain for each option the parameters amplitude A, wavelength λ_x, wavelength λ_y and phase ϕ at each data point. The parameters are plotted to obtain a statistic about their magnitude at each location.

5 Case Studies

In order to test our analysis toolbox we carried out some case studies on selected examples of the AIRS data. But we also tested different strategies on synthetic data. We now present results from the analysis of a gravity wave in the south-east of the African continent. We work in the coordinate system of the satellite. The x-axis corresponds to the across-track direction and the y-axis to the along-track direction.

Figure 7a shows the selected extract in detail. The gravity wave is situated in the lower left part. We



(a) The red rectangle shows the extract of perturbation data which is analysed.

(b) The extract is plotted in the coordinate system of the satellite with the x-axis in across-track direction and the y-axis in along-track direction

Figure 7: In order to demonstrate our gravity-wave-toolbox we chose the case of a gravity wave in the south of the African continent.

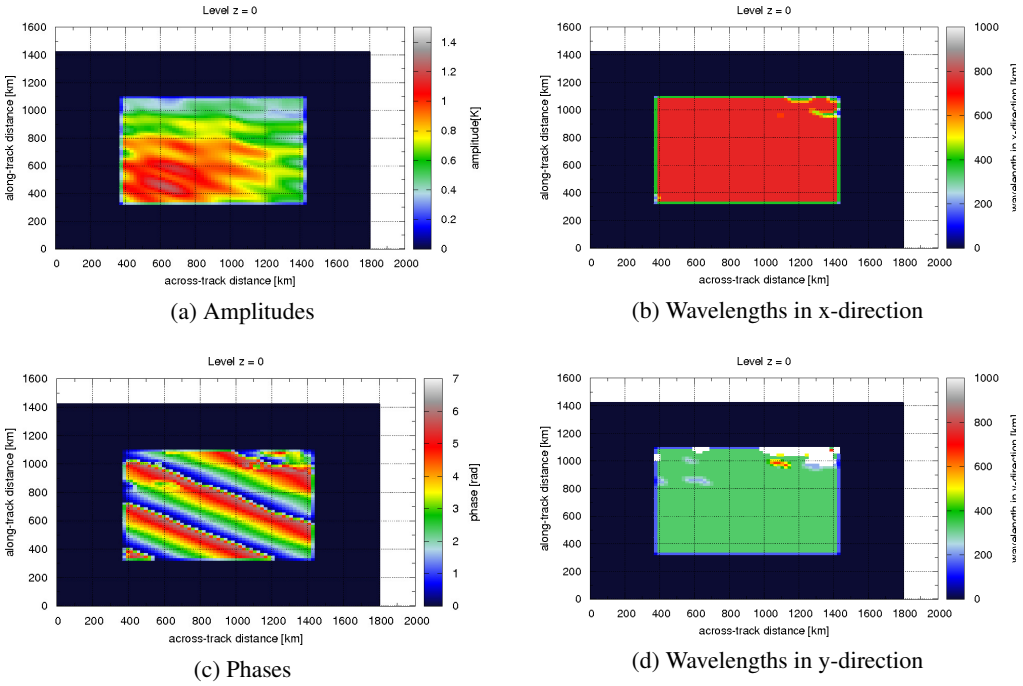
can determine an amplitude of about [1.5]K, a wavelength in x-direction of about 600 km and in y-direction of about 300 km.
We will now take a look on the results we obtain from our analysis tool. First of all we accomplish a FFT in two dimensions with a box size of 40x40 data points. The results are displayed in Figure 8. For each data point we have calculated from its surrounding box amplitude, wavevector and phase.

As Figure 8a indicates the amplitude is estimated with up to 1.3 K lower than the real amplitude. This is due to the distribution of the energy to other wavenumbers. As expected the amplitude decreases to the right and upper part of the extract. The phase (figure 8c) shows clearly the wavefronts. The wavevector is estimated homogeneously in the whole extract. In x-direction we get a wavelength of 800 km and in y-direction of 400 km. Thus, the result from the FFT is only a bad estimate for the wavelengths of 600 km and 300 km. Reason is the low resolution of the FFT.
The application of zero padding leads to much better results. In Figure 9a mainly two wavelengths for the x-direction are distinguished. The obtained 600 km fit to the extract as well as the 300 km in y-direction.

The approximated values from FFT were used as starting parameters for the sine fit and provide the results pictured in Figure 10.

The gaps correspond to areas in the data where the structure of the perturbation data cannot be represented by a sine. These areas are confirmed by a look on Figure 7. The amplitudes and the phase found by the sine fit correspond to our former observations. But the great advantage of the sine fit method is the good resolution of the wavelengths. As Figure 10b pictures, the wavelength increases in x-direction an the expected value of 600 km is yielded. In y-direction a more or less homogeneous wavelength of 300 km is confirmed.

Variation of the box size has shown, that, especially for the sine fit, its size should not be chosen too small. In this case study a size of 40 x 40 data points turns out to be appropriate. As experiments



(a) Amplitudes

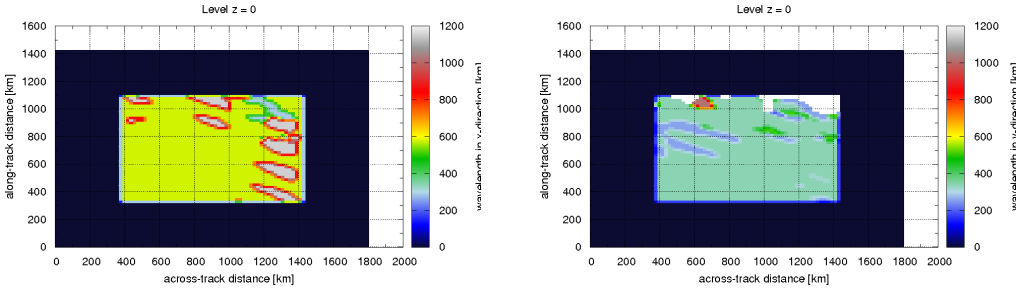
(b) Wavelengths in x-direction

(c) Phases

(d) Wavelengths in y-direction

Figure 8: The FFT is applied using a box of size 40x40.

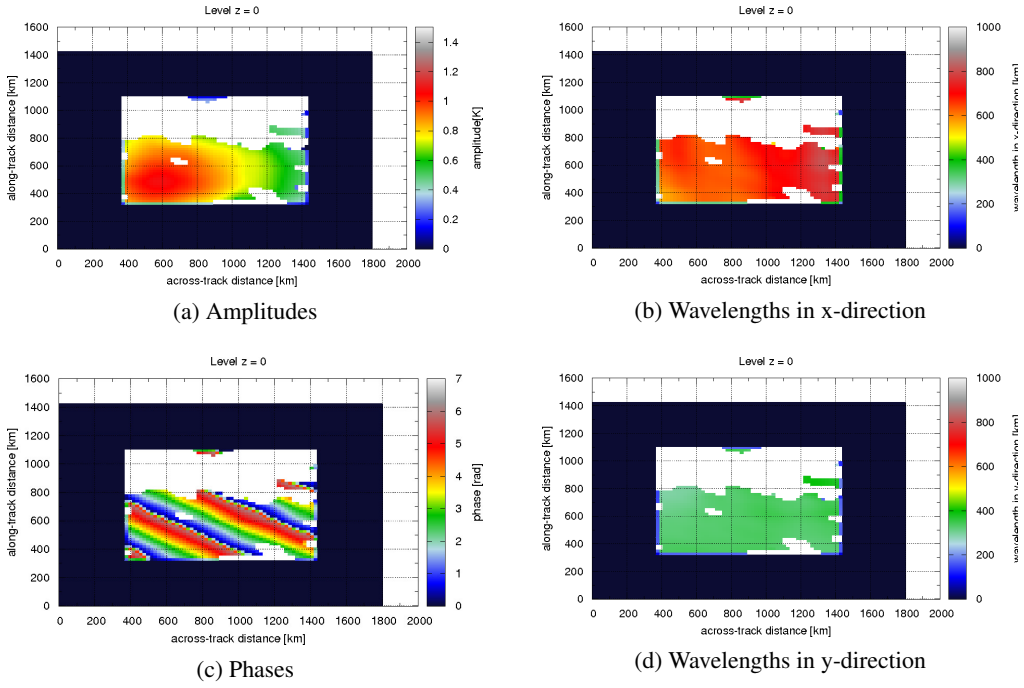
with synthetic data confirm, the optimal box size depends on the wavelength we have to identify. Furthermore, by testing synthetic data with different magnitudes of noise we found out that both of our methods are quite robust towards noise.
The application of FFT and sine fit in only one direction leads to results with a more detailed structure that also contain values of a bad quality. Therefore we prefer the two dimensional analysis, which implies more data.



(a) Wavelengths in x-direction

(b) Wavelengths in y-direction

Figure 9: Applying zero padding improves the results by providing a better resolution.



(a) Amplitudes

(b) Wavelengths in x-direction

(c) Phases

(d) Wavelengths in y-direction

Figure 10: The sine fit method is applied using the results of the FFT as starting parameters.

6 Conclusion and Outlook

We developed a toolbox to identify gravity waves in AIRS data and to analyse their properties. In this report we presented the result of one case study. Further case studies were accomplished and led to similar conclusions. The quality of the identification depends on the choice of different parameters, such as box size, method and dimension. For convective gravity waves the sine fit does not always fullfill the expectations because of their circular form.

The FFT is a reliable method but has a poor resolution, whereas the sine fit yields to accurate results but reveals convergence problems. This disadvantage of the FFT was resolved to some extend by applying zero padding and a special window function. The sine fit was improved by checking different starting parameters.

All in all the best way to identify the gravity waves turned out to be the following: We use the two dimensional approach by selecting a certain box size. Then we apply the FFT using zero padding. Finally we refine the results from the FFT by achieving the sine fit with different sets of starting parameters.

The next step will be to apply the gravity-wave-toolbox on the whole AIRS data set and gain global statistics about the occurrence and properties of gravity waves. Furthermore the toolbox shall be extended for the analysis of three dimensional brightness temperature data.

7 Acknowledgements

I would like to thank my adviser Dr. Lars Hoffmann for his support with knowledge and hints. I am very grateful for the programming skills I learned from him and the interesting tasks he found for me. Furthermore I appreciated a lot the introduction to gravity waves, which was given to me by Dr. Peter Preusse. Besides I would like to thank Prof. Dr.-Ing. Jürgen Kusche for indicating the guest student program to me and for his recommendation. I also would like to thank Dipl.-Ing. Lutz Roese-Koerner for telling me about his experiences with this program. Finally I would like to put into words my gratitude to Mathias Winkel and Ivo Kabadshow for organising the guest student program and thank the other students for their contribution to a nice and instructive time.

References

1. H. H. Aumann, et al. *AIRS/AMSU/HSB on the Aqua Mission: Design, Science Objectives, Data Products, and Processing Systems*. IEEE Trans. Geosci. Remote Senes., 41 (2003), 253:264.
2. C. I. Lehmann, Y. -H. Kim, P. Preusse, H. -Y. Chun, M. Ern, S. Y. Kim. *Consistency between Fourier transform and small-volume few-wave decomposition for spectral and spatial variability of gravity waves above a typhoon*. Atmos. Meas. Tech. Discuss. 5 (2012); 1763:1793; doi:10.5194/amtd-5-1763-2012.
3. L. Hoffmann, M. J. Alexander. *Retrieval of stratospheric temperatures from Atmospheric Infrared Sounder radiance measurements for gravity wave studies*. J. Geophys.Res. 114 (2009); D07105; doi:10.1029/2010JD014401.
4. J Houghton. *The Physics of Atmospheres*. 3rd ed. New York: Cambridge University Press; 2002.
5. D. C. Fritts, M. J. Alexander. *Gravity wave dynamics and effects in the middle atmosphere*. Rev. Geophys. 41(1) (2003); 1003; doi:10.1029/2001RG000106.
6. *AIRS Atmospheric Infrared Sounder*; 2012. Available from: `http://airs.jpl.nasa.gov/`.

Information sharing and collaboration between agents in evacuation simulations

David Haensel

Dresden University of Technology
Faculty of science
Department of Mathematics
01062 Dresden
E-mail: david@davidscorner.de

Abstract:
This work describes a graph based navigation algorithm for pedestrian dynamics simulation in case of evacuation. The main goal of every pedestrian is to leave the building over the shortest path. Additionally to an implementation of the classical shortest path strategy, an information gathering and sharing scheme is modelled. We introduce a reasoning structure for the agents in the simulation. They are able to notice closed or broken escape routes and share it with other pedestrians in the surrounding. We qualitatively analyzed the influence of the radius and the information propagation speed in an office building.

1 Introduction

The investigation of pedestrian dynamics applies to situations with many people crowded in a specific space. These situations occur in subway stations, airports, well known sights and many other places in everydays life. The complexity and the size of those infastructures is frequently increasing and with it the number of people using it. Additionally there are more often large indoor and outdoor events with high pedestrian densities. Unfortunately casualties happened at those places or events through history. For example the fire in a textile factory in Karachi (Pakistan) in 2012 or rumors during coronation of tsar Nicholas II in Moscow (Russia) in 1896. Therefore, the investigation of pedestrian dynamics in those situations is needed to improve buildings, evacuation routes and events.

According to Hoogendoorn et al [1] one can divide pedestrian dynamics simulations in three levels of operation: the strategical level, the tactical level and the operational level (Figure 1). At the strategical level pedestrians decide for a final destination and choose the route to reach that. On the tactical level pedestrians perform short term decisions, e.g. changing decisions because of closed doors or jams or avoiding obstacles. The operational level concerns the modeling of pedestrian motion including acceleration, deceleration and stopping.

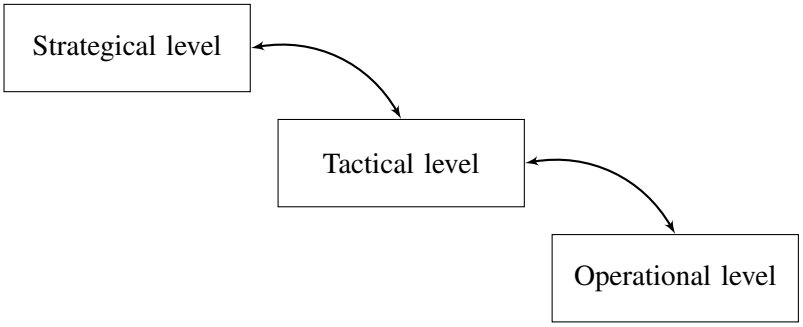


Figure 1: Three levels of operation in pedestrians simulation framework as described in [1]

Most evacuation simulation software neglects individual knowledge of pedestrians about the building and the actual state of escape routes. Unexpected states of escape route e.g. closed doors are either ignored or known from the beginning of the simulation. To obtain a more realistic behavior it is necessary that the pedestrians have individual knowledge about the status of escape routes. Depending on this knowledge the decisions taken by the pedestrians during the simulation will vary.

The ideas developed in this work mainly apply to the tactical and strategical levels. For finding the final destination and the route to it we used a graph generated from the given geometry data. In case of closed doors this graph was changed and the information was shared between pedestrians. The algorithm is implemented in the Jülich pedestrian simulator, a collection of tools for pedestrian dynamics simulations developed at the research centre Jülich in Germany.

For the modeling of pedestrian motion at the operational level the generalized centrifugal force model mentioned in [3] is used. In this model the motion of every pedestrian i is defined by the equation of motion (described in equation 1), where the force $\vec{F}_i$ is the sum of influencing forces:

$$m_i \frac{d^2\vec{R}_i}{dt^2} = \vec{F}_i = \vec{F}_i^{drv} + \sum_{j \in \mathcal{N}_i} \vec{F}_{ij}^{rep} + \sum_{w \in \mathcal{W}_i} \vec{F}_{iw}^{rep} \; . \tag{1}$$

Here, $\vec{F}_i^{drv}$ is the force which drives the pedestrian i towards a certain destination, $\vec{F}_{ij}^{rep}$ is the repulsive force acting from pedestrian j on pedestrian i and $\vec{F}_{iw}^{rep}$ is the repulsive force from walls and obstacles acting on pedestrian i. In the simulation pedestrians are modeled as ellipses. The size of the ellipses in moving direction is velocity dependent and increases with the velocity [3].

2 Graph based navigation

For the navigation in the building a graph generated from the given geometry data is used. With this graph pedestrians are able to navigate through the building and find the emergency exit with the shortest distance.

2.1 Graph structure

The simulated geometry is divided in navigation areas. The boundaries of the navigations areas are the navigation lines (see Figure 2). There are three kinds of navigation lines:

Crossings are virtual doors between two navigation areas in the same room.

Transitions are doors between two navigation areas in different rooms.

Hlines are help lines in navigation areas to facilitate pedestrians movement around obstacles.

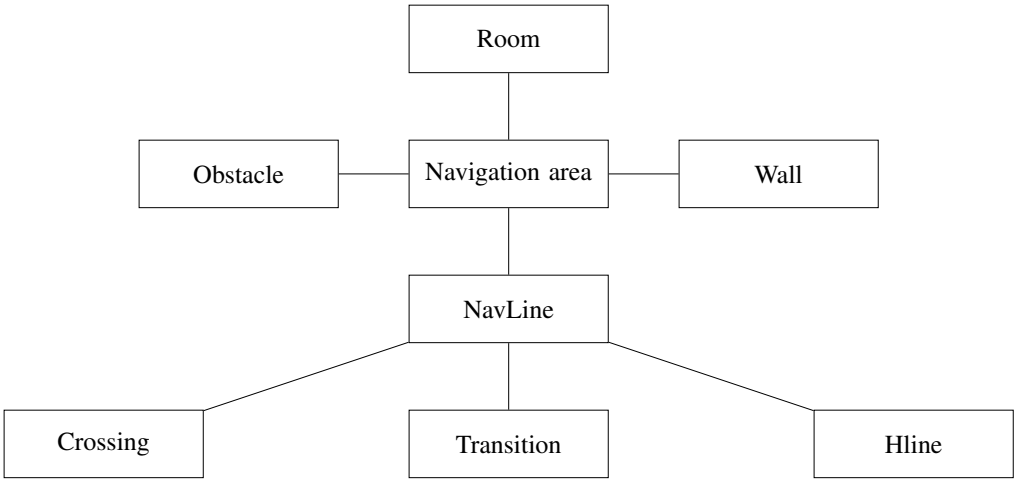


Figure 2: Data structure of the geometry.

From this geometry data we derived a graph structure. The graph uses a navigation line as vertex and connects two vertexes with an edge when they are visible to each other. An edge is always related to a certain navigation area connecting the two navigation lines (vertexes). Moreover it expresses the absolute distance between the two navigation lines (Figure 3).

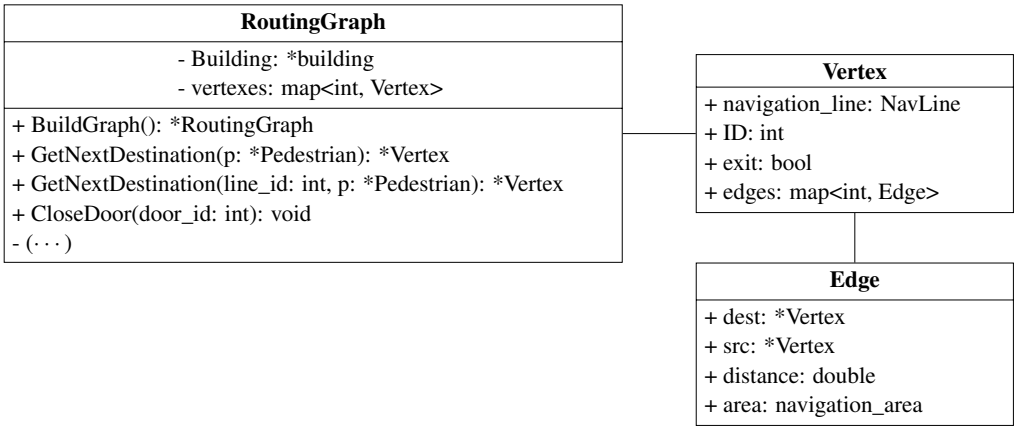


Figure 3: Routing graph class-diagram

In order to create the graph we iterate through all rooms and the included navigation areas. For every navigation area we consider all navigation lines in this area. After adding the new lines as vertexes it

is checked if they are visible to each other. Therefore, three points in each line are taken and checked if one connection of all possible connections (between the three points each) does not intersect with an obstacle, wall or hline. If this applies it means, that there is at least one visible connection between the two lines.

After the creation of the graph it is possible to navigate pedestrian through the building. To find the emergency route we used the absolute distance between vertexes. For navigation with graphs and other heuristics more information is provided in [5] or [6]

2.2 Shortest path computation as sub-graph

To navigate pedestrians to the next emergency exit one could use the absolute distances. Using that approach lead to the global shortest path (compare [2]). This is achieved by calculating the absolute edge distance and from this the route distance to a certain emergency exit. We used a slightly modified Dijkstra algorithm [4]. The modification is a constraint on the set of next possible edges. The assumption limiting the set of next possible edges is, that the next chosen edge can not be in the same navigation area than the last edge coming from to the vertex, except if the vertex represents a hline. This assumption ensures, that the pedestrian chooses the direct way through a navigation area. The pedestrian is just allowed to go through doors and not tangent to them. Hlines are an exception because they are in a navigation area and all incident edges are in the same navigation area. After calculating the distances, every emergency exit is the root of a tree with the shortest paths from vertexes with a path to the emergency exit. This approach makes a navigation possible.

2.3 Navigating pedestrians with global knowledge

With the described graph, pedestrians are able to navigate through a building assuming that they have global knowledge about closed or barred doors and are always taking the shortest way to the emergency exit. If a door is closed it is simply ignored during the creation of the graph.

For the navigation each pedestrian gets an initial destination. At this stage all visible navigation lines are taken in consideration. To choose the shortest path, the distance between the pedestrian and the line is added to the distance between the line and the emergency exit. If the pedestrian reaches a vertex (navigation line) the next destination is given from the tree with shortest paths. Then the navigation is just from vertex to vertex (Figure 4).

3 Information collection and propagation

As aforementioned the knowledge of closed doors is known for all pedestrians before. This is not quite realistic. The pedestrians should discover closed doors during the simulation. The goal is, that pedestrians arrive at a closed door, take notice that the door is closed, search a new route and share this information with other pedestrians.

To keep track knowledge about closed doors we implemented a collection of door state objects. If a pedestrian reaches a closed door he/she will remember the status of the door and the time he/she saw

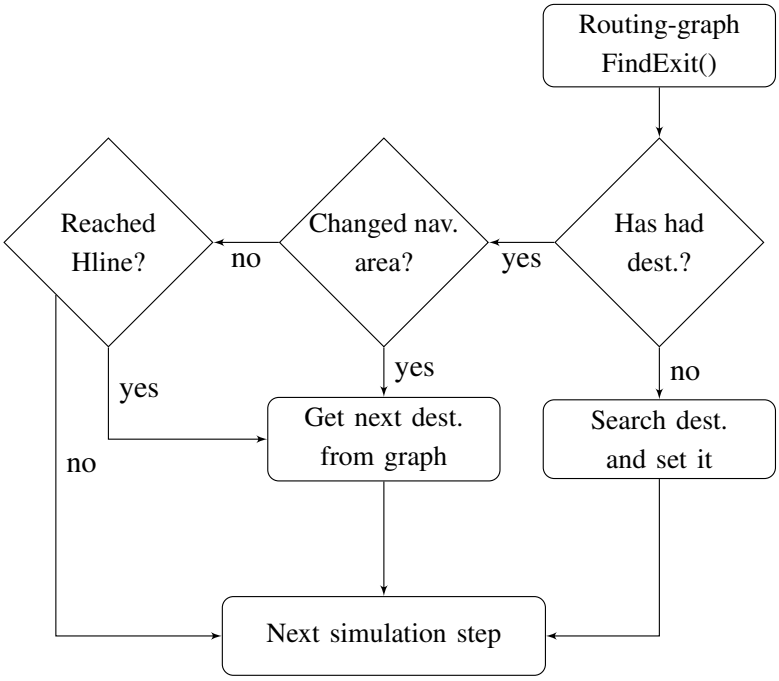


Figure 4: The graph based navigation algorithm.

it. From now on he/she has the information he/she needs to avoid this door. After some changes in the routing graph the pedestrian has to search for a new route. This route search is similar to the initial route choice.

3.1 Graph changes and graph storage

Due to the fact that all information used for the navigation is taken from the graph and not anymore from the geometry data, changes in the graph are needed. To model a closed door, the corresponding vertex and all adjacent edges are deleted. After this the trees with the shortest ways have to be recalculated. After this every pedestrian would need his own graph to make these manipulations without influencing the other pedestrians. This would not be really memory efficient. This is the reason why a graph storage is used (Figure 5).

GraphStorage
- graphs: collection
+ GetGraph(IDs:set): *graph - CreateGraph(IDs:set, FromIDs:set):void - Initialize():void

Figure 5: GraphStorage class-diagram

The graph storage manages all used graphs. It takes a set of IDs representing the closed doors (representing vertexes) and delivers the corresponding graph for a pedestrian (See Figure 5 GetGraph()). If the graph does not exist yet, the storage creates the graph from an existing graph by deleting the

given vertexes (CreateGraph()). With this approach it is possible to take different knowledge about the building in consideration. For this purpose we implemented the knowledge about closed doors. In general every knowledge about the building which could be applied to the routing graph could be used here.

At this point every pedestrian has to discover closed doors by its own. To solve this problem we implemented an information sharing algorithm.

3.2 Information propagation

To share information we assumed that every pedestrian shares information with all other pedestrians in the neighborhood. While this assumption might be realistic, it lacks empirical evidence at the moment.

3.2.1 Information sharing space

In real life situations humans have many possibilities for sharing information. One method is sending visual signals (a hand wave for instance). The receptor of the signal should be in a specific sight range. Using audio signal is one other option. This option is also constrained in space and will be influenced by doors and walls.

In all cases the information sharing is restricted in space and time. In our implementation we defined a sharing space:

Definition 1 (Information sharing space r_{share})**.**
The information sharing space is the space with radius r_{share} in a navigation area around a pedestrian in which this pedestrian informs all other pedestrians.

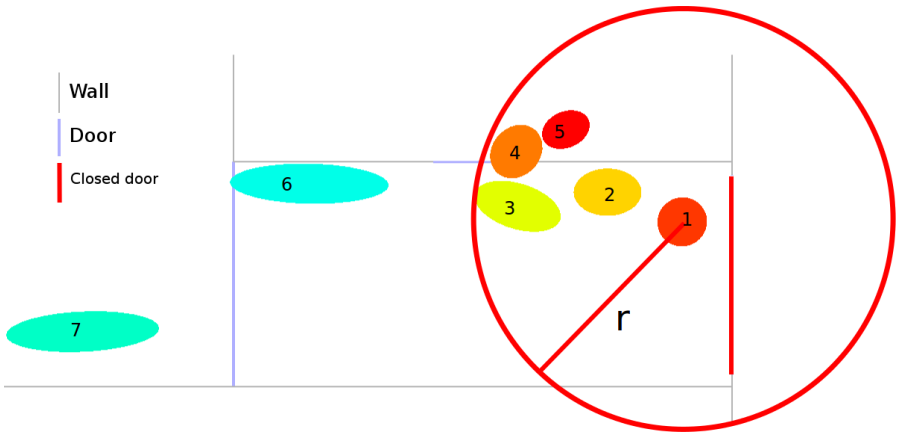


Figure 6: Information sharing space. Pedestrian 1 informs pedestrian 2 and 3. Pedestrian 4 and 5 do not get the information because they are in another navigation area. Pedestrian 6 and 7 are to far for getting the information.

With this limitation the information is shared with pedestrians in a certain space in the same navigation area.

3.2.2 Information sharing delay time

A pedestrian would usually need some time to receive the information before he could share it again. In the simulation this fact introduces a natural delay time:

Definition 2 (Information sharing delay time Δt_{share})**.**
The information sharing delay time is the time Δt_{share} a pedestrian has to wait before sharing new information.

With the delay time and the sharing radius it is possible to calibrate the information propagation especially the propagation velocity for realisticity.

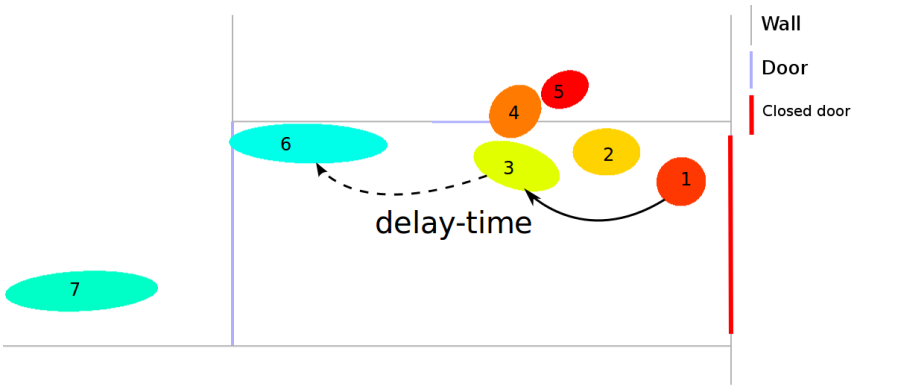


Figure 7: Information sharing delay time. Pedestrian 1 shares information with pedestrian 3. Pedestrian 3 has to wait until he could share the information with 6.

4 Simulation and results

All simulations (Figure 8, 9 & 10) are done with the Jülich pedestrian simulator, with the geometry shown below.

In this images the color of pedestrians is velocity dependent. Fast pedestrians are green, slow pedestrians are red. In both simulations the emergency exit on the right is closed and the emergency exit on the left is open.

In the first simulation (Figure 8) every pedestrian had the same knowledge. Everyone knows from the beginning that the right emergency exit is closed, so everyone is going directly to the exit on the left side. Jams just appear on bottlenecks or doors.

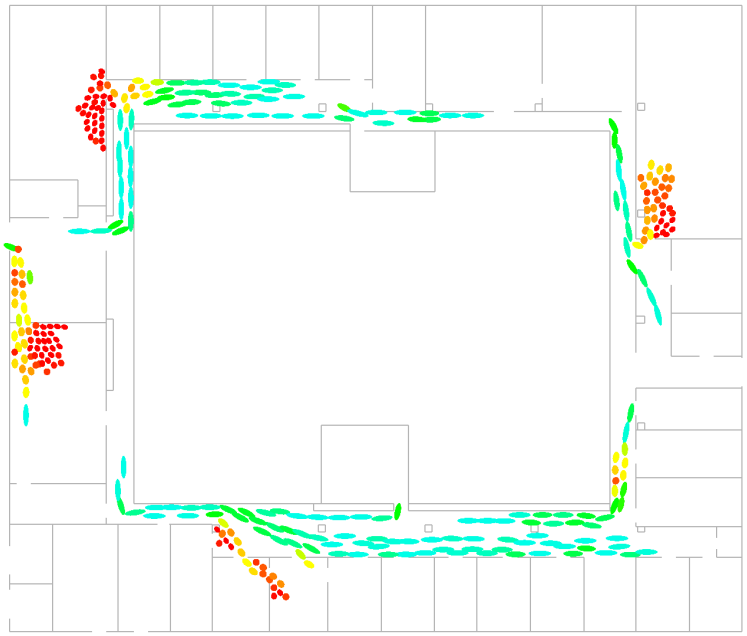


Figure 8: The first simulation. Everybody knows from the beginning that the right emergency exit is closed.

In the second simulation (Figure 9 & 10) the right emergency exit is closed again but no one knows it at the beginning and the parameters are $\Delta t_{share} = 1.5s$ and $r_{share} = 2m$. In Figure 9 the first pedestrian reaches the closed emergency exit at the right side (the red pedestrian in front of the door). The pedestrians in the right half of the building still have the goal to go to the right emergency exit. In the second image (Figure 10) some pedestrians already changed the direction and shared information about the closed emergency exit. In the lower right corner of the floor a jam arises because two groups of pedestrians are moving in opposite directions. A few seconds later all pedestrians are informed about the closed door and everyone is going to the left emergency exit.

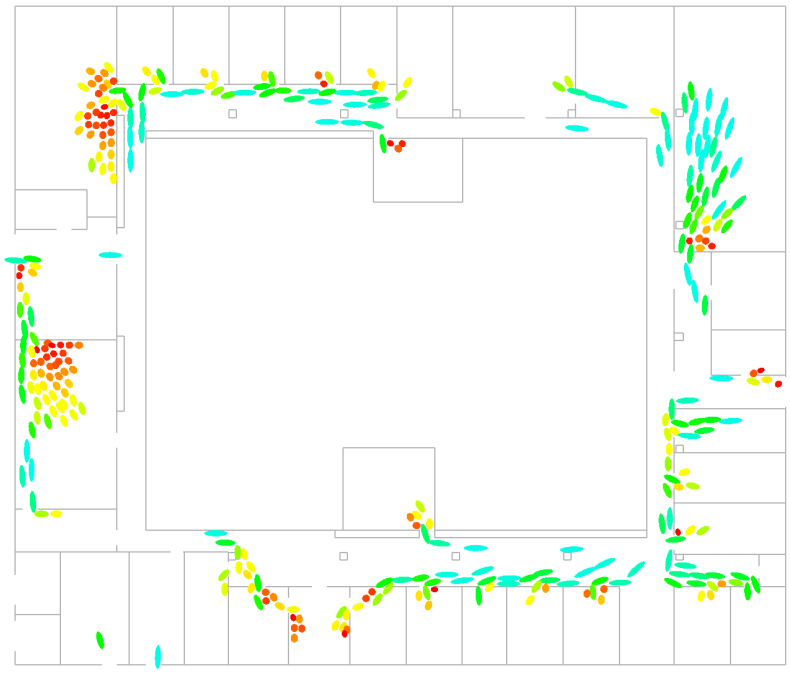


Figure 9: The second simulation. The red pedestrian in front of the right emergency exit just reached the closed door.

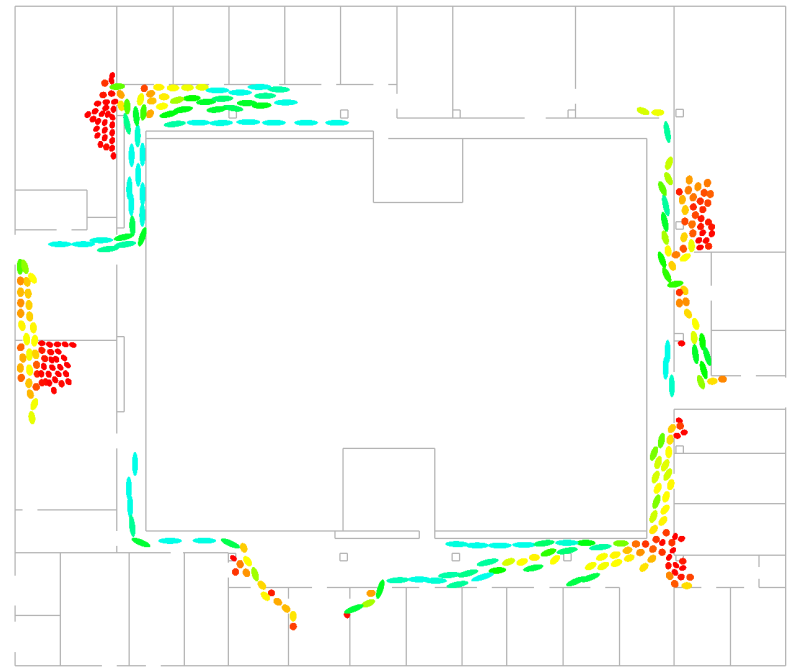


Figure 10: The second simulation. Some pedestrians already changed direction. Jam arises in the lower right corner of the floor.

5 Conclusion

In this work we implemented a graph based navigation algorithm. Further we implemented a first reasoning structure which enables the pedestrians to avoid broken escapes routes, for instance closed doors. With the two information sharing parameters, the space and the delay a quite realistic information sharing is obtained. The graph storage could be used to have different routing graphs depending on individual knowledge about the building. There are further things one should do to use the approach described. The parameters information sharing space (Definition 1) and information sharing delay time (Definition 2) have to be calibrated for more realisticity. The results of the simulations have to be quantified and proofed. For more realisticity one could apply a probability to the door state information and reduce it during the propagation to model uncertainty of the information. Additionally there are further things which could be interesting to investigate. One could apply weights depending on the capacity of navigation areas to the graph edges to make a flow optimization. Automatic re-routing in case of jams and automatic creation of navigation lines could be interesting.

References

1. Hoogendoorn SP, Bovy P, Daamen W. Microscopic Pedestrian Wayfinding and Dynamics Modelling. In: Schreckenberg M, Sharma S, editors. Pedestrian and Evacuation Dynamics. Springer; 2002, 123–155.
2. Kemloh U, Seyfried A, Holl S. Modeling The Dynamic Route Choice Of Pedestrians To Assess The Criticality Of Building Evacuation. Advances in Complex Systems [Internet]. 2012 [cited 2012 Oct 2];15(3):1–22. Available from: http://www.worldscientific.com/doi/abs/10.1142/S0219525912500294
3. Chraibi M, Kemloh U, Schadschneider A, Seyfried A. Force-based models of pedestrian dynamics. Networks and Heterogeneous Media. 2011; 6:425-442
4. Dijkstra EW. A note on two problems in connexion with graphs. Numerische Mathematik. 1959; 1:269-271
5. Berkhahn V, Kneidl A, Klein W. Graph-based approaches for simulating pedestrian dynamics in building models. In: Scherer R, Menzel K, editors. Proceedings of European Conference on Product and Process Modelling; 14-16 sep 2010: Cork, Republic of Ireland: CRC Press; 2010. p. 389-394
6. Kneidl A, Borrmann A, Hartmann D. Generation and use of sparse navigation graphs for microscopic pedestrian simulation models. Advanced Engineering Informatics [Internet]. 2012 Apr [cited 2012 Oct 5];1–9. Available from:http://linkinghub.elsevier.com/retrieve/pii/S1474034612000365


JÜLICH
FORSCHUNGSZENTRUM