CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Parallel Simulation of Biomacromolecules using the DFTB method
Student:	Bc. Jakub Pekař
Supervisor:	Ing. Jiří Černý, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2016/17

Instructions

Examine the Density Functional-based Tight Binding (DFTB) method, a fast and efficient quantum mechanical (QM) method for biomolecular simulations. Study the existing source codes for this method and data formats for representation of molecular data (http://cbp.cfn.kit.edu/joomla/index.php/downloads/18-gromacs-with-qm-mm-using-dftb3). Perform profiling of these codes and identify key routines for parallelization. Design and implement parallel version of the code for GP GPU platform using CUDA libraries. Test correctness of simulations and performance of the resulting programs on biomolecules supplied by thesis supervisor (simulation of solvated protein/DNA with increasing size of the QM part).

References

Will be provided by the supervisor.

L.S.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 2, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Computer Systems

Master's thesis

Parallel Simulation of Biomacromolecules using the DFTB method

Bc. Jakub Pekař

Supervisor: Ing. Jiří Černý, Ph.D.

8th January 2017

Declaration

In Prague on 8th January 2017 ....

Citation of this thesis

Pekař, Jakub. *Parallel Simulation of Biomacromolecules using the DFTB method.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Cílem této práce je akcelerace DFTB metody používané k simulaci biomakromolekul. Časově nejnáročnější části simulace jsou identifikovány a akcelerovány s použitím interních možností původního řešení, paralelizace pomocí OpenMP a možností použít knihovnu MAGMA, která využívá hybridních CPU/GPU algoritmů ke zrychlení operací lineární algebry. Jednotlivé metody akcelerace jsou popsány a otestovány na systémech vod různých velikostí. Výsledky testů, prokazující zrychlení oproti jiným knihovnám implementujícím operace lineární algebry, minimální dopad paralelizace kódu a problémy s použitím interní akcelerace, jsou prezentovány v závěru práce.

Klíčová slova DFTB, paralelní simulace, GROMACS, MAGMA, GPU akcelerace

Abstract

The purpose of this work is to accelerate DFTB method used for biomacromolecule simulation by paralellization. The most time consuming parts of simulation are identified and parallelized using internal options and CPU and GPU acceleration of used linear algebra routines and the code itself. Possibility to use MAGMA – hybrid GPU/CPU linear algebra library and OpenMP parallelization of the code were implemented. All of those methods are described and tested on water systems of different size. Test results of MAGMA acceleration, parallelization and problems with internal optimizations are presented at the end of the work.

Keywords DFTB, parallel simulation, GROMACS, MAGMA, GPU acceleration

Contents

List of Figures

Introduction

Density Functional-based Tight Binding (DFTB) is fast and efficient quantum mechanical method for biomolecular simulations. It is used to simulate biochemical molecules like proteins, lipids, nucleic acids, etc.

Its implementation in GROMACS package allows to use its parts to process various inputs and outputs, provides internal routines, schemes and structures to be reused in the DFTB code and many more features that are not included in standalone implementations of the method. Disadvantage of this implementation is that it is not fully implemented and cannot be used for all of possible configurations of the simulation.

Simulation of biomolecular systems can be really time consuming process. For larger systems with more elements and bonds between those elements it can consume hours or even days to simulate few femtoseconds. The only available option to accelerate the simulation is to use different library for linear algebra operations. Whole simulation is running in a single thread without any further optimizations. GROMACS has support for internal parallelization and even GPU acceleration on CUDA capable graphic cards. To use those features, Verlet non-bonding cut-off scheme must be used to store the data, but only older Group scheme is supported by the DFTB implementation.

By parallelizing the DFTB code, performance of the simulation may be raised and capabilities of processing machine may be used more effectively. Using GROMACS's internal acceleration using Verlet non-bonding cut-off scheme might be a possibility to gain additional performance boost. GPU acceleration might be implementd in order to raise performance even more.

Chapter 1

Theoretical background for biomolecular simulations

1.1 Molecular Mechanics (MM)

The Molecular Mechanics (MM) describes the energy of a (bio)molecule in terms of classical (non-quantum) physics. For the purpose of MM the studied system is defined by the topology of a molecule and corresponding forcefield parameters. The topology contains the information on the chemical composition, atomic types, and atomic partial charges. The atomic types for a selected element cover different hybridization states of the atom within a molecule and/or chemical environment defined by the molecular surroundings. The partial charges are then assigned based on experimental or theoretical data. Although the details can be found in classical books[1] a short summary will be presented also here.

The energy function can be defined as a sum of valence term, cross-term, and the non-bonding term

$$E_{total} = E_{valence} + E_{cross-term} + E_{non-bonding}, \tag{1.1}$$

where

$$E_{valence} = E_{stretch} + E_{angle} + E_{torsion} + E_{out-of-plane}, \tag{1.2}$$

describes (intra)molecular vibrations defined by a set of spring constants and corresponding potential depths,

$$E_{cross-term} = E_{str-str} + E_{str-angle} + E_{angle-angle-tors} + E_{tors-angle} + \dots, \tag{1.3}$$

accounts for cross-terms correcting for the coupling between the internal coordinates, and

$$E_{non-bonding} = E_{vdW} + E_{elec} + E_{H-bond}, \tag{1.4}$$

includes the van der Waals interaction defined by the repulsion and attractive dispersion usually in the form of Lennard-Jones (12-6) potential [2], the electrostatics interactions of point charges and multipoles from Coulomb's law, and the hydrogen-bonding term including the H-bonding directionality.

For the efficient calculation of electrostatic interactions three main concepts proved successful. It was the introduction of cut-off distances, the Periodic Boundary Conditions (PBC), and the Particle Mesh Ewald (PME)[3] method for the calculation of non-covalent interactions. Within the PBC in three dimensions the system is modeled by placing the molecule within a large enough box[4]. The system is treated as periodic with each box effectively surrounded by its 26 identical nearest neighbor boxes. In the PME, the potential describing the interaction is defined as two terms, dividing the problem into a short range and long range contributions. The basic idea is to substitute the summation of interaction energies between point charges at particles

$$E_{\mathrm{TOT}} = \sum_{i,j} \varphi(\mathbf{r}_j - \mathbf{r}_i) = E_{sr} + E_{\ell r} \tag{1.5}$$

with two summations, a sum of potential at the short range in real space (the "particle" part of PME)

$$E_{sr} = \sum_{i,j} \varphi_{sr}(\mathbf{r}_j - \mathbf{r}_i) \tag{1.6}$$

and a summation in reciprocal Fourier space for the the long range interactions across the infinite periodic images of the system

$$E_{\ell r} = \sum_{\mathbf{k}} \tilde{\Phi}_{\ell r}(\mathbf{k}) \, |\tilde{\rho}(\mathbf{k})|^2 \tag{1.7}$$

The sharp cut-off for the short range interactions could cause energy artifacts when for example only a part of the molecule lies within the cut-off distance. To remove these artifacts the non-bonded neighbor list proposed by Verlet or the Group-based cut-offs are widely used.

1.2 Molecular Dynamics (MD)

The Molecular Dynamics (MD) allows to study a time evolution of molecular system. The classical MD employs the analytical gradient of the energy defined above. The trajectory (time evolution of the system) is obtained by solving the differential equations based on the Newton's second law ($F = ma$), allowing to relate the force (as a negative gradient of the energy) with the mass and acceleration of the particle/atom in a molecule. There are many algorithms like leap-frog[5] or velocity Verlet[6] for efficient integration of the equations of motion using finite difference methods.

For example the velocity Verlet method provides positions, velocities and accelerations at the same time while retaining the precision:

$$r(t + \delta t) = r(t) + \delta t v(t) + \frac{1}{2}\delta t^2 a(t) \tag{1.8}$$

$$v(t + \delta t) = v(t) + \frac{1}{2}\delta t[a(t) + a(t + \delta t)] \tag{1.9}$$

To start the simulation, it is necessary to assign initial velocities to all studied particles. This is usually done by random selection from a Maxwell-Boltzmann distribution at the temperature of interest.

$$p(v_{ix}) = (\frac{m_i}{2\pi k_B T})^{\frac{1}{2}} e^{[-\frac{1}{2}\frac{m_i v_{ix}^2}{k_B T}]} \tag{1.10}$$

The length of the time step δt depends on the simulated molecular system as well as integration algorithm and other simulation details. Usually for a biomolecular simulation the time step of 2 fs is used. This clearly shows that to simulate a biologically relevant conformational change of biomolecule (corresponding to for example μseconds) the number of integration steps grows rapidly showing the necessity for efficient code.

1.3 Self-Consistent-Charge Density Functional-based Tight-Binding with empirical dispersion correction (SCC-DFTB-D)

The full name of the method implemented in the GROMACS package is Self-Consistent-Charge Density Functional-based Tight-Binding with empirical dispersion correction (SCC-DFTB-D), the shorter DFTB abbreviation will be used throughout the text. As the abbreviation suggests, the full method composes of three main theoretical blocks – the Tight Binding or Density Functional-based Tight-Binding (TB[7] or DFTB[8]), the Self-Consistent-Charge (SCC[9]) and the Dispersion term (D[10]). This deserves a short description of the theory behind and development of each of the blocks.

1.3.1 Tight-Binding Theory (TB)

The "classical" TB approach[7] (developed and used for simulations in solid state material physics) is based on the assumption that the total electronic energy can be written in the form

$$E^{TB} = \sum_{i}^{occ} \varepsilon_i + E_{rep}, \tag{1.11}$$

where ε_i are the eigenvalues of non-selfconsistent equation

$$H\Psi_i(r) = \varepsilon_i\Psi_i(r) \tag{1.12}$$

and E_{rep} is a short-range pairwise repulsion term. The solution leads to

$$|H - \varepsilon S| = 0, \tag{1.13}$$

where

$$H =< \phi_\alpha|H|\phi_\beta > \text{ and } S =< \phi_\alpha|\phi_\beta > \tag{1.14}$$

define the Hamiltonian and Overlap matrices.

Contrary to other non-empirical/ab initio methods and similarly to semi-empirical quantum mechanical (QM) methods, the most common approach in TB is to treat the matrix elements, assumed to extend only to the first or second neighbors (given by the chemical topology of the molecule), as parameters and fit them to experiment or other calculations.

Example of data distribution inside Hamiltonian matrices of various systems is visible in figure 1.1. Non-zero values inside the matrix are marked with black color.

1.3.2 Slater and Koster parametrization for TB

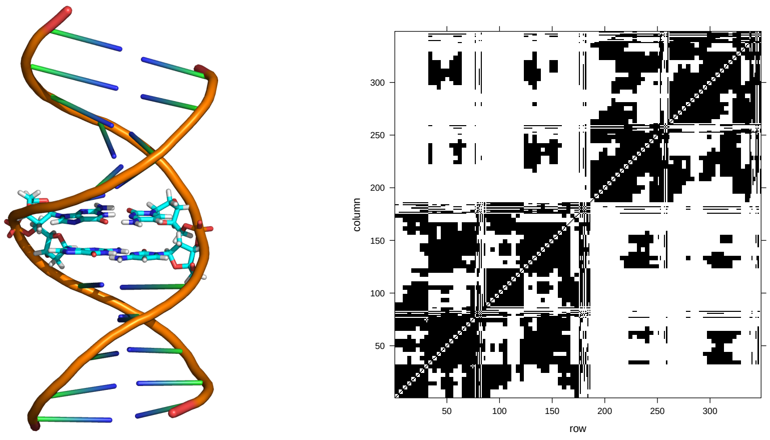
The DFTB[8] employs the so called Slater and Koster parametrization which is a two-center parametrization with simple distance and angle dependent functions. For example in case of homonuclear C - C interaction, with each carbon atom having one s orbital and three p orbitals in its valence shell, there are simple functions describing all orbital/bonding type interactions, i.e. ssσ, spσ, ppσ, and ppπ. Density Functional Tight-Binding Method (DFTB) Single-particle wave functions Ψ_i within the Linear Combination of Atomic Orbitals (LCAO) ansatz are expanded into Slater-type atomic orbitals

$$\Psi_i(\vec{r}) = \sum_v c_{vi}\phi_v(\vec{r} - \vec{R_\alpha}) \tag{1.15}$$

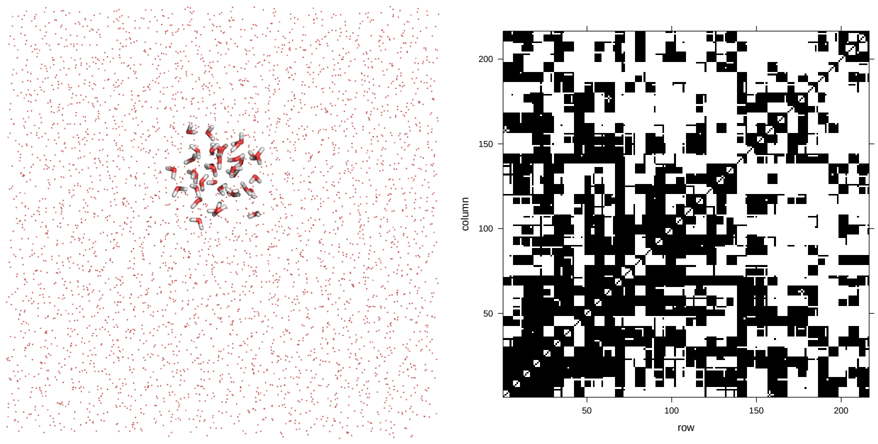
Basis set is constructed by self-consistent ab initio Density Functional Theory (DFT) calculation for single atom (here comes the DF from) and the obtained non-selfconsistent Kohn-Sham (KS) equations transform to a set of algebraic equations

$$\sum_v^M c_{vi}(H^0_{\mu v} - \varepsilon_i S_{\mu v}) = 0 \tag{1.16}$$

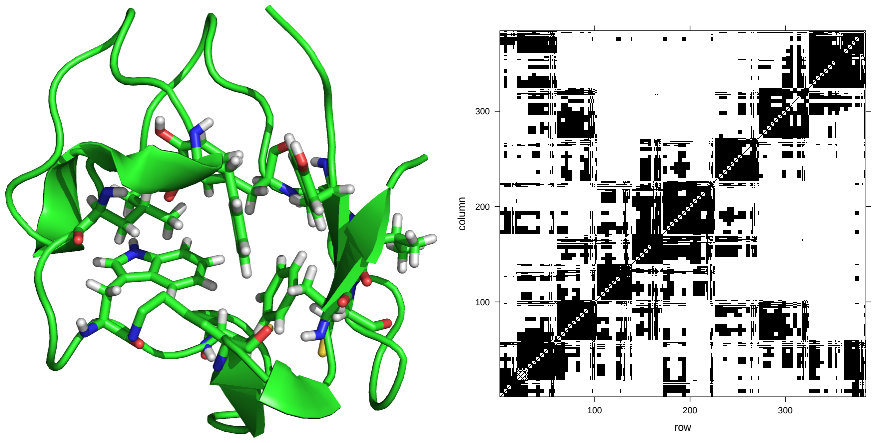
The DFTB method defined in such a way works well for different systems and materials, covering many of C, Si, Ge structures, for which the many-atom structure may be represented as a sum of atomic-like densities.



(a) A pair of dinucleotides in DNA dodecamer



(b) A cluster of 36 (QM) water in a simulation water box (MM)



(c) Hydrophobic core of the Rubredoxin protein

Figure 1.1: Non-zero values in Hamiltonian matrices

1.3.3 Self-Consistent-Charge Tight-Binding Method (SCC-DFTB)

The heteronuclear, charged and polar systems are not properly described at the non-SCC DFTB level. In other words, belonging into all of the mentioned categories, the biomolecules were not described reliably. In an attempt to improve the method for biomolecules, this led to introduction of the second order energy term by Elstner[9]

$$E^{TB} = \sum_{i}^{occ} < \Psi_i | H_0 | \Psi_i > + \frac{1}{2} \sum_{\alpha,\beta}^{N} \gamma_{\alpha\beta} \Delta q_\alpha \Delta q_\beta + E_{rep} \tag{1.17}$$

This equation has to be solved iteratively to obtain converged, self-consistent Mulliken charges. The resulting energies are then nicely improved, but it also introduces a few drawbacks.

Mainly, this SCC part of calculation represents a significantly time consuming routine, so the performance is lower, compared to non-SCC version (depending on the number of SCC cycles needed for obtaining the converged charges/energy). Further, especially for extended and highly charged biomolecular systems there are common problems with convergence of Mulliken charges. This problem can be at least partially solved by employing the non-zero electronic temperature concept to improve the convergence rate. The last weakness of DFTB was originally only a limited number of Slater Koster parameters, especially for biomolecules. However, nowadays there are parameters available for all common main row elements present in biomolecules, as well as for various metals which often play an important functional role.

1.3.4 Empirical Dispersion Term for SCC-DFTB-D

It was shown by various authors that dispersion interaction plays a critical role in the biomolecular structure and function[11]. The dispersion interaction is, however, not included in both the DFTB and the DFT used for its parametrization. To overcome this there are many slightly different approaches based on the known r^{-6} distance dependence of the dispersion contribution. This simple empirical term itself would lead to overestimation of dispersion at short range, thus a damping function has to be used.[10]

$$E_{disp} = - \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{C_6^{ij}}{R_{ij}^6} f_{dmp}(R_{ij}) \tag{1.18}$$

with the f_{dmp} damping function defined as

$$f_{dmp}(R) = \frac{1}{1 + e^{-\alpha(\frac{R}{R_0} - 1)}} \tag{1.19}$$

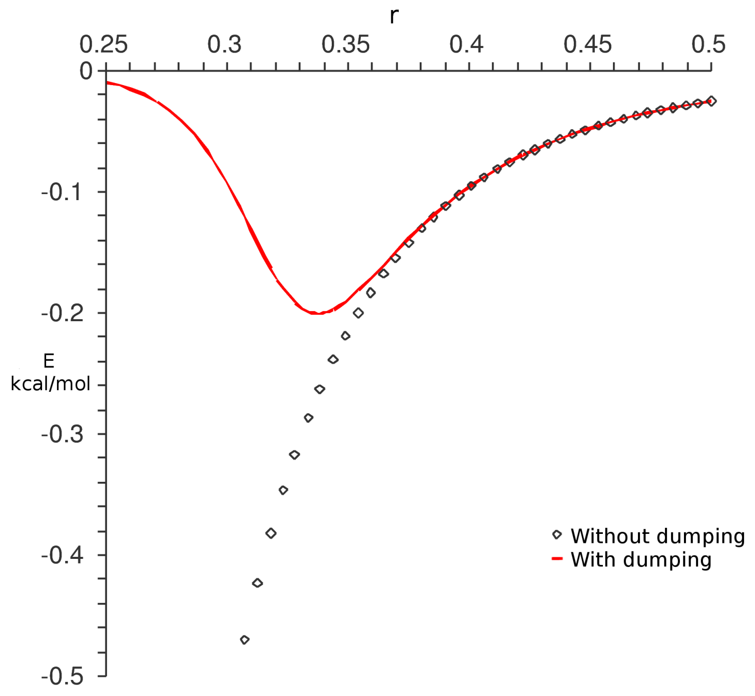


Figure 1.2: Dispersion of C-C

1.4 Combination of QM with MM (QM/MM)

The classical MM and MD allow to study only molecules close to their equilibrium state with the immutable chemical topology and atomic types. In other words it does not allow to rearrange chemical bonding pattern which would be necessary for description of chemical reactions. The classical concept would for example lead to a steep increase of energy while elongating a chemical bond.

The DFTB offers a relatively low cost QM calculations of biomolecules. Due to the inherent availability of the wave function of the system, it can be used not only to study various chemical transformations and their mechanisms, but also to study for example photochemical properties (electronic excited states) or NMR properties. On the other hand, with larger biomolecular systems, especially including solvent molecules, the DFTB would become impractical. A common solution is to use the higher cost QM method only for an important part of the biomolecule and describe the rest, including solvent molecules at the MM level. One of the QM/MM implementations is available also for the GROMACS package, including the DFTB/MM option.[12]

Chapter 2

Original solution

2.1 GROMACS

2.1.1 About

GROMACS[13] is versatile package to perform molecular dynamics on biochemical molecules like proteins, nucleic acids and lipids. GROMACS provides extremely high performance compared to other programs. In current version it supports SIMD instructions and it also has CUDA-based GPU acceleration for nvidia graphic cards with compute capability at least 2.0. It can also run in parallel using standard MPI communication protocol. GROMACS source code and binaries are freely available at http://www.gromacs.org.

It offers multiple methods used in simulations and various tools for input and output file conversion. It also allows to define precision, enable or disable its parts like MPI or CUDA acceleration, choose custom linear algebra library and more.

2.1.2 Dependencies

2.1.2.1 Compiler and libraries

In order to build GROMACS binary C and C++ compiler must be available in the system. It is recommended to use latest versions of those – Intel compilers version 12 or later or GNU compilers version 4.7 or later. On Mac OS X it is recommended to use Intel compilers because of the performance.

CMAKE 2.8.8 or later is required to properly configure and build GROMACS.

If GPU acceleration is to be used CUDA version 4.0 software development kit is required. It is strongly recommended to use latest CUDA version and drivers. CUDA compiler may not be compatible with the latest versions of C compilers. This situation can be solved by using different C compilers for

CUDA and for the rest of the project by setting `CUDA_HOST_COMPILER` variable with path to supported compiler version while configuring with cmake.

2.1.2.2 FFT

GROMACS needs a library to perform Fast Fourier Transform (FFT). It is possible to use any supported library available in your system or to let GROMACS download and compile it's own library. To download GROMAC's FFTW library it is needed to set `GMX_BUILD_OWN_FFTW` to `ON` while configuring with cmake. Supported libraries are FFTW, MKL and FFTPACK.

If any of supported library is present on the system it can be used by setting `GMX_FFT_LIBRARY` to `mkl`, `fftw` or `fftpack` when you run cmake. In the case that MKL is used MKL's BLAS and linear algebra libraries will be used as well. In this case `MKL_LIBRARIES` containing all MKL libraries to use and `MKL_INCLUDE_DIR` must be set.

2.1.2.3 BLAS & LAPACK

To perform basic and advanced linear algebra routines BLAS and LAPACK libraries are used. It is possible to use any available library or let GROMACS use its internal libraries which are not optimized to maximize the performance. Cmake will look for any BLAS library using standard paths by default. If none is found internal library will be used.

To specify custom library, specific variables must be provided for cmake. `GMX_EXTERNAL_BLAS` or `GMX_EXTERNAL_LAPACK` respectively has to be set to `ON` and path to search must be specified in `CMAKE_PREFIX_PATH` or path to library must be specified in `GMX_BLAS_USER` or `GMX_LAPACK_USER` respectively.

If MKL is used for FFT custom libraries may be specified to use instead of MKL libraries.

2.2 DFTB implementation for GROMACS

DFTB implementation[12] is provided as patch file for GROMACS source and adds dependency on Plumed library. Path to Plumed library must be specified in `Plumed.inc` and `Plumed.cmake` files within root folder of GROMACS sources.

Implementation contains few new files with whole code of DFTB method and changes of other files to integrate this new code in the GROMACS project.

DFTB implementation supports only older Group non-bonding cut-off scheme for storing data. This scheme does not support any kind of internal CPU or GPU acceleration. The code itself is also unparallelized.

2.3 Simulation

2.3.1 Input

Input of the simulation is pack of files describing the topology of the system. Here the position, type, mass, charge and other attributes of each atom is described as well as the bonds between atoms. There is also file containing configuration of simulation used by GROMACS. Those files contains configuration variables like used data scheme, path for directories with additional data, additional information about simulation, etc.

Input processing is initiated by running command

```
gmx grompp -f disp.mdp -c conf.gro -p topol.top
```

where `disp.mdp` file contains configuration for QM/MM, `conf.gro` contains GROMACS specific configuration and `topol.top` contains topology information. Output of this process is `topol.tpr` file.

2.3.2 Run

To run the simulation the topology file must be provided and output file must be specified. Simulation is started by command

```
gmx mdrun -s topol.tpr -o disp.trr
```

where `disp.trr` is output file that will contain trajectory after simulation. If GROMACS was compiled with double precision support, use `gmx_d` binary instead. While the single precision is sufficient for the typical classical MD simulations, the double precision is necessary for QM/MM, including the DFTB.

To speedup the simulation by internal parallelization it is possible to set number of used threads or ranks by using additional parameters followed by the number.

-nt specifies total number of threads to start

-npme specifies number of used MPI ranks for PME calculation

-ntmpi to set the number of MPI threads

-ntomp sets number of OpenMP threads in one MPI rank

-ntomp_pme specifies number of OpenMP threads in one PME rank

These parameters must be considered for given configuration like that Group cut-off scheme does not support parallelization, etc.

2.3.3 Output

Output of the simulation is a trajectory file after specified amount of time / steps, containing the time evolution of the simulated system as coordinates saved after every given number of steps. This output is written to the output file specified in run command.

Additionally to the standard output various information is written. The information about configuration is written at the beginning of simulation, times of individual parts and subresults during the simulation and final performance is written to the output at the end. Some pieces of information can be used for determining the state of simulation (step, part, etc.), some for performance testing (times of individual parts) and some for debug purposes.

All errors are written to the standard error output.

Chapter 3

Analysis of original solution

3.1 Code

As was told before, the original solution does not support any kind of parallelization. Only option to improve performance was to use other linear algebra library for diagonalization, which could significantly improve performance of this part of simulation.

GROMACS implements parallelization and vectorization for PME calculation and other internal routines but Group cut-off scheme does not support internal parallelization of any kind.

3.2 Profiling

3.2.1 Testing environment

All tests are performed on machine containing Intel® Core™ i7-5820K CPU @ 3.30GHz processor having 6 physical cores with hyperthreading capabilities. Testing machine is also equipped with two NVIDIA GeForce GTX 980 GPU cards with CUDA Compute Capability 5.2.

For testing the performance, following software tools and libraries were used in their indicated versions:

- GROMACS 5.0
- DFTB implementation for GROMACS 5.0 version 6a
- Plumed 2.1.5
- LAPACK 3.6.0
- ATLAS 3.10.3
- MKL 11.0.2

- MAGMA 2.2.0
- CUDA 7.5.17

3.2.2 Testing data

For the efficient and reliable testing of the performance a series of molecules containing increasing number of atoms is necessary. With the high level ab initio methods a molecule would be described by dense matrices and the performance of the underlying diagonalization scale similarly for systems with similar size. As mentioned above, the features of the DFTB Hamiltonian and Overlap matrices depend not only on the size of the studied system, but also on the arrangement of the atoms in molecules. The non-zero element originate from the nearest neighbor atoms and their chemical surroundings. The matrices for parts of protein or nucleic acid growing in size have different structure and the scaling could be more complicated. To overcome this complication a system consisting of increasing number of QM water molecules within a pre-equilibrated box of 4141 waters was selected for the testing of DFTB/MM performance. This ensures that the molecular environment will be very similar among systems while bearing high number of neighboring contributions, well comparable to the protein or nucleic acid system. The choice of water also offers high flexibility in selecting the size of QM part without the need to partition the aminoacids or nucleic acid bases.

The following water systems of different size were used.

QMW	contains 36 QM water residues surrounded by 4105 MM waters in a 50x50x50Å simulation box. Dimension of square matrix processed in diagonalization is 216.
QMW+4A	contains 133 QM water residues surrounded by 4008 MM waters in a 50x50x50Å simulation box. Dimension of square matrix processed in diagonalization is 798.
QMW+8A	contains 344 QM water residues surrounded by 3797 MM waters in a 50x50x50Å simulation box. Dimension of square matrix processed in diagonalization is 2064.
QMW+12A	contains 595 QM water residues surrounded by 3546 MM waters in a 50x50x50Å simulation box. Dimension of square matrix processed in diagonalization is 3570.

All testing simulations use LAPACK library, Group non-bonding cut-off scheme, time step of 0.5 femtosecond and performs 10 steps unless it is told otherwise.

3.2.3 Individual parts of simulation

There are 4 dominant parts in each step of simulation. The first part is molecular mechanics (MM) simulation that runs on start of every step. Second part is composed by calculations before self-consistent charge part of DFTB simulation (SCC). Third part is SCC calculation itself. Last part of one step is post-SCC calculation of forces.

SCC calculation is composition of multiple iterations. Every iteration can be divided to three parts: PME calculation, diagonalization of the Hamiltonian matrix and the rest of calculations.

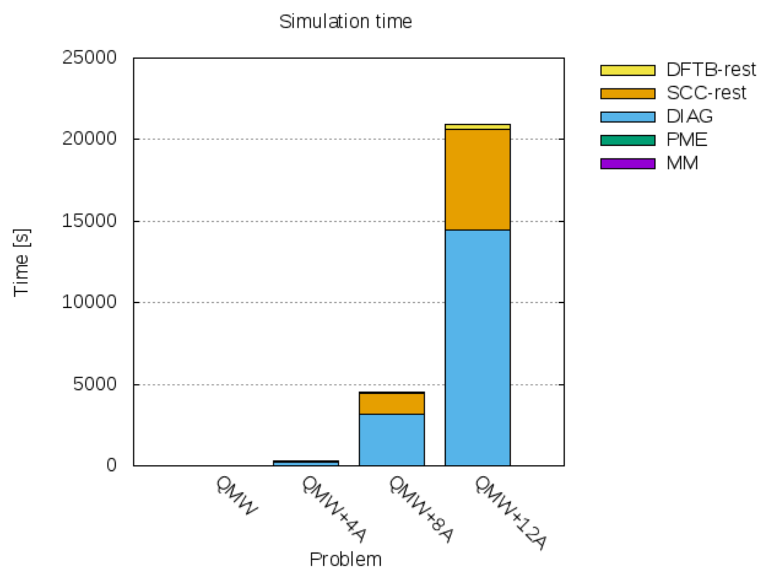


Figure 3.1: Running time of individual parts of simulation

Time distribution of individual steps is visible on figure 3.1. Time of diagonalization is growing rapidly with the amount and type of QM atoms the problem contains. Huge amount of time is also spent in the rest of the SCC cycle. From the time results it is visible that the time of MM and PME calculation is negligible.

3.2.4 Performance impact by used linear algebra library

Time of diagonalization is dependent on performance of used library implementing LAPACK routines. Some of them support multithreading, some of them are simply single-threaded. GROMACS supports any linear algebra library that is using standard names for routines including LAPACK[14] , ATLAS[15] and MKL[17] which are presented.

LAPACK is basic one-threaded library implementing linear algebra routines.

ATLAS implements some linear algebra methods in parallel but it does not implements all of them. The rest of methods is omitted or imported from LAPACK library. Although ATLAS supports multithreading it does not allow to specify the number of threads to be used. Maximal number of threads is determined at compile time from number of cores. It can be used on any platform.

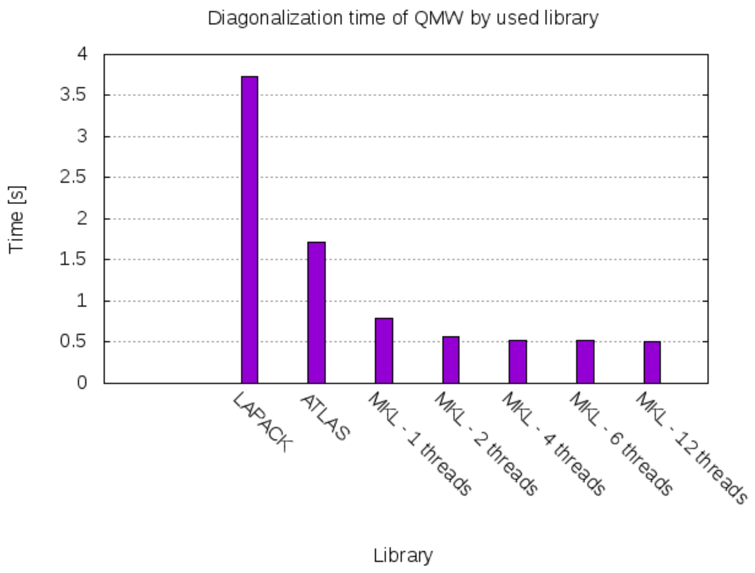
MKL is developed by Intel and has support only on Intel processors. It provides multithreaded versions of LAPACK methods with further optimizations for used processors. Number of used threads can be set via environmental variable `MKL_NUM_THREADS`.

As shown in figure 3.2 diagonalization time while using LAPACK library is really high. On smaller problems it is few times slower than other libraries but for bigger problems the difference is significant. The reason is that LAPACK library can use only single thread for computation and does not have any further optimizations.

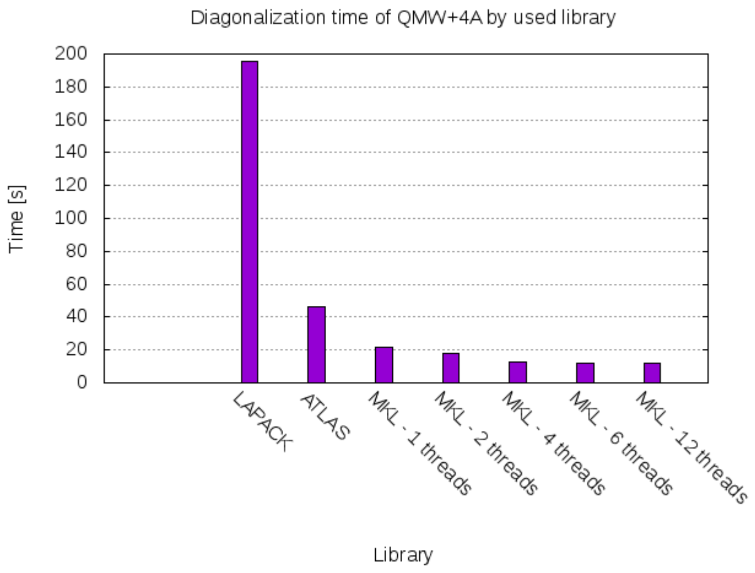
Performance of the ATLAS library is quite better, but it is not as good as MKL. Advantage of ATLAS is that it is not dependent on hardware and it can be used on all processors. Sadly, this has impact on the performance of the library. Another disadvantage may be that number of used threads cannot be specified and is determined in compilation of library itself. This may be a problem if target system is partially used.

MKL is highly optimized solution for Intel processors and the results prove it. Even with one thread the performance is better than the performance of ATLAS using multiple threads. Using multiple threads leads to additional performance boost even for small problems.

For testing purposes, all mentioned versions were compiled without GPU support and use the version of FFTW downloaded during GROMACS installation.

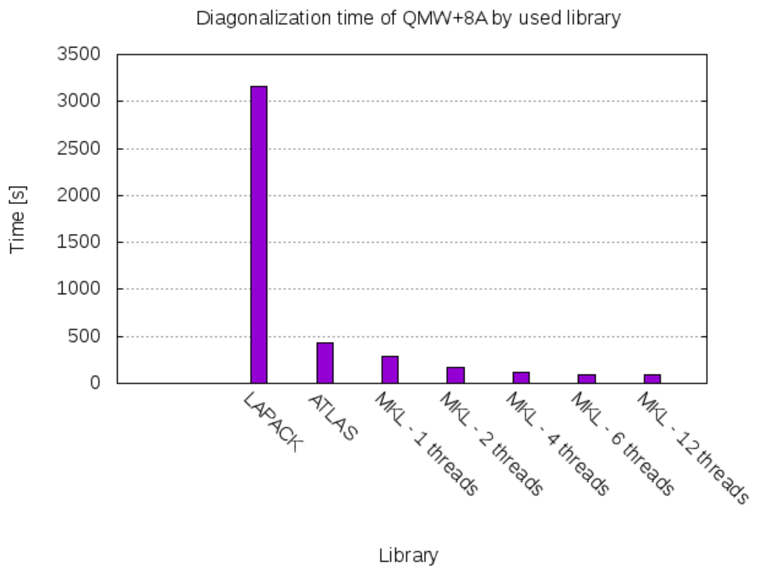


(a) QMW

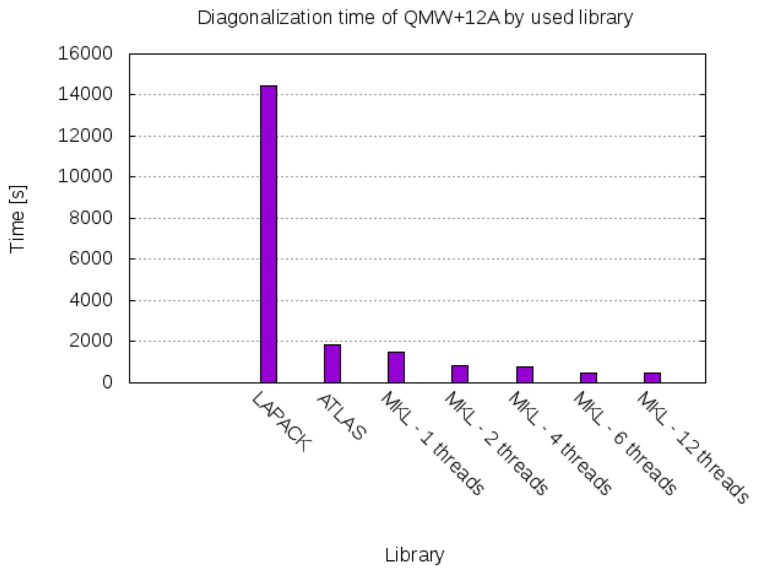


(b) QMW+4A

Figure 3.2: Diagonalization time by used library



(c) QMW+8A



(d) QMW+12A

Figure 3.2: Diagonalization time by used library (cont.)

Chapter 4

Optimization plans

4.1 Diagonalization

There is huge amount of time spent in diagonalization of the matrix as the analysis has shown. There are many libraries that accelerate this routine in parallel on CPU and some of them are present in the analysis results. But there are also libraries that use GPU acceleration. Combining GPU and CPU acceleration could result in further speedup.

4.1.1 MAGMA

4.1.1.1 About

MAGMA[18] is an open project started at University of Tennessee. It is a hybrid library using CPU as well as GPU for the acceleration of some linear algebra routines. It can enable many/multicore systems with multiple GPUs to use all of the resource available.

MAGMA is programmed for CUDA capable devices and requires basic linear algebra library. MAGMA currently supports MKL, ATLAS, OpenBLAS and ACL (AMD Compute Library) libraries to be used. Those libraries are used for the CPU part of calculation. Custom code is used for the GPU part.

Naming of routines is slightly different for MAGMA. It has multiple variants for almost each implemented routine. It is possible to choose number of GPUs to be used and it is possible to set if CPU or GPU memory is going to store computed matrix at the beginning of calculation. Used variant is determined from the name of called function. MAGMA implemented routines have prefix `magma_` and suffix dependent on chosen options. Available suffixes:

	Without a suffix the single GPU algorithm will be used with matrix in CPU memory
_m	Multiple GPUs can be used

_gpu Matrix is stored in GPU memory

_mgpu Multiple GPUs can be used and matrix is stored in GPU memory

As the middle part of the function name the usual LAPACK name of the routine is used.

Number of used GPUs and threads is determined from used environmental variables. For number of GPUs the value of `MAGMA_NUM_GPUS` variable is used. Number of threads in routine is determined by variable `MKL_NUM_THREADS` if MAGMA is compiled with MKL, `OMP_NUM_THREADS` is used otherwise. For parallel blocks the number of threads can be set in `MAGMA_NUM_THREADS` variable. If it is not set, value in `OMP_NUM_THREADS` will be used.

4.1.1.2 Library instalation

MAGMA library has similar dependencies as GROMACS. It needs C and C++ compiler, CUDA compiler and fortran compiler. BLAS library is also needed to compile library. MAGMA in version 2.2 supports ACML, ATLAS, MKL and OpenBLAS.

Before the compilation itself it is needed to create `make.inc` file. There are many examples that can be copied and edited to match targeted configuration. Those examples are using environmental variables like `CUDADIR` and `MKLROOT`, which must be set properly.

4.2 DFTB code

Current DFTB implementation is simple single-thread program. This code can be parallelized to speedup the calculation by dividing work between multiple threads. In addition to that, lot of operations on arrays may be vectorized. Both optimizations are described in appropriate sections below.

4.2.1 Vectorization

Modern processors have the capability to perform calculations on vectors – multiple adjacent values. For such operations special instructions are available in the instruction set. One operation can be performed on multiple values at the same time on single processor by using those instructions.

Vectorization is process where scalar implementation is converted to vector implementation. This conversion is done by compiler and can be done automatically on some of them. To improve the results of vectorization compiler could be instructed to process vectorization using different methods. To perform this such methods must be supported. The most widely used solution is OpenMP which specifications are supported in most of the major compilers. By using OpenMP specific constructs the compiler can be provided with informations about alignment, byte size, steps, etc. With OpenMP, collapsing of inner cycles and even reduction can be used.

In this case there are many parts of code where a single operation is performed on all elements of an array. Vectorization of those parts of code will lead to faster processing of those operations.

4.2.2 Parallelization

All modern processors have the capability to process multiple operations at the same time by using multiple cores. Many of those processors have hyperthreading capability which enables one physical core to process multiple threads at the same time. This is limited by operations that can be performed simultaneously on one core.

Running code in parallel can improve the performance of the simulation by running specified operations on multiple elements simultaneously using multiple threads. It is widely used form of acceleration on single machine with multiple cores or processors. As was told before, running code in parallel can increase performance of application, but can also decrease it, if it is used incorrectly. The main problem of parallelization is synchronization of threads and its management. There are many ways to manage and synchronize threads using one of many available libraries. Many of those libraries are not available on multiple platforms or systems, some of them have specific usage, etc.

OpenMP, which is used for vectorization of code, also supports parallelization. It is not needed to create any structures in code or to create threads by calling specific functions while using OpenMP. Parallelization is completed by supported compiler using OpenMP specifications the same way the vectorization is processed. Only thing that is needed is to specify parallel blocks for compiler by using specific constructs. Those constructs allows to divide loops between multiple thread, create tasks, and more. While parallelizing loops, collapsing nested loops, reduction, scheduler, shared and private variables and many more options can be defined. Also, number of used threads can be adjusted using `OMP_NUM_THREADS` environmental variable.

In DFTB code, there is a lot of loops that can be processed in parallel. Most of the loops work with single element and its neighbors in one iteration of the loop. Data changes are for current element only, so those operations can be safely run in parallel without any additional needs for critical sections.

4.3 Verlet non-bonding cut-off scheme

Verlet cut-off scheme is an internal scheme of GROMACS for persisting charge data. Usage of Verlet cut-off scheme could improve performance of used GROMACS internals, however there might be some differences in the calculation. Verlet cut-off scheme provides CPU and GPU acceleration for PME calculations and for other vector operations. However GPU acceleration is only available while using single precision. Number of used threads can be specified by parameters specified in section 2.3.2.

Chapter 5

Implementation

5.1 MAGMA

To make MAGMA working with GROMACS it was needed to include header files for use in DFTB initialization and diagonalization to the code and integrate MAGMA dependencies into cmake files of the project.

5.1.1 Source code

Specific headers must be included in all files, where MAGMA methods are being used. Because GROMACS does not have to be compiled with GPU and MAGMA support and needs to remain universal, it is needed to evaluate some conditions from build configuration. Because of that, all stated conditions are evaluated in compile-time.

MAGMA headers are included in `qm_dftb.c` file, where the initialization of MAGMA environment is completed, and in `qm_dftb_eglcao.c` file, where diagonalization of matrix is processed. Both files are located in `src/gromacs/mdlib` directory inside the project sources. Header files are included only if variable `GMX_GPU_MAGMA` is defined by configuration. Otherwise MAGMA is not used.

In the `qm_dftb.c` file the initialization of DFTB is processed. If magma is used, which is determined with the exactly same condition as includes, `magma_init` function is called. This initializes MAGMA internals and prepare MAGMA environment.

Function calling diagonalization routine have been rewritten to call the proper method in `qm_dftb_eglcao.c` file because MAGMA methods have custom prefix and suffix. Proper method is determined in compile-time by the definitions of variables `GMX_DOUBLE` for recognition of precision and `GMX_GPU_MAGMA` for recognition of MAGMA usage. Based on the evaluation of this condition, proper function is used – `magma_dsygvd_m` for double precision and `magma_ssygvd_m` for single precision. Those are multi-GPU versions of used routine. Therefore there are more arguments passed to this function instead of appropriate LAPACK method. Some arguments must be passed as MAGMA-specific constants and those variables must be converted. Storing data on GPU using _mgpu suffix is not supported for this routine. Prototype of the magma_dsygvd_m function is described below.

```
magma_int_t magma_dsygvd_m (
    magma_int_t         ngpu,
    magma_int_t         itype,
    magma_vec_t         jobz,
    magma_uplo_t        uplo,
    magma_int_t         n,
    double *            A,
    magma_int_t         lda,
    double *            B,
    magma_int_t         ldb,
    double *            w,
    double *            work,
    magma_int_t         lwork,
    magma_int_t *       iwork,
    magma_int_t         liwork,
    magma_int_t *       info
)
```

At first, number of GPUs must be set as the ngpu. This value is obtained by calling magma_num_gpus method and is determined from environmental variable MAGMA_NUM_GPUS. Using this variable, user is able to specify number of used GPUs, but cannot specify, which GPUs are going to be used. Specification of such information is not possible. Variables jobz and uplo must be converted to corresponding MAGMA types using magma_vec_const and magma_uplo_const methods respectively. The last change in the function prototype is that MAGMA is not expecting pointers for read-only arguments.

The final code of selecting appropriate routine looks like

```
 static long sygvd(long itype, char jobz, char uplo, long n,
↪   double *a, long lda, double *b, long ldb, double *w, double
↪   *work, long lwork, long *iwork, long liwork)
 {
  long info;

  #if !defined GMX_GPU AND defined GMX_DOUBLE
    extern void dsygvd_(long *, char *, char *, long *, double
↪   *, long *, double *, long *, double *, double *, long *,
↪   long *, long *, long *);
    dsygvd_(&itype, &jobz, &uplo, &n, a, &lda, b, &ldb, w, work,
↪   &lwork, iwork, &liwork, &info);
```

```
  #elif !defined GMX_GPU
    extern void ssygvd_(long *, char *, char *, long *, double
↪   *, long *, double *, long *, double *, double *, long *,
↪   long *, long *, long *);
    ssygvd_(&itype, &jobz, &uplo, &n, a, &lda, b, &ldb, w, work,
↪   &lwork, iwork, &liwork, &info);

  #elif defined GMX_GPU_MAGMA AND defined GMX_DOUBLE
    magma_dsygvd_m(magma_num_gpus(), itype,
↪   magma_vec_const(jobz), magma_uplo_const(uplo), n, a, lda, b,
↪   ldb, w, work, lwork, iwork, liwork, &info);

  #elif defined GMX_GPU_MAGMA
    magma_ssygvd_m(magma_num_gpus(), itype,
↪   magma_vec_const(jobz), magma_uplo_const(uplo), n, a, lda, b,
↪   ldb, w, work, lwork, iwork, liwork, &info);

  #endif
  return info;
}
```

5.1.2 CMAKE

To use MAGMA functions, the library must be located in the system and properly imported if needed.

Necessity of GPU library is determined by simple condition that GPU library is only needed if GPU acceleration is enabled and used QMMM program is DFTB. This is represented by variables `GMX_QMMM_PROGRAM` to be equal `DFTB` and `GMX_GPU` set to `ON`. In this case `GMXDFTBOnGPU.cmake` file is included and used for further processing. Otherwise GPU library is not necessary and no processing will be done.

New file named `GMXDFTBOnGPU.cmake` is used to process requirements of running DFTB on GPU. In this file, used library is determined from value of cmake variable `GMX_DFTB_GPU_LAPACK`. This variable currently supports only value `magma` indicating that MAGMA will be used. This value is also used as default, so it is safe to leave the variable unset. This variable is created for potential support of multiple GPU libraries.

If MAGMA is chosen to be used it is localized in the system and included in list of GPU libraries the code must be linked to. For the purpose of locating MAGMA in filesystem cmake variable `GMX_GPU_MAGMA_DIR` is used. By using this variable, user can define location of the library in the system. If the variable is not set and the GPU support is enabled default path (/usr/local/magma) is used. If MAGMA is not present in the specified folder error message will be written and user will be suggested to use `GMX_GPU_MAGMA_DIR` variable with path to MAGMA root folder.

Contents of `GMXDFTBOnGPU.cmake`:

```
gmx_option_multichoice(
  GMX_DFTB_GPU_LAPACK
  "GPU LAPACK library"
  magma
  magma)

if(${GMX_DFTB_GPU_LAPACK} STREQUAL MAGMA)
  set(GMX_GPU_MAGMA 1)
else()
  gmx_invalid_option_value(GMX_DFTB_GPU_LAPACK)
endif()

if(GMX_GPU_MAGMA)
  option(GMX_GPU_MAGMA_DIR "Magma directory root"
↪   "/usr/local/magma")
  include_directories(${GMX_GPU_MAGMA_DIR}/include
↪   /usr/local/cuda/include)
  find_library(MAGMA_LIBRARY magma ${GMX_GPU_MAGMA_DIR}/lib/)
  message("Magma location:" ${MAGMA_LIBRARY})
  if(MAGMA_LIBRARY)
    set(GMX_GPU_LIBRARIES ${GMX_GPU_LIBRARIES} ${MAGMA_LIBRARY})
  else()
    message(FATAL_ERROR "Magma was not found. You can define
↪   GMX_GPU_MAGMA_DIR with magma root directory")
  endif()
endif()
```

Because GROMACS does not support GPU acceleration while double precision is used but MAGMA does, compilation with both variables `GMX_GPU` and `GMX_DOUBLE` set to `ON` was enabled for the DFTB method. This may lead to problem while the Verlet non-bonding cut-off scheme is used, because its GPU support may be used, although it is only implemented for single precision, but compilation was forced due the usage of MAGMA library. In this case the simulation has to be started with parameter `-nt cpu` that cause GROMACS to use only CPU for acceleration.

5.2 OpenMP optimizations

5.2.1 Parallelization

5.2.1.1 qm_dftb_eglcao.c

Whole code for one DFTB step is parallelized using one `parallel` block. This block is composed by many `for` loops and some `single` constructs. Number of threads to be used in the parallel block is determined from environmental variable `DFTB_NUM_THREADS` in initialization of DFTB and is saved in global variable. Omitting this environmental variable sets number of threads in this parallel section to 1. Because of the GROMACS internal optimizations by OpenMP and parallelization of other files described below, nested parallelism is enabled for this block. This means that every one of the threads can create new thread group. Changes described below are processed in a way, that running more than the defined number of threads is not possible.

Private variables are defined and the rest is left to be shared between threads for the whole parallel block. Many of those private variables are used in loops as iteration counters and some of them as local variables with subresults. For some of parallelized cycles new variables must be created out of shared structures. Those variables are not used outside the cycle but were included in shared structure by the original solution.

Most of the `for` blocks use static scheduler. Static scheduler splits all iterations between available threads equally at the beginning of the loop. There is no communication between threads till the end of the loop, where synchronization will occur unless told otherwise. If synchronization is not needed and done threads can continue in next processing `nowait` clause is used. Static scheduler is used for loops where all iterations will do the same procedure and therefore computation time of every iteration should be the same.

Guided scheduler is used for loops where equal times for all iterations cannot be guaranteed. This is caused mostly by conditions in the procedure. Guided scheduler divides work between threads dynamically. Assuming that `n` is number of used threads, the first thread will begin with $^1/_n$ of the iterations, second thread will get $^1/_n$ of the rest of iterations and so on. If any thread completes its work, it tries to get next $^1/_n$ of remaining work. Minimum number of iterations given for processing is set to 5. If there is none work left, thread will wait for others to finish unless `nowait` clause is used.

There is also one specific cycle shown below where `reduction` clause is used. This option utilizes parallel reduction of specified variables. Till the end, all threads are using their private variables to store the subresults. In the end, specified operation combines all those subresults into the result. Reduction is utilizing multiple variables.

```
#pragma omp for reduction(+:ecoul,ecoul3,eext) schedule(static)
for (i=0; i<nn; i++) {
  ecoul  += dftb1.shift[i] * (dftb1.qmat[i] +
↪  dftb->qzero1[dftb1.izp[i]]);
  ecoul3 += dftb1.shift3[i] * (dftb1.qmat[i] +
↪  dftb->qzero1[dftb1.izp[i]]) + dftb1.shift3a[i] *
↪  dftb1.qmat[i];
  eext   += dftb1.shiftE[i]  * QM_CHARGE(i);
  if (dftb->cdko)
    eext += 2. * (dftb1.shiftE[i] + dftb1.shiftE2[i]) *
↪  dftb1.qmat[i]; // the factor of 2 because eext will be
↪  divided by 2 below
}
```

Some of original cycles were merged because of their independence and to eliminate synchronization overhead.

`Single` blocks are mainly used to print the outputs and to perform some internal routines that are done once for all the elements. Many of those routines are parallelized in their own files described below.

Because the number of SCC cycles is not known at the beginning of the SCC loop, but is determined by calculation processed inside the loop, breaking the SCC cycle had to be rewritten. This computed variable, which may cause the end of the loop, was calculated by the previously mentioned cycle using reduction. Problem is that the value, which may cause the break of the cycle, was changed by single thread right after the breaking condition. Because the variable may be changed before all threads have evaluated the condition, it is sure, that all of remaining threads will evaluate this condition as true and exit the loop, which will cause the deadlock. To prevent this, barrier was inserted right after the decision to end. This ensures that all of threads will evaluate the condition with the exact same values.

5.2.1.2 qm_dftb_dispersion.c

This file contains one method with one cycle that was parallelized using guided scheduler. This loop performs reduction of one variable and changes in gradient vectors. Those vector changes are performed in critical section of the code because multiple threads could work with the same vector.

5.2.1.3 qm_dftb_dispersion_dftd3.c

Multiple methods are contained in this file. Some initialization loops were parallelized but most of them reads from some files and parallelization of those is therefore not possible. Some loops in other methods were parallelized using various options like collapsing and reduction. The one containing the calculation itself was parallelized in most of the parts but the main loop performing majority of calculations cannot be parallelized because of data dependencies on values calculated in previous iterations.

5.2.1.4 qm_dftb_fermi.c

This file does not contain any suitable loop for parallelization. Only loop available in this method has number of iterations dependent on values calculated within and therefore cannot be parallelized by OpenMP.

5.2.1.5 qm_dftb_gamma.c

Three cycles were parallelized in this file. One is simple loop clearing vectors used in this calculation which was parallelized using static scheduler. Other loops were parallelized using guided scheduler because called methods are data dependent and does not assure the same processing time.

5.2.1.6 qm_dftb_gammamat.c

Two cycles was parallelized using guided scheduler because of data dependent code.

5.2.1.7 qm_dftb_gradient.c

In usual_gradient method parallel block was created. This method contains a loop where all interactions between elements are calculated. Part of this calculation temporarily changes data of computed element while working with data of other elements. This behavior in parallel environment may cause a situation, where other thread is working with those temporarily changed data and thus becomes to accumulate error. To prevent this a copy of elements data array is created for each thread and temporary changes are done on this copy. The original data copy keeps the correct data all the time.

In addition to those changes, some variables were previously part of shared dftb structure, but those variables have to be used privatelly in each thread.

Edited usual_gradient method is shown in below. Variables au, auh, bu and buh were created locally and used in appropriate places. Variable x_copy is used to store copy of x vector array. Memory for this copy is allocated by each thread.

```
void usual_gradient(dftb_t *dftb, dvec *x, dvec *grad)
{
  int i, j, k, izpj, izpk;
  int m, n, indj, indk, lj, mj, nu, lk, mk, mu;
  double ocmcc, xhelp, dgrh, dgrs, dgr, dgr3;
  double au[LDIM][LDIM], bu[LDIM][LDIM], auh[LDIM][LDIM],
↪   buh[LDIM][LDIM];
  dvec * x_copy;
  const double deltax = 0.01;
  dftb_phase1_t dftb1 = dftb->phase1;
```

```
dgr = dgr3 = 0.;

#pragma omp parallel num_threads(dftb_threads_get())
↪  private(m, n, j, i, k, ocmcc, indj, indk, izpj, izpk, xhelp,
↪  mu, nu, dgrh, dgrs, dgr, dgr3, au, auh, bu, buh, lj, mj, lk,
↪  mk, x_copy)
{
  #pragma omp for simd collapse(2)
  for (m=0; m<dftb1.norb; m++)
    for (n=0; n<dftb1.norb; n++)
      dftb1.b[m][n] = 0.e0;

  #pragma omp single
  for (i=0; i<dftb1.norb; i++) {
    if (dftb1.occ[i] < dftb->dacc)
      break;
    for (m=0; m<dftb1.norb; m++)
      for (n=0; n<m; n++) {
        ocmcc = dftb1.occ[i] * dftb1.a[m][i] * dftb1.a[n][i];
        dftb1.b[n][m] += ocmcc * dftb1.ev[i];
        dftb1.b[m][n] += ocmcc;
      }
  }
  #pragma omp for schedule(static)
  for (m=0; m<dftb1.norb; m++)
    for (n=0; n<m; n++)
      if (fabs(dftb1.b[m][n]) < dftb->dacc)
        dftb1.b[m][n] = 0.e0;

  /* copy of x needed for each thread! */
  snew(x_copy, dftb->atoms);
  for(j=0; j<dftb->atoms; j++)
    for(i=0; i<3; i++)
      x_copy[j][i] = x[j][i];

  #pragma omp for schedule(guided, 1)
  for (j=0; j<dftb->atoms; j++) { /* for every atom that
↪  forces act upon */
    indj = dftb1.ind[j];
    izpj = dftb1.izp[j];
    for (k=0; k<dftb->atoms; k++) if (k != j) { /* for every
↪  atom acting on the studied atom j */
      indk = dftb1.ind[k];
```

```
        izpk = dftb1.izp[k];

        for (i=0; i<3; i++) {
          xhelp = dftb1.x[j][i];
          x_copy[j][i] = xhelp + deltax;
          slkmatrices(k, j, x_copy, au,  bu,  dftb->lmax,
↪   dftb->dim1, dftb->dr1, dftb1.izp, dftb->skstab1,
↪   dftb->skhtab1, dftb->skself1);
          x_copy[j][i] = xhelp - deltax;
          slkmatrices(k, j, x_copy, auh, buh, dftb->lmax,
↪   dftb->dim1, dftb->dr1, dftb1.izp, dftb->skstab1,
↪   dftb->skhtab1, dftb->skself1);
          x_copy[j][i] = xhelp;
          for (lj=1; lj<=dftb->lmax[izpj]; lj++)
            for (mj=1; mj<2*lj; mj++) {
              n  = SQR(lj-1) + mj - 1;
              nu = n + indj;
              for (lk=1; lk<=dftb->lmax[izpk]; lk++)
                for (mk=1; mk<2*lk; mk++) {
                  m  = SQR(lk-1) + mk - 1;
                  mu = m + indk;
                  dgrh = (au[m][n] - auh[m][n]) / deltax;
                  dgrs = -(bu[m][n] - buh[m][n]) / deltax;
                  dgr = -0.5 * dgrs * (dftb1.shift[k] +
↪   dftb1.shift[j]);
                  if (dftb->sccmode == 3)
                    dgr3 = -0.5 * dgrs * (2.*dftb1.shift3[k] +
↪   dftb1.shift3a[k] + 2.*dftb1.shift3[j] +
↪   dftb1.shift3a[j])/3.;
                  if (mu > nu) {
                    dgrh *= dftb1.b[mu][nu];
                    dgrs *= dftb1.b[nu][mu];
                    dgr  *= dftb1.b[mu][nu];
                    if (dftb->sccmode == 3) dgr3 *=
↪   dftb1.b[mu][nu];
                  } else {
                    dgrh *= dftb1.b[nu][mu];
                    dgrs *= dftb1.b[mu][nu];
                    dgr  *= dftb1.b[nu][mu];
                    if (dftb->sccmode == 3) dgr3 *=
↪   dftb1.b[nu][mu];
                  }
                  grad[j][i] += dgrh + dgrs + dgr;
                  if (dftb->sccmode == 3) grad[j][i] += dgr3;
                }
            }
        }
      }
    }
    sfree(x_copy);
  }
  return;
}
```

Loops in other methods were parallelized without any changes in the code logic.

5.2.1.8 qm_dftb_mulliken.c

In Mulliken calculation one of the loops was parallelized using static scheduler and one using dynamic. One cycle was not parallelized.

5.2.1.9 qm_dftb_neighborlist.c

Cycle searching for element neighbors was parallelized using guided schedule, because of additional operations when neighbor is found.

5.2.1.10 qm_dftb_repulsive.c

Repulsive calculation was parallelized using reduction for `erep` variable. Because of gradient changes, critical sections were added to prevent collision on write.

5.2.1.11 qm_dftb_shift.c

Parallel block was created and loops were parallelized using static scheduler. No loops were collapsed because collapsing would cause data hazards.

5.2.1.12 qm_dftb_cdko.c

Loop inside `cdkopotential` was parallelized using guided scheduler, because number of iterations of inner cycle is dependent on number of neighbors of computed element. `cdkograd` method was parallelized using guided scheduler. Gradients are changed inside the function, so the critical section was used.

5.2.1.13 qm_dftb_broyden.c

Many of really short loops were parallelized using static scheduler. Some cycles were not parallelized because of data dependencies.

5.2.2 Vectorization

Vectorization of the code is also processed by OpenMP library and its `simd` construct. Because of the used parallelization, the `for simd` construct was used. It combines the effect of vectorization and parallelization so the vectorized code can be run in parallel.

All of vectorized loops contain simple operations like assigning static values in array or basic arithmetic operations.

To use this feature, compiler supporting OpenMP 4.0 or newer must be used. This means that GNU compilers version 4.9 or later or Intel compilers version 15 or later should be used. To enforce use of those, it is possible to specify path to C compiler in `CC` and C++ complier path in `CXX` environmental variable. Those versions might not be compatible with CUDA compiler, so path to CUDA compatible compiler must be set in `CUDA_HOST_COMPILER` variable.

CHAPTER 6

Results

6.1 MAGMA

Two MAGMA versions were used for testing purposes. Version where MKL is used as CPU linear algebra library and second version with ATLAS. Comparison of those versions is available in figure 6.1.

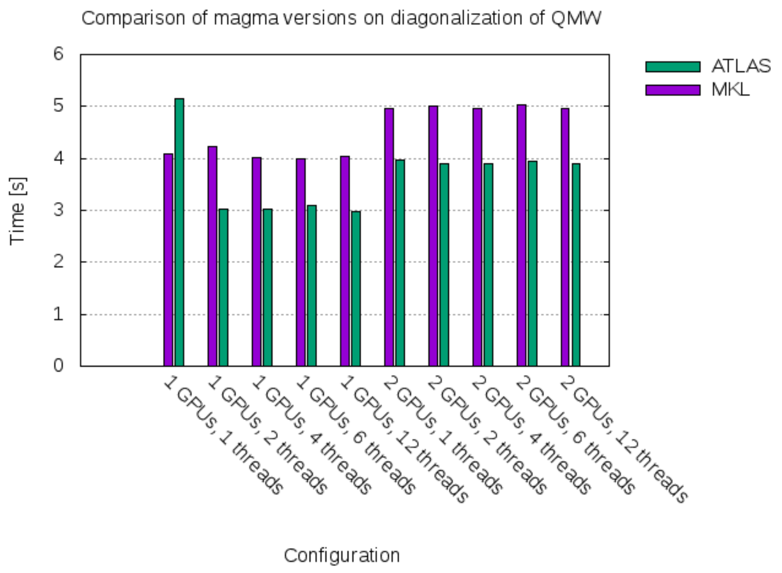
ATLAS version is faster in all test cases. Using multiple GPUs leads to larger speedup of bigger problems (6.1c, 6.1d) but can lead to slowdown while simulating only few atoms (6.1a, 6.1b). No additional speedup is provided by using more threads.

Figure 6.2 shows comparison of MAGMA with ATLAS and CPU libraries used in analysis.

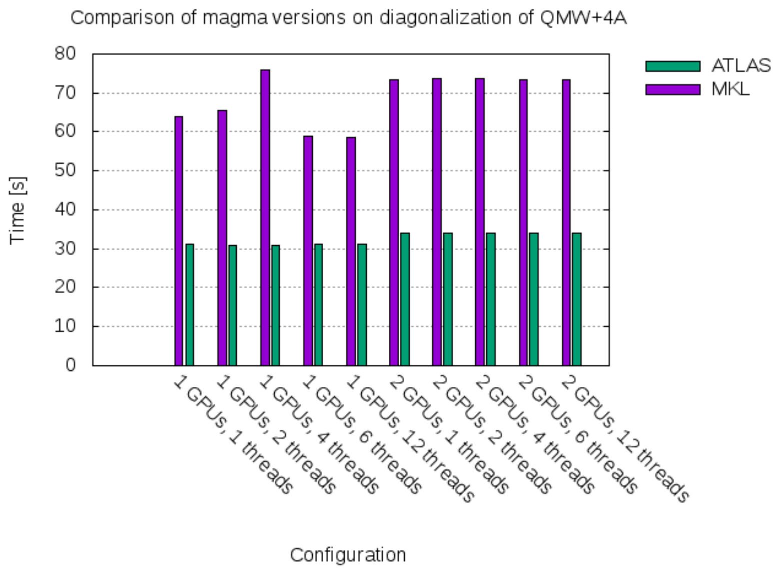
For really small problems as QMW (shown in 6.2a) MAGMA is slower than MKL or even ATLAS. This is mainly because of data transfers that needs to be done between CPU and GPU.

At figures 6.2b and 6.2c the speedup of MAGMA is visible even though MKL version is still faster for this size of problem.

At 6.1d MAGMA is the fastest solution. From these trends it is safe to assume that for even bigger problems MAGMA should be significantly faster than any CPU library.

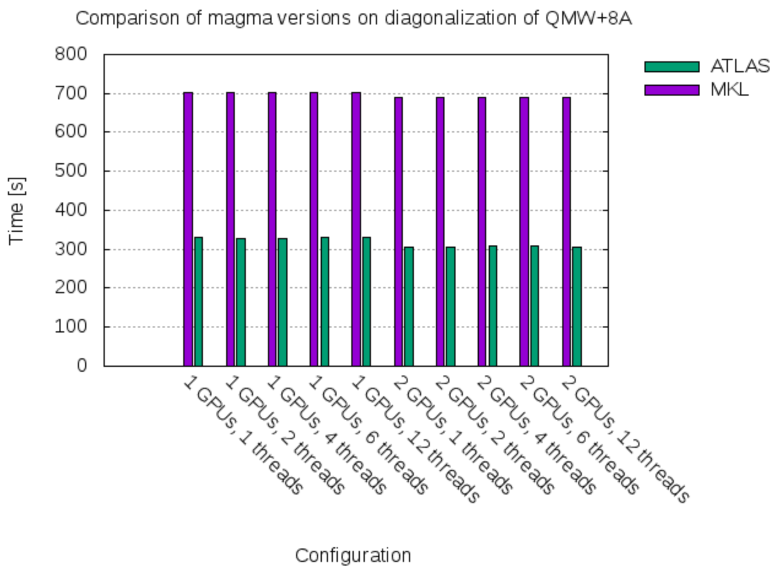


(a) QMW

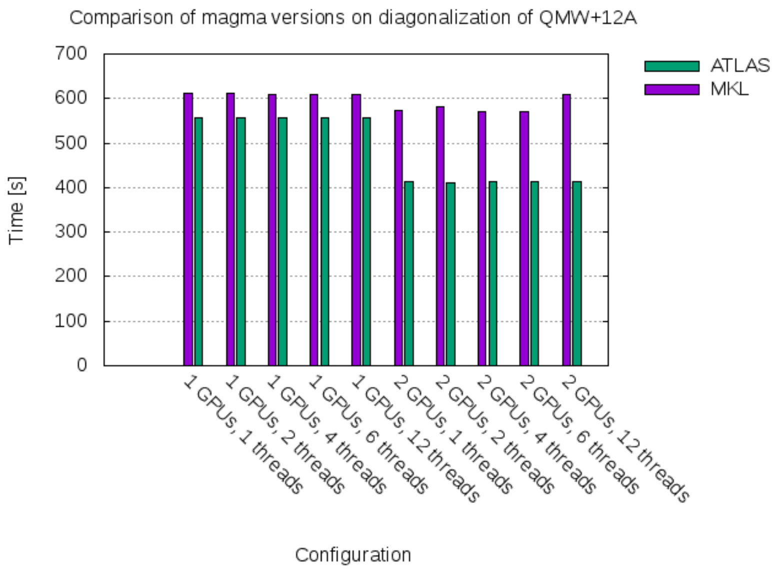


(b) QMW+4A

Figure 6.1: Comparison of diagonalization times of magma versions

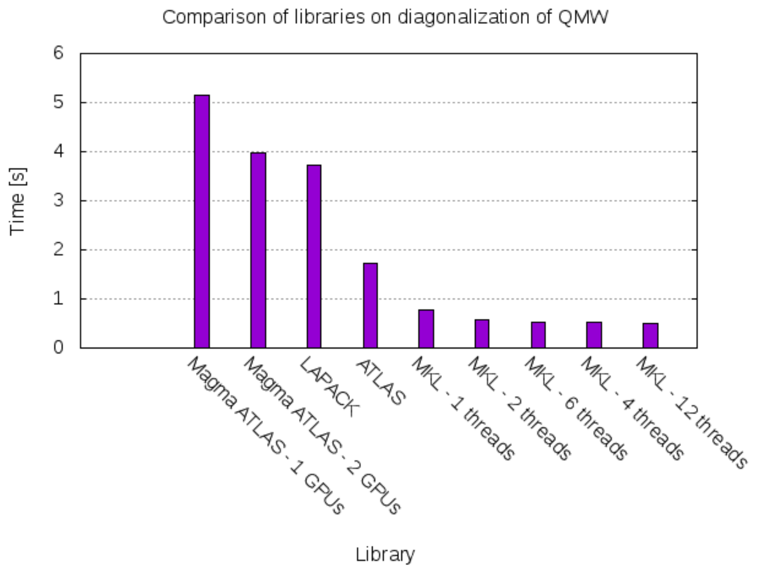


(c) QMW+8A

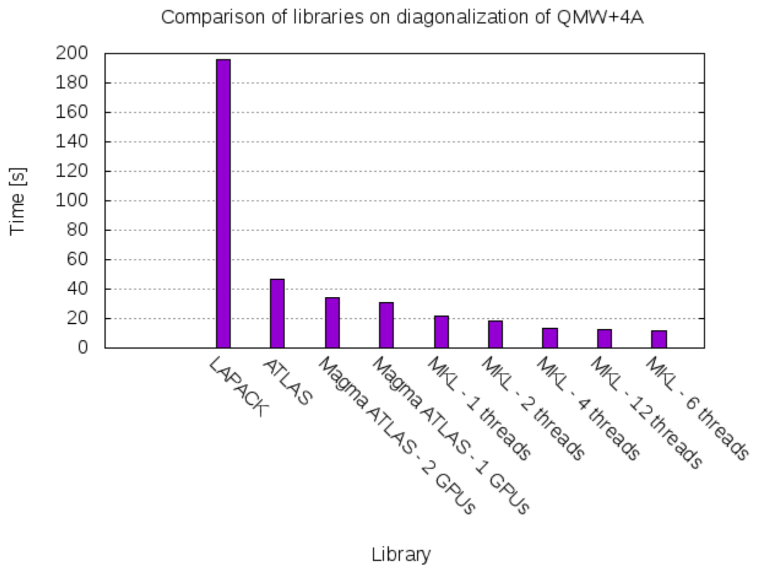


(d) QMW+12A

Figure 6.1: Comparison of diagonalization times of magma versions (cont.)

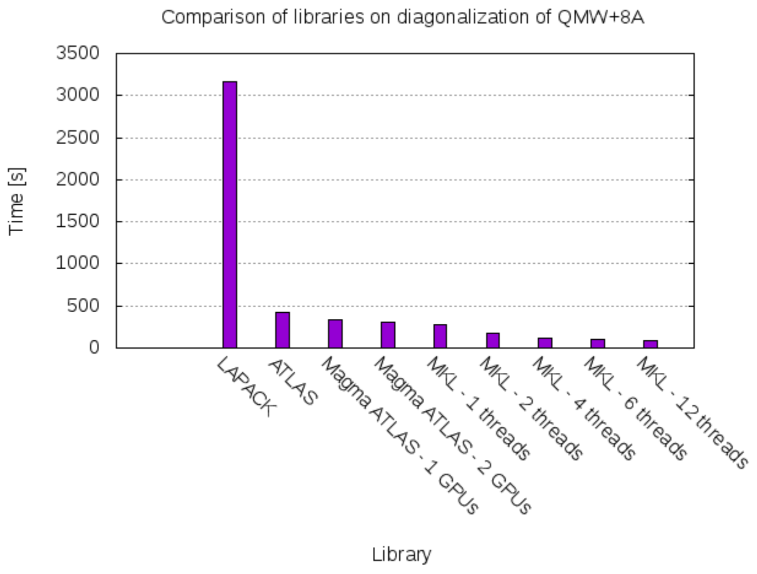


(a) QMW

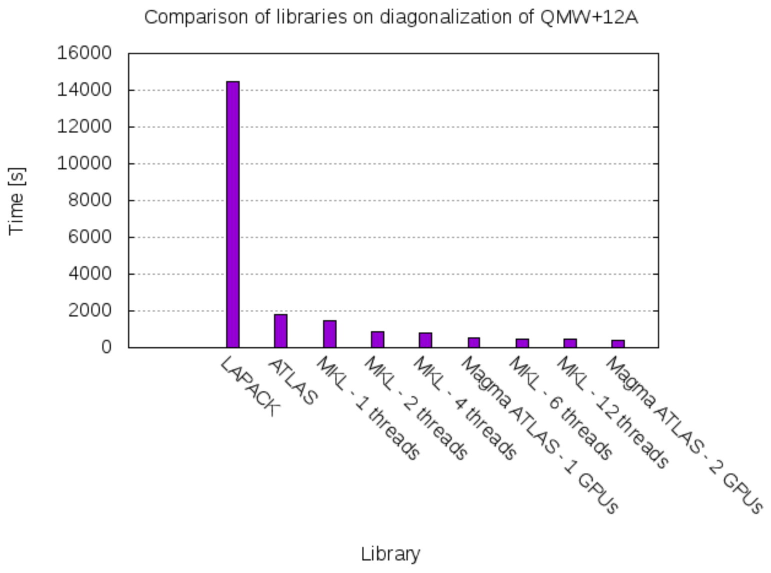


(b) QMW+4A

Figure 6.2: Comparison of diagonalization times using various linear algebra libraries



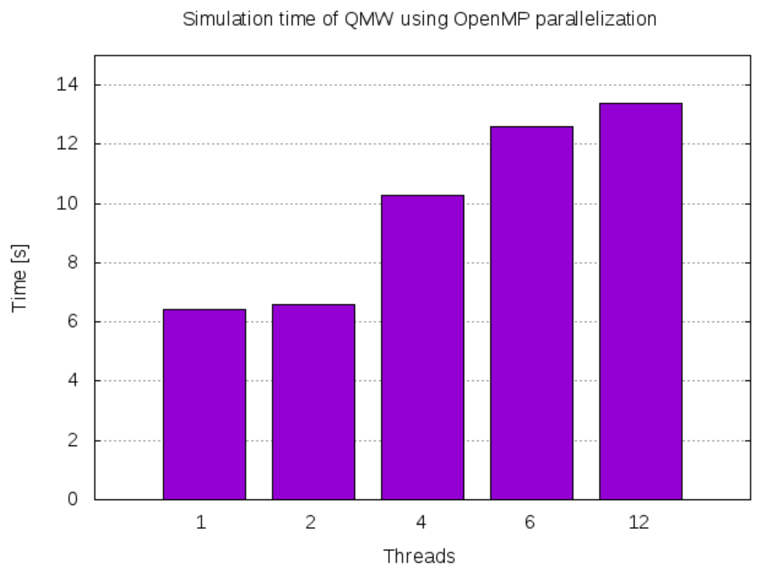
(c) QMW+8A



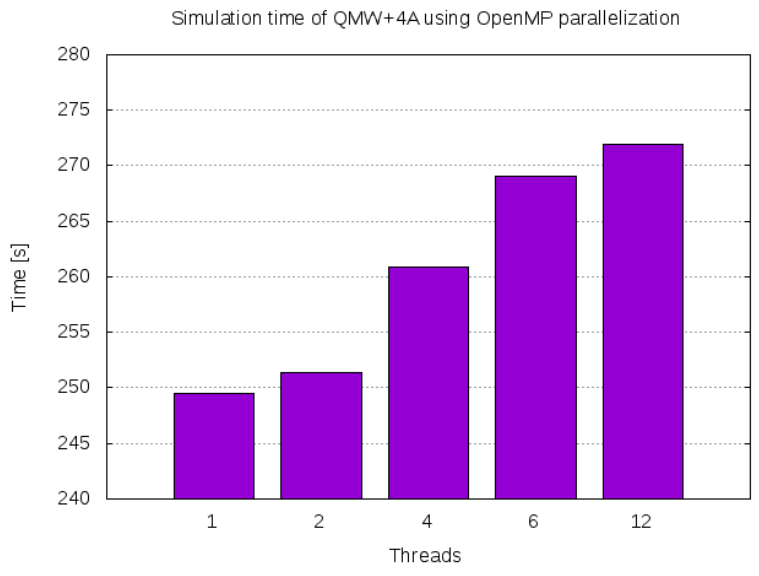
(d) QMW+12A

Figure 6.2: Comparison of diagonalization times using various linear algebra libraries (cont.)

6.2 OpenMP



(a) QMW



(b) QMW+4A

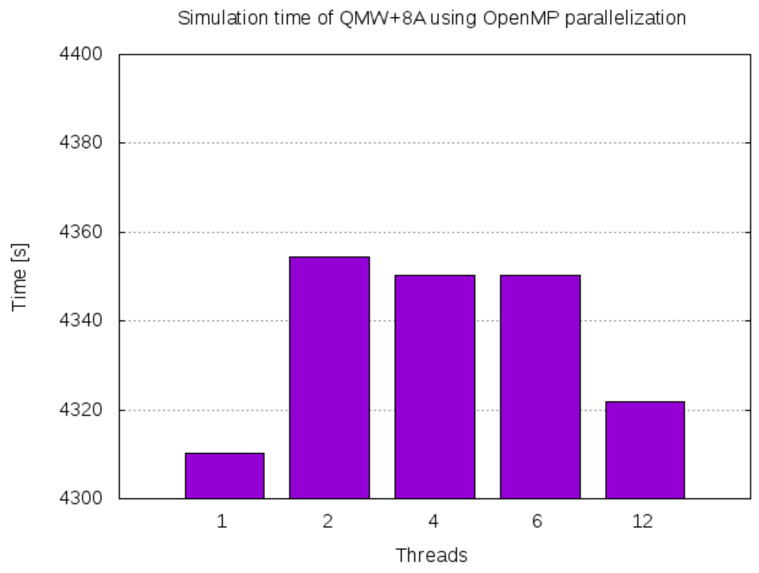
Figure 6.3: Times using OpenMP parallelization

Results of OpenMP parallelization and vectorization are shown at figure 6.3. There is almost none visible acceleration in the results.

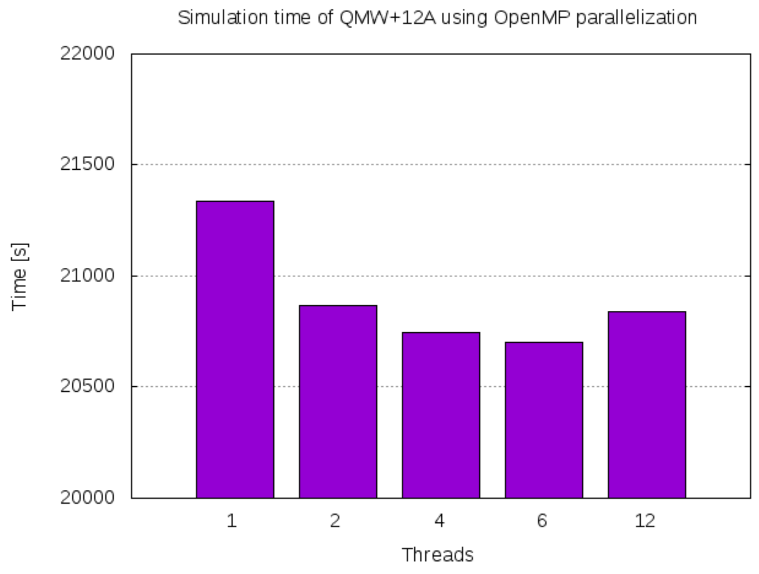
The SCC code is composed from large amount of fast loops (mostly one operation per element). Although those simple loops are easily parallelized the synchronization delay between threads is huge. For example: Solving small problem with two threads using static planning, when all iterations are equally divided between threads, may lead to situation, where one thread starts and completes even before the second thread starts the calculation. The completion of one loop in parallel is then much longer than using single thread. The synchronization time is growing with multiple loops placed in a sequence.

Because of these quick loops, using additional threads on small problems is quite contra productive and the results will be similar to those at figure 6.3a, where time is growing for more threads. For growing number of elements, the results are getting better. At figure 6.3d some improvement with multiple threads is visible. While using multiple threads, almost 10 minutes were saved in this example. For even bigger problems acceleration should be more significant.

Synchronization impact is specific for each machine and is dependent on system CPU time planning, load, performance, etc. Presented results are specific for tested conditions and may differ significantly on different systems.



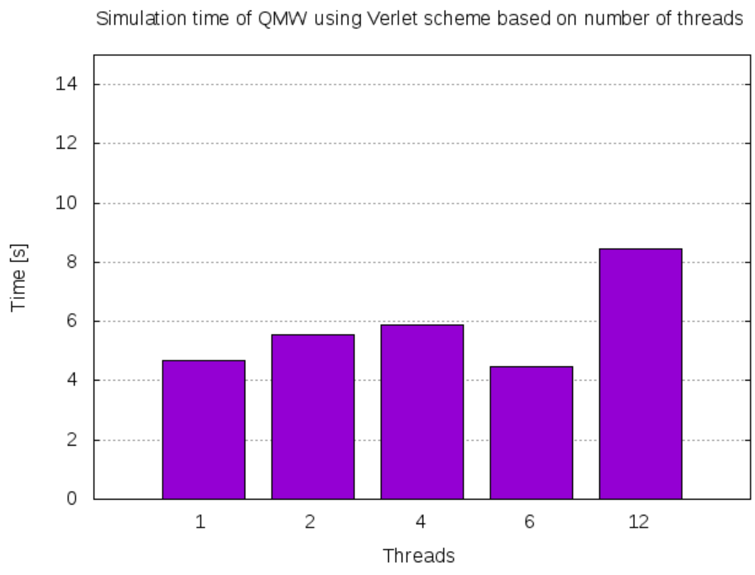
(c) QMW+8A



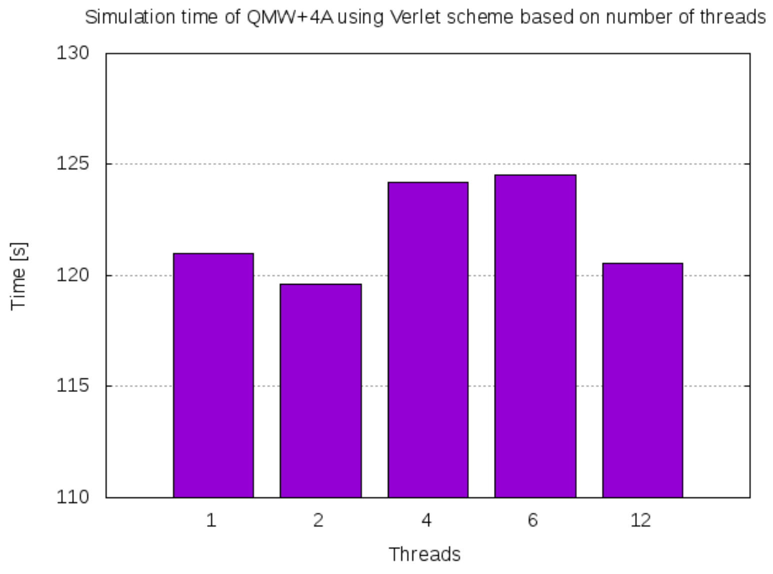
(d) QMW+12A

Figure 6.3: Times using OpenMP parallelization (cont.)

6.3 Verlet non-bonding cut-off scheme



(a) QMW



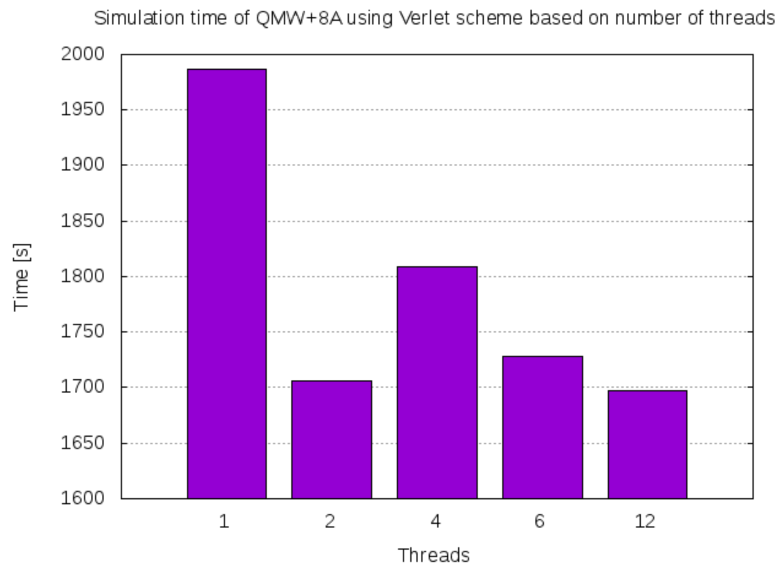
(b) QMW+4A

Figure 6.4: Times using Verlet non-bonding cut-off scheme

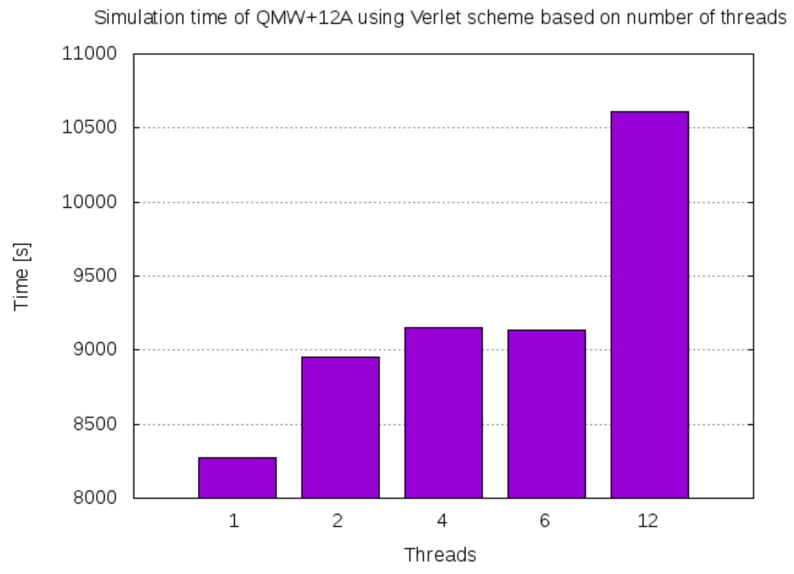
At figure 6.4 results of simulation using Verlet scheme are shown. There are results where using Verlet scheme accelerates the calculation (6.4c) but there are also results where using more threads does not mean any significant change in simulation speed (6.4b) and results, where using more threads even slows the simulation down (6.4a and 6.4d). There is no way to determine the acceleration based on problem size, but this is not the biggest problem.

Even if the simulation is sometimes faster in parallel the problem is that for different number of threads the calculation gives different results. This is the main reason why the results look like they do. In simulation logs it is observable that for various number of used threads the number of SCC iterations in one step differs. In addition to this, the PME calculation times also differ and are often higher for multiple threads. This leads to slightly different subresults in each step and causes bigger inaccuracy of the whole simulation. Those differences are based on simulated data. Therefore it is not possible to predict performance gain from size of the problem.

To prevent occurrence of those errors only one thread may be used. In this case the only difference between the results of Group and Verlet cut-off schemes is nonbonded calculation where are some differences causing minor changes in results. The difference in results is however not significant.



(c) QMW+8A



(d) QMW+12A

Figure 6.4: Times using Verlet non-bonding cut-off scheme (cont.)

Conclusion

Simulation time of water systems is growing with number of residues in it. The most significant parts of this simulation are diagonalization of matrix and the rest of self-consistent charge loop. Diagonalization is processed by generalized symmetric eigenvalue solver using divide and conquer algorithm, which is part of linear algebra package and its performance is dependent on implementation. The code was non-parallelized, without any further optimizations.

As an alternative to CPU libraries, support of MAGMA, one of GPU accelerated libraries, was implemented. MAGMA uses hybrid algorithm to perform diagonalizotion of matrix. This means that part of code is processed on GPU and other part is processed on CPU simultaneously. Number of used threads and GPUs is adjustable via environmental variables. Code of the original solution was changed to support this library, and the project was prepared for possible implementations of other GPU accelerated libraries.

While simulating small problems, using CPU libraries for diagonalization provides the best results. For the smallest problem even the LAPACK library, which does not supports parallelization or optimizations, provides good time results. For slightly bigger problems it is better to use parallelized libraries like ATLAS or MKL, which provides significantly better results on those problems. The best choice for library is definitely MKL, which is highly optimized library for Intel processors. Its performance is growing with number of used threads. For those, who does not have Intel processor, ATLAS library is available. It is widely supported library that is not limited only to Intel processors. Its performance is not as good as performance of MKL, but is still quite better choice than LAPACK. ATLAS is parallelized, but number of used treads cannot be specified. Number of used threads is determined in compile-time from number of processors available in targeted system.

For simulation of bigger problems, MAGMA starts to provide better performance results in diagonalization part. Number of used GPUs and threads has to take into account hardware and software capabilities, configuration, problem size and other aspects that may affect the performance. It is possible that newer GPU cards with better I/O performance will become better for smaller problems.

Parallelization of code may lead to additional acceleration for big problems. Dividing work between multiple threads should decrease the time needed to process all elements. Number of used threads can be set in environmental variable. Due to the structure of parallelized code, the overhead in synchronization, while simulating small problems, may increase computation time and can even lead to decrease of the final performance. For larger problems the time of synchronization overhead should be negligible compared to the time of parallelized calculation. The boundary where parallelization becomes useful is highly dependent on problem size, thread management and the performance of the system. On slower machines the boundary can be at significantly smaller problems.

Many operations are being done on adjacent elements in an array. Vectorization of these parts of code leads to faster processing of operations on arrays. Those optimizations are for every thread, so combination of vectorized and parallelized code should decrease the time significantly for bigger problems.

Using Verlet non-bonding cut-off scheme instead of Group could improve performance. Verlet cut-off scheme supports CPU acceleration and even GPU acceleration if double precision is not used. Running simulation with Verlet cut-off scheme revealed major problems in parallel run of the simulation. If one thread is used results are almost identical to those provided by Group cut-off scheme, if more threads are used results become to differ and the simulation becomes to bring different results based on number of used threads. In addition to those errors in simulation the time of simulation is often even longer. Using Verlet non-bonding cut-off scheme is therefore not recommended.

Using MAGMA, parallelization and vectorization at the same time should summarize those acceleration effects. Because those improvements are not dependent, combination of them would not affect precision of the result.

Bibliography

[1] Leach, A. *Molecular Modelling: Principles and Applications (2nd Edition)*. Pearson, 2001, ISBN 0582382106. Available from: `https://www.amazon.com/Molecular-Modelling-Principles-Applications-2nd/dp/0582382106`

[2] Jones, J. E. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 106, no. 738, 1924: pp. 463–477, ISSN 0950-1207, doi:10.1098/rspa.1924.0082, `http://rspa.royalsocietypublishing.org/content/106/738/463.full.pdf`. Available from: `http://rspa.royalsocietypublishing.org/content/106/738/463`

[3] Essmann, U.; Perera, L.; et al. A smooth particle mesh Ewald method. *The Journal of Chemical Physics*, volume 103, no. 19, 1995: pp. 8577–8593, doi:10.1063/1.470117, `http://dx.doi.org/10.1063/1.470117`. Available from: `http://dx.doi.org/10.1063/1.470117`

[4] de Souza, O.; Ornstein, R. Effect of periodic box size on aqueous molecular dynamics simulation of a DNA dodecamer with particle-mesh Ewald method. *Biophysical Journal*, volume 72, no. 6, jun 1997: pp. 2395–2397, doi:10.1016/s0006-3495(97)78884-2. Available from: `http://dx.doi.org/10.1016/S0006-3495(97)78884-2`

[5] Birdsall, C.; Langdon, A. *Plasma Physics via Computer Simulation (Series in Plasma Physics)*. CRC Press, 2004, ISBN 0750310251. Available from: `https://www.amazon.com/Plasma-Physics-via-Computer-Simulation/dp/0750310251`

[6] Verlet, L. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, volume

159, Jul 1967: pp. 98–103, doi:10.1103/PhysRev.159.98. Available from: http://link.aps.org/doi/10.1103/PhysRev.159.98

[7] Slater, J. C.; Koster, G. F. Simplified LCAO Method for the Periodic Potential Problem. *Phys. Rev.*, volume 94, Jun 1954: pp. 1498–1524, doi:10.1103/PhysRev.94.1498. Available from: http://link.aps.org/doi/10.1103/PhysRev.94.1498

[8] Porezag, D.; Frauenheim, T.; et al. Construction of tight-binding-like potentials on the basis of density-functional theory: Application to carbon. *Phys. Rev. B*, volume 51, May 1995: pp. 12947–12957, doi:10.1103/PhysRevB.51.12947. Available from: http://link.aps.org/doi/10.1103/PhysRevB.51.12947

[9] Elstner, M.; Porezag, D.; et al. Self-consistent-charge density-functional tight-binding method for simulations of complex materials properties. *Phys. Rev. B*, volume 58, Sep 1998: pp. 7260–7268, doi:10.1103/PhysRevB.58.7260. Available from: http://link.aps.org/doi/10.1103/PhysRevB.58.7260

[10] Elstner, M.; Hobza, P.; et al. Hydrogen bonding and stacking interactions of nucleic acid base pairs: A density-functional-theory based treatment. *The Journal of Chemical Physics*, volume 114, no. 12, mar 2001: pp. 5149–5155, doi:10.1063/1.1329889. Available from: http://dx.doi.org/10.1063/1.1329889

[11] Černý, J.; Kabeláč, M.; et al. Double-Helical → Ladder Structural Transition in the B-DNA is Induced by a Loss of Dispersion Energy. *Journal of the American Chemical Society*, volume 130, no. 47, 2008: pp. 16055–16059, doi:10.1021/ja805428q, http://dx.doi.org/10.1021/ja805428q. Available from: http://dx.doi.org/10.1021/ja805428q

[12] Kubař, T.; Welke, K.; et al. New QM/MM implementation of the DFTB3 method in the gromacs package. *Journal of Computational Chemistry*, volume 36, no. 26, 2015: pp. 1978–1989, ISSN 1096-987X, doi:10.1002/jcc.24029. Available from: http://dx.doi.org/10.1002/jcc.24029

[13] About Gromacs [online]. [cit. 2016-11-25]. Available from: http://www.gromacs.org/About_Gromacs

[14] LAPACK - Linear Algebra PACKage [online]. [cit. 2016-11-25]. Available from: http://www.netlib.org/lapack/

[15] Automatically Tuned Linear Algebra Software (ATLAS) [online]. [cit. 2016-11-25]. Available from: http://math-atlas.sourceforge.net/

[16] Whaley, R. C.; Petitet, A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, volume 35, no. 2, February 2005: pp. 101–121, `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[17] Intel® Math Kernel Library (Intel® MKL) [online]. [cit. 2016-11-25]. Available from: `https://software.intel.com/en-us/intel-mkl`

[18] Innovative Computing Laboratory, University of Tennessee Department of Electrical Engineering and Computer Science. *MAGMA Users' Guide.* [online][cit. 2016-11-26]. Available from: `http://icl.cs.utk.edu/projectsfiles/magma/doxygen/`

[19] Tomov, S.; Dongarra, J.; et al. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, volume 36, no. 5-6, June 2010: pp. 232–240, ISSN 0167-8191, doi:10.1016/j.parco.2009.12.005.

[20] OpenMP Architecture Review Board. *OpenMP Application Programming Interface.* [online][cit. 2016-11-30]. Available from: `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`

Appendix A

Acronyms

DFTB Density Functional-based Tight Binding

MM Molecular mechanics

QM Quantum mechanics

TB Tight binding

SCC Self-consistent charge

PME Particle Mesh Ewald

PBC Periodic boundary conditions

LCAO Linear combination of atomic orbitals

DFT Density functional theory

KS Kohn-Sham

BLAS Basic linear algebra subprograms

LAPACK Linear algebra package

MKL Math kernel library

ATLAS Atomatically tuned linear algebra system

Appendix B

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
lib .......................... the directory of used software and libraries
src ........................................ the directory of source codes
  implementation ....................... implementation source codes
  tex ...... the directory of LaTeX source codes and images of the thesis
text ........................................ the thesis text directory
  DP_Pekař_Jakub_2017.pdf ............ the thesis text in PDF format
```