Analysis of Open Source Software for Lattice Quantum Chromo Dynamics

A comparison between the Chroma software system and the Domain-Decomposition-Hybrid-Monte-Carlo software

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom Physiker
(Dipl.-Phys.)

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Humboldt-Universität zu Berlin

von
Jens Grieger
geboren am 26.05.1980 in Darmstadt

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Dr. h.c. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät I:
Prof. Dr. Christian Limberg

Gutachter:

1. Prof. Dr. Ulrich Wolff
2. Dr. Hubert Simma

eingereicht am: 07. Juni 2007
Tag der mündlichen Prüfung: 18. Juli 2007

Abstract

This thesis analyses two Open Source software projects for Lattice Quantum Chromo Dynamics (LQCD), in fact the Chroma software system and the Domain Decomposition Hybrid Monte-Carlo (DD-HMC) software. The goal was to obtain information about the usage of the software and its libraries from the point of view of a new user.

Chroma is a comprehensive software project which can be used for several applications for LQCD like simulation and measurement with different algorithms. To use the software it is sufficient to learn how to deal with XML input files whereas the understanding of the source code in detail can be a difficult task. The correctness of the simulation and measurement with Chroma could be proved for a sample calculation. For it, the effective mass of a pion was determined for a certain set of parameters. This result was compared with other computations [1].

DD-HMC is a software for the generation of gauge configurations by means of the Hybrid Monte-Carlo (HMC) algorithm used with domain decomposition methods. Methods for measurements are not provided by the software. Therefore, the possibilities of DD-HMC are lesser in comparison to the Chroma software system. Thus, the source code of DD-HMC is not as comprehensive as the code of the Chroma and is fairly understandable.

To analyse the efficiency of the Chroma software system the application of the Wilson-Dirac operator was benchmarked. Since it is the most computationally intensive task of a simulation with dynamical fermions, the execution time of the whole simulation can depend on the efficiency of this application. For the analysis a theoretical model was discussed for single processors as well as for parallel machines.

Keywords:
Lattice QCD, Open source software, Chroma software system, DD-HMC software

Zusammenfassung

Die vorliegende Arbeit untersucht zwei “Open Source” Projekte für die Gitter-Quanten-Chromo-Dynamik (Gitter-QCD), und zwar das “Chroma Software Projekt” und die “Domain-Decomposition-Hybrid-Monte-Carlo (DD-HMC) Software”. Ziel war, Informationen für die Anwendung der Programme aus der Sicht eines neuen Benutzers zu erhalten.

Chroma kann für unterschiedliche Anwendungen der Gitter-QCD benutzt werden und ist somit sehr umfangreich. Sowohl Simulationen als auch Messungen können mit verschiedene Algorithmen durchgeführt werden. Um die Software zu benutzen, reicht es aus, den Umgang mit XML-Dateien zur Programmsteuerung zu erlernen. Das Verständnis von Details des Quellcodes kann jedoch eine umfassende Aufgabe sein. Um die Richtigkeit der Simulation und Messung von Chroma zu prüfen, wurde die effektive Pionmasse für einen bestimmten Parametersatz ermittelt. Das Ergebnis konnte mit anderen Messungen verglichen werden [1].

Die Funktionalität der DD-HMC Software beschränkt sich auf die Produktion von Eichfeldkonfigurationen. Dafür wird der Hybrid-Monte-Carlo (HMC) Algorithmus mit der Methode der “Domain-Decomposition” benutzt. Das DD-HMC Programm bietet im Vergleich zu Chroma weniger Möglichkeiten. Somit ist der Quellcode nicht so umfangreich und weitestgehend verständlich.

Um die Effizienz von Chroma zu analysieren, wurde die Laufzeit des Wilson-Dirac Operators getestet. In einer Simulation mit dynamischen Fermionen ist diese Anwendung die rechenaufwendigste. Daher hängt die Laufzeit der ganzen Simulation von dieser Anwendung ab. Zur Analyse wurde ein theoretisches Modell, sowohl für die Benutzung eines Prozessors als auch für die Benutzung eines Computers mit mehreren Prozessoren, diskutiert.

Schlagwörter:
Gitter-QCD, Open Source Software, Chroma Software Projekt, DD-HMC Software

Danksagung

Ich danke allen Mitgliedern der Arbeitsgruppe Computational Physics für ihre Hilfsbereitschaft. Insbesondere danke ich Herrn Prof. Wolff und Hubert Simma für die Betreuung. Ich danke Burkhard Bunk für die Hilfe bei Computerproblemen aller Art.

Des Weiteren danke ich Aiko, Julia und Max für die Hilfe bei der Fertigstellung dieser Arbeit, sowie Martin, Marko, Niko, Alex, Andreas und Hai für spannende Gesprächsrunden beim Mittagessen.

Ich möchte besonders meinen Eltern, meinen Großeltern und meiner Schwester für ihre Unterstützung danken. Des Weiteren danke ich meinen Mitbewohnerinnen und Mitbewohnern, sowie Daniel und Celine für Hilfsbereitschaft und nötige Ablenkung.

Ich danke allen Personen, die in jeglicher Weise an dem Arbeitsprozess beteiligt waren, jedoch namentlich nicht erwähnt wurden.

Contents

1 Introduction **1**

2 Description of the Chroma software system **5**
2.1 Software modules 5
2.1.1 QCD Message Passing API (QMP) 8
2.1.2 QCD Linear Algebra (QLA) 9
2.1.3 QCD Input/Output API (QIO) 9
2.1.4 QCD Data Parallel API (QDP) 10
2.1.5 Chroma 13
2.2 Installation 14
2.2.1 Scalar build 16
2.2.2 Parallel Build 19
2.3 "The HackLatt 2006 Chroma Tutorial" 21
2.3.1 First measurements with the "black box" chroma . . 21
2.3.2 Using the library of Chroma to write Programs 26
2.4 Simulations and measurements with Chroma 31
2.4.1 Dynamical fermions with Hybrid Monte-Carlo 31
2.4.2 Pure gauge simulations with the Heat Bath Algorithm 35
2.4.3 Simulations and measurements at an Opteron PC Cluster 35
2.4.4 Binary storage of gauge configurations 38
2.5 Summary of the experiences with the Chroma software system 39

3 Description of the Domain Decomposition Hybrid Monte-Carlo (DD-HMC) software **41**
3.1 Overview of the software 41
3.2 Structure of the directories 42
3.3 Installation 43
3.4 First steps 43
3.5 Description of the Schwarz-preconditioned Hybrid Monte-Carlo algorithm 44
3.5.1 Factorisation of the quark determinant 44
3.5.2 The Schwarz-preconditioned HMC algorithm 45

3.5.3 Solution of the Dirac equation on the full lattice . . . 46
3.5.4 Multiple step size integration 49
3.5.5 Partial block decoupling with active links 50
3.5.6 Description of one update cycle 50
3.6 Simulation with DD-HMC 51
3.7 Binary storage of gauge configurations 53
3.8 Summary of the experiences with the DD-HMC simulation software 53

4 Benchmarks 55
4.1 Opteron PC-Cluster 55
4.2 Comparison of different compiler flags 55
4.3 Analysis of implementations of the Wilson-Dirac operator . 57
4.3.1 Introduction 57
4.3.2 Definitions 59
4.3.3 Performance and execution time 59
4.3.4 Single node implementation 62
4.3.5 Time measurement of the Wilson-Dirac operator on a single node 64
4.3.6 Multi node implementation 66

5 Effective Mass 72
5.1 Timeslice correlations of pions 72
5.2 Extracting masses from timeslice correlations 73
5.3 Numerical analysis 74

6 Conclusion 76

A Conventions 79
A.1 Gauge Fields 79
A.2 Wilson-Dirac operator 79
A.3 Representation of the Dirac matrices and the possible advantages 80
A.4 Effectice Action 82
A.5 Hybrid Monte Carlo Algorithm 83
A.5.1 HMC Hamiltonian and Hamilton's Equations of Motion 83
A.5.2 Leap Frog Integration 84
A.5.3 Metropolis Algorithm 84

B Parameters for XML input files for the Chroma software system 85
B.1 General posibilities for "InlineMeasurements" 85
B.1.1 Possible sources for "MAKE_SOURCE" 87

B.1.2 Possible wavefunctions for sink smearing for "SPECTRUM" 87
B.2 Possibilities for main program "hmc" 87

List of Figures

2.1 SciDAC software module architecture. Higher level modules have to be built on the top of the lower ones. Three levels of basic software modules and the applications for LQCD are shown. 6
2.2 Software map of required modules to build the Chroma software system. 7

3.1 Two-dimensional plain of the lattice covered by non-overlapping blocks Λ (Σ). The union of the black and white blocks are given by Ω (Y) and Ω^* (Y^*), respectively, whereas their exterior boundaries (open circles) are given by $\partial\Omega$ (∂Y) and $\partial\Omega^*$ (∂Y^*). 46

4.1 Comparison of the compiler optimisation flag. The triangles are for the optimisation level 2, the crosses for level 3 and the dots have no optimisation. 57
4.2 Comparison of different SSE usages. The crosses show the inline assembly written optimisation, the dots show the attempt of the SSE optimisation by compiler flags and the triangles show both optimisations 58
4.3 Execution time of the measurement and the model for $S_0 = 0$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s. 66
4.4 Execution time of the measurement and the model for $S_0 = 12$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s. 67
4.5 Execution time of the measurement and the model for $S_0 = 24$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s. 67
4.6 Execution time of the Wilson-Dirac operator for $V = 1024$ on multiple nodes. The bandwidth of the network $\beta_{\bar{P}P} = 0.405$ Gbyte/s is a lower bound. The triangles refer to the measured time, whereas the crosses show the execution time on a single processor with the corresponding local lattice. The model described in section 4.3.5 was used ($S_0 = 12$). The plus indicates the time that corresponds to the data exchange between the different processors. 70

4.7 Execution time of the Wilson-Dirac operator for $V = 2048$ on multiple nodes. The bandwidth of the network $\beta_{\bar{P}P} = 0.405$ Gbyte/s is a lower bound. The triangles refer to the measured time, whereas the crosses show the execution time on a single processor with the corresponding local lattice. The model described in section 4.3.5 was used ($S_0 = 12$). The plus indicates the time that corresponds to the data exchange between the different processors. 71

5.1 Effective mass of the pion for $\beta = 5.6, \kappa = 0.1575$ and $L_X = L_Y = L_Z = 12, L_T = 16$. The dotted line is to guide the eye, whereas the dashed line is the value for the pion mass for a lattice with $L_X = L_Y = L_Z = 12, L_T = 32$ [1]. 75

List of Tables

3.1 Pseudo code for the solution of the Dirac equation with a generalised conjugate residual solver preconditioned with the Schwarz procedure. 48

4.1 Global volumes V for the application of the Wilson-Dirac operator on different numbers of processors. The dimensions of the global lattice are given by L_X, L_Y, L_Z, L_T whereas the dimensions of the local lattices are given by l_X, l_Y, l_Z, l_T. . . . 69

5.1 Result of the measurement of the effective pion mass $m_\pi^{\mathrm{eff}}\,(t, t+1, T)$. The value and its autocorrelation time are shown with the errors of the analysis. 75

Chapter 1

Introduction

Most of the matter around us is made of protons and neutrons. These particles represent the nuclei of atoms bounded by the strong force or strong interaction. Nucleons are hadrons, i.e. bound states of quarks which are elementary particles. The strong force interacts between quarks, mediated by gluons. The masses of the three valence quarks in the nucleon are very small in comparison to the mass of the nucleon itself, i.e. most of the mass of the matter around us is a consequence of the strong interaction.

The generally accepted theory of the strong interaction is Quantum Chromo Dynamics (QCD) which forms the standard model of elementary particle physics beside the electromagnetic and the weak interaction which is combined in the Glashow-Weinberg-Salam (GSW) theory of electroweak interaction. Both the GSW theory and QCD are renormalizable quantum field theories with the principle of local gauge invariance. The QCD Lagrangian has only a few parameters but it is difficult to compute masses and other characteristics of light hadrons, like the nucleon, from first principles. Characteristics of hadrons are known from experiments.

The gauge group $SU(3)$ of QCD is non-abelian in contrast to the gauge group $U(1)$ of Quantum Electro Dynamics (QED). Therefore, the gluon can interact with itself unlike the photon in QED. For the analysis of QED, perturbation theory is very successful. This is only possible for a small coupling constant. The strong coupling α_s becomes quite large for low energies. Therefore, it is not possible to use perturbation theory for the analysis of the characteristics of light hadrons, like nucleons. A non-perturbative approach to the theory is the numerical simulation of Lattice QCD (LQCD).

To formulate QCD on a discrete lattice the classical theory is quantised by means of Feynman's path integral mechanism. A common discretisation of the fermionic part of the Langrangian was formulated by Wilson and is described in appendix A.2. Expectation values of observables can be obtained by integrating over all possible "paths" and "configurations", respectively. Each measure is weighted by an analogy to the Boltzmann

factor in statistical mechanics. This analogy becomes apparent if the time variable is continued to imaginary values. Then, the weight becomes real. It is possible to use Monte-Carlo methods to produce a set of importance sampled gauge configurations. Therefore, the Hybrid Monte-Carlo (HMC) algorithm is widely used which is introduced in appendix A.5. During the execution of this algorithm a huge Dirac matrix has to be inverted several times which is the most computationally intensive task of the simulation. Thus, the numerical simulation of LQCD is a high performance computing application which needs a high capacity of computing power. This can be achieved by supercomputers with a typical peak performance of several Teraflops[1]. For an efficient use of these massive parallel systems highly optimised software is necessary. The simulation of LQCD is done on a global lattice with volume V. This lattice is split into local ones on a parallel machine with N_P processors. Therefore, it is important that the software controls the communication without a big overhead. Furthermore, computations on each processor have to be done in an efficient way. Beside the used algorithm, the way of its implementation has an important role.

The performance of software can be analysed by means of benchmarks. Those have to be done on a single processor as well as on a parallel machine. In this way, it is possible to get information about the efficiency of the computations on the local lattice and of the communication between the different processors. For this, the execution time of a certain task of the code is measured. These tasks can have a completely different complexity, i.e. it is possible to analyse the whole program or just a part of it. In LQCD software, such a part could be a solver for the Dirac equation, the application of the Dirac operator or just linear algebra computations. The inversion of the huge Dirac matrix which is necessary for the HMC algorithm is done by iterative solvers, i.e. the application of the Dirac operator to a spinor is done several times. Most of the execution time for the whole program is needed for this repeated application. Thus, the benchmark of the Dirac operator gives important information for the performance of the whole code. The execution time of an arbitrary task of a program is determined by different characteristics of the hardware, such as the clock frequency of the processor and the bandwidth between the processor and the memory (RAM). Furthermore, when dealing with floating point numbers, the ratio of multiplications and additions of a task can be important if the processor does both of these operations in parallel, as common processors do, i.e. when applying just additions or just multiplications it is not possible to obtain more than 50 % of peak performance. Therefore, each application has a theoretical execution time. The determination of it and the comparison with the corresponding benchmark can give estimate of the

[1] Flop means floating point operations per second, whereas a floating point operation is a multiplication or addition of floating point numbers.

possibilities to improve the implementation and thus, to minimise the execution time of a given algorithm.

The development of software for LQCD applications can be a time-consuming task. Therefore, it can be helpful to use Open Source software projects. This software is freely available, i.e. everybody can do enhancements of different tasks of the project. Therefore, it is not necessary to reproduce existing code which has proven to be useful. Even if the improvement of several tasks are wanted it can be helpful to build it up on available basics of a software project. The number of potential developers of such a project can be huge. That can lead to fast enhancements of the software. Furthermore, bugs in the software are normally fixed after a short while, because each user has the chance to detect as well as to fix them. Of course, useful software is a must for numerical researches. Free available software is a division of work that gives a community the potential to get results in a faster way. In general, software has to be documented. At least it must be possible to get the necessary information to use it. Open Source software does only make sense if the sources are understandable, too. Therefore, a useful documentation of the code is necessary. It is not realizable that each user of Open Source software can stay in close contact to the developers of it. Therefore, it is important to have good documentations and comprehensive tests of the software as well as field reports of it.

In this thesis two Open Source projects were analysed, namely the Chroma software system and the Domain Decomposition Hybrid Monte Carlo software. The goal was to get information about the possibilities of the software and its libraries. Both the usage of the software and the possibilities to extend certain tasks should be analysed from the point of view of a new user of the software.

The used conventions in this thesis can be found in appendix A.

Outline of the thesis

Description of the Chroma software system

In this chapter the Chroma software system developed by the USQCD Collaboration is presented. This comprehensive software package is build of different software modules which are described in the first subsection. Then, an introduction for the installation on a single node as well as on a parallel machine is given. After the description of the first steps with the Chroma software system, the way to deal with simulations and measurements is presented.

Description of the Domain Decomposition Hybrid Monte-Carlo (DD-HMC) software

The DD-HMC software is a LQCD simulation software which uses the HMC algorithm by means of domain decomposition ideas. It is developed by Martin Lüscher. As this work was started, this software was not published yet. Therefore, the experiences are not so extensive in comparison to the Chroma software system. After an introduction for the installation, a short description of the algorithm is given in this chapter. At the end the necessary set of parameters for a simulation is explained.

Benchmarks

This chapter deals with the performance test of the Wilson-Dirac operator of the Chroma software system. After the description of the hardware of the used Opteron PC-Cluster, first tests with different compiler flags are shown. Then, a model for the theoretical execution time of the application is discussed and compared with measurements on a single node as well as on several processors of the Opteron Cluster.

Effective mass

In this chapter the determination of hadron masses from LQCD simulations is introduced. After a short discussion of timeslice correlation functions, the way to obtain the effective mass of a pion is shown. The goal of this chapter was not to determine the pion mass in a high precision and in physical ranges. The idea was to do a sample calculation and to compare the value of the effective mass for a certain set of parameters with other measurements to check the correctness of the simulation and measurement with used software on the Opteron PC-Cluster.

Chapter 2

Description of the Chroma software system

2.1 Software modules

The software modules described below are developed by the USQCD Collaboration founded by the Department of Energy through the "Scientific Discovery through Advanced Computation" (SciDAC) program, and through the U.S. Department of Energy Office of Science's High Energy Physics and Nuclear Physics programs.

The Chroma software system [2] has been designed to be highly portable. It is possible to build this application on several different architectures, e.g. linux scalar workstations, linux clusters, supercomputers like the QCDOC, IBM P-series computers and BlueGene/L machines. To achieve this portability of a high performed code different layers of software modules were created (figure 2.1). In this way it is possible to adapt certain parts of the low level code to the specific architecture whereas other parts of the upper levels stay unchanged.

One can split the software modules into several layers. The applications for Lattice QCD (LQCD) computations like simulation and measurement have to be build on the top of different software modules. Three layers of this software modules exist. For applications like Chroma, modules of the first and the second level are necessary. The third level modules are not necessary but useful like the Wilson-Dirac operator implementation for SSE.

The first level contains the LQCD Message Passing (QMP) Application Programming Interface (API) and the LQCD Linear Algebra API (QLA). QMP arranges the communication between the processors on parallel machines whereas QLA provides an interface for linear algebra routines.

The second level consists of the LQCD Input/Output API (QIO) and the LQCD Data Parallel API (QDP/C respectively QDP++). QIO arranges

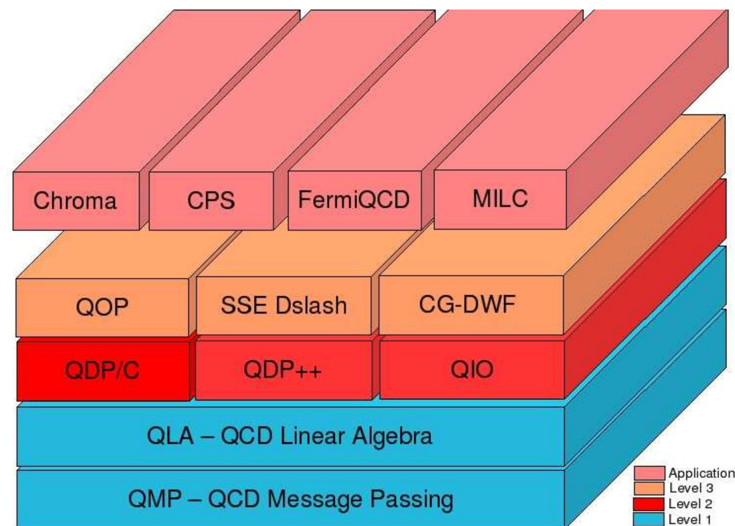

Figure 2.1: SciDAC software module architecture. Higher level modules have to be built on the top of the lower ones. Three levels of basic software modules and the applications for LQCD are shown.

the input and output for lattice data on single node systems as well as on parallel machines using QMP. The SciDAC LIME implementation is also part of this module.

The main features of QDP are the handling of different data types, e.g. data which is only defined on one site of the lattice and lattice-wide data types. Therefore, the creation of the layout of a lattice of any size and any dimensions are tasks of this API, too. Furthermore the modules QMP, QLA and QIO can be used by this API when building it. Therefore, on parallel systems QDP uses the QMP library for the handling of parallel operations. QDP/C is the C implementation whereas QDP/C++ is the C++ implementation of QDP.

The modules of the third level consist of several optimisation packages, e.g. the SSE optimised Wilson-Dirac operator (SSE Wilson Dslash). To build the applications the level three modules are not necessary. However the SSE optimised Wilson-Dirac operator is very useful to build efficient code like shown in chapter 4.

The last layer of software packages contains four applications used by USQCD, the Chroma code, the Columbia Physics System (CPS), FermiQCD and the MIMD Lattice Collaboration (MILC) code. The present work covers only one of these applications, the Chroma code.

All software is available from the USQCD Software Release page [3]. Manuals for the level one and level two modules can be found either on the corresponding web pages [4–7] or are obtained when downloading

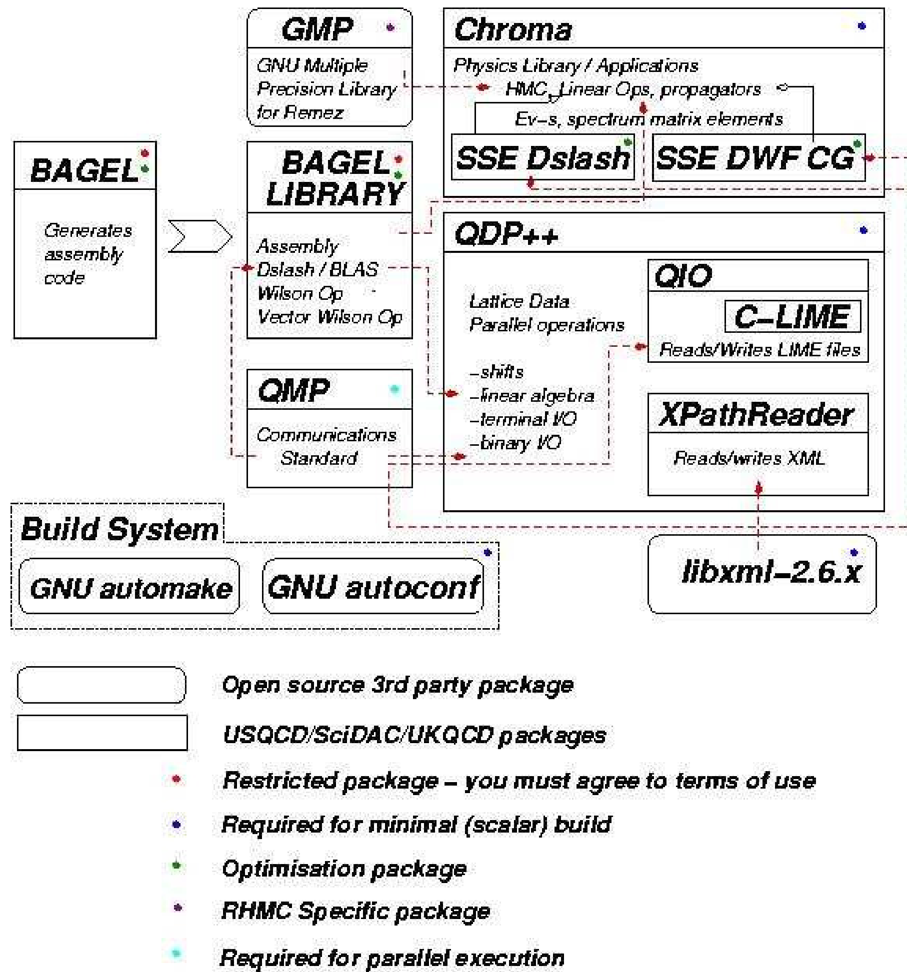

Figure 2.2: Software map of required modules to build the Chroma software system.

the sources. A short overview is presented in the following subsections. The main informations about the application Chroma can be found on the "HackLatt 2006 Chroma Tutorial" web page [8] where some talks are presented of the according workshop, a documentation how to build the simplest version of the Chroma software system [9] and the Chroma tutorial that starts with general ideas of input files and ends with writing own programs for measurements. The introduction for a scalar build as well as for the parallel usage of the software system can be found in chapter 2.2. A summary of the tutorial with additional impressions is given in chapter 2.3. To get further informations of the structure, the implementation and the usage of the Chroma software system one is forced to browse a lot in the source code. To do so, the source code documentation generator tool Doxygen [10] can be used for QDP++ [11] and Chroma [12].

2.1.1 QCD Message Passing API (QMP)

On massive parallel systems one has to handle communication between different processing nodes. An efficient code has to organise communication without a significant overhead. There are different possibilities to connect the computing nodes of supercomputers. Either all nodes are connected with a central switch that controls the communication (star cluster) or the nodes have the possibility to communicate with each other without a central controller (mesh cluster). The communication in a mesh cluster takes place between adjacent nodes sometimes in a N-dimensional torus (periodic boundary conditions). The QMP API is optimised for this style of communications.

The idea of QMP is to provide a flexible API for all LQCD applications and an efficient possibility to use massive parallel systems without platform specific message passing routines. Computer clusters mostly use the Message Passing Interface (MPI) whereas other architectures, e.g. the QCD On a Chip (QCDOC) custom machine, use other standards which are more efficient on such large platforms (small problem size per node). All applications on the top of QMP should be portable without a relevant overhead.

There are three implementations of QMP:

- QMP-MPI that is implemented on the top of MPI and is tested above
  - MPICH-GM (MPI over Myrinet),
  - MPICH-SM (shared memory),
  - MPICH-P4 (MPI over ethernet),
- QMP-QCDOC for the custom QCDOC machine and
- QMP-MVIA for Gigabit ethernet mesh clusters.

In the present work the QMP-MPI implementation with MVAPICH (MPI over infiniband) was used without any problems.

2.1.2 QCD Linear Algebra (QLA)

The QLA API is a first level software module. The data that is handled by the interface are data primitives such as SU(N) matrices and vectors on single nodes. It provides functions for several operations on these data. Furthermore a set of elementary functions such as, e.g sin, exp or cosh, are given. Beside multiplication and addition of data further operations like building traces of matrices or reductions like computing the norm or calculating global sums are provided. Operations to fill datatypes with numbers (random numbers or zeros for instance) are also provided.

Like other software modules QLA can be used independently. For the QCD Data Parallel API (QDP) it is the basis for the linear algebra calculations. QLA is written in C. The C++ version of QDP called QDP++ has its own linear algebra functionality but with respect to QLA.

2.1.3 QCD Input/Output API (QIO)

The input/output (I/O) API QIO was created to support the level two software module called QCD Data Parallel API (QDP), but it also works independly of QDP. On parallel machines it is built on the top of QMP which handles communication. On the one hand it deals with global data that is constant across the whole lattice, on the other hand with data consisting of lattice fields that have the same format but other values on each lattice site e.g. gauge fields. On parallel machines, there are three different file volumes for the I/O: single-file, partition-file and multiple-file volumes. Single files are read and written by one I/O node, multiple files are read and written by every node and partition files are read and written by a specified set of I/O nodes. Data in single-file volumes is contained in one file whereas data is split into separate files when using multiple or partion file volumes. Then, each file gets a unique name by attaching the extension `.vol`*nnnn* where *nnnn* is the number of the I/O node with leading zeros.

For the input and output of binary data QIO uses the LIME file format [13]. LIME can stand for “Lattice QCD Interchange Message Encapsulation” or more generally for “Large Internet Message Encapsulation”. In these files both, binary records and ASCII metadata can be included. For the metadata, Extensible Markup Language (XML) is used. QDP++ that contains the QIO API uses XML for the handling with input and output files, too. More information about this format concerning the I/O with XML will be presented in section 2.3.1.

Lattice QCD Interchange Message Encapsulation (LIME)

LIME is a packaging scheme for combining records containing ASCII and/or binary data. One can compare it with the UNIX "cpio" or "tar" formats. A LIME file contains one or more LIME messages and each message consists of one or more LIME records. For example, a LIME message can contain a binary record like a gauge configuration and the corresponding ASCII metadata like lattice size etc. The LIME software package consists of a C-language API for manipulating LIME files and a set of utilities for examining, packing and unpacking LIME files.

If a LIME file contains a binary configuration, indices are ordered according to the ILDG [14] standard. A SU(3) gauge configuration consists of several SU(3) matrices. This set of matrices regarding the 8 dimensional array of floating point numbers is stored in the following way (first index is the slowest one):

1. Site index in time-direction t $(t = 0, \ldots, L_T - 1)$
2. Site index in space-direction z $(z = 0, \ldots, L_Z - 1)$
3. Site index in space-direction y $(y = 0, \ldots, L_Y - 1)$
4. Site index in space-direction x $(x = 0, \ldots, L_X - 1)$
5. Direction index μ $(\mu = 0, \ldots, 3; \quad 0 \to x, 1 \to y, 2 \to z, 3 \to t)$
6. Colour index a $(a = 0, 1, 2)$
7. Colour index b $(b = 0, 1, 2)$
8. Index corresponding to real or imaginary part $(0 \to \mathrm{Re}, 1 \to \mathrm{Im})$

In the notation of C, this array can be written in the following way:

```
double U[T][Z][Y][X][NDIM][NCOLOUR][NCOLOUR][2] .
```

Intel x86 processors (and their clones) use the little-endian format that means increasing numeric significance with increasing memory addresses. The big-endian format used by ILDG files is its opposite.

2.1.4 QCD Data Parallel API (QDP)

There are some differences between the C version (QDP/C) and the C++ version (QDP++) of QDP. These differences become less apparent in the possibilities of the QDP functions than in the general structure of the modules. QDP++ comprehends the packages QLA and QIO. If one wants to use

QDP/C with QLA and QIO, one has to build this package on the top of the wanted modules. Furthermore QDP++ uses some functionalities of C++ like object oriented programming, overloading operators and functions as well as templates.

The Chroma software system only works with QDP++. Therefore, this work does not cover QDP/C in detail.

Datatypes in QDP++

With respect to programming the structure of datatypes is somewhat different in QDP/C and QDP++. In the C++ API possibilities of object oriented programming are used as well as aspects of polymorphic programming that is, for instance, overloading operators and functions. The names of the datatype definitions in both API's are similar. The linear algebra part of QDP is produced to amount the functionality of QLA. Because this work does not deal with the C implementation of QDP, we want to focus on the description of the datatypes of QDP++. Lattice fields are defined on all sites of the N_d dimensional lattice. Datatypes on single lattice sites are called data primitives, i.e. lattice fields consist of primitive data over all sites. Assuming N_C colours and N_s dimensions in spin vector space these primitives have the tensor structure

$$Lattice \otimes Colour \otimes Spin \otimes Complexity.$$

In the space of colour and spin the possible variables to construct data types are matrices and vectors. Furthermore it is possible to distinguish between lattice-wide data and data that does not depend on the sites of the lattice. Variables that have no imaginary part are so called scalar in the space of complexity. The following table shows an example of possible datatypes:

	Lattice	*Colour*	*Spin*	*Complexity*
Scalar:	`Scalar`	`Scalar`	`Scalar`	`Scalar`
Gauge fields:	`Lattice`	`Matrix(Nc)`	`Scalar`	`Complex`
Fermions:	`Lattice`	`Vector(Nc)`	`Vector(Ns)`	`Complex`
Propagator:	`Lattice`	`Matrix(Nc)`	`Matrix(Ns)`	`Complex`
Gamma:	`Scalar`	`Scalar`	`Matrix(Ns)`	`Complex`

To organise these structure QDP++ differs between four and five types respectively:

- Word type: basic machine types like `int`, `float`, `double` and so on.
- Reality type: complex or scalar.
- Primitive type: scalar, vector, matrix.

- Inner grid type: scalar or lattice. This type is for vector like architectures.
- Outer grid type: scalar or lattice. This type is used for scalar like machines. In combination with the inner grid type it should be possible to use a super-scalar machine with short vector instructions.

The inner grid type was not tested in this thesis. For the definition of data types in QDP++ the template structure is used to combine these five subtypes:

```
OuterGrid<Primitive<Primitive<Reality<Word>>>>
```

Some classes need further parameters when defining a data type. For example, a gauge field needs the number of colours:

```
typedef OLattice<PScalar<PColorMatrix<RComplex<REAL>\
,Nc>>> LatticeColorMatrix
```

The name of a type without a further extension implies that the default values are used for N_d, N_C and N_s that are defined at compile time. It is possible to choose other values for the data types obeying the following convention: `datatyp`*PC*, where *P* stands for the precision and *C* for the number of colours. For single precision *P* gets the value `F`, for double precision the value `D`. The number of colours *C* can get an arbitrary integer. For example a gauge field in single precision with two colours becomes

```
LatticeColorMatrixF2,
```

whereas a gaugefield in double precision with three colours becomes

```
LatticeColorMatrixD3.
```

Further informations can be found in the following header files that are located in the `include` directory of QDP++:

- typedefinitions for scalar like machines (using outer grid types) - `qdp_scalarsite_defs.h`,
- the location of the template classes `OLattice` and `OScalar` - `qdp_outer.h`,
- the primitive scalar - `qdp_primscalar.h`,
- matrix `qdp_primmatrix.h`,

- and vector type template classes - `qdp_primvector.h`,
- the reality type template classes with their overladed functions and operators - `qdp_reality.h`,
- the word type template classes and their operators - `qdp_word.h`.

Layout creation with QDP++

One of the basic routines of QDP++ is called `Layout::create()`. On a parallel machine each node has to call this function that is fundamental for lattice wide operations. The dimensions of the lattice must be defined before creating the layout but need to be defined at compile time. It is possible to define three different orders of the storage of the lattice data before compiling QDP++. These orders are called lexicographic, checkerboard with two colours (even/odd) or hypercubic checkerboard called (cb32). The even/odd ordering is the default one. For some computations it will be advantageous to operate on subsets of the lattice, e.g. on a checkerboard subset. Two different classes of subsets exist namely `OrderedSubset` and `UnorderedSubset`. It is only possible to have one `OrderedSubset` because the sites of this subset have to be continuous. This one is determined by the storage order while compile time. `UnorderedSubset` can be defined by the user as much as wanted.

2.1.5 Chroma

The application Chroma is able to do LQCD simulations with Wilson, Domain Wall and Overlap fermions. Several inverters and integration schemes are provided for this task. A staggered fermion library exists, too. Furthermore a lot of measurements are possible such as measuring spectroscopies, decay constants, nucleon form factors and structure function moments. The computation of hadronic 2-point and 3-point functions is also included.

For this tasks several main programs are provided. This work covers predominantly two of these executables namely `chroma` and `hmc`. The first one deals with several measurements. The wanted tasks are chosen by input files. The structure of these files is explained in chapter 2.3.1. To get an idea of possible measurement tasks, have a look at the list of the keywords for the input files in appendix B.1. The executable `hmc` is the main simulation program for the production of gauge configurations with Hybrid Monte-Carlo algorithms. This program can be completely controlled with XML input files. The reader will find more details in chapter 2.4. A list of possible keywords is shown in appendix B.2.

Furthermore, a SSE optimised Wilson-Dirac operator is provided for this application. This operator needs the even/odd storage ordering of lattice data that can be defined when compiling QDP++.

2.2 Installation

In this section the necessary steps to build the simplest version of the Chroma software system are described first. Then, the steps for a parallel build using QMP are shown.

For building the simplest version of the Chroma software system some preparations have to be done. The dependencies of all software modules are shown in figure 2.2. For the build of the Chroma software system the following Open Source packages are needed:

- GNU C/C++ compiler (`gcc/g++` version 3.x),
- automake-1.9.3,
- autoconf-2.58,
- libxml2.

By default it is not possible to compile the sources with other compilers. When building the software modules one has to execute the provided `configure` file with specific options. This generates a `Makefile` by means of makefile templates. It is possible to generate makefiles without `automake` and `autoconf`, but in practice errors could occur if the makefiles and makefile templates have other timestamps than the important `Makefile.am` files. Subsequently, the dependency tracking of the build system will try to run `automake` and `autoconf`.

`libxml2` is the Gnome foundations open source XML parsing package used by QDP++ and Chroma to read XML files. Here it is important to have the libraries of `libxml2` and not just the applications like `xmllint`, `xmlcatalog` etc.

The first three of these packages are freely available from the GNU Operating System web-site [15]. `automake` and `autoconf` in the correct version are also available on [9].

It is possible to get `libxml2` from its web page [16].

When using a linux system these open source packages should also be available by means of the package managers of the linux distribution.

This preparation is necessary to build the SciDAC software modules needed by the Chroma application. The general concept for building any package of the software map (fig. 2.1 respectively fig. 2.2) follows the general open source standards. The basic instructions are:

- `configure`,

- make[1],
- make install.

and have to be executed in this specific order. To specify the software one can use several options when executing configure. A short list of these options can be seen by typing configure --help. Either these options can be used to specify certain other packages, e.g. when building one software module on the top of another one (--with-qdp=/home/chroma-user/install/qdp), or one wants to enable or disable some features of the packages, e.g. enabling the SSE optimisation (--enable-sse2). Additionally the compiler flags and the compilers themselves are options for the configure script (e.g. CXX=gcc or CXXFLAGS="-O2").

Three directories play a role when building any software module:

- the source directory,
- the build directory,
- the install directory.

The first one contains the untarred files of the package, i.e. the sources, the configure script, makefile templates, README files etc. The makefile generated by the configure script will be in the build directory. Therefore, make and later make install have to be executed in this directory. The compiled code can be found in the build directory after executing make. The configure script is called from the build directory. That is the way to define this directory. It can be identical with the source directory. The install directory contains the installed applications and libraries. It is chosen by the configure script command --prefix=<install-dir>. If no --prefix argument is given, the default install directory is /usr/local.

For the examples described in the next section the following structure of directories is assumed. Every package consists of three different directories:

- <PACKAGE>_SRC=/home/chroma-user/src/<package>
- <PACKAGE>_BUILD=/home/chroma-user/build/<package>
- <PACKAGE>_INSTALL=/home/chroma-user/install/ \
  <package>

[1]It is recommended to use 'GNU Make' which is also known as gmake because automake generates makefiles that are compatible with the 'GNU Makefile standards'. In the installation notes for the Chroma software system it is mentioned that one needs 'GNU Make'. In the present work 'GNU Make' was the only tested make tool.

Furthermore, it is assumed that all sources are untarred and located in the corresponding source directory.

The several files in the different directories can be categorised by their filename extensions:

- Header files with extension `.h`,
- C Sources with extension `.c`,
- C++ Sources with extension `.cc`,
- Object files with extension `.o`,
- Libraries with extension `.a`,
- Executables without extension.

2.2.1 Scalar build

Building QDP++

The simplest version of the Chroma software system consists of the level two module QDP++ and the application Chroma. Assuming that the sources are located in the accordant directory one has to create at least another one:

```
mkdir /home/chroma-user/build/qdp++_single_scalar .
```

Change the directory to run the configure script:

```
cd /home/chroma-user/build/qdp++_single_scalar
../../src/qdp++/configure \
--prefix=/home/chroma-user/install/ \
qdp++_single_scalar --enable-parallel-arch=scalar .
```

Following, the install directory is created automatically. The default precision is single and the default compiler is `g++`.

Four possible options exist for `--enable-parallel-arch=<arch>`, namely `scalar`, `parscalar`, `scalarvec`, `parscalarvec`. The first two are for scalar like machines, on the one hand for a single node system, on the other hand for parallel machines. The last two options are for vector like machines. Only the first two options have been tested in this work.

More useful options for `configure` are shown in the following example (a test of different compiler flags can be found in section 4.2):

```
../../src/qdp++/configure \
--prefix=/home/chroma-user/install/ \
qdp++_double_scalar \
--enable-parallel-arch=scalar --enable-sse2 \
--enable-precision=double \
CXXFLAGS="-O2 -finline-limit=50000" CFLAGS="-O2"
```

If `configure` did not produce any errors, it is possible to call the generated `Makefile` by typing `make` followed by `make install`.

In addition to the `Makefile` the `configure` script produces a log file called `config.log`. This log file is located in the build directory. In this way, it is possible to control the executed commands of the `configure` script by means of this file. If any errors occurred, more information can be obtained by studying the log file.

One possible error relates to `automake`. It is possible that the `configure` script runs without errors but when calling the produced `Makefile` by typing `make` the error could occur and the following information is printed:

```
WARNING: `automake-1.9' is missing on your system.
 You should only need it if you modified
 `Makefile.am', `acinclude.m4' or `configure.ac'
 You might want to install the `Automake' and `Perl'
 packages.
 Grab them from any GNU archive site.
```

It is possible, that the makefiles in the source directory do not have the correct date stamp dependencies. The easiest way to deal with this problem is to ensure that there are the installed version `autoconf-2.58` and `automake-1.9.3` on the machine. Then the necessary files will be recreated. Another way to fix the problem is to go to the source directory and run the following commands:

```
aclocal
automake
autoconf
```

and afterwards to go back to the build directory and try again. Nevertheless it will be necessary to have the installed version `autoconf-2.58` and `automake-1.9.3`.

After a successful installation of QDP++, it is possible to build Chroma on top of this package.

Building Chroma

Assuming the structure of the directories described above, the sources are located in /home/chroma-user/src/chroma. As in the case of QDP++, it is necessary to create an additional directory:

```
mkdir /home/chroma-user/build/chroma_double_scalar .
```

The way of building Chroma and QDP++ is very similar. One has to change the directory and execute the configure script:

```
cd /home/chroma-user/build/chroma_double_scalar
../../src/chroma/configure \
--prefix=/home/chroma-user/install/ \
chroma_double_scalar \
--with-qdp=/home/chroma-user/install/ \
qdp_double_scalar \
CXXFLAGS=" " CFLAGS=" " .
```

The empty quotation marks ensure that autoconf does not use the default -g option which would enable debugging.

This unoptimised code can be improved on SSE enabled systems by means of a SSE assembler tuned Wilson Dslash operator. There are several versions of this optimised operator in the package depending on the chosen precision and architecture. The right one is chosen when enabling this option in the configure script. Then the last command becomes:

```
../../src/chroma/configure \
--prefix=/home/chroma-user/install/ \
chroma_double_scalar \
--with-qdp=/home/chroma-user/install/ \
qdp_double_scalar \
--enable-sse-wilson-dslash \
CXXFLAGS=" " CFLAGS=" " .
```

The definition of this optimised operator is located at $CHROMA_SRC/other_libs/intel_sse_wilson_dslash and the corresponding subdirectories.

Each package has a configuration file located in the subdirectory bin of its install-directory. This shell script called <package-name>-config can give informations about the installed version of the package, e.g its version number, default precision, the used compiler and its flags. If one executes this file, it prints a list of possible input options. Using these options, it is possible to prompt the accordant informations.

2.2.2 Parallel Build

Information about the parallel build of the Chroma software system cannot be found in a documentation but in principle, the concept of building it is straight forward. For a parallel running version of Chroma, it is necessary to use the additional package that is responsible for the communication (QMP). The two other packages described in the previous section have to be able to use the functionality of QMP. Therefore, it is necessary to build these packages on the top of each other, beginning with the lowest level.

The QCD Message Passing API is a module written in C. The utilised version has to make use of MPI. So it is important to choose the right options for the MPI specification and for the compiler when running the `configure` script of this package.

```
mkdir /home/chroma-user/build/qmp
cd /home/chroma-user/build/qmp
```

As in the case of the other packages, one can list the possible options for the `configure` script by typing

```
configure --help
```

in the source directory. QMP does not use MPI by default. The additional options are shown below:

```
../../src/qmp/configure \
--prefix=/home/chroma-user/install/qmp \
--with-qmp-comms-type=MPI CXX=mpicc .
```

Here `mpicc` is the MPICH C compiler. When running this script an error could occur. Depending on the installation of MPI the compiler does not link all needed libraries to execute a binary. MPI compiled code has to be executed by means of `mpirun`. With this tool it is possible to allocate the number of wanted processors with the specified machinefile. The missing libraries are added by `mpirun`. Like all other `configure` scripts this one does some tests of the system and the specified options before producing the `Makefile`. So the compiler creates an executable of a small test program and the script tries to execute it by calling the default output file `a.out`. The following example shows the possible error that occurs because `a.out` was executed without `mpirun`:

```
configure:1530: ./a.out
 ./a.out: error while loading shared libraries:
 libmpich.so.1.0: cannot open shared object file:
```

```
No such file or directory .
```

To solve this problem the location of the wanted library has to be exported before running `configure`.

```
locate libmpich.so.1.0;
export LD_LIBRARY_PATH=/usr/ibgd/mpi/gcc/ \
 mvapich-0.9.5-mlx1.0.3/lib/shared
```

Now, if the configure script runs without errors, one can call the Makefile by typing the usual commands `make` and `make install`. The next needed module is QDP++. To build it on the top of QMP, one has to modify several options when calling its `configure` script. On the one hand, the parallel architecture is not `scalar` anymore but `parscalar` for the used PC cluster. On the other hand, the MPICH C++ compiler called `mpiCC` is used. QDP++ has to know the location of the install directory of QMP. For this example, a new build directory is created:

```
mkdir /home/chroma-user/build/qdp++_double_parscalar
cd /home/chroma-user/build/qdp++_double_parscalar

../../src/qdp++/configure \
--prefix=/home/chroma-user/install/ \
qdp++_double_parscalar \
--enable-parallel-arch=parscalar \
--enable-sse2 --enable-precision=double \
--with-qmp=/home/chroma-user/install/qmp \
CXX=mpiCC CXXFLAGS="-O2 -finline-limit=50000" \
CFLAGS="-O2" .
```

This `configure` has to be followed by `make` and `make install`, of course.

The code of the additional package QIO in QDP++ is written in C. In the case of the scalar build the default compiler is used for this software module. Now the C++ compiler is specified to `mpiCC` but not the one for C. This information comes from the QMP configuration file called `qmp-config`.

It is now possible to build the application Chroma on the top of QMP and QDP++. The way to do this is exactly the same like in the scalar case described above. Only the correct location of the install directory of the parallel version of QDP++ has to be set.

2.3 “The HackLatt 2006 Chroma Tutorial”

In this section parts of the “HackLatt 2006 Chroma Tutorial” are summarised and additional information to several points of the tutorial are given.

To learn the first steps of Chroma, it is helpful to pass the provided tutorial. The first parts use the executable `chroma` as a “black box” and show how to deal with the program by means of input files. In this way, the anatomy of this input files which are in XML format is illustrated. In further parts one can learn more about the structure of the software because it is explained how to compile own code with Chroma, i.e. reading and writing XML files is shown here. The whole tutorial can be found at [8].

2.3.1 First measurements with the “black box” `chroma`

Measurements are always done on a single configuration. These computations are called `InlineMeasurements`. This part of the tutorial aims to calculate the spectroscopy on a single configuration. To do so, the following steps have to be done:

- Read the gauge configuration,
- Make a colour source,
- Invert the fermion matrix on the colour source to compute a quark propagator,
- Discard the colour source,
- Potentially save the propagator,
- Perform various contractions on the quark propagator.

It is up to the reader if the colour source is discarded after using it. This object has the same size like a full propagator. So one can save memory by discarding it. Saving the propagator is not a necessary step but it is shown how to do it.

In this example the computations are done on a configuration with the volume $4^3 \times 8$. This configuration and a corresponding input file are available at [8].

Assuming an installed version of the Chroma software system, it is necessary to copy the executable `chroma`, the input file and possibly needed configurations to a working directory. The program needs an input file and writes an output file. These files are specified with the options `-i` and `-o`. Without these options the executable tries to read its input from a file `DATA` and writes the output into a file called `XMLDAT`. Again, own filenames can be chosen by the following command:

```
./chroma -i <input file name> -o <output file name>
```

Anatomy of the XML input file

One can view XML files with common editors, but for a better overview one should open them with a web browser, e.g. "firefox". XML files consist of tags starting with <> and ending with </>. All input parameters and the whole output are inside these tags. In a web browser one can see the XML file as a tree. To get a better overview it is possible to close outer tags.

The outermost tag pair is called `<chroma>`. The next layer contains four tags:

- `<annotation>`
  - Open for everything.
- `<Param>`
  - This is the main parameter tag for chroma.
- `<RNG>`
  - The random number generator seed can be set here.
- `<Cfg>`
  - This tag deals with the default input configuration.

When starting simulations or measurements there is either the possibility to read a saved configuration or to use a given internal one such as `UNIT` for the unit gauge, `DISORDERED` for a hot start and `WEAK_FIELD` which is a slightly perturbed unit gauge. This can be chosen in `<cfg_type>`, a subtag of `<Cfg>`. To load a given configuration the keywords for this tag are `SZIN`, `NERSC`, `CPPACS`, `SCIDAC` and `ILDG`, depending on the file format of the configuration. For the internal ones one has to use either `UNIT`, `DISORDERED`, or `WEAK_FIELD`. The other subtag of `<Cfg>` called `<cfg_file>` corresponds to the filename of the configuration. It is ignored for the internal configurations. The configuration that is read by means of this tag is internally saved as the default configuration. It is possible to read in more than one configuration and to declare them as a `NamedObject` like propagators, sources and so on. Further details can be found at section 2.4.3.

The most interesting tag is `<Param>` since all `InlineMeasurements` are specified here. Expanding this tag, two other ones are at the next layer:

- `<InlineMeasurements>`

  - This tag contains the list of measurement tasks. Each task is specified by an `<elem>` tag pair. In this example five tags are shown corresponding to five measurement tasks.

- `<nrow>`
  - The dimensions of the lattice can be found here. The four numbers correspond to the X, Y, Z and T direction in this order.

Now, the general possibilities of the `InlineMeasurements` are explained but not all points of the special five measurement tasks of this example, i.e. the principle concept of dealing with `NamedObjects` will be described below. A list of more possible tasks can be found in appendix B.1. For further details concerning the measurement of the spectroscopy the reader is recommended to have a look at [8].

The following measurement tasks contain some common tags at their lower layer:

- `<Name>`,
  - The keyword inside specifies the action of this task. All possible keywords are listed in appendix B.1.
- `<Frequency>`
  - The action is only accomplished if this variable is an integral fraction of the update number. This variable is useful when using inline measurements during the simulation.
- `<Param>`
  - The parameters for the measurement are specified. Not all tasks have these tag (e.g. `ERASE_NAMED_OBJECT`), but it is common.
- `<NamedObject>`
  - This tag labels the objects that are dealt with. On the one hand it is possible that the task creates an object which needs an ID, on the other hand it could happen that the measurement needs the ID of a previous object. Some tasks may contain a different number of this ID's depending on their needs.
- `<xml_file>`
  - By default the output of the measurements is written to the global XML file. If the output should be written to a separate XML file, the file name of this one has to be specified here.

In the example the first <elem> task deals with the needed source for the computation of the propagator. For the creation of a forward propagator source the keyword for the <name> tag is MAKE_SOURCE. In the lower level of its <Param> tag, namely in the <SourceType> tag, one can specify the source. A whole list of possible sources is shown in appendix B.1.1. Here, <NamedObject> has only one subtag called <source_id>. The created source gets an ID inside this tag, i.e. all other measurements can use this source as one will see in the following paragraph dealing with the computation of the propagator.

In the <NamedObject> tag of the task with the <Name> PROPAGATOR are two subtags. One is for the needed source, the other one for the computed propagator. So this propagator can be passed on the next measurement task that calculates the spectroscopy.

In the example, there is an intermediate step before calculating the spectroscopy. First, the source that is not needed anymore is erased. Then, the propagator is saved in a separate file. The task with the <Name> ERASE_NAMED_OBJECT does not contain any parameter tags. The quoted ID will just be erased and will not be in memory anymore. With the action QIO_WRITE_NAMED_OBJECT it is possible to write any NamedObject to a file. In the subtag <object_type> one can choose the type of object that should be written to the file quoted in the <file_name> tag. This means that this routine can deal with all possible Chroma types (LatticePropagator, LatticeColorMatrix,etc) described in section 2.1.4. In the implementation of the Chroma software system there are datatypes which are put together of several classes by means of templates. If one wants to write a type like multi1d<LatticeColorMatrix> to a file, the symbols <> have not to appear in the <object_type> tag, i.e. the specification in this tag would be multi1dLatticeColorMatrix. The <file_volfmt> tag can have the values SINGLEFILE, MULTIFILE or PARTFILE. More details can be found in section 2.1.3. These files have the LIME file format that is described in section 2.1.3.

The computation of the spectrum is the last measurement task in the example. The details of input parameters can be found at [8]. Most of them consist of flags that can have the value true or false, e.g. whether one wants to do the spectroscopy with an unsmeared (point), a wall or a smeared sink. For the smearing one can choose the applicable wavefunction. The possible wavefunctions are listed in appendix B.1.2.

Anatomy of the XML output file

As in the input file, the outermost tag is <chroma> . Five tags can be found below this outermost one:

- <Input>

  - The whole input file is echoed here.

- <ProgramInfo>

  - Besides the version of Chroma and QDP++ information about the geometry of the lattice can be found in the subtag <Setgeom>, namely the size of the lattice (<latt_size>), the size of the processor grid (<logical_size>), the size of the local lattice on each processor (<subgrid_size>), the lattice volume (<total_volume>) and the local volume on one processor (<subgrid_volume>). The values inside these five subtags consist of four integers corresponding to the four space-time dimensions in this example. The values inside the tags with the index $i = 0, 1, 2, 3$ fulfil the following relations:

$$
\begin{aligned}
< \text{total_volume} >= \sum_i < \text{latt_size} > \qquad (2.1)\\
\#\text{processors} = \sum_i < \text{logical_size} >_i \\
< \text{subgrid_size} >_i = \frac{< \text{latt_size} >_i}{< \text{logical_size} >_i} \\
< \text{subgrid_volume} >= \sum_i < \text{logical_size} >_i
\end{aligned}
$$

- <RNG>

  - The current seed of the random number generator is echoed here.

- <Config_info>

  - The possible metadata of the configuration is presented here if available.

- <Observables>

  - These observables are measured by default.

- <InlineObservables>

  - Every inline measurement task is itemised here with its own output. Further details can be found in [8].

The output of the spectroscopy is more comprehensive. In the directory

/home/chroma-user/src/chroma/tests/chroma/hadron/ ,

are a lot of example input and corresponding output files to get an overview of the possibilities for the measurement of spectrums.

The correlators of mesons are 2-point functions with accordant Dirac matrices inside corresponding to their quantum numbers. In the Chroma software system the Dirac matrices are labelled $\gamma_{1,2,3,4}$ for x, y, z, t. Each matrix gets a 4-digit binary number to name it such as

$$\begin{aligned} \gamma_1 &\rightarrow (0001)_2 \rightarrow 1_{10}, \\ \gamma_2 &\rightarrow (0010)_2 \rightarrow 2_{10}, \\ \gamma_3 &\rightarrow (0100)_2 \rightarrow 4_{10}, \\ \gamma_4 &\rightarrow (1000)_2 \rightarrow 8_{10}. \end{aligned}$$

The multiplication of different γ matrices is labelled as shown in the following example:

$$\begin{aligned} \gamma_1 \cdot \gamma_2 &\rightarrow (0011)_2 \rightarrow 3_{10}, \\ \gamma_5 &\rightarrow (1111)_2 \rightarrow 15_{10}. \end{aligned}$$

In the output file of the spectroscopy one can find a lot of correlators inside `<elem>` tags. Each correlator is labelled with decimal numbers inside the tag `<gamma_value>` corresponding to the Dirac matrix that has been inside the 2-point function during the measurement.

For the analysis of this correlators the software package ADAT is provided. This can be found at [3] or via CVS. The latter is described within the tutorial. Two possibilities of ADAT are described namely getting the data of the correlators from the XML output files into plain text files and the computation of the effective mass. It is also possible to use other tools to get the data out of the XML files either complex tools for the transformation of XML files like XSLT [17] or just a shell script. The effective mass utility uses jackknifing for the error analysis. But the software is not useful for an exact computation of the effective mass because it uses only one exponential function to fit the data. Further details in computing effective masses are reported in chapter 5.

2.3.2 Using the library of Chroma to write Programs

To get more information about the functionality of the Chroma software system the next part of the tutorial [8] is discussed here. First steps will deal with `Makefiles` for own code using the Chroma libraries by means of the configuration scripts `<package>-config`. Furthermore, one will learn about the I/O mechanism with XML files. Thus, the possibilities to import and export data from or to XML files are shown when writing own programmes for instance.

If one wants to compile own code with the functionality of an installed version of the Chroma software system one needs some information about this version:

- location of the install directory,
- compiler and its flags,
- needed libraries,
- location of additional directories where needed libraries may be found.

The provided `Makefile` is shown below:

```
CHROMA=/home/../install/chroma_scalar_single
CONFIG=$(CHROMA)/bin/chroma-config
CXX=$(shell $(CONFIG) --cxx)
CXXFLAGS=$(shell $(CONFIG) --cxxflags) -I.
LDFLAGS=$(shell $(CONFIG) --ldflags)
LIBS=$(shell $(CONFIG) --libs)
HDRS=
OBJS= tut3.o
tut3: $(OBJS)
  $(CXX) -o $@ $(CXXFLAGS) $(OBJS) $(LDFLAGS) $(LIBS)
%.o: %.cc $(HDRS)
  $(CXX) $(CXXFLAGS) -c $<
clean:
        rm -rf tut3 $(OBJS) *~
```

The only thing that has to be changed is the variable `CHROMA` that specifies the location of the install directory of Chroma. Appropriate to the choice of default install directories this variable would get the value `/home/chroma-user/install/chroma_double_parscalar`.
All other informations are generated by the script `chroma-config`.

In the previous chapter the anatomy of the XML files was described. To read the values inside the tags one has to open the XML file and to read the parameters with a provided `read` function. The tree structure of the tags can be interpreted as a path structure. For example, the lattice dimension was located in the following tag structure:

```
<chroma>
 <Param>
  <nrow>X Y Z T</nrow>
 </Param>
```

```
</chroma>
```

That leads to the path structure `/chroma/Param/nrow`. Assuming this tag contains only one single integer value, this value is read in the following way:

```
int number;
XMLReader xml_in;
xml_in.open(<input_filename>);
read(xml_in, "/chroma/Param/nrow", number);
```

The data is read only by the primary node. If one wants to read the lattice dimensions of a XML input file, the steps are quite similar:

```
multi1d<int> nrow(Nd);
XMLReader xml_in(<input_filename>);
read(xml_in, "/chroma/Param/nrow", nrow);
```

The first line defines an array of `Nd` integers (`Nd` is the space-time dimension constant specified at compile time). The function for reading the values is called `read` in both cases as well as for several more datatypes. The choice of the internally called function depends on the input parameters. This mechanism of overloading functions is a part of polymorphic programming.

There are at least two possibilities to open the input file. In both examples `xml_in` becomes an object of the class `XMLReader`. In the first case the method `open` of the object `xml_in` is called, in the second example the `<input_filename>` is specified by means of the constructor of the class `XMLReader`. Beside the filename the constructor can get the path or parts of it from the tag in the XML file. This is illustrated by the following example:

```
multi1d<int> nrow(Nd);
XMLReader xml_in(<input_filename>, "/chroma/Param");
read(xml_in, "nrow", nrow);
```

Constructors can also be overloaded methods of their classes.

The declaration of this class is located in
`$QDP++_SRC/lib/qdp_xmlio.cc` whereas one can find the prototypes in the accordant header file that is located in `$QDP++_SRC/include` and `$QDP++_INSTALL/include`, respectively.

In the previous chapter the options `-i` and `-o` were used to specify the global input and output files, respectively. The filenames behind these options are used in the code with the method `Chroma::getXMLInput\ FileName()` and `Chroma::getXMLOutputInstance()`, respectively. Therefore, the last example becomes:

```
multi1d<int> nrow(Nd);
XMLReader xml_in(Chroma::getXMLInputFileName(), \
 "/chroma/Param");
read(xml_in, "nrow", nrow);
```

When trying to read data from files it could happen that some errors occur, e.g. if the names of the tags inside the file and the path in the reading method differ. Normally the program would abort immediately with an error message. In C++ this is realized by means of exceptions. This mechanism enables the user to catch these exceptions and to choose how to handle them. To do so one has to put the methods that could produce any errors into a `try` block with a following `catch` block. In the `catch` block the user can decide how to deal with the potential errors. The `XMLReader` 'throws' exceptions of the type `string` that contains the error message. Other methods can 'throw' exceptions of other types. The following examples show how to catch a `string` exception and how to catch any exceptions.

```
try
 {
   // Read anything
   read();
 }
catch (const string& e)  // Catch any exceptions
 {
   QDPIO::cerr << "Error reading data:"<< e << endl;
   QDP_abort(1);
 }
```

The `string` contains:

```
XPath Query: /layout/row did not return unique node.\
 nodes returned = 0
```

Furthermore, if one does not know which kind of exception is thrown or if one wants to handle different exceptions in other ways, it is possible to use more than one `catch` block with other exception types such as

```
catch(type_1 variable_1){}
catch(type_2 variable_2){}
.
.
catch(type_n variable_n){}
```

The way to catch any exceptions is

```
catch(...)
```

If the method catches the exception and the user has decided not to write the `string` to the error stream, no message will occur. If no function for aborting is called, the program will hold. `void QDP_abort(int status)` should send kill signals to all nodes [18] but all tests of this showed that only one task is killed by this function. This is exactly the same task like in the case if one does not catch the thrown exception. However, this method has the advantage that the thrown exception can give more informations about the occurred error.

The way to read data from an XML file was shown. The next part will explain how to write data to an XML file. To open an XML file for writing one has to use the class `XMLFileWriter` defined in the file `/lib/qdp_xmlio.cc` like the reader class. Here, there are also different ways to specify the filename. Either one can use methods of this class or the provided constructors. The XML output file that was chosen with the option `-o` when executing the binary is specified with the command `Chroma::getXMLOutputInstance()`. For opening and closing tags in the XML output file the operations `push()` and `pop()` are available, respectively. The following commands

```
XMLFileWriter xml_1("test.xml");
push(xml_1,"Root");
write(xml_1, "Integer", 5);
pop(xml_1);
```

open the file `test.xml` with the tag `Root` and write the value `5` into the tags `Integer`.
`XMLFileWriter xml_2(Chroma::getXMLOutputInstance());`
would write these things into the standard XML output file. The content of the file is

```
<?xml version="1.0"?>
<Root>
 <Integer>5</Integer>
```

```
</Root> .
```

The QDP++ data types have a defined output style that is shown at page 24 of [18].

2.4 Simulations and measurements with Chroma

In the previous sections most of the information that can be found about the Chroma software system are summarised. In the following section some aspects are discussed that can be found in example files inside the tar archive of Chroma or inside the source code itself. Furthermore some information is presented resulting from private communication with the "Qcd-measure-chroma e-mail list" [19].

Simulations with the Chroma software system can be done with the executables `purgaug` or `hmc` depending on the desired simulation and algorithm. `purgaug` does a pure gauge field generation with the heatbath algorithm whereas `hmc` generates dynamical fermions with hybrid monte carlo simulations. The input files are in XML style.

2.4.1 Dynamical fermions with Hybrid Monte-Carlo

To perform simulations including dynamical fermions one can use the executable `hmc` as a "black box" if one has understood the XML input file. There are several input file examples located in `$CHROMA_SRC/tests/hmc`. Nevertheless, some general parameters of these files are shown below.

The list of parameters splits into two parts. As described in section 2.3.1 one can open the files in a web browser. The first tag `<MCControl>` contains general information about the random number generator, the input or start configuration, the frequency and file volume for the storage of produced configurations, the total number of updates, the number of warm ups and wanted `InlineMeasurements` during the simulation. In the tag `<HMCTrj>` the rest of the needed information can be found, e.g. the actions and the integrator for the molecular dynamics.

The subtags `<Cfg>`, `<RNG>` and `<InlineMeasurements>` are already explained in section 2.3.1. The generation of configurations with the Chroma software system allows for easy continuation of a simulation after an interruption, because several parameters are implemented in the tags `<StartUpdateNum>`, `<NProductionUpdates>` and `<NUpdatesThisRun>`. The maximum number of updates for one setup is given by `<NProductionUpdates>` when setting `<StartUpdateNum>` to `0` (ignoring `<NWarmUpUpdates>`). Anyhow, the number of updates for one run is given by the value for `<NUpdatesThisRun>`. The calculation

for the number of updates N_{todo} is done in the file hmc.cc which is located in $CHROMA_SRC/mainprogs/main. For a better understanding the calculation is shown below.

$$N_{\text{ProductionUpdates}} + N_{\text{WarmUpUpdates}} > N_{\text{UpdatesThisRun}} + N_{\text{StartUpdateNum}}$$
$$\rightarrow N_{\text{todo}} = N_{\text{UpdatesThisRun}}$$
$$N_{\text{ProductionUpdates}} + N_{\text{WarmUpUpdates}} < N_{\text{UpdatesThisRun}} + N_{\text{StartUpdateNum}}$$
$$\rightarrow N_{\text{todo}} = N_{\text{ProductionUpdates}} + N_{\text{WarmUpUpdates}} - N_{\text{StartUpdateNum}}$$

The value inside the tag <NWarmUpUpdates> specifies the number of thermalisation trajectories. InlineMeasurements are not done while warming up. After the production of a trajectory an accept/reject step is necessary to get the wanted distribution of configurations. During the warm up this accept/reject step is omitted, i.e., all trajectories are accepted. The accept/reject step is done in the class AbsHMCTrj defined in the file abs_hmc.h in the directory
$CHROMA_SRC/lib/update/molecdyn/hmc.

The three last subtags of <MCControl> deal with the storage of produced configurations. During the warm up no configuration is saved. Afterwards configurations are written if the update number is an integral multiple of <SaveInterval>. The filename is specified by the string inside <SavePrefix> and the update number. Therefore, the name is given by<SavePrefix>_cfg_<updatenumber>.lime, for instance. Each time a configuration is saved, an additional input file for a restart is written. Both the current seeds of the random number generator and all other parameters are saved. One can use this input file to restart at this point and to reproduce exactly the same configurations as without interrupting. The tag <SaveVolfmt> can have the values SINGLEFILE, MULTIFILE or PARTFILE. Further details can be found in section 2.1.3.

The second subtag of <Params> called <HMCTrj> again consists of three subtags, <MC_Hamiltonian>, <MDIntegrator> and <nrow>. The last one contains the space-time dimensions of the lattice and is already known. In the tag <MDIntegrator> the information for the integration scheme is given. Here, at least the total length of the trajectory <tau0> and the number of integration steps <n_steps> have to be specified. Based on these two values the step size for the integration scheme is calculated. If an integrator needs more than these values, one has to specify them in this tag. After explaining the idea of <Monomials> further details about the additional parameters for the integration scheme are shown.

The only subtag of <MC_Hamiltonian> is <Monomials>. Inside this tag the action for the simulation is defined. This is done by splitting the action in terms called monomials corresponding to the <elem> tags sitting in the <Monomials>. Each monomial has a name inside the corresponding tag. The possible monomials are listed in appendix B.2. So

it is up to the user how to mix several monomials to create the wanted action. The input file examples mentioned above show several possibilities. A short example for two flavours with an even/odd preconditioned Wilson action is given below. The `<Name>` of the `<Monomials>` is called `TWO_FLAVOR_EOPREC_CONSTDET_FERM_MONOMIAL` in this case. There is also another even/odd preconditioned Wilson-Dirac operator in the Chroma software system. The specifications of these two operators can be found in the files `$CHROMA_SRC/lib/prec_constdet_linop.h` and `$CHROMA_SRC/lib/prec_logdet_linop.h`. Further details of the differences can be found there.

```
<Monomials>
 <elem>
  <Name>
   TWO_FLAVOR_EOPREC_CONSTDET_FERM_MONOMIAL
  </Name>
  <InvertParam>
   <invType>CG_INVERTER</invType>
   <RsdCG>1.0e-7</RsdCG>
   <MaxCG>1000</MaxCG>
  </InvertParam>
  <FermionAction>
   <FermAct>WILSON</FermAct>
    <Kappa>0.11</Kappa>
   <FermionBC>
    <FermBC>SIMPLE_FERMBC</FermBC>
    <boundary>1 1 1 -1</boundary>
   </FermionBC>
  </FermionAction>
  <ChronologicalPredictor>
   <Name>LAST_SOLUTION_4D_PREDICTOR</Name>
  </ChronologicalPredictor>
 </elem>
 <elem>
  <Name>GAUGE_MONOMIAL</Name>
  <GaugeAction>
   <Name>WILSON_GAUGEACT</Name>
   <beta>5.7</beta>
   <GaugeBC>
    <Name>PERIODIC_GAUGEBC</Name>
   </GaugeBC>
  </GaugeAction>
 </elem>
</Monomials>
```

In the first monomial the fermionic part of the action is defined. All keywords in capital letters correspond to a specified command possibly with its own set of parameters respectively tags. All facilities are shown in appendix B.2. Most of the parameters are intuitively understandable. The value inside the tag `<RsdCG>` which is a subtag of `<InvertParam>` corresponds to the residual error yielding the exit condition of the inverter. The tolerance is weighted with the norm of the solution vector. Assuming one has to solve $A \cdot x = b$, the value of `<RsdCG>` is δ and the residual vector is called r. Then, the exit condition is fulfilled if

$$\delta^2 \cdot \|b\|^2 > \|r\|^2.$$

The inverters are defined in the corresponding files in `$CHROMA_SRC/lib/actions/ferm/invert`.

After the description of dealing with monomials, further possibilities of integration schemes are shown. A multi-timescale integrator uses different step sizes for the different parts of the action that are defined as monomials. The computationally intensive parts of the action get the larger step size. Therefore, one has to define the different monomials. The following example shows a second order minimum norm integrator with multiple time scales. The definitions of the integrators can be found in `$CHROMA_SRC/lib/update/molecdyn/integrator`.

```
<MDIntegrator>
 <Name>LCM_TWO_SCALE_MINIMUM_NORM_INTEGRATOR</Name>
 <tau0>1</tau0>
 <n_steps>4</n_steps>
 <n_short_steps>2</n_short_steps>
 <S_short_monomials>1</S_short_monomials>
 <S_long_monomials>0</S_long_monomials>
</MDIntegrator>
```

The monomials are numbered by integers starting from zero. In the example above, there are two monomials, the fermionic part with number `0` and the gauge part with number `1`. In the example of the integrator `<S_long_monomials>` has the value `0` and would relate to the fermionic part whereas the gauge part would get the shorter step size. `<n_steps>` specifies the total number of integration steps. The number of short steps for the second monomial is given by the value of `<n_short_steps>` times the total number of steps. Here, the timescale is a half of the long one. Because of the possibility to use several monomials one can also type a list of numbers in the corresponding tag, e.g. `<S_short_monomials>1`

2</S_short_monomials>. To see more examples the reader is invited to have a look at $CHROMA_SRC/tests/t_leapfrog.

2.4.2 Pure gauge simulations with the Heat Bath Algorithm

In this case the executable purgaug with a corresponding input file has to be used. Examples for input files are in $CHROMA_SRC/tests/purgaug. In these files the first set of parameters has the same meaning as in the case of hybrid monte carlo simulations. The tags <Cfg>, <RNG> and all tags for the control of the number of produced configurations, the frequency of storage and the file names for this configurations are also found in these input files. Additionally a part for <InlineMeasurements> is available. To control the algorithm, there is the tag <HBItr> with the following subtags:

```
<HBItr>
 <HBParams>
  <BetaMC>6.0</BetaMC>
  <nOver>3</nOver>
  <NmaxHB>1</NmaxHB>
  <t_dir>3</t_dir>
  <xi_0>2.4643</xi_0>
  <anisoP>true</anisoP>
 </HBParams>
 <nrow>4 4 4 8</nrow>
</HBItr> .
```

For example, <nOver> is the number of overrelaxation steps, <anisoP> is a boolean variable which decides about anisotropy. <xi_0> is the anisotropy factor and <t_dir> its direction. These parameters are saved in a 'struct' called HBParams defined in hb_params.h in $CHROMA_SRC/lib/update/heatbath. In this directory the most necessary methods for the update are defined.

Further details dealing with pure gauge theory are beyond the scope of this thesis.

2.4.3 Simulations and measurements at an Opteron PC Cluster

The hardware of the Opteron PC-Cluster of the Department of Physics of the Humboldt University is introduced in section 4.1. To test the Chroma software system at this PC-Cluster, several simulations and measurements have been performed. Because of the complexity it was not possible to use the whole functionality of this package. All simulations were done with Wilson fermions with periodic boundary conditions in space and antiperiodic boundary conditions in time. Both unpreconditioned and even-odd

preconditioned Wilson-Dirac operators were used. To invert the Dirac operator the method of conjugate gradients was applied. For the molecular dynamics the leap-frog integrator was used.

For the simulations the hopping parameter $\kappa = 0.1575$ and $\beta = 5.6$ were chosen.

Simulations on different numbers of nodes

It is necessary to have sufficient statistics to measure observables in LQCD. Observables should be independent of the used algorithm, computer architecture and computing nodes within their statistical errors. But it is not obligatory that a certain value, e.g. the plaquette of each configuration is exactly equal when using a different number of computing nodes. It is even possible that not exactly the same configurations are generated on a different number of processors. On the one hand this depends on how the random number generator creates different random numbers on each node, on the other hand global sums that need the communication between all nodes could depend on the number of these nodes. The random number generator of the Chroma software system generates the same random numbers independently of the number of computing nodes. The random number generator is an application of QDP++. The production of the random numbers depends on the datatype that should be filled with them. Lattice wide datatypes get seeds for the random number generator depending on their coordinates on the lattice. It is guaranteed with this mechanism that different nodes have different random numbers because they deal with different coordinates of the lattice wide datatypes. A variable that depends not on the coordinate of the lattice would get the same numbers on every node.

When creating the layout of the lattice that was discussed in section 2.1.4 global lattice has to be portioned in local ones for each processor. The Chroma software system can do this partitioning automatically. When using a parallel-scalar ("parscalar" which was described in section 2.2.1) architecture this is done in the file `$QDP++_SRC/lib/qdp_parscalar_layout.cc`. The definition of the layout of the machine and thus the partitioning of the global lattice can also be specified by the user. For this the command `-geom <x> <y> <z> <t>` has to be used when executing the wanted program such as

```
./chroma -geom <x> <y> <z> <t>
```

wherein `<x> <y> <z> <t>` correspond to $< \text{logical_size} >_i$ of equation (2.1).

Measurement on several configurations

In section 2.3.1 it was described how to do measurements on a single configuration. This configuration was read in by means of the `<Cfg>` tag in the input file. It is possible to read the configuration as a task of `<InlineMeasurements>`. The list of possible tasks is shown in appendix B.1. The corresponding keyword is called `QIO_READ_NAMED_OBJECT`. So one can read a configuration that is saved in the LIME format. This object gets an ID corresponding to the idea of `NamedObjects`. The following example shows the part of the inline measurement of an input file which can deal with an arbitrary number of configurations. Some irrelevant parts of the input file are skipped and the corresponding tags are retracted.

```
<InlineMeasurements>
 <elem>
  <annotation>Read config</annotation>
  <Name>QIO_READ_NAMED_OBJECT</Name>
  <Frequency>1</Frequency>
  <NamedObject>
   <object_id>cfg</object_id>
   <object_type>
    Multi1dLatticeColorMatrix
   </object_type>
  </NamedObject>
  <File>
   <file_name>
    kappa_01575_run01_cfg_1.lime
   </file_name>
  </File>
 </elem>
 <elem>
  <Name>MAKE_SOURCE</Name>
  <Frequency>1</Frequency>
  <Param></Param>
  <NamedObject>
   <source_id>pt_source</source_id>
  </NamedObject>
 </elem>
 <elem>
  <Name>PROPAGATOR</Name>
  <Frequency>1</Frequency>
  <Param></Param>
  <NamedObject>
   <gauge_id>cfg</gauge_id>
```

```
    <source_id>pt_source</source_id>
    <prop_id>pt_prop</prop_id>
   </NamedObject>
  </elem>
  <elem>
   <annotation>Get rid of the config</annotation>
   <Name>ERASE_NAMED_OBJECT</Name>
   <NamedObject>
    <object_id>cfg</object_id>
   </NamedObject>
  </elem>
  <elem>
   <annotation>Get rid of the source</annotation>
   <Name>ERASE_NAMED_OBJECT</Name>
   <NamedObject>
    <object_id>pt_source</object_id>
   </NamedObject>
  </elem>
  <elem>
   <annotation>Get rid of the propagator</annotation>
   <Name>ERASE_NAMED_OBJECT</Name>
   <NamedObject>
    <object_id>pt_prop</object_id>
   </NamedObject>
  </elem>
 </InlineMeasurements>
```

This `<InlineMeasurements>` task computes a propagator without using the `<Cfg>` tags but by means of `NamedObjects`, i.e. one can calculate the spectrum of a large number of configurations with the executable `chroma` and a corresponding input file. For every configuration one needs at least these steps with the additional task of the measurement of the spectrum. It is important to erase the object after using them so that they are not in memory anymore. It is common that these input files have several thousand lines when dealing with few configurations. For example, an input file for the measurement of the spectrum of 800 configurations has a size about 2.5 MB, but that should not be a problem since the limit on the size of the file is apparently huge.

2.4.4 Binary storage of gauge configurations

Gauge configurations are saved in the LIME file format. The default storage precision is single. A possibility to change the default storage precision is to

modify the source code in the file `$CHROMA_SRC/lib/io/gauge_io.cc`. The function `void writeGauge(...)` defined in line 57 of this file gets the gauge field that has to be written as an input parameter. This field has the precision that was determined at compile time. The field is written to second field of the type `multi1d<LatticeColorMatrixF>`, i.e. to a single precision gauge field. After this conversion the configuration is stored. It is possible to skip this step of converting the gauge field and to write the field that was one of the input parameters of the `writeGauge(...)` method. Following, the configuration will be saved in the precision that was defined at compile time.

2.5 Summary of the experiences with the Chroma software system

The Chroma software system is a comprehensive software project. On the one hand a lot of different possibilities are provided for simulation and measurement in LQCD. On the other hand it is affirmed that an acceptable efficiency of the code can be reached on several different computer architectures. This work deals only with the Chroma software system on an Opteron PC-Cluster. Benchmarks of certain parts of the software can be found in chapter 4. To reach both a high efficiency and portability the software system consists of several software modules. For each module exists a manual that describes its functionality. The main description of the application Chroma can be found in the tutorial [8].

Because of the description how to build a running version of the software system on a single node machine [9] it is possible to get this application without bigger problems. If the control mechanism of the binaries by means of XML input files is understood it is possible to do first steps with the black boxes `chroma`, `hmc` and `purgaug`. Possibilities of the software system that are not described in the tutorial have to be obtained from the several examples files in the directory `$CHROMA_SRC/tests`. After a short while it should be possible to do several simulations and measurement with the Chroma software system.

Greater problems occur if one wants to understand the code in more details. For QDP++ and Chroma (not for QMP) the source code documentation generator tool Doxygen exists to browse the source code. With this tool it is more comfortable to find certain functions and the files in which these are defined. Nevertheless it is hard work to understand details of the code. One problem is the several of thousands coded lines in the sources of the different software modules. Sometimes the idea where to start when analysing one task is missing. Beside the complexity of the software system another problem relates to the polymorphic programming of the higher level modules like QDP++ and Chroma, i.e. overloading of any functions

and operators of the software modules. The advantages for the user emerge when writing own code. Then functions or operators with same names can be used to do related operations corresponding to the set of input variables. For the multiplication of a matrix with a vector or for the multiplication of two vectors the same operator is used although it does different things. These advantages can lead to problems if the user wants to manipulate the accordant operation because it is hard to find out in which function the actual operation is done.

As this work was started the current version of QDP++ was 1.20.2. The output of all datatypes has a precision of 6 digits. Each variable for instance the plaquette is an object of a certain class in the meaning of object oriented programming. It is a nontrivial task to change this output precision because it is hard to find the right function that writes the accordant variable into an XML file. In the current version of QDP++ the output precision is increased.

The Chroma software system provides a comprehensive sample of simulation and measurement tasks for LQCD. It is remarkable that only few things are determined at compile time. For instance variables like the dimensions of the lattice as well as the discretisation of the action or other specifications of the algorithm can be defined at run time. A lot of artifices of programming are necessary to build efficient code with this flexibility at run time. For the usage of the manifold possibilities of the Chroma software system it is more or less sufficient to be able to deal with XML input files. But the complexity of the software and its way of programming can lead to problems if enhancements of the functionality are wanted.

Chapter 3

Description of the Domain Decomposition Hybrid Monte-Carlo (DD-HMC) software

The sources of the software described below are available on web page [20]. Furthermore the papers [21, 22] that describe the ideas of the domain decomposition algorithm for two flavour LQCD are presented there. Beside the code several `README` files can be found in the source tar archive. Some content of these files is presented below whereas the ideas of the papers are summarised in section 3.5. The set of parameters needed for a simulation with the DD-HMC software and how to deal with them is discussed in section 3.6.

The DD-HMC software was published during the processing of this thesis. It was not the main task to analyse this code. Therefore, the experiences with the DD-HMC software are fewer compared to the Chroma software system.

3.1 Overview of the software

Program scope

The DD-HMC code presents a software package for LQCD simulations using hybrid monte carlo methods combined with domain decomposition ideas. Furthermore a multiple-time integration scheme is used for the molecular dynamics. For the action $O(a)$-improved Wilson fermions are implemented. The solution of the Dirac equation is available through a set of several solvers such as a CG inverter, a BiCGstab inversion program and a Schwarz-preconditioned GCR solver. In the simulation program this set is

used to generate gauge configurations by means of domain decomposition methods in an efficient way.

The program runs on single node machines as well as on parallel systems. It parallelises in all dimensions. An implementation for an SSE optimised Wilson-Dirac operator for current Intel or AMD processors using inline assembly code is also presented.

For this C-written code a installation of MPI is necessary. It compiles with the GNU `mpicc` compiler without problems. Other compilers like the ones of Intel or Portland did not work by default.

Program documentation

In the tar archive, beside the source code several `README` files are available. The one in the root directory gives an overview of the software. The scope and the documentation of the program and aspects of the compilation are presented and first steps for learning more of the code are recommended as well as how to start a simulation with a correct input file is explained. Furthermore some informations about the SSE and other optimisations are shown.

In the root directory a file called `INDEX` can be found, too. Here the content of the other directories is presented. In several directories there are further `INDEX` files showing the content of the subdirectories. `README` files explain the functions in the accordant directories while an overview of these functions is presented in additional `INDEX` files in the corresponding directories.

The directory `doc` contains several documentation files describing basic parts of the code such as the implementation of the Dirac operator and the random number generator.

3.2 Structure of the directories

With respect to the several `INDEX` files, a short overview of the content of the directories is given in this section. There are six subdirectories in the root directory, namely `data`, `devel`, `doc`, `include`, `main` and `modules`. All necessary files for the software are located in the directories `include`, `main` and `modules`. In the latter there are several subdirectories containing the source code for all necessary modules for the software. The `include` directory contains the header files, each for one subdirectory in the directory `modules` whereas the `main` directory contains the simulation program `run3`.

Further information of the content of the `modules` directory can be found in the corresponding `INDEX` file that is located in this directory.

3.3 Installation

In the main and in the devel directory there are makefiles corresponding to the 'GNU Makefile standards'. Two shell variables have to be defined before calling the makefiles, namely GCC and MPIR_HOME. The first one must be set to the compiler command for the GNU C compiler and the second one gets the install directory of MPICH. Then the MPI compiler has to be located in $MPIR_HOME/bin/mpicc and all code is compiled with this command. One has to specify the compiler flags in the makefiles. Some flags are already given as an example in the Makefile. The ones for the SSE optimisation and the ones for prefetching are explained in the README file in the root directory. The SSE and prefetching optimisations are enabled with the compiler option -D<NAME> (-DSSE2 and -DPM for instance). The option defines the <NAME> behind -D<NAME> as a macro with definition 1. One could compare it with the command

```
#define SSE2 1;
```

In the code there are preprocessor requests that enable SSE2 instructions if the macro is defined. Then the inline assembly written Dirac operator is used. The compiler does not do any SSE optimisations when using this flag -DSSE2 The prefetch optimisation -DPM uses the same mechanism.

Some global variables have to be set before compiling the code such as the lattice volume in each direction and the number of processors for each direction, too. This has to be done in include/global.h starting from the source directory. Some informations about that are in main/README.global starting from the source directory.

3.4 First steps

These steps are also described in the README file in the source directory. Here is shown what can be done first after a successful compilation of the code. On the one hand it is suggested to read README.global to understand some basic structure of the code. Furthermore the documentation in doc/dirac.ps is recommended. On the other hand it is suggested to have a look at the test programs that can be found in the directory devel. Both, tests without the use of MPI and tests using several nodes of parallel machines are given, e.g. the test of the random number generator, benchmarks of the linear algebra and of the Wilson-Dirac operator on the decomposed blocks or on the full lattice. Furthermore it is mentioned where to start if one wants to read the source code for a better understanding.

3.5 Description of the Schwarz-preconditioned Hybrid Monte-Carlo algorithm

In this section the paper [21] is summarised whereas the content of [22] is shown in subsection 3.5.3. The explanation deals not with $O(a)$ improvement although it is included in the simulation program.

Preconditioning is a common technique for solving ill-conditioned systems of linear equations. Both preconditioning of the solver for the Dirac equation and preconditioning of the Hybrid Monte-Carlo (HMC) algorithm itself is possible. For the last aspect the quark determinant is factorised.

3.5.1 Factorisation of the quark determinant

The massive Dirac operator D can be decomposed in several ways. It is possible to write it in the following block form

$$D = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \tag{3.1}$$

in position space. The structure of the four elements A_{ij} depends on the method of the decomposition. The determinant of this operator factorises in three parts such as

$$\det D = \det A_{11} \det A_{22} \det \left\{ 1 - A_{11}^{-1} A_{12} A_{22}^{-1} A_{21} \right\}. \tag{3.2}$$

The operator in the curly bracket is the *Schur complement* of the Dirac operator corresponding to the block structure (3.1). Thus preconditioning leads to a factorisation of the quark determinant

$$\det D = \det R_1 \ldots \det R_n. \tag{3.3}$$

The number of factors depends on the chosen preconditioner.

The factorisation of the quark determinant with respect to the decomposition (3.1) can be used in the HMC algorithm. A short introduction in this algorithm as well as in dealing with quark determinants by means of pseudo-fermions can be found in appendix A.5.

Every part of the factorised quark determinant leads to a pseudo-fermion. With the factorisation according to (3.3) the HMC Hamiltonian has the form

$$H\left[\Pi, U, \phi^\dagger, \phi\right] = T\left[\Pi\right] + S_g + \sum_{k=1}^{n} \phi_k^\dagger \left(R_k^\dagger R_k\right)^{-1} \phi_k, \tag{3.4}$$

wherein $T\left[\Pi\right]$ is the kinetic term of the Hamiltonian and S_g is the action of the gauge field.

Hamilton's equations of motion for the molecular dynamics are given by

$$\frac{\mathrm{d}}{\mathrm{d}\tau}U(x,\mu) = \frac{\partial H\left[\Pi, U, \phi^{\dagger}, \phi\right]}{\partial \Pi(x,\mu)} = \Pi(x,\mu)\,U(x,\mu)\,, \tag{3.5}$$
$$\frac{\mathrm{d}}{\mathrm{d}\tau}\Pi(x,\mu) = -\frac{\partial H\left[\Pi, U, \phi^{\dagger}, \phi\right]}{\partial U(x,\mu)} = -\sum_{k=0}^{n} F_k(x,\mu)\,.$$

$F_k(x,\mu)$ is the quark force that corresponds to the operator R_k whereas F_0 corresponds to the gauge part of the action.

3.5.2 The Schwarz-preconditioned HMC algorithm

The Schwarz procedure is a domain decomposition method and thus deals with the covering of the lattice with several domains. In this case these domains are non-overlapping blocks Λ and for technical reasons the block size as well as the number of blocks in all dimensions is assumed to be even. The union of the black blocks are given by Ω and the union of the white ones by Ω^*.

With respect to equation (3.1) the block decomposition leads to

$$D = \begin{pmatrix} D_{\Omega} & D_{\partial\Omega} \\ D_{\partial\Omega^*} & D_{\Omega^*} \end{pmatrix} \tag{3.6}$$

wherein D_{Ω} is the Dirac operator on Ω with Dirichlet boundary conditions, $D_{\partial\Omega}$ represents the hopping terms from the exterior boundary $\partial\Omega$ of Ω to $\partial\Omega^*$ of Ω^*, whereas $D_{\partial\Omega^*}$ deals with the complement. The structure of these non-overlapping blocks is shown in figure 3.1.

When computing $D\psi$ and D is in the representation (3.6) the quark fields ψ consist of parts ψ_{block} corresponding to the four parts of D in (3.6). Each ψ_{block} lives on a subset of the whole lattice. The quark fields can also be defined on the whole lattice. The extension is done by filling the corresponding fields with zeros to get the representation

$$D = D_{\Omega} + D_{\Omega^*} + D_{\partial\Omega} + D_{\partial\Omega^*}. \tag{3.7}$$

This leads to

$$D_{\Omega} + D_{\Omega^*} = \sum_{all\Lambda} D_{\Lambda}, \tag{3.8}$$
$$D_{\partial\Omega} = \sum_{black\Lambda} D_{\partial\Lambda}, \tag{3.9}$$
$$D_{\partial\Omega^*} = \sum_{white\Lambda} D_{\partial\Lambda}, \tag{3.10}$$

where D_{Λ} is the Dirac operator on the block Λ with Dirichlet boundary conditions and $D_{\partial\Lambda}$ is the interacting part between the blocks.

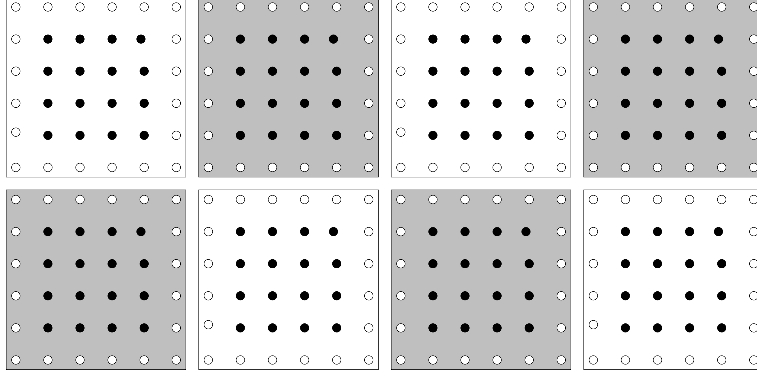

Figure 3.1: Two-dimensional plain of the lattice covered by non-overlapping blocks Λ (Σ). The union of the black and white blocks are given by Ω (Y) and Ω^* (Y^*), respectively, whereas their exterior boundaries (open circles) are given by $\partial\Omega$ (∂Y) and $\partial\Omega^*$ (∂Y^*).

With respect to equation (3.2) this leads to the factorisation

$$\det D = \det D_{\Omega} \det D_{\Omega^*} \det \left\{ 1 - D_{\Omega}^{-1} D_{\partial\Omega} D_{\Omega^*}^{-1} D_{\partial\Omega^*} \right\}. \tag{3.11}$$

Due to equation (3.8) the first two elements on the right hand side of equation (3.11) can be combined such as

$$\det D_{\Omega} \det D_{\Omega^*} = \prod_{all \Lambda} \det \hat{D}_{\Lambda}, \tag{3.12}$$

where $\hat{D}_{\Lambda}$ is the Dirac operator on the block Λ with Dirichlet boundary conditions which can be preconditioned as well. In this case it is the even-odd preconditioned Wilson-Dirac operator.

This factorisation in the representation of (3.3) leads to $n = 2$ that means two operators

$$R_1 = \sum_{all \Lambda} \hat{D}_{\Lambda}, \tag{3.13}$$

$$R_2 = 1 - D_{\Omega}^{-1} D_{\partial\Omega} D_{\Omega^*}^{-1} D_{\partial\Omega^*}. \tag{3.14}$$

3.5.3 Solution of the Dirac equation on the full lattice

If the force F_2 that corresponds to the operator R_2 is computed the Dirac equation on the full lattice has to be solved. For this the Schwarz procedure can be used.

Then the Dirac operator gets the same block form as in equation (3.6) but the blocks in the case here and in the case of the factorisation of the quark determinant are not the same ones. Furthermore, they can differ in

size. Thus here the union of the black blocks is called Y and the union of the white blocks is called Y^* (figure 3.1). The procedure updates iteratively the solution of the Dirac equation. Arriving at block Σ the equations that have to be solved are given by

$$D\psi'(x)|_{x\in\Sigma} = \eta(x) \tag{3.15}$$

$$\psi'(x)|_{x\notin\Sigma} = \psi(x), \tag{3.16}$$

which leads to

$$\psi' = \psi + D_{\Sigma}^{-1}(\eta - D\psi), \tag{3.17}$$

where ψ is the last guess of the solution and ψ' the updated new one. One Schwarz cycle, i.e. a complete sweep over all blocks Σ is done by the expression

$$\psi' = (1 - KD)\,\psi + K\eta, \tag{3.18}$$

wherein the operator K is given by

$$K \equiv D_{Y}^{-1} + D_{Y^*}^{-1} - D_{Y^*}^{-1} D_{\partial Y} D_{Y}^{-1}. \tag{3.19}$$

Starting with $\psi = 0$ it is possible to write this expression as a Neumann series with the operator $1 - DK$.

$$\psi = M_{\mathrm{s}}\eta, \tag{3.20}$$

$$M_{\mathrm{s}} = K \sum_{\nu=0}^{n_{cy}-1} (1 - DK)^{-1}, \tag{3.21}$$

where ψ is now the solution of the full Dirac equation after n_{cy} Schwarz cycles.

Often the convergence of this procedure is disappointing. The idea is to combine the procedure with a Krylov space solver namely the generalised conjugate residual (GCR) algorithm so that the Schwarz procedure is not the solver but the preconditioner for the Dirac equation. The equation

$$DM_{\mathrm{s}}\phi = \eta, \tag{3.22}$$

is anticipated to be better conditioned.

The GCR algorithm solves the Dirac equation iteratively, i.e. produces a set of approximate solutions ψ_k of the equation. The accordant residues are given by

$$\rho_k = \eta - D\psi_k. \tag{3.23}$$

The new approximation ψ_{k+1} is the field in the Krylov space spanned by $\psi_1, \ldots, \psi_k, \rho_k$ that minimises $\|\rho_{k+1}\|$. The residual ρ_k gives the direction in

```
ρ_0 = η
for  k = 0,1,2,...  do
  ξ_k = M_s ρ_k
  χ_k = D ξ_k
  for  l = 0,...,k−1  do
    a_lk = (χ_l, χ_k)
    χ_k = χ_k − a_lk χ_l
  end  do
  b_k = ||χ_k||
  χ_k = χ_k / b_k
  c_k = (χ_k, ρ_k)
  ρ_{k+1} = ρ_k − c_k χ_k
end  do
```

Table 3.1: Pseudo code for the solution of the Dirac equation with a generalised conjugate residual solver preconditioned with the Schwarz procedure.

that the Krylov space is extended and in that the new approximation ψ_{k+1} is shifted.

In the case of the the combination of the Schwarz procedure with this solver the operator M_s is used to calculate a better approximation of the solution in every iteration step of the solver. The scheme is shown in pseudo code in table (3.1). In the outer loop the Schwarz procedure is applied to the residue ρ_k to advance the approximation in the iteration step. The additional fields χ_i that are needed to construct the next approximation fulfil the condition

$$\left(\chi_i, \chi_j\right) = \delta_{ij}, 0 \leq i, j \leq k, \tag{3.24}$$

wherein the brackets stand for the scalar product. The fields χ_i are constructed in the inner loop of the pseudo code in table (3.1) through a Gram-Schmidt orthogonalisation process.

Thus the residual ρ_{k+1} is given by

$$\rho_{k+1} = \eta - \sum_{l=0}^{k} \alpha_l D\xi_l, \tag{3.25}$$

and the field ψ_{k+1} that is the approximate solution of the Dirac equation

can be computed with

$$\psi_{k+1} = \sum_{l=0}^{k} \alpha_l \xi_l. \tag{3.26}$$

In the update step (3.17) the inversion of D_Σ has to be done. For this the minimal residual algorithm is used. Only a few iterations n_{mr} are necessary here because the step (3.17) is only for preconditioning the GCR solver and further iterations would not lead to a better performance of the whole algorithm. The block Dirac operator D_Σ can be preconditioned as well. In this case it is even-odd preconditioned. Thus the blocks must have an even number of points in all dimensions.

Because of the efficiency it can be reasonable to restart the solver if a fixed number of Krylov vectors n_{kv} have been generated. This corresponds to the iteration number k of the pseudo code in table (3.1).

3.5.4 Multiple step size integration

For the integration of equations (3.5) from the fictitious molecular dynamic time $t = 0$ to $t = \tau$ in N steps a numerical integration scheme is necessary. The widely used leap frog integrator is introduced in appendix A.5.2.

With respect to the different forces F_k the fundamental operations for the step size $\epsilon = \frac{\tau}{N}$ are

$$\mathcal{I}_U(\epsilon): \quad U(t+\epsilon) = \exp\left[\epsilon \Pi(t)\right] U(t), \tag{3.27}$$

$$\mathcal{I}_k(\epsilon): \quad \Pi(t+\epsilon) = \Pi(t) - \epsilon F_k(t), k = 0, \ldots, n. \tag{3.28}$$

Because of the different magnitudes of the forces $\|F_k\|$ the integration scheme can be accelerated by using different step sizes for the different parts. Ideally the magnitude of the forces whose computation takes the most time should be the smallest. With this aspect and by means of a multiple time step integrator the calculation of the force that takes the longest time can be done least. One can construct an integrator recursively starting with $\mathcal{I}_0$ that neglects all quark forces $F_1 \ldots F_n$.

$$I_0(\tau, N_0) = \left\{ \mathcal{I}_0\left(\frac{1}{2}\epsilon\right) \mathcal{I}_U(\epsilon) \mathcal{I}_0\left(\frac{1}{2}\epsilon\right) \right\}^{N_0}, \epsilon = \frac{\tau}{N_0}. \tag{3.29}$$

Then the further integrators are defined by the recursive scheme such as

$$I_k(\tau, N_0, N_1 \ldots, N_k) = \tag{3.30}$$

$$\left\{ \mathcal{I}_k\left(\frac{1}{2}\epsilon\right) I_{k-1}(\tau, N_0, N_1 \ldots, N_{k-1}) \mathcal{I}_k\left(\frac{1}{2}\epsilon\right) \right\}^{N_k}, \epsilon = \frac{\tau}{N_k}. \tag{3.31}$$

The full integrator $I_n(\tau, N_0, \ldots, N_n)$ can be constructed by means of this recursion scheme. At the time interval

$$\epsilon_k = \frac{\tau}{N_k N_{k+1} \ldots N_n}, \tag{3.32}$$

the force F_k has to be computed. This means that F_0 is the most computed one whereas F_n is computed least. With respect to 3.13 and 3.14 there are two operators R_1 and R_2. Therefore the forces F_0, F_1 and F_2 exist. F_0 corresponds to the gauge part of the LQCD action and its computation takes fewest time. The calculation of F_2 which corresponds to R_2 is the most computationally intensive task.

3.5.5 Partial block decoupling with active links

With the factorisation of the quark determinant the different blocks Λ are decoupled as much as possible. Because the Dirac operator in this case uses only nearest-neighbour interaction the equation with respect to the operator $\hat{D}_\Lambda$ that leads to the quark force F_1 are independent and can be solved in parallel. To decouple the calculation of F_0 during the molecular dynamics evolution, only a subset of links called *active links* is updated whereas the other ones are fixed. Both points connected by an active link have to be in the same block. Furthermore only one endpoint can be at the interior boundary of the block. So the only part in the hamiltonian (3.4) that needs communication with other blocks during the integration of the equations of motion (3.5) is the one that comes from the operator (3.14) that leads to the force F_2. The construction of the integration scheme (3.30) is such that the calculation of the force F_2 needed for the update $\mathcal{I}_2$ corresponding to (3.28) is done least.

To ensure that all links are covered equally on average the gauge field $U(x, \mu)$ is translated by a random vector v at the beginning or at the end of each update cycle such as

$$U(x, \mu) \rightarrow U(x + v, \mu). \tag{3.33}$$

3.5.6 Description of one update cycle

(a) Like it is common for HMC algorithms initial momenta $\Pi(x, \mu)$ and pseudo-fermion fields $\phi_1(x)$ and $\phi_2(x)$ have to be generated with the probability proportional to e^{-H}. This can be done by generating a gaussian random field η and applying $\psi = D\eta$. For the pseudo-fermion field ψ_1 that breaks up into block fields this leads to

$$\psi_1 = \sum_{all \Lambda} \psi_\Lambda, \psi_\Lambda = \hat{D}_\Lambda \eta_\Lambda, \tag{3.34}$$

whereas ψ_2 is given through

$$\psi_2 = R_2 \eta_2. \tag{3.35}$$

It is sufficient to generate the momenta only on the active links because all other ones are not updated.

(b) Hamilton's equations of motion (3.5) have to be integrated for the active links and the accordant momenta. The integration scheme $I_2(\tau, N_0, N_1, N_2)$ defined through (3.30) is used. The forces F_0, F_1 and F_2 have to be calculated for this. Again, F_0 corresponds to the gauge action S_G of the hamiltonian (3.4), F_1 to the operator R_1 that is given by the sum of the decoupled, even-odd preconditioned block Dirac operators $\hat{D}_\Lambda$. For the computation of F_2 which corresponds to the operator R_2 the Dirac equation on the full lattice has to be solved. This is done with the Schwarz-preconditioned GCR solver described above.

(c) To get the desired probability distribution an accept-reject step is necessary. For this the values of the hamiltonian (3.4) before the update H and after the update H' are calculated. The probability to accept the new configuration is given by

$$P_{acc} = \min\left\{1, \mathrm{e}^{-\Delta H}\right\}, \quad \Delta H = H' - H. \tag{3.36}$$

If a configuration is not accepted the old one is then the new configuration.

3.6 Simulation with DD-HMC

After this brief description of the algorithm the way how to start a simulation is analysed. In the directory `main` starting from the source directory one can find both a set of example input files in the subdirectory `examples` and an explanation of the most important things in the file `README.run3`. Some parameters of the input file will be explained below. The expression `nmx` stands for 'maximal number of iterations' whereas `res` means 'required relative residue'.

- `bs_hmc`
  - defines the size of the blocks Λ for the factorisation of the quark determinant.
- `N0, N1, N2`
  - are the step numbers for the integrator corresponding to the forces F_0, F_1 and F_2 that are defined in (3.30).

- `nmx_acf, res_acf`
  - are used by the BiCGstab inverter to calculate the part of the action that corresponds to the operator R_1 defined in (3.13).
- `nmx_gcr_acf, res_gcr_acf`
  - are used by the Schwarz preconditioned GCR inverter to compute the action corresponding to the operator R_2 defined in (3.14).
- `nmx_rop, res_rop`
  - are used by the BiCGstab inverter when applying the operator R_2 to get pseudo-fermion field ψ_2 like it is defined in (3.35). This is done once for each trajectory.
- `nmx_frf, res_frf`
  - are used by a CG inverter to compute the force on the blocks Λ that leads to F_1.
- `nmx_gcr_frf, res_gcr_frf`
  - are used by the preconditioned GCR solver to solve the Dirac equation on the full lattice when computing the force F_2.
- `bs_sap`
  - defines the size of the blocks Σ for the Schwarz preconditioner of the GCR inverter.
- `nkv_sap`
  - is the maximal number of Krylov vectors n_{kv} before the GCR inverter is restarted.
- `nmr_sap`
  - defines the number of iterations n_{mr} for the minimal residual solver when inverting D_Σ to update ψ' as shown in (3.17).
- `ncy_sap`
  - is the number of Schwarz cycles n_{cy} defined in (3.20).

3.7 Binary storage of gauge configurations

The default storage precision of the gauge fields is double. The configurations are saved with the endianness of the machine. The functions for the input and output are located in the file `modules/misc/archive.c`. The structure of the binary gauge configurations is also explained there. Beside the gauge fields, each file contains the dimensions of the lattice and the value of the plaquette of the corresponding configuration. At first the dimensions of the lattice L_0, L_1, L_2, L_3 are written, then the plaquette is the next value. After this the gauge configuration is written to the output file.

The storage order in the output file is done in another way in comparison with the ILDG format which is explained in section 2.1.3. For a lattice site not only the four links in positive direction are saved. All eight links that start or end respectively at this site are stored in succession. Therefor, it is sufficient to have a look at only the half of all sites. In this case the eight links corresponding to the odd lattice sites are stored successively. The storage order is given by

1. Site index in time-direction x_0 $(x_0 = 0, \ldots, L_0 - 1)$
2. Site index in space-direction x_1 $(x_1 = 0, \ldots, L_1 - 1)$
3. Site index in space-direction x_2 $(x_2 = 0, \ldots, L_2 - 1)$
4. Site index in space-direction x_3 $(x_3 = 0, \ldots, L_3 - 1)$
5. Direction index $\mu(\mu = 0, \ldots, 7; 0 \rightarrow +t, 1 \rightarrow -t, 2 \rightarrow +x$
   $3 \rightarrow -x, 4 \rightarrow +y, 5 \rightarrow -y, 6 \rightarrow +z, 7 \rightarrow -z)$
6. Colour index a $(a = 0, 1, 2)$
7. Colour index b $(b = 0, 1, 2)$
8. Index corresponding to real or imaginary part $(0 \rightarrow \text{Re}, 1 \rightarrow \text{Im})$

whereas x_0, x_1, x_2, x_3 has to be considered on only odd lattice sites.

3.8 Summary of the experiences with the DD-HMC simulation software

The domain decomposition hybrid monte carlo simulation software is written in C. It provides a simulation algorithm using two mass degenerate Wilson quarks but there are no methods for measurements for LQCD. Because of this clear task it is possible to understand a lot of the structure of the software in a short time. The necessary functions are located in a manageable number of directories. A good overview of these functions is given

by means of several `INDEX` files. Furthermore, there are `README` files in the corresponding subdirectories that explain the functionality of the methods in a short way.

Machines with current Intel and AMD processors can use the inline assembly written SSE optimisations to build efficient code.

Some tasks of DD-HMC are not completely independent of the used computer. For instance the storage of the gauge configurations depend on the endianness of the machine. Configurations that have to be exchanged between computers with different endianness have to be swapped in all bytes.

It could be necessary to change the code for parallel machines which do not use a common MPI standard. It is not possible to replace just a sub-package of the software because the MPI communication routines are integrated in several functions of the code.

For the compilation of DD-HMC it is necessary to determine some global variables such as the dimensions of the lattice and the dimensions of the processor grid of the parallel machine. For every change of these variables the code has to be compiled.

The DD-HMC code provides efficient simulation software. If one wants to have more comprehensive software that can also simulate LQCD with further actions or if one wants to do certain measurements one has to extend the DD-HMC code. But this should be possible because of the manageable structure of the software.

Chapter 4

Benchmarks

4.1 Opteron PC-Cluster

The Opteron PC-Cluster of the Department of Physics of the Humboldt University consists of 34 computing nodes. Each node contains two AMD Opteron processors located on the same board. The two processors share a total memory of 4GB. Each of these 64 Bit processors has a clock frequency of 2.4 GHz. The first level cache (L1) as well as the second level cache (L2) are on-chip. The L1 cache has a size of 64 KB for instructions and 64 KB for data whereas the L2 cache has a size of 1024 KB. The processors support the SSE and SSE2 instruction set and have 16 MMX 128 Bit registers for doing so. The computing nodes are connected through an infiniband network. Two central switches connect all computing nodes with each other, i.e., the network is not optimised for nearest-neighbour communication but sharing the data is organised centrally over these two switches. The switches are connected through three infiniband cables with each other. Thus the communication can interfere if more than three nodes that are connected over the one switch have to communicate with nodes that are connected over the other switch. In practice the machine is in use by other parallel jobs as well as single node jobs. Communication between nodes and data I/O is going through the network, i.e. if the PC-cluster is used by other jobs that are noncontrollable it is hard to do reproducible benchmarks.

4.2 Comparison of different compiler flags

Both the simulation with dynamical fermions and the measurement in LQCD may take a lot of computing time. The most time intensive task is solving the Dirac equation $D\psi = \eta$ which is usually done with iterative solvers that have to apply the Dirac operator D to a spinor ϕ several times. The efficiency of the whole simulation or measurement algorithm strongly depends on the efficiency of the application of the Dirac operator. On a

single node system the global lattice is kept by one processor. On parallel machines each computing node uses a sub-lattice, the local lattice of the node. Thus, both the calculation on the local lattice and the communication of the several nodes have to be done in an efficient way.

The optimisation of the code is completely done by the compiler when using higher level languages like C, C++ or FORTRAN to implement an algorithm. By means of inline assembly mechanisms the optimisation can be controlled and possibly enhanced.

In the following section a comparison of the application of the Wilson-Dirac operator with different compiler flags is shown for the Chroma software system. Furthermore, the advantages of the inline assembly SSE optimised Wilson-Dirac operator are presented.

This time measurements were done on a single node machine. This machine has the same hardware components like the PC-Cluster described above. During the measurement it was ensured that one of the two processors on the mainboard idled to avoid interference on the busses or the memory. The time measurements were done with QDP++ version 1.20.2 and with Chroma version 2.15.3. For this the provided test program `t_lwldslash_sse` which can be found in `$CHROMA_SRC/mainprogs/tests` was used. The binary can be found in `$CHROMA_INSTALL/bin`. In this program the operator is first applied on the sub-lattice of one checkerboard colour and then on the other one. The time is measured separately and the printed to the standard output. The total execution time is the sum of the times of both checkerboard sub-lattices. The figures 4.1 and 4.2 illustrate the execution time per floating operation. Furthermore, the triangular datapoints in the figures in this section correspond to the compiletime options that were suggested in the install instruction for the scalar build of the Chroma software system [9]. These options for the build of QDP++ and Chroma are shown in section 2.2.1.

The execution time was measured in dependence of the volume of the lattice varying from $V = 2 \times 2 \times 2 \times 2$ to $V = 32 \times 32 \times 32 \times 32$. For each measurement the volume of the lattice was doubled. For each lattice with inhomogeneous lattice dimensions, e.g., $x = y = z \neq t$, all permutations keeping the volume fixed were measured separately, i.e., it was distinguished between the layout $V = 32 = 2 \times 2 \times 2 \times 4$ and $V = 32 = 2 \times 2 \times 4 \times 2$, for instance. Furthermore, the program was executed several times. The values shown in the plots refer to the minimal execution time of both, all reruns of the executable and all permutations of the dimensions. In this way it was possible to get fairly reproducible data. It is a little bit dubious what the operating system of a computer is doing all the time and how influential this is to accurate time measurements. It is assumed that one gets the most meaningful execution time if the operating system does least. This means that no statistical analysis of the data was made, neither mean values nor statistical errors were calculated. The problem of an inter-

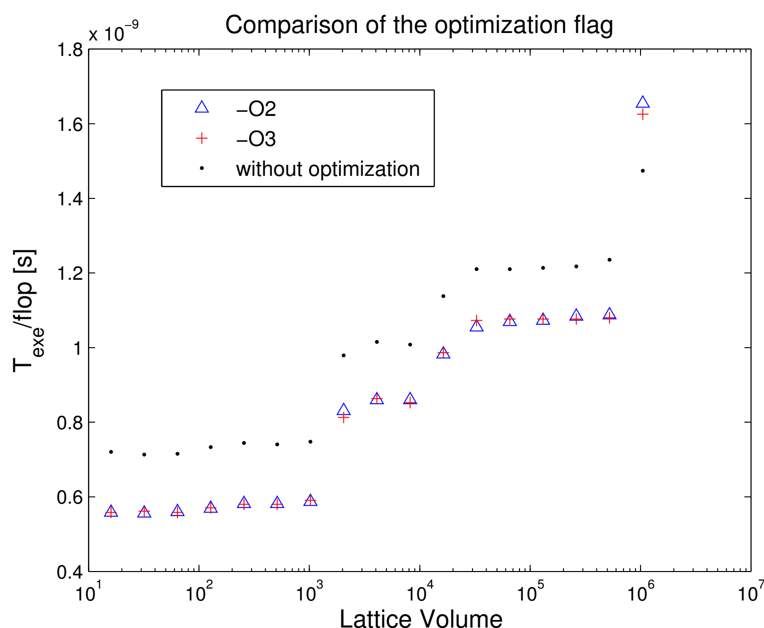

Figure 4.1: Comparison of the compiler optimisation flag. The triangles are for the optimisation level 2, the crosses for level 3 and the dots have no optimisation.

fering operation system is more a systematic error because it has a unique direction but it is difficult to estimate this error.

Figure 4.1 shows that it is advantageous to use the optimisation options `-O` of the compiler to build efficient code. It is not surprising that it is better to use one these optimisation levels than using non. However, one can see that it not necessary to use higher level optimisations since the differences between `-O2` and `-O3` are negligible.

In figure 4.2 the differences between SSE optimisation attempts of the compiler and the inline assembly optimisations are shown. It is obvious that the compiler has not the possibilities to use the advantages of SSE.

4.3 Analysis of implementations of the Wilson-Dirac operator

4.3.1 Introduction

As mentioned in section 4.2, the most computationally intensive task for LQCD simulations and measurements is the application of the Dirac operator. A theoretical analysis of the execution time of this task can be helpful to improve the efficiency of software for LQCD.

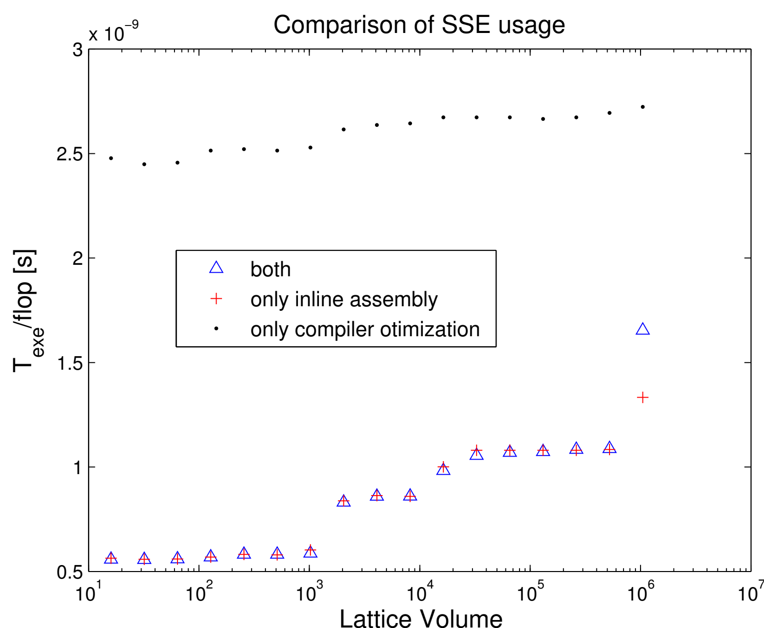

Figure 4.2: Comparison of different SSE usages. The crosses show the inline assembly written optimisation, the dots show the attempt of the SSE optimisation by compiler flags and the triangles show both optimisations

In this section different implementations of the Wilson-Dirac operator for LQCD are analysed. An implementation means the translation of a computational task, like the application of the Dirac Operator, into an executable code. For this, different levels of programming can be considered. At a high level, the programmer translates the computational tasks into a high level language like C or C++ and with assembly to have more control of the program. At a low level, the compiler does the translations.

A common computer architecture has different layers for the storage of data. The smallest storage system is the register of the processor. The next layer consists of the cache system. Different processors have a different number of cache levels (the used Opteron processor has L1 and L2 cache). The third layer of storage systems is the memory (RAM). The bandwidth of a storage system increases if it is closer to the processor whereas the storage size usually decreases.

For the computation of the Wilson-Dirac operator the storage requirement of the different levels is computed. Furthermore, the information exchange between the different levels is calculated. If the execution time of a code is limited by the bandwidth of the data paths between the storage systems, the execution time is proportional to the information exchange.

A general discussion of the methodology of performance modelling can be found in [23–25].

4.3.2 Definitions

Information exchange and storage requirement

For a given implementation i of a problem of the size n the data that is necessary to exchange in both directions between two sub-systems X and Y of a computer is given by the function

$$I_{XY}(n,i) = I_{X\to Y}(n,i) + I_{Y\to X}(n,i)\,, \tag{4.1}$$

where $I_{Y\to X}(n,i)$ means the transfer of data from the system Y to X. The second value that can be important for the execution time of a computational task is the required storage $S_X(n,i)$ in the sub-system X.

The information exchange function $I_X(n,m)$ [26] is independent of the implementation i. It is defined as the number of bytes that are necessary to exchange between a sub-system X of the computer and the rest of it to solve a given problem of size n if the sub-system X has a storage size m. If X is a sub-system of a computer and $\bar{X}$ is the rest of the computer, the information exchange function is given by

$$I_X(n,m) = \min_{\{i:S_X(n,i)\leq m\}} I_{X\bar{X}}(n,i)\,. \tag{4.2}$$

Possible sub-systems on a computer are, for instance, processors (P), registers (R), cache (C) or memory (M).

4.3.3 Performance and execution time

The performance for an implementation on a hardware architecture depends on the execution times T_{XY} of the different data exchange between the systems X and Y over the connecting data paths. These paths are characterised by the bandwith β_{XY} and the latency λ_{XY}. Furthermore, each sub-system has a storage size σ_X. The arithmetic operations on a processor can be described with this model, too. Here, I_{RR} is the number of needed instructions for the implementation and β_{RR} is the throughput of the arithmetic units of the processor depending on its clock frequency. The estimated execution time for the data exchange between the sub-systems X and Y is thus given by

$$T_{XY} \approx \frac{I_{XY}(n,i)}{\beta_{XY}} + O(\lambda_{XY})\,, \tag{4.3}$$

where the implementation i has to fulfil

$$S_X(n,i) \leq \sigma_X. \tag{4.4}$$

The upper bound for the total execution time T_{exe} for solving a problem by means of the implementation i is given by

$$T_{exe} \leq \sum_{X,Y} T_{XY}. \tag{4.5}$$

If the hardware supports full concurrency i.e. if all data exchange can be done in parallel and if all latencies are nonexistent respectively are hidden by pipelining and prefetching[1], the total execution time is given by

$$T_{exe} \sim \max_{X,Y} T_{XY}. \tag{4.6}$$

Operation count of the Wilson-Dirac operator

The Wilson-Dirac operator $M = 1 - \kappa H$ is introduced in appendix A.2 where H is here the hopping term. To count the operations when applying the Dirac operator M to a spinor ψ the representation of the gauge fields $U(x,\mu)$ and the spinors ψ is mentioned. At each lattice site lives a spinor with 4 spin $\times$ 3 colour components, i.e., a spinor consists of 12 complex numbers. The gauge fields $U(x,\mu)$ in each direction $\mu = 0,1,2,3$ are represented by $SU(3)$ matrices consisting of 9 complex numbers.

Each multiplication $\mathcal{M}$ or addition $\mathcal{A}$ of two floating point numbers is counted as a floating point operation. A multiplication with ± 1 or with the complex unit i is not counted. As all fields are complex, it is necessary to have a look at the complex multiplications and additions:

$$\text{complex multiplication} : 4 \times \mathcal{M} + 2 \times \mathcal{A}, \tag{4.7}$$

$$\text{complex addition} : 2 \times \mathcal{A}. \tag{4.8}$$

For the multiplication of a $SU(3)$ matrix with a colour vector 6 complex additions and 9 complex multiplications are needed. With respect to (4.7) one gets

$$A_{SU(3)} \times \phi_{colour} : 30 \times \mathcal{A} + 36 \times \mathcal{M}. \tag{4.9}$$

To get the solution (spinor with 4 spin $\times$ 3 colour components) of one part of the Wilson-Dirac operator given by

$$\left(1 - \gamma_\mu\right) U(x,\mu)\, \psi(x + \hat{\mu}), \tag{4.10}$$

[1] An operation, e.g. the multiplication of two floating point numbers, is done in several steps. First, the command for the operation has to be fetched. In the second step the numbers are loaded from the registers. Then the operation is executed and in the last step the solution is written in the register. This approach is called pipelining. In every clock cycle of the processor one of these steps can be done. This can have advantages if the multiplication of two N-dimensional array is done, for instance. With a filled pipeline it is possible to obtain one result of the N multiplications each clock cycle although the the operation takes at least four cycles. Often it is only possible to fill the pipeline if the needed data is prefetched, i.e. loading the data from the memory before it is necessary. Therefore, it is available when it is needed.

the advantage of the projections are used to save floating point operations and data exchange in the communications (further details can be found in appendix A.3). Two colour vectors for two spin components in each case have to be added for this. This leads to

$$2 \times [6 \times \mathcal{A} + 30 \times \mathcal{A} + 36 \times \mathcal{M}] \,. \tag{4.11}$$

The hopping term H of the Wilson-Dirac operator contains two terms corresponding to (4.10). These are summed over $\mu = 0, 1, 2, 3$. In total, H consists of 8 accordant terms (spinors with 12 components) that have to summed up. Finally one gets

$$H : 8 \times [72 \times \mathcal{A} + 72 \times \mathcal{M}] + 2 \times 7 \times 12 \times \mathcal{A}, \tag{4.12}$$

for the hopping term H. For the full Dirac operator M, the multiplication with the hopping parameter κ which is a real number, has to be done as well as the complex addition with a spinor with 12 components, i.e., $2 \times 12 \times \mathcal{A} + 2 \times 12 \times \mathcal{M}$ have to be done in addition. This leads to 1320 floating point operations for the hopping term H and 1368 floating point operations for the Wilson-Dirac operator M.

Lattice geometry

LQCD is formulated on a finite lattice with three dimensions in space and one dimension in time. The total volume of the global lattice is given by

$$V = L_t \times L_x \times L_y \times L_z. \tag{4.13}$$

On a parallel machine with N_P processors the lattice is separated into sublattices. Each processor works on a local lattice with the volume

$$v = l_t \times l_x \times l_y \times l_z, \tag{4.14}$$

whereas the relation $v = \frac{V}{N_P}$ is valid. The surface of the local lattice is given by

$$a_+ \equiv v \sum_{\{k : l_k < L_k\}} \frac{1}{l_k}, \tag{4.15}$$

where a_+ denotes only the nearest-neighbour sites on the faces in the positive direction, i.e. in total there are $2a_+$ nearest-neighbour sites outside one local lattice.

General remarks

In the following the unit *cword* is introduced for one complex number consisting of two floating point numbers. Furthermore, for the required storage of a sub-system S_X as well as for the data exchange I_{XY} the values

$$|\psi| = 12cword, \tag{4.16}$$

$$|U| = 9cword, \tag{4.17}$$

are introduced.

For the further discussion one has to distinguish between the computation of the hopping term H and the complete Dirac operator M. The data exchange of these two operators is different because for H only the exchange of the solution is considered whereas for M the loading of the ψ field on the central site is necessary in addition. For a general discussion, the parameter

$$I_0 \equiv \begin{cases} 1|\psi| & = 12 \quad cword \quad \text{for} \quad H \\ 2|\psi| & = 24 \quad cword \quad \text{for} \quad M, \end{cases} \tag{4.18}$$

is introduced.

4.3.4 Single node implementation

In the following discussion only the data exchange between memory and cache I_{CM} is considered because this data path is the slowest in this task. Thus, the execution time of the code will be dominated by the data exchange I_{CM}.

To compute the hopping part H it is necessary to read 8 ψ and 8 U fields corresponding to the positive and the negative direction in each dimension. Each of these fields are accessed two times for the computation of H. In total when calculating M each ψ field is accessed three times because of the central term. Due to the reuse of many fields it is profitable to use faster storage systems than memory such as the cache of the processor. Fields that have been loaded into cache can be kept there until they are not needed anymore.

Implementation with maximal cache use

The optimal case is that the storage size of the cache σ_C is bigger than the size of all ψ fields and U fields of the full lattice. The result of the operation is not kept in cache in the optimal case but written directly to memory. To consider this the term

$$S_0 = \begin{cases} 0 & \text{for optimal cache use} \\ |\psi| & \text{for the storage of the result} \end{cases} \tag{4.19}$$

is introduced. Thus, the storage requirement is given by

$$S_C/v = S_0 + 1|\psi| + 4|U| = S_0 + 48cword. \tag{4.20}$$

The data exchange between the memory (M) and the cache (C) contains one additional $|\psi|$. That corresponds to the result of the application that has to be written in the memory.

$$I_{CM}/v = (1+1)\,|\psi| + 4|U| = 60cword \tag{4.21}$$

Implementation with "disjoint" blocks

In practice, the storage size of the cache σ_C is often not large enough to hold all spinor and gauge fields of the system. In this case at least a partial caching might be exploited. It is not known how this is actually done by the Chroma software system but it is tried to model a behaviour which keeps sub-lattices (blocks) in cache. Such a situation might arise, for instance, if the fields are arranged as multi-dimensional arrays and the handling of the cache by hardware yields to a caching of lower dimensional sub-arrays. It is assumed that these blocks are disjoint, i.e., the cached data from other blocks is not reused.

The volume of one block is given by

$$v' \equiv l'_X \times l'_Y \times l'_Z \times l'_T, \tag{4.22}$$

where $l'_X \ldots l'_T$ is the the block size in the corresponding dimension. In theory these sizes are arbitrary but in this discussion only full blocks are considered, i.e., either $l'_i = 1$ or $l'_i = l_i$ whereas l_i is the size of the local lattice on the processor in dimension i. The number of dimensions in the block with extension $l'_i > 1$, i.e., $l'_i = l_i$ in the case of full blocks, is introduced with $d = 1, 2, 3, 4$.

As in the case of equation (4.15) the surface in positive direction of one block is given by

$$a'_+ = v' \sum_{\left\{k : l'_k < l_k\right\}} \frac{1}{l'_k}. \tag{4.23}$$

The expression simplifies in the case of full blocks and is given by

$$a'_+ = v'\,(4 - d)\,. \tag{4.24}$$

For $d = 4$ the face in positive direction is zero, $a_+ = 0$. This is equivalent with maximal cache use described in the previous subsection. In the case $d < 4$, it is not necessary to keep all U fields corresponding to one site inside the cache. The fields that connect sites of two different blocks with each other are needed only once when computing on one block. They are needed

the next time when computing on the other block. The number of links per volume that concerns this fact corresponds to a_+ and is given by $(4-d)$. This leads to less storage requirement and data exchange. Implementations that do not uses these advantages are here characterised as "naive" caching.

In comparison to the maximal cache use (equation 4.20) the storage requirement is thus given by

$$S_C/v' = S_0 + 1|\psi| + (4-4+d)\,|U| \quad (4.25)$$
$$= S_0 + 1|\psi| + d|U| \quad (4.26)$$
$$= S_0 + (12+9d)\,cword. \quad (4.27)$$

The data exchange per volume of the block between the memory and the cache gets bigger with decreasing d. Since it is necessary to load the ψ fields of all nearest neighbours outside the block $(v' \cdot 2 \cdot a_+)$. The U fields which connect the actual block with the other ones and which are not saved on sites inside the block $(v' \cdot a_+)$ have to be loaded in addition, too. Thus, the data exchange is given by

$$I_{CM}/v' = I_0 + (1+8-2d)\,|\psi| + (4+4-d)\,|U| \quad (4.28)$$
$$= I_0 + (9-2d)\,|\psi| + (8-d)\,|U| \quad (4.29)$$
$$= I_0 + (180-33d)\,cword. \quad (4.30)$$

The following table gives an overview to the different cases of $d = 1, \ldots, 4$.

block	d	l'_X	l'_Y	l'_Z	l'_T	v'	a'_+	$S_C/v' - S_0$	$I_{CM}/v' - I_0$
full line	1	l_X	1	1	1	l_X	$3v'$	$1\|\psi\| + 1\|U\|$	$7\|\psi\| + 7\|U\|$
full plane	2	l_X	l_Y	1	1	$l_X l_Y$	$2v'$	$1\|\psi\| + 2\|U\|$	$5\|\psi\| + 6\|U\|$
full cube	3	l_X	l_Y	l_Z	1	$l_X l_Y l_Z$	v'	$1\|\psi\| + 3\|U\|$	$3\|\psi\| + 5\|U\|$
full volume	4	l_X	l_Y	l_Z	l_T	$l_X l_X l_Y l_Z$	0	$1\|\psi\| + 4\|U\|$	$1\|\psi\| + 4\|U\|$

For each value of d exist a corresponding volume v', a surface a'_+ a storage requirement S_C and a data exchange I_{CM}. In the table above the order of the dimensions $l'_0, \ldots, l'_3$ is just one example. To get the volume v', all other permutations for the values of l'_i are also valid.

4.3.5 Time measurement of the Wilson-Dirac operator on a single node

The time measurement of the Wilson-Dirac operator was done with the Chroma software system. The same test program as in section 4.2 was used. The single node machine was also the same. To obtain reproducible data the mechanism was used which was described in section 4.2. The compiler options which were provided in section 2.2.1 are chosen, i.e., the options which corresponds to the triangular data in figure 4.1 and figure 4.2.

The theoretical data are from the model described above. The block partition is done obeying the condition

$$S_C/v' \le \sigma_C = 1024\,\text{Kbyte}. \tag{4.31}$$

Furthermore, the differentiation between optimal and naive caching is done. Thus the storage requirement is assumed to be

$$S_C/v' = S_0 + \left\{ \begin{array}{ll} 1|\psi| + d|U| & \text{(optimal)} \\ 1|\psi| + 4|U| & \text{(naive)} \end{array} \right. , \tag{4.32}$$

whereas the data exchange is given by

$$I_{CM}/v' = I_0 + \left\{ \begin{array}{ll} (9-2d)\,|\psi| + (8-d)\,|U| & \text{(optimal)} \\ (9-2d)\,|\psi| + 8|U| & \text{(naive)} \end{array} \right. . \tag{4.33}$$

The theoretical execution time T_{exe} is assumed to be given by equation (4.6), i.e. all processes occur in concurrency and the data path between the memory and the cache is the slowest in this task. Thus, the execution time depends on the bandwidth between these sub-systems. In figures 4.3, 4.4 and 4.5 the execution time per flop was plotted (T_{exe}/flop). With respect to the operation count of the Wilson-Dirac operator described in section 4.3.3, this differs by a factor 1320 in coparison to the execution T_{exe}/V which corresponds to (4.33).

The data exchange and thus the execution time gets larger for decreasing d. In figures 4.3, 4.4 and 4.5 one can see steps corresponding to a smaller number of d. The lowest execution time corresponds to $d = 4$, i.e. the full lattice can be kept in cache. For each further step, d decreases to the next smaller integer value.

For the bandwidth $\beta_{CM} = 1.33\,\text{Gbyte/s}$ the measured execution time is bigger than the execution time in the model for each volume (figure 4.3 and figure 4.4), i.e. this bandwidth is a lower bound and the real one has to be bigger.

The comparison between figure 4.3 and figure 4.4 shows that there are no differences if the value for S_0 defined in equation (4.19) is changed from $S_0 = 0$ cword to $S_0 = 12$ cword. The model agrees with the measurement for all volumes but $V = 32 \times 32 \times 32 \times 32$. For this volume the measurement shows a gap which could correspond to $d = 1$. With the chosen parameters $S_0 = 0$ cword and $S_0 = 12$ cword, the model does not show these characteristics. Assuming that the use of the cache is less optimised, the value for S_0 could be larger. The assumption for $S_0 = 24$ cword is shown in figure 4.5. It is possible to enforce the last gap, i.e., the case $d = 1$, but then one can see that the model does not agree with the measurement in the case $V = 8 \times 8 \times 4 \times 4 = 1024$. The measured execution time is smaller than the one for the optimal caching in the model.

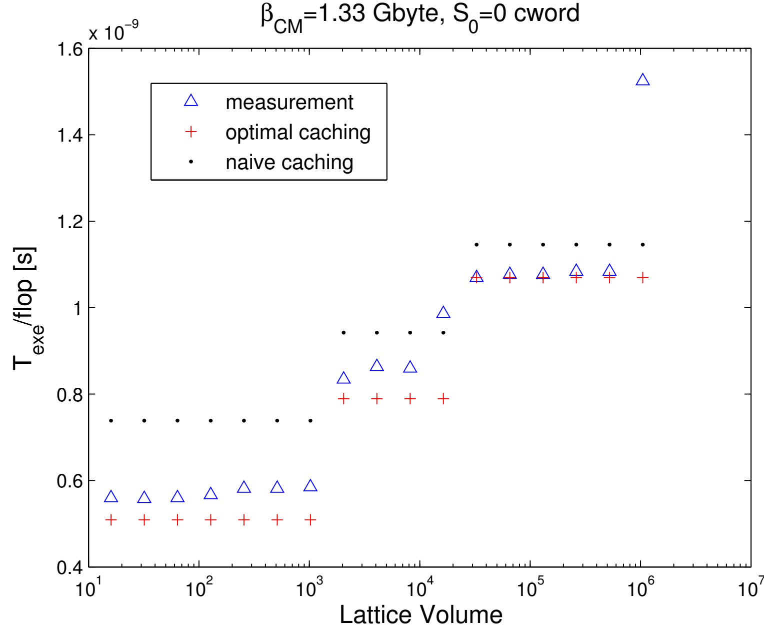

Figure 4.3: Execution time of the measurement and the model for $S_0 = 0$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s.

The implementation of the Wilson-Dirac operator in the Chroma software system was not analysed in details. It could be a combination between the optimal case of caching and the naive implementation. Some characteristics of the measurement are not understood yet. Nevertheless, one can see that the model agrees with the time measurement for a long range of different volumes.

4.3.6 Multi node implementation

On parallel systems with N_P processors the global lattice V (4.13) is split into local lattices $v = \frac{V}{N_P}$ (4.14). Because of the nearest neighbour interaction of the Wilson-Dirac operator the data exchange between different local lattices corresponds to the sites on the surface of each local lattice. The number of nearest neighbour sites in the positive direction is given by a_+ defined in equation (4.15). a_+ corresponds to the local lattice of one processor. The application of the Wilson-Dirac operator on a local lattice needs the ψ fields of all nearest neighbour sites outside the local lattice. Furthermore, the U fields which connect these sites are necessary. The needed data exchange for the computation of the Dirac operator on one local lattice between processor P and the the rest of the parallel machine $\bar{P}$ is given by

$$I_{\bar{P}P}/N_P = 2a_+|\psi| + a_+|U|. \tag{4.34}$$

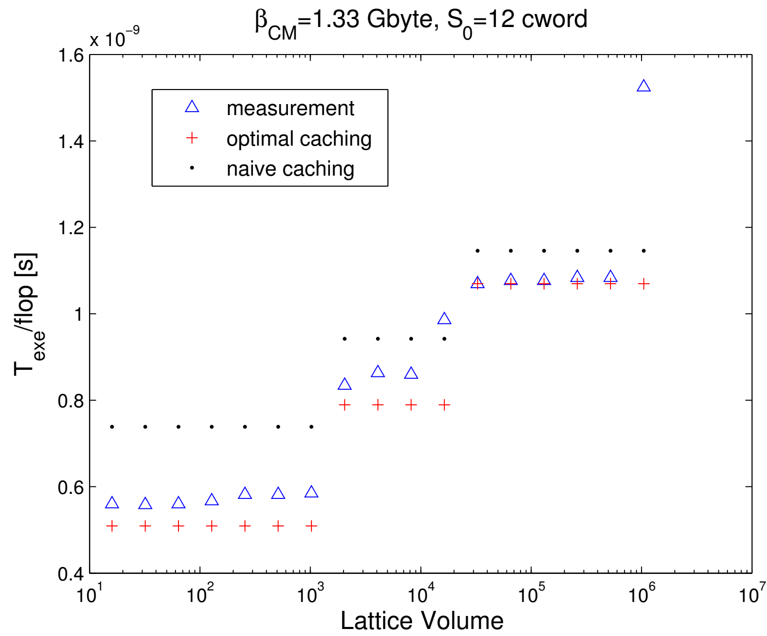

Figure 4.4: Execution time of the measurement and the model for $S_0 = 12$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s.

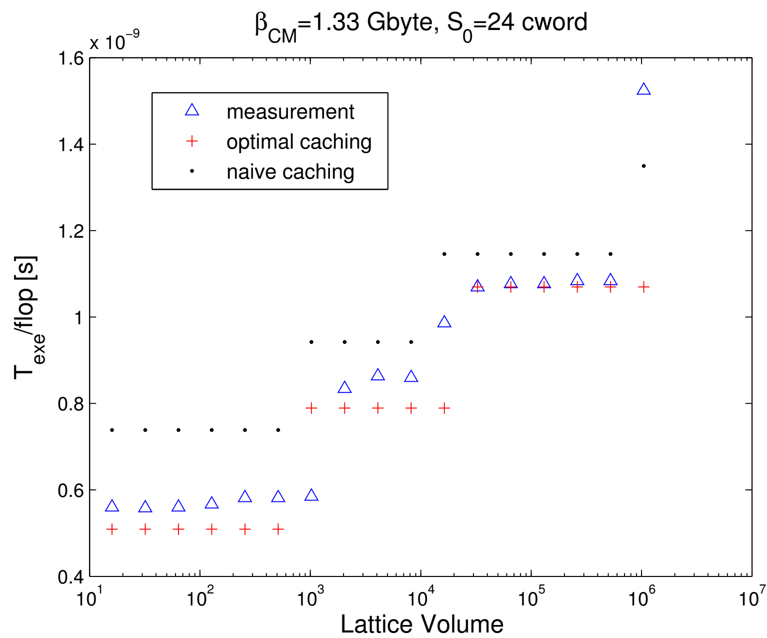

Figure 4.5: Execution time of the measurement and the model for $S_0 = 24$ and (the lower bound for the bandwidth) $\beta_{CM} = 1.33$ Gbyte/s.

Half of the U fields that connect nearest neighbours outside the local lattice are stored inside the local lattice corresponding to the sites on the surface. Therefore, the data exchange $I_{\bar{P}P}$ considers only half of the U fields. With respect to (4.14), equation (4.34) is given by

$$\begin{aligned} I_{\bar{P}P}/V &= \frac{N_P}{V} a_+ \left(2|\psi| + |U|\right) \\ &= \frac{1}{v} a_+ \left(2|\psi| + |U|\right) \\ &= \left(2|\psi| + |U|\right) \sum_{\{k: l_k < L_k\}} \frac{1}{l_k}. \end{aligned} \tag{4.35}$$

The execution time T_{exe} for this application is still given by (4.5) or (4.6) depending on the possibility of full concurrency when applying the Wilson-Dirac operator on a parallel machine. To calculate the total execution time of the full application one has to consider that N_P processors are computing concurrently. Therefore, in the following discussion the execution time T_{exe} is assumed to be

$$\frac{T_{exe}}{V \cdot N_P} = \frac{I_{\bar{P}P}}{V \cdot \beta_{XY}}. \tag{4.36}$$

In figure 4.6 and figure 4.7 the execution time T_{exe}/flop was plotted. This time differs by a factor 1320 in comparison to T_{exe}/V with respect to the opearation count of the Wilson-Dirac operator (section 4.3.3).

Furthermore, it is important which data path is the slowest in this task, i.e. which one dominates the execution time. The infiniband network has a bandwidth up to 0.9 Gbyte/s for the exchange of huge data packages but it is not sure that this bandwidth can be reached during the application of the Wilson-Dirac operator. The data exchange over the network $I_{\bar{P}P}$ is fewer than the exchange from memory to cache I_{CM} on each local machine. Thus, it is not obvious which data path will dominate the total execution time.

The time measurements were done on the Opteron PC-Cluster described in section 4.1.

Two different global volumes $V = 1024$ and $V = 2048$ with different numbers of processors were used. The partition of the local lattices and the value for $\sum_{\{k:l_k<L_k\}} \frac{1}{l_k}$ necessary for the computation of $I_{\bar{P}P}$ is shown in the table 4.1.

The execution time for $V = 8 \times 8 \times 4 \times 4 = 1024$ and $V = 8 \times 8 \times 8 \times 4 = 2048$ on $N_P = 1, 2, 4, 8, 16$ processors is shown in figure 4.6 and figure 4.7, respectively. The time for the application of the Wilson-Dirac operator on the local lattice with the volume $v = \frac{V}{N_P}$ is plotted (optimal caching) as well as the time corresponding to the data exchange between the processors over the infiniband network. For the bandwidth between memory and cache the same value as in section 4.3.5 was used, namely

V	L_X	L_Y	L_Z	L_T	N_P	l_X	l_Y	l_Z	l_T	$\sum_{\{k:l_k<L_k\}} \frac{1}{l_k}$
1024	8	8	4	4	2	8	4	4	4	1/4
1024	8	8	4	4	4	8	2	4	4	1/2
1024	8	8	4	4	8	8	2	4	2	1
1024	8	8	4	4	16	8	2	2	2	3/2
2048	8	8	8	4	2	8	8	4	4	1/4
2048	8	8	8	4	4	8	4	4	4	1/2
2048	8	8	8	4	8	8	2	4	4	3/4
2048	8	8	8	4	16	8	2	2	4	1

Table 4.1: Global volumes V for the application of the Wilson-Dirac operator on different numbers of processors. The dimensions of the global lattice are given by L_X, L_Y, L_Z, L_T whereas the dimensions of the local lattices are given by l_X, l_Y, l_Z, l_T.

$\beta_{CM} = 1.33$ Gbyte. To ensure that the measured time is not smaller than the time in the model, the bandwidth for the network was set to $\beta_{\bar{P}P} = 0.405$ Gbyte/s. This is a lower bound for the bandwidth of the infiniband network.

In figure 4.6 the execution time for the Wilson-Dirac operator is dominated by data exchange between $I_{P\bar{P}}$ for a number of processors $N_P > 4$. The measurement and the theory agree qualitative whereas the measured execution time for $N_P = 16$ is much greater than the theoretical one. This could be explained with interference in the network but this is not considered in the model.

In figure 4.7 the execution time for the application is also dominated by data exchange between $I_{P\bar{P}}$ for a number of processors $N_P > 4$. Not all measured data points agree with the model. In particular, the time is too large for $N_P = 2$ which can not be explained by the model. As in section 4.3.5 it is difficult to get reproducible data at this measurement. The problem is competing data in the network. The running program can interfere with itself as well as other tasks and it is hard to control the data communication at the network. Therefore, it is possible that the measured time for $N_P = 2$ can be much smaller without interference in the network.

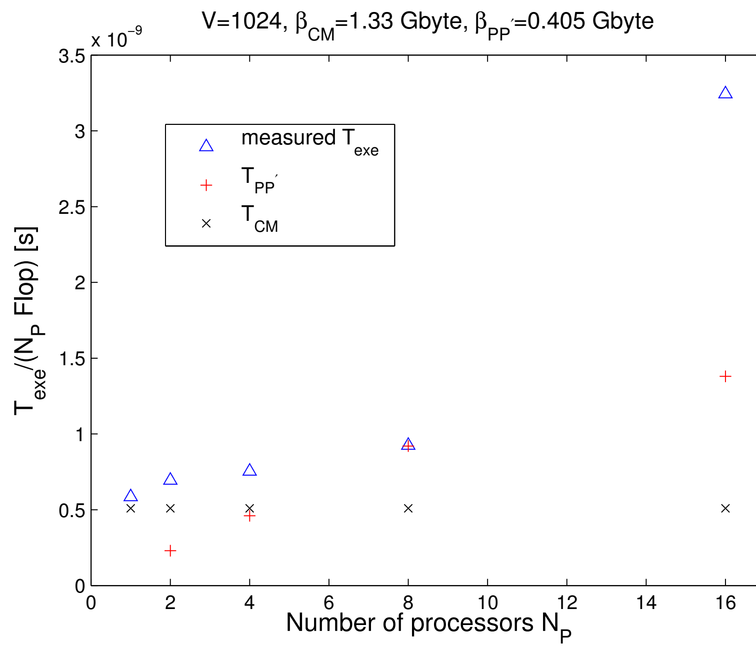

Figure 4.6: Execution time of the Wilson-Dirac operator for $V = 1024$ on multiple nodes. The bandwidth of the network $\beta_{\bar{P}P} = 0.405$ Gbyte/s is a lower bound. The triangles refer to the measured time, whereas the crosses show the execution time on a single processor with the corresponding local lattice. The model described in section 4.3.5 was used ($S_0 = 12$). The plus indicates the time that corresponds to the data exchange between the different processors.

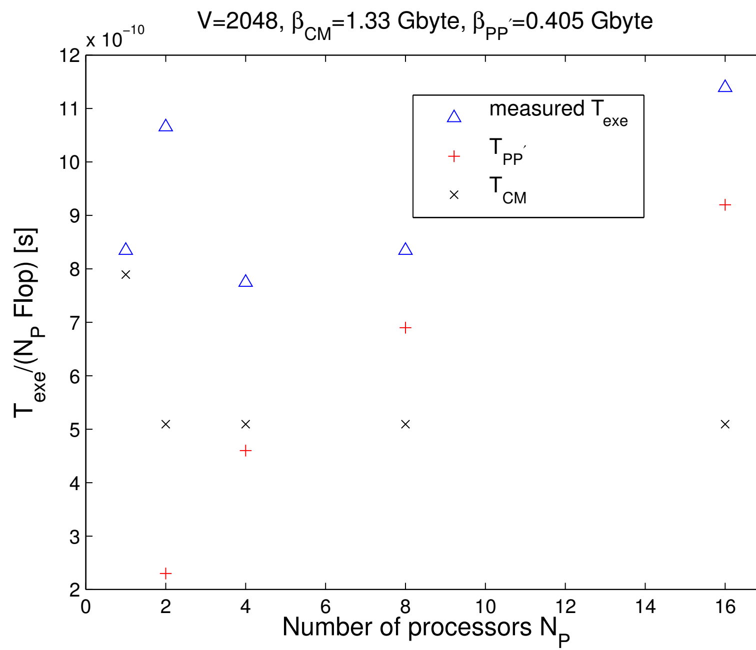

Figure 4.7: Execution time of the Wilson-Dirac operator for $V = 2048$ on multiple nodes. The bandwidth of the network $\beta_{\bar{P}P} = 0.405$ Gbyte/s is a lower bound. The triangles refer to the measured time, whereas the crosses show the execution time on a single processor with the corresponding local lattice. The model described in section 4.3.5 was used ($S_0 = 12$). The plus indicates the time that corresponds to the data exchange between the different processors.

Chapter 5

Effective Mass

Masses of hadrons are precisely known from experiments. One important test of the simulation of LQCD can be the determination of the these masses. The goal of this section is the determination of the effective mass of a pion. This mass cannot be compared with experiments because in the simulation a small volume and huge input parameters for the quark masses were used. But it is possible to compare the result with measurements of other simulations to check the correctness of our simulation and measurement. Both, the simulation and the measurement of the correlation functions were done with the Chroma software system on the Opteron PC-Cluster which is described in section 4.1. A detailed discussion of this task can be found in [27, 28].

5.1 Timeslice correlations of pions

A partial fourier transform $g(t, \mathbf{p})$ of an arbitrary function $f(t, \mathbf{x})$ is given by

$$g(t, \mathbf{p}) \equiv \sum_{\mathbf{x}} \mathrm{e}^{-i\mathbf{p}\mathbf{x}} f(t, \mathbf{x}) \,. \tag{5.1}$$

The lattice spacing is set to $a = 1$ for a simpler notation.

The four components of the momentum p consists of the spatial part $\mathbf{p}$ and the total energy E. For zero spatial momentum $\mathbf{p} = \mathbf{0}$ physical masses are given by

$$m = E(\mathbf{p} = \mathbf{0}) \,. \tag{5.2}$$

Therefore, it is possible to use timeslice correlations, with respect to equation (5.1), to determine masses.

The used action in this task is introduced in appendix A.5.1. The zero momentum operator carrying the quantum numbers to create a pion and

that was used in the measurements is given by

$$O_\pi(t) = \sum_{\mathbf{x}} \bar{\psi}(\mathbf{x},t)\,\gamma_5 \psi(\mathbf{x},t)\,. \tag{5.3}$$

Thus, the timeslice pion correlation function is given by

$$\begin{aligned} \Gamma(t) &= \left\langle O_\pi^\dagger(t)\, O_\pi(0) \right\rangle & (5.4)\\ &= \sum_{\mathbf{x},\mathbf{y}} \left\langle \mathrm{Tr}\left(\gamma_5 M^{-1}(\mathbf{x},0;\mathbf{y},t)\,\gamma_5 M^{-1}(\mathbf{y},t;\mathbf{x},0)\right)\right\rangle_{S_{\mathrm{eff}}}, & (5.5) \end{aligned}$$

wherein $M(x,y)$ is the Wilson-Dirac operator described in appendix A.2. Therefore, M^{-1} is the quark propagator. $\langle\rangle_{S_{\mathrm{eff}}}$ denotes the average over the ensemble of gauge configurations which are Boltzmann distributed corresponding to the effective action S_{eff} which is introduced in appendix A.2.

5.2 Extracting masses from timeslice correlations

The timeslice correlation function can be written as

$$\Gamma(t) = \left\langle O_\pi^\dagger(t)\, O_\pi(0) \right\rangle = \frac{1}{Z}\mathrm{Tr}\left(O_\pi^\dagger(0)\, \mathrm{e}^{-\hat{H}t} O_\pi(0)\, \mathrm{e}^{-\hat{H}(T-t)}\right), \tag{5.6}$$

wherein $\hat{H}$ is the Hamiltonian and the normalisation is given by

$$Z = \mathrm{Tr}\; \mathrm{e}^{-T\hat{H}}. \tag{5.7}$$

Assuming $\{|n\rangle\}$ is a set of eigenstates with

$$\hat{H}\,|n\rangle = E_n\,|n\rangle\,. \tag{5.8}$$

Inserting complete sets for the trace as well as intermediate states, equation (5.6) becomes

$$\Gamma(t) = \frac{1}{Z}\sum_{n,m} \left\langle n \left| O_\pi^\dagger(0) \right| m \right\rangle \langle m\,|O_\pi(0)|\,n\rangle\, \mathrm{e}^{-E_m t}\mathrm{e}^{-E_n(T-t)}. \tag{5.9}$$

For large values for t and $T-t$ only the states with the lowest energy have to be considered. Setting the vacuum energy equal to zero, $E_0 = 0$, this leads to

$$\Gamma(t) \propto \left(\mathrm{e}^{-tE} + \mathrm{e}^{-(T-t)E}\right), \tag{5.10}$$

wherein E corresponds to the lowest energy of the particle state in the sector with the used quantum numbers. In this case the quantum numbers for

a pion were considered, i.e. the energy in equation (5.10) can be set equal to the mass of the pion, $E = m_\pi$.

$$\Gamma(t) \propto \left(\mathrm{e}^{-tm_\pi} + \mathrm{e}^{-(T-t)m_\pi}\right) \tag{5.11}$$

Therefore it is possible to extract the mass m_π from the timeslice correlation function $\Gamma(t)$. Considering the function at time t_1 and t_2 it is possible to obtain the mass m_π numerically from the ratio

$$r_{t_1,t_2} = \frac{\Gamma(t_1)}{\Gamma(t_2)} = \frac{\mathrm{e}^{-t_1 m_\pi} + \mathrm{e}^{-(T-t_1)m_\pi}}{\mathrm{e}^{-t_2 m_\pi} + \mathrm{e}^{-(T-t_2)m_\pi}}. \tag{5.12}$$

The mass m_π depends on the lattice dimension T and the timeslice pair t_1, t_2 and is called effective mass.

$$m_\pi^{\mathrm{eff}} = m_\pi^{\mathrm{eff}}(t_1, t_2, T). \tag{5.13}$$

5.3 Numerical analysis

The simulation was done on a lattice with volume $V = 12 \times 12 \times 12 \times 16$ for $\beta = 5.6$ and $\kappa = 0.1575$. Several configurations were produced to ensure that the measurement can be done with a thermalized ensemble. The measurement of the correlation function (5.5) was done as it was introduced in section 2.3.1. To determine the effective mass of the pion, 800 gauge fields were analysed. The Monte Carlo time between two gaugefields is $\tau_{\mathrm{mc}} = 1$. The handling of such a number of configurations was described in section 2.4.3.

For the determination of the effective mass the ratio (5.12) with the timeslice pair $t, t+1$ was used. Further analysis was done with the Newton-Raphson method to find zeros of the function

$$h\left(m_\pi^{\mathrm{eff}}\right) = r_{t,t+1} - \frac{\mathrm{e}^{-tm_\pi^{\mathrm{eff}}} + \mathrm{e}^{-(T-t)m_\pi^{\mathrm{eff}}}}{\mathrm{e}^{-(t+1)m_\pi^{\mathrm{eff}}} + \mathrm{e}^{-(T-(t+1))m_\pi^{\mathrm{eff}}}} = 0. \tag{5.14}$$

The error analysis was done as described in [29] with the MATLAB code published by the author. For each value of t one gets the effective mass $m_\pi^{\mathrm{eff}}(t, t+1, T)$. The data of the analysis is shown in table 5.1. The autocorrelation time τ_{auto} is apparently huge because the measurement was done with successive configurations which are highly correlated. The result is plotted in figure 5.1. The dashed line shows the value of the pion mass $m_\pi = 0.3576(89)$ which can be found in [1]. It is for the used set of parameters β and κ and a lattice volume $V = 12 \times 12 \times 12 \times 32$. One can see the agreement within the errors for $m_\pi^{\mathrm{eff}}(t = 7, t+1 = 8, T = 16)$. On the one hand this shows the correctness of the measurement with the used parameters, on the other hand one can see that the lowest energy state is already obtained for this time t.

t	$m_\pi^{\text{eff}}(t, t+1, T)$	τ_{auto}
0	2.032(3)	5(1)
1	1.32(1)	15(5)
2	0.92(2)	21(8)
3	0.63(3)	27(10)
4	0.47(3)	32(12)
5	0.41(3)	37(14)
6	0.38(3)	39(14)
7	0.37(3)	35(13)

Table 5.1: Result of the measurement of the effective pion mass $m_\pi^{\text{eff}}(t, t+1, T)$. The value and its autocorrelation time are shown with the errors of the analysis.

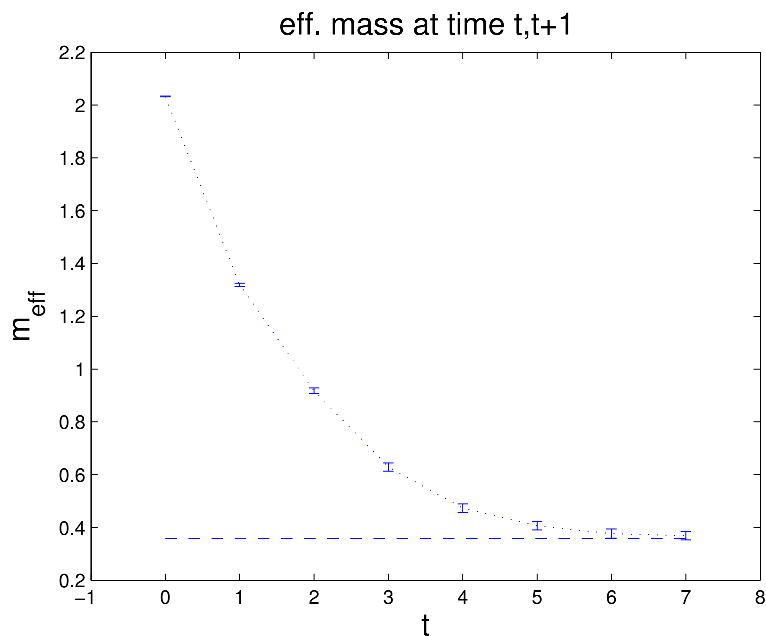

Figure 5.1: Effective mass of the pion for $\beta = 5.6, \kappa = 0.1575$ and $L_X = L_Y = L_Z = 12, L_T = 16$. The dotted line is to guide the eye, whereas the dashed line is the value for the pion mass for a lattice with $L_X = L_Y = L_Z = 12, L_T = 32$ [1].

Chapter 6

Conclusion

In the present work two Open Source software projects for LQCD applications were analysed, namely the Chroma software system and the Domain Decomposition Hybrid Monte-Carlo software. The installation on a single node as well as on a Opteron PC-Cluster was shown for both of them. The way to do first steps with the software was presented. Subsequently, it was explained how to perform simulations, i.e. how to generate gauge configurations. Furthermore, the measurement of hadronic 2-point functions with Chroma was introduced.

The Chroma software system provides a lot of applications for simulations and measurements in LQCD. Because of the different software modules of the package it is affirmed that the software is efficient and highly portable with respect to several computer architectures. The main introduction to the Chroma software system can be found in the tutorial [8]. The binaries of the application Chroma are controlled by XML input files. In the tar archive of Chroma, beside the source code several XML file examples could be found. But it was not possible to find a complete discussion of the whole functionality of the software system. Nevertheless, because of the installation guide for single node machines [9] and the tutorial [8] first steps with the software could be done after short while. For this, it was only necessary to be able to deal with XML input files. The understanding of details of the source code was difficult because of the complexity of the software system. Therefore, it would not be easy to enhance the software by own ideas.

The DD-HMC software uses one certain algorithm for the generation of gauge configurations. A short introduction of the algorithm was presented in section 3.5. Further information about this algorithm can be found in [21, 22]. The software provides no methods for measurements for LQCD, i.e. the functionality of DD-HMC is limited to the production of gauge configurations by means of the corresponding algorithm. This functionality and the usage of the software is described in the several `README` files lo-

cated in the tar archive beside the source code. The DD-HMC software is not as complex as the Chroma software system. Therefore, the structure of the software can be understood in a short while. It should be a possible task to do enhancements of the software if wanted. Such additional programmings are necessary if one wants to use the software for further LQCD applications apart from the generation of gauge configurations. Furthermore, if the software is applied, for instance, on a machine which does not use the common MPI standard, it will be necessary to modify several parts of the code because the MPI commands are integrated in several functions of the program.

The decision for a software depends on the project which has to be dealt with. The Chroma software system provides a library which can be used for several applications for LQCD. It is thinkable that this software system can be used to analyse configurations which were produced with some other software. For this, the gauge configurations have to be stored in a file format which can be handled by Chroma, e.g. ILDG files generated with the LIME package introduced in section 2.1.3. For instance, if configurations are used which were generated with the DD-HMC software and stored as described in section 3.7, it is not sufficient to convert the configurations in the ILDG storage order described in section 2.1.3 to import them into Chroma. An ILDG file, for instance, contains some metadata beside the binary configuration [14]. Importable file formats were shown in section 2.3.1.

Beside the actual functionality of the DD-HMC software system, its library could be the base for a more comprehensive software system for the simulation of LQCD. This would require the extension of the code according to the particular needs but adding new ideas and own improvements should be a manageable task.

In comparison with the Chroma software system, the programming structure of DD-HMC can lead to more specialised software which is not as portable as the one which is built of several software modules. However, the specialised software system remains possibly more understandable.

The most time consuming task in LQCD simulations with dynamical fermions and measurements is the application of the Dirac operator. To analyse the performance of the Chroma software system the application of the Wilson-Dirac operator was benchmarked. A theoretical model for the execution time of this application on a single node as well as on a parallel machine was discussed. To determine the theoretical execution time of this task it was assumed that this time is proportional to the data exchange of the slowest data path between different storage systems. For a single node the slowest data path is the one between memory (RAM) and cache. For multiple nodes one also has to consider the network which connects the different computing nodes. In practice it is often not fulfilled that the whole system of gauge fields and spinors fits into cache of one single

processor. Therefore, it is necessary to split the multi-dimensional array of fields into blocks which are successive loaded into cache. The discussed model agreed with the time measurement on a single node for a large range of different lattice volumes. It was more difficult to obtain reproduceable time measurements on a parallel machine because of noncontrollable interfering with other running processes. Nevertheless, it was possible to obtain a qualitative agreement with the theory.

One important test of the simulation of LQCD is the determination of hadron masses. Finally, a sample calculation for the effective pion mass for a certain set of parameters was done with the Chroma software system. The correctness of the simulation and measurement could be proved within the error in comparison with other computations [1].

Appendix A

Conventions

A.1 Gauge Fields

$U(x,\mu)$ defines the gauge link at the lattice site x in positive direction $\mu = 0,1,2,3$. The coordinate next to x in positive direction μ is defined as $x+\hat{\mu}$ whereas the coordinate in negative direction is given by $x-\hat{\mu}$. Therfore, a gauge link at the lattice site x in positive direction μ is given by

$$U(x,-\mu) = U^{\dagger}(x-\hat{\mu},\mu)\,. \tag{A.1}$$

A gauge configuration is a set of gauge links at all lattice sites $x \in V$ in each direction $\mu = 0,1,2,3$.

A.2 Wilson-Dirac operator

A common discretisation of the fermionic part of the ation of QCD is the formulation of Wilson. Details can be found in [27]. The Wilson action S_q^{Wilson} can be written as

$$S_q^{Wilson} = \sum_{x,y}\left(\bar{\psi}(x)\,M(x,y)\,\psi(y)\right), \tag{A.2}$$

$$M(x,y) = \delta_{xy} - \kappa\sum_{\mu}\left(1-\gamma_\mu\right)U(x,\mu)\,\delta_{x+\hat{\mu},y} + \left(1+\gamma_\mu\right)U^{\dagger}(x,\mu)\,\delta_{x-\hat{\mu},y}, \tag{A.3}$$

wherin the mass parameter κ (also called hopping parameter) is given by

$$\kappa = \left(8+2m_0\right)^{-1}. \tag{A.4}$$

Therefore, the mass of a Wilson quark is given by

$$m_0 = \frac{1}{2\kappa} - 4. \tag{A.5}$$

Furthermore, the interaction matrix $M(x,\mu)$ can be written as

$$M(x,y) = \delta_{xy} - \kappa H(x,y), \tag{A.6}$$

wherin $H(x,y)$ is called hopping term of the Wilson Dirac operator $M(x,y)$.
For a simpler notation the lattice spacing was set to $a=1$ in this definition.

A.3 Representation of the Dirac matrices and the possible advantages

To fulfil the algebra

$$\{\gamma_\mu, \gamma_\nu\} = 2\delta_{\mu\nu}, \tag{A.7}$$

it is possibile to use the representation

$$\gamma_\mu = \begin{pmatrix} 0 & e_\mu \\ e_\mu^\dagger & 0 \end{pmatrix}, \tag{A.8}$$

whereas e_μ is

$$e_0 = -1, \tag{A.9}$$

$$e_k = -i\sigma_k, \tag{A.10}$$

and σ_k are the Pauli matrices whereas $k = 1,2,3$.
Furthermore $\gamma_5 = \gamma_0\gamma_1\gamma_2\gamma_3$ can be defined. Then the expressions

$$\gamma_\mu^\dagger = \gamma_\mu, \tag{A.11}$$

$$\gamma_5^\dagger = \gamma_5, \gamma_5^2 = \gamma_5, \tag{A.12}$$

are valid.
For the calculation of the Dirac operator the expression

$$\psi = (1 - s\gamma_\mu)\,\phi, s = \pm 1, \tag{A.13}$$

has to be computed several times, wheras ψ and ϕ are spinors. Because of the representation of the Dirac matrices the half of the computations can be saved like it shown below.

$$\begin{aligned} & s = +1\,, \quad \mu = 0\,: \\ & \psi_1 = \phi_1 + \phi_3 \\ & \psi_2 = \phi_2 + \phi_4 \\ & \psi_3 = \psi_1 \\ & \psi_4 = \psi_2 \end{aligned}$$

$s = -1\,, \quad \mu = 0\,:$
$\psi_1 = \phi_1 - \phi_3$
$\psi_2 = \phi_2 - \phi_4$
$\psi_3 = -\psi_1$
$\psi_4 = -\psi_2$

$s = +1\,, \quad \mu = 1\,:$
$\psi_1 = \phi_1 + i\phi_4$
$\psi_2 = \phi_2 + i\phi_3$
$\psi_3 = -i\psi_2$
$\psi_4 = -i\psi_1$

$s = -1\,, \quad \mu = 1\,:$
$\psi_1 = \phi_1 - i\phi_4$
$\psi_2 = \phi_2 - i\phi_3$
$\psi_3 = i\psi_2$
$\psi_4 = i\psi_1$

$s = +1\,, \quad \mu = 2\,:$
$\psi_1 = \phi_1 + \phi_4$
$\psi_2 = \phi_2 - \phi_3$
$\psi_3 = -\psi_2$
$\psi_4 = \psi_1$

$s = -1\,, \quad \mu = 2\,:$
$\psi_1 = \phi_1 - \phi_4$
$\psi_2 = \phi_2 + \phi_3$
$\psi_3 = \psi_2$
$\psi_4 = -\psi_1$

$s = +1\,, \quad \mu = 3\,:$
$\psi_1 = \phi_1 + i\phi_3$
$\psi_2 = \phi_2 - i\phi_4$
$\psi_3 = -i\psi_1$
$\psi_4 = i\psi_2$

$$
\begin{aligned}
& s=-1\,, \quad \mu=3\;: \\
& \psi_1=\phi_1-i\phi_3 \\
& \psi_2=\phi_2+i\phi_4 \\
& \psi_3=i\psi_1 \\
& \psi_4=-i\psi_2
\end{aligned}
$$

A.4 Effectice Action

The action for LQCD with fermions has the form

$$S\left[U,\bar{\psi},\psi\right]=S_g\left[U\right]+S_q\left[U,\bar{\psi},\psi\right], \tag{A.14}$$

wherin S_g is the gauge part of the action and S_q is given by

$$S_q\left[U,\bar{\psi},\psi\right]=\sum_{x,y}\left(\bar{\psi}\left(x\right)M\left(x,y\right)\psi\left(y\right)\right). \tag{A.15}$$

The expectation value of the observable A is given by

$$\langle A\rangle=\frac{1}{Z}\int\mathcal{D}U\mathcal{D}\bar{\psi}\mathcal{D}\psi A\mathrm{e}^{-S[U,\bar{\psi},\psi]}, \tag{A.16}$$

wherein Z is given by

$$Z=\int\mathcal{D}U\mathcal{D}\bar{\psi}\mathcal{D}\psi\mathrm{e}^{-S[U,\bar{\psi},\psi]}. \tag{A.17}$$

Introducing anticommuting Grassmann numbers for the anticommuting fermion fields ψ in S_q, the integration of the quark degrees of freedom leads to

$$\int\mathcal{D}\bar{\psi}\mathcal{D}\psi\;\mathrm{e}^{S_q[U,\bar{\psi},\psi]}=\det M. \tag{A.18}$$

A common way to deal with this determinant is introducing *pseudo-fermions* ϕ. Considering two flavours of dynamical quarks with equal mass the quark determinant can be written as

$$\left(\det M\right)^2=\det M^\dagger M=\int\mathcal{D}\phi^\dagger\mathcal{D}\phi e^{-S_{pf}\left[U,\phi^\dagger,\phi\right]}, \tag{A.19}$$

where $S_{pf}\left[U,\phi^\dagger,\phi\right]$ is given by

$$S_{pf}\left[U,\phi^\dagger,\phi\right]=\phi^\dagger\left(M^\dagger M\right)^{-1}\phi. \tag{A.20}$$

With respect to A.14 and with the representation A.19 it is possible to introduce an effective action

$$S_{eff}\left[U,\phi^\dagger,\phi\right]=S_g\left[U\right]+S_{pf}\left[U,\phi^\dagger,\phi\right]. \tag{A.21}$$

Therefore, the expectation value of the observable A is given by

$$\langle A\rangle = \frac{1}{Z}\int \mathcal{D}U\mathcal{D}\phi^{\dagger}\mathcal{D}\phi A \mathrm{e}^{-S_{eff}\left[U,\phi^{\dagger},\phi\right]}, \tag{A.22}$$

with the function

$$Z = \int \mathcal{D}U\mathcal{D}\phi^{\dagger}\mathcal{D}\phi \mathrm{e}^{-S_{eff}\left[U,\phi^{\dagger},\phi\right]}. \tag{A.23}$$

A.5 Hybrid Monte Carlo Algorithm

Hybrid Monte Carlo (HMC) is the basic algorithm for the generation of gauge configurations with dynamical fermions. More details can be found in [27].

A.5.1 HMC Hamiltonian and Hamilton's Equations of Motion

The goal of a simulation is to generate gauge configurations U which are distributed by their importance in the factor $\exp\left[-S\left[U,\bar{\psi},\psi\right]\right]$. This method is called importance sampling. The HMC algorithm uses molecular dynamic methods in combination with a Metropolis accept/reject step. The latter is described in subsection A.5.3. Gauge configurations are generated in a markov process. This process represents a sequenz of states which are produced with a cerain transition probability from a given state to the next one, i.e. a given gauge configuration $\{U_i\}$ is used to generate the next configuration $\{U_j\}$ such as

$$\{U_i\} \overset{P(\{U_i\}\rightarrow\{U_j\})}{\longrightarrow} \{U_i\}. \tag{A.24}$$

To obtain a new state in the markov process the gauge configurations are evaluated by means of Hamilton's equations of motion. The introduced HMC Hamiltonian has the form

$$H\left[\Pi, U, \phi^{\dagger}, \phi\right] = T\left[\Pi\right] + S_{eff}, \tag{A.25}$$

where the kinetic term is

$$T\left[\Pi\right] = \frac{1}{2}\sum_{x,\mu}\mathrm{Tr}\,\Pi^2\left(x,\mu\right). \tag{A.26}$$

To get a new configuration Hamilton's equations of motion

$$\frac{d}{dt}U\left(x,\mu\right) = \frac{\partial H\left[\Pi, U, \phi^{\dagger}, \phi\right]}{\partial\Pi\left(x,\mu\right)}, \tag{A.27}$$

$$\frac{d}{dt}\Pi\left(x,\mu\right) = -\frac{\partial H\left[\Pi, U, \phi^{\dagger}, \phi\right]}{\partial U\left(x,\mu\right)}, \tag{A.28}$$

that lead to

$$\frac{d}{dt}U\left(x,\mu\right)=\Pi\left(x,\mu\right)U\left(x,\mu\right), \tag{A.29}$$

$$\frac{d}{dt}\Pi\left(x,\mu\right)=-F\left(x,\mu\right), \tag{A.30}$$

have to be solved, wherein the force $F\left(x,\mu\right)$ corresponds to the action S_{eff}. t is the fictious molecular dynamic time.

A.5.2 Leap Frog Integration

For the integration of equations (A.29) from time $t=0$ to $t=\tau_{mc}$ in N steps a numerical integration scheme is necessary. The fundamental operations for the step size $\epsilon=\frac{\tau_{mc}}{N}$ are

$$\mathcal{I}_U\left(\epsilon\right):U\left(t+\epsilon\right)=\exp\left[\epsilon\Pi\left(t\right)\right]U\left(t\right), \tag{A.31}$$

$$\mathcal{I}_\Pi\left(\epsilon\right):\Pi\left(t+\epsilon\right)=\Pi\left(t\right)-\epsilon F\left(t\right). \tag{A.32}$$

Those integration schemes have to preserve the measure in phase space and have to be time-reversible. The widely used leap-frog scheme satisfies the requirements and is given by

$$\left\{\mathcal{I}_U\left(\frac{1}{2}\epsilon\right)\mathcal{I}_\Pi\left(\epsilon\right)\mathcal{I}_U\left(\frac{1}{2}\epsilon\right)\right\}^N,\epsilon=\frac{\tau_{mc}}{N}. \tag{A.33}$$

A.5.3 Metropolis Algorithm

In this algorithm the transition probability to get a new configuration is given by

$$P\left(\{\xi\}\rightarrow\{\xi'\}\right)=\min\left\{1,\frac{\mathrm{e}^{-S[\xi]}}{\mathrm{e}^{-S[\xi']}}\right\}, \tag{A.34}$$

wherein $S\left[\xi\right]$ and $S\left[\xi'\right]$ are the actions corresponding to the state ξ and ξ', respectivly.

In the HMC algorithm the Metropolis step is done at the end of the integration of (A.29) by the numerical integrator. The action S_{eff} is not conserved during the solution of (A.29) because of the discretisation of these equations of motion. Therefore, the Metropolis step is necessary.

Appendix B

Parameters for XML input files for the Chroma software system

B.1 General posibilities for “InlineMeasurements”

- APE_SMEAR
- BAR3PTFN
- BUILDING_BLOCKS
- COULOMB_GAUGEFIX
- EIGBNDSMDAGM
- ERASE_NAMED_OBJECT
- FUZZED_WILSON_LOOP
- GAUSSIAN_INIT_NAMED_OBJECT
- HYP_SMEAR
- LINK_SMEAR
- LIST_NAMED_OBJECT
- MAKE_SOURCE
- MAKE_SOURCE_FERM
- MESON_SPECTRUM
- MRES
- MULTI_PROPAGATOR

- NERSC_WRITE_NAMED_OBJECT
- NOISY_BUILDING_BLOCKS
- PLAQUETTE
- POLYAKOV_LOOP
- PROPAGATOR
- PROPAGATOR_FERM
- QIO_READ_NAMED_OBJECT
- QIO_WRITE_ERASE_NAMED_OBJECT
- QIO_WRITE_NAMED_OBJECT
- QPROPADD
- QPROPQIO
- QQBAR
- QQQ
- QQQ_NUCNUC
- RITZ_KS_HERM_WILSON
- SEQSOURCE
- SINK_SMEAR
- SNARF_NAMED_OBJECT
- SPECTRUM
- SPECTRUM_OCT
- SPECTRUM_S
- STOUT_SMEAR
- SZIN_READ_NAMED_OBJECT
- SZIN_WRITE_NAMED_OBJECT
- WILSLP
- XML_WRITE_NAMED_OBJECT

B.1.1 Possible sources for "MAKE_SOURCE"

- PARTIAL_WALL_SOURCE
- POINT_SOURCE
- RAND_Z2_WALL_SOURCE
- SHELL_SOURCE
- WALL_SOURCE

B.1.2 Possible wavefunctions for sink smearing for "SPECTRUM"

- GAUSSIAN
- EXPONENTIAL
- GAUGE_INV_GAUSSIAN
- WUPPERTAL
- JACOBI

B.2 Possibilities for main program "hmc"

Monomial:

- GAUGE_MONOMIAL
- N_FLAVOR_LOGDET_EVEN_EVEN_FERM_MONOMIAL
- ONE_FLAVOR_EOPREC_CONSTDET_FERM_RAT_MONOMIAL
- ONE_FLAVOR_EOPREC_CONSTDET_FERM_RAT_MONOMIAL5D
- ONE_FLAVOR_UNPREC_FERM_RAT_MONOMIAL
- ONE_FLAVOR_UNPREC_FERM_RAT_MONOMIAL5D
- TWO_FLAVOR_EOPREC_CONSTDET_FERM_MONOMIAL
- TWO_FLAVOR_EOPREC_CONSTDET_FERM_MONOMIAL5D
- TWO_FLAVOR_EOPREC_CONSTDET_HASENBUSCH_FERM_MONOMIAL
- TWO_FLAVOR_EOPREC_CONSTDET_POLYNOMIAL_FERM_MONOMIAL
- TWO_FLAVOR_EOPREC_CONSTDET_POLYPREC_FERM_MONOMIAL
- TWO_FLAVOR_EOPREC_LOGDET_FERM_MONOMIAL

- TWO_FLAVOR_UNPREC_FERM_MONOMIAL
- TWO_FLAVOR_UNPREC_FERM_MONOMIAL5D
- TWO_FLAVOR_UNPREC_HASENBUSCH_FERM_MONOMIAL

Inverter:

- CG_INVERTER
- MR_INVERTER
- BICG_INVERTER
- SUMR_INVERTER
- REL_CG_INVERTER
- REL_SUMR_INVERTER
- REL_GMRESR_SUMR_INVERTER
- REL_GMRESR_CG_INVERTER
- BICGSTAB_INVERTER

Fermion action:

- CLOVER
- OVERLAP_PARTIAL_FRACTION_4D
- PARWILSON
- POLYNOMIAL_CHEBYSHEV
- UNPRECONDITIONED_CLOVER
- UNPRECONDITIONED_DWFTRANSF
- UNPRECONDITIONED_HAMBER-WU
- UNPRECONDITIONED_PARWILSON
- UNPRECONDITIONED_WILSON
- WILSON

Fermion boundary condition:

- PERIODIC_FERMBC
- SCHROEDINGER_CHROMOMAG_FERMBC

- SCHROEDINGER_COUPLING_FERMBC
- SCHROEDINGER_DIRICHLET_FERMBC
- SCHROEDINGER_NONPERT_FERMBC
- SCHROEDINGER_TRIVIAL_FERMBC
- SIMPLE_FERMBC
- TWISTED_FERMBC

Chronological predictor:

- LAST_SOLUTION_4D_PREDICTOR
- LINEAR_EXTRAPOLATION_4D_PREDICTOR
- MINIMAL_RESIDUAL_EXTRAPOLATION_4D_PREDICTOR
- ZERO_GUESS_4D_PREDICTOR

Integrator:

- LCM_MINIMUM_NORM_2ND_ORDER_INTEGRATOR
- LCM_MINIMUM_NORM_2ND_ORDER_INTEGRATOR_MTS
- LCM_MINIMUM_NORM_2ND_ORDER_QPQ_INTEGRATOR_MTS
- LCM_PQP_LEAPFROG_INTEGRATOR
- LCM_PQP_LEAPFROG_INTEGRATOR_MTS
- LCM_SEXTON_WEINGARTEN_INTEGRATOR
- LCM_TWO_SCALE_MINIMUM_NORM_INTEGRATOR

Bibliography

[1] Boris Orth, Thomas Lippert, and Klaus Schilling. Finite-size effects in lattice qcd with dynamical wilson fermions. *Phys. Rev.*, D72:014503, 2005.

[2] Robert G. Edwards and Balint Joo. The chroma software system for lattice qcd. *Nuclear Physics B - Proceedings Supplements*, 140:832, 2005.

[3] Usqcd software releases. URL `http://www.usqcd.org/usqcd-software/`.

[4] Balint Joo. Lattice qcd message passing (qmp), 2007. URL `http://usqcd.jlab.org/usqcd-docs/qmp/`.

[5] Balint Joo. Qla linear algebra interface for qcd, 2007. URL `http://usqcd.jlab.org/usqcd-docs/qla/`.

[6] Balint Joo. Qcd input/output api for lqcd, 2007. URL `http://usqcd.jlab.org/usqcd-docs/qio/`.

[7] Balint Joo. Qdp++ data parallel interface for qcd, 2007. URL `http://usqcd.jlab.org/usqcd-docs/qdp++/`.

[8] The hacklatt 2006 chroma tutorial, 2006. URL `http://usqcd.jlab.org/usqcd-docs/chroma/HackLatt06/index.html`.

[9] Balint Joo. Installation notes for the chroma software system, April 2005. URL `http://usqcd.jlab.org/usqcd-docs/chroma/HackLatt06/Installation/html/index.html`.

[10] Dimitri van Heesch. Doxygen - source code documentation generator tool. URL `http://www.doxygen.org`.

[11] Qdp++ doxygen source code documentation. URL `http://usqcd.jlab.org/usqcd-software/qdp++/qdp++/docs/ref/html/`.

[12] Chroma doxygen source code documentation, . URL `http://usqcd.jlab.org/usqcd-software/chroma/chroma/docs/doxygen/html/`.

[13] SciDAC. Lime, May 2006. URL http://usqcd.jlab.org/usqcd-docs/c-lime/lime_1p2.pdf.

[14] ILDG Metadata Working Group. Ildg binary file format, December 2005. URL http://www-zeuthen.desy.de/~pleiter/ildg/ildg-file-format-1.1.pdf.

[15] Gnu operating system web-site. URL http://www.gnu.org.

[16] The xml c parser and toolkit of gnome. URL http://www.xmlsoft.org/.

[17] Xsl transformations (xslt). URL http://www.w3.org/TR/xslt.

[18] Robert G. Edwards. Qdp++ data parallel interface for qcd, May 2006. URL http://www.jlab.org/~edwards/qcdapi/docs/level2/qdp++/manual.pdf.

[19] Qcd-measure-chroma - main email list for chroma, . URL http://forge.nesc.ac.uk/mailman/listinfo/qcd-measure-chroma.

[20] Martin Luscher. Dd-hmc algorithm for two-flavour lattice qcd. URL http://luscher.web.cern.ch/luscher/DD-HMC/index.html.

[21] Martin Luscher. Schwarz-preconditioned hmc algorithm for two-flavour lattice qcd. *Computer Physics Communications*, 165:199, 2005.

[22] Martin Luscher. Solution of the dirac equation in lattice qcd using a domain decomposition method. *Computer Physics Communications*, 156:209, 2004.

[23] Hubert Simma. Analysis and modeling of the performance of lqcd kernels, 2006. URL https://indico.desy.de/contributionDisplay.py?contribId=4&confId=132.

[24] Hubert Simma. Lattice qcd on the cell processor, 2007. URL http://www.fz-juelich.de/zam/datapool/cell/Lattice_QCD_on_Cell.pdf.

[25] Hubert Simma. Scientific high performance computing, lecture 7, 2007. URL http://moby.mib.infn.it/~simma/slides07.pdf.

[26] G. Bilardi. The potential of on-chip multiprocessing for qcd machines. *Lecture Notes in Computer Science*, 3769:386, 2005.

[27] István Montvay and Gernot Münster. *Quantum Fields on a Lattice*. Cambridge University Press, Cambridge, 1994.

[28] H.J. Rothe. *Lattice Gauge Theories: An Introduction*. World Scientific, Singapore, 1992.

[29] Ulli Wolff. Monte carlo errors with less errors. *Comput. Phys. Commun.*, 156:143–153, 2004.

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig sowie ohne unerlaubte fremde Hilfe verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Berlin, 06. Juni 2007

Jens Grieger