Universität
Regensburg

Fakultät Physik

Using the GRID library for lattice QCD on new computer architectures

Masterarbeit

im Studiengang Computational Science
Schwerpunkt Physik

zur Erlangung des akademischen Grades
Master of Science

Autor: Moritz Fink
MatNr. 1539016

Eingereicht bei: Prof. Dr. Tilo Wettig

Betreuer: Dr. Piotr Korcyl

Version vom: 26. August 2017

Abstract

This thesis discusses the basics of lattice QCD and the architecture of modern hardware and software. This is later on used to compare the performances of the Chroma, QDP++ & LibHadronAnalysis libraries to the new GRID implementation, especially when it comes to Meson Distribution Amplitude (MDA) calculations.

Contents

List of Figures

List of Tables

Nomenclature

API	Application Programming Interface
BDA	Baryon Distribution Amplitude
DDR	Double Data Rate
DIMM	Dual Inline Memory Module
Flop	Floating Point Operation
HFI	Host Fabric Interface
I/O	Input/Output
ISA	Instruction Set Architecture
JSC	Jülich Supercomputing Center
KNL	Knights Landing
MCDRAM	Multi-Channel Dynamic Random-Access Memory
MDA	Meson Distribution Amplitude
MIC	Many Integrated Core
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OPA	Omni-Path Architecture
QCD	Quantum Chromodynamics
QPACE	QCD Parallel Computing Engine
RAM	Random-Access Memory
SIMD	Single Instruction Multiple Data
UMA	Uniform Memory Access
VPU	Vector-Processing Unit
XML	Extensible Markup Language

1 Introduction

A lot of problems in all different scientific fields are studied with the support of the computational power of supercomputers. Besides the simulations in molecular dynamics, climate models and astronomy those machines are also used for Quantum Chromodynamics (QCD) on the lattice. As the latter is often very time and resource consuming and large computer clusters are expensive to purchase and maintain, the biggest task is to use software and hardware as efficiently as possible. In the case of lattice QCD this means that the given problem has to be divided in such a way that it can be distributed among many cores and be processed in parallel in order to achieve a minimal execution time. As this is a complex task and presumes a deep understanding of the matter, there is software available that allows the user to formulate the problem in terms of comprehensive code and deals with the job of parallelization on its own.
This thesis will guide the reader through the process of developing and testing optimized code for new computer architectures. This task implies the analysis of the currently used software and the newly developed C++ library: GRID. Therefore, benchmarks are written and run for the different libraries in order to test and compare the performances for specific cases.
The thesis starts with a few notes on lattice QCD and introduces the Meson Distribution Amplitude (MDA) in chapter 2 that will later serve as a case study. The hardware, namely the QPACE 3 supercomputer and especially its processors are discussed in chapter 3. After a brief look at the libraries (Chroma, QDP++ & LibHadronAnalysis) currently used for the calculations in chapter 4, the new library GRID is introduced, containing information about the implementation and its application in chapter 5. With the knowledge from those previous chapters the old and new libraries can then be compared for specific tasks in chapter 6 which is afterwards used to implement the computation of the MDA using GRID in chapter 7.

2 Lattice QCD

Since the first publication on this subject by Kenneth Wilson in 1974[1] the field of lattice Quantum Chromodynamics (lattice QCD) provides a wide range of methods for the numerical calculation of hadron properties. The idea behind this theory is to discretise the continuous four dimensional spacetime and thereby construct a hypercubic lattice (with spacing a) with a finite volume. A quark can only be placed on the sites of this grid and is created and annihilated through the corresponding operators. The gluons are represented by links $U_{x,\mu}$ in between the sites. Figure 1 shows the general structure of such a lattice. As such a construct is finite in every spacetime direction (typically $\sim 8^4$ sites), one establishes periodic conditions on the hyperplane boundaries for the gauge fields and anti-periodic boundary conditions for the fermions. The path integrals emerging from this theory can then be evaluated via Monte Carlo methods and importance sampling.
This chapter introduces common lattice QCD calculations, especially correlation functions and derivatives which will later be used for benchmarking. Those can then be used to find an efficient way of computing the Meson Distribution Amplitude (MDA).

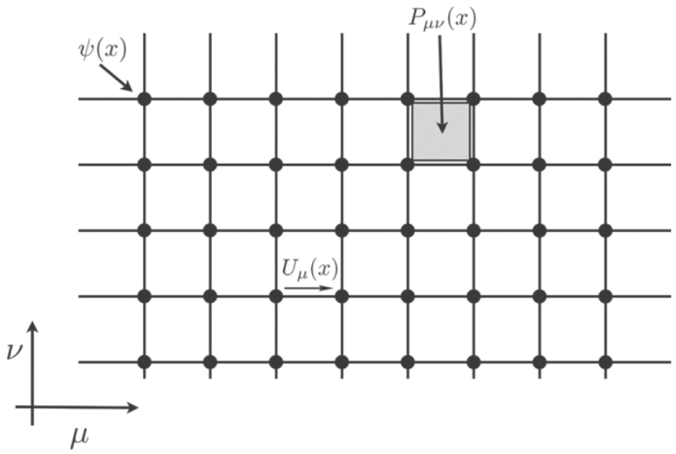

Figure 1: Two dimensional slice of the four dimensional lattice. Each site contains a quark field $\psi(x)$ and is connected to its neighbours via the gauge link $U_\mu(x)$. The $P_{\mu\nu}(x)$ is a so called plaquette which will not be discussed in this thesis (source: [2, Figure 1]).

2.1 Correlation functions

Correlation or two-point functions (cf. figure 2) represent an important object in the field of lattice QCD, e.g. for spectroscopic calculations. Consider the general form of the correlator for an interpolating operator $O(t)$ (following [3, Chapter 11.1]):

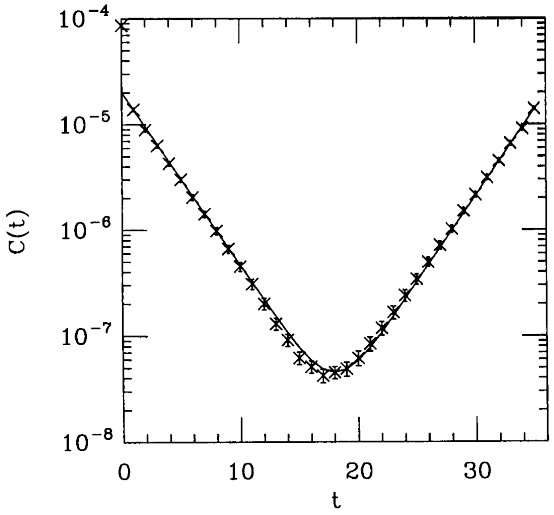

Figure 2: A correlation function using $N_T = 32a$ and its corresponding fit (source: [3, Figure 11.1]).

$$G(t,0) = \frac{1}{Z} Tr\left[O(t)O(0)e^{-\beta H}\right] \tag{1}$$

where the partition function $Z = Tr\left[e^{-\beta H}\right]$, H is the Hamiltonian and $\beta = \frac{1}{k_B T} = \frac{1}{T}$ (in simulations the Boltzmann constant is usually set to one). With $O(t) = e^{Ht}O(0)e^{-Ht}$ and the insertion of a complete set of eigenstates equation (1) becomes

$$G(t,0) = \frac{1}{Z}\sum_{m,n}\langle m|O(0)|n\rangle\langle n|O(0)|m\rangle e^{-E_n t}e^{-E_m(N_T-t)} \tag{2}$$

where E_n is the energy relative to the ground state. For large t the first excited state dominates so that

$$G(t,0) \propto |\langle 0|O(0)|1\rangle|^2\left(e^{-E_1 t} + e^{-E_1(N_T-t)}\right) \tag{3}$$

After all these simplifications, the mass and energy can be obtained by fitting the two-point function for various times t to the following function

$$E_1 \approx m_{eff} = \lim_{t\to\infty} acosh\left[\frac{G(t+1,0)+G(t-1,0)}{2\ G(t,0)}\right] \tag{4}$$

In order to get to a more vivid case, the operator $O(x,t) = \overline{\psi}(x,t)\Gamma\psi(x,t)$ (now with space coordinates x) shall represent the source (for $t = 0$) and sink (t) of a meson in the Wilson formalism with Γ being one of the Dirac matrices. The correlation function can now be written as follows:

$$G(t,0) = \sum_x \langle O(x,t)O(0,0)\rangle \tag{5}$$

$$= \sum_x \langle 0|\overline{\psi}_i^\alpha(x,t)\Gamma_{ij}\psi_j^\alpha(x,t)\overline{\psi}_k^\beta(0,0)\Gamma_{kl}\psi_l^\beta(0,0)|0\rangle \tag{6}$$

In the second line, spin (i,j,k,l) and colour indices (α,β) have been explicitly denoted. Introducing the quark propagator

$$S_{jk}^{\alpha\beta}(x,t;0,0) = \langle 0|\psi_j^\alpha(x,t)\overline{\psi}_k^\beta(0,0)|0\rangle \tag{7}$$

the correlator becomes

$$\begin{aligned} G(t,0) = &\sum_x Tr\left[S(x,t;x,t)\Gamma\right] Tr\left[S(0,0;0,0)\Gamma\right] \\ &- \sum_x Tr\left[S(x,t;0,0)\Gamma S(0,0;x,t)\Gamma\right] \end{aligned} \tag{8}$$

The trace is taken over spin and colour indices. The anti-quark propagator can be found using the identity $S(0,0;x,t) = \gamma_5 S^*(x,t;0,0)\gamma_5$, which will be relevant for section 6 where the second term of equation 8 will be further discussed. The first term in equation 8 is often omitted as it is very sophisticated to calculate.

2.2 Covariant lattice Dirac operator

In the Wilson formalism the gauge potential $A_\mu(x)$ is replaced by gauge links $U_{x,\mu}$ which are located between the sites x and $x + a\hat{\mu}$ of the lattice. Their relation is given by

$$U_{x,\mu} \equiv U(x, x+\hat{\mu}) = \exp\left[igaA_\mu^c(x)\lambda_c\right] \tag{9}$$

where λ_c $(c = 1, ..., 8)$ are the eight generators of $SU(3)$ and g is the gauge coupling constant (cf. [3, Chapter 5.1]). In order to provide gauge invariance those variables must transform as

$$U_{x,\mu} \to G(x)U_{x,\mu}G^{-1}(x+a\hat{\mu}) \tag{10}$$

The Dirac operator can then be discretised through (following [4, Chapter 2])

$$\mathcal{D}(U) = \gamma_\mu D_{s,\mu}(U) + m \tag{11}$$

$$\left(D_{s,\mu}(U)\psi\right)_x = \frac{1}{2a}\left(U_{x,\mu}\psi_{x+a\hat{\mu}} - U^\dagger_{x-a\hat{\mu},\mu}\psi_{x-a\hat{\mu}}\right) \tag{12}$$

where $U^\dagger \equiv U(x+\hat{\mu}, x)$ is the reversed gauge link. The emerging fermion doubling problem is solved by adding a term proportional to the Laplace operator Δ_L as proposed by Wilson [5]:

$$\mathcal{D}_W(U) = \gamma_\mu D_{s,\mu}(U) - r\Delta_L \quad (0 < r \leq 1), \qquad \mathcal{D}^\dagger{}_W = \gamma_5 \mathcal{D}_W \gamma_5 \tag{13}$$

2.3 Meson Distribution Amplitude

In order to present an application for QCD simulations on the lattice, the Meson Distribution Amplitude (MDA) shall be introduced here. This function contains information about momentum sharing between the quark - anti-quark pair of the meson [6], e.g. the pion (as discussed in [6][7]). It can be obtained through a Fourier transform of the following term[1]:

$$MDA(x) = \sum_{\alpha,\beta} \sum_{A,B} \sum_{\gamma,\delta} S^{AB}_{\alpha\beta}(x) \left(\gamma_5 S^\dagger \gamma_5\right)^{AB}_{\gamma\delta}(x) \tag{14}$$

So the evaluation of equation (14) on a specific lattice site x is performed by taking the trace over colours A and B for all possible spin index combinations α, β and γ, δ. Additionally, each combination of quark propagators including their derivatives has to be considered.

[1] The Fourier transform will be left out in the further discussion as it is not of any special interest in this thesis.

3 QPACE 3

The QCD Parallel Computing Engine 3 (QPACE 3) is a project of the universities in Regensburg and Wuppertal, located at the Jülich Supercomputing Center (JSC). It has been specifically designed for lattice QCD simulations. One special feature is its high energy-efficiency[2], that currently puts it among the 20 most energy-efficient supercomputers in the world[8]. This chapter discusses the layout of QPACE 3 and the most relevant attributes of its processors.

3.1 Architecture

QPACE 3 consists of eight racks with 672 nodes in total. Each node contains one Intel Xeon Phi 7210 (also called Knights Landing / KNL), 48GB RAM (racks 1-4) or 96GB RAM (racks 5-8) and a 100Gb/s Omni-Path Host Fabric Interface (HFI). Figure 3 shows the layout of one KNL block. Other than on the picture, the Omni Path is not on the package but outside in the version used for QPACE 3. The hardware configuration set at boot time is using the SNC-4 mode for the NUMA layout and cache mode for the MCDRAM, which will be further discussed below.

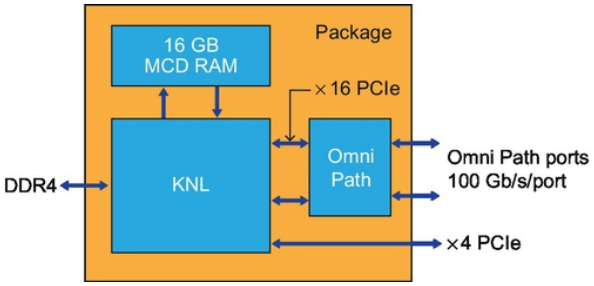

Figure 3: One node consists of a Intel Xeon Phi 7210 (KNL), two types of memory and the Omni-Path Architecture (source: [9, Chapter I.2]).

3.2 Knights Landing

The KNL is the second generation of Intel's Many Integrated Core (MIC) architecture processors. Its purpose is to deliver hardware that is able to execute parallel tasks in a very efficient way. Therefore it provides a multi-core architecture, high-bandwidth memory and 512 bit vector instructions [9, Chapter I.2] [3].

3.2.1 Layout

The KNL basically consists of 36 tiles placed in a two-dimensional mesh, as can be seen in figure 5. Each of those tiles contains two cores with two vector-processing units (VPU) each and a shared 1 MB L2 cache (cf. figure 4).

[2] Energy-efficiency is measured as the ratio of performance over power (Floating point operations per Watt).

[3] Information in this section was taken from chapter I.2. Knights Landing overview in [9].

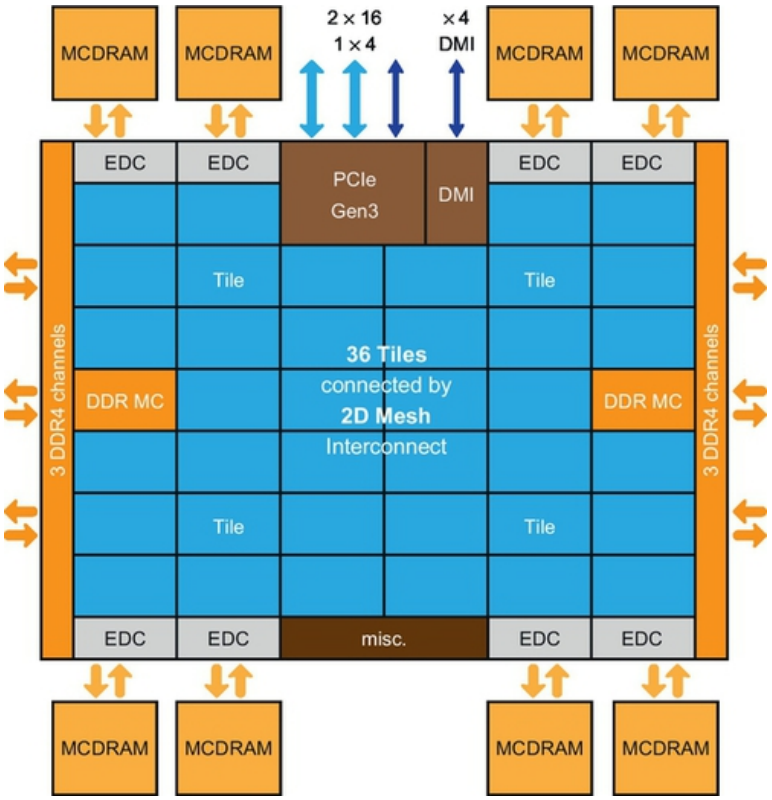

Figure 5: Schematic representation of KNL's structure. Although 38 tiles are drawn only 36 of them are active (source: [9, Chapter I.2]).

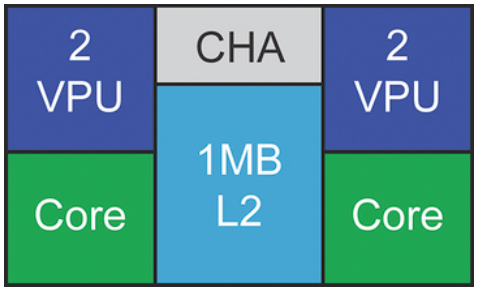

Figure 4: One tile of the KNL consisting of two cores, four VPU and a 1MB L2 cache (source: [9, Chapter I.2]).

If a core needs data which is not stored in its local tile, it must communicate via the on-die interconnect. In this case the information needed travels across the mesh following the YX-routing rule (first in the y direction and then in x direction), forming a ring for every full transaction. In order to achieve a better performance several cluster modes can be applied: All-to-all, quadrant (or hemisphere) and sub-NUMA (SNC-4 or SNC-2).
These modes basically differ in the way memory can be accessed. The all-to-all, quadrant and hemisphere modes are Uniform Memory Access (UMA), which means that the latency from any given core to any location in memory is essentially the same. Whereas the SNC-4 and SNC-2 modes use Non-Uniform Memory Access (NUMA), so that small core-memory distances result in a lower latency but larger distances in a higher latency compared to the UMA modes. As the 2D mesh is divided into two (hemisphere/SNC-2) or four (quadrant/SNC-4) regions, the NUMA modes can access memory faster which is located in the same hemisphere/quadrant. Those statements are independent of the memory type, MCDRAM or DDR.

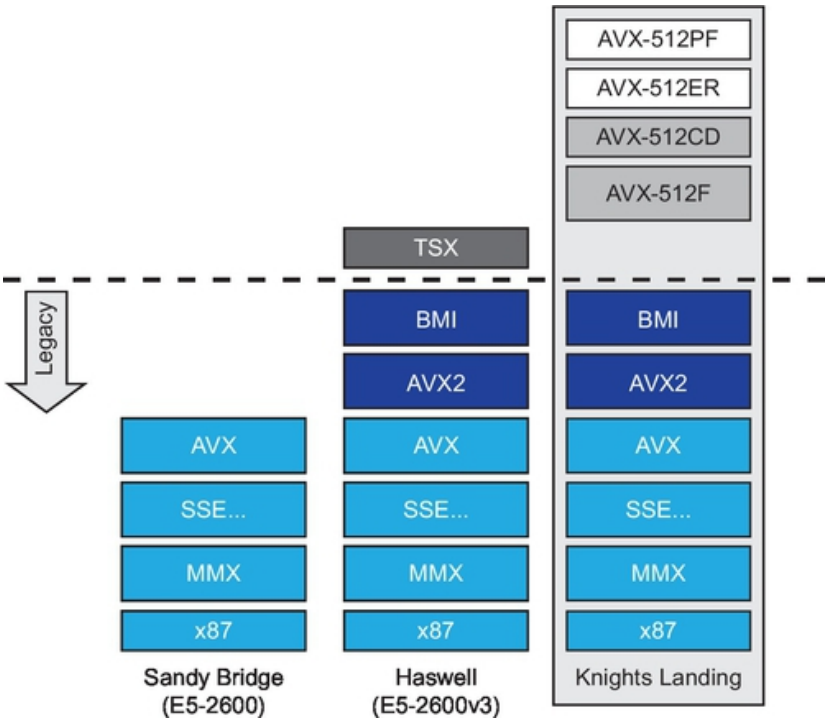

Figure 6: The different instruction sets of Sandy Bridge, Haswell and Knights Landing (source: [9, Chapter I.2]).

3.2.2 MCDRAM & DDR4

The KNL has two different types of memory: a high-bandwidth Multi-Channel Dynamic RAM (MCDRAM) and a high-capacity DDR4 RAM. The eight MCDRAM devices provide 16 GB of memory (2 GB each) and an aggregate Streams Triad bandwidth of over 450 GB/s. It either serves as cache for the DDR, additional memory in the same address space as DDR or a mixture of both, according to the mode (cache/flat/hybrid) set at boot time.
The DDR is composed of six Dual Inline Memory Modules (DIMM) with clock speeds of up to 2400 MHz and capacity of 8 GB each. The aggregate Streams Triad bandwidth is 90 GB/s.

3.2.3 Instruction set

Figure 6 shows the KNL's instruction set architecture (ISA) which besides the instructions from the other Intel Xeon processors features support 512 bit vector operations. Those AVX-512 instructions can perform eight double-precision and sixteen single-precision multiply-add operations of the same type at once. Those basic operations are included in the AVX-512 Foundation instructions (AVX-512F). Many other optimizations for various purposes are added by the Intel AVX-512 Conflict Detection (AVX-512CD), Exponential & Reciprocal (AVX-512ER) and Prefetch (AVX-512PF) instructions.

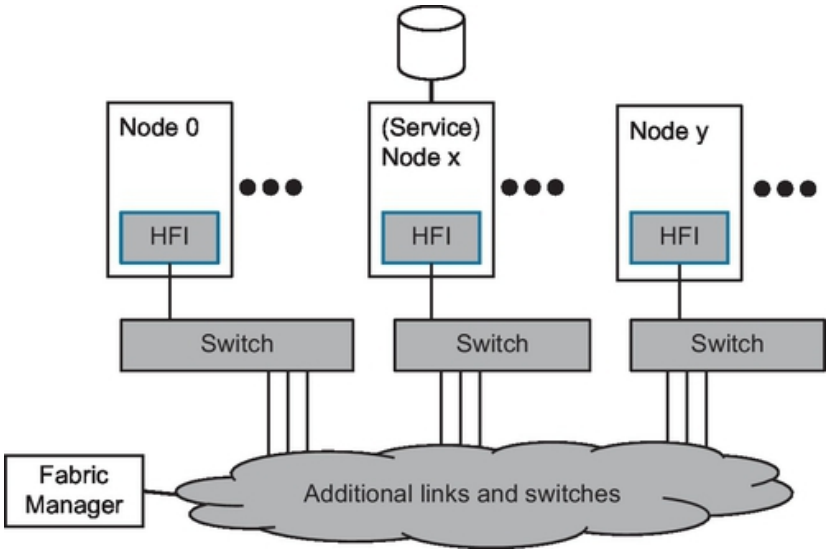

Figure 7: The Omni-Path Architecture connects all nodes via several links and switches and the HFI (source: [9, Chapter I.5]).

3.2.4 Omni-Path

The nodes are connected via an Intel Omni-Path Architecture (OPA) (cf. figure 7) that guarantees high bandwidth (100Gb/s) and low latency (under 110ns) as well as scalability up to hundreds of nodes. Therefore, each host (node) is connected to a fabric of links and switches via the HFI which is used to implement the physical and link layers of the fabric. This enables the nodes to communicate with each other by sending and receiving packets. Those are forwarded by switches which in turn are managed by the Fabric Manager software [9, Chapter I.5].

3.2.5 Input/Output

For Input/Output (I/O) transactions 36 lanes of PCIe Gen3 are used. Those consist of two x16 lane connected to the Omni-Path ports and and one x4 lane for external devices (cf. figure 3).

4 Current QCD libraries

Lattice QCD calculations are performed using predefined classes and routines from Application Programming Interfaces (API). This chapter briefly introduces the currently used libraries Chroma, QDP++ and LibHadronAnalysis. As most of the code was developed almost 15 years ago, a new implementation will be regarded in chapter 5.

4.1 Chroma & QDP++

The Chroma[10] and QDP++ libraries (or data parallel interfaces) contain several algorithms for QCD on the lattice. Besides the application of the internal classes and functions inside a C++ framework, simulations can be run via Extensible Markup Language (XML) input files and information can then be gathered from binary hdf5 output files. A detailed look into the source code is not necessary here, only the underlying data structure will be relevant in the following chapters.
For later comparison the arrangement of data in memory is given by the following order. The outermost index runs over the sites in temporal and spatial directions. Each site then contains a spin matrix of colour matrices storing the real and imaginary part (in the case of complex numbers). This order reflects the physical composition of these tensors, but will be modified in chapter 5.4 for the GRID library implementation.

4.2 LibHadronAnalysis

The LibHadronAnalysis was particularly developed for calculations of Baryon Disribution Amplitudes (BDA), Meson Distribution Amplitudes (MDA) and Baryon Spectra (light and heavy). It provides efficient implementations tailored to computations on these objects on the Many Integrated Core (MIC) architecture and therefore performs better than previous implementations. As input and output parameters Chroma classes are used, so that the LibHadronAnalysis can easily be integrated into normal Chroma code.

5 GRID

As the old libraries mentioned in chapter 4 might not be the best choice for modern computer architectures anymore, the GRID project aims for a better interplay of code and modern hardware. It is currently developed by Peter Boyle, Antonin Portelli, Azusa Yamaguchi[4] and Guido Cossu[5] using the C++11 standard and explicitly addressing lattice QCD simulations on supercomputers.
This chapter introduces the basic data types and tensors of this new C++ data parallel library and their corresponding classes for QCD computations on the lattice. After a brief look at the data structure in memory (as done in chapter 4), a variety of examples for the implemented optimizations are given and eventually the basic employment of the library is demonstrated.

5.1 Basic data types

GRID provides data types that fit the specific architecture of the target machine. These are `Real` and `Complex` with the suffix 'F' for single and 'D' for double precision as well as a prefix 'v' for the vector types (no prefix is scalar), i.e. a vector containing complex double values is denoted as `vComplexD`. Those data type classes essentially wrap the built-in data types and use their inline operator functions for implementations of the arithmetic operations. Consequently, GRID supports several different Single Instruction Multiple Data (SIMD) architectures with the following instruction sets: SSE4 (128 bit) AVX, AVX2, QPX (256 bit), IMCI and AVX512 (512 bit).

5.2 Basic tensors

The following basic classes provide scalars, vectors and matrices. The elements are stored in the member `_internal`, which is respectively a scalar, array or matrix. All major tensors of the QCD section of GRID are composed of those three classes, e.g. the `ColourMatrix` is defined (in QCD.h) as

```
iScalar<iScalar<iMatrix<Complex, Nc> > >.
```

[4]School of Physics, The University of Edinburgh.
[5]Theory Center, IPNS, High Energy Accelerator Research Organization (KEK), Tsukuba.

- **iScalar**

```
template <class vtype>
class iScalar {
  public:
    vtype _internal;
```

Listing 1: Excerpt from the class declaration of iScalar in Tensor_class.h.

- **iVector**

```
template <class vtype, int N>
class iVector {
  public:
    vtype _internal[N];
```

Listing 2: Excerpt from the class declaration of iVector in Tensor_class.h.

- **iMatrix**

```
template <class vtype, int N>
class iMatrix {
  public:
    vtype _internal[N][N];
```

Listing 3: Excerpt from the class declaration of iMatrix in Tensor_class.h.

5.3 QCD

The GRID library was primarily designed to allow for fast calculations in the field of lattice QCD and so it comes with a lot of classes and functions that fit this goal. This section introduces the most important aspects of this QCD module and how to use it.

5.3.1 Conventions

In the code some constants will appear that are frequently used and thus shortly explained in table 1. All matrices are indexed in the order: Lorentz, Spin, Colour. Hence e.g. a `SpinMatrix` is written as `iScalar<iMatrix<iScalar<Complex>,Ns> >`.

	Description	Value
Nd	Number of space-time dimensions	4
Ns	Dimension of the spin vector space	4
Nc	Dimension of the colour vector space	3

Table 1: Summary of QCD relevant constants.

5.3.2 Tensors

The basic QCD tensors are composed of the basic tensors described in 5.2 (using the template typename `vtype`):

$$\begin{aligned}
\texttt{iSinglet} &= \texttt{iScalar} < \texttt{iScalar} < \texttt{iScalar} < \texttt{vtype} >>> \\
\texttt{iSpinMatrix} &= \texttt{iScalar} < \texttt{iMatrix} < \texttt{iScalar} < \texttt{vtype} >, \texttt{Ns} >> \\
\texttt{iColourMatrix} &= \texttt{iScalar} < \texttt{iScalar} < \texttt{iMatrix} < \texttt{vtype}, \texttt{Nc} >>> \\
\texttt{iSpinColourMatrix} &= \texttt{iScalar} < \texttt{iMatrix} < \texttt{iMatrix} < \texttt{vtype}, \texttt{Nc} >, \texttt{Ns} >> \\
\texttt{iLorentzColourMatrix} &= \texttt{iVector} < \texttt{iScalar} < \texttt{iMatrix} < \texttt{vtype}, \texttt{Nc} >>, \texttt{Nd} > \\
\texttt{iDoubleStoredColourMatrix} &= \texttt{iVector} < \texttt{iScalar} < \texttt{iMatrix} < \texttt{vtype}, \texttt{Nc} >>, \texttt{Nds} > \\
\texttt{iSpinVector} &= \texttt{iScalar} < \texttt{iVector} < \texttt{iScalar} < \texttt{vtype} >, \texttt{Ns} >> \\
\texttt{iColourVector} &= \texttt{iScalar} < \texttt{iScalar} < \texttt{iVector} < \texttt{vtype}, \texttt{Nc} >>> \\
\texttt{iSpinColourVector} &= \texttt{iScalar} < \texttt{iVector} < \texttt{iVector} < \texttt{vtype}, \texttt{Nc} >, \texttt{Ns} >> \\
\texttt{iHalfSpinVector} &= \texttt{iScalar} < \texttt{iVector} < \texttt{iScalar} < \texttt{vtype} >, \texttt{Nhs} >> \\
\texttt{iHalfSpinColourVector} &= \texttt{iScalar} < \texttt{iVector} < \texttt{iVector} < \texttt{vtype}, \texttt{Nc} >, \texttt{Nhs} >>
\end{aligned}$$

Those in turn are used to define the complex tensors, such as the following[6]:

$$\begin{aligned}
\texttt{SpinMatrix} &= \texttt{iSpinMatrix} < \texttt{Complex} > \\
\texttt{ColourMatrix} &= \texttt{iColourMatrix} < \texttt{Complex} > \\
\texttt{SpinColourMatrix} &= \texttt{iSpinColourMatrix} < \texttt{Complex} > \\
\texttt{LorentzColourMatrix} &= \texttt{iLorentzColourMatrix} < \texttt{Complex} > \\
\texttt{DoubleStoredColourMatrix} &= \texttt{iDoubleStoredColourMatrix} < \texttt{Complex} > \\
\texttt{SpinVector} &= \texttt{iSpinVector} < \texttt{Complex} > \\
\texttt{ColourVector} &= \texttt{iColourVector} < \texttt{Complex} > \\
\texttt{SpinColourVector} &= \texttt{iSpinColourVector} < \texttt{Complex} >
\end{aligned}$$

Furthermore there are tensors with suffix 'F' or 'D' indicating that `ComplexF`/`ComplexD` is used, as well as tensors with prefix 'v' for the vectorized data types such as `vComplex`, e.g.

```
vSpinMatrixD = iSpinMatrix<vComplexD>.
```

[6]The presented tensor list is not complete.

5.3.3 Lattice class

The `Lattice` template class is the basic class for any lattice QCD related code. The vector `_odata` stores the elements defined by the template argument `vobj` in an order corresponding to its position on the lattice.

```
1 template<class vobj>
2 class Lattice : public LatticeBase {
3   public:
4     Vector<vobj> _odata;
```

Listing 4: Excerpt from the class declaration of Lattice in Lattice_base.h.

With this class the tensors from 5.3.2 are placed on the lattice in the following manner (e.g. for the `LatticeSpinColourMatrix`):

```
LatticeSpinColourMatrix = Lattice<vSpinColourMatrix>
```

In this example an object of type `SpinColourMatrix` is allocated to each single site of the lattice. Hence for the initialization of a `Lattice` object the lattice's layout is required and must be provided to the constructor. This process will be further explained in section 5.7.1.

5.3.4 Physical tensors

Finally, the physical tensors on the lattice are simply written as follows[7]:

$$\begin{aligned} \texttt{LatticeFermion} &= \texttt{LatticeSpinColourVector} \\ \texttt{LatticePropagator} &= \texttt{LatticeSpinColourMatrix} \\ \texttt{LatticeGaugeField} &= \texttt{LatticeLorentzColourMatrix} \end{aligned}$$

5.4 Data structure

The tensors are stored in a specific order so that a high performance can be achieved for most applications. As the `LatticePropagator` will frequently be used in the following sections it shall serve as an example here.
The `LatticePropagator` has a $N_s \times N_s$ spin matrix with elements of type colour matrix ($N_c \times N_c$) for every lattice site. The values are either single or double precision complex numbers and therefore have a real and imaginary part. The total number

[7] This list is just a small excerpt of the whole type definitions in the source code.

of bytes used in such a single precision `LatticePropagator` on a lattice of volume $V = L_1 \times L_2 \times L_3 \times L_4$ can now be calculated as follows:

$$N_{bytes}^{\texttt{LatticePropagator}} = V \times N_s^2 \times N_c^2 \times 2 \times 4 \; bytes \tag{15}$$

where the factor 2×4 corresponds to real and imaginary single precision numbers. To get $N_{bytes}^{\texttt{LatticePropagator}}$ with double precision numbers simply multiply by a factor of two.

The way this tensor is stored in memory is not as straight forward as in the Chroma implementation. There are two sets of indices to be considered: lattice site and `SpinColourMatrix` element. The outermost index (describing the elements furthest away from each other in memory) is the temporal component of the lattice followed by spin and then colour index of the `SpinColourMatrix`[8]. The innermost index is left for the spatial component of the lattice. The real and imaginary parts are stored directly next to each other. The peculiarity of this structure is that the elements with the same spin-colour index on different lattice sites (with same time t) lie directly next to each other in memory. The efficiency of this structure will be investigated later on.

5.5 Optimizations

5.5.1 Matrix-vector product

One of the most frequently used operation in lattice QCD calculations is the computation of matrix-matrix and matrix-vector products. Among these are products of colour (3×3) and spin matrices (4×4). When it comes to an efficient implementation of those operations two problems should be considered. First of all, a 512 bit vector with eight single precision complex numbers is suboptimal for a 3×3 matrix with nine elements. Secondly, after the scalar multiplication of the matrix/vector elements and accumulating them in a SIMD vector, one needs to horizontally sum over those results at the end of each row. This leads to a reduction in performance as this addition can only be calculated after all multiplications are done.

The solution to these observations lies in the computation of several matrix-vector products at once [12, Section 2.1] (see figure 8). The SIMD lane now consists of elements with the same index but in different matrix-vector pairs (cf. figure 8). For the right number of such pairs this process is up to 100% efficient. GRID uses the implementation given in listing 5. Here, the template type `simd` can be as simple as a `float` for a standard matrix-vector product or one of GRID's vector types. One should realize that the utilization of the GRID types automatically leads to the optimized approach discussed above.

[8] The Lorentz index is left out for simplification.

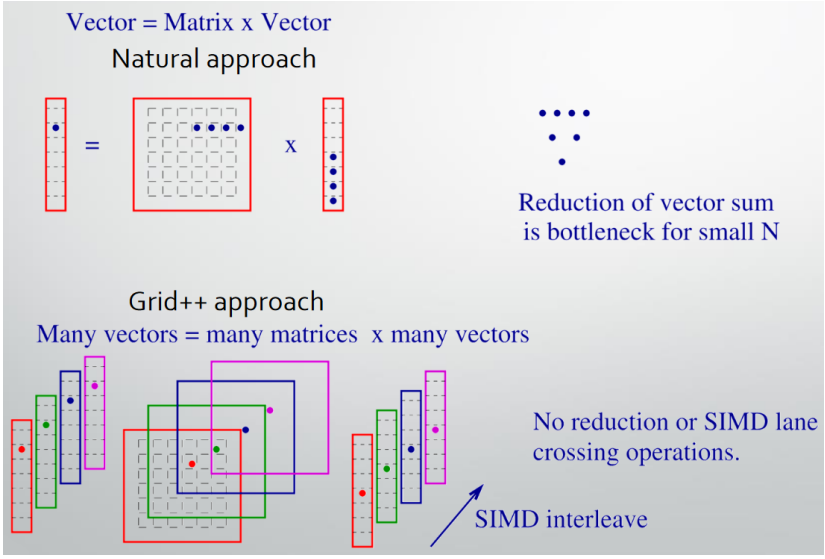

Figure 8: Top: Natural approach of a matrix-vector multiplication. Bottom: GRID's approach for many matrix-vector pairs (source: [11]).

```
inline template<int N, class simd>
void matmul(simd* x, simd* y, simd* z)
{
   for(int i=0;i<N;i++){
      for(int j=0;j<N;j++){
         x[i] = x[i]+y[i*N+j]*z[j];
      }
   }
}
```

Listing 5: Matrix-vector multiplication in GRID.

5.5.2 Circular shift function

The circular shift (`Cshift`) of the lattice is a function that will later be used for the computation of derivatives. It basically takes a `Lattice` object and returns an instance of the same type that is shifted by an arbitrary number of sites in a given direction (shown in listing 6).

```
LatticePropagator prop1(&Grid), prop2(&Grid);
int dimension = 0;
int length = 1;
prop2 = Cshift(prop1,dimension,length);
```

Listing 6: The `Cshift` function returns a `LatticePropagator` that has been shifted by one unit in dimension zero.

By shifting the lattice at the surface of a sub-lattice, some sites may now become part of a new neighbourhood. In order to fix this, a permutation as shown in figure 9 can be applied onto those elements. This is SIMD efficient as well as suppressed by the small surface to volume ratio [12, Section 2.3].

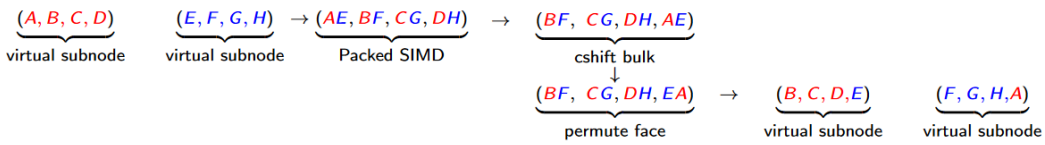

Figure 9: `Cshift` operation with permutation of sites A and E. A becomes now part of the blue sub-lattice and B part of the red sub-lattice so that locality is restored (source: [12, Figure 3]).

5.6 Over-decomposed layout

In order to reach a maximum efficiency of the SIMD architecture, the lattice is over-decomposed into more sub-lattices than the number of Message Passing Interface (MPI) tasks would suggest [12, Section 2.2]. Thus, one physical node may hold more than one virtual node, each handling one sub-lattice. When an operation has to be made, GRID is now able to fill the SIMD vectors in an optimal way, using data from different virtual nodes in one vector if possible. The detailed mapping for a given GRID data type on a target machine with a certain instruction set is shown in table 2. This way multiple GRID objects with arbitrary dimensions can be active in one programme and conformable operations on the same GRID object can be done parallel and efficient.

ISA	vRealF	vRealD	vComplexF	vComplexD	default layout
SSE	4	2	2	1	1 1 1 2
AVX	8	4	4	2	1 1 2 2
AVX512	16	8	8	4	1 2 2 2

Table 2: Vector lengths for different instruction sets and the corresponding over-decomposed default layout (x,y,z,t).

5.7 Basic usage

As later code snippets might become easier to understand with some previous knowledge about the most common classes and functions, this section serves as an introduction for those who are about to read or even write GRID programs.

5.7.1 Initialization

The `main` function of every GRID program starts with an initialization `Grid_init()` and ends with the corresponding `Grid_finalize()` function call, as demonstrated in listing 7.

```
#include <Grid.h>
using namespace Grid;

int main(int argc, char ** argv)
{
  Grid_init(&argc,&argv);
  // do something ...
  Grid_finalize();
}
```

Listing 7: Basic pattern of a GRID program.

The next step is setting up the grid that contains all necessary information for the communication and organization of the lattice data. This can be achieved by calling the `GridCartesian` constructor (listing 8) which needs to know the size of the lattice, the SIMD layout and the MPI layout. All those parameters can easily be obtained by calling the `GridDefaultLatt()`, `GridDefaultSimd()` and `GridDefaultMpi()` functions. `GridDefaultLatt()` returns a {8,8,8,8} vector and thus corresponds to a 8^4 lattice. `GridDefaultMpi()` returns a {1,1,1,1} vector. Both vectors can also be set as command line arguments `--grid n.n.n.n` and `--mpi n.n.n.n`, so that in case of multiple tasks the lattice is distributed among the corresponding cores and/or processors. The programmer has to make sure that the product of the elements equals the number of tasks in the MPI environment.

```
std::vector<int> latt_size   = GridDefaultLatt();
std::vector<int> simd_layout = GridDefaultSimd(Nd,vComplex::Nsimd());
std::vector<int> mpi_layout  = GridDefaultMpi();
GridCartesian Grid(latt_size,simd_layout,mpi_layout);
```

Listing 8: Setting up a default grid.

When the `GridCartesian` is set up it must also be used for the initialization of `Lattice` types, as done in listing 9. This is mandatory because there is no default initialization for those `Lattice` types.

```
LatticePropagator prop(&Grid);
//LatticePropagator prop; <- no default constructor
```

Listing 9: Initialization of a `Lattice` type with a `GridCartesian`.

5.7.2 Peek & poke

The GRID library provides peek and poke functions allowing for a comfortable handling of the lattice tensors. In practice, one obtains spin, colour and Lorentz elements by applying the `peekSpin()`, `peekColour()` and `peekLorentz()` functions on the tensors. The latter can be manipulated by `pokeSpin()`, `pokeColour()` and `pokeLorentz()` (see listing 10). Individual sites of `Lattice` objects can be addressed via `peekSite()` and `pokeSite()` (see listing 11).

```
1 LatticeSpinMatrix spinMat(&Grid);
2
3 LatticeComplex result(&Grid);
4 result = peekSpin(spinMat,0,0);
5
6 LatticeComplex c(&Grid);
7 pokeSpin(spinMat,c,0,0);
```

Listing 10: `Peek` & `poke` functions applied to a `LatticeSpinMatrix`.

```
1 LatticeColourMatrix lcMat(&Grid);
2 ColourMatrix cMat(&Grid);
3 std::vector<int> site = {2,3,1,3};
4
5 pokeSite(cMat,lcMat,site);
6 peekSite(cMat,lcMat,site);
```

Listing 11: Manipulation of a certain site on a `LatticeColourMatrix` via `peek` and `poke` functions.

As the data of GRID's classes (stored in `_internal`) is public, one might also address it directly as shown in listing 12.

```
1     LatticeSpinMatrix spinMat(&Grid);
2
3     //Set the spin of spinMat to 42 on each site:
4     for(int i=0; i<spinMat._odata.size(); i++) {
5       spinMat._odata[i]._internal._internal[0][0]._internal = 42;
6     }
7     //Print spin of site 0:
8     std::cout << spinMat._odata[0]._internal._internal[0][0]._internal;
```

Listing 12: Direct manipulation of a certain site on a `LatticeSpinMatrix` without `peek` and `poke` functions.

6 Comparison

After the basics about lattice QCD, QPACE 3 and the software have been introduced in the previous chapters, the task is now to apply this knowledge. The libraries can be compiled, the benchmarks can be written and evaluated. With regard to the implementation of the MDA in chapter 7, the code is set up to test the performance of the computation of correlation functions and derivatives, the main ingredients of the MDA calculation.

6.1 Configurations

The GRID library is available on Github[9] in different versions (or branches). Among those are e.g. two releases (release/v0.6.0 and release/v0.7.0), a branch for the current development (develop) and of course a master branch. In order to get the results based on the newest optimizations and developments, the later discussed benchmarks will be done with the develop branch.
For the compilation GRID supports Intel's ICPC, Clang and GCC compilers. For this thesis several builds were set up with those compilers except for Clang. For the communication, several versions of MPI were used, namely Open MPI, MVAPICH 2 and Intel MPI.
In order to get the best comparability between GRID and Chroma, the latter was also compiled with the compilers above and MVAPICH 2 and Intel MPI. All configurations use double precision numbers. Additionally, GRID features a configuration option (`--enable-simd=KNL`) particularly designed for the SIMD configuration on the KNL. It replaces the `--enable-simd=AVX512` option that enables 512 bit vector operations. For the evaluation of the efficiency of these vector sizes in section 6.2.4, the GRID GCC MVA builds have also been compiled with 256 bit (`AVX2`) and 128 bit (`SSE4`) instructions. Table 3 gives a detailed overview of the exact compiler and MPI versions chosen and the abbreviations which will from now on be used in this document.

Abbreviation	Library	Compiler	MPI
Chroma GCC MVA	Chroma	GCC 6.2.1	MVAPICH 2-2.2-psm2
Chroma ICC Intel	Chroma	ICC 17.0.2	Intel MPI library 2017.2
GRID GCC Open	GRID	GCC 6.2.1	Open MPI 1.10.3
GRID GCC MVA	GRID	GCC 6.2.1	MVAPICH 2-2.2-psm2
GRID ICC Intel	GRID	ICC 17.0.2	Intel MPI library 2017.2

Table 3: Chroma and GRID were compiled with two different compilers and two/three MPI versions each.

[9] https://github.com/paboyle/Grid.

6.2 Basic benchmarking

Now that the libraries are compiled, code for the testing process is required that executes the same tasks for the GRID as well as the Chroma library. So there will be a framework where the initialization of objects takes place and a section where the actual computation takes place. The latter will be done several times (typically 1000 or 10000) inside a loop in order to get a good estimate of the average performance and to be able to compute the error.

6.2.1 Source code

The two benchmark files discussed in this section can be run with several command line arguments. The most important ones are `--mpiLayout` which requires four arguments indicating the number of tasks in the respective direction and `--lattice` also with four arguments indicating the number of sites in each direction, where the last one is thought to be in the temporal direction. These are used for the initialization of a `GridCartesian` object as shown in listing 8. Among the other options are settings for the number of threads and loops.

benchmarkCorrelation.cpp This program basically just tests the capability of multiplying `Lattice` objects, in this case `LatticePropagators`. There is no communication between the tasks involved. As an example case, a correlation function is computed, like the one introduced by equation (8). The output then consists of the time used per iteration and the number of GFlops per second (each with error). Listing 13 shows an excerpt of the full code used for the computation with GRID. This part is almost identical to the one written for Chroma.

```
Gamma gamma5(Gamma::Algebra::Gamma5);
LatticePropagator quark_propagator(&Grid);
LatticeComplex corr(&Grid);

LatticePropagator anti_quark = gamma5 * quark_propagator * gamma5;
anti_quark = adj(anti_quark);

double timeData[nLoops];

for(int i=0; i<nLoops; i++) {
  double start = usecond();

  corr = trace(anti_quark * gamma5 * quark_propagator * gamma5);

  double stop = usecond();
  timeData[i] = stop-start;
}

std::cout << corr << std::endl;
```

Listing 13: Excerpt of the most important parts from benchmarkCorrelation.cpp (GRID version).

benchmarkDerivative.cpp This benchmark differs from the previous one only by the line where the computation takes place (see listing 14). This is again just a `LatticePropagator` multiplication but the quark propagator is now replaced by its covariant derivative (cf. section 2.2), which contains the circular shift function `Cshift` from section 5.5.2. For the shift of the propagator into any direction communication between the MPI ranks and also between the tiles is required on the borders of the sublattices.
The covariant derivative is already implemented in Chroma, thus it can be easily computed by calling the `rightNabla` function and the whole computation can be done in one line of code (see Listing 15).

```
Gamma gamma5(Gamma::Algebra::Gamma5);
LatticePropagator quark_propagator(&Grid);
LatticeComplex corr(&Grid);

LatticeGaugeField Umu(&Grid);
LatticeColourMatrix gField = U[mu];
LatticePropagator tmp(&Grid);

LatticePropagator anti_quark = gamma5 * quark_propagator * gamma5;
anti_quark = adj(anti_quark);

double timeData[nLoops];

for(int i=0; i<nLoops; i++) {
  double start = usecond();

  ctmp = adj(gField) * quark_propagator;
  corr = trace(anti_quark * gamma5 * (gField*
         Cshift(quark_propagator,mu,length)-Cshift(tmp,mu,-length))*
         gamma5);

  double stop = usecond();
  timeData[i] = stop-start;
}
```

Listing 14: Excerpt of the most important parts from benchmarkDerivative.cpp (GRID version).

```
corr = trace(anti_quark * gamma5 *
       rightNabla(quark_propagator,u,mu,length) * gamma5);
```

Listing 15: In the Chroma version of benchmarkDerivative.cpp the computation inside the loop can be done in one line, as the covariant derivative is implemented by the `rightNabla` function.

6.2.2 Execution

After compilation the programs can now be executed on the QPACE 3 via "slurm", a workload manager and job scheduling system, designed for small and large clusters. A slurm script as shown in listing 16 must be written and submitted in order to run a certain command on a cluster as the QPACE 3. This script consists of a header which contains information concerning the job itself and the cluster. Among these are the job's name and estimated duration, the number of nodes, tasks per node and cores per task. Additionally one is able to specify a working directory where the output is written

to, the partition of the cluster (in this case "qp3" for QPACE 3) and the maximum number of switches used for the job allocation (always 1 on the QPACE 3). After that the actual commands are executed, in this case loading the right MPI version (Mvapich, Open MPI or Intel), setting the number of threads used per task and pinning those threads to certain cores[10]. The latter is done in order to gain performance and reduce the variability of computation times as OpenMP typically would migrate some of the processes from one core to another. The bash script `get_thread_pinning_env.sh` takes three arguments:

1. **`intel or gnu/gcc`**: Defines the OMP runtime library
2. **`1/2/3/4`**: Defines the number of threads per core
3. **`scatter/compact`**: Defines the affinity setting (irrelevant in the case of one thread per core)

This script then returns the corresponding pinning environment, e.g. in the case of "`intel 1 scatter`":

```
export KMP_PLACE_THREADS=1T;
export KMP_AFFINITY=scatter
```

The last command in the slurm script eventually calls the executable. Two different options are used in this line: `--cpu_bind=sockets` & `rank_ldom`. The first one is applied in combination with one task per node and 64 cores per task and binds the tasks to the CPU's with respect to the whole socket, which is in this case one KNL. As the processors of QPACE 3 are currently using the SNC-4 mode (discussed in chapter 3.2), it might be beneficial to use four tasks per node with 16 cores each and then set `--cpu_bind=rank_ldom`, which binds the cores according to the NUMA domains. Those two options will later be notated as "Socket" and "NUMA".
While the `srun` command is used for the Mvapich and Intel MPI scripts, the Open MPI scripts employ `mpirun` with options `--map-by socket` or `numa` and `--bind-to socket` or `numa` (cf. listing 17).

[10]The script used for thread pinning has been written by Daniel Richtmann (University of Regensburg).

```
#!/bin/bash
#
#SBATCH --job-name=benchmark
#SBATCH --time=1:30:00
#SBATCH --partition=qp3
#SBATCH --switches=1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1 # or 4
#SBATCH --cpus-per-task=64  # or 16
#SBATCH --workdir=/home/fim16418/slurm/grid/corr

module load mpi/mvapich2-2.2-psm2-x86_64
#module load mpi/openmpi-x86_64
#module load mpi/intel/2017.2.174

export OMP_NUM_THREADS=64 # or 16

thread_pinning_env=$(get_thread_pinning_env.sh gnu 1 scatter);
$thread_pinning_env

srun --mpi=pmi2 --cpu_bind=sockets benchmarkCorrelation
#srun --mpi=pmi2 --cpu_bind=rank_ldom benchmarkCorrelation
```

Listing 16: This slurm script example already shows the most common instructions used in this thesis.

```
MPIENV="--map-by socket --bind-to socket"
#MPIENV="--map-by numa --bind-to numa"
mpirun $MPIENV benchmarkCorrelation
```

Listing 17: Execution of a program compiled with Open MPI.

6.2.3 Number of threads

As already determined in the slurm script from listing 16, 64 (Socket) or 16 (NUMA) threads per task are employed. This number is not arbitrary but was obtained by a few benchmarks depicted in figures 10 and 11. Here, the code from benchmarkCorrelation.cpp ran on one node with one task (Socket) or four tasks (NUMA) and a 8^4 lattice.

It can easily be seen that the best performance is obtained at a number of threads that is divisible by the number of cores, e.g. 64 and 128 for 64 cores. Between those peaks some cores may have been assigned fewer threads than others and hence they have to wait for those to finish. Despite the fact that GRID acts in a more inconsistent manner for 129 to 256 threads, all builds roughly show the same behaviour. The performance

breaks down if more threads are allocated than the maximum of four threads per core. For all the following benchmarks the number of threads has been set to equal the number of cores in order to get a peak performance and to be able to pin the threads accordingly.

Determination of the number of threads (Socket)

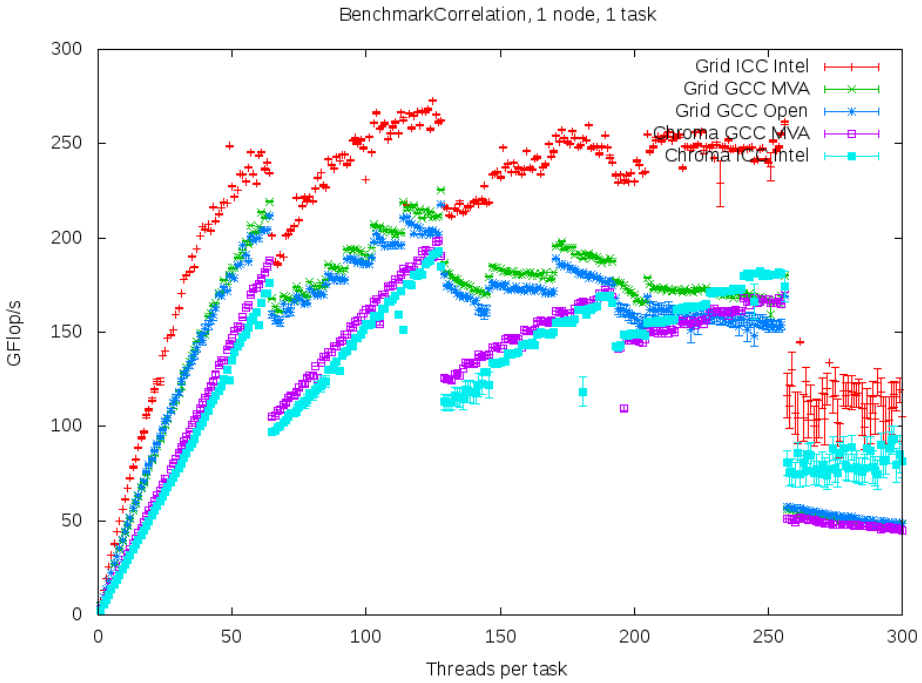

Figure 10: This benchmark shows the GFlop/s vs. threads per task for builds on one node with one task. It applied the correlation benchmark on a 8^4 lattice.

6.2.4 Vectorization

One novelty of the KNL is (as already stated in 3.2) the support for 512 bit vector instructions. Figures 12 and 13 show the number of GFlop/s versus the lattice volume (for lattices of size $2^4, 4^4, 6^4, ...$). As the volume increases, so does the performance for the simple reason that for very small lattices there is too few data for too many cores. E.g. a 2^4 lattice has only 16 sites and for a total of 64 cores each core handles on average only 0.25 sites. This effect soon vanishes as the lattice grows.
Certainly, the 512 bit builds in figures 12 and 13 deliver the best performance as they are able to compute up to four times the amount of data as the 128 bit build. But the GRID GCC MVA build is not as efficient as this factor of four would suggest. Regarding the number of flops on the 36^4 lattice, the expected factor of two is observed between the 256 bit and 128 bit point. The 512 bit (GCC MVA) point however only gives a factor of three instead of four. This can be observed in both cases, socket and NUMA. The ICC Intel build however delivers the expected performance in both plots. This is not surprising as Intel's compiler is naturally the best choice for the Intel product KNL.

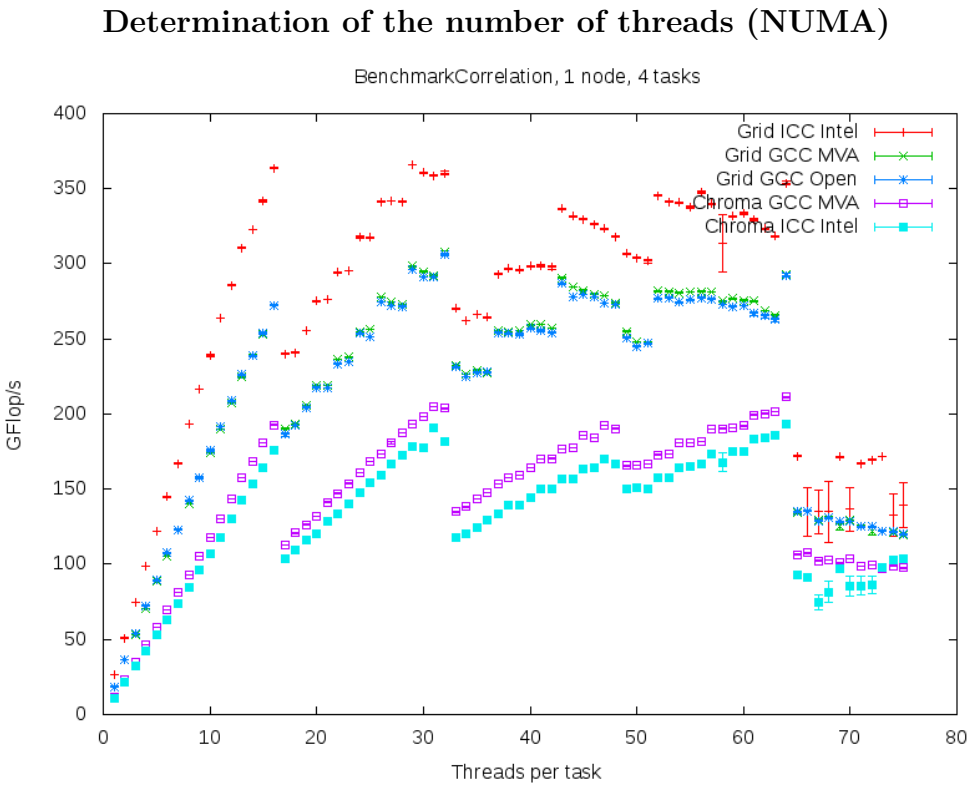

Figure 11: This benchmark shows the GFlop/s vs. threads per task for builds on one node with four tasks. It applied the correlation benchmark on a 8^4 lattice.

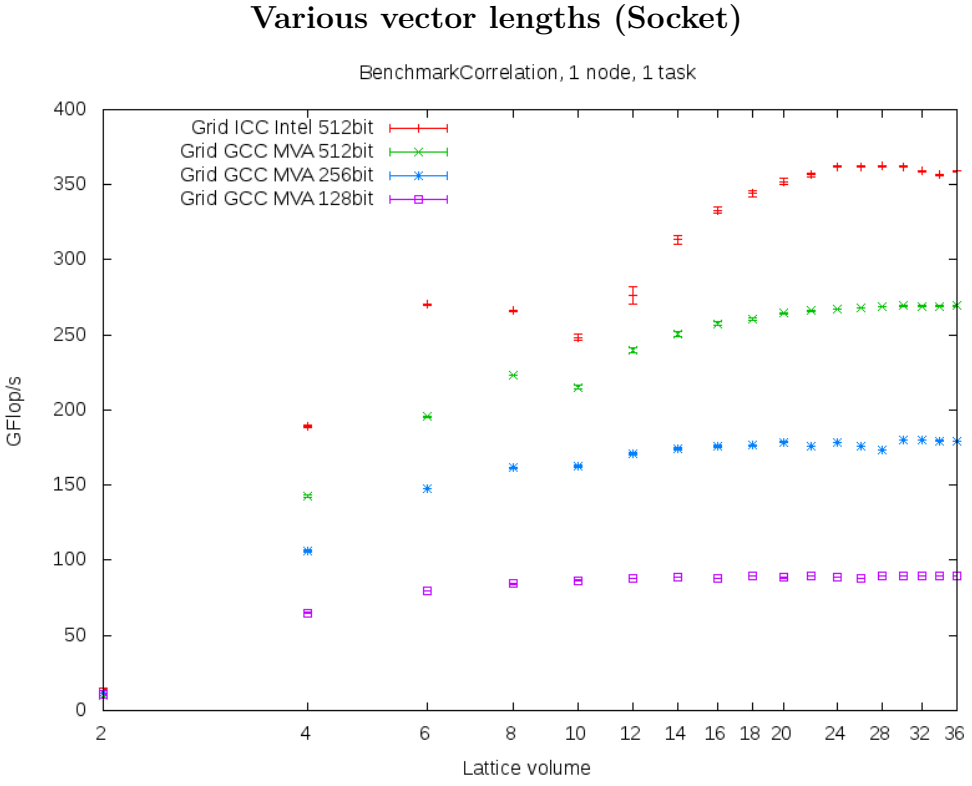

Figure 12: Correlation benchmark for GRID GCC MVA builds with different vector lengths and the 512 bit GRID ICC Intel build. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

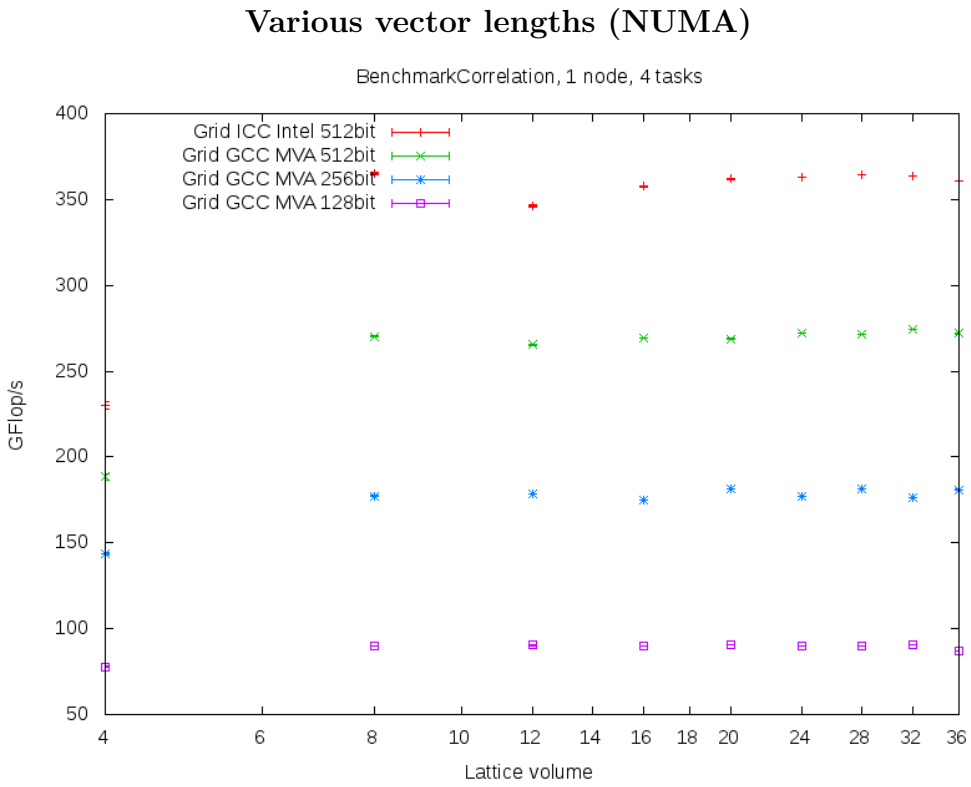

Figure 13: Correlation benchmark for GRID GCC MVA builds with different vector lengths and the 512 bit GRID ICC Intel build. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

6.3 Benchmarks

After all these preparations, the actual correlation and derivative benchmarks can be finally run and evaluated.

6.3.1 Correlation benchmarks

One node The correlation benchmark does not require any kind of communication between the cores and thus poses a simple way of comparing the performances of GRID and Chroma when it comes to matrix multiplications on the lattice. Figure 14 shows the average number of GFlop/s (with errors) on lattices of different sizes, computed on one node with one task.

Clearly in this case, all GRID benchmarks yield higher performances compared to Chroma. GRID GCC MVA (green) and GRID GCC Open (blue) are approximately even as the only difference between these two is the choice of MPI, which is of course irrelevant here as there is no MPI communication involved. While for the Chroma benchmarks there is no big discrepancy for the Intel and GCC compiler versions, GRID is able to benefit from the ICC due to the better usage of the vector registers (as discussed in section 6.2.4).

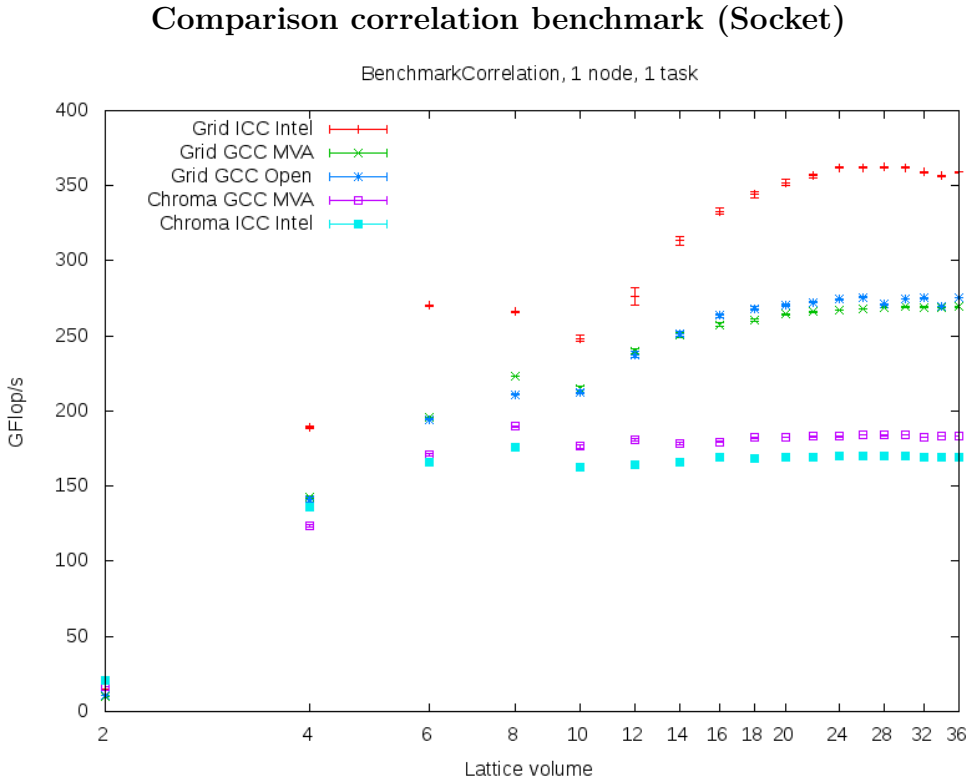

Figure 14: Correlation benchmark for all GRID and Chroma builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

All curves show but a similar behaviour. The small descent between the 8^4 and 10^4 lattice results from the tensors not fitting into the L2 cache (32MB) anymore, because one (of at least two) `LatticePropagator` on a 10^4 lattice already needs about 23MB of memory:

$$\begin{aligned} N_{bytes}^{\texttt{LatticePropagator}} &= V \times N_s^2 \times N_c^2 \times 2 \times 8 \; bytes \\ &= 10^4 \times 4^2 \times 3^2 \times 2 \times 8 \; bytes \\ &= 23\,040\,000 \; bytes \\ &\approx 23 \; MB \end{aligned}$$

Hence the data required is only accessible with a higher latency. From this point on, GRID in contrast to Chroma manages to gain computational power as the lattice volume increases. This can most probably be assigned to a more effective pipelining. The same benchmarks were run on one node with four tasks (see figure 15) and they show a similar picture. Chroma is not really affected from this change in the number of tasks but GRID already shows the maximum performance for lattices of 8^4 and bigger.

Multiple nodes The purpose of the next benchmarks is now to go from computations on one node to a cluster of nodes as can be found on the QPACE 3. The data will be

presented as a comparison of GRID and Chroma on one, two and four nodes with one or four tasks each. Additionally, the plots are divided into ICC Intel and GCC MVA versions. The x-axis represents the lattice volume per node (*weak scaling*) so that e.g. the total lattice volume on four nodes is four times the local volume on one node, etc. The y-axis now shows the GFlop/s per node in order to make the benchmarks more comparable.

Figure 16 is a comparison of GRID GCC MVA and Chroma MVA on various numbers of nodes with one task per node. The general behaviour of the performances on multiple nodes reflects the one from only one node. To be more precise, the total number of GFlop/s increases linearly with the number of nodes. For big lattices GRID overall reaches roughly 50% more GFlop/s than Chroma. This difference even increases for the ICC Intel builds in figure 17. Here, the performance of GRID is around two times the performance of Chroma, so that GRID requires only half of the nodes for the same computational power (on large lattices).

While for the same computations with four tasks per node (see figure 18) Chroma stays unchanged, GRID reaches the peak performance already on a 8^4 lattice (per node).

In figure 19 the same procedure is repeated with the ICC Intel builds resulting again in higher performances for GRID compared to GRID GCC MVA. Comparing the socket and NUMA performances it is obvious that Chroma is not affected by the NUMA setting, in contrast to GRID.

Comparison correlation benchmark (NUMA)

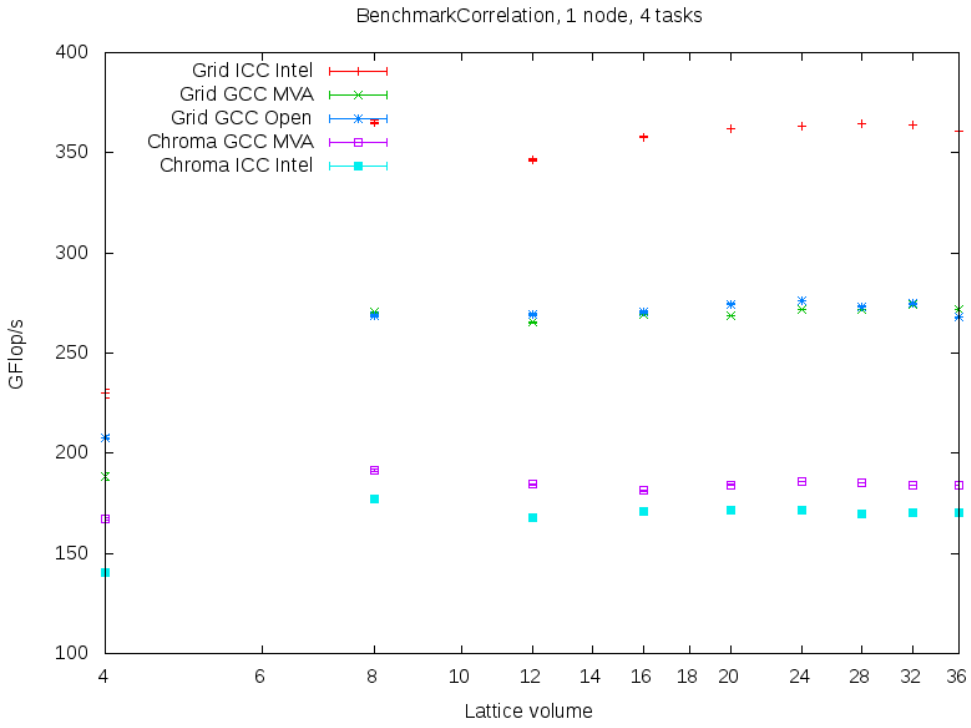

Figure 15: Correlation benchmark for all GRID and Chroma builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

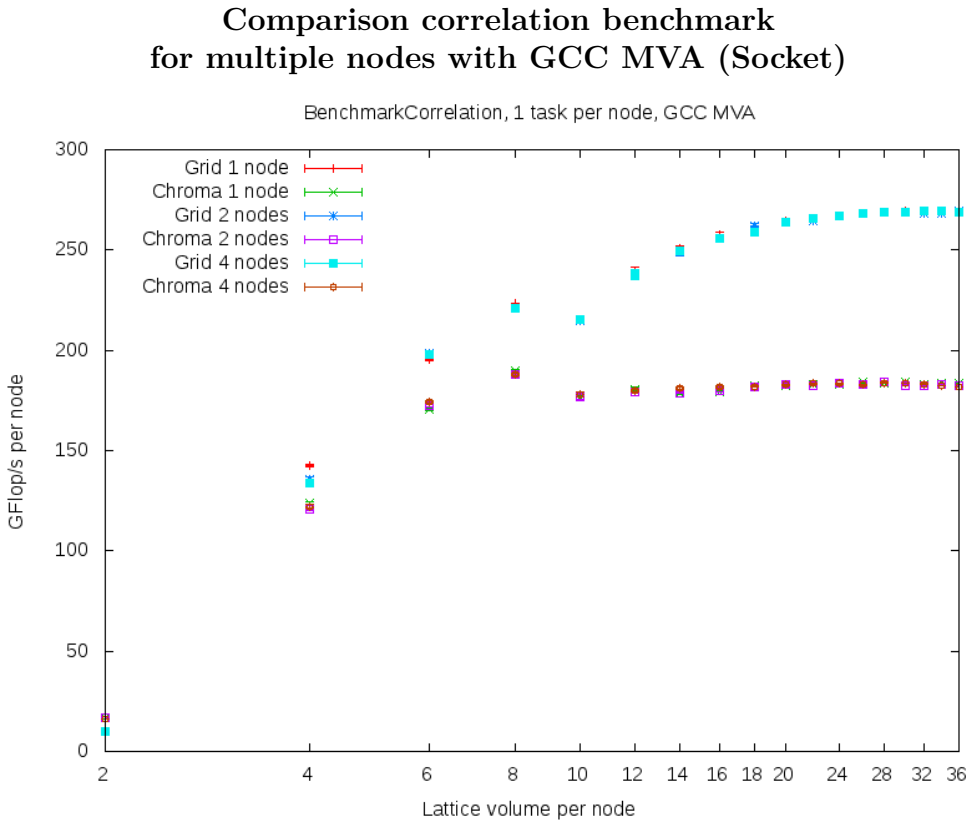

Figure 16: Correlation benchmark for the GCC MVA builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

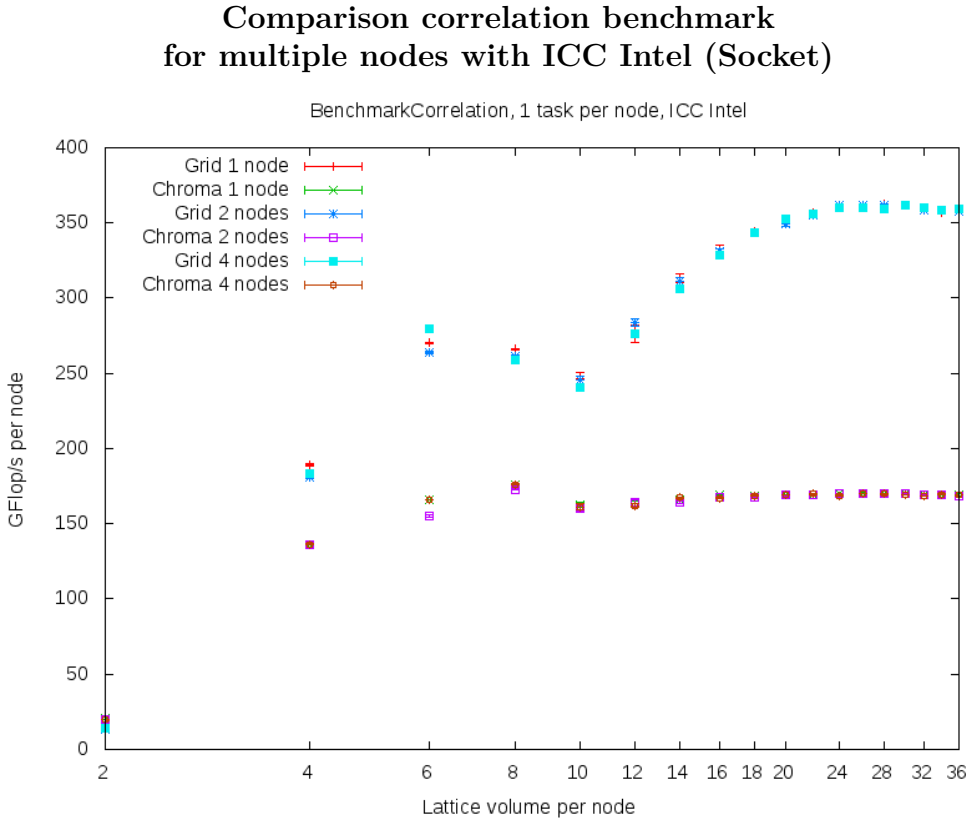

Figure 17: Correlation benchmark for the ICC Intel builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

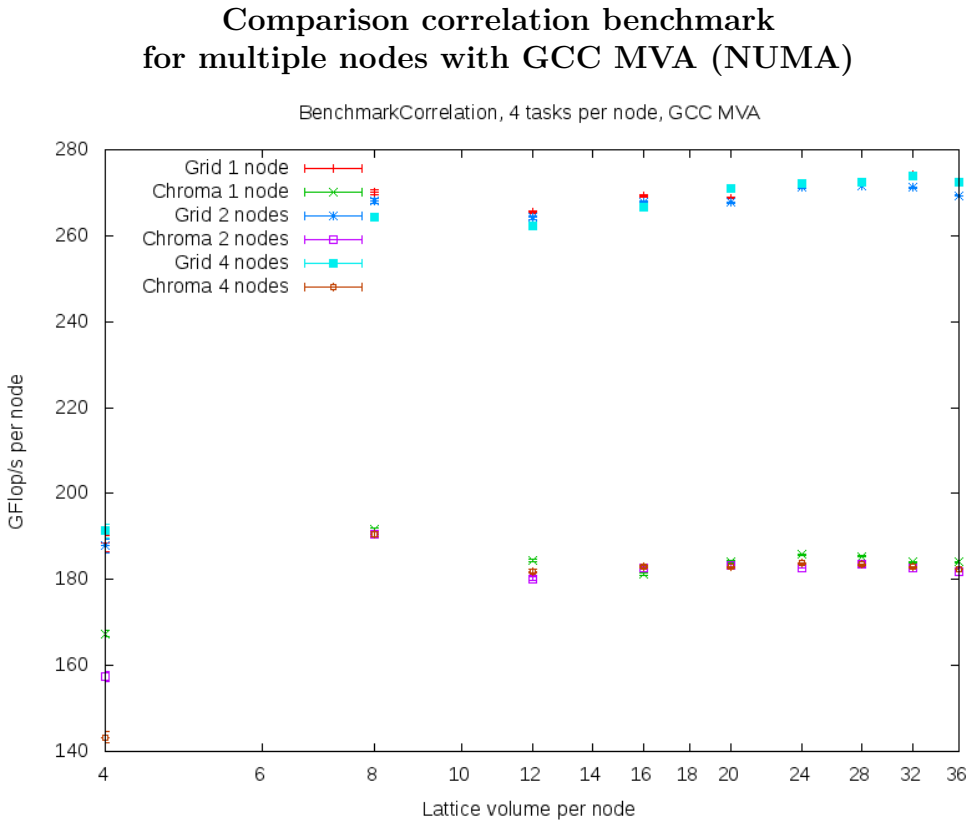

Figure 18: Correlation benchmark for the GCC MVA builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

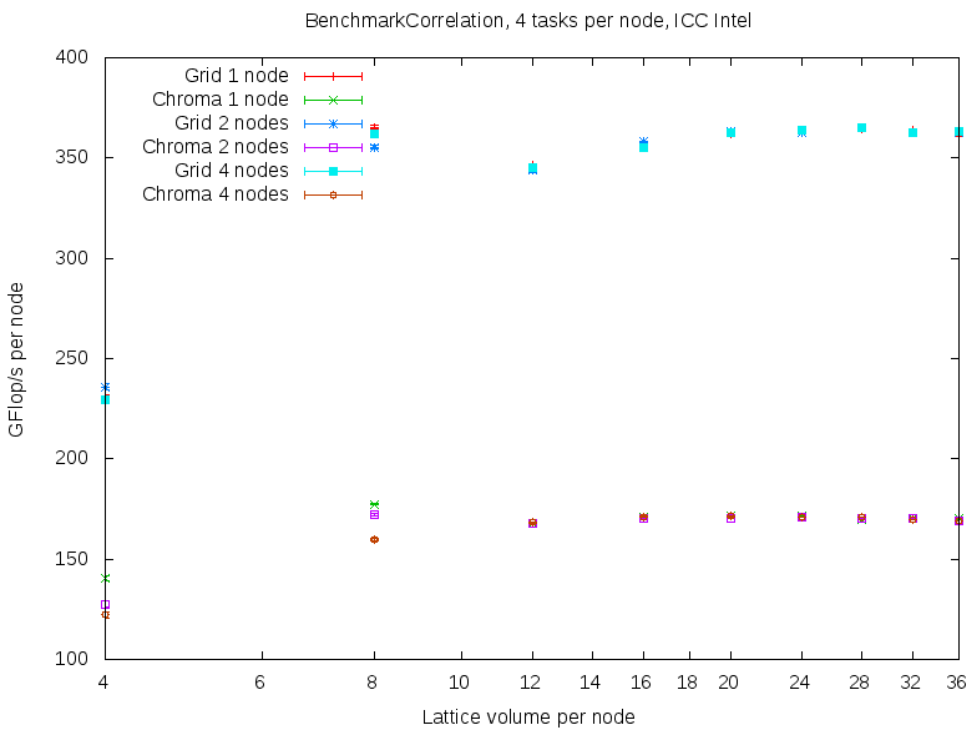

Figure 19: Correlation benchmark for the ICC Intel builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

6.3.2 Derivative benchmarks

In contrast to the previous benchmarks, this one employs communication between the cores, tasks on one node and tasks on different nodes. The crucial point here is the `Cshift` function inside the benchmarked computation. By means of the following plots the impact of this communication shall be observed and discussed in both libraries, GRID and Chroma. The whole setup and procedure is the same as for the correlation benchmarks, except for the code and consequently the amount of allocated memory (i.e. more `LatticePropagator`s are involved here).

One node At first, the benchmarks are again run on only one node with one task (see figure 20) and four tasks (figure 21). The obvious difference to the plots before is that for the socket setup Chroma is faster than GRID by around 50% (for big lattices). The communication's impact is in this case not as grave as for GRID, for which the performance is now only a fraction of the one obtained in figure 14.
Regarding the overall behaviour of all benchmarks in figure 20 the performance drop for the data not fitting into the L2 cache lies here between the 6^4 and 8^4 volume on account of more `Lattice` objects, i.e. the shifted `LatticePropagator`s and the gauge links. Eventually, the benchmark results show a drop in performance for bigger lattices because the sublattices do not fit into the L2 cache anymore and as a consequence not only the responsible cores have to get the data from the DDR if needed, but also if requested from other cores for the computation of the circular shift. This becomes more and more problematic as the number of sites per sublattice increases.
The NUMA plot in figure 21 gives but a different impression. Chroma is - like in the correlation benchmarks - only scarcely touched by this change from one to four tasks whereas GRID is able to draw even with Chroma's performance. GRID ICC Intel is again the build with the best results, outrunning Chroma by a small factor.

Multiple nodes For the comparison on multiple nodes, GRID's and Chroma's derivative benchmarks are run on one, two and four nodes with one or four tasks each. The plots for one task per node with GCC MVA (figure 22) and ICC Intel (figure 23) confirm the observations from above on just one node. The overall behaviour stays the same for more than one node and Chroma remains the faster library in this case, although some points from the GRID ICC Intel build come close to the ones from Chroma. Furthermore GRID shows a wider spread of performances on different numbers of nodes, especially in figure 23.
The benchmarks are repeated with four tasks per node and the results are shown in figures 24 (GCC MVA) and 25 (ICC Intel). Again the conclusions are the same as for one node as the GFlop/s scale linearly with the number of nodes.

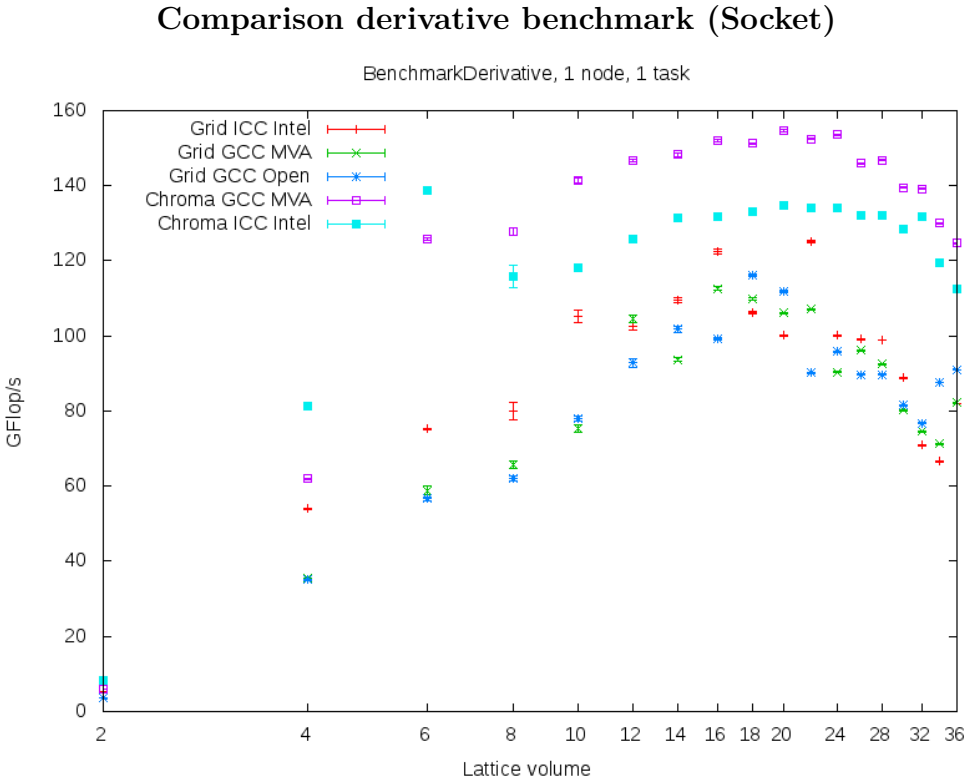

Figure 20: Derivative benchmark for all GRID and Chroma builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

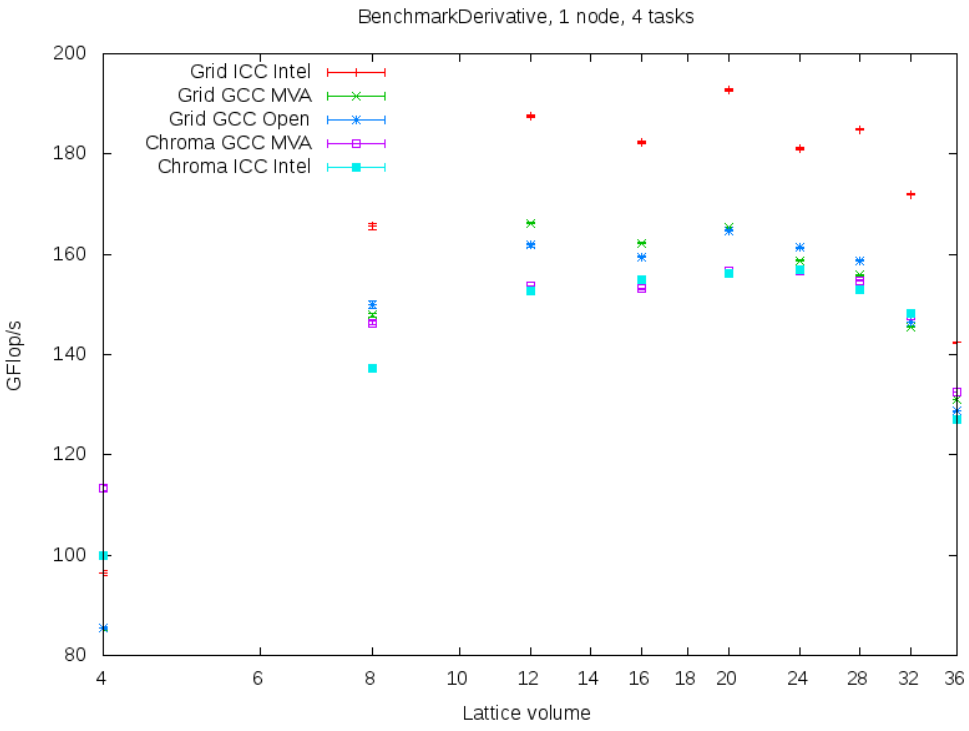

Figure 21: Derivative benchmark for all GRID and Chroma builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

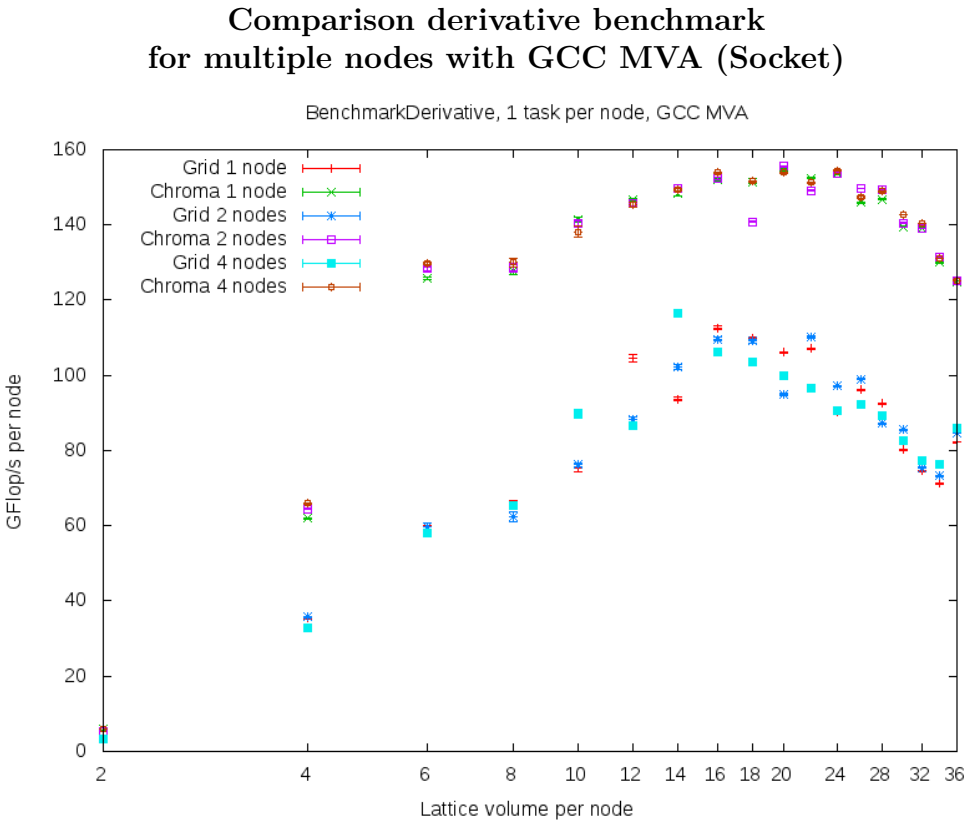

Figure 22: Derivative benchmark for the GCC MVA builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

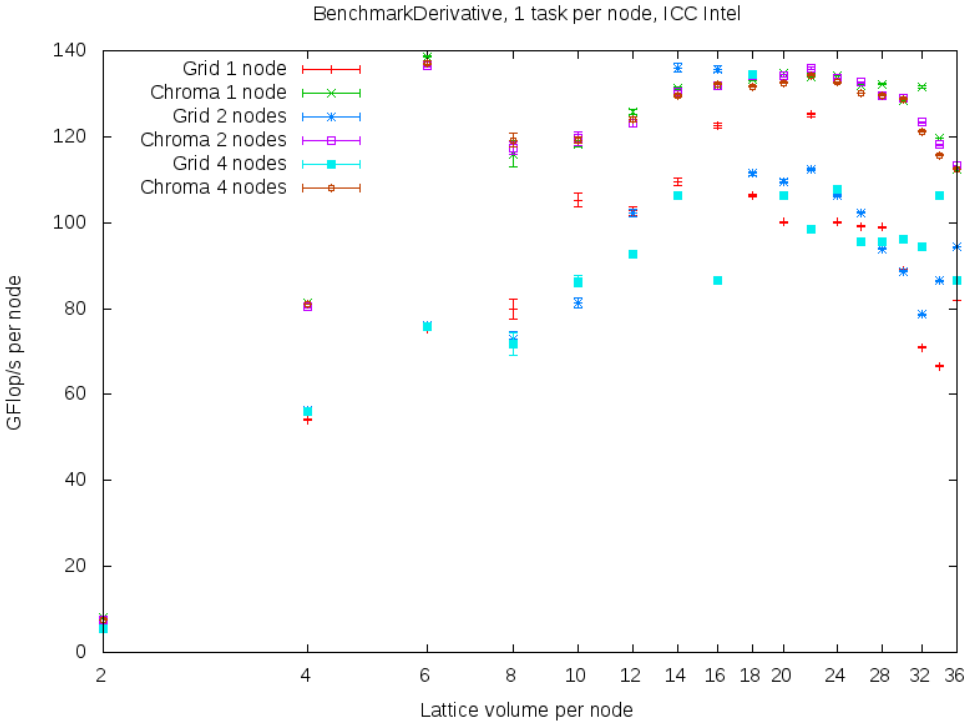

Figure 23: Derivative benchmark for the ICC Intel builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

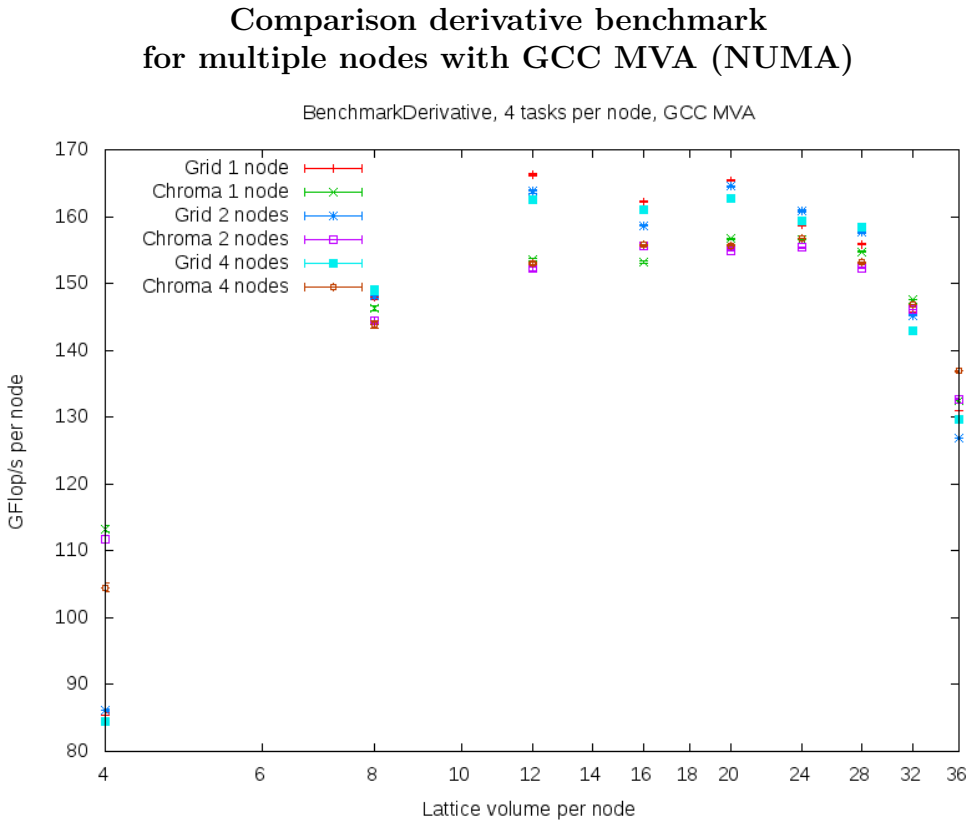

Figure 24: Derivative benchmark for the GCC MVA builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

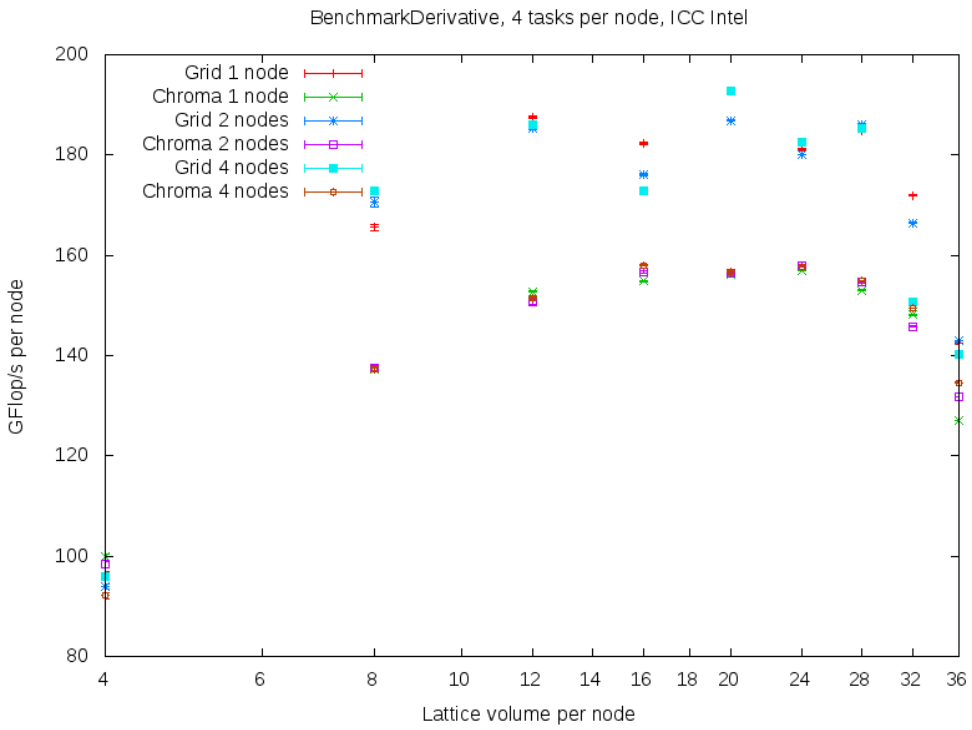

Figure 25: Derivative benchmark for the ICC Intel builds. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$) and given per node.

7 Meson Distribution Amplitude

In contrast to the previous chapter, the following discussion is about whether a sufficiently efficient implementation of the MDA can be found using the GRID classes and functions. The results are compared to the already optimized implementations in LibHadronAnalysis.

7.1 MDA implementations

For the computation of the MDA with the GRID library essentially two approaches are tested. The first one can be found in algorithm 1. It takes two propagators as parameters and contracts them in two steps: preparation and computation. In the first part the colour matrices are extracted from the propagators and stored in arrays where the index corresponds to the original spin index. After that, the MDA is computed by taking the (colour) trace over those arrays with all possible spin index combinations. The implementation of this function in GRID is shown in listing 18.

```
1  LatticePropagator p1(&Grid);
2  LatticePropagator p2(&Grid);
3
4  LatticeColourMatrix colMat1[Ns*Ns](&Grid);
5  LatticeColourMatrix colMat2[Ns*Ns](&Grid);
6
7  LatticeComplex mda[Ns*Ns*Ns*Ns](&Grid);
8
9  // Preparation:
10 for(int s1=0; s1<Ns; s1++) {
11 for(int s2=0; s2<Ns; s2++) {
12   colMat1[s1*Ns+s2] = peekSpin(p1,s1,s2);
13   colMat2[s1*Ns+s2] = peekSpin(p2,s1,s2);
14 }}
15
16 // Computation:
17 for(int s1=0; s1<Ns; s1++) {
18 for(int s2=0; s2<Ns; s2++) {
19 for(int s3=0; s3<Ns; s3++) {
20 for(int s4=0; s4<Ns; s4++) {
21   tmp = colMat1[s1*Ns+s2] * colMat2[s3*Ns+s4];
22   mda[s1*Ns*Ns*Ns+s2*Ns*Ns+s3*Ns+s4] = trace(tmp);
23 }}}}
```

Listing 18: Excerpt from the GRID code for the MDA computation (first approach).

Algorithm 1 First approach for computing the MDA

procedure MDA1($P1$,$P2$) ▷ Input two propagators
  for α, β in $1...N_s$ **do** ▷ Arrange data in `LatticeColourMatrix` arrays
    $C1[\alpha, \beta] \leftarrow P1_{\alpha,\beta}$
    $C2[\alpha, \beta] \leftarrow P2_{\alpha,\beta}$
  end for
  for $\alpha, \beta, \gamma, \delta$ in $1...N_s$ **do** ▷ Actual computation
    $MDA[\alpha, \beta, \gamma, \delta] = tr^c\{C1[\alpha, \beta]C2[\gamma, \delta]\}$
  end for
  return MDA
end procedure

Algorithm 2 Improved approach for computing the MDA

procedure MDA2($P1$,$P2$) ▷ Input two propagators
  for A, B in $1...N_c$ **do** ▷ Arrange data in `LatticeComplex` arrays
    for α, β in $1...N_s$ **do**
      $a[A, B, \alpha, \beta] \leftarrow P1^{A,B}_{\alpha,\beta}$
      $b[A, B, \alpha, \beta] \leftarrow P2^{A,B}_{\alpha,\beta}$
    end for
  end for
  for $\alpha, \beta, \gamma, \delta$ in $1...N_s$ **do** ▷ Actual computation
    $MDA[\alpha, \beta, \gamma, \delta] = a[1, 1, \alpha, \beta]\ b[1, 1, \gamma, \delta] +$
      $a[2, 1, \alpha, \beta]\ b[1, 2, \gamma, \delta] +$
      $a[3, 1, \alpha, \beta]\ b[1, 3, \gamma, \delta] +$
      $a[1, 2, \alpha, \beta]\ b[2, 1, \gamma, \delta] +$
      $a[2, 2, \alpha, \beta]\ b[2, 2, \gamma, \delta] +$
      $a[3, 2, \alpha, \beta]\ b[2, 3, \gamma, \delta] +$
      $a[1, 3, \alpha, \beta]\ b[3, 1, \gamma, \delta] +$
      $a[2, 3, \alpha, \beta]\ b[3, 2, \gamma, \delta] +$
      $a[3, 3, \alpha, \beta]\ b[3, 3, \gamma, \delta]$
  end for
  return MDA
end procedure

The second approach for the MDA computation arranges the data in arrays of `LatticeComplex` objects. As can be seen in algorithm 2, the procedure is again divided in a first part where the complex values of a certain spin and colour index are taken from the propagators and a second part where those complex numbers are used to compute the MDA. This second step now looks a bit more difficult as in the first approach because the propagator multiplication and the trace are combined in a single step. The corresponding GRID code (see listing 19) also involves a "`pragma omp parallel for collapse(4)`" statement that indicates that the following four `for` loops are independent and can (in principle) be processed in parallel. This line is essential for an efficient implementation.

```
LatticePropagator p1(&Grid);
LatticePropagator p2(&Grid);

LatticeSpinMatrix sMat1(&Grid);
LatticeSpinMatrix sMat2(&Grid);

LatticeComplex a[Ns*Ns*Nc*Nc](&Grid);
LatticeComplex b[Ns*Ns*Nc*Nc](&Grid);

LatticeComplex mda[Ns*Ns*Ns*Ns](&Grid);

// Preparation:
for(int c1=0; c1<Nc; c1++) {
for(int c2=0; c2<Nc; c2++) {
  sMat1 = peekColour(p1,c1,c2);
  sMat2 = peekColour(p2,c1,c2);

  for(int s1=0; s1<Ns; s1++) {
  for(int s2=0; s2<Ns; s2++) {
    a[c1*Nc*Ns*Ns+c2*Ns*Ns+s1*Ns+s2] = peekSpin(sMat1,s1,s2);
    b[c1*Nc*Ns*Ns+c2*Ns*Ns+s1*Ns+s2] = peekSpin(sMat2,s1,s2);
  }}
}}

// Computation:
#pragma omp parallel for collapse(4)
for(int s1=0; s1<Ns; s1++) {
for(int s2=0; s2<Ns; s2++) {
for(int s3=0; s3<Ns; s3++) {
for(int s4=0; s4<Ns; s4++) {

  mda[s1*Ns*Ns*Ns+s2*Ns*Ns+s3*Ns+s4] =
    a[0*Ns*Ns+s1*Ns+s2] * b[0*Ns*Ns+s3*Ns+s4] +
    a[3*Ns*Ns+s1*Ns+s2] * b[1*Ns*Ns+s3*Ns+s4] +
    a[6*Ns*Ns+s1*Ns+s2] * b[2*Ns*Ns+s3*Ns+s4] +
    a[1*Ns*Ns+s1*Ns+s2] * b[3*Ns*Ns+s3*Ns+s4] +
    a[4*Ns*Ns+s1*Ns+s2] * b[4*Ns*Ns+s3*Ns+s4] +
    a[7*Ns*Ns+s1*Ns+s2] * b[5*Ns*Ns+s3*Ns+s4] +
    a[2*Ns*Ns+s1*Ns+s2] * b[6*Ns*Ns+s3*Ns+s4] +
    a[5*Ns*Ns+s1*Ns+s2] * b[7*Ns*Ns+s3*Ns+s4] +
    a[8*Ns*Ns+s1*Ns+s2] * b[8*Ns*Ns+s3*Ns+s4];
}}}}
```

Listing 19: Excerpt from the GRID code for the MDA computation (second approach).

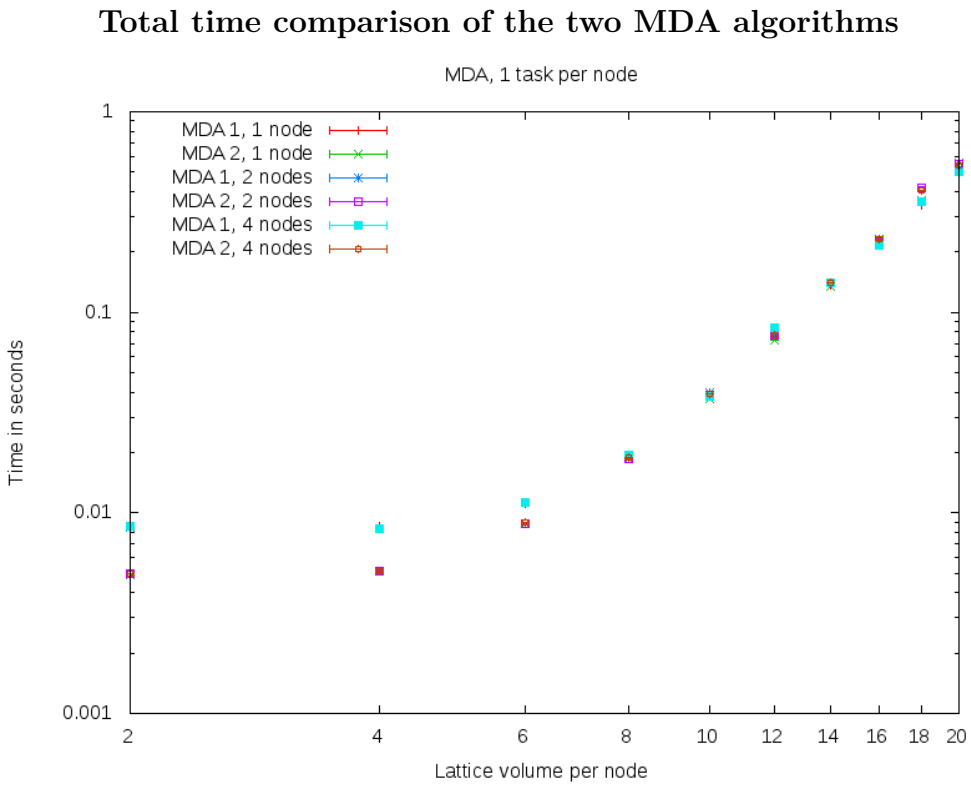

Figure 26: The plot shows the total time (logarithmic scale) used for several lattice sizes for both MDA algorithms. The lattice volume N^4 is simply denoted as N (for $N = 2, 4, 6, ...$).

Both discussed approaches for the computation of the MDA are compared in figure 26. This plot depicts the total time for one call of the MDA functions, either with algorithm 1 (MDA 1) or algorithm 2 (MDA 2) averaged over one thousand calls and its error. The benchmarks are again done on one, two and four nodes each with the GRID ICC Intel build. Although algorithm 2 is faster for small lattices the difference between the two approaches almost vanishes for lattice volumes (per node) greater than 8^4.

But for a decision in favour of one of these algorithms, another attribute can be studied: the time spent arranging the data and actually computing the MDA. This comparison is done in figure 27 on one node and a 8^4 lattice. The bar plot shows the average time (with error) for each part of the algorithms (preparation and computation). As already depicted in figure 26 the total time is almost equivalent on this lattice, but the first algorithm takes much longer to compute the MDA from the prepared data. With regard to the following task of implementing the full MDA algorithm, MDA 2 is favoured. The idea is to split up the two parts of the MDA 2 algorithm, prepare the data once and compute many MDA's on this data in order to save time.

7.2 Full MDA implementation

The full MDA routine[11] is basically done in the same way as already implemented in LibHadronAnalysis. The task is to take several `LatticePropagators`, get their first and second derivative with respect to the four spacetime directions and compute the MDA contraction for each combination of those.

Algorithm 3 shows the procedure. At first, one `LatticePropagator` is taken from the set of all propagators and derived in direction μ. The result is stored in an additional `LatticePropagator` object $\Delta p1$ which can then be rearranged into a `LatticeComplex` array *data1*. This is used in the following loop (lines 6-9) to compute the contraction of $D_\mu p1$ and $p2$. The next part (lines 10-18) is similarly computing all combinations of the second derivatives $D_\nu D_\mu p1$ and the original propagators $p2$. Finally, the MDA (i.e. MDA 2) contraction is calculated for all pairs of $D_\mu p1$ and $D_\nu p2$ (lines 19-30).

Detailed comparison of the two MDA algorithms

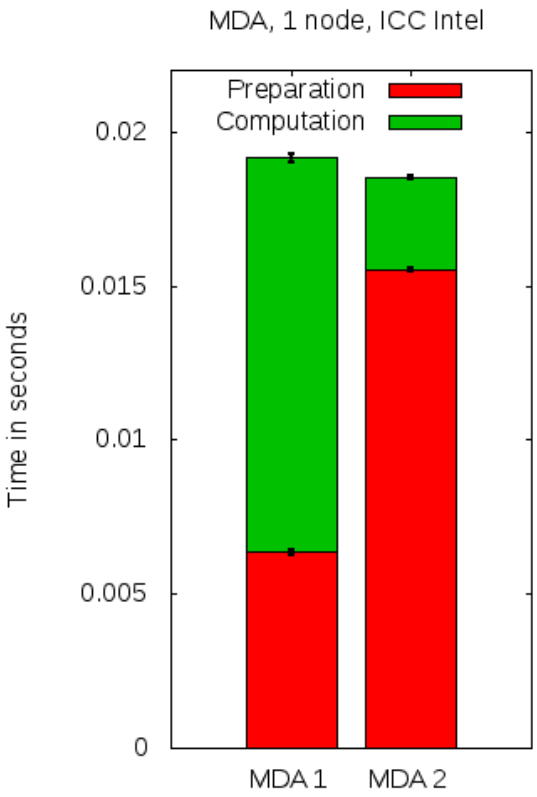

Figure 27: The bars show the average time (with error) spent arranging the data (preparation) and actually computing the MDA (computation).

For the comparison, the GRID and the LibHadronAnalysis implementation - both compiled with the ICC - are run with socket and NUMA configurations, with different lattice sizes and on multiple nodes. The task of computing the full MDA is split up into two parts, the total times for the computation of all derivatives and the total time of all MDA contractions. In LibHadronAnalysis the derivative is calculated inside two functions: `setPropagator`, where the data is prepared for a faster computation and `getDerivative` for the calculation of the derivative itself. The GRID benchmark simply uses the implementation from listing 20.

[11] Test code fullMDA.cpp is located in the appendix.

Algorithm 3 Efficient computation of the MDA

procedure FULLMDA(*props*) ▷ Input array of propagators
  for $p1$ in *props* **do**
    for μ in $1...N_d$ **do**
      $\Delta p1 \leftarrow D_\mu p1$ ▷ Calculate and store derivative $D_\mu p1$
      $data1 \Leftarrow \Delta p1$ ▷ Rearrange the data
      for $p2$ in *props* **do**
        $data2 \Leftarrow p2$ ▷ Rearrange the data
        MDA($data1, data2$) ▷ Compute MDA of $D_\mu p1$ and $p2$
      end for
      for ν in $1...N_d$ with $\nu \neq \mu$ **do**
        $\Delta p2 \leftarrow D_\nu \Delta p1$ ▷ Calculate and store derivative $D_\nu D_\mu p1$
        $data1 \Leftarrow \Delta p2$ ▷ Rearrange the data
        for $p2$ in *props* **do**
          $data2 \Leftarrow p2$ ▷ Rearrange the data
          MDA($data1, data2$) ▷ Compute MDA of $D_\nu D_\mu p1$ and $p2$
        end for
      end for
    end for
    for $p2$ in *props* **do**
      for μ in $1...N_d$ **do**
        $\Delta p1 \leftarrow D_\mu p1$ ▷ Calculate and store derivative $D_\mu p1$
        $data1 \Leftarrow \Delta p1$ ▷ Rearrange the data
        for ν in $1...N_d$ with $\nu > \mu$ **do**
          $\Delta p2 \leftarrow D_\nu p2$ ▷ Calculate and store derivative $D_\nu p2$
          $data2 \Leftarrow \Delta p2$ ▷ Rearrange the data
          MDA($data1, data2$) ▷ Compute MDA of $D_\mu p1$ and $D_\nu p2$
        end for
      end for
    end for
  end for
end procedure

```
inline void derivative(const LatticePropagator& prop, const ↩
   LatticeGaugeField& u, int dir, int len, LatticePropagator& ret)
{
  LatticeColourMatrix u_mu = PeekIndex<LorentzIndex>(u,dir);
  LatticePropagator tmp = adj(u_mu)*prop;
  ret = 0.5 * ( u_mu*Cshift(prop,dir,len) - Cshift(tmp,dir,-len) );
}
```

Listing 20: Implementation of the derivative in direction `dir` with length `len`.

The benchmarks for a 8^4 lattice on one node are given in figure 28. The bars show the total time spent in the derivative and MDA functions (and both together) for the given configurations. Additionally, the data is presented for the socket and the NUMA configuration.
As already seen in previous benchmarks, the NUMA setup increases the performance (especially GRID) by a remarkable factor. While LibHadronAnalysis shows better results on one task, the difference between the implementations is completely gone for four tasks. But this is only the case for such small lattices. Increasing the lattice volume to 16^4 and computing on four nodes as done in figure 29 shows the strength of LibHadronAnalysis, for which the calculations can be finished in about half of the time compared to GRID. Still it has to be considered that the GRID benchmarks depend on the internal structure of the GRID classes while the implementations in LibHadronAnalysis are detached from the Chroma classes to fit this special task.

Total time comparison of the full MDA implementations on one node with a 8^4 lattice

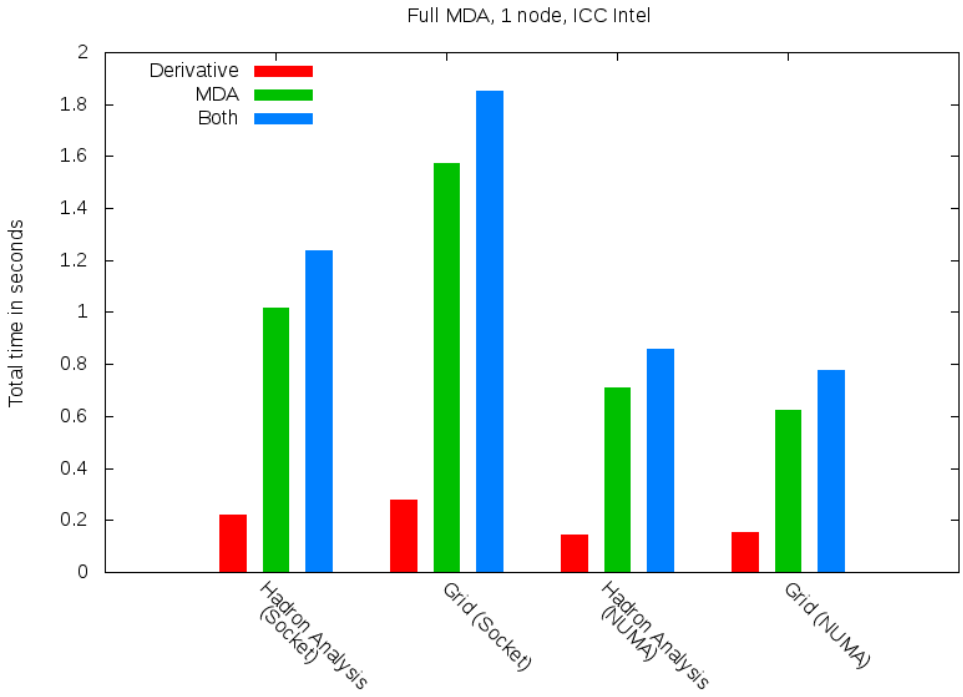

Figure 28: Comparison of the total time spent in each part of the full MDA calculation on a 8^4 lattice. The plot shows the results of the LibHadronAnalysis and GRID library (both compiled with the ICC) on one node with one and four tasks.

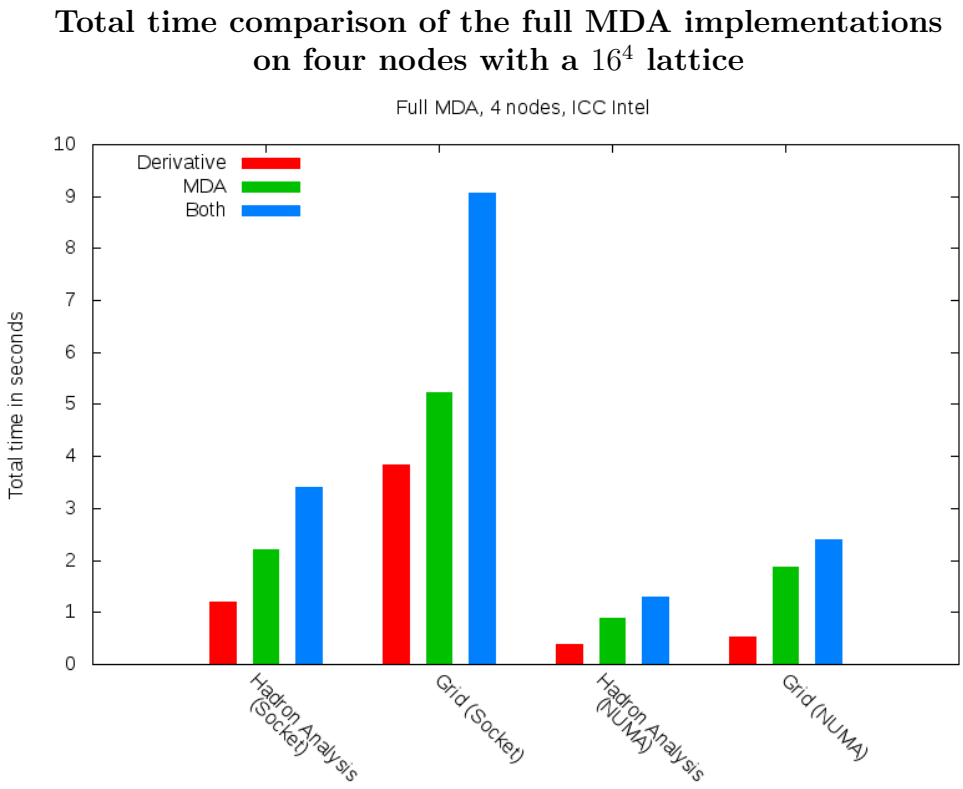

Figure 29: Comparison of the total time spent in each part of the full MDA calculation on a 16^4 lattice. The plot shows the results of the LibHadronAnalysis and GRID library (both compiled with the ICC) on four nodes with one and four tasks per node.

8 Summary

In this thesis the basics of QCD on the lattice and the architecture of modern hardware (QPACE 3) and software (GRID) were discussed in order to be able to compare their performance to currently used libraries (Chroma, QDP++ & LibHadronAnalysis). The benchmarks were chosen to focus on the main issues of computing the Meson Distribution Amplitude (MDA), such as propagator multiplication and derivatives.
The comparison of GRID and Chroma showed a promising performance for the new library. For the benchmarks on correlation functions GRID executed the same calculations in about half of the time compared to Chroma when the ICC compiler and Intel MPI were used. It also benefited from the NUMA mode which implies the application of four tasks per node. For the derivative benchmarks GRID could only keep up with Chroma by means of this NUMA setup.
Those results were then used to design two different approaches for the calculation of the MDA with GRID's classes and functions. As both ways resulted in the same computation time, the one yielding the smaller time spent in the actual computation sub-routine was chosen. The results were afterwards used for the design of the full MDA algorithm. Although those implementations perform well under the right conditions (compilation, execution, etc.), they could not keep up with the optimized implementations of the LibHadronAnalysis most of the time.
Yet GRID represents an alternative for the outdated Chroma & QDP++ code. The results of this thesis demonstrate that the right choice of compiler (ICC) and MPI (Intel MPI) as well as the application of hardware features such as the NUMA clustering mode on the Knights Landing processor lead to a significant advantage of GRID over Chroma for most of the calculations tested.

Bibliography

[1] Kenneth G. Wilson. Confinement of quarks. *Phys. Rev. D*, 10:2445–2459, 1974.

[2] Silas R Beane, William Detmold, Kostas Orginos, and Martin J Savage. Uncertainty quantification in lattice qcd calculations for nuclear physics. *Journal of Physics G: Nuclear and Particle Physics*, 42(3):034022, 2015.

[3] Thomas DeGrand and Carleton DeTar. *Lattice Methods for Quantum Chromodynamics*. World Scientific Publishing Co. Pte. Ltd., 2006.

[4] Claude Roiesnel. On the definition of the covariant lattice dirac operator, 2012. Available online `https://arxiv.org/pdf/1211.1944.pdf` as arXiv:1211.1944 [hep-lat] (visited on 22.6.2017).

[5] Kenneth G. Wilson. *Quarks and Strings on a Lattice*. Springer US, Boston, MA, 1977.

[6] A. V. Radyushkin. Shape of Pion Distribution Amplitude. *Phys. Rev.*, D80:094009, 2009.

[7] G. S. Bali, V. M. Braun, M. Göckeler, M. Gruber, F. Hutzler, P. Korcyl, B. Lang, and A. Schäfer. Second moment of the pion distribution amplitude with the momentum smearing technique. 2017.

[8] Wu Feng and Tom Scogland. Green500 list, 2017. (visited on 21.8.2017).

[9] Jim Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming*. Elsevier Inc., 2016.

[10] Robert G. Edwards and Balint Joo. The Chroma software system for lattice QCD. *Nucl. Phys. Proc. Suppl.*, 140:832, 2005.

[11] Peter Boyle, Guido Cossu, Antonin Portelli, and Azusa Yamaguchi. Grid: A next generation data parallel c++ qcd library. Available online `http://www2.epcc.ed.ac.uk/~adrianj/Tsukuba2016/day1/Cossu-Grid.pdf` (visited on 21.6.2017) and `https://indico.ph.ed.ac.uk/event/20/session/2/contribution/7/material/slides/0.pdf` (visited on 21.8.2017).

[12] Peter Boyle, Azusa Yamaguchi, and Antonin Portelli. Grid: A next generation data parallel c++ qcd library, 2015. Available online `https://arxiv.org/pdf/1512.03487.pdf` as arXiv:1512.03487 [hep-lat] (visited on 21.6.2017).

Appendix

Test code for the full MDA implementation with GRID

```
#include <Grid/Grid.h>
#include <iostream>
#include <fstream>

#define MYNAMESPACE TIMING // or MDA

using namespace std;
using namespace Grid;
using namespace Grid::QCD;

int nLoops;
std::vector<int> latt_size(4);
std::vector<int> mpi_layout(4);
int nThreads;
std::string outFileName;
int nProps;


void error(double* array, int len, double& average, double& error)
{
  //...
}

bool processCmdLineArgs(int argc,char** argv)
{
  //...
}

namespace MDA {

  inline void derivative(const LatticePropagator& prop,
                         const LatticeGaugeField& u, int dir,
                         int len, LatticePropagator& ret)
  {
    LatticeColourMatrix u_mu = PeekIndex<LorentzIndex>(u,dir);
    LatticePropagator tmp = adj(u_mu)*prop;
    ret = 0.5 * ( u_mu*Cshift(prop,dir,len) - Cshift(tmp,dir,-len) );
  }

  inline void arrangeData(const LatticePropagator& prop,
                          LatticeComplex* data, bool gamma)
  {
    if(gamma) {
```

```
44        Gamma gamma5(Gamma::Algebra::Gamma5);
45        arrangeData(gamma5*adj(prop)*gamma5,data,false);
46      } else {
47        LatticeSpinMatrix sMat(prop._grid);
48
49        for(int c1=0; c1<Nc; c1++) {
50        for(int c2=0; c2<Nc; c2++) {
51          sMat = peekColour(prop,c1,c2);
52
53          for(int s1=0; s1<Ns; s1++) {
54          for(int s2=0; s2<Ns; s2++) {
55            data[c1*Nc*Ns*Ns+c2*Ns*Ns+s1*Ns+s2] = peekSpin(sMat,s1,s2);
56          }}
57        }}
58      }
59    }
60
61    inline void mda(LatticeComplex* a, LatticeComplex* b,
62                    LatticeComplex* ret)
63    {
64    #pragma omp parallel for collapse(4)
65      for(int s1=0; s1<Ns; s1++) {
66      for(int s2=0; s2<Ns; s2++) {
67      for(int s3=0; s3<Ns; s3++) {
68      for(int s4=0; s4<Ns; s4++) {
69
70        ret[s1*Ns*Ns*Ns+s2*Ns*Ns+s3*Ns+s4] =
71            a[0*Ns*Ns+s1*Ns+s2] * b[0*Ns*Ns+s3*Ns+s4] +
72            a[3*Ns*Ns+s1*Ns+s2] * b[1*Ns*Ns+s3*Ns+s4] +
73            a[6*Ns*Ns+s1*Ns+s2] * b[2*Ns*Ns+s3*Ns+s4] +
74            a[1*Ns*Ns+s1*Ns+s2] * b[3*Ns*Ns+s3*Ns+s4] +
75            a[4*Ns*Ns+s1*Ns+s2] * b[4*Ns*Ns+s3*Ns+s4] +
76            a[7*Ns*Ns+s1*Ns+s2] * b[5*Ns*Ns+s3*Ns+s4] +
77            a[2*Ns*Ns+s1*Ns+s2] * b[6*Ns*Ns+s3*Ns+s4] +
78            a[5*Ns*Ns+s1*Ns+s2] * b[7*Ns*Ns+s3*Ns+s4] +
79            a[8*Ns*Ns+s1*Ns+s2] * b[8*Ns*Ns+s3*Ns+s4];
80      }}}}
81    }
82
83  }
84
85  namespace TIMING {
86
87    inline void derivative(const LatticePropagator& prop,
88                           const LatticeGaugeField& u, int dir,
89                           int len, LatticePropagator& ret)
90    {
91      double start = usecond();
```

```
92
93      MDA::derivative(prop,u,dir,len,ret);
94
95      double stop = usecond();
96      std::cout << std::endl << "derivative time = "
97                << (stop - start)/1000000.0 << " secs" << std::endl;
98    }
99
100   inline void arrangeData(const LatticePropagator& prop,
101                           LatticeComplex* data, bool gamma)
102   {
103     double start = usecond();
104
105     MDA::arrangeData(prop,data,gamma);
106
107     double stop = usecond();
108     std::cout << std::endl << "arrangement time = "
109               << (stop - start)/1000000.0 << " secs" << std::endl;
110   }
111
112   inline void mda(LatticeComplex* a, LatticeComplex* b,
113                   LatticeComplex* ret)
114   {
115     double start = usecond();
116
117     MDA::mda(a,b,ret);
118
119     double stop = usecond();
120     std::cout << std::endl << "computation time = "
121               << (stop - start)/1000000.0 << " secs" << std::endl;
122   }
123
124 }
125
126
127 int main (int argc, char ** argv)
128 {
129   if(!processCmdLineArgs(argc,argv)) {
130     return 1;
131   }
132
133   Grid_init(&argc,&argv);
134
135   /*////////////////
136   // Initialization //
137   ////////////////*/
138
139   std::vector<int> simd_layout = GridDefaultSimd(Nd,vComplex::Nsimd());
```

```
140  GridCartesian     Grid(latt_size,simd_layout,mpi_layout);
141
142  GridParallelRNG rng(&Grid);
143  rng.SeedFixedIntegers(std::vector<int>({1,2,3,4}));
144
145  int derivativeLen = 1;
146
147  // Work-around for LatticePropagator props[nProps](&Grid);
148  void* raw_memory =
149          operator new[](nProps * sizeof(LatticePropagator(&Grid)));
150  LatticePropagator* props =
151          static_cast<LatticePropagator*>( raw_memory );
152  for(int i=0; i<nProps; i++) new( &props[i] )LatticePropagator(&Grid);
153
154  for(int i=0; i<nProps; i++) {
155    random(rng,props[i]);
156  }
157
158  LatticeGaugeField U(&Grid);
159  random(rng,U);
160
161  LatticePropagator dProp1(props[0]._grid);
162  LatticePropagator dProp2(props[0]._grid);
163
164
165  // Work-around for LatticeComplex data1[Nc*Nc*Ns*Ns](props[0]._grid);
166  void* raw_memory2 =
167          operator new[](Nc*Nc*Ns*Ns * sizeof(LatticeComplex(&Grid)));
168  LatticeComplex* data1 = static_cast<LatticeComplex*>( raw_memory2 );
169  for(int i=0; i<Nc*Nc*Ns*Ns; i++)
170      new( &data1[i] )LatticeComplex(&Grid);
171
172  // Work-around for LatticeComplex data2[Nc*Nc*Ns*Ns](props[0]._grid);
173  void* raw_memory3 =
174          operator new[](Nc*Nc*Ns*Ns * sizeof(LatticeComplex(&Grid)));
175  LatticeComplex* data2 = static_cast<LatticeComplex*>( raw_memory3 );
176  for(int i=0; i<Nc*Nc*Ns*Ns; i++)
177      new( &data2[i] )LatticeComplex(&Grid);
178
179  // Work-around for
180  // LatticeComplex resultBuffer[Ns*Ns*Ns*Ns](props[0]._grid);
181  void* raw_memory4 =
182          operator new[](Ns*Ns*Ns*Ns * sizeof(LatticeComplex(&Grid)));
183  LatticeComplex* resultBuffer =
184          static_cast<LatticeComplex*>( raw_memory4 );
185  for(int i=0; i<Ns*Ns*Ns*Ns; i++)
186      new( &resultBuffer[i] )LatticeComplex(&Grid);
187
```

```
188   // (Work-arounds only necessary for the ICC compiler)
189
190
191   /*///////////////
192   // Calculation //
193   // Measurement //
194   ///////////////*/
195
196   double timerTime[nLoops];
197   double timerStart, timerStop;
198
199   for(int i=0; i<nLoops; i++) {
200     timerStart = usecond();
201
202     for(int p1=0; p1<nProps; p1++) {
203
204       const LatticePropagator& prop1 = props[p1];
205
206       for(int mu=0; mu<Nd; mu++) {
207
208         MYNAMESPACE::derivative(prop1,U,mu,derivativeLen,dProp1);
209         MYNAMESPACE::arrangeData(dProp1,data1,false);
210
211         for(int p2=0; p2<nProps; p2++) {
212
213           const LatticePropagator& prop2 = props[p2];
214
215           MYNAMESPACE::arrangeData(prop2,data2,true);
216           MYNAMESPACE::mda(data1,data2,resultBuffer);
217         }
218
219         for(int nu=0; nu<Nd; nu++) {
220
221           if(mu == nu) continue;
222
223           MYNAMESPACE::derivative(dProp1,U,nu,derivativeLen,dProp2);
224           MYNAMESPACE::arrangeData(dProp2,data1,false);
225
226           for(int p2=0; p2<nProps; p2++) {
227
228             const LatticePropagator& prop2 = props[p2];
229
230             MYNAMESPACE::arrangeData(prop2,data2,true);
231             MYNAMESPACE::mda(data1,data2,resultBuffer);
232           }
233         }
234       }
235
```

```
236       for(int p2=0; p2<nProps; p2++) {
237
238         const LatticePropagator& prop2 = props[p2];
239
240         for(int mu=0; mu<Nd; mu++) {
241
242           MYNAMESPACE::derivative(prop1,U,mu,derivativeLen,dProp1);
243           MYNAMESPACE::arrangeData(dProp1,data1,false);
244
245           for(int nu=0; nu<Nd; nu++) {
246
247             if(mu >= nu) continue;
248
249             MYNAMESPACE::derivative(prop2,U,nu,derivativeLen,dProp2);
250             MYNAMESPACE::arrangeData(dProp2,data2,true);
251             MYNAMESPACE::mda(data1,data2,resultBuffer);
252           }
253         }
254       }
255     }
256
257     timerStop = usecond();
258     timerTime[i] = timerStop - timerStart;
259   }
260
261   /*///////////////
262   // Evaluation //
263   ///////////////*/
264
265   double time, timeError;
266   error(timerTime,nLoops,time,timeError);
267
268   time /= 1000000.0;
269   timeError /= 1000000.0;
270
271   /*/////////////////
272   // Print results //
273   /////////////////*/
274
275   if(Grid.IsBoss()) {
276     ofstream file;
277     file.open(outFileName,ios::app);
278     if(file.is_open()) {
279       int vol = latt_size[0]*latt_size[1]*latt_size[2]*latt_size[3];
280       file << omp_get_max_threads() << "\t"
281            << latt_size[0] << latt_size[1]
282            << latt_size[2] << latt_size[3]
283            << "\t" << vol << "\t"
```

```
284             << time << "\t" << timeError << std::endl;
285       file.close();
286     } else {
287       std::cerr << "Unable to open file!" << std::endl;
288     }
289   }
290
291   /*///////////////
292   // Destructors //
293   ///////////////*/
294
295   for(int i=nProps-1; i>=0; i--) {
296     props[i].~LatticePropagator();
297   }
298   for(int i=Ns*Ns*Nc*Nc-1; i>=0; i--) {
299     data1[i].~LatticeComplex();
300     data2[i].~LatticeComplex();
301   }
302   for(int i=Ns*Ns*Ns*Ns-1; i>=0; i--) {
303     resultBuffer[i].~LatticeComplex();
304   }
305   operator delete[]( raw_memory );
306   operator delete[]( raw_memory2 );
307   operator delete[]( raw_memory3 );
308   operator delete[]( raw_memory4 );
309
310   Grid_finalize();
311 }
```

Listing 21: fullMDA.cpp

Affirmation

Eidesstattliche Erklärung

Ich habe die Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und bisher keiner anderen Prüfungsbehörde vorgelegt. Außerdem bestätige ich hiermit, dass die vorgelegten Druckexemplare und die vorgelegte elektronische Version der Arbeit identisch sind, dass ich über wissenschaftlich korrektes Arbeiten und Zitieren aufgeklärt wurde und dass ich von den in § 26/27 Abs. 5 vorgesehenen Rechtsfolgen Kenntnis habe.

$Unterschrift$: $Ort, Datum$: