Lattice QCD Optimization and Polytopic Representations of Distributed Memory

Michael Kruse

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

Inria INVENTEURS DU MONDE NUMÉRIQUE

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

Université Paris-Sud

Ecole Doctorale d'Informatique de Paris-Sud
INRIA Saclay Île-de-France/Laboratoire de Recherche en Informatique

Discipline : Informatique

Thèse de doctorat

Soutenue le 26 Septembre 2014 par

Michael Kruse

Lattice QCD Optimization and Polytopic Representations of Distributed Memory

Directeur de thèse :	Mme. Christine Eisenbeis	Directrice de Recherche (INRIA, LRI, Université Paris-Sud XI)
Composition du jury :		
Rapporteurs :	M. Philippe Clauss	Professeur (Université de Strasbourg)
	M. Jagannathan Ramanujam	Professeur (Louisiana State University)
Examinateurs :	M. Sid Touati	Professeur (Université Nice Sophia Antipolis)
	M. Paul Feautrier	Professeur émérite (ENS Lyon)
	M. Sylvain Conchon	Professeur (Université Paris-Sud XI)
	M. Olivier Pène	Directeur de Recherche retraité (CNRS, LPT Orsay)
Invités :	M. Gilbert Grosdidier	Directeur de Recherche retraité (CNRS, LAL Orsay)

Abstract

Motivated by modern day physics which in addition to experiments also tries to verify and deduce laws of nature by simulating the state-of-the-art physical models using large computers, this thesis explores means of accelerating such simulations by improving the simulation programs they run. The primary focus is Lattice Quantum Chromodynamics (QCD), a branch of quantum field theory, running on IBM newest supercomputer, the Blue Gene/Q.

In a first approach, the source code of tmLQCD, a Lattice QCD program, is improved to run faster on the Blue Gene machine. Its most performance-relevant operation is a 8-point stencil in 4 dimensional space. Two different optimization strategies are pursued: One with the prioritizing spatial and temporal locality, and a second making use of the hardware's data stream prefetcher. On Blue Gene/Q the first strategy reaches up to 20% of the peak theoretical floating point operation performance of that machine. The second strategy with up to 54% of peak is much faster at the cost of using 4 times more memory by storing the data in the order they will be used in the next stencil operation, duplicating data where necessary.

Other techniques exploited are direct programming of the messaging hardware (called MUSPI by IBM), a low-overhead work distribution mechanism for threads, explicit data prefetching of data (using dcbt instructions) and manual vectorization (using QPX; width-4 SIMD instructions). Hardware-based list prefetching and transactional memory – both distinct and novel features of the Blue Gene/Q system – did not improve the program's performance.

The second approach is the newly-written LLVM compiler extension called *Molly* which optimizes the program itself, specifically the distribution of data and work between the nodes of a cluster machine such as Blue Gene/Q. Molly represents arrays using integer polyhedra and uses another already existing compiler extension Polly which represents statements and loops using polyhedra. When Molly knows how data is distributed among the nodes and where statements are executed, it adds code that manages the data flow between the nodes. Molly can also permute the order of data in memory.

Molly's main task is to cluster data into sets that are sent to the same target into the same buffer because single transfers involve a massive overhead. We present an algorithm that minimizes the number of transfers for unparametrized loops using anti-chains of data flows. In addition, we implement a heuristic that takes into account how the programmer wrote the code. Asynchronous communication primitives are inserted right after the data is available respectively just before it is used. A runtime library implements these primitives using MPI.

Molly manages to distribute any code that is representable by the polyhedral model, but does so best for stencils codes such as Lattice QCD. Compiled using Molly, the Lattice QCD stencil reaches 2.5% of the theoretical peak performance. The performance gap is mostly because all the other optimizations are missing, such as vectorization. Future versions of Molly may also effectively handle non-stencil codes and use make use of all the optimizations that make the manually optimized Lattice QCD stencil so fast.

Résumé

La physique actuelle cherche, à côté des expériences, à vérifier et déduire les lois de la nature en simulant les modèles physiques sur d'énormes ordinateurs. Cette thèse explore comment accélérer ces simulations en améliorant les programmes qui les font tourner. L'application de référence est la chromodynamique quantique sur réseaux (LQCD pour "Lattice Quantum Chromodynamics"), une branche de la théorie quantique des champs, tournant sur le plus récent des supercalculateurs d'IBM, le Blue Gene/Q.

Dans un premier temps, on améliore le code source de tmLQCD, un programme de LQCD, dont l'opération clef pour la performance est un stencil à 8 points en dimension 4. On étudie deux stratégies d'optimisation différentes : la première se donne comme priorité d'améliorer la localité spatiale et temporelle ; la seconde utilise le préchargement matériel de flux de données. Sur le Blue Gene/Q, la première stratégie permet d'atteindre 20% de la performance crête théorique. La seconde, avec jusqu'à 54% de la performance crête est bien meilleure mais utilise 4 fois plus de mémoire car elle stocke les résultats dans l'ordre où les utilise le stencil suivant, ce qui requiert de dupliquer des données.

Les autres techniques exploitées sont la programmation directe du système de communication (appelé MUSPI chez IBM), un mécanisme allégé de gestion des threads, le préchargement explicite de certaines données (à l'aide de l'instruction dcbt) et la vectorisation manuelle (en utilisant les instructions SIMD de largeur 4 ; appelé QPX par IBM). Le préchargement de liste et la mémoire transactionnelle - deux nouveaux mécanismes du Blue Gene/Q - n'améliorent pas les performances.

Dans un second temps, on présente la réalisation d'une extension appelé *Molly* au compilateur LLVM, pour optimiser automatiquement le programme, et plus précisément la distribution des données et des calculs entre les nœuds d'un cluster tel que le Blue Gene/Q. Molly représente les tableaux par des polyèdres entiers et utilise l'extension existante Polly qui représente les boucles et les instructions par des polyèdres. Partant de la spécification de la distribution des données et de l'emplacement des calculs, Molly ajoute le code qui gère les flots de données entre les nœuds de calcul. Molly peut aussi permuter l'ordre des données en mémoire.

La tâche principale de Molly est d'agréger les données dans des ensembles qui sont envoyés dans le même tampon au même destinataire, pour éviter l'overhead des transferts trop petits. Nous présentons un algorithme qui minimise le nombre de transferts pour des boucles non-paramétrées, basé sur les antichaînes du flot des données. De plus, nous implémentons une heuristique qui tient compte de la manière dont le programmeur a écrit son code. Les primitives de communication asynchrone sont insérées juste après que les données soient disponibles – respectivement juste avant qu'elles soient utilisées. Une bibliothèque runtime implémente ces primitives en utilisant MPI.

Molly gère la distribution pour tout code représentable dans le modèle polyédrique, mais fonctionne mieux pour du code à stencil tel LQCD. Compilé avec Molly, le code LQCD atteint 2,5% de la performance crête. L'écart de performance est surtout dû au fait que les autres optimisations ne sont pas faites, par exemple la vectorisation. Les versions futures de Molly pourraient aussi gérer efficacement les codes non à stencil et exploiter les autres optimisations qui ont rendu le code LQCD optimisé à la main si rapide.

Zusammenfassung

Zusätzlich zur Experimentalphysik versucht die moderne Naturwissenschaft auch Naturgesetze durch Computersimulationen auf Großrechnern herzuleiten und zu verifizieren. Hierdurch motiviert, untersucht diese Arbeit Möglichkeiten zur weiteren Beschleunigung durch das Verbessern der darauf laufenden Programme. Der Schwerpunkt liegt auf das Optimieren von Anwendungen für Gitter-Quantenchromodynamik (Lattice QCD) für IBMs neuestem Großrechner, Blue Gene/Q.

Der erste Ansatz besteht aus dem Anpassen des Quelltextes von tmLQCD, einem Programm für Lattice QCD-Simulationen, an den Blue Gene Rechner. Seine für die Geschwindigkeit wichtigste Operation ist ein 8-Punkt-Stencil in 4 Dimensionen. Dabei werden zwei verschiedene Optimierungsstrategien verfolgt : Eine zur Verbesserung der zeitlichen und örtlichen Nähe und eine Zweite um der Hardware das Vorausladen der Datenströme zu ermöglichen. Auf dem Blue Gene/Q erreicht die erste Strategie bis zu 20% der theoretisch möglichen Geschwindigkeit für Gleitkommaoperationen. Die zweite Strategie ist mit bis zu 54% derselben deutlich erfolgreicher, mit dem Nachteil eines bis zu 4x höherem Speicherverbrauchs durch eine Umsortierung der Daten in Lesereihenfolge inklusive Duplizierung mehrfach gelesener Werte.

Weitere genutzte Techniken beinhalten sind der direkte Zugriff auf die Kommunikationseinheit des Blue Gene/Q (von IBM auf den Namen MUSPI getauft), ein schneller Mechanismus für die Thread-Arbeitsverteilung, explizites Vorausladen von in kürze benötigter Daten und manueller Vektorisierung mittels 256-Bit SIMD-Instruktionen (QPX genannt). Transaktionaler Speicher sowie Hardware List Prefetching – beides neue und einzigartige Funktionen des Blue Gene/Q-Prozessors – führten leider nicht zu einer Geschwindigkeitsverbesserung.

Der zweite Ansatz ist eine Erweitung für den LLVM-Compiler mit dem Namen Molly welche Programme selbststätig parallelisiert, speziell für die Systeme mit verteiltem Speicher wie dem Blue Gene/Q. Molly stellt Arrays mit Hilfe von Polyedern dar und benutzt die bereits existierende Erweitung Polly welche Anweisungen und Schleifen als Polyeder darstellt. Bei gegebener Datenverteilung zwischen den Knoten eines Clusters kann Molly den zur Datenfluss notwendigen Anweisungen generieren. Molly kann auch Daten im Speicher eines Einzelknotens umordnen.

Mollys Hauptaufgabe ist das Zusammenstellen von Daten zum selben Ziel zu einer Nachricht weil einzelne Übertragungen zu lange benötigen würden. Wir stellen einen Algorithmus mittles Antiketten vor welcher die Anzahl der Übertragungen minimiert wenn die Schleifen unparametrisiert sind. Eine Heuristik, die in Molly implementiert ist, orientiert sich stattdessen an der Quelltextstruktur. Anweisungen für asynchrone Datenkommunikation werden an der Stelle eingefügt, an dem die zu übertragenen Daten verfügbar sind, sowie kurz bevor sie auf dem Ziel benötigt werden. Eine Laufzeitbibliothek übersetzt diese Anweisungen in MPI-Aufrufe.

Molly kann prinzipiell jeden das Polytopmodell darstellbare Quelltexte parallelisieren, funktioniert aber am besten für Stencilcodes wie Lattice QCD. Mittels Molly kompiliert erreicht es 2.5% der theoretisch möglichen Gleitkommageschwindigkeit. Der Unterschied zur manuell optimierten Version kommt hauptsächlich durch das Fehlen anderer Optimierungen wie der Vektorisierung zustande. Zukünftige Versionen von Molly könnten auch andere Codes effektiv parallelisieren und ebenfalls die noch fehlenden Optimierungen anwenden wegen derer die erste Strategie so erfolgreich war.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

In 1974 the physicist Kenneth Geddes Wilson published a theory on how to approximate one of the four fundamental forces in continuous space-time using a finite number of representative measurement points [1]. The points are arranged in an equidistant mesh, also called a *lattice*. The force he described this way was the strong force that pulls quarks to each other. The theory about this strong force is also known as *Quantum Chromodynamics* (QCD), hence the discretized theory is known as *Lattice Quantum Chromodynamics* (Lattice QCD, or just LQCD).

Field theories are different from other physics theories in that these are not about explicit objects that interact and move in space or space-time, like Newtonian gravity. Instead, space-time is understood as having a value at each coordinate in space and time. The value most often is a scalar (a single number like temperature in Kelvin) or a vector (like gravity; the direction of the vector is the direction an object with mass is pulled to and its length is the "strength" of the pull), but Lattice QCD uses mathematical objects called *spinors* instead. As space and time are continuous, there are infinitely such coordinates and therefore cannot be numerically computed, but the discretized approximation introduced by Wilson can.

The computers in 1972 were not yet powerful enough to do such computations, but became stronger over time. With every generation of computers higher lattice density become more and more feasible and therefore it is hoped that the approximation becomes closer and closer to reality.

These simulations are used to derive observable physical constants. Those constants can also be measured experimentally, but confirming them using a second method derived solely from the physical model describing them increases the confidence in the model's correctness. In case the simulations do not match the observed reality, either the simulation has to be refined or current physics is incorrect. Being falsifiable is a quality of a theory. In this case a new theory deduced from the simulations can advance physics by giving new insights. Moreover, knowing that same physical constants can be derived from others (like electric permeability in vacuum ϵ_0 from speed of light and magnetic susceptibility) reduces arbitrariness from the theory and therefore simplifies it. Some results of such simulations are presented in [2].

Today lattice simulations run on some of the fastest computers pursuing better approximations by increasing the density of the mesh that results from the discretization of space-time. It increases the number of measurement points and therefore the computational effort. The principle is simple: The more lattice points are simulated the more accurate results can be expected. Simulations on relatively small lattice sizes can be done on every personal computer, we cannot expect new discoveries on this scale.

Computers on large scale are cluster computers – lots of individual compute nodes with a fast connection between them. This implies that every node has its own memory that is not directly accessible from the other nodes, hence such machines are also called *Distributed Memory Machines* (DMMs). A single global memory does not scale to the size of these machines.

Unfortunately, to achieve such computation speed, programming such computers in an efficient manner is more complicated. Just buying a faster computer is not enough, the programs running on it has to be adapted to the specific machine for maximal speed.

1.1 Motivation

The program suite *tmLQCD* [3] is a software one can use for such Lattice QCD simulations. It has been written by the ETMC, a high energy physics research collaboration. Like in nearly any software, some parts are executed more often than others; the most executed code is referred to as hotspots, critical parts or kernels. tmLQCD already includes an optimization for IBM's *Blue Gene/Q*.

The successor of the *Blue Gene/P* has been introduced in late 2012, named Blue Gene/Q. Naturally, one also wants to support this architecture as some computing centers replaced their Blue Gene/P machine with the new generation. To continue fast Lattice QCD simulations, the kernels have to be optimized to the new machine as well. The plain C version also works, but is comparatively slow as it does not use many of the hardware features that make the Blue Gene/Q fast.

Compared to Blue Gene/P, the new machine has some different characteristics that need to be taken into account, as well as new features that did not exist in the previous generation. Straightforward translation of an optimized kernel to the new architecture does not yield the best performance. For optimal performance even the way data is stored in memory needs to be changed. Unfortunately, a lot of code in tmLQCD assumes a specific way the data it stored. Changing it, required those parts to be rewritten, even if they do not belong to a hotspot.

Some optimizations are contradictory. One can optimize access patterns or reduce required bandwidth, but generally not both at the same time. One may also try to find a tradeoff between optimization goals, but which choice results in the fastest program is hard to predict. There is usually only one way to find out: Implement both optimizations and measure their execution time. Which version is faster may also depend on what the program's input. In the case of Lattice QCD the size of the lattice has a huge impact on what the bottleneck – the speed-limiting factor – is and therefore there is no single optimal, always-fastest way to optimize a program.

Finding the sweet-pot of optimizations for specific cases is a big task that involves trial-and-error, one that is tackled in this thesis. For each experiment, some code parts must be written and re-written again. Normally in software engineering, the different optimization aspects would be abstracted into different layers such that each of them can be modified independently. For speed optimizations this is not viable because every such abstraction will slow down the application. If lucky, the compiler may remove the abstraction in the compiled program, for instance by inlining a function. This works for relatively easy cases, but generally compilers keep abstractions out of fear to break some special case and/or to limit compile time. In extreme cases, one even has to write the assembler code manually because the compiler does not find the machine code one had in mind. Hence, the optimized hotspots have to be written to closely reflect the output machine code.

This means a lot of repetitive work when trying out a different optimization. Want to change the iteration order? Have to rewrite all the hotspot loops. This is unfortunate since there exists a mathematical model that can cope with many classes of such reorderings. Its research community calls it the *polyhedral model*. Some compilers make use of this model for loop transformations, but their implementations usually lack maturity, and to the programmer it is not visible what the compiler is doing or why not. Dedicated source-to-source transformation tools seem more successful, partly because the user has more control and on average better understanding of what the tool is supposed to do.

The math behind the polyhedral model can also be applied to the element ordering in

memory, something no mainstream compiler does yet. The combination of both may take the bulk from the programmer when doing such speed optimizations. Different versions of the same kernels become unnecessary, replaced by either the compiler knowing what the best reordering is for the platform, or by per-platform annotations chosen by the programmer. It decreases the maintenance cost of existing code since less code needs to be maintained.

Such an evolution already happened with other compiler optimizations. For instance, in the early days of C the only ways to avoid a function call overhead was to "declare" the function as a preprocessor macro. Today's compilers can inline function if annotated with the inline keyword or if the compiler thinks it is advantageous to do so. The general advice is to leave this decision to the compiler, avoiding premature optimization[1]. Actually, assuming a specific reordering is best without measurements *is* premature optimization, especially if those require a rewrite of the non-hotspot parts. The compiler being able to apply reorderings encourages trying out different program transformations.

1.2 Contributions of this Thesis

This thesis explores the two main ways of parallelization and general program speed optimizations[2] that do not involve changing the hardware: manual optimization of the source code and optimization by the compiler. Any optimization can be implemented in source – if necessary by going down to assembly level. But new transformations in the compiler must work for the general case, ensure that any well-defined source code does not change its meaning and as far as possible also never runs slower than without such changes. The reward is that the transformation is then applicable to any program automatically.

The use case is Lattice QCD on the Blue Gene/Q supercomputer. First, we change the source code of the tmLQCD program suite, searching for the best transformation for this architecture. Using this experience, we modify a compiler, enabling it to apply some of those optimizations automatically.

The following list describes the work done in the first part of the thesis.

- Adding Blue Gene/Q-optimized versions of the kernels to tmLQCD
- Experimenting with different memory layouts
- Optimization of assembly code generated by IBM's *IBM XL compiler* [5] (XLC) compiler, especially, *SIMD* instructions
- Reduction of multi-threading overhead
- Implement inter-node communication using low-level hardware commands (*Message Unit System Programming Interface* (MUSPI) instead of *Message Passing Interface* (MPI))

In the second part contributes the following items.

- Development of a model for polyhedral reordering of data in memory and between distributed memory nodes
- Implementing an extension called *Molly* to the *Clang* [6] compiler that does such transformations

[1] "There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%" (*Donald Knuth*) [4]

[2] Optimization in this thesis is used in the sense of improving a program's execution speed. It is not meant to find the fastest ("optimal") program possible.

- Evaluation of the transformations done by Molly

The Blue Gene/Q-optimized version of tmLQCD is not the fastest possible (“optimal”) implementation of Lattice QCD on Blue Gene/Q. Also, Molly has been prototyped with this single purpose in mind. It is not a ready-to-use implementation for any program, but could be extended that way.

1.3 Thesis Structure

Following the two different kinds of optimizations, this thesis consists of two parts. The first Part starting at Page 19 copes with the manual optimization of tmLQCD for the Blue Gene/Q computer. Part II at Page 91 and onward is about the implementation of Molly, the compiler extension for data layout transformations using the polyhedral model.

Part I starts with a brief overview of the theoretical physics behind Lattice QCD in Chapter 2. Chapter 3 explains the Blue Gene/Q platform and its hardware details that are important for program optimizations and Chapter 4 presents the various optimzations applied to tmLQCD. The experimental results using different combinations of optimization are show in Chapter 5. The part ends with a summary, more optimization ideas, and related work in Chapter 6.

The second Part begins with an introduction to the polyhedral model in Chapter 7. The following Chapter presents how the same theory can be applied to reordering data in local memory and between nodes of a distributed memory computer. Chapter 9 goes on presenting how it has been implemented as Molly, the extension to the Clang compiler. Experimental results of how well the Molly-optimized programs perform are shown in Chapter 10. The thesis finishes with a summary of this part, similar work by other researchers and ideas for improving Molly in Chapter 11.

Part I

Optimization of Lattice QCD for the Blue Gene/Q

2 Lattice Quantum Chromodynamics

The current physics standard model knows four fundamental forces that attract or repel particles: Gravity, electromagnetism, the weak force and the strong force. The latter three can be described using *Quantum Field Theories* (QFTs). For instance, the QFT of electromagnetism is called QCD. Finding a QFT of gravity would be a major breakthrough for today's physics, a big step towards a *unified field theory* which would unify all 4 forces into a single theory.

Field theories are a mathematical model in continuous space and time (Euclidean or Minkowski space). It assigns a value to the fields to each point in this space. Particles – themselves expressible as excitation in a field – are influenced by the value of the space-time coordinate they occupy , but the particles may influence the field as well.

In QFT, all the dynamic variables are fields. Considering for example *Quantum Electrodynamics* (QED), there are fields for electrons and positrons, which annihilate or create electrons/positrons from/to the vacuum. There are also photon fields which create or annihilate photons. QFT is more complex to solve. In the case of QED the small coupling of the electron to the photon allows for an analytical computational technique called "Feynman diagrams", but QCD which corresponds to strong interactions cannot be solved analytically anymore. Numerical simulation using a computer would be an option, but also impossible because space-time is a continuum. A computer simply cannot store values for infinitely many space-time coordinates. Space-time must be discretized first, with the necessary consequence that such a numerical solution is only an approximation of reality.

2.1 Quantum Chromodynamics

The force between color charged particles is mediated by exchange particles called *gluons* which exist in 8 different colors. Although hadrons are always white and therefore in principle do not attract or repel each other, they still influence each other when in direct proximity. This is why electric positively charged protons are kept together in a nucleus. It can be explained by residue forces – color charges not evenly distributed inside the hadrons. Merely, color charge is distributed in a quantum field. The spin of gluons is 1, which makes them bosons like the other exchange particles (photons, etc.).

QCD field theory has two main space-time fields. First, there is the quark field[1] describing the charge distribution. For every point in time and space it defines a Dirac 4-spinor of color vectors. Such an element consists of 4 SU(3)-vectors, i.e. an element of $\mathbb{C}^{4\times 3}$. One can define such a 4-spinor for multiple of the 6 quark flavors (up, down, strange, charm, top, bottom) but in the thesis we assume just one flavor. The three complex components of the SU(3) vector represent the amount of charge per color. There are four of them because of the four components of relativistic spinors (Dirac spinors). They are represented as linear combinations

[1] Alternative names: fermion field, spinor field, Grassmann field

of the 4 gamma matrices (or Dirac matrices) in the four-dimensional space of Dirac spinors. Spinor fields are typically denoted using the Greek letter ψ.

The second field is the gluon field[1]. It assigns a SU(3) (3×3 complex) matrix to each point in space-time and euclidean dimension, i.e. a subset of $\mathbb{C}^{4\times3\times3}$. It characterizes the exchange of color charges into the euclidean direction, by right-multiplying an SU(3) vector with the matrix. In the following, the capital letter A denotes a gluon field(s) for the 8 flavors. μ stands for one of the 4 space-time dimensions. By convention, $\mu = 0$ identifies the time dimension whereas 1, 2, 3 are the x, y and z space dimensions respectively.

The influence of all fields can be described using an *action functional* $\mathcal{S}$. Typically, it is the path-integral of a Lagrangian or Hamiltonian. Such an action maps a path through space and time to a scalar. In the case of classical field theory the principle of least action now states that the path with the smallest result is the one that is observed in reality, or at least is the one with highest probability. For instance, the Lagrangian of a particle of mass m, velocity v and potential energy $V(t, x)$ at time t and position x is

$$\mathcal{L} = \frac{1}{2}mv^2 - V(t, x)$$

and the action functional becomes

$$\mathcal{S}[\phi] = \int_\phi \mathcal{L}(\phi(t, x))\mathrm{d}t\mathrm{d}^3x = \int_\phi \left(\frac{1}{2}m(\partial_t\phi(t, x))^2 - V(\phi(t, x))\right)\mathrm{d}t\mathrm{d}^3x \ .$$

In classical field theory one chooses the path, the trajectory $\phi(t, x)$, which minimizes $\mathcal{S}[\phi]$. In QFT, all trajectories are taken into account and interfere with each other, and one defines a partition function Z as

$$Z = \int \mathcal{D}\phi e^{\frac{i}{\hbar}\mathcal{S}[\phi]}$$

where ϕ represents generically all the fields and $\int \mathcal{D}\phi$ is a continuous sum over all possible values of the fields on all possible paths in space-time. An observable quantity is defined via a given functional of the fields which, say $\mathcal{O}(\phi)$. The expected value of the observable $\mathcal{O}$ is given by

$$\langle\mathcal{O}(\phi)\rangle = \int \mathcal{D}\phi\mathcal{O}(\phi)e^{\frac{i}{\hbar}\mathcal{S}[\phi]}/Z \ .$$

The observable corresponds to the probability of a path being observed in the quantum-mechanical sense. That is, a measurement and therefore a wave function collapse have to happen in order to get a probability density of events.

The Lagrangian of continuum QCD is

$$\mathcal{L} = \underbrace{\sum_q \bar{\psi}_q[\gamma_\mu(\partial_\mu - igA_\mu) + m_q]\psi_q}_{\text{fermionic action}} + \underbrace{\frac{1}{4}F^2_{\mu\nu}}_{\text{gauge action}} \ . \tag{2.1}$$

Here μ and ν are one of the 4 space-time dimensions, q one of the 6 quark flavors (up, down, strange, charm, bottom, top), g is the strong force coupling constant, m_q is the quark's mass, A is the gluon field, ϕ_q the flavor's quark field and $\bar{\psi}_q$ ($= \psi^\dagger\gamma_0$ for relativistic quantum mechanics [7]). We will just assume a single flavor in the following. The field strength tensor is

$$F_{\mu\nu}(x) = \partial_\mu A_\nu(x) - \partial_\nu A_\mu(x) + ig[A_\mu(x), A_\nu(x)] \ .$$

[1] Alternative names: gauge field, color space

The fermionic action describes the interaction between gluons and quarks (fermions) and the gauge action is the self-interaction of gluons since these also carry a color charge. The QCD Lagrangian is a function of quark- and gluon-field.

The fermionic part in (2.1) can be simplified with the introduction of the *Dirac operator* defined as

$$D = (\partial_\mu - igA_\mu) - m \ , \tag{2.2}$$

also called the covariant derivative, and using the Feynman-slash notation

$$\not{D} = \gamma_\mu D = \gamma_\mu(\partial_\mu - igA_\mu) - m \tag{2.3}$$

(written out: *Dslash* operator) such that the Lagrangian simplifies to

$$\mathcal{L} = \bar{\psi}\not{D}\psi + \frac{1}{4}F_{\mu\nu}^2 \ .$$

The gamma matrices γ_μ are constant and must obey the euclidean Clifford algebra

$$\gamma_\mu\gamma_\nu = \begin{cases} -\gamma_\nu\gamma_\mu & \texttt{if } \mu \neq \nu \\ I & \texttt{if } \mu = \nu \end{cases}$$

with I being the identity matrix. The corresponding γ matrices in the representation used by tmLQCD is

$$\gamma_0 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \gamma_1 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$
$$\gamma_2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad \gamma_3 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} \tag{2.4}$$

There is also a fifth gamma matrix, historically named $\gamma_5 = i\gamma_0\gamma_1\gamma_2\gamma_3$. With the gamma basis above:

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} .$$

2.2 Lattice QCD

Lattice QCD is a numerical approach to QCD by discretization of space-time which is then simulated on a computer. Such simulations typically consist of two parts. To begin, a gluon field is generated using a probability process [8]. The second step it to compute a *quark propagator*. Operationally, this is the inversion of the Dirac operator in Equation (2.2) and with the generation of the gluon field the computationally most intensive part.

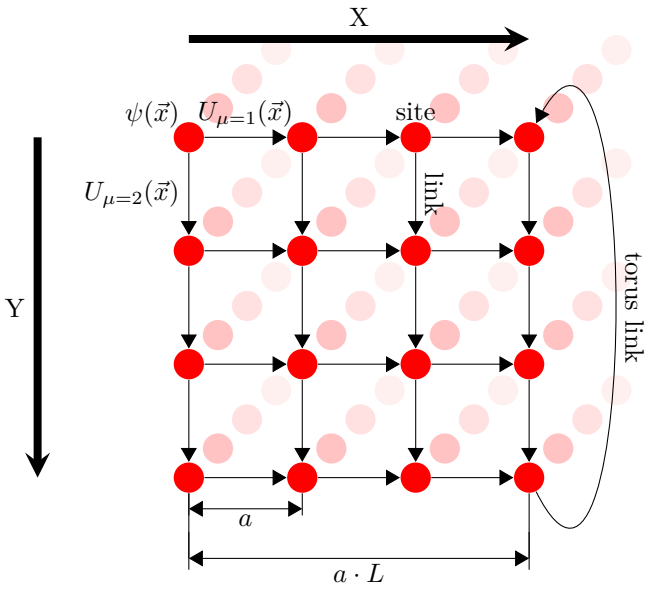


Figure 2.1: Discretized space-time (only 2 dimensions shown, a third one hinted)

The lattice version of QCD splits space and time into a Cartesian grid with interval distance a, as in Figure 2.1. Typical values for a are in the scale of a tenth of a femtometer ($10^{-16}m$) in space and $a/c = 3.3 \cdot 10^{-25}$ seconds in time dimension. The number of sites per dimension are L_t, L_x, L_y and L_z, or as a vector:

$$\vec{L} = \begin{pmatrix} L_t \\ L_x \\ L_y \\ L_z \end{pmatrix} .$$

Often the number of sites in the space dimensions are the same and identified with capital letter L and $T = L_t$ for the time dimension. The total volume covered therefore is $(aL)^3$. A site in the lattice is identified by four coordinates (t, x, y, z) or a 4-vector $\vec{x}$ for brevity. Each site holds a spinor $\psi(\vec{x})$ just as in the continuous version.

The gluon field is represented by connections between the sites. Only paths to direct neighbors – called links – exist in this model. Other paths must be constructed transitively. The boundary condition is periodic, i.e. borders are connected to each other such that the geometry of space-time is a torus. This is to reduce the impact of a hard border which does not exist in reality. tmLQCD also gradually shifts phases between sites such that walking around the torus results in a different phase when arriving at the same site again. Expressed as a formula, this is [9]

$$\psi(\vec{x}) = e^{i\theta_\mu \pi} \psi(\vec{x} + a\vec{L}) \; . \tag{2.5}$$

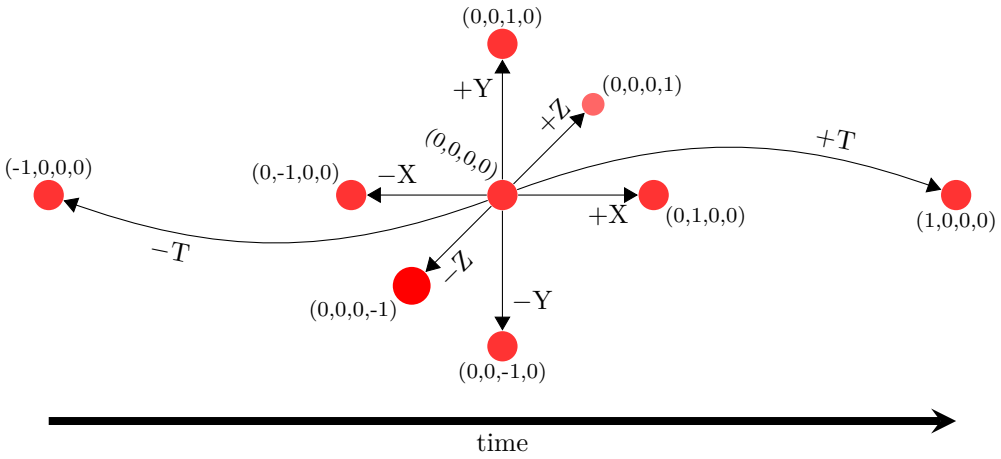


Figure 2.2: 4D neighbors

The directions from an origin vector are shown in Figure 2.2. The eight directions are named by the dimension and sign. Positive directions point to coordinates with increasing coordinate components whereas negative direction point to the opposite.

The links are annotated by the effect of the gluon field on a color charge when traversing them, which is a SU(3) matrix. This is a *link variable* U_μ instead of a description of gluons A in the continuum version. The relation between the two is

$$U_\mu(\vec{x}) = \mathcal{P} e^{-i \int_{\vec{x}}^{\vec{x}+a\hat{\mu}} g A_\mu(\vec{y}) \mathrm{d}\vec{y}} \tag{2.6}$$

where $\mathcal{P}$ signifies a path-ordered product. In this notation $\hat{\mu}$ is one of canonical vectors in dimension $\mu \in \{0, 1, 2, 3\}$, namely

$$\mu_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \mu_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad \mu_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad \mu_3 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The field A_μ hereby becomes unimportant and we continue to work on the gauge field U only. That is, we replace the description of gluon charges A with their effect on quark charges U.

This integral of Equation (2.6) is commonly approximated using

$$U_\mu(\vec{x}) = U(\vec{x}, \vec{x} + a\hat{\mu}) = e^{-iagA_\mu(\vec{x} + \frac{a}{2}\hat{\mu})}.$$

This is the effect for going into positive direction. The effect of going into negative direction is the inverse of coming from the target site and therefore

$$U(\vec{x}, \vec{x} - \vec{a\hat{\mu}}) = (U(\vec{x} - \vec{a\hat{\mu}}, \vec{x}))^{-1} = U(\vec{x} - \vec{a\hat{\mu}})^\dagger$$

(† = complex conjugate) because the gauge matrices are unitary.

2.2.1 Discrete Fermionic Action

There are various approaches on how to design a discretized version of the action of Equation (2.1). We cover the fermionic part first. The main difficulty is the derivative of the Dirac operator in Equation (2.2).

The classic replacement of the Dirac operator as suggested by Wilson himself [1, 9] – self-suggestively called the Wilson-Dirac operator D_W – is

$$\not{D}_W = \underbrace{\frac{1}{2}\sum_\mu \gamma_\mu(\nabla_\mu + \nabla^*_\mu)}_{\text{Naïve discretization}} - \underbrace{\frac{a}{2}\sum_\mu \nabla_\mu \nabla^*_\mu}_{\text{Wilson correction term}} + m$$

with

$$\begin{aligned}\nabla_\mu\psi(\vec{x}) =& \frac{1}{a}(U(\vec{x},\vec{x}+a\hat{\mu})\psi(\vec{x}+a\hat{\mu}) - \psi(\vec{x}))\\ \nabla^*_\mu\psi(\vec{x}) =& \frac{1}{a}(\psi(\vec{x}) - U(\vec{x}-a\hat{\mu},x)\psi(\vec{x}-a\hat{\mu}))\ .\end{aligned}$$

$\nabla_\mu\psi(\vec{x})$ and $\nabla^*_\mu\psi(\vec{x})$ are known as the forward gauge covariant difference operator, respectively the backward gauge covariant difference operator. The correction term is intended to eliminate doubler poles (an artifact from discretization) and vanishes for $a \to 0$, although it may be very large for finite a. The difference operators are the "slopes" of the lines between two lattice sites; the center point and one Cartesian neighbor. When inserted into crefeq:diracwilson, the contribution of the center point vanishes:

$$\frac{1}{2a}\sum_\mu \gamma_\mu \left(U(\vec{x},\vec{c}+a\hat{\mu})\psi(\vec{x}+a\hat{\mu}) - U(\vec{x}-a\hat{\mu},x)\psi(\vec{x}-a\hat{\mu})\right)\ . \tag{2.7}$$

Alternatively, there is the Wilson-Twisted Mass formulation that has one more correction term:

$$\not{D}_t exttm = \frac{1}{2}\sum_\mu \gamma_\mu(\nabla_\mu + \nabla^*_\mu) - \frac{a}{2}\sum_\mu \nabla_\mu\nabla^*_\mu + m + i\mu_q\gamma_5 \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

with the twisted mass parameter $\mu_q \in \mathbb{C}$. The matrix multiplying the γ_5 term acts in the doublet isospin space : the up- and down-quarks. The additional term also reduces the error due to discretization. Normally, the discretization artifact is in the scale of $O(a)$ and is reduced to $O(a^2)$ when using a set of parameters called "maximal twist".

2.2.2 Discrete Gauge Action

The gluons also carry charge and therefore also influence the gauge field themselves. Its gluon-gluon influence on the total action manifests in the gauge action term of Equation (2.1).

Wilson discretized the gauge action by summing up the closed loop walks (walk along links that starts and ends at the same point) on the lattice. The walk must be non-trivial (i.e. following at least one link) and not traveling links twice. The smallest such loop walks along 4 links on a square. It is called a *plaquette*. The SU(3) matrix effect collected on such a walk is

$$U_{\mu\nu}(\vec{x}) = U(\vec{x},\vec{x}+a\hat{\mu})U(\vec{x}+a\hat{\mu},\vec{x}+a\hat{\mu}+a\hat{\nu})U(\vec{x}+a\hat{\mu}+a\hat{\nu},\vec{x}+a\hat{\nu})U(\vec{x}+a\hat{\nu},\vec{x})$$

The *Wilson Gauge Action* is the sum over all plaquettes, but not going in reverse, starting at a point $\vec{x}$:

$$\frac{6}{g^2}\sum_{\mu<\nu}\left(1 - \frac{1}{6}\operatorname{Tr}\left(U_{\mu\nu}(\vec{x})U^\dagger_{\mu\nu}(\vec{x})\right)\right)$$

This thesis' focus is on the fermionic action, not the gauge action, for the sole reason that in practice programs spend more time on the Dirac operator. Optimizing it therefore saves more computation time than by putting the same effort into optimizing plaquettes.

2.3 Iterative Solver

Arguably the most computationally intensive part in tmLQCD and any Lattice QCD application is the evaluation of the Dirac operator of Equation (2.7). It is applied again and again for instance when inverting D represented as matrix. It is actually not finding D^{-1} but a solution ψ of the equation $\not{D}\psi = \phi$. Solving this equation system is already hard enough, even storing $\not{D}^{-1}$ explicitly in memory is not possible with today's computers and typical lattice sizes such as $128 \times 64^3 \approx 33.6$ million elements. This squared yields up to 1.1×10^{15} elements to be stored for an explicit representation.

Direct solvers such as Gaussian elimination, LU/Cholesky-decomposition etc. are also not preferred as they require $O(n^3)$ operations ($\sim 3.8 \times 10^{22}$ for typical lattice sizes). Iterative solvers are preferable here. These typically have a complexity of $O(kn^2)$ where k is the number of iterations which varies between iterative algorithms, but in practice k is asymptotically smaller that n.

Conjugate gradient (CG, Algorithm 2.1) is the "gold standard" used in Lattice QCD. Other algorithms are typically compared to CG. Without rounding errors it is guaranteed to return the exact result at most $k = n$ iterations. However, the result is not needed with exact precision and therefore the algorithm can abort when the solution is close enough to the optimal solution, defined by an ϵ-limit of the residual ($r = Ax - b$ in Algorithm 2.1). The number of iterations is typically sub-linear and therefore a solution can be found quicker than with a direct method.

Algorithm 2.1: Conjugate Gradient (CG) iterative solver

Input: $n \in \mathbb{N}$, $b \in \mathbb{K}^n$, $A \in \mathbb{K}^{n \times n}$
Result: $x \in \mathbb{K}^n$ such that $Ax \approx b$
$x \leftarrow \vec{0}$; // Initial guess
$r \leftarrow b$;
$p \leftarrow r$;
$n_r \leftarrow \|r\|^2$;
while $(n_r \geq \epsilon)$ **do**
  $p' \leftarrow Ap$;
  $\alpha \leftarrow \frac{n_r}{\langle p|p' \rangle}$;
  $x \leftarrow x + \alpha p$; // Next guess
  $r \leftarrow r - \alpha p'$;
  $n_{r1} \leftarrow \|r\|^2$;
  $\beta \leftarrow \frac{n_{r1}}{n_r}$;
  $p \leftarrow r + \beta p$;
  $n_r \leftarrow n_{r1}$;
end
$x = Ax - b$;

For CG to be applicable, A must be hermitian ($A^\dagger = A$) and positive definite which unfortunately is not the case for the Dirac operator. However, the matrices $A^\dagger A$ and $AA^\dagger$ have both properties and their solutions provided by CG can be used to calculate the solutions of the original systems. These variants are called *Conjugate Gradient Normal Residual* (CGNR) and *Conjugate Gradient Normal Equation* (CGNE) respectively.

Per iteration there is one application of the matrix, one scalar products, one norm, three vector additions/subtractions and three scalar-vector multiplications. The matrix-vector multiplication here is not implemented using dense matrix-vector multiplication, but a function call that applies the Dirac operator. This function is discussed in the next section. Because of its computational complexity and memory access pattern it is also the dominant part in every iteration.

2.4 Wilson-Dirac Operator Stencil

As already stated, the Dirac operator (Equation (2.7)) is one of the most important kernels of any Lattice QCD program. The operator is also called *Dslash*, D standing for Dirac and using the Feynman-slash notation $\not{D} = \sum_\mu \gamma_\mu D_\mu$ (compare Equation (2.3)). The function applying it is called *Hopping Matrix* within tmLQCD. We will use the same name in this thesis, and explore its properties in this section.

Equation (2.7) does not include all the complexities necessary. E.g. the twisted boundary of Equation (2.5) is missing. The actual Wilson twisted mass Dirac operator as implemented in tmLQCD is [9]

$$\begin{aligned}(\not{D}\psi)(\vec{x}) =&(m_0 + 4 + i\mu_q\gamma_5)\psi(\vec{x}) \\ &- \frac{\kappa}{2}\sum_\mu \Big(e^{i\pi\theta_\mu/L_\mu}U(\vec{x},\vec{x}+a\hat{\mu})(1+\gamma_\mu)\psi(\vec{x}+a\hat{\mu}) \\ &\qquad + e^{-i\pi\theta_\mu/L_\mu}U(\vec{x},\vec{x}-a\hat{\mu})(1-\gamma_\mu)\psi(\vec{x}-a\hat{\mu})\Big)\end{aligned}$$

The coefficient $\kappa \in \mathbb{R}$ is a constant and depends on the quark's mass. One may think of a weight relative to $\psi(\vec{x})$.

2.4.1 Optimizing the Formula

For the hardware, accesses to $\psi(\vec{x})$ are faster to execute that accesses to neighbor cells. Separating them makes the same-cell access hardware-predictable and also reduces the working set of the neighbor accesses (see Section 4.4). The split-up Dirac operator is

$$\begin{aligned}(\not{D}\psi)(\vec{x}) =&(m_0 + 4 + i\mu_q\gamma_5)\psi(\vec{x}) - (\not{D}'\psi)(\vec{x}) \\ (\not{D}'\psi)(\vec{x}) =&\frac{\kappa}{2}\sum_\mu \Big(U(\vec{x},\vec{x}+a\hat{\mu})e^{i\pi\theta_\mu/L_\mu}(1+\gamma_\mu)\psi(\vec{x}+a\hat{\mu}) \\ &\qquad + U(\vec{x},\vec{x}-a\hat{\mu})e^{-i\pi\theta_\mu/L_\mu}(1-\gamma_\mu)\psi(\vec{x}-a\hat{\mu})\Big).\end{aligned} \tag{2.8}$$

We simplify D' further by

- merging the coefficients $\frac{\kappa}{2}$ and $e^{i\pi\theta_\mu/L_\mu}$ into a coefficient κ_μ per dimension and therefore remove one multiplication
- exploiting the relation $U(\vec{x},\vec{x}-a\hat{\mu}) = U^\dagger(\vec{x}-a\hat{\mu},\vec{x})$
- shortening $U(\vec{x},\vec{x}+a\hat{\mu})$ to $U_\mu(\vec{x})$; with the previous rule $U(\vec{x},\vec{x}-a\hat{\mu}) = U_\mu^\dagger(\vec{x}-a\hat{\mu})$

such that we gain

$$(D'\psi)(\vec{x}) = \sum_\mu \left(\kappa_\mu U_\mu(\vec{x})(1+\gamma_\mu)\psi(\vec{x}+a\hat{\mu}) + \bar{\kappa}_\mu U_\mu^\dagger(\vec{x}-a\hat{\mu})(1-\gamma_\mu)\psi(\vec{x}-a\hat{\mu})\right) .$$

In this formulation the tensor products are hidden. A written-out formulation of the same formula is

$$\begin{aligned}(\not{D}'\psi)(\vec{x}) = \sum_\mu \Big(&\kappa_\mu(\mathbb{1}_4 \otimes U_\mu(\vec{x}))((\mathbb{1}_4+\gamma_\mu)\otimes\mathbb{1}_3)\psi(\vec{x}+a\hat{\mu}) \\ &+ \bar{\kappa}_\mu(\mathbb{1}_4 \otimes U_\mu^\dagger(\vec{x}))((\mathbb{1}_4-\gamma_\mu)\otimes\mathbb{1}_3)\psi(\vec{x}-a\hat{\mu})\Big)\end{aligned}$$

where

$\mathbb{1}_n \in \mathbb{C}^{n\times n}$ are the identity matrices

$\otimes$ is the tensor product: $A = (a_{ij}) \in \mathbb{C}^{m\times n}$, $B \in \mathbb{C}^{o\times p}$

$$A \otimes B = \begin{pmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \\ a_{m1}B & & a_{mn}B \end{pmatrix} \in \mathbb{C}^{mo\times np}$$

$\vec{x} \in \mathbb{R}^4$ is a space-time coordinate on the lattice

$\mu \in \{0, 1, 2, 3\}$ is a dimension (0=time; 1,2,3=space)

U_μ is the lattice link field in positive direction

$\gamma_\mu \in \mathbb{C}^{4\times 4}$ are the gamma matrices (Equation (2.4))

$\kappa_\mu \in \mathbb{C}$ are coefficients as defined above

ψ is the spinor field

In computer science this structure is commonly known as a stencil: An operator that is applied on each coordinate of a field and depends on data with constant offset to that coordinate. In this case, it is a 8-point stencil for each of the neighbor spinors $\psi(\vec{x} \pm a\bar{\mu})$. Including the diagonal part of Equation (2.8), it actually is a 9-point stencil, which we keep separate out of efficiency reasons. Including the gauge field it even is a 16 point stencil.

If directly implemented as a function, its data flow would resemble Figure 2.3. Every source stencil point has a computation path until the result is accumulated and stored in the target buffer ϕ. One cannot overwrite the source spinor field ψ as the original value might be used in stencil for the neighbor site.

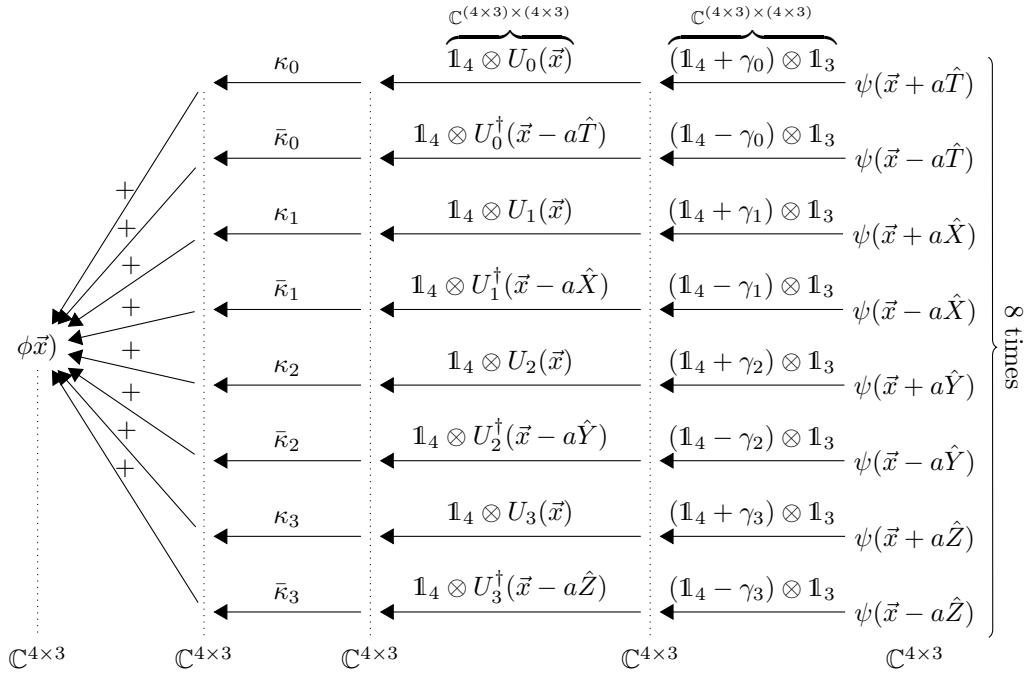


Figure 2.3: Dirac operator stencil data flow

The gauge field does not change during the execution of CG. The constant κ therefore can be multiplied in advance into the gauge field U.

$$\begin{aligned} U_\mu(\vec{x}) &\to \kappa_\mu U_\mu(\vec{x}) \\ U_\mu^\dagger(\vec{x}) = U_\mu(\vec{x} - a\hat{\mu}) &\to \kappa_\mu U_\mu(\vec{x} - a\hat{\mu}) = \kappa_\mu U_\mu(\vec{x})^\dagger = \bar{\kappa}_\mu U_\mu^\dagger(\vec{x}) \end{aligned}$$

This has not been done in in tmLQCD. Performance is usually measured in floating point operations per seconds, but with the additional κ-multiplication, more floating point operations are executed than with the optimized version without doing more work. In this thesis, we therefore provide performance numbers for both versions.

2.4.2 Spin Projection

The third and fourth line of the matrices $(\mathbb{1}_4 \pm \gamma_\mu)$ are factors of the first and second line. We can use the linear dependency by removing the redundant lines, applying any (linear) operation f on the remaining lines and then reconstructing the other lines. For instance with $\mu = 0$:

$$\begin{aligned} (\mathbb{1}_4 \otimes f)(\mathbb{1}_4 + \gamma_0) = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} (\mathbb{1}_2 \otimes f) \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \\ &= ((\mathbb{1}_4 + \gamma_\mu)P^T)(\mathbb{1}_2 \otimes f)(P(\mathbb{1}_4 + \gamma_\mu)) \\ (\mathbb{1}_4 \otimes f)(\mathbb{1}_4 - \gamma_0) = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} (\mathbb{1}_2 \otimes f) \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \\ &= ((\mathbb{1}_4 - \gamma_\mu)P^T)(\mathbb{1}_2 \otimes f)(P(\mathbb{1}_4 - \gamma_\mu)) \end{aligned}$$

The matrix P is the projection matrix that only keeps the first two rows. P and its left-inverse P^T are representable by the following matrices.

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad P^T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Therefore just half of the operations following are unnecessary. There are only 2 instead of 4 evaluations of f. In the case of the Dirac operator, f is the multiplication with the SU(3) gauge matrix and κ. A 2-component spinor may also be called *halfspinor*, 2-spinor or Weyl-spinor, in contrast to the full 4-component spinor that we call *fullspinor*, 4-spinor or Dirac-spinor. The original 4×4 matrix can be reconstructed by (in this example) negating the first respectively second line. More generally, the reconstruction matrix is built by throwing away the last two columns of the gamma matrix, or equivalently by right-multiplication by transposed P.

Incorporating this projection and reconstruction, the Dirac stencil becomes

$$\begin{aligned} (\not{D}'\psi)(\vec{x}) = \sum_\mu \Big(&((\mathbb{1}_4 + \gamma_\mu)P^T \otimes \mathbb{1}_3)\kappa_\mu(\mathbb{1}_2 \otimes U_\mu(\vec{x}))(P(\mathbb{1}_4 + \gamma_\mu) \otimes \mathbb{1}_3)\psi(\vec{x} + a\hat{\mu}) \\ &+ ((\mathbb{1}_4 - \gamma_\mu)P^T \otimes \mathbb{1}_3)\bar{\kappa}_\mu(\mathbb{1}_2 \otimes U_\mu^\dagger(\vec{x}))(P(\mathbb{1}_4 - \gamma_\mu) \otimes \mathbb{1}_3)\psi(\vec{x} - a\hat{\mu}) \Big) \end{aligned}$$

whose data flow is illustrated in Figure 2.4.

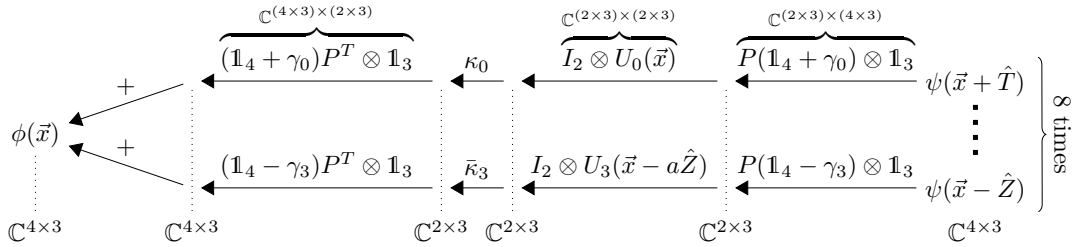


Figure 2.4: Flow in the Dirac stencil with projection

2.4.3 Number of Floating-Point Operations per Stencil

A spinor has $4 \times 3 = 12$ complex values, which makes a total of $8 \times 12 = 96$ values to read. the next step is to apply the spin projector per direction. Every projector matrix has 2 non-zero entries per row, each non-zero entry either 1, -1, i or $-i$. We do not consider multiplication by such a constant as an operation because it can be implemented trivially. For instance, instead of negating a value the next operation may be a subtraction instead of a addition. What remains is the sum of the two non-zero entries, i.e. $8 \times 2 \times 3 = 48$ complex additions.

The multiplication by the SU(3) matrices is done on both of the halfspinor vectors, each involving 6 complex additions and 9 multiplications. In total, $8 \times 2 \times 6 = 96$ additions and $8 \times 2 \times 9 = 144$ multiplications. Therefore, it is the computationally most costly part.

Multiplication by the complex number κ_μ is done individually on all (half-)spinor components, i.e. $8 \times 2 \times 3 = 48$ multiplications.

The re-expansions of the halfspinors to spinors again is trivial as every coefficient involved is either 1, i or their negatives. Summing up the results from all directions is more expensive though and done elementwise. In total $7 \times 4 \times 3 = 84$ additions take place.

The sum of one stencil operation is $48+96+84 = 228$ complex additions and $144+48 = 192$ complex multiplications. However, we do not compute complex numbers directly, but must be lowered to real number calculation. Every complex addition has the equivalent of 2 real additions:

$$\begin{pmatrix} a_r \\ a_i \end{pmatrix} + \begin{pmatrix} b_r \\ b_i \end{pmatrix} = \begin{pmatrix} a_r + b_r \\ a_i + b_i \end{pmatrix}$$

An implementation of complex multiplication needs 2 real additions and 4 real multiplications:

$$\begin{pmatrix} a_r \\ a_i \end{pmatrix} \begin{pmatrix} b_r \\ b_i \end{pmatrix} = \begin{pmatrix} a_r b_r - a_i b_i \\ a_r b_i + a_i b_r \end{pmatrix}$$

Therefore, $2 \times 228 + 2 \times 192 = 840$ real additions and $4 \times 192 = 768$ real multiplication are required, i.e. 1608 total floating-point operations.

If κ is integrated into the gauge field, some arithmetic operations are saved. The remaining number of floating-point operations is $2 \times 212 + 2 \times 144 = 712$ additions and $4 \times 144 = 576$ multiplications, i.e. $744 + 576 = 1320$ floating-point operations. This is the most often cited number in terms of complexity of Lattice QCD.

The Tables 2.1 and 2.2 give an overview on the required number of operations per stencil and the memory required per element to store element in memory. Table 2.3 also contains the amount of memory that is to be read and written for every execution of the stencil.

The computational complexity is the ratio between floating-point operations and bandwidth. As seen from the tables there are 3.93 respectively 4.79 operations (with κ multiplication) per floating-point to load. Hence, depending on whether we use single or double precision, the computational complexity of Hopping Matrix is between 0.49 and 1.2 operations per byte.

	Complex		Real		
Operation	Adds	Muls	Adds	Muls	Flops
Spin projection	48		96		96
SU(3) mul	96	144	480	576	1056
κ_μ mul		48 (0)	96 (0)	192 (0)	288 (0)
Spinor sum	84		168		168
Total	228	192 (144)	840 (744)	768 (576)	1608 (1320)

Table 2.1: Computational complexity of a single stencil

	Mathematical		Bytes in precision	
Structure	representation	Floats	single	double
Gauge	SU(3) ($\subset \mathbb{C}^{3\times3}$)	18	72	144
Fullspinor	$\mathbb{C}^3 \times \mathbb{C}^3 \times \mathbb{C}^3 \times \mathbb{C}^3$	24	96	192
Halfspinor	$\mathbb{C}^3 \times \mathbb{C}^3$	12	48	96

Table 2.2: Field element sizes

2.5 Domain Decomposition

The more sites the lattice has the more reliable the results. The size of a lattice easily grows larger than today's shared-memory computer systems can store. The amount of memory can be increased but with larger lattices one also wants to add more processors to the system. The more processors there are the more problematic their concurrent access to the same memory regions becomes.

Therefore the use of processors with distinct memories – called nodes – becomes unavoidable. The data must be split between all the nodes and every node computes just a part of the global lattice. The data split-up is done using the domain decomposition method. Every node is assigned a rectangular sublattice of the global lattice. The split may happen in any of the 4 dimensions, Figure 2.5 for instance shows a decomposition in two dimensions into an array of 3×2 nodes. The difficult part is the data exchange between the nodes when such as the Dirac operator stencil when a neighbor site is not on the local node, but a remote neighbor node.

tmLQCD uses MPI to exchange the neighbor elements between nodes. Elements in the fields are ordered such that all the elements that have to go to a remote node are in a single contiguous block of memory. When data is needed from the remote nodes, these blocks are transferred and received from the 8 neighbor nodes. The calculation requiring that data can continue when all the data has been received.

	Elements	Bandwidth for precision	
Element of	per stencil	single	double
Read gauge	8	576	1152
Read fullspinor	8	768	1536
Write fullspinor	1	96	192
Total		1344 (write 96)	2688 (write 192)

Table 2.3: Bytes of memory access per stencil execution

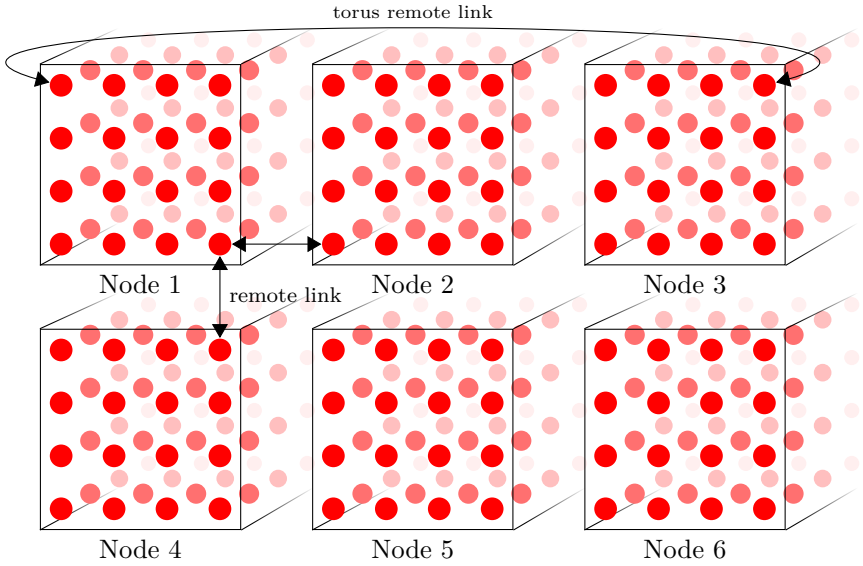


Figure 2.5: Domain decomposition of a $12 \times 8 \times L_z$ lattice with 6 nodes in a 3×2 cluster

2.6 tmLQCD

As already mentioned we base our optimization in tmLQCD. Next to a version written in plain C that works with any architecture for which there is a C compiler available, tmLQCD also has specialized versions optimized for specific computer architectures. The optimized architectures include x86 (with SSE2 or SSE3), NVIDIA GPUs and IBM Blue Gene/P. A more recent addition to tmLQCD is a straightforward optimization for Blue Gene/P without memory layout changes.

The original and general (not specific to any platform) source code for Hopping Matrix is shown in Appendix A. It is the base for all platform-specific versions including the one developed for this thesis.

3 The Blue Gene/Q Supercomputer

The Blue Gene supercomputer family from IBM has been primarily constructed for protein folding, hence the name. *Blue* for IBM's corporate identity color and *Gene* for the genomes that encode the 20 amino acids proteins are made of. But its architecture originates from *QCDOC* (QCD-On-a-Chip, [10]), a cluster computer dedicated to Lattice QCD.

Hence the Blue Gene series features everything that is also essential to Lattice QCD: Multidimensional torus network, high bandwidth links, homogeneous nodes, high memory data rate and fast a double precision floating point unit. The first product of the family, the *Blue Gene/L*, featured a width-2 vector floating point unit called *Double Hummer*. The two double precision values can be interpreted as a complex value with corresponding instructions, making it very suitable for scientific computing. The next generation, Blue Gene/P was mostly more of the same. More cores per node, slightly higher CPU frequency, faster interconnection, etc., but also introduced shared memory threading (*Symmetric Multi-Processing* (SMP)/*OpenMP*) that the predecessor did not support because it did not ensure cache coherency.

The most recent generation – Blue Gene/Q – features some characteristics that have not been available in previous commercial/off-the-shelf computers. One of the most noteworthy is transactional memory, but also the list prefetch algorithm. The processor switched to 64 bit *PowerPC* which is capable of *Simultaneous Multi-Threading* (SMT). Otherwise, Blue Gene/Q also provides "more of the same", i.e. doubled clock frequency, memory, bandwidth and the SIMD vector has been increased to 4 double precision floating point values. The new floating point unit is called *Quad Processing Extension* (QPX). Table 3.2 summarizes the technical differences between the generations.

A noteworthy point from the manufacturing perspective is that its processor is produced with 18 physical cores, but one of the cores is used as spare part in case one other is defect. It is similar Intel's and AMD's 3-core processors: Presumably these are 4-core processors with one defect, therefore disabled core, sold for a lower price that the 4 core version. In contrast, IBM's design always has one deactivated core even if it is perfectly working.

QCDOC has been designed for and only for Lattice QCD. The Blue Gene family follows the same design, although the primary purpose has changed to protein folding. The design decisions that make these platform good for Lattice QCD are high-throughput double precision floating point unit suitable for complex math, high memory and network bandwidth and a torus-shaped network that allows static workload distribution on the lattice.

3.1 Blue Gene/Q Architecture

The Blue Gene family follows the maxim of a homogeneous architecture. That is, every processing unit is equal in nature, in contrast to accelerator-based machines that offload most work from the CPU, most often to a *GPGPU* (General Purpose Graphics Processing Unit) or Intel's *MIC* (Many Integrated Cores). The major disadvantage of such heterogeneous

architectures is that they are more difficult to program. Indeed, one of the design goals was simple programmability. It still get complicated if seeking maximal performance.

The performance relevant parts of the Blue Gene/Q system are the processor, memory and the network, which are discussed in the following sections. The information presented here is basically is a summary of the official IBM documentations [11, 12, 13], computing center tutorials (IDRIS, FZ-Jülich) and presentation slides that IBM employees showed on conferences and sales events.

3.2 The Processor

The A2 processor [11] is a member of the PowerPC processor family and therefore also uses nearly the same *Instruction Set Architecture* (ISA) [12], or the 64 bit variant of it. It is clocked at 1600 Mhz and is manufactured with 18 execution cores. However, only 16 cores are available to the programmer. One of the cores serves as a replacement in case one of the cores is faulty which is deactivated before shipping. Defects may always occur during the photolithography of the die. Having a spare core reduces the number of processors that have to be thrown away.

A 17th core is reserved for system use. It is responsible for asynchronous operations like data transfer between nodes including routing. This reliefs the other cores from this work, but the main advantage is that the computations of the other cores is more deterministic: The hardware triggers an trap signal that interrupts the execution of the current program. The program continues execution when the operating system has handled the interrupt. Possible reasons are e.g. that a message from another node has been received or a timer interval has expired. The latter is configured for instance by the operating system to notify that a program's quantum has expired so another application is allocated processing time on the CPU. The 16 user-programmable Blue Gene/Q nodes, by default, do not have such interrupts unless the application configures the hardware to do so. Therefore no more than 4 threads can run on a core (no), but also those threads are guaranteed to run without pauses (see also Section 3.2.4).

3.2.1 The Core

Every A2 processor core consists of an *Instruction Unit* (IU) and an *Execution Unit* (XU) which together execute the standard PowerPC instructions. In addition, the A2 has an interface for a third function unit, the *Auxiliary Unit* (AXU), shown in Figure 3.1. The AXU of the Blue Gene/Q version of the A2 is the QPX functional unit or *QFPU* (Quad-vector Floating-Point Unit) which replaces the floating point and AltiVec units of ordinary PowerPC processors and the Double Hummer unit that the Blue Gene/L and Blue Gene/P had. A QPX register contains 4 floating point values in IEEE 754 double precision format, i.e. 256 bits. There is no support for single precision calculations, but values can be loaded and stored from memory from/to single precision and converted on the fly to/from double precision. The XLc and XLc++ compilers have a special data-type named `vector4double` to hold the data of one register. Syntactically it resembles an array of 4 doubles but can be passed around like a `struct`.

Most operations on QPX registers are defined elementwise. In case of non-vector instructions just the first 64 bits are actually used and defined in the output. One of the most powerful instruction is the fused multiply-add (**qvstfdux**) which multiplies two elements and adds a third, summing up to a total of 8 floating point operations. There are also variants that subtract instead or negate the result. Other variants interpret the 4 double values as 2 complex values with real and imaginary part. These are designed such that one complex multiplication can be done with two instructions. In total there are 8 different *Fused Multiply-Add* (FMA) instructions. In contrast, the Double Hummer unit held only 2 double precision values (128

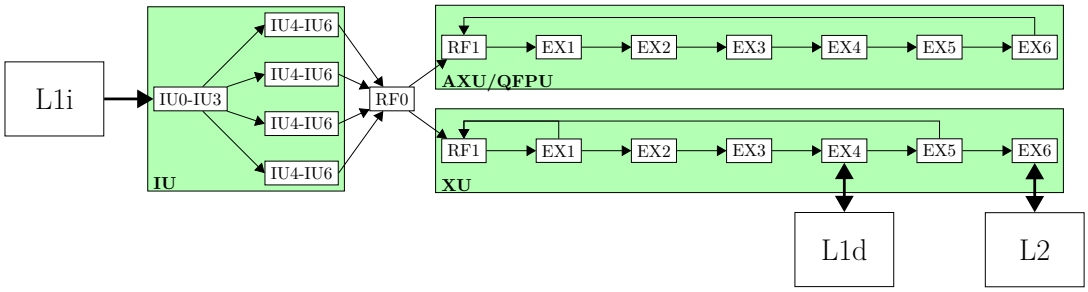


Figure 3.1: A2 processor pipeline

bits) that could be directly interpreted as a complex value, but 24 different FMA instructions such that every combination of swapping and negating an argument is possible.

In addition the QPX unit has instructions for reordering elements as well as loading and storing elements from/to memory in various variants. Memory access instructions are not executed by the AXU itself but by the XU as it has no memory interface. Instructions not interesting for Lattice QCD are estimation of the reciprocal and square root, conversions from and to integer, and instruction that do boolean logic.

The instruction unit is responsible for reading the instruction stream (usually from the L1 instruction cache (L1i) which may contain up to 4096 instructions) and decoding them in stages IU1 to IU3. It can read and decode up to 4 consecutive instructions (16 bytes) at once. Decoded instructions are put into one of four instruction buffers of stage IU4. It can keep track of up to 4 instruction streams, each also having its own buffer of 8 instructions. Stage IU5 dequeues an instruction from the buffer, checks whether it is ready to execute (i.e. the input data is available) and if so, is held by IU6 until the required function is available. The instruction streams correspond to the 4-way SMT ("Hyperthreading"), therefore the stages IU4, IU5 and IU6 each exist four times. IU0 to IU3 exist once only, they alternate between the streams they decode. The instruction unit never reorders instructions, i.e. the A2 is an in-order processor.

Both functional units – XU and AXU– have a pipeline depth of up to 6 cycles. Both units can also issue one instruction per cycle except a few instructions that are microcoded. One core can therefore issue 2 instructions if the instructions use different functional units. Data dependencies may bypass the register file in stage RF0 and RF1.

Different classes of instructions have a different pipeline depth. Most integer-arithmetic instructions are executed directly in EX1 such that the next instruction may receive the result in the bypass phase RF1. The effective latency therefore is 1, i.e. the instruction appears to execute in one cycle only.

Most QPX instructions have a latency of 6, i.e. there must be 5 cycles between the instruction and an instruction using its result. The 5 cycles can be filled up with non-dependent instructions, instructions from other threads, or bubbles in the pipeline that execute nothing. The latter results in a performance reduction, called penalty cycles. The FPU handles denormalized floats, infinities and NaNs in hardware without microcoding, i.e. at full speed.

Memory access instructions have more performance-relevant details to consider. The first stages are equal for read and write accesses: EX1 computes the effective (virtual) address to access. For instance, the assembly signature of the load instruction is `ld R,S` where R is a register and S is another register or the constant 0. The effective address is the sum of both registers, respectively the content of R only. The addition is performed in EX1.

EX2 does the virtual-to-physical address translation by looking up the Data-ERAT. The *ERAT* (Effective-to-Real Address Translation) is a cache of the *Translation Lookaside Buffer* (TLB), also called shadow-TLB. Since the Blue Gene/Q is intended for high-performance

computing systems are usually configured such that the complete memory mapping fits into the ERAT and TLB misses never occur[1]. There is another shadow TLB for instructions, the I-ERAT. Also EX2 contains the test whether, and if so at which position, the data is in the L1d cache.

From EX3 on the actions depend on whether the access is a read or write, and whether it is an L1 hit or miss. In case of a load that hits the position looked up in the previous stage is sent to the L1d SRAM in EX3 and received in EX4. The data is then stored in the register file and/or bypassed to the following instruction after stage EX5. The latency of a L1d load hit therefore is 5 cycles. However, floating-point loads (loads to a QPX register) have a latency of 7 due to the missing bypass to the other functional unit.

If the line is not found in the L1d, EX3 and EX4 effectively do nothing. Instead, the request is appended to the *LMQ* (Load-Miss Queue) that holds up to 8 elements, waiting to be sent to the L1p and L2 caches. If it is the front entry, the request may already be sent in EX5. While in the queue, the pipeline may continue executing instructions from the same thread, so in this case the processor does some instruction reordering.

For store instructions, if the L1d hits in EX2, the position is also sent in EX3, but then the cache line updated in EX4. If it missed, both stages effectively do nothing. In both cases the data to be written is sent to the L2 cache in stage EX5. There is no special queue for writes, the L2 is expected to buffer write requests. The A2 therefore implements a write-through no-write-allocate strategy at the level 1 cache.

In case something does not happen as expected, the core "panics" with a *flush*. A flush removes all instructions of a thread that are somewhere in the pipeline and restarts execution after the last committed (successfully executed) instruction. Flushes are not that exceptional, for instance it occurs when data arrives from the L2 cache and there is a read instruction in EX4 at the same time. The L1d has a single read/write port such that it can handle just one of the transactions. In this case writing the data from L2 has priority and the thread trying to read is flushed. Another flush condition is writing at an address that is in the LMQ or the LMQ is full. Flushing ensures the progress of the instructions of the other threads. Stalling the pipeline would have the consequence that none of the threads could continue.

3.2.2 Theoretical Performance per Node

Using the aforementioned technical details we can compute the peak performance per core. In scientific computing performance is usually measured in floating-point operations per second (Flop/s). For applications whose bottleneck is the FPU the effectiveness of the code can be measured in fractions of this theoretical peak performance.

The theoretical maximal number of floating-point operations is calculated using the formula

$$8 \text{ Flops per FMA} \times 16 \text{ cores} \times 1.6 \text{ Ghz} = 204.8 \text{ GFlop/s}.$$

This speed corresponds to a code that executes nothing but FMA instructions in all 16 cores with a gap of at least 5 instructions between dependent instructions. SMT is irrelevant here because the XU does not contribute to any floating point operations. The FMA instruction stream has to fit into the L1i cache though, i.e. up 4096 instructions in a row.

3.2.3 Blue Gene Performance Monitoring

The predecessor Blue Gene/P had hardware performance counters that were difficult to configure and maybe even more difficult to interpret. For instance, only 2 of the 4 cores could be configured to use hardware performance counters.

[1] They still occur when accessing the NULL pointer which is not mapped in the ERAT

This has changed on the Blue Gene/Q. All of the cores can be monitored at the same time, only the number of counters is limited. The interface to enable them is called *Blue Gene Performance Monitoring* (BGPM). Using BGPM one can count the number of executed instructions of a class (floating-point, memory access, etc.), number of pipeline flushes, cache hits and misses, prefetch activity, etc.

The counters can also queried using the standard programming interface *PAPI* (Performance API). However, only a subset of performance counters can be queried using PAPI.

3.2.4 Compute Node Kernel

The nodes run a lightweight operation system called *CNK* (Compute Node Kernel). A static portion of the main memory is reserved for it, so not all memory is available to the user application, though it is only a few megabytes.

The CNK tries to emulate the system calls of a Linux kernel, such that most programs that run on the front-end also run on the nodes. However, use of shared library is discouraged and of course a terminal input stream is not available.

By default, the CNK does not configure a timer interrupt. The user program therefore can execute predictably without quantum elapses that switch threads. Only the 64 hardware threads can execute.

3.3 Memory

The memory hierarchy of a node has 4 levels, not including the thread registers. Details on the properties of the individual levels can also be found in Table 3.2.

3.3.1 Main Memory

Every node has 16 GB of DDR3 memory, shared between the all 17 cores. Physical addresses are interleaved between banks such that consecutive memory accesses combine the bandwidth of all memory banks.

3.3.2 L2

The last level cache is the L2, also shared between all cores, but located on the chip. Its size is 32 MB in 16 slices. The smallest memory unit (line size) is, like the main memory's, 128 bytes. Every physical address has 16 locations it might be stored at, i.e. 16x associativity. The 16 locations are replaced using a least-recently-used (LRU) strategy.

The L2 does not only store data, but has functionality on its own. Very fast atomic instructions are implemented on this level. 16 such instructions are available, e.g. interlocked increment, interlocked xor, interlocked swap etc.. They may replace operating system synchronization objects like semaphores/`pthread_mutex` and the older **lwarx**/**stwcx** instructions. In contrast to them, L2 atomic operations scale well to all 64 threads.

Also working on the level of the L2 cache is the transactional memory. Every cache line gets a tag corresponding to the version of the line. Different threads may work on different sets of cache line version, i.e. the same physical address might exist multiple times in the cache. Only when one threads decides to commit its version, cache lines with different version tags are thrown away. The threads that used these other versions must restart their work. Speculative execution is also implemented using these tags.

In addition, the *Direct Memory Access* (DMA) works by directly accessing the L2. For instance the networking hardware directly reads from L2 and writes received data directly to the L2.

3.3.3 L1

Every core has two level 1 caches: The L1i for instruction fetching and the L1d for data, each 16 KB in size. Data between the L2 and L1 caches is transferred using a crossbar switch. The crossbar connects all 17 cores with the 16 L2 slices. As a result, only one core can access a slice at once. The cache line size is smaller than the L2's: 64 bytes. Cache lines are replaced using an approximated (pseudo-) LRU policy. The L1d has an associativity of 8x while the L1i has an associativity of 4 (matching the 4 SMT threads). If the core is charged with 4 threads, effectively only 4 KB are available per thread.

When enabled, the L1 use a lock bit for every cache line. When set by a special instruction, the cache line is never evicted. It allows exact control on which lines are in the cache. Code can fetch a cache line before the first use, set the flag, and release it after the last use.

3.3.4 L1p

Between the L1 and L2 caches there is a special-purpose cache called the L1p. It serves as buffer for the in-hardware data prefetchers. The predecessor generations also had such a special-purpose cache but were called L2. Correspondingly, the predecessor's last level cache was named L3. Like the L1, every core has its own L1p of 4 KB. The hardware can follow up to 16 data streams, i.e. there are only 2 cache lines of 128 bytes available per stream.

3.3.5 Prefetchers

Compared to the execution speed of the floating-point unit, access to the main memory is embarrassingly slow. Execution of a multiplication has a latency of 6 cycles, and 6 of them can in the pipeline at the same time in different execution stages. Fetching data from the main memory takes mores than 350 cycles. This is why there are multiple cache levels. The closer the cache is to the core, the lower the latency is.

Without a prefetcher only the most recently used data is in the cache. Any data used for the first time, or a long time since the last access, need to be fetched from the lower levels with their high latencies. The prefetcher is supposed to estimate which datum is going to be used in the near future, and load it into the cache in advance. When done with low priority, the performance penalty when the prefetcher mispredicts is small.

Both Blue Gene/Q hardware prefetchers load data into the L1p cache. If the prefetcher decides to load some address, a request is sent to the L2 cache. If the data is not available in the L2, it is fetched from main memory. From the L2's point of view, there is no difference between normal memory accesses and those triggered by a prefetcher.

3.3.5.1 Stream Prefetcher

The stream prefetcher loads consecutive memory addresses in direction of increasing virtual addresses. It can track up to 16 streams. Once a stream has been established, it prefetches up to 8 cache lines (8x 128 = 1024 bytes) in advance of the last load instruction reading that stream. This number of cache lines to be read is called the depth. The depth can be configured to a fixed number or to be adaptive. In adaptive mode the depth shortens and lengthens dynamically.

There is also the possibility to configure when a stream is established. In optimistic mode every L1 cache miss makes the prefetcher load the following cache lines. And there is the confirmed mode where the prefetcher waits until a second line in the prefetch depth also causes a miss. Finally, the stream prefetcher can be disabled. The stream prefetcher can also be disabled entirely.

The limitation to 16 lines means that if 4 threads are used per core, only 4 streams are available per thread.

3.3.5.2 List Prefetcher

The list prefetcher is also called the perfect prefetcher because its goal is to prefetch all data without any mispredicts. It does so by recording all L1 cache load miss addresses into a list of memory accesses. This load miss sequence is later replayed with some cache lines in advance. Obviously, this only makes sense if the code portion has the same read pattern for multiple executions: Record the first execution, and reuse this information for prefetching the following executions.

There is some tolerance when the replay does not exactly match the recorded pattern. The hardware always sees the 8 next misses in the list. If a actual miss is not in the list it is ignored, but a counter is increased. If the counter exceeds a configurable threshold the list is assumed to be out of sync and the prefetching stops. If the actual miss is in the list, the following number of lines addresses are prefetched, where that number is the configurable prefetch depth. The cache miss might not be the first in the list. It means that up to 7 misses can be skipped. The loads either hit in the cache or have not been executed due to a conditional statement. In total, the prefetcher keeps an active queue of the next 24 addresses. New addresses are fetched from L2 when dequeuing addresses from the front.

Every the thread has its own list prefetcher to track up to four different lists per core. List- and stream prefetcher can be used at the same time. There is a configuration parameter for setting how many of the L1p cache lines are available to the stream prefetcher and how many are left to the perfect prefetcher.

3.3.5.3 Cache Line Prefetch Instruction

There is also an explicit way of prefetching data into the L1d cache before its actual use. This is done by the **dcbt** (Data Cache Block Touch) instruction. It fetches the 64 byte cache line the indicated effective address points to. It is also non-blocking, i.e. following instructions cannot depend on it and cause a stall. However, like every instruction it has an overhead for being decoded by the instruction unit and then executed by the XU. It delays the following instruction by about one cycle.

A **dcbt** instruction is generated by XLc using the `__dcbt` intrinsic or with *GCC* using the `__builtin_prefetch` builtin. The XLc also has an intrinsic `__prefetch_by_load` which generates a **lbz** (load byte and zero) instruction. Since the loaded byte is not used, it neither causes a dependency stall (and waits in the LMQ until the data arrives), but requires a spare register and causes an error if the address in question does not belong to a virtual memory region (in the TLB or D-ERAT).

In contrast to the hardware prefetchers, **dcbt** transfers data to the L1d cache. The latency to access the L1i is smaller than from L1p (see Table 3.2), concluding that for best performance we need explicit prefetching even for data fetched by the hardware prefetchers. The data comes from the L1p instead though, so the **dcbt** can be closer to the actual load instruction.

3.3.5.4 Stream Prefetch Instructions

Prefetch streams are also established using the **dcbt** instruction. It optionally takes an additional parameter indicating the direction of a stream. “1” means the stream follows increasing addresses and “3” means it follows decreasing addresses. Mode 3 is not supported since the A2 prefetcher only supports forward direction without holes. The documentation [11, 12] is unclear about whether even the first version is supported, or any **dcbt** triggers a stream establishment if configured.

3.4 Network

The main network of a Blue Gene/Q cluster is the torus network extended into 5 dimensions. The Blue Gene/L also had a separate collective network, for broadcasting and reduction operations (global sum, etc.), but in a Blue Gene/Q machine it is mapped on the torus network. Every node has 11 bi-directional networks links: 10 to the nearest neighbor nodes and one to an I/O node. The I/O node connects to the "outside world" like disk storage typically connected with InfiniBand. Each link has a bandwidth of 2 GB/s in each direction, summing up to a total bandwidth of 44 GB/s on a node.

Name	Nodes	Shape (AxBxCxDxE)	Wraparound
Node board	32	2x2x2x2x2	E
Pair	64	2x2x4x2x2	C,E
Quadrant	128	2x2x4x4x2	C,D,E
Halves	256	4x2x4x4x2	A,C,D,E
Midplane	512	4x4x4x4x2	A,B,C,D,E
Rack	1024	8x4x4x4x2 4x8x4x4x2 4x4x8x4x2 4x4x4x8x2	A,B,C,D,E

Table 3.1: Small Blue Gene/Q job sizes

The network hardware is able to transfer data asynchronously without interaction with the running program. Users add messages to send to a injection FIFO (a queue), respectively data that they want to receive into a reception FIFO. A special unit on each core – the *Messaging Unit* (MU)– will transfer the messages one after the other. The MU can transfer multiple messages in parallel if put in different FIFOs. Programs must actively query the MU for knowing whether their transfer is done (busy waiting). The interface between MU and software is called MUSPI.

MUSPI is close to the hardware and difficult to program, especially because it sends and receives data directly to/from the L2 cache that operates on physical addresses, not the virtual addresses the programmer sees. For this reasons there are abstraction layers to make the transfer of messages easier for the programmer. Actually, there are even two such abstractions: *Parallel Active Message Interface* (PAMI) and MPI.

MPI is the standard transport layer for about 20 years now and exists for virtually every cluster computer. Not surprising most software written for DMMs use MPI.

PAMI is an interface developed by IBM itself with the goal to reduce overhead. MPI has the problem that implementations must closely adhere the standard, resulting that, for example, the MPI implementations often have to copy the message within local memory. PAMI is more flexible in the regard, handles asynchronicity and multi-threading better than MPI, but is only available for recent machines from IBM. On Blue Gene/Q, MPI is actually implemented using PAMI.

A node coordinate in the 5D torus is built from the components AxBxCxDxE. The E dimension is fixed to a width of two nodes, i.e. a node that sends data into the up and down direction of dimension E communicates with the same node. The bi-directional speed between these two therefore is 8 GB/s.

The width of the other dimensions depends on the job size, seen in Table 3.1. The smallest job size possible is one node board with 32 nodes organized as a hypercube of size 2x2x2x2x2. Smaller jobs must leave some nodes idle. Also, the shape is a mesh rather than a torus, only the dimension E is circle-shaped. Dimensions gradually become circles until a midplane-sized

job which is torus-shaped in all dimensions. Larger jobs are multiples of midplanes, and can be appended to dimensions A to D to form a larger torus. For instance, a rack with 2 midplanes might be shaped 8x4x4x4x2 or 4x8x4x4x2. The job scheduler may transparently exchange the physical dimensions (rotate) to fit the requested shape.

If executing more than one *Single Program Multiple Data* (SPMD) rank per node by artificially dividing the node's memory between multiple processes, the additional ranks are interpreted as a 6th dimension called "T". Using PAMI or MPI those ranks can directly communicate to each other over shared memory. Hence virtually, every rank connects to every other as long they are executed on the same node. The MUSPI does not support such in-memory transfers (i.e. memcpy) because the MU refuses to do so.

	Blue Gene/Q [13]	Blue Gene/P [14]	Blue Gene/L [15]	QCDOC [16]
Core				
CPU name	PowerPC A2	PowerPC 450d	PowerPC 440d	custom 440
ISA	64-bit POWER	32-bit POWER	32 bit POWER	32 bit POWER
Frequency	1600 Mhz	850 Mhz	700 Mhz	500 Mhz
SIMD	4x double (256 bit)	2x double (128 bit)	2x double (128 bit)	1x double (64 bit)
SMT	4 threads	1 thread	1 thread	1 thread
Peak	12.8 GFlop/s	3.4 GFlop/s	2.8 GFlop/s	1 GFlop/s
L1i cache	16 KB	32 KB	32 KB	32 KB
L1d cache	16 KB	32 KB		
L1d line size	64 byte	32 byte		
L1d latency	6 cycles	3 cycles		
L2/L1P cache	4 KB	3.5 KB		
L2/L1P line size	128 byte	128 byte		
L2/L1P latency	24 cycles	11 cycles		
Prefetch streams	16	14	14	
Node				
Cores	16+1	4	2	1
Peak	204.8 GFlop/s	13.9 GFlop/s	5.6 GFlop/s	1 GFlop/s
L3/L2 cache	32 MB	8 MB	4 MB	4 MB
L3/L2 line size	128 byte	128 byte		
L3/L2 latency	82 cycles	50 cycles		
Main memory	16 GB DDR3	2 or 4 GB DDR2	0.5 or 1 GB DDR	up to 2 GB DDR
Bandwidth	42.6 GB/s	13.6 GB/s	5.6 GB/s	2.6 GB/s
Latency	$\geq$ 350 cycles	104 cycles		
Rack & Network				
Nodes	1024	1024	1024	512[1]
Networks	three[2]	five[3]	five	three[4]
Peak	209.7 TFlop/s	13 TFlop/s	5.7 TFlop/s	512 GFlop/s
Torus dims	5	3	3	6
Torus shape	4x4x4x8x2	8x8x16	8x8x16	2x2x2x2x2x2
Bandwidth	2 GB/s	425 MB/s	175 MB/s	62.5 MB/s
Latency	80 ns	100 ns	200 ns	
Largest system	Sequoia[5]: 96 Racks,20 PF/s at LLNL	Jugene[6]: 72 Racks,1 PF/s at FZ-Jülich	BlueGene/L[7]: 104 Racks,596 TF/s at LLNL	QCDOC[8]: 16 Backplanes, 6.6 TFlop/s at BNL

[1] per backplane [2] Torus grid, IO over PCI-e+InfiniBand QDR, JTAG [3] Torus grid, collective tree, 10G Ethernet, barrier/interrupt network, JTAG [4] Torus grid, 100M Ethernet, interrupt tree [5] `http://www.top500.org/system/177556` [6] `http://www.top500.org/system/176321` [7] `http://www.top500.org/system/175171` [8] `http://www.top500.org/system/174752`

Table 3.2: QCDOC and Blue Gene family comparison

4 Manual Optimizations

Given the characteristics of the hardware and the definition of the problem to compute we have to find the most promising optimizations. These are presented and discussed in the following sections. The emphasis will be on computational optimization, and only few remarks on the arithmetic level. Most of it appears in the following section.

4.1 Separation of Even and Odd Elements

A very common optimization on the calculus for Lattice QCD is even/odd preconditioning. Using this, the large equation system becomes separated into smaller ones that are presumably easier to solve. Coming from the equation system $\not{D}\psi = \phi$, where ψ is unknown, it is transformed to the preconditioned equation system

$$\begin{pmatrix} \not{D}_{ee} & \not{D}_{eo} \\ \not{D}_{oe} & \not{D}_{oo} \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix} = \begin{pmatrix} \phi_e \\ \phi_o \end{pmatrix}$$

Where ψ_e, ϕ_e contain sites with even coordinates only, ϕ_o, ϕ_o sites with odd coordinates. Oddness of the site is the oddness of the sum of the coordinates, i.e. $(t + x + y + z) \mod 2$ equal to 0 (even) or 1 (odd) respectively. The result is a checkerboard-like pattern, but in 4 dimensions.

The matrix D then divides into 4 sub-matrices for the 2×2 combinations of oddness of ψ and ϕ. In case of Wilson(-twisted mass) fermions, the $\not{D}_{ee}$ and $\not{D}_{oo}$ happen to be diagonal and therefore easy to invert or multiply with. Solving the upper line to ψ_e gives $\psi_e = \not{D}_{ee}^{-1}(\phi_e - \not{D}_{eo}\psi_o)$ with two unknowns ψ_e and ψ_o. Replacing ψ_e by this expression in the bottom line gives

$$\not{D}_{oe}\not{D}_{ee}^{-1}(\phi_e - \not{D}_{eo}\psi_o) + \not{D}_{oo}\psi_o = \phi_o$$

and reordered

$$\left(\not{D}_{oo} - \not{D}_{oe}\not{D}_{ee}^{-1}\not{D}_{eo}\right)\psi_o = \phi_o - \not{D}_{oe}\not{D}_{ee}^{-1}\phi_e$$

which can be solved for ψ_o. Reinsertion then also allows to compute ψ_e.

This is an algorithmic trick to solve a system of just half the size of the original, but lends itself to store the even and odd parts of ψ and ϕ in two different arrays in memory, not naïvely interleaved. Interleaving meant that for the computation of one oddness the data of the other oddness stays untouched but occupies space and bandwidth in/to the caches.

But even without even/odd preconditioning, separation of even and odd sites makes sense to reduce the size of the working sets. The 4-dimensional Hopping Matrix stencil only depends

on neighboring sites, i.e. even sites only read from odd sites an vice versa. Computing them non-interleaved but in succession reduces the working set size per part during the computation as illustrated by Figure 4.4 in Section 4.4.1, Page 52.

4.2 Floating-Point Instruction Vectorization

In some situations today's compilers, including IBM's XLc, can vectorize floating-point operations as well as if done manually. However during experiments, the compiler never generated the same quality of manually vectorized instructions. Therefore, we opt to vectorize everything manually. Using compiler intrinsic functions, this is not too hard to accomplish.

Nearly all required operations are on complex numbers. Whereas on Blue Gene/P the vector registers map directly to real and imaginary part of complex numbers and every possible combination of FMA instruction between them are available, the designers of Blue Gene/Q chose only 6 FMA instructions for the 4 double precision values in its registers. These are [17]:

qvstfdux Quad-Vector Floating-point FMA

$$dst[0] \leftarrow acc[0] + (src_A[0] * src_B[0])$$
$$dst[1] \leftarrow acc[1] + (src_A[1] * src_B[1])$$
$$dst[2] \leftarrow acc[2] + (src_A[2] * src_B[2])$$
$$dst[3] \leftarrow acc[3] + (src_A[3] * src_B[3])$$

qvfmsub Quad-Vector Floating-point Multiply-Subtract

$$dst[0] \leftarrow -acc[0] + (src_A[0] * src_B[0])$$
$$dst[1] \leftarrow -acc[1] + (src_A[1] * src_B[1])$$
$$dst[2] \leftarrow -acc[2] + (src_A[2] * src_B[2])$$
$$dst[3] \leftarrow -acc[3] + (src_A[3] * src_B[3])$$

qvfnmadd Quad-Vector Floating-point Negative FMA

$$dst[0] \leftarrow -acc[0] - (src_A[0] * src_B[0])$$
$$dst[1] \leftarrow -acc[1] - (src_A[1] * src_B[1])$$
$$dst[2] \leftarrow -acc[2] - (src_A[2] * src_B[2])$$
$$dst[3] \leftarrow -acc[3] - (src_A[3] * src_B[3])$$

qvfxmadd Quad-Vector Floating-Point Cross FMA

$$dst[0] \leftarrow acc[0] + (src_A[0] * src_B[0])$$
$$dst[1] \leftarrow acc[1] + (src_A[0] * src_B[1])$$
$$dst[2] \leftarrow acc[2] + (src_A[2] * src_B[2])$$
$$dst[3] \leftarrow acc[3] + (src_A[2] * src_B[3])$$

qvfxxnpmadd Quad-Vector Floating-Point Double-Cross FMA

$$dst[0] \leftarrow acc[0] - (src_A[1] * src_B[1])$$
$$dst[1] \leftarrow acc[1] + (src_A[0] * src_B[1])$$
$$dst[2] \leftarrow acc[2] - (src_A[3] * src_B[3])$$
$$dst[3] \leftarrow acc[3] + (src_A[2] * src_B[3])$$

qvfxxcpnmadd Quad-Vector Floating-Point Double-Cross Conjugate FMA

$$dst[0] \leftarrow acc[0] + (src_A[1] * src_B[1])$$
$$dst[1] \leftarrow acc[1] - (src_A[0] * src_B[1])$$
$$dst[2] \leftarrow acc[2] + (src_A[3] * src_B[3])$$
$$dst[3] \leftarrow acc[3] - (src_A[2] * src_B[3])$$

qvfxxcpnmadd Quad-Vector Floating-Point Double-Cross FMA
$dst[0] \leftarrow acc[0] + (src_A[1] * src_B[1])$
$dst[1] \leftarrow acc[1] + (src_A[0] * src_B[1])$
$dst[2] \leftarrow acc[2] + (src_A[3] * src_B[3])$
$dst[3] \leftarrow acc[3] + (src_A[2] * src_B[3])$

qvfxmul Quad-Vector Floating-Point Cross Multiply
$dst[0] \leftarrow src_A[0] * src_B[0]$
$dst[1] \leftarrow src_A[0] * src_B[1]$
$dst[2] \leftarrow src_A[2] * src_B[2]$
$dst[3] \leftarrow src_A[2] * src_B[3]$

Each except the last do 8 floating-point operations. **qvfxmul** can be seen as a **qvfxmadd** with zero-initialized accumulator register.

In the representation chosen here there is one target register for the result (dst), two factor registers (src_A and src_B) and one accumulator register to add the result to (acc). **qvfmsub** and **qvfnmadd** violate this rule in that the accumulator is negated before the multiply result is added. However, the view as accumulator makes sense with the cross-form instructions. All 4 registers, the three input and the target register, can be freely chosen out of the set of 32 registers, so called "FMA4" operations.

All instructions do the same operations on the third and fourth elements as on the first and second elements. They are designed such that the two elements can represent the real and imaginary parts of a complex number. Hence, a QPX register stores 2 complex numbers such that in the following we only mention the complex operation, knowing that the exact same operations are also performed on the complex consisting of the third and fourth element. We use the symbol `re` for the real parts in registers $reg[0]$ and $reg[2]$, and `im` for the imaginary parts of the double-precision floats of register elements $reg[1]$ and $reg[3]$.

Other useful operations are those that change the order in the vector registers. This might be necessary if an operation's argument is available in one (or two) vector register(s), but not at the right position. A *splat* which writes one value to all 4 vector elements. *Align* rotates the elements by an offset. The permute operation is the most flexible one and can assign to each element individually an arbitrary element from two other registers. The i arguments of **qvesplati** and **qvalign** are encoded into the instruction, but the i-arguments for **qvfperm** are taken from a specially encoded forth register.

qvesplati Quad-Vector Element Splat Immediate i
$dst[0] \leftarrow src[i]$
$dst[1] \leftarrow src[i]$
$dst[2] \leftarrow src[i]$
$dst[3] \leftarrow src[i]$

qvalign Quad-Vector Align i
$dst[0] \leftarrow src[(0+i)\%4]$
$dst[1] \leftarrow src[(1+i)\%4]$
$dst[2] \leftarrow src[(2+i)\%4]$
$dst[3] \leftarrow src[(3+i)\%4]$

qvfperm Quad-Vector Floating-point PERMute i_0, i_1, i_2, i_3
$dst[0] \leftarrow (i_0 < 4)?src_A[i_0] : src_B[i_0 - 4]$
$dst[1] \leftarrow (i_1 < 4)?src_A[i_1] : src_B[i_1 - 4]$
$dst[2] \leftarrow (i_2 < 4)?src_A[i_2] : src_B[i_2 - 4]$
$dst[3] \leftarrow (i_3 < 4)?src_A[i_3] : src_B[i_3 - 4]$

Complex operations can be built with a series of instructions as seen in the following.

Complex addition $lhs + rhs = \begin{pmatrix} lhs.re & + & rhs.re \\ lhs.im & + & rhs.im \end{pmatrix}$

qvfadd lhs, rhs
$re \leftarrow lhs.re + rhs.re$
$im \leftarrow lhs.im + rhs.im$

Complex subtraction $lhs - rhs = \begin{pmatrix} lhs.re & - & rhs.re \\ lhs.im & - & rhs.im \end{pmatrix}$

qvfsub lhs, rhs
$re \leftarrow lhs.re - rhs.re$
$im \leftarrow lhs.im - rhs.im$

Complex multiplication $lhs \cdot rhs = \begin{pmatrix} lhs.re * rhs.re & - & lhs.im * rhs.im \\ lhs.im * rhs.re & + & lhs.re * rhs.im \end{pmatrix}$

qvfxmul $src_A = lhs$, $src_B = rhs$
$re \leftarrow lhs.re * rhs.re$
$im \leftarrow lhs.im * rhs.re$
qvfxxnpmadd $src_A = rhs$, $src_B = lhs$
$re \leftarrow re - rhs.im * lhs.im$
$im \leftarrow im + rhs.re * lhs.im$

Complex addition and multiplication are commutative, therefore arguments can also be exchanged.

Conjugated multiplication $lhs \cdot \overline{rhs} = \begin{pmatrix} lhs.re * rhs.re & + & lhs.im * rhs.im \\ lhs.im * rhs.re & - & lhs.re * rhs.im \end{pmatrix}$

qvfxmul $src_A = lhs$, $src_B = rhs$
$re \leftarrow lhs.re * rhs.re$
$im \leftarrow lhs.im * rhs.re$
qvfxxcpnmadd $src_A = rhs$, $src_B = lhs$
$re \leftarrow re + rhs.im * lhs.im$
$im \leftarrow im - rhs.re * lhs.im$

Complex vector-vector scalar product $\sum_{i=0}^{N-1} vec_{lhs}[i] * \overline{vec_{rhs}[i]}$

qvfxmul $src_A = vec_{lhs}[0]$, $src_B = vec_{rhs}[0]$
$re \leftarrow lhs.re * rhs.re$
$im \leftarrow lhs.im * rhs.re$
qvfxxcpnmadd $src_A = vec_{rhs}[0]$, $src_B = vec_{lhs}[0]$
$re \leftarrow re + rhs.im * lhs.im$
$im \leftarrow im - rhs.re * lhs.im$
for $i = 1..N-1$ **do**
  qvfxmadd $src_A = vec_{lhs}[i]$, $src_B = vec_{rhs}[i]$
    $re \leftarrow re + lhs.re * rhs.re$
    $im \leftarrow im + lhs.im * rhs.re$
  qvfxxcpnmadd $src_A = vec_{rhs}[i]$, $src_B = vec_{lhs}[i]$
    $re \leftarrow re + rhs.im * lhs.im$
    $im \leftarrow im - rhs.re * lhs.im$

This is nothing else than the repeated execution of the standard conjugated multiplication, but with carried accumulator. The first iteration has been moved out of the loop so the accumulator register does not need to be initialized with 0. The same scheme also works for every result entry in matrix-vector and matrix-matrix multiplication. For a squared vector norm, just do the same operation with $vec_{lhs} = vec_{rhs}$.

Real-Complex multiplication $c \cdot rhs$
qvfxmul $src_A = c$, $src_B = rhs$
$re \leftarrow c.re * rhs.re$
$im \leftarrow c.re * rhs.im$

For the second complex in the register the constant c must also be copied to the elements $c[2]$, for instance using **qvesplati**.

Imaginary addition $lhs + rhs * i = \begin{pmatrix} lhs.re & - & rhs.im \\ lhs.im & + & rhs.re \end{pmatrix}$
qvfxxnpmadd $acc = lhs$, $src_A = rhs$, $src_B = (?, 1)$
$re \leftarrow lhs.re - rhs.im * 1$
$im \leftarrow lhs.im + rhs.re * 1$

This uses the property of the xx-instructions to exchange the real and imaginary parts of src_A. We need one register filled with ones, or at least the second and fourth element being one because the first and fourth element are not used. This can again be done using the **qvesplati** instruction. Using the imaginary unit as constant works as well. The load or materialization can be moved out of a loop so asymptotically there is no additional cost, but permanently occupies one QPX register.

The fact that we multiply with a constant one means that effectively no operation is performed, but for the hardware it is still a floating-point operation that consumes energy and is counted by the hardware performance counters (Section 3.2.3). If there was an appropriate instruction the additional operation could be avoided, therefore we consider this a hardware deficiency. The logical flop count is different from what the hardware is actually doing.

Imaginary subtraction $lhs - rhs * i = \begin{pmatrix} lhs.re & + & rhs.re \\ lhs.im & - & rhs.im \end{pmatrix}$
qvfxxcpnmadd $acc = lhs$, $src_A = rhs$, $src_B = (?, 1)$
$re \leftarrow lhs.re + rhs.re * 1$
$im \leftarrow lhs.im - rhs.im * 1$

The only difference to imaginary addition is the negation of the product result, as done by **qvfxxcpnmadd**. The same one-filled QPX register can be reused.

4.2.1 Vectorized Hopping Matrix Kernel

All logical operations in the Lattice QCD stencil can be built from the previously mentioned operations.

After the projection the two Weyl vectors are multiplied with the same SU(3)-matrix. Therefore the two projections could be put side-by-side into one QPX register. Still, the other operations including the projection itself could not be fully vectorized, hence we prefer to process two stencils at once, as shown in the next section.

As seen in Table 2.1 (Page 30) there are 840 additions and 768 multiplications per stencil. If all multiplications are carried out with FMA-instructions, then 72 additions still need to be

executed with non-FMA instructions, i.e. at most 97.7 % of the raw floating-point performance can be used.

This is a very optimistic estimation. The most potent instructions on Blue Gene/Q are the FMA instructions that multiply two floats and add a third number to it. This is perfect for vectors-vector scalar product operations, but the Dslash stencil is more complicated. With the vectorization suggested in Section 4.2, only 30 of the 36 multiplications of in an SU(3) matrix-vector are FMA-vectorizable. For multiplication with κ (vector-complex multiplication), half of the 12 multiplications are part of FMA-instructions. In both cases because the first summand of the sum-of-products is not added to anything.

In summary, there are 240 FMA instructions per physical stencil (288 with κ-multiplication) and 180 non-FMA additions or multiplications (respectively 228) such that in total 420 instructions are required (respectively 516). If all these were FMA instructions, 3360 floating-point operations could be executed (respectively 4128 with κ). Hence, only $2 * 1320/3360 = 78.57\%$ of the theoretical floating-point performance can actually be used on this ISA. With κ-multiplication, the percentage falls to $2 * 1608/4128 = 77.91\%$.

4.3 Load-Store Vectorization

Loading and storing single floating-point values is blatantly ineffective. Every load potentially triggers a cache miss and with it a long penalty. Even if not, it takes four times as many instructions compared to those that uses the 4 values of a QPX register as a whole, shown below.

The effective address (address + offset) is always down-rounded to match the alignment of the byte length, i.e. when loading 32 bytes, the 5 least significant bits are set to zero. Since we do not intend to exploit this, we assume that effective addresses are always multiples of the byte length.

qvlfdx Quad-Vector Load Floating-point Double indeXed[1]
$dst[0] \leftarrow [\text{address} + \text{offset} + 0]$
$dst[1] \leftarrow [\text{address} + \text{offset} + 8]$
$dst[2] \leftarrow [\text{address} + \text{offset} + 16]$
$dst[3] \leftarrow [\text{address} + \text{offset} + 24]$

qvlfdux Quad-Vector Load Floating-point Double with Update indeXed
$dst[0] \leftarrow [\text{address} + \text{offset} + 0]$
$dst[1] \leftarrow [\text{address} + \text{offset} + 8]$
$dst[2] \leftarrow [\text{address} + \text{offset} + 16]$
$dst[3] \leftarrow [\text{address} + \text{offset} + 24]$
$\text{address} = \text{address} + \text{offset}$

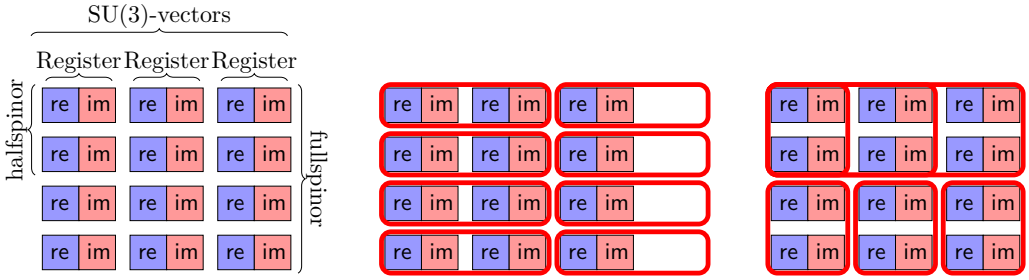
qvlfcdx Quad-Vector Load Floating-point Complex Double indeXed
$dst[0] \leftarrow [\text{address} + \text{offset} + 0]$
$dst[1] \leftarrow [\text{address} + \text{offset} + 8]$
$dst[2] \leftarrow dst[0]$
$dst[3] \leftarrow dst[1]$

qvlfcdux Quad-Vector Load Floating-point Complex Double with Update indeXed
$dst[0] \leftarrow [\text{address} + \text{offset} + 0]$
$dst[1] \leftarrow [\text{address} + \text{offset} + 8]$
$dst[2] \leftarrow dst[0]$
$dst[3] \leftarrow dst[1]$
$\text{address} = \text{address} + \text{offset}$

[1] offset register $r0$ is interpreted as 0 (non-update version)

For each of the load instruction a corresponding store instruction exists, with **qvstf** instead of **qvfl**. Single double-precision values can be loaded and stored using the standard PowerPC instructions **lfd(u)(x)** and **stfd(u)(x)** [12]. Hereby, only the first QPX element is affected. The other three are undefined or ignored. Moreover, equivalents to load/store in single precision are available that automatically convert from/to double precision in registers. The XLc compiler intrinsic `vec_ld` maps to **qvlfdx**, and `vec_ld2` to **qvlfcdux**, or their single precision versions, depending on the argument's type.

Motivated by the previous section, the real and imaginary parts of a complex value should be loaded together, they are always processed further in combination. There is no reason to load real and imaginary parts separately. So there are two complex values per QPX register loaded. The question which ones to combine remains.



(a) Load complex values individually (b) Load SU(3)-vectors vectorized (c) Strip-mine SU(3)-vector components

Figure 4.1: Choices of combining two complex values

Figure 4.1 visualizes multiple variations on how to load a single spinor. In Figure 4.1a all complex values are loaded separately using **qvlfcdx**. 12 such instructions are required. With continuing processing without recombination, one of the two complexes per QPX register is wasted.

The variant of Figure 4.1b has the problem that the second load reads unused data, but its advantage is that by the algorithm's definition the same operations are applied on all three complexes of a SU(3)-vector, allowing optimal use of vectorized arithmetic instruction at least for the first register, but 1/4 of memory and flops is wasted.

In Figure 4.1c, same components of different vectors are combined into one register. Direct processing is not possible because with the application of the gamma matrices (Equation (2.4)) the four spinor components are treated differently.

Mixing the choices is possible, but still will not yield a good solution. Also, rearranging the data in the registers after loading them is no solution; they could also have been loaded that way.

Note that Figure 4.1 is not as bad as it may seem at first sight. The result of the halfspinor projection are two SU(3) vectors, on which the same operations are applied on. Therefore the two vectors can be put in the lower and upper part of a QPX register. It is merely only the projection to (and the expansion of) the halfspinors where they are treated differently and are less efficient. Variant 4.1c followed by appropriate rearrangement instructions might even be a bit more efficient. The authors of tmLQCD chose this variant for their Blue Gene/Q optimization. No data layout changes are required which limits the amount of code requiring a rewrite.

If we value performance over ease of implementation, the only way is to combine the complex values of two different spinors to the same site. We define a physical site as an element of an array that stores the (spinor-)field. An array element must have a fixed byte length. A logical site always consists of a single spinor at a (T,X,Y,Z)-coordinate. For a physical site, we

interleave two spinors . There are also multiple choices for which two logical sites to merge, which is discussed in Section 4.4.

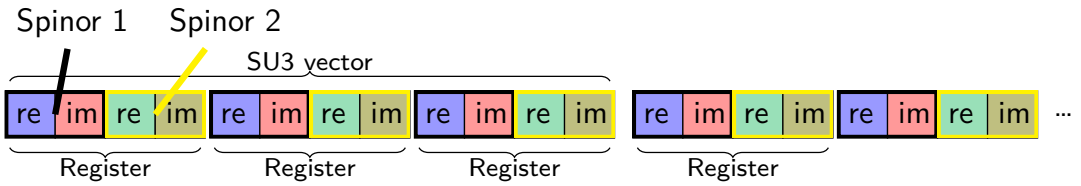


Figure 4.2: Data ordering of physical site

Shown in Figure 4.2, a physical site stores corresponding complex values of two different values in a single QPX register. Every stencil applies the same operations, so applying an arithmetic instruction on the register applies it to two logical spinors at the same time. This is a classic technique when compilers vectorize, except that we also sites to physical sites to vectorize memory accesses as well.

4.4 Lattice Linearization and Iteration Order

When stored in memory, the 4-dimensional lattice space must somehow be mapped to logical address space. Allocated memory is always returned in a consecutive block in address space and therefore 1-dimensional (linear).

The *C* programming language linearizes fixed-size arrays using the row-major rule. That is, elements that differ only in the rightmost coordinate are placed consecutively, in order of that coordinate. Such fixed-length memory blocks again build up an array without the rightmost coordinate but larger element size. These can again be placed consecutively according to the now rightmost coordinated. This repeats until no more coordinates are left.

For the sake of performance the linearization of an array and the order in which its elements are processed are strongly connected. Performance is best if elements are iterated over in the same order as they are stored in memory[1]. For non-elementwise (more than one element accessed per iteration) operations this is difficult to do because the elements are probably not required in the same order. A compromise is required.

In this section two alternative layouts are presented. One optimized for the case that memory bandwidth is the bottleneck and another if the latency is the limiting factor. As Hopping Matrix (Section 2.4) is the hotspot kernel, we specifically optimize for it. All other operations are element-wise where the linearization is less important.

4.4.1 Fullspinor Layout

The fullspinor layout has its name because it always stores the complete spinor at one memory location. Actually, due to the vectorization presented in the previous section, two spinors are merged into one array element; two logical sites to one physical one.

This layout tries to optimize reuse of data already in a cache to avoid fetching it from later level caches or main memory. On most platforms predictability of the memory access pattern is more important than data reuse so the hardware can predict what data is required in advance. On Blue Gene/Q we hope that the “perfect” list prefetcher eliminates such concerns.

It does not use index lookup arrays which also consume cache space. Instead site locations are computed using integer arithmetic. Because Hopping Matrix is dominated by floating-point

[1]This needs clarification for GPUs. To allow coalescing it is best to access memory in strides of the work group (also called warp or wavefront) size. However, when handled as a SIMD machine as in Section 4.3 the same rules are applicable again.

arithmetic, such integer operations can be computed without slowdown on the XU that would have been idle otherwise (cf. Section 3.2.1 on Page 34).

The addressing of a physical site is illustrated in Figure 4.3. Logical sites have four coordinates: t, x, y and z. The spatial dimensions are mapped directly to dimensions of a multi-dimensional array. Only the Z-coordinate is shown in the figure.

A physical coordinate of a logical site is split into multiple components: tv ("t-vector"), eo ("even-odd") and k. Every physical site is indexed using an x, y, z and a tv coordinate and contains four logical sites, respectively spinors. Of these four spinors there are two with odd coordinates and two with even (logical) coordinates[1]. Even coordinates are displayed as circles and odd coordinates as diamonds in Figure 4.3. The even sites go under the coordinate $eo = 0$ while the odd ones are stored with the coordinate $eo = 1$. Since both parities contain two elements they can be stored vectorized (Section 4.3), which we distinguish using the coordinate k. For fullspinor layout we chose to vectorize *neighboring* sites into a physical site to exploit spatial locality of the Hopping Matrix stencil.

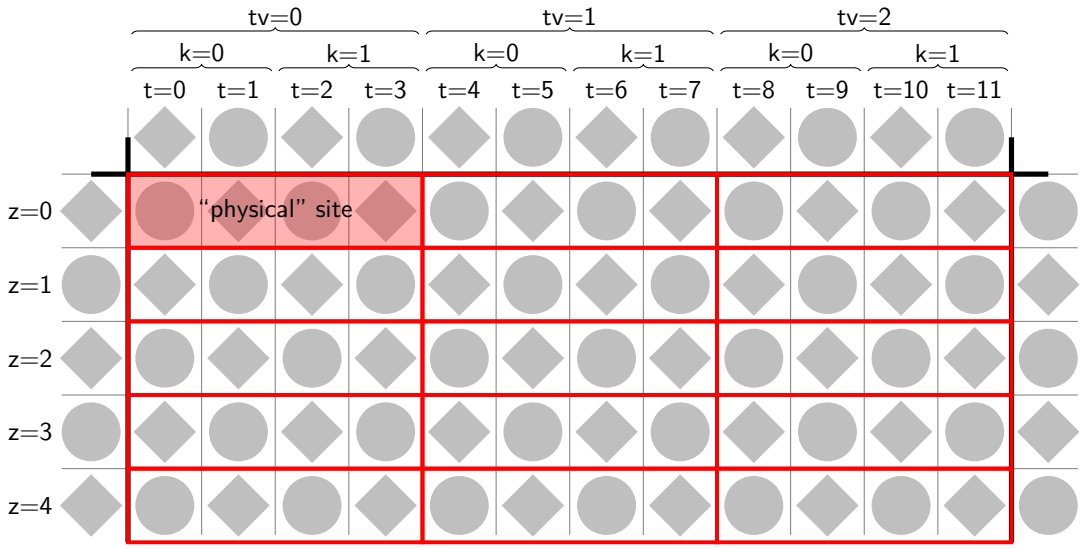


Figure 4.3: Coordinates of physical sites

The 8-point stencil also has the property that to compute an even site, only odd sites are used as input, and vice versa. The working set of the complete field can therefore be split into two sets that can be computed independently. Because of the even/odd preconditioning from Section 4.1 this separation is important for performance. The result is illustrated in Figure 4.4.

Effectively, the physical storage of a spinor field is a multi-dimensional array declared as

```
typedef vector4double fullspinorlayout[2][LT/4][LX][LY][LZ][4][3];
```

To get the spinor from the logical site at coordinate (`t`,`x`,`y`,`z`), one uses

```
field[eo][tv][x][y][z][v][c][2*k+p];
```

with `v` the index of the SU(3)-vector, `c` the vector's element index, $p = 0$ for the real part or

[1] For even/odd preconditioning, also see Section 4.1 on Page 43

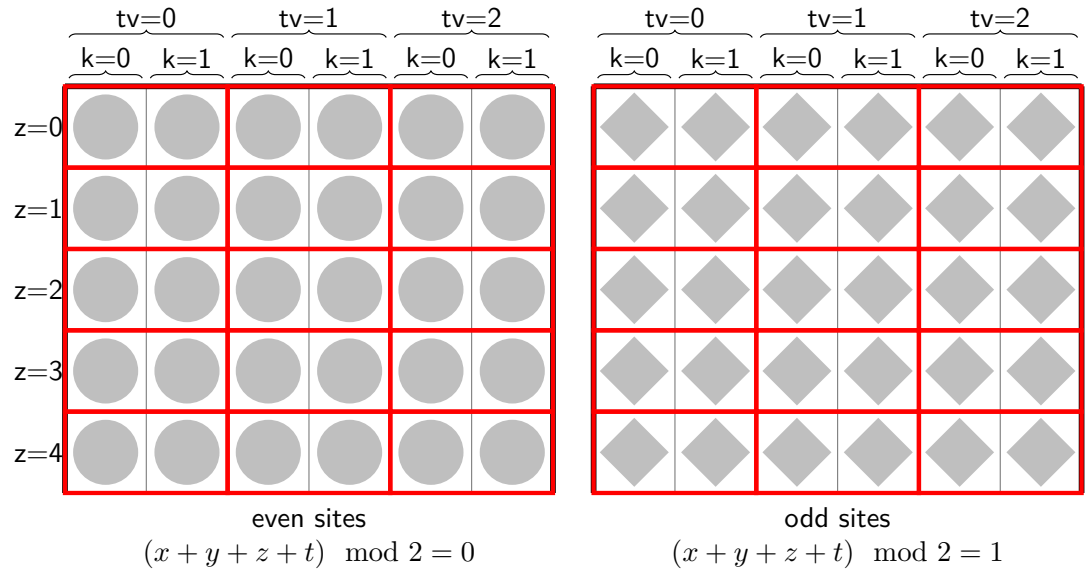


Figure 4.4: Storing sites with even and odd coordinates in different arrays

$p = 1$ for the imaginary part, and

$$\begin{aligned} eo &= (x+y+z+t) \mod 2 \\ tv &= \left\lfloor \frac{t}{4} \right\rfloor \\ k &= \left\lfloor \frac{t}{2} \right\rfloor \mod 2. \end{aligned}$$

The layout is row-major, i.e. the byte position in memory can be computed using the formula

$$8\left(p+16\left(k+2\left(c+3\left(v+4\left(z+L_Z\left(y+L_Y\left(z+L_Y\left(tv+\left\lfloor\frac{L_T}{4}\right\rfloor eo\right)\right)\right)\right)\right)\right)\right)\right). \tag{4.1}$$

This storage order leads to the natural iteration order

```
for (eo = 0; eo < 2; ++eo)
  for (tv = 0; tv < LT/4; ++tv)
    for (x = 0; x < LX; ++x)
      for (y = 0; y < LY; ++y)
        for (z = 0; z < LZ; ++z)
          // Process the 2 spinors at the physical site
```

where the parities are handled on the algorithmic level, i.e. all operations are performed on the even sites first, then on the odd sites.

4.4.1.1 Hopping Matrix

Physical sites where the first of the four sites is an even site access neighbor sites differently than those with an odd first logical site. For instance, in Figure 4.5, the physical site at $z = 0$, $tv = 0$ has an even leftmost logical site. This means that when computing the stencil at positions $z = 0$, $t = 1$ and $t = 3$ in a vectorized way, the neighbors in direction -T are the even

elements at the same physical site and therefore both can be loaded using vectorized loads. Unfortunately, the neighbor in direction +T has to be merged from elements of the physical sites $tv = 0$ and $tv = 1$. Conversely, the odd stencil for position $z = 1$, $tv = 0$ can directly use the 2 vectorized spinors from direction +T, but needs to access two physical sites in direction -T. This does not mean there is an additional memory access because T+ and T- both reuse the middle site.

This is why the implementation combines both cases in a single iteration such that 4 stencils are computed per iteration. The alternative would be some if-conditions in the kernel, for which high-performance computers – including Blue Gene/Q – are most often not optimized for. The kernel loop therefore is

```
for (eo = 0; eo < 2; ++eo)
  for (x = 0; x < LX; ++x)
    for (y = 0; y < LY; ++y)
      for (tv = 0; tv < LT/4; ++tv)
        for (z = 0; z < Z; z += 2)
          // Process the 4 spinors (iteration body)
```

This iteration order is meant to improve the reuse of data in the cache, illustrated in Figure 4.5. While traversing with increasing z-coordinate, elements are reused up to 4 times (e.g. $z = 1$, $t = 1$). Ideally, the data is kept in registers from the first use to the last use. This spans 3 physical sites which unfortunately exhausts the available 32 QPX registers. But these should still be held in the L1 cache so they can be loaded without penalty cycles.

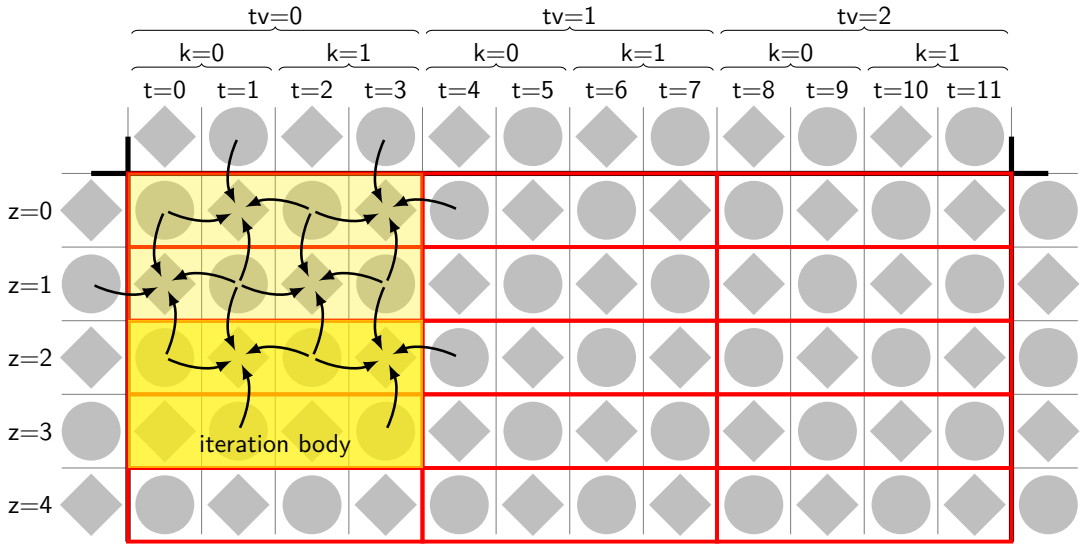


Figure 4.5: Element reuse while iterating through the field

4.4.1.2 Threading

While data in Z-direction is reused well as presented, the other stencil points are not reused at all. One can do tiling to try to keep the working set limited to the size of the tile. Unfortunately, the L1 cache is not big enough to hold a significant number of spinors until reuse and the L2 cache is likely big enough to store the complete field. Additionally, the lengths of the lattice per node is often not big enough to do tiling at all (In the illustration $L_T = 12$ with only three physical sites side-by-side).

Instead, we use wavefronting. Multiple threads work on the same lattice. Data used by one thread is loaded into the cache. Another thread processing the elements in the neighborhood

may then use the data directly from the cache. The frontmost thread in the group is slowest and the other threads may catch up such that all the threads keep in about the same distance.

The illustration in Figure 4.6 shows three threads sharing data at their borders. Other threads in neighboring X and Y-dimensions share even twice as much data because the common border between physical sites is larger.

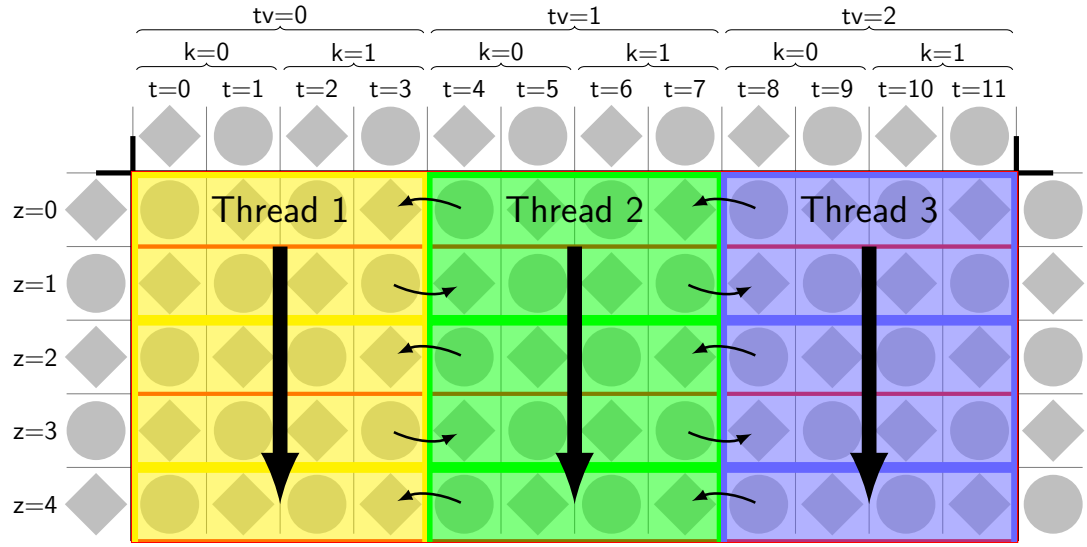


Figure 4.6: Wavefronting with multiple threads

For Blue Gene/Q, this data sharing works best withing the L1/L1p caches, and therefore between the 4 threads on the same core. Other cores cannot access their L1 cache. Sharing at L2 may benefit if the working set is larger than the L2 cache.

Here, we chose to vectorize and to collapse parity in the time-dimension. One argument to support this is that in typical Lattice QCD simulation the time dimension is often twice the size of the spatial dimensions. Collapse and vectorization may even happen on two different dimensions, which results in a 2x2 (x1x1) pattern for a physical site. Unfortunately this means that at four stencil points where the two spinors that must be loaded from two different physical sites each. The Tetris-shaped alternative in Figure 4.7 has just two such irregular points (depending on the fill pattern, up-down or left-right), but is also more difficult to implement, especially for surface sites.

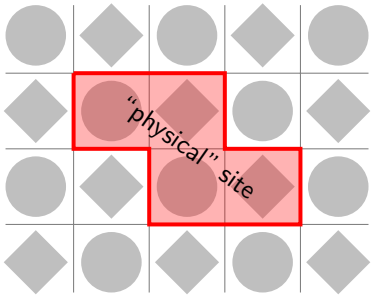


Figure 4.7: Alternative shape of physical sites

4.4.1.3 Inter-node Transfers

Not all spinors are stored on the node where they are needed. Stencils on the inner rim of the section stored locally need data from the neighboring nodes. We will use the terms from Figure 4.8 to classify different sites: The *body* consists of sites that do require remote memory to compute the stencil at its position. A stencil on the *surface* has at least one neighbor that is not stored on the same node. The *halo* sites are the spinors on remote nodes that are required to compute all the stencils on the local node.

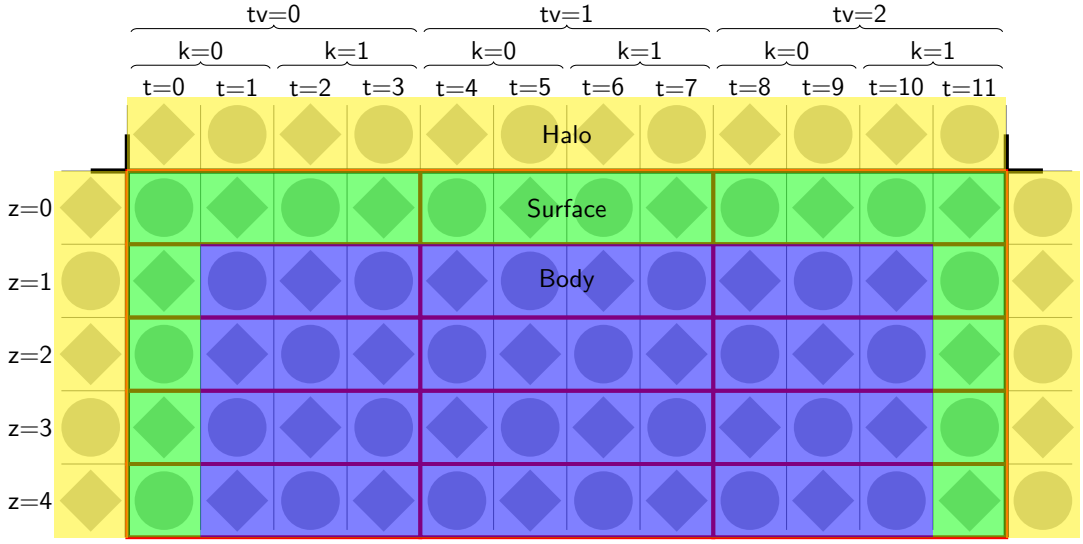


Figure 4.8: Body, surface and halo

When applied to physical sites, body and surface of the local lattice look odd (see Figure 4.9). By definition, surface sites are those that require data from other nodes to compute the stencil at this position. Even and odd sites are computed separately which means that only every second site on the logical surface is computed for a specific value of *eo*. In addition, the values are stored in a vector for which only one stencil may need data from a remote node. Thanks to vectorization both stencils are computed in the same operation. It also means that both need to wait for the data required by one of them.

Before the stencil's application the foreign node's surface data (the halo from the local node's viewpoint) must be put into a special receive-buffer. MPI is told to write the received the data into that buffer. In order to be used the received data has either to be copied into the array to be directly indexable using Equation (4.1), or the stencil operation must redirect its data source directly to the receive buffer. The fullspinor layout uses the latter option to avoid one unnecessary copy. Due to asynchronous transfers the surface sites need to be processed separately anyway.

Asynchronous MPI and MUSPI is exploited by computing the body stencils while that data from the halo is being transferred. This splits up the Hopping Matrix into five operations.

1. Issue the commands to receive the halo into their buffers (`MPI_Irecv`).

2. Copy the halo required by the neighborhood to the send-buffers of each direction. Before writing the data to the send buffer, the spinors are projected to halfspinors. This reduces the amount of data to be transferred by half and otherwise had to be done on the receiving node anyway.

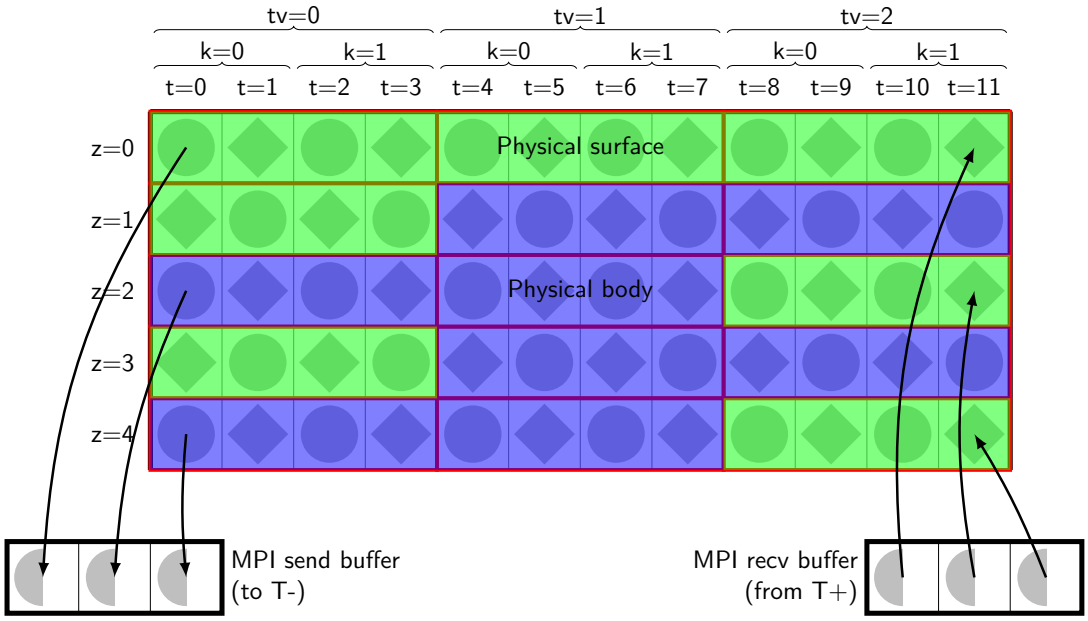


Figure 4.9: Physical body and surface (for odd sites, the diamonds)

3. Issue the command that the hardware can now send the data to the neighboring nodes (`MPI_Isend`).
4. Compute the body stencils
5. Wait for the receive operations to finish (`MPI_Wait`)
6. Compute the surface stencils using the data from the receive-buffers.

It is possible to call `MPI_Wait` separately for each neighbor such that stencils are computed for the first surface data arrives for. It requires some detection for which buffer is complete first. In this implementation stencils that require data from just one remote node have their own `MPI_Wait` but in a fixed order.

4.4.1.4 Gauge Field

The ordering of the gauge field elements directly depends on the order of the spinor field. A site in the gauge field consists of four SU(3) matrices, one for each dimension in up-direction. For the down-direction the matrices can be reused by inverting them. The SU(3) matrices shall be stored the same order as the corresponding elements in the spinor fields.

The exception is the direction of the gauge link. When applying stencils, not all the SU(3) matrices of a site are used. For the down-directions, only one is required per stencil point. To make the data required consist of streams, the directions are stored separately.

Therefore, the declaration of a gauge field is the equivalent of

```
typedef vector4double gaugefield[4][2][LT/4][LX][LY][LZ][3][3];
```

where a link between the coordinates (t, x, y, z) and $(t, x, y, z) + \mu_d$ can be retrieved using the expression

```
gaugefield[d][eo][tv][x][y][z][j][i][2*k + p]
```

where `d`= μ, `j` is the matrices' row and `i` its column. The rest is defined as for the spinor field.

4.4.2 Halfspinor Layout

The performance of the previous layout on large subvolumes (Chapter 5) is disappointing. The analysis shows that the memory access latency is the problem, i.e. neither of the prefetchers is able to prefetch the data required. Therefore, we implement another layout that is optimized for the Blue Gene/Q stream prefetcher, not data reuse. That is, that data is laid out in memory such that Hopping Matrix reads it strictly consecutively.

Every stencil has 8 input sites which are stored in a block. Every site is required by 8 stencils as input and therefore must be copied into all of these blocks. Only the projections are required for computing the stencil result from these 8 sites such that only halfspinors need to be stored there. The total memory required is therefore 4 times the size required by the fullspinor layout, i.e. $8 * 2 * 3 * 2 * 8 * 2 = 1536$ bytes per vectorized pair of halfspinors in double-precision.

The layout is illustrated in Figure 4.10. Every physical site contains 8 halfspinors, from which a fullspinor can be computed. Again, the halfspinors are vectorized such that two logical sets of 8 halfspinors are stored in a physical site. In contrast to the fullspinor layout, not the neighboring sites are vectorized together, but the local volume is split in halves of which the vector contains one element each.

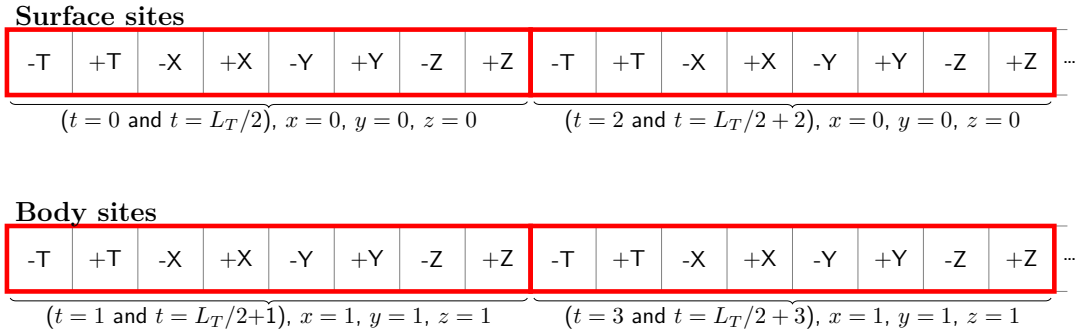


Figure 4.10: Halfspinor layout for even sites

For the same reasons as for the fullspinor layout, the even and odd sites are stored separately. In addition, the physical surface sites are stores separately from the sites of the body. The intend is that the surface sites can be computed first and then sent to the neighboring nodes in parallel to the computation of the body stencils. The order within the surface- or body list is not important.

In summary, the definition of a spinor field in halfspinor layout is the equivalent of

```
typedef struct {
  vector4double surface[N_SURFACE_SITES][8][2][3];
  vector4double body[N_BODY_SITES][8][2][3];
} halfspinorlayout[2];
```

which is naturally iterated over by

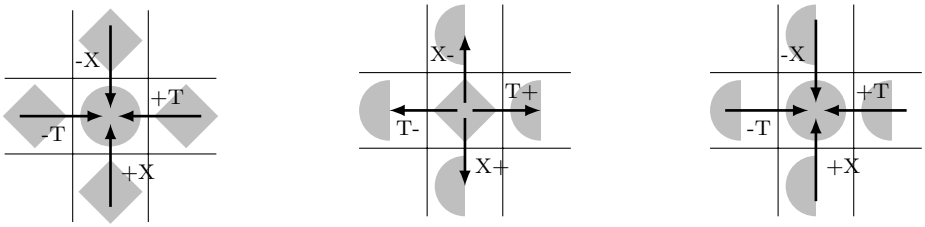
```
for (eo = 0; eo < 2; ++eo)
  for (i = 0; i < N_SURFACE_SITES; ++i)
    // Process the 2 spinors at field[eo].surface[i]
  for (i = 0; i < N_BODY_SITES; ++i)
    // Process the 2 spinors at field[eo].body[i]
```

4.4.2.1 Hopping Matrix

With the changed layout the Dirac stencil has a different structure. For each of the 8 input points, the fullspinor stencil (Figure 4.11a) reads the spinors, projects them to halfspinors,

applies the SU(3) matrices, expands them again and finally accumulates the 8 results to the stencil's result. The operation will be repeated multiple times.

Using the halfspinor layout, the stencil directly uses the projected spinors. That is, it first computes the 8 projections of a single spinor in what looks like an inverted stencil (Figure 4.11b) as preparation for the stencil itself. The halfspinors are later picked up by another stencil to compute the result of the stencil (Figure 4.11c).



(a) Standard stencil: Compute the new value of a site using the neighbor sites.

(b) Inverted stencil: Compute all projection halfspinors at a site.

(c) Halfspinor stencil: Compute the new value of a site using the neighbor halfspinors.

Figure 4.11: Stencil and inverse stencils

Writing halfspinors involves no latency, hence these can be written to any location without performance penalty. That is why we store them in the order they are read in the next operation, such that the 8 reads of Figure 4.11b are one block of memory (Figure 4.9).

The next operation might not be the Dirac operator, but an elementwise operations like a scalar-vector multiplication. In this case, not the halfspinors, but the fullspinor is required. For this reason the order of operations is different: Given a fullspinor as input, compute the projections for all neighbor stencils that use this spinor as input (Figure 4.11c). This includes the SU(3)-matrix multiplication. Then store the halfspinors such that the inputs of the forward stencil are stored in one block. Whenever the fullspinor is required, including another Hopping Matrix, the halfspinors are expanded and summed over on-the-fly.

The halfspinor layout therefore is the result of a Hopping Matrix operation, an inverted stencil. It is split into two phases: The invocation of the operation and the reconstruction of the fullspinor when it is used. Both phases are sketched in Listings 4.1 and 4.2.

```
/* 1  */ for (i = 0; i < N_SURFACE_SITES; ++i)
           // Read surface spinor from input field
           for (d = 0; d < 8; ++d)
             // Project spinor into direction d
             // Read SU(3) matrix from gauge field
             // Multiply weyl spinor with SU(3) matrix
             *target_ptr[eo].surface[i][d] = /* halfspinor */;
/* 1b */ // Copy to send buffers
/* 2  */ MPI_Start(send);
/* 3  */ for (i = 0; i < N_BODY_SITES; ++i)
           // Same as for surface sites
```

Listing 4.1: First phase of Hopping Matrix: Multiplication and send

The first step is to compute the data that is to be transferred to the neighborhood nodes. It would be enough to project only those directions that are on the neighbor node, but then the surface sites need to be iterated over again. The halfspinors are multiplied by SU(3) matrices here already so it does not need to be done on the receiving node anymore. This pattern is the inverted stencil (see Figure 4.11b), it does not compute the result of a stencil operation,

but reads a spinor exactly once and puts the required data at each of the 8 stencil its data is needed.

The array `target_ptr` contains the destination address for each of the halfspinors. This can either be a location in the target field – at the exact location it is expected in the second phase – or alternatively an address into the send buffer for the neighbor node in that direction.

The step 1b is optional since most of the data has been written into the buffer by the previous step. This is different for the directions -T and +T: Only half of the values in a `vector4double` need to be actually transferred to the other node ($t = 0$ for direction -T and $t = L_T - 1$ into direction +T), the other is required locally. One can ignore this and transfer both which is twice as much data to transfer as necessary. Alternatively, step 1b iterates over all elements of the buffer and splits the vectorized halfspinors. One half is stored into the real send buffer (the one they have been written to is just temporary for this split) and the other half gets stored at its location in the target buffer on the same node.

The transfer is invoked in the second step. Although in the listing MPI is mentioned explicitly, MUSPI is possible as well.

In step 3 all the remaining inverted stencils are computed and written to the target field. This is the field's body and therefore all the halfspinors are local to the node. This happens in parallel to the transfer that stated in step 2.

Now the program can continue while the data transfer is possibly still in progress. The SPMD program can continue with any operation that neither does MPI (or MUSPI) point-to-point communication nor requires the target's field data. As soon as the target spinor field is accessed, phase two in Listing 4.2 begins.

```
        if (/* transfer pending */) {
/* 4  */   MPI_Waitall(recv);
/* 5  */   // Copy back from receive-buffers to field
        }
/* 6  */ for (i = 0; i < N_SURFACE_SITES+N_BODY_SITES; ++i)
          // Do the follow-up operation
```

Listing 4.2: Second phase of Hopping Matrix: Receiving and reconstruction

At first it checks that the second phase has not run yet. If it did the following two steps can be skipped as they already have been done.

The first step in the second phase, or the fourth of Hopping Matrix, is to ensure that all the data has been received and written to the receive-buffers. If not, wait for them since because the program cannot continue without them.

Then, in the fifth step, the data from the receive buffers is copied over to the target buffer. For the data that resides on the same node this already happened in step one. If step 1b has been skipped the data from the receive buffer from -T is merged with the data from the send buffer to -T. This is because one vector element in the target field comes from the local node (now in the send buffer), the other from the remote neighbor in direction +T. Conversely, data in the receive buffer from +T is merged with the send buffer to +T. Thereafter the buffers are free for reuse in another operation.

In the sixth and last step the data is available as in Figure 4.10. When the follow-up operation iterates over all sites it actually does it in the order defined in this layout. For each element the spinor is reconstructed by accumulation using the 8 halfspinors on-the-fly and therefore with contiguous memory access. The follow-up operation can again be another Hopping Matrix.

4.4.2.2 Threading

Threading is relatively simple in this layout. The loops in steps 1, 3 and 6 can be split up into equally sized parts per thread. Similarly, this is also possible for the copy operations in steps

1b and 5. Locality is not a concern is this layout, so the placement of workload is irrelevant as long as it is distributed evenly among the cores and their threads.

4.4.2.3 Gauge Field

The gauge field is also optimized for linear access. The SU(3) matrices are arranged such that the steps 1 and 3 in Listing 4.1 also access them consecutively. For every physical site there are therefore 8 SU(3) matrices, one for each halfspinor in the inversed stencil. In comparison to the fullspinor layout this means twice as much memory are used since both directions are stored explicitly.

In summary, a declaration of the gauge field in C looks like:

```
typedef struct {
  vector4double surface[N_SURFACE_SITES][8][3][3];
  vector4double body[N_BODY_SITES][8][3][3];
} gaugefield_t[EO];
```

4.5 Precision

In the previous sections we assumed that floating-point values are stored in double precision (`vector4double`) as it is the only precision the AXU supports. But there are also instructions that can load and store `vector4double` registers in single IEEE precision. This may speed-up any operation when bandwidth is the bottleneck. Also, the caches hold effectively twice as many single precision values than double precision values.

The source code has alternative implementations for any algorithm in single precision by recompiling the same source files with different preprocessor macros. The intention is to measure the speed-up.

If the single precision solver is significantly faster, one can exploit the speed-up but keep the effective precision by using multi-precision solvers. Such a solver would start in a lower precision mode to get an estimation of the solution. At some point, when the solution encounters the lower precision's limits, it switches to higher precision ad continues. The total number of solver iterations may be higher but is compensated by the faster low-precision iterations. This has not been implemented in this thesis.

4.6 Layout Selection at Runtime

There are implementation of 5 different layouts for spinor fields. There is the fullspinor layout as presented in Section 4.4.1 and the halfspinor layout presented in Section 4.4.2. Both of them have a variant in single precision. In addition, the previously existing layout in tmLQCD remains intact, which will be called the legacy layout. This is to avoid needing to change parts of the code that are not relevant for the performance, like reading the input files containing gauge fields, done once only per program execution.

Which layout is used is decided at runtime depending on which operation has been executed previously. The functions for loading and storing a gauge field from disk are not modified and therefore can only handle the legacy layout and must be converted to/from some other layout. This does not happen very often and therefore no further optimization is required.

Elementwise and reductive operations may use input fields in any layout. These read spinors consecutively in the order given to them. The spinor field output of elementwise operations is always in fullspinor layout of the selected precision.

The Hopping Matrix for the fullspinor layout also returns the same layout as its input. But if using the other Hopping Matrix version, then the output will be in halfspinor layout. Any operation has the capability added to read halfspinors by using a "filter" that reads in a

physical halfspinor site and passes the equivalent vectorized spinor to the follow-up operation. That is, the second phase of the halfspinor Hopping Matrix is done on the fly.

This works best if the direct output of Hopping Matrix is read exactly once. If used multiple times, converting it first to fullspinor layout may reduce the overhead of on-the-fly conversion done twice. A spinor field does not "know" how often it will be used before being overwritten without some hint by the programmer. Hence, this is not done automatically. The programmers themselves have to invoke layout conversion in this case.

There is a combinatorial explosion if operations with multiple input fields must accept all combinations of layouts. Therefore only common combinations (e.g. all inputs fields in the same layout) are supported on-the-fly. in other cases the fields' layouts are converted into a common format.

4.7 Cache Management

The different cache levels need to be handled very differently due to the granularity of data and the availability of different mechanisms per level.

Element	Precision	Size in bytes	Load instructions	Cache lines	
				L1	L1p/L2
Vectorized fullspinor	Double	384	12	6	3
	Single	192	12	3	1.5
Vectorized halfspinor	Double	192	6	3	1.5
	Single	96	6	1.5	0.75
Vectorized SU(3) matrix	Double	288	9	4.5	2.25
	Single	144	9	2.25	1.128
Fullspinor stencil	Double	3072	96	48	24
	Single	1536	96	24	12
Halfspinor stencil	Double	1536	48	24	12
	Single	768	48	12	6
SU(3) matrices per stencil	Double	2304	72	36	18
	Single	1152	72	18	9
Total per fullspinor stencil	Double	5376	168	88	48
	Single	2688	168	48	32
Total per halfspinor stencil	Double	3840	120	60	30
	Single	1920	120	30	15

Table 4.1: Cache lines occupied by elements

The amount of data required per Dirac stencil is shown in Table 4.1. If using 4 threads per core, there are 4096 bytes available per threads in the L1d. In case of a stencil using the fullspinor layout, there is not enough space in the L1 for even one stencil operation in double precision. In addition, the data may not be perfectly distributed between the 256 cache lines because they are associative only in groups of 8 lines. The 32 groups are assigned according the bits 53 to 57 of the address. So effectively, the L1 may appear smaller than it is. We deduce that one cannot load a complete spinor in advance, but has to be loaded while the stencil is in progress.

4.7.0.4 Alignment

To use the vectorized memory access instructions effectively, all array element must be aligned to 32-byte boundaries, respectively 16 byte boundaries for the single precision versions. Fortunately, all the elements are a multiple of this such that if the first element is aligned, no padding between elements is required.

4.7.1 Prefetching to L2

This is a short section because the implementation does not specifically prefetch to L2. We just assume that the working set fits completely into the L2 cache. Otherwise, the DDR-to-L2 bandwidth becomes the bottleneck and prefetching does not reduce the amount of data to be transferred. That is, the DDR-to-L2 is always busy to fetch data such that some core can continue working with no idle time in which we could preload some data.

The halfspinor layout doubles the total amount of data, therefore the working set less likely fits into the L2. The bandwidth also becomes saturated much easier. Halfspinor layout should not be used in this case.

The fullspinor layout becomes the preferable strategy. Tiling is the technique of choice to reduce the bandwidth. This implementation, however, used wavefronting to improve reuse on the L1 cache. Implementing tiling instead could be the better choice for large working sets. Tiling also makes the stream prefetcher (Section 4.7.2.1 below) less effective.

4.7.2 Prefetching to L1p

The L1p cache can be filled by the hardware stream- or list prefetcher only. The two alternatives are presented below.

4.7.2.1 Stream Prefetching

In stream prefetching mode, the hardware preloads consecutive memory addresses. The A2 is able to follow up to 16 streams per core, i.e. any thread can use up to 4 streams. Not exceeding the 4 streams is essential. Establishing a fifth prefetch line will stop a different stream. When the thread continues accessing the lost stream, another stream will need to stop again. This effect is a special kind of thrashing and renders the prefetcher ineffective. Even accessing local variables on the stack may establish new streams and therefore significantly performance. Hence, spilling (not being able to keep all local variables in registers) should be avoided as well.

There are a lot more than 4 streams used by Hopping Matrix in fullspinor layout. There are 7 alone for reading the spinors (the thread moves into Z-direction, therefore the -Z spinor stream reuses the +Z spinor stream) and additional 7 for reading the gauge field. Hence, the stream prefetcher can be used efficiently only with one thread per core, i.e. without SMT .

The halfspinor layout has been designed specifically with the stream prefetcher in mind. The spinor and gauge fields are both rearranged for consecutive access, both requiring one stream. A third is required to load the indirect access pointers of `target_ptr`. The forth stream remains unused; the processor may occasionally use it when the function's stack frame is accessed instead of deactivating one of the required streams.

Writing the stencil results doesn't require a stream because of the Blue Gene/Q's write-through design.

4.7.2.2 List Prefetching

The Blue Gene/Q's list prefetching capability has been described in Section 3.3.5.2 on Page 39. The fullspinor layout has been designed with list prefetching in mind to resolve all prefetching

issues.

Unfortunately, it is not very useful for Lattice QCD due to its structure and the prefetcher's overhead. The list of accesses is only useful in a loop body that accesses the same elements again and again in the same order. This is not the case for the Hopping Matrix loop where in every iteration different elements are accessed. However, the condition is fulfilled for solver iterations.

The culprit seems to be that there are too many accesses in a solver's loop. In addition to the call to Hopping Matrix, many elementwise operation are performed (Algorithm 2.1). Only in the next solver iteration the access pattern begins again and can be reused.

During the operations scalar multiplication, field addition, assignment and the like it is better to temporarily disable the list prefetcher. The operations can be implemented with strictly contiguous access pattern per field and therefore the stream prefetcher already does a perfect job here.

Experiments show that with list prefetching enabled the Hopping Matrix runs slightly slower than with stream prefetcher. Without internal knowledge of the hardware it hard to confirm any assumption on the reason. The prefetcher adds additional overhead, e.g. by reading even more data for the memory addresses it has to prefetch. It may also not be able to keep up with the frequency new data is needed. In addition, a spinor or SU(3) matrix already spans multiple consecutive cache lines and in the innermost loop accesses are consecutive such that the stream prefetcher can already prefetch some data in advance, reducing the potential benefit of the list prefetcher.

Experiments with a less complex loop (non-vectorized matrix-matrix multiplication) show that list prefetching improves performance only for a limited range of working set sizes. The fact that one iteration reads just two values from different locations instead of multiple consecutive pages makes this loop more sensitive to memory latency.

This unfortunately undermines the principle the halfspinor layout was designed for and so the halfspinor layout was developed. The halfspinor layout is optimized for consecutive read access and therefore list prefetching cannot improve anything.

4.7.3 Prefetching to L1

Per thread, there are 64 cache lines available (4 KB in total). For computing one stencil, one needs to read 8 spinors 192 bytes each and 8 SU(3) matrices from the gauge field with 144 bytes each, therefore 2688 bytes (42 cache lines). This means that not even two complete sets of stencil source data fits into the cache. With vectorization not even a complete stencil working set fits into the L1 cache[1]. This makes tiling useless at this level.

Prefetching into the L1 is done via the **dcbt** instruction. If we assume that the data resides in the L1p cache, **dcbt** should be inserted about 24 cycles (the L1p hit latency) before the first instruction using the data. If using more than one thread per core a shorter distance may be chosen as the other threads result in a delay of the instruction issue. The 24 cycles are hard to match exactly as the compiler can move instruction into and out from the gap between the two instructions. It is even hard to do when writing assembler directly because the other SMT thread's activity is unknown.

More practically, the **dcbt** is put one stencil point before, which is more than 24 cycles of distance. It is illustrated by Algorithm 4.1. No gain is observed by slightly moving the **dcbt** instructions closer to the data's use or increasing the gap between them. However, removing them or put all of them before the stencil does hurt performance.

[1] See also Table 4.1

Algorithm 4.1: Explicit prefetch

dcbt -T;
Compute +T;
dcbt -X;
Compute -T;
...
dcbt +T;
Compute -Z;

4.7.3.1 Alignment

Unfortunately some elements may not be aligned to the 64 byte L1 cache lines. This is the case for halfspinors in single precision and SU(3) matrices. As a result, some elements begin in the middle of a L1 cache lines. This is no problem for the **dcbt** instruction which just prefetches the complete cache lines. However, the cache line may already been prefetched for the previous element, i.e. the same cache line is prefetched twice. The implementation just accepts that there is one superfluous instruction instead of special-casing it.

4.8 Assembly-Level Optimizations

In the previous sections, optimization was done on a level that can be coded using a programming language and then transformed to machine code by the compiler. But if the compiler does not generate the intended machine code, the programmer has to go to a lower level. Sometimes the compiler accepts hints on what to do, like pragmas or attributes. In other cases programmers have to supply the machine code intended themselves.

This section specifically deals with sub-optimal code generated by the IBM's XLc.

4.8.1 qvfldux and qvstdux

When a 32 byte register is filled with data from memory – either by dereferencing a pointer of type `vector4double` or by using the `vec_ld` compiler intrinsic – XLc generates the assembler instruction **qvlfdx**. If loading a consecutive array, the generated code looks like this (assuming the initial address is stored in `r1`):

```
qvlfdx q0, 0, r1
li r2, 32
qvlfdx q0, r2, r1
li r3, 64
qvlfdx q0, r3, r1
li r4, 96
qvlfdx q0, r4, r1
li r5, 128
qvlfdx q0, r5, r1
...
```

That is, it will load a constant for every offset from the base pointer. If inside a loop and there are enough general purpose registers available, the `li` instructions can be moved outside the loop. Otherwise, for instance with the Hopping Matrix stencil, the constant have to be rematerialized one or multiple times within the loop.

The Blue Gene/Q instruction set has a special command for load and store sequences with constant stride: **qvlfdux** (respectively **qvstfdux**). It increments the address such that issuing another load will read the next bytes:

address ← ...
offset ← 32
qvlfdux
$vec_{dst}[0][0..3] \leftarrow [address + offset]$
address = address + offset
qvlfdux
$vec_{dst}[1][0..3] \leftarrow [address + offset]$
address = address + offset
...

Unfortunately XLc does not use this instruction except once per sequence[1], nor there is an intrinsic that always generates **qvlfdux**. Therefore, for optimal speed, we program this sequence manually in with inline assembly. The same trick works analogously when storing data using the **qvstfdux** instruction.

4.8.2 dcbt

The same phenomenon occurs when prefetching a consecutive memory area using the **dcbt** instruction.

dcbt offset,address
Load L1 cache line with block at address + offset without blocking[2] [12]

The XLc-generated code of a **dcbt** instruction sequence is shown below. A **dcbt** loads a 64 byte cache line, i.e. there are 6 instructions required for reading two interleaved double-precision fullspinors, 3 for halfspinors and 5 for the SU(3) matrices in the gauge field.

```
dcbt 0, r1
li r2, 64
dcbt r2, r1
li r3, 128
dcbt r3, r1
li r4, 192
dcbt r4, r1
...
```

The load instruction read at most 32 bytes, so half the number of **dcbt** instruction that load instructions are required. However, this is still a waste of registers such that these constants need to be rematerialized in a loop.

Unfortunately there is no variant that updates the register containing the address like the **qvlfdux** instruction does. Instead, for each fullspinor/halfspinor/SU(3)-matrix, we take their base address and emit C-inline assembly in 64 byte strides. The result for prefetching a halfspinor is:

```
asm (
   "dcbt      0 ,%[ptr]  \n"
   "dcbt %[c64 ],%[ptr]  \n"
   "dcbt %[c128],%[ptr]  \n"
     : :
   [ptr ] "r" (addr),
   [c64 ] "b" (64),
   [c128] "b" (128)
);
addr += 192; // Assuming addr is a char*
```

[1] Without knowing the compiler's internals we can only guess what the reason for this behavior is. A reasonable assumption is that it is because **qvlfdux** overwrites the register containing the address but it is still needed by the next load instruction in the sequence. This blocks the register from being overwritten except for the last update which is not reused.

[2] offset can be 0 without reading a register

The strides between the vectors is 192 bytes that are to be added to the address between the prefetches. Just three general purpose registers for the constants 64, 128 and 192 are used no matter how many consecutive vectors are to be loaded.

4.8.3 Avoiding Spilling

One of XLc's optimization goals is to keep dependent instructions apart from each other. Instructions have latencies and if another instruction uses a result within the latency, the processor adds penalty cycles. The compiler tries to avoid that by inserting non-dependent instructions in between. Due to SMT a different thread can execute during the penalty cycles which makes instruction scheduling less useful, but remains important because there are only up to 4 threads that cannot fill the delay if all threads execute latency-6 instructions.

The drawback is that instruction scheduling increases the number of registers required for intermediary results. If scheduling is done before register allocation, the chances increase that the 32 registers (QPX or general-purpose) are not sufficient and the compiler as to temporarily swap the registers to memory ("spilling"). And this is exactly what happens.

Scheduling can be restricted by a special construct

```
#define REORDER_BARRIER asm volatile ("");
```

The **asm** statement is usually used to insert machine instructions, but here it is empty. The volatile keyword notifies the compiler that the machine instructions inside have undefined side-effects. The compiler therefore has to assume that those side-effects interact with the code before and/or after the statement. It will not move any instructions across the barrier anymore. The assembler code does not have side-effects, but the volatile property has the effect of restricting the compiler's scheduler.

By inserting such barriers between the 8 stencil point computation such that their instructions do not interleave anymore, any spilling was removed from the halfspinor Hopping Matrix generated by XLc. There is still spilling in the fullspinor variant.

4.9 Symmetric Multiprocessing and Simultaneous Multithreading

The standard paradigms for shared memory parallelization in C are *Pthreads* and OpenMP. The Pthreads API defines functions to start new threads and synchronization primitives and therefore the programmers themselves have to write the code that organizes the execution of his program. In contrast, OpenMP lets the programmer define sections of code that can run in parallel using the fork & join model.

Although IBM optimized its OpenMP implementation for the Blue Gene/Q architecture [18], its overhead has a noticeable impact on the overall performance. One reason is that the compiler encapsulates the loop body into a function. Every loop iteration becomes a call to that function by the OpenMP runtime. The most hurtful is that code motion before the loop cannot take place, and for instance loading of the constants $\kappa_0, dots, \kappa_3$ has to take place in every iteration. Inlining could resolve this, but unfortunately the OpenMP runtime is only added at link-time.

There are also some difficult-to-explain performance characteristics of IBM's implementation of OpenMP. For instance, the `nowait` clause, meant to reduce synchronization overhead, actually increases the overhead on parallel for-loops.

Therefore we chose to implement a minimal parallel runtime that does avoid such kind of overhead. The requirements for this basic parallel library are (1) some way to start threads at the beginning of the program, (2) a possibility to uniquely identify the threads[1] and (3) a barrier implementation. Both, Pthreads and OpenMP fulfill these fundamental requirements,

[1] Otherwise two indistinguishable threads necessarily do the same computation

but OpenMP has better support by IBM. The overhead of OpenMP only applies once at program startup and therefore is not relevant.

We are going to use (1) the `#pragma omp parallel` construct to start the threads, (2) `omp_get_thread_num` to get a thread identifier (`tid`) and (3) we have multiple choices for the barrier implementation which are going to be presented in Section 4.9.1.

The basic algorithm is shown in Listing 4.3. There are two sections of infinite loop, one in which only the master thread (thread id 0) is allowed to write global data, and a second where all the worker may read that data. Specifically, the workers read the description of the work they have to do. Note that the master thread also joins the workers.

```
global work, threads;

#pragma omp parallel
{
  local tid = omp_get_thread_num();
  if (tid == 0)
    threads = omp_get_num_threads();
  while (1) {
    if (tid == 0)
      work = master_ControlProgram();

    barrier();

    // Fork
    local mywork = work;
    worker_DoTheWork(mywork);

    barrier(); // Join
  }
}
```

Master writes to global

Workers (including master) read global

Listing 4.3: Master/worker algorithm (in C-like pseudocode)

Listing 4.3 has a message loop-like structure, however C does not support coroutines[1] such that `master_ControlProgram` could return when entering a parallel part and then continue from that position after the computation has finished. Therefore the master thread actually exits the endless loop to continue execution and re-enters it for the next parallel execution instead, without leaving the scope of `#pragma omp parallel`.

`worker_DoTheWork` calls a function pointer from `work`. These worker callees typically have the structure in Listing 4.4. Each thread derives for which fraction of the workload it is responsible for and executes it. It will return into the barrier in the endless loop and wait for the next work descriptor.

If the workload is not equally distributable between all threads, the thread with the highest `tid` does the least amount of work. For this reason the thread identifiers are rotated by one to the left such that the master thread has the highest `tid` in the worker, because the master does the most overhead work.

4.9.1 Thread Synchronization

There are multiple thread barrier implementations available from which we have to pick one. The barrier has to ensure that either all threads are in the master-writes or in the workers-read section. The choices are:

[1] Procedures that interrupt their execution to run a different part of the program; a typical example is the `yield` and `async` features from C# 5.0

```
void worker_callee(void *arg_untyped, size_t tid, size_t threads) {
  struct work_descriptor *arg = arg_untyped;

  size_t workload = arg->count;
  size_t threadload = (workload+threads-1)/threads;
  size_t begin = tid*threadload;
  size_t end = min(workload, begin+threadload);
  for (size_t i = begin; i < end; i+=1) {
    stencil(i); // inlined
  }
}
```

Listing 4.4: Worker callee

1. `#pragma omp barrier`
2. `pthread_barrier_t`
3. `L2_Barrier_t`

The first is also the most straightforward strategy since we are using OpenMP anyway. The POSIX alternatives, although not guaranteed to also work under threads started by OpenMP, involve an even higher overhead because these are defined as kernel calls.

The third option is a barrier implementation found in the header files of every Blue Gene/Q system. It is a busy-waiting implementation using L2 atomic operations. `L2_Barrier_t` atomically increases a counter and then loops until all threads have done so.

4.10 Elementwise Operations

A per-spinor operation is a function taking one or multiple spinors from the same coordinate of different fields as input and returning another spinor. An elementwise operation on a field is the application of such a function on all the sites of a spinor field. If the function takes multiple spinors as input (e.g. addition) then the field operation also takes multiple spinor fields and applies the function on all sites *at the same coordinates*.

Such operations are the simplest to implement. The contiguous stream of local sites is split equally between the threads. Each thread loads the input vectorized spinor(s), possibly already prefetches the next, invokes the function, and stores the spinor again at the new location. Accesses are always linear such that there is no issue with the stream prefetcher. The size of a working set (bytes loaded per physical site) is 384 times the arity of the per-spinor function which does fit well into the L1 cache. The only things needing being cared for is how to vectorize the per-spinor function and to avoid spilling. Ideally, the compiler already generates the optimal code.

4.11 Reductions

Similar to the elementwise operations, reductions read all the elements of a field (or vector) but their output is a single value instead of a field. The scalar product is an example. Reductions can usually be described as an induction: The reduced value of a zero-length vector, a per-element function f to reduce a one-element vector, and an associative function $\circ$ that combines the reductions of two vectors.

The loads of the spinors and the per-element function can be organized as the elementwise operations in the previous section. The only problem is how to combine the reductions of

different threads and nodes. The implementation uses a schema as illustrated in Figure 4.12. It heavily uses the associativity of the combine function.

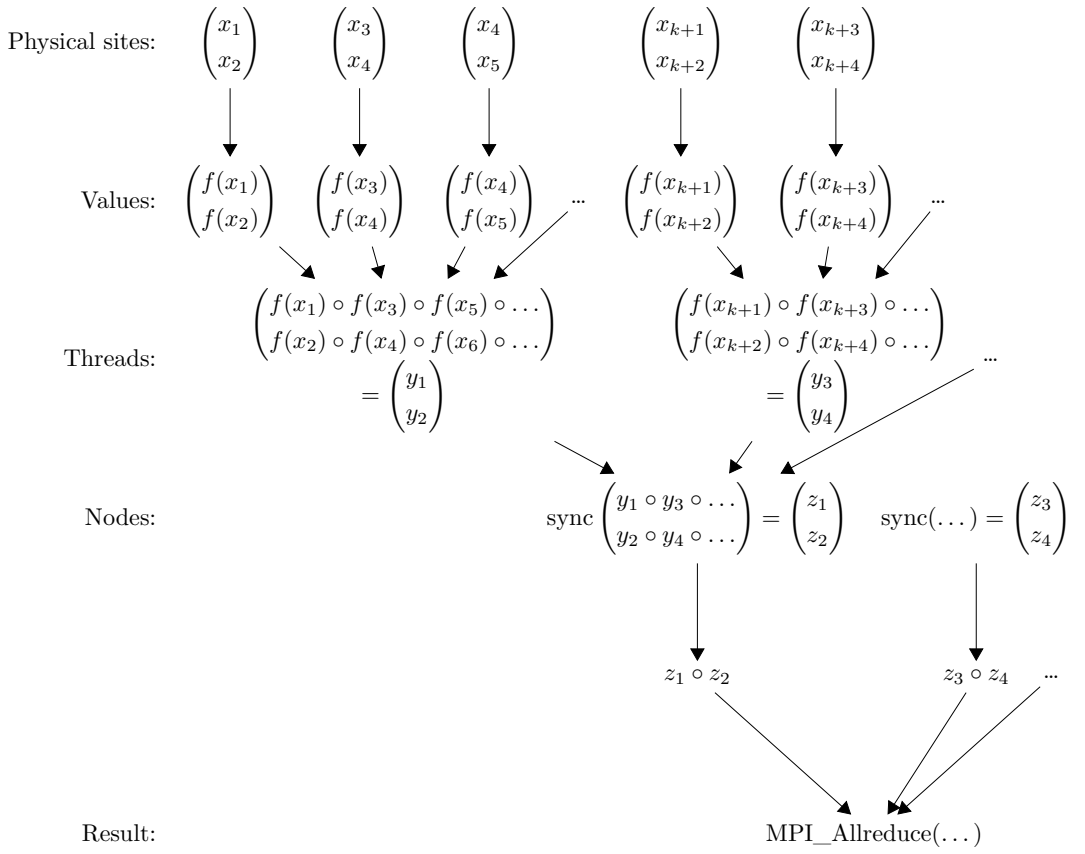


Figure 4.12: Reduction schema

Each thread applies the per-element function f on its portion of the spinor field and combines them to a per-thread intermediate result y_i. This is essentially data privatization since every thread operates on its own copy of the reduction result, no synchronization is required. When all threads are done, the node's master thread reads all intermediate results (up to 64 on Blue Gene/Q) and combines them using $\circ$ to a per-node intermediate result. Up to to here all values are vectorized in QPX registers. If the reduction result is a complex value, then two of them fit into one QPX register. In case of a real-valued reduction even 4 values can be processed at once. The individual vector results are folded into a scalar result. Finally, MPI combines the results of all nodes to a single reduction result.

4.12 Messaging Unit System Programming Interface

Experiments show that the MPI implementation on Blue Gene/Q does not transfer data asynchronously, in parallel during the computation of the body stencils. Instead, it seems to wait for the call to `MPI_Wait` and only then starts the data transfer. Total execution time could be reduced significantly – in some situations down to one half – if overlapping of communication and computation worked.

Instead of trying PAMI (IBM's proprietary alternative for MPI), we directly program

the hardware using MUSPI, the API for programming the message unit of the A2 chip. IBM's implementation of MPI and PAMI also use MUSPI as their backends, therefore some intermediate layers are simply skipped. It is possible to use these higher-level APIs and MUSPI in the same program and therefore we continue to use MPI for synchronous operations like reductions (Section 4.11).

The skeleton of MUSPI node-to-node communication has been taken from an IBM example program demonstrating how every node sends a message to every other node. It needed to be modified such that messages are only sent to a node's neighbors.

The three main objects of MUSPI are FIFOs (First-In-First-Out queues), message descriptors and memory regions. A message descriptor describes an network operation, including the memory position and length of the data to transfer. Such message descriptors are appended to a FIFO for the hardware to process. There are 16 such FIFOs which the message unit processes in parallel. We use one FIFO per neighbor and leave the remaining 8 FIFOs to be used by MPI. The message descriptors can describe different types of operations, of which we only use the "Direct Put" operations, a remote memory write. The receiving node has to set up a region in memory that receives the data.

The facilities required by tmLQCD that both interfaces need to provide are:

- For each neighbor node one send buffer and one receive buffer.
  The MPI interface allows to specify any memory address as source or target buffers, but MUSPI has special memory allocators for such buffers. Hence, MPI needs another operation to copy the data to these buffers which becomes unnecessary by directly writing into those buffers.

- A function that starts sending data to the neighbor nodes.
  MPI: `MPI_Isend` or `MPI_Start`
  MUSPI: Injection of a job into a hardware-managed FiFo list

- A function that waits until all data has been received and sent.
  MPI: `MPI_Wait`
  MUSPI: Busy-waiting until a counter reached the expected number of received bytes

- A function for resetting the status such that data can be received again.
  MPI: `MPI_Irecv` or `MPI_Start`
  MUSPI: Reset the receive counter and FiFos

Any network operation is initiated by the master thread of each node to avoid additional thread synchronization.

5 Results

In this chapter we present how the presented techniques perform on Juqueen, the Blue Gene/Q cluster in Jülich, Germany with 28 racks. The times specified are the execution time of one Hopping Matrix execution on a spinor field. Hopping Matrix is the most time consuming part of any Lattice QCD simulation. It is the base of, for instance, the iterative solver as presented in Section 2.3 Page 25.

In the Lattice QCD research community the execution performance is most often measured in number of floating points operations (*flop*, plural: flops) per second (*flop/s*). It is possible because a Dslash stencil in most programs has 1320 floating point operations which makes this measure comparable between many Lattice QCD program suites. In addition, every hardware has a theoretical number of floating point operations it can execute per second, called the peak flop/s. As seen in Section 3.2.2 (Page 36), a Blue Gene/Q node can theoretically execute up to 204800 million flops per second (mflop/s).

In practice, other hardware resources such as memory bandwidth reduce the execution speed as well, and as a result it is close to impossible to have any productive program reach the peak performance. However, to measure the level of optimization of a program, one can use a percentage of the peak performance (% peak). The reader may remember from Section 4.2.1 (Page 47) that when counting raw flops, only about 78% can be reached due to the kinds of FMA-instructions available.

When Hopping Matrix is executed in another algorithm such as conjugate gradient, other operations may have different performance characteristics such as the number of flops executed. Other solvers have different characteristics. They may execute more flops per iteration but require fewer iterations to reach the same goal. The number of iterations depend on system to solve itself making comparisons on this level difficult. Therefore we compare the performance of an Hopping Matrix operation only although in practice it is never used as the only operation.

In contrast to most programs tmLQCD uses 1608 flops per stencil as explained in Section 2.4.3. In order to remain compatible to tmLQCD, implementations of tmLQCD also have to use this algorithm. As a result, comparisons to 1320 flops stencil variants are now longer meaningful as more work is done. The original benchmark program from the tmLQCD suite returned numbers still assumed 1320 flop per stencil, effectively defining a stencil to have 1320 units with no real relation to the actual work done by the processor.

To resolve this issue we prefer to report the number of executed stencils in this thesis, not the number of floating-point operations. The unit is *mlup/s* – mega/million lattice updates per second. A lattice update corresponds to the storage of the result's stencil into the target spinor field. The number of flops can be obtained by multiplying with 1320, receptively 1608. Also, the memory bandwidth can be computed by multiplying with the data size loaded and stored per stencils as seen in Table 2.3 on Page 30.

The number of work done per time unit depends on the size of the cluster involved in the calculation. The more nodes participate the more work is done per second. The measure that remains comparable is the amount of work done per node. As we will see in this chapter

the performance per node does not vary a lot with changing cluster size making it a good measure to compare optimizations and configurations. Another alternative, performance per core, suffers from shared memory bandwidth with the L2. A core is faster whin it is the only active core of the node.

Timings shown here are the averages of 5 executions with 2 two non-timed executions beforehand to avoid having the warm-up in the measurement.

5.1 Fullspinor Layout

This section is the evaluation of the Hopping Matrix optimization that uses fullspinors as described in Section 4.4.1 Page 50 and following. It is specifically optimized to make best use of limited memory bandwidth. The halfspinor layout came into existence because it became clear that best performance on Blue Gene/Q cannot be gained with too many prefetch streams.

Because it is the first implementation, not all of the later optimizations added to the halfspinor variant have been backported to this layout. Most notably it does not feature direct programming of the messaging hardware using MUSPI. Also, this implementation shows how large the overhead of OpenMP really is, which later lead to the custom work dispatch system as described in Section 4.9. Nonetheless it makes sense to take a look on the performance characteristics.

Time	Stencils/node	Flop/node	Peak %	Variant
1.17 ms	16.46 mlup/s	26466 mflops/s	12.9 %	64 bit, 1608 flop/stencil, MPI
0.97 ms	19.73 mlup/s	31733 mflops/s	15.5 %	32 bit, 1608 flop/stencil, MPI
1.09 ms	17.63 mlup/s	23268 mflops/s	11.4 %	64 bit, 1320 flop/stencil, MPI
0.85 ms	22.60 mlup/s	29828 mflops/s	16.6 %	32 bit, 1320 flop/stencil, MPI
0.84 ms	22.78 mlup/s	36632 mflops/s	17.9 %	64 bit, 1608 flop/stencil, nocom
0.73 ms	26.40 mlup/s	42450 mflops/s	20.7 %	32 bit, 1608 flop/stencil, nocom
0.77 ms	24.82 mlup/s	32757 mflops/s	16.0 %	64 bit, 1320 flop/stencil, nocom
0.61 ms	31.55 mlup/s	41345 mflops/s	20.3 %	32 bit, 1320 flop/stencil, nocom

Table 5.1: Execution time on a 64x24x20x20 lattice using 32 nodes (4x2x2x2), 32 OpenMP threads each

Table 5.1 and Figure 5.1 show the results of some of the most characteristic experiments. The table needs some explanation. Every line represents an execution of Hopping Matrix on a lattice of size 64x24x20x20, that is 614400 sites. The 32 nodes are partitioned in a 4x2x2x2 pattern, i.e. each node is responsible for a rectangle of 16x12x10x10 = 19200 sites. This volume was chosen because it performs the fastest (in double precision). For the same reason every node executes 32 OpenMP threads – it is the fastest.

The physical shape of a job with 32 nodes on Blue Gene/Q is 2x2x2x2x2. Because the width in every dimension is just 2 it practically is a torus although technically only the last is implemented as one (see Table 3.1 on Page 40). The two nodes connect to each other with twice as much bandwidth that the other neighbors. One of the dimensions can be folded into a circle such that the logical geometry of 4x2x2x2 has torus-only dimensions.

The time column shows the execution time of one Hopping Matrix call in milliseconds. The number of stencils (614400) divided by the time in seconds and number of nodes (32) gives the performance measure in lup/s in the second column. Multiplied by the number of flops per stencils (1320, respectively 1608) is the performance in flop/s found in the third column. This number divided by the node's theoretical peak of 204800 mflop/s gives us the percentage of peak in the fourth column.

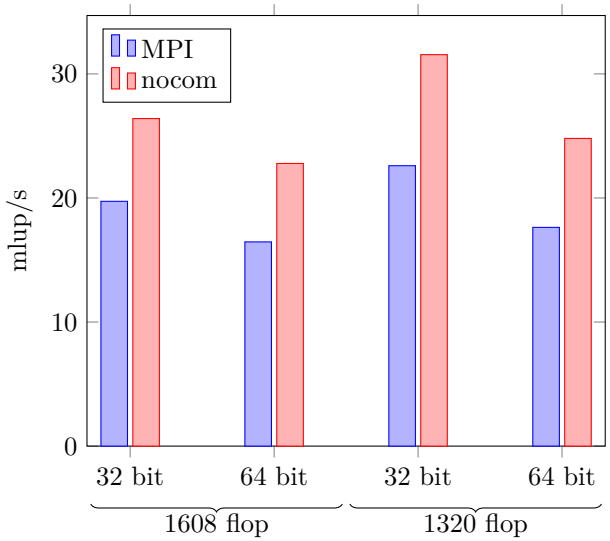


Figure 5.1: Visualization of Table 5.1

The last column specifies which configuration was used for this line. 64 bit means that the calculation was carried out using double precision IEEE 754 floating point numbers. In 32 bit configurations the values are stored and loaded as 4 byte single precision values, but the calculations are still done in double precision because the QPX unit has no dedicated single precision mode. The differences between the modes gives an impression on the influence of memory bandwidth.

The difference between 1608 and 1320 flop/stencil has already been explained. In addition to the configuration doing data transfers using MPI, there are configurations where communication is disabled ("nocom "), i.e. data transfer commands are just skipped. Although the Dslash output will not be correct, we can deduce the communication overhead by comparing to the normal configuration with MPI enabled.

From the numbers we can wee that the single precision variants are about 20% faster than with double precision. This is the influence of the memory bandwidth. With communication enabled, the gap increases to 28% because less data has to be transferred between nodes.

As expected, the 1608 flop configurations execute more floating point operations per second but fewer stencils. The communication takes up to 30% of the total runtime.

In essence, the fullspinor layout reaches up to 20% of the machines theoretical peak. This is the performance in single precision and without communication only. Further speed improvements are realistic by replacing MPI and OpenMP as done for the halfspinor stencil.

5.1.1 Threads per Node

Here we compare configurations in double precision, 1320 flop per stencil and disabled communication and vary the number of threads running within an MPI process. The performance results can be seen in Table 5.2.

The speedup shows nearly perfect scaling up to 32 threads, although there are only 16 physical cores. The most reasonable explanation is because every core has two functional units. The AXU does the floating-point math while the XU executes all other instructions (see Section 3.2.1). The almost perfect scaling is still surprising because this requires a 50%/50% split of QPX and other instructions. Data fetch latency might also contribute to this.

When even more threads are added to the system the performance decreases again. The 48 threads case shows a slight increase in this run but not in most other configurations. There

Time	Stencils/node	Threads
0.98 ms	19.61 mlup/s	64
0.77 ms	24.92 mlup/s	48
0.77 ms	24.82 mlup/s	32
1.29 ms	14.88 mlup/s	16
2.40 ms	8.01 mlup/s	8
4.54 ms	4.23 mlup/s	4
8.81 ms	2.18 mlup/s	2
16.22 ms	1.18 mlup/s	1

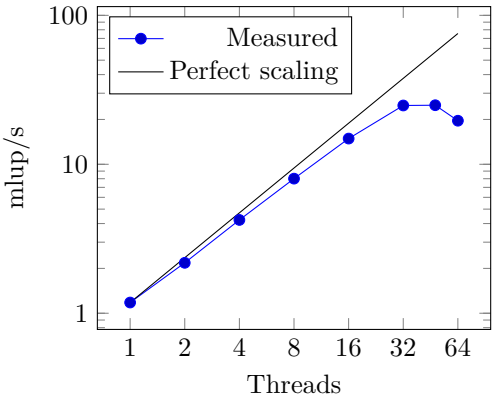


Table 5.2: Varying number of threads, 64x24x20x20 lattice on 32 (4x2x2x2) nodes, 64 bit, 1320 flop/stencil, nocom

are two possible explanations for this behavior: The first one being thrashing of data in the comparatively small L1 caches. The second are the exhaustion of the 16 prefetch streams shared between all threads of a core.

5.1.2 Lattice Size

In this section we vary the size of the lattice to tackle. The results are shown in Table 5.3 and Figure 5.2, both, with MPI and disabled communication.

MPI		nocom		
Time	Stencils/node	Time	Stencils/node	Volume per node
0.33 ms	1.56 mlup/s	0.13 ms	3.94 mlup/s	8x4x4x4
0.37 ms	2.07 mlup/s	0.18 ms	4.36 mlup/s	12x4x4x4
0.41 ms	3.71 mlup/s	0.20 ms	7.50 mlup/s	24x4x4x4
0.46 ms	8.88 mlup/s	0.27 ms	14.94 mlup/s	8x8x8x8
0.6 ms	10.16 mlup/s	0.39 ms	15.76 mlup/s	12x8x8x8
0.91 ms	13.58 mlup/s	0.63 ms	19.62 mlup/s	24x8x8x8
0.93 ms	14.80 mlup/s	0.64 ms	21.51 mlup/s	8x12x12x12
0.97 ms	14.81 mlup/s	0.70 ms	20.68 mlup/s	12x12x10x10
1.17 ms	16.46 mlup/s	0.84 ms	22.76 mlup/s	16x12x10x10
1.30 ms	12.00 mlup/s	0.98 ms	21.22 mlup/s	12x12x12x12
2.71 ms	10.21 mlup/s	2.25 ms	12.30 mlup/s	16x12x12x12
8.82 ms	7.43 mlup/s	7.63 ms	8.59 mlup/s	16x16x16x16
21.73 ms	7.26 mlup/s	20.23 ms	7.91 mlup/s	20x20x20x20
44.24 ms	7.50 mlup/s	41.65 ms	7.97 mlup/s	24x24x24x24

Table 5.3: Varying local volume on 32 (4x2x2x2) nodes, 64 bit precision, 1608 flop/stencil, 32 threads, MPI

Without communication there is a speedup until the local volume of 16x12x10x10 is reached. This is the effect of lower relative overhead. For instance, the border sites are treated differently such that more non-surface sites generally improves performance.

Larger local volumes result in a slowdown again. The reason is that the working set no longer fits into the L2 cache so the much slower (bandwidth and latency) main memory is

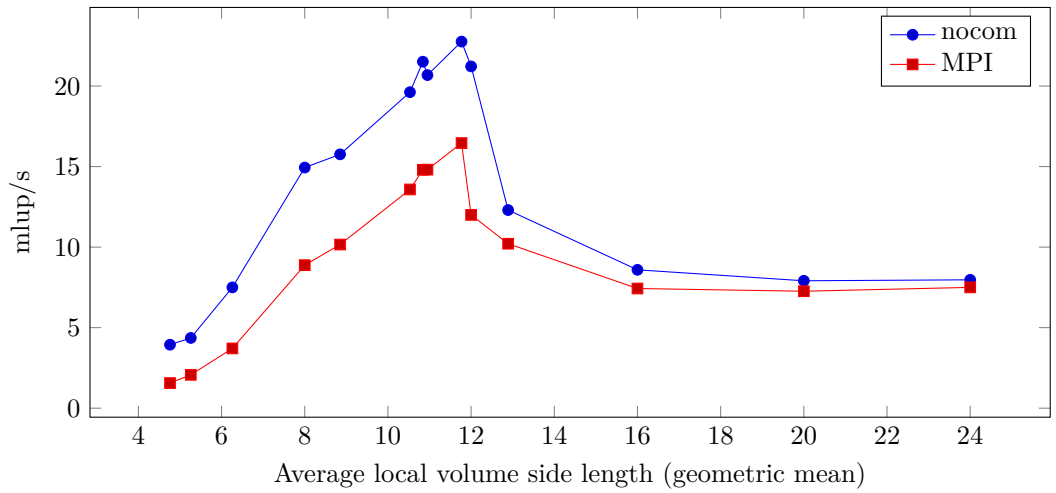


Figure 5.2: Visualization of Table 5.3

accessed. The shape of the lattice has a lesser impact on the performance. The exception is 8x12x12x12 which has fewer sites than 12x12x10x10 but has better performance. It has more sites in z-direction which is also the direction of the innermost loop. Evidently, the longer the inner loop runs the better the performance.

The trend is the same even if communication is enabled again. As expected, the communication overhead relative to total execution time is larger with smaller lattices since the amount of sites to be transferred relative to the number of sites is larger.

5.1.3 Weak Scaling

In weak scaling we increase the size of the cluster when growing the problem size at the same scale. In other words, the volume per node remains constant. Table 5.4 shows what happens when more nodes are added to the system.

Time	Stencils/node	Volume	Nodes
1.47 ms	13.08 mlup/s	1x1x1x1	1
1.34 ms	14.36 mlup/s	2x1x1x1	2
1.17 ms	16.45 mlup/s	2x2x1x1	4
1.13 ms	16.96 mlup/s	2x2x2x1	8
1.17 ms	16.43 mlup/s	2x2x2x2	16
1.17 ms	16.46 mlup/s	4x2x2x2	32
1.17 ms	16.42 mlup/s	4x4x2x2	64

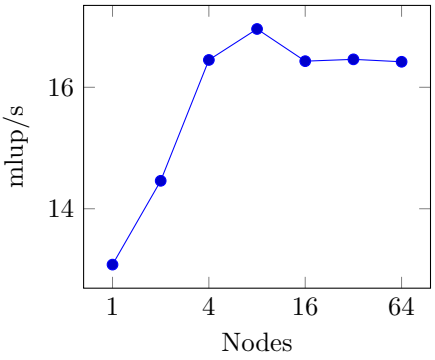


Table 5.4: Weak scaling, 16x12x10x10 local lattice, 64 bit precision, 1608 flop/stencil, 32 threads, MPI

It can be observed that the scaling is perfect from 4 nodes onwards. The lower performance with just one or two nodes can also be explained. The implementation does not recognize when the neighbor is the node itself, hence it will still call MPI to transfer data into that direction. The MPI runtime supports this by copying the data from the send buffer to the

receive buffer on the same node. This copy is carried out in software in contrast to inter-node transfers which are done by the MU in hardware.

With one node, all the 8 neighbors refer to the node itself. With two nodes, 6 of them require a `memcpy` operation. With 4 and 8 nodes the effect seems negligible, hidden by the overhead of the communications themselves.

5.2 Halfspinor Layout

This is the evaluation of the halfspinor optimization as described in Section 4.4.2, starting at Page 57. The goal of the memory layout is to ensure that the number of data load stream does not exceed the number of prefetch streams (16 per core). The most notable results are shown in Table 5.5 and Figure 5.3.

Time	Stencils/node	Flop/Node	Peak %	Variant
0.37 ms	38.92 mlup/s	62864 mflop/s	30.7 %	64 bit, 1608 flop/stencil, MUSPI
0.28 ms	50.62 mlup/s	81398 mflop/s	39.7 %	32 bit, 1608 flop/stencil, MUSPI
0.35 ms	41.14 mlup/s	54300 mflop/s	26.5 %	64 bit, 1320 flop/stencil, MUSPI
0.26 ms	54.78 mlup/s	72316 mflop/s	35.3 %	32 bit, 1320 flop/stencil, MUSPI
0.25 ms	57.84 mlup/s	93014 mflop/s	45.4 %	64 bit, 1608 flop/stencil, nocom
0.21 ms	67.07 mlup/s	107856 mflop/s	52.7 %	32 bit, 1608 flop/stencil, nocom
0.23 ms	62.77 mlup/s	82860 mflop/s	40.5 %	64 bit, 1320 flop/stencil, nocom
0.19 ms	74.71 mlup/s	98612 mflop/s	48.2 %	32 bit, 1320 flop/stencil, nocom
0.55 ms	26.21 mlup/s	34597 mflop/s	16.9 %	64 bit, 1320 flop/stencil, MPI
0.48 ms	30.00 mlup/s	39602 mflop/s	19.3 %	64 bit, 1320 flop, sync. MUSPI

Table 5.5: Execution time of a 12x10x10x12 local lattice per node on 32 (4x2x2x2) nodes, 64 threads each.

Time	Stencils/Node	Flop/Node	Peak %	Variant
0.25 ms	57.70 mlup/s	92769 mflop/s	45.3 %	64 bit, 1608 flop/stencil
0.25 ms	69.00 mlup/s	110991 mflop/s	54.2 %	32 bit, 1608 flop/stencil
0.24 ms	60.60 mlup/s	79988 mflop/s	39.1 %	64 bit, 1320 flop/stencil
0.20 ms	72.37 mlup/s	95529 mflop/s	46.6 %	32 bit, 1320 flop/stencil

Table 5.6: Execution time of a 12x10x10x12 lattice on a single node, 64 threads

The explanations for the columns and configuration are mostly the same as for Table 5.1 with some additions explained below. In general the performance level is much higher compared the fullspinor layout. The highest percentage of peak here is 52.7 % compared to the 20% of the previous layout.

One recognizes that already the use of MUSPI is a huge performance boost. In direct comparison, MUSPI reaches 26.5% of peak, but using MPI in an otherwise equal configuration has 16.9% only. For comparison, there is a configuration with synchronous MUSPI which reaches 19.3%. It is implemented by adding a wait directly after the data transfer has been started such that computation does not start before the communication has finished. It performs still better than the MPI version. One may assume that the MPI operations are synchronous as well although they are used properly using MPI persistent communication and `MPI_Wait`.

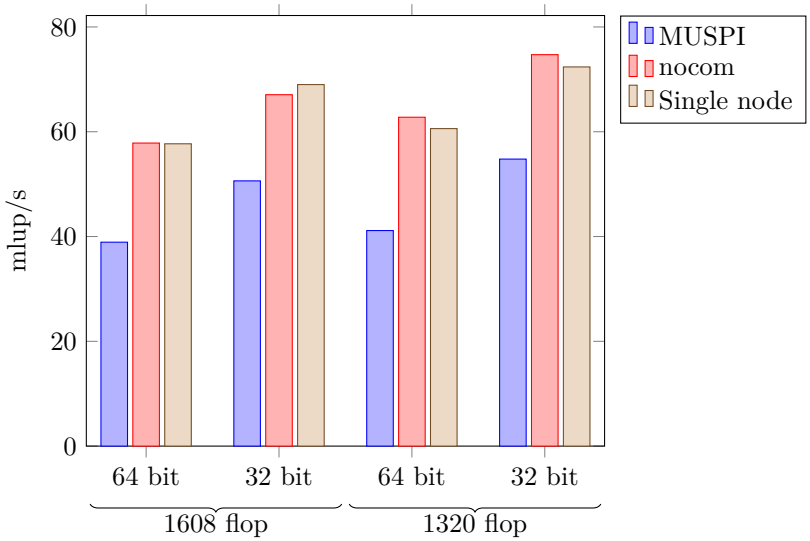


Figure 5.3: Visualization of Tables 5.5 and 5.6

The configurations in Table 5.6 use just one node instead of 32. They are shown here because they are the fastest of all the configurations. Because the MUSPI API does not support transfers within the same node, detection of dimensions of size one had to be implemented. In those dimensions the implementation does not use transfer buffers but writes directly to the correct location of the target spinor field. It therefore has the least overhead, even less than when communication is disabled but data is still copied to and from buffers ("nocom" configurations). The maximum speed reach by this variant is 54.2% of peak flop performance, the fastest speed of any experiment in this thesis.

5.2.1 Threads per Node

Here we again change the number of OpenMP threads while keeping all the other parameters the same, visualized in Table 5.7

Time	Stencils/node	Threads
6.53 ms	2.21 mlup/s	1
3.46 ms	4.16 mlup/s	2
1.74 ms	8.28 mlup/s	4
0.88 ms	16.36 mlup/s	8
0.45 ms	32.00 mlup/s	16
0.24 ms	60.00 mlup/s	32
0.25 ms	57.60 mlup/s	48
0.23 ms	62.77 mlup/s	64

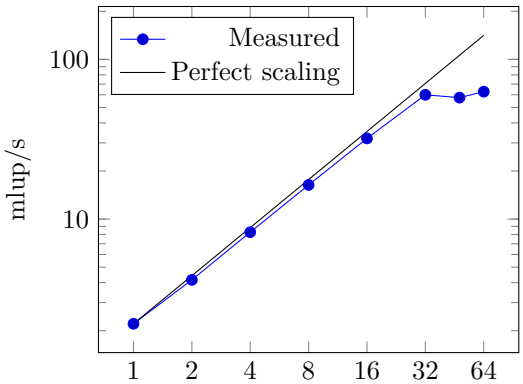


Table 5.7: Varying number of threads, 48x20x20x12 lattice on 32 (4x2x2x2) nodes, 64 bit, 1320 flop/stencil, nocom

The data shows about the same characteristic as for the fullspinor implementation with the notable difference that the 64 threads variant is faster than the one with only 32 threads.

A thread in the halfspinor configuration requires 3 prefetch streams such that with 4 threads per core, 12 of the available 16 them are used. Other configurations actually show a larger difference between the 32 threads and 64 thread case.

5.2.2 Lattice Size

MUSPI		nocom		
Time	Stencils/node	Time	Stencils/node	Volume per Node
0.06 ms	4.4 mlup/s	0.02 ms	14.6 mlup/s	4x4x4x4
0.07 ms	7.2 mlup/s	0.03 ms	17.5 mlup/s	8x4x4x4
0.15 ms	27.1 mlup/s	0.08 ms	52.1 mlup/s	8x8x8x8
0.37 ms	39.1 mlup/s	0.25 ms	58.8 mlup/s	12x10x10x12
1.35 ms	15.3 mlup/s	1.27 ms	16.3 mlup/s	12x12x12x12
7.26 ms	9.0 mlup/s	6.82 ms	12.7 mlup/s	16x16x16x16

Table 5.8: Varying local volume, 48x20x20x12 lattice on 32 (4x2x2x2) nodes, 64 bit, 1608 flop/stencil, 64 threads, MUSPI

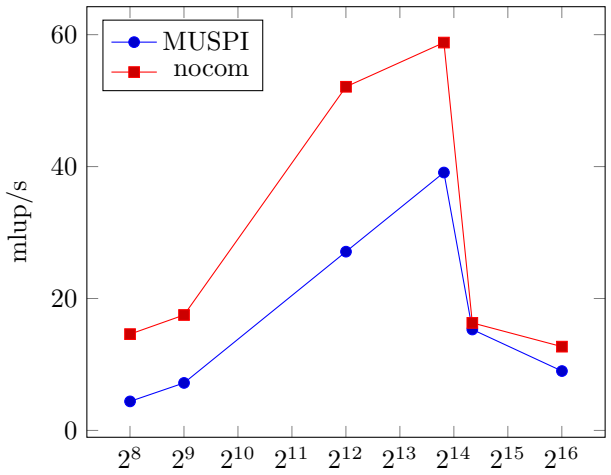


Figure 5.4: Visualization of Table 5.8

The data in Table 5.8 and Figure 5.4 shows the behavior when the lattice volume is changed without changing the number of involved nodes. The basic characteristics are already known from the fullspinor implementation: An improvement in performance until the working set exceeds the size of the L2 cache.

The difference is where this point is reached. The best size for the fullspinor implementation was 16x12x10x10 where here it is 12x10x10x12 for the halfspinor case, i.e. ¼ fewer sites. This is not surprising because the spinor field is 4 times as large. The gauge field however grows by a factor of 2 only so we do not see a factor 4 difference. Another contributing factor is that the main memory can be used to some extend, but its latency matters less if the stream prefetcher has enough time in advance with strictly consecutive accesses.

Since the fullspinor layout requires less memory bandwidth set for the same lattice size one may assume that it is faster if the working set is mostly in main memory because the memory bandwidth becomes the limiting factor. Actually, the halfspinor implementation is still faster

as can bee seen with the local lattice size of 16x16x16x16, although not by much (8.59 mlup/s to 12.7 mlup/s). This might be due to effective prefetching, even from main memory.

5.2.3 Weak Scaling

As the last measurement we also experiment with the weak scaling behavior of the halfspinor implementation. The data analyzed here are available in Table 5.9.

Time	Stencils/node	Shape	Nodes
0.26 ms	55.00 mlup/s	1x1x1x1	1
0.29 ms	48.90 mlup/s	2x1x1x1	2
0.32 ms	45.60 mlup/s	2x2x1x1	4
0.38 ms	37.80 mlup/s	2x2x2x1	8
0.37 ms	39.40 mlup/s	2x2x2x2	16
0.37 ms	38.90 mlup/s	4x2x2x2	32
0.36 ms	40.48 mlup/s	4x4x2x2	64

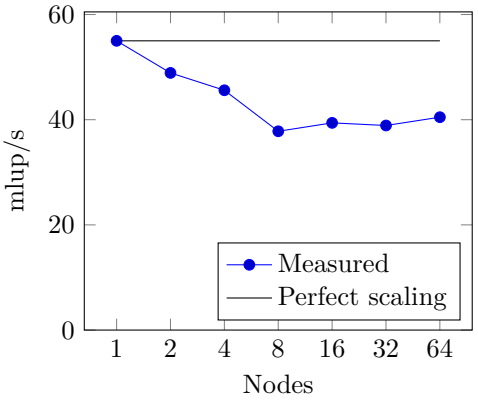


Table 5.9: Weak scaling, 12x10x10x12 local lattice, 64 bit, 1608 flop/stencil, 64 threads, MUSPI

This time the characteristics are quite different than in the fullspinor implementation because of the use of MUSPI with code that detects length-one dimensions and does not use communication in these dimensions. Instead of the `memcpy` overhead it actually reduces the overhead. A one node run effectively is the same as a pure OpenMP-based implementation. Of course the same mechanics can be implemented in MPI which is even supported by the halfspinor implementation.

6 Discussion

In the past a lot of effort has been put in optimizing Lattice QCD simulations to which this thesis contributes. Certainly the research will go on in the future with the goal of running simulations even faster even if only a few percentages of improvement seem realistic. But future hardware in the will have different, still unknown characteristics and the optimization game starts over.

This chapter finishes the first part of this thesis by summarizing its contributions, discussing its achievements, pointing to ideas for further improvements and comparing results with work from other research groups.

6.1 Summary

This thesis describes several ways of improving the performance of a *Lattice Quantum Chromodynamics* (Lattice QCD, or just LQCD) simulation, in particular the stencil operation of the Dslash operator. Lattice QCD tries to show that the physical fundamental strong force interaction between quarks and bosons can be accurately described using *Quantum Field Theory* (QFT). The faster the simulation runs, the more accurate simulations are feasible.

Fast supercomputers are just one of the requirements. The other is software that make good use of the available hardware. This thesis describes how program suite *tmLQCD* [3] was modified in order to run faster on the *Blue Gene/Q* supercomputer from IBM. Challenges encountered include inefficiencies in the runtime library that ships with the Blue Gene/Q, especially IBM's implementation of *OpenMP* and *Message Passing Interface* (MPI) which were replaced by custom, close-to-the-hardware implementations. The second challenge was the hand-optimization of assembly code with focus of using vector instructions. And the third challenge was the performance-sensitivity of the processor to the order of data in memory. The latter means that the memory layout had to be reorganized such that data is accessed in a consecutive memory that allows the hardware to prefetch data in advance. This is done by redundantly storing data in the order it is going to be used. If it is used multiple times, it is also stored multiple times. In addition, making use of the Dslash operator's structure reduces the number of operations actually executed.

When the working set exceeds the size of the L2 cache, the bandwidth between main memory and the L2 cache becomes the bottleneck. In this case another implementation is provided that aims for data reuse of data in caches. Data cannot be prefetched as efficiently in this case, but reducing the amount of data to be fetched from main memory helps against the slowdown.

With this optimization we reach up to 54% of the machine's theoretical double precision peak performance in case the first layout is used and all the data fits into the L2 cache. Otherwise, using the second layout, we get up to 20% of the peak performance, compared to 5% of the unoptimized code in plain *C*.

6.2 Conclusions

Program adaptation for specific computer architectures is a necessity. The biggest chunks of improvement are often low-hanging fruits. Some simple annotations added to the code enable the compiler to apply optimizations such as vectorization. Something the compiler does not dare to do otherwise because it cannot prove that it does not change the output of the program. The `#pragma ivdep` tells the compiler that the loop iterations are independent of each other. The keyword `restricts` tells the compiler that pointer cannot alias.

It helps a lot if one has some specific assembly code in mind such that one can check whether the compiled program looks similar. If it does: perfect. The compiler did the hard work for you. But very often it does not which means the hunt for why it does not begins. Sometimes one of the aforementioned hints to the compiler helps. But maybe the compiler does not even implement the kind of optimization one has in mind. The only choice that remains is to implement the low-level, platform specific code by hand. From the experience in this thesis, this is necessary for a lot of cases. For instance, XLc never replaces `switch` statements even if the discriminator is constant. Jump tables are a no-go for performance-relevant loops (remembering the jump destination is a hard task for the CPU's branch predictor; also, it impedes optimizations that follow from knowing there is just one jump location), therefore one has to insert the specific branch manually.

The easier cases are the ones with local changes only. Once a performance-sensitive part of the program has been identified, it is replaced by a variant fo which the compiler produces faster code. Only this part or function is affected. The harder ones are the ones that require replacing multiple locations. One example is if the signature of a function changes, but the primary example of this thesis is the change of the way data is ordered in memory. Different parts of a typical program have to "agree" on how they access data in memory.

Another, classically difficult optimization, is parallelism. Finding a rule on how to distribute work between threads is relatively easy. The hard part is to ensure that those threads do not interfere with each other. For instance by modifying data another thread has currently in use. Transactional memory seems to be a nice framework implemented by Blue Gene/Q but unfortunately the overhead is too high (See Section 6.3.3). Fortunately the work distribution of stencil algorithms such as Lattice QCD does not have such problems. Every thread cares only for its set of stencils. Different stencils do not interfere.

The next level of difficulty is distributed memory parallelism. In addition to tasks assigned to threads, the programmer has to take care that the processed data is also available in the memory the thread has access to. Using MPI, such data transfer must be invoked explicitly and well in advance because the transfer itself is not instantaneous. Adding to the difficulty, one does want to transfer as few data as possible and avoid packing and unpacking of data into message which would again cost processor time.

Most often there is also no single obvious and best optimization. The fastest code for one case may not be the fastest for another case. The two field layouts for tmLQCD, one for large local values and another for small ones, is a good example for this. As a result it is not enough to implement just one optimization per platform, but multiple. Maybe just for trying out which of them really improves performance. In terms of software maintenance, this is a nightmare.

Nevertheless such work is necessary. Unfortunately, the optimized code became largely incompatible with the original tmLQCD code mostly due to the changed memory layout. The speed-up gained seems worth the effort, although this implementation is not as fast as claimed by others (See Section 6.4).

However, one can learn a lot about a platform when doing such optimization. This will turn out useful for the second part of this thesis about compiler-implemented optimization. Actually, many of the techniques used for the manual optimization are either shortcomings of the compiler or are just mechanical application of a principle such as changing iteration order

of a loop. There is no reason that stops a compiler from applying the same transformations described in this part of the thesis.

6.3 Further Possibilities for Optimizations

The optimized version of tmLQCD has implemented many improvements relevant on the Blue Gene/Q supercomputer. However, it does not implement all the optimizations one can think of. Also, there are techniques one does not expect to run faster but still might be worth to try out. This section presents some ideas to further improve the program's performance, or techniques that are just interesting.

6.3.1 Tiling

Both layouts try more or less to read data as sequentially in order to exploit the hardware prefetchers. However, if the working set is larger than 32 MB, trying to reuse data that is already in the L2 cache might stop the DDR memory access from becoming the bottleneck. The usual technique used for this case is tiling.

Tiling further subdivides the local volume into smaller subvolumes (tiles). For instance, if a node owns a hypercube of size 16x16x16x16, it can be subdivided into 16 hypercubes of size 8x8x8x8 which are processed sequentially. The 8x8x8x8 volumes do fit into the L2 cache, making main memory accesses unnecessary for the tile itself. Main memory accesses remain when a tile has been processed and the processor switches to the next one, i.e. there is a warm-up phase per tile.

6.3.2 L2 Prefetching

The hardware prefetchers should work equally well for the L2 cache as it does for the L1p cache so nothing special is required for the halfspinor layout (Section 4.4.2 on Page 57). However, explicit prefetching using the **dcbt** instruction has a longer latency when the data has to be fetched from main memory. Consequently, another **dcbt** should be issued before that with the flag to only prefetch to the L2 cache. Given the latencies of an access, this should be done approximately one iteration before the access itself. A lot of fine-tuning for the correct prefetch depth is required.

L2 prefetching is especially useful in combination with tiling. While the last stencils of a subvolume are being processed, not all the data in the L2 is required anymore. The processor can therefore already start loading the data of the next subvolume.

6.3.3 Transactional Memory

It is tempting trying to exploit a technology that has not been commercially available before, but the overhead of transactional memory is immense: the same code can run 8 times slower compared to the sequential version [19]. There needs to be a significant amount of potential conflicts before a transactional memory version is faster than when critical sections, mutexes or atomic intrinsics are used.

In the case of Lattice QCD there is just one occasion when threads could conflict which is in reduction operations. But this is already resolved very efficiently thanks to Blue Gene/Q's write-through mechanism and by data privatization in Section 4.11.

A different idea is instead iterating over spinors and apply stencils to them, one may iterate over the gauge links. Ever gauge SU(3) matrix is used for 2 stencils. Hence, for every link, the alternative implementation loads the SU(3) matrix and the two (half-)spinors on each side. It applies the matrix-vector multiplication, expands the result to fullspinors, and transactionally

adds the result to the target spinor field. If another thread writes to the target site at the same time, the transactional memory will abort one of the threads which has to retry the operation.

The advantage is that every halfspinor and SU(3) matrix has to be read just once (usually every SU(3) matrix is read twice), reducing the required memory bandwidth. The problem is the transaction overhead, making this scheme unlikely to work faster than the old-fashioned stencil.

6.3.4 Cache Locking

The A2 ISA supports L1 (and L2) cache locking [11]: Cache lines prefetched using the **dcbtls** instruction are established in the cache with a lock bit. The lock bit indicates that this line is never evicted by the cache line replacement policy. The lock bit can be reset by the **dcbf** instruction which also removes the line from the cache.

The lock bit can ensure that data remains in the cache until its last use. If a kernel iteration is small enough such that even with the default replacement policy (8-way associative Least Recently Used) no cache line is evicted before its last local use then cache locking is of no use. Contrariwise, the additional **dcbf** instructions impose some overhead[1]. Some detailed knowledge about when cache lines are established is necessary to use cache locking efficiently. Therefore it is not used in this implementation.

6.3.5 Instruction Cache

No effort has been made that take the L1i into account. It is just assumed that it is big enough to hold all instructions of the Hopping Matrix function. However, due to aggressive inlining and loop unswitching (preference of copying the Dirac stencil instead of having a conditional jump in it) the kernel is quite large. The performance counters (Section 3.2.3, Page 36) do not show a bottleneck here which does not mean that smaller code might not improve the performance.

6.3.6 Combining Even and Odd Hopping Matrices

In Section 4.1 (Page 43ff) we explained the motives for separating each field into an odd and an even part. However, this also has disadvantages. The Dirac operator is always applied on all sites of a field, the even as well as the odd sites. Due to the separation of the sites the sites may be treated differently, but for every Hopping Matrix on even sites there is also a corresponding Hopping Matrix call for odd sites. These can be combined into one function.

The buffers for the surface exchange then is twice as big, but there is just one communication event per direction and therefore less overhead compared to two distinct events. DMM nodes are the coarsest level of parallelism on such machines. Inter-node operations have a large overhead per event but events can be combined for reduced latency.

There are also twice as many body stencils such that the time for the transfer is twice as large. In addition, if each thread knows that it needs to process two half-fields sequentially, an explicit synchronization between the threads is saved. One may still benefit the disjoint working sets by keeping the memory regions of the odd and even sites apart.

6.3.7 Overlapping Per-Node Volumes

A different idea is to exchange surfaces less often, but let the nodes redundantly compute the stencils of the neighbor's surface [20]. For instance, a surface exchange takes place only every second Hopping Matrix. Then, during the executions where no transfer takes place, the node

[1] Alternatively, setting the XU control register XUCR0 can flash-reset all lock bits; useful only if the L1d was large enough to contain the complete stencil.

computes the halo itself (see Figure 4.8 on Page 55). It also needs the halo's neighbor nodes for these stencil which must also be transmitted during the execution where exchanges take place. This is again twice as much data to transfer, but just every second Hopping Matrix.

Effectively, this broadens the stencil's size such that the stencil points are up to 2 sites away instead of just the immediate neighbors. Although it reaches the goal of coarsening data transfer, but it is not clear whether this improves anything because of the additional, redundant stencil computations. The Blue Gene/Q point-to-point transfers are also very fast with low latency when using MUSPI.

6.3.8 Per-Direction Asynchronous Transfers

In the current implementation the processor computes the data that is transferred to any neighbor node before initiating the transfer to them. The processor could also first compute the data to be transferred to one neighbor only, invoke the transfer to it, then continue with the remaining 7 nodes the same way.

After having computed the body stencils it would first wait for the transfer from the node that first started the transfer. When it arrived, it can compute the border stencils adjacent to that border. Then it waits for the second slab of data to receive to compute more stencils, and so on.

For instance, the first node to transfer to is always the left one. After the remaining send invocations and body stencils, it would expect the first data to arrive from the right. Stencils only requiring from the right neighbor node could be computed directly without having to wait for the data from the other neighbors to arrive. Overall, this scheme gives more time for each data transfer while the processor can do other work.

Managing the communication at this fine granularity increases the total overhead as the stencil loops must be interrupted every time data is to be sent or have been received. Also, Blue Gene/Q's torus network is fast such that the transfers are likely to be finished during the body stencil computation anyway, at least if the body-to-surface ration is large enough. Not much is to be expected from this optimization.

6.3.9 Special Assembly Instructions

The PowerPC ISA supports a number of instructions meant for optimizing write accesses to memory. They are to hint to the processor how some data should be managed in the caches. The **dcbst** instruction establishes a cache line and marks it as exclusive to the current core. **dcbz** establishes a cache line by zeroing its content. It does not read the data from later-level caches and therefore does not have a latency penalty. The instruction **dcbf** removes a line from the cache and writes back its content if it has been modified (and, as seen before, resets the lock bit).

None of these instructions are useful for Lattice QCD. Blue Gene/Q does write-through if a cache line has not been established in a cache, i.e. does not establish for a write. Therefore writes have no latency caused by waiting for data. The write prefetch instruction are possibly useful if some data written are soon read again very soon which is not the case for Hopping Matrix.

In either case, **dcbz** is probably fastest way to clear memory. The size of a L1 cache line is 64 bytes which no other instruction can write at once. However, this implementation uses memset from the standard library for which IBM has multiple implementations. The QPX-enabled version always uses **qvstfdux**. Only the generic PowerPC version uses **dcbz** if writing zeros.

6.3.10 Mixed Precision Solver

Memory layouts for double as well as for single precision floating point values have been defined, mostly to see what difference in speed is. Indeed the single precision versions are a little faster. For production runs single precision is not enough so it is not used in practice.

The single precision implementation can still be used as a forerunner of the double precision solver. The solver algorithm such as CG (Section 2.3, Page 25) would first run in single precision to get a low precision result. This result is then used as the first estimation of a double precision solver which then will require the fewer iterations the more precise the first estimation was.

A potential drawback is that switching the algorithm resets caches, branch predictor, etc. The second stage has to warm up to reach its maximum speed.

6.3.11 Assembly Micro-Optimizations

The source code is written in C with a few hints to the compiler to avoid its restrictions. The gap between the theoretical limit of 78% (Section 4.2.1, Page 47) and the highest measured performance of 54% in Table 5.6 (Page 76) is not that large. Lots of the remaining potential probably comes from non-optimal use of the processor pipeline (Figure 3.1, Page 35).

To fix this, an even closer inspection on the pipeline's state is necessary, closer than possible with hardware counters alone since these do not show the location of pipeline stalls. Ideally, one would use a hardware simulator which shows the inner working of the processing for every timestep.

6.4 Related Work

Due to the success of the Blue Gene/Q computer and its roots in Lattice QCD simulation it is not surprising that other research groups optimized their implementation for it as well. The goal of this section is to give an overview over them.

Probably most important is the work of Peter Boyle who has already been involved with the QCDOC project and collaborated with IBM on the design of Blue Gene/Q. His approach is to write a special-purpose machine code generator named Bagel [21, 22]. Bagel is configured with a processor's specification (like number and purpose of functional units, available registers) and then serves as a library that generates optimized code in assembly format. The library's API allows adding instructions that load/store complex numbers, do arithmetic on them, and declare prefetch streams. Boyle vectorizes using *virtual nodes*. A QPX instruction operates on two complex values from two virtual nodes on the same physical node. It is essentially the same principle as used for the halfspinor layout.

The Lattice QCD code generator that uses Bagel is called *BFM* (Bagel Fermion sparse-Matrix). It is a program to generate a variety of Dirac operators, one of which is Wilson Twisted Mass. In addition, it ships with a library for solver algorithms (including CG). This also includes the multi-threading and border exchange primitives for various architectures, including Blue Gene/Q and its MUSPI.

Bagel and BFM have primarily been written for the Columbia Physics System [23] (CPS), the software initially written for QCDOC. Its primary targets now changed to include the Blue Gene family computers as well, including Blue Gene/Q. It also runs on other system using an intermediate layer called QMP. It uses Bagel to generate its kernel code.

Another Lattice QCD software suite using Bagel is Chroma [24]. In contrast to CPS, Chroma is more general purpose in terms of supported platforms, but can be configured to run on Blue Gene/Q using Bagel-optimized kernels. They claim to get 65% of the peak floating-point performance[1] (136.3 TFlop/s per rack).

[1] `http://www.usqcd.org/BGQ.html`

Both suites use the SciDAC (US Department of Energy research program) Lattice QCD infrastructure consisting of QDP++ (QCD Data Parallel lattice operations in C++), QLA (QCD Linear Algebra), the previously mentioned QMP (QCD Message Passing), QIO (QCD/ Input/Output for reading/writing standardized files) and others. Earlier versions of QDP++ were capable of using the Bagel-generated kernels as well, so any software built on QDP++ could use them. However, this support seems deprecated. In addition, QMP supports MPI, but not MUSPI, so it cannot run at highest performance.

Like this thesis, the work of Jun Doi [25] considered multiple data layouts for spinor and gauge field. He does, however, only consider fullspinors and also does not interleave them in pairs of two. But he puts more work in optimizing the communication by dividing the communication for each dimension (Section 6.3.8) and applying a custom ring-mapping to match the 5-dimensional torus of the hardware to a 4-dimensional torus. The performance reaches up to 66 GFlop/s per node (32% of theoretical peak) with a 16x12x8x8 subvolume per node. The kernel is used by IroIro++ [26] from the JLQCD collaboration. For MUSPI communication and threading it uses libraries called BGNET and BGQThreads. Moreover, it is also able to use the BFM and Bagel output.

Another program using the SciDAC stack is QLUA[1]. It uses the scripting language Lua for the high-level logic that invokes the more performance-relevant algorithms of the SciDAC libraries.

The Budapest-Marseille-Wuppertal Collaboration[2] developed their own Lattice QCD software they call *dynQCD*[3]. Unfortunately, the collaboration did not publish articles on the software itself, but Stefan Krieg – the responsible for dynQCD's Blue Gene/Q optimization – held many presentations about his effort. They claim a performance of 40% of peak.

The European Twisted Mass Collaboration (ETMC) also optimized tmLQCD for Blue Gene/Q, but had to be compatible with the rest of the source. This means for instance, the data organization of spinor field could not be changed. Especially physical sites (respectively virtual nodes) are not used.

Heavy optimization of Lattice QCD code is not limited to the Blue Gene/Q. Other platform may require similar, or even identical, adaptations. Most of the software mentioned before are also optimized for Intel (and AMD) x86/x64 processors by using kernels with SSE(2/3/4) instructions, or even the newer AVX(2) instructions.

Intel published an article [27] on how they optimized Hopping Matrix for the Xeon Phi accelerator. They use multiple levels of blocks, for vectorization with multiple SIMD widths, T-dimension slices, per-core tiles that fit into the L2 caches, per-NUMA domain tiles and of course per-node subvolumes that communicate to each other using QMP. The article compares the Xeon Phi accelerators against NVIDIA GPUs that run QUDA.

QUDA[4] [28] like Bagel can be considered part of the SciDAC stack used by Chroma, CPS, MILC or QLUA. It is a library for Lattice QCD kernel for GPUs written in NVIDIA's CUDA framework.

[1] `https://usqcd.lns.mit.edu/w/index.php/QLUA`
[2] `http://www.bmw.uni-wuppertal.de`
[3] `http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/dynQCD/_node.html`
[4] `https://lattice.github.io/quda/`

Part II

Representing Memory Layout in the Polyhedral Model

7 The Polyhedral Model

This chapter gives a brief introduction of the theory of loop scheduling at compile-time. Leslie Lamport [29, 30] maybe was the first who applied linear algebra to loops[1]. Most of the contents of this chapter is based on the work of Paul Feautrier [32, 33, 34, 35]. Today, this framework of program transformation is often called the polyhedral model.

This chapter does not give a complete overview on the topic and includes simplifications. A good book for a more thorough discussion on the theory of integer *polyhedra* is "Theory of linear and integer programming" by Alexander Schrijver [36]. For more details on the polyhedral model, the book "Scheduling and Automatic Parallelization" by Alain Darte, Yves Robert and Frédéric Vivien [37] is a good choice.

7.1 Polyhedra

The mathematical notation of polyhedra in this section is inspired by *ISL* (Integer Set Library) [38]. ISL is the component used by Molly that implements the polyhedral math. The main difference from the notation used by Schrijver [36] is that its coordinates are divided into tuples, something very useful for practical uses.

An n-dimensional polyhedron P is a subset of the vector space $\mathbb{R}^n$ that can be described by conditions for a vector $\vec{v} \in \mathbb{R}^n$ to be an element of P.

A *linear hyperplane* is an $n-1$ dimensional subset of $\mathbb{R}^n$ whose points v fulfill the scalar product equation $\vec{a} \cdot \vec{v} = 0$, where $\vec{a} \neq \vec{0}$ is specific to the hyperplane, i.e. the set $\{\vec{v} \mid \vec{a} \cdot \vec{v} = 0\} \subset \mathbb{R}^n$.

While a linear hyperplane always contains the origin $\vec{0}$ ($\vec{a} \cdot \vec{0} = 0$ for every $\vec{a}$), an affine hyperplane is shifted from it by an offsets, i.e. the scalar product does not necessarily evaluate to zero. Therefore the set $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c = 0\} \subset \mathbb{R}^n$ is an affine hyperplane with parameters $\vec{a} \in \mathbb{R}^n$ and $c \in \mathbb{R}$. Every linear hyperplane is also affine, just with $c = 0$.

Every hyperplane, whether affine or linear, divides the entire space $\mathbb{R}^n$ into three disjoint subspaces. The two spaces on either side of the hyperplane and the hyperplane itself. The sides can be described by the sets $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c > 0\}$ and $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c < 0\}$, which are called *halfspaces*. The complement spaces and therefore the inclusive sets $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c \geq 0\}$ and $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c \leq 0\}$ are halfspaces as well. Any halfspace can be represented using either the $>$ or the $\geq$-form by negating $\vec{a}$ and c of the complement.

[1] "Compass (Massachusetts Computer Associates) had a contract to write the *Fortran* compiler for the Illiac-IV computer, an array computer with 64 processors that all operated in lock-step on a single instruction stream. I developed the theory and associated algorithms for executing sequential DO loops in parallel on an array computer that were used by the compiler. The theory is pretty straightforward. The creativity lay in the proper mathematical formulation of the problem. Today, it would be considered a pretty elementary piece of work. But in those days, we were not as adept at applying math to programming problems. Indeed, when I wrote up a complete description of my work for my colleagues at Compass, they seemed to treat it as a sacred text, requiring spiritual enlightenment to interpret the occult mysteries of linear algebra." (*Leslie Lamport*) [31]

A polyhedron is the intersection of finitely many $\geq$-halfspaces. Let m be the number of intersecting halfspaces, which may also be called *constraints* in this context. Note that the $\geq$-hyperplanes are always closed sets, therefore a polyhedron is as well a closed set. If we interpret the $\vec{a}_k$-vectors of the halfspaces as rows of a matrix $A \in \mathbb{R}^{m \times n}$ and the constants c as a vector $\vec{c} \in \mathbb{R}^m$, then the polyhedron can be represented by

$$P_{\mathbb{R}}(A, \vec{c}) = \bigcap_{k=1}^{m} \{\vec{v} \in \mathbb{R}^n \mid \vec{a}_k \cdot \vec{v} + c_k \geq 0\} = \{\vec{v} \mid A\vec{v} + \vec{c} \geq \vec{0}\} \subseteq \mathbb{R}^n \tag{7.1}$$

Polyhedra may also contain equality constraints, i.e. intersections of hyperplanes (and other halfspaces). The hyperplane $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c = 0\}$ is the same set as the intersection of the two halfspaces $\{\vec{v} \mid \vec{a} \cdot \vec{v} + c \geq 0\}$ and $\{\vec{v} \mid -\vec{a} \cdot \vec{v} - c \geq 0\}$, already representable by Equation (7.1). By construction, a polyhedron is always convex.

A *polytope* is a polyhedron which is bounded. That is, there are vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$ such that the box $\{\vec{v} \mid \forall k \in \{1..n\} : a_k \leq v_k \leq b_k\}$ is a superset of the polyhedron. The smallest such rectangle is the minimal bounding box.

In this thesis we are only interested in points with integer coordinates. For-loops only iterate over integers and array indices are integer as well. Hence, we introduce $\mathbb{Z}$-*polyhedra* by replacing any real-valued variable by integer variables.

$$\begin{aligned} P_{\mathbb{Z}}(A, \vec{c}) &= \bigcap_{k=1}^{m} \{\vec{v} \in \mathbb{Z}^n \mid \vec{a}_k \cdot \vec{v} + c_k \geq 0\} \\ &= \{\vec{v} \in \mathbb{Z}^n \mid A\vec{v} + \vec{c} \geq \vec{0}\} \subseteq \mathbb{Z}^n \text{ with } A \in \mathbb{Z}^{m \times n}, \vec{c} \in \mathbb{Z}^m \end{aligned} \tag{7.2}$$

A $\mathbb{Z}$-*polytope* is a bounded $\mathbb{Z}$-polyhedron, i.e. there is a bounding box which is a superset. Note that an alternative definition of a $\mathbb{Z}$-polyhedron is the intersection of a polyhedron with the integer lattice $\mathbb{Z}^n$, which is not equivalent. Any rational factor in the vector $\vec{a}$ or c of a constraint can be resolved by multiplying $\vec{a}$ and c with the denominator until we get integer values, but this is not possible with irrational numbers. For instance, the polyhedron and halfspace $\{\vec{v} \mid (1, \pi)^T \cdot \vec{v} \geq 0\} \cap \mathbb{Z}^2$ can only approximated using Equation (7.2). Schrijver [36] uses rational numbers in the definition of polyhedra to avoid this mismatch.

Only $\mathbb{Z}$-Polyhedra are useful in the polyhedral model. We may use the term polyhedron instead of the more specific $\mathbb{Z}$-polyhedron in the following of this thesis. The type that represents a $\mathbb{Z}$-polyhedron in ISL is `isl_basic_set`.

7.1.1 Dimensions with Special Meanings

Equation (7.2) defines a polyhedron as a set of vectors. The coordinates of a vector $\vec{v}$ do not differ in semantics. For our application the different coordinates may have different uses, defined at different contexts and with different qualifiers.

7.1.1.1 Constant Dimension

In some sense, the offset c of a hyperplane is always multiplied with a special element of $\vec{v}$ that is always 1 in a scalar product and therefore becomes a part of the hyperplane-specific vector $\vec{a}$. The first row of the matrix A become the coefficients of the constant one. The definition of a $\mathbb{Z}$-polyhedron changes to

$$\begin{aligned} P_{\mathbb{Z}}(A, \vec{c}) &= \left\{ \vec{v} \in \mathbb{Z}^n \;\middle|\; \begin{pmatrix} \vec{c} & A \end{pmatrix} \begin{pmatrix} 1 \\ \vec{v} \end{pmatrix} \geq \vec{0} \right\} \subseteq \mathbb{Z}^n \\ &= \left\{ \vec{v} \;\middle|\; A\vec{v} + \vec{c} \geq \vec{0} \right\} \text{ with } \begin{pmatrix} \vec{c} & A \end{pmatrix} \in \mathbb{Z}^{m \times (1+n)} . \end{aligned}$$

While this does not change any semantics, understanding $\vec{c}$ as a dimension reduces the amount of code for handling $\vec{c}$ in an implementation.

7.1.1.2 Existentially Quantified Dimensions

Some dimensions of $\vec{v}$ can be projected out such that the resulting vector set has less dimensions than the underlying polyhedron. A vector is element of that vector set only if there exists an extension such that the lengthened vector is element of the polyhedron. Formally:

$$\begin{aligned} P_{\mathbb{Z}}(A, A_e, \vec{c}) &= \left\{ \vec{v} \in \mathbb{Z}^n \;\middle|\; \exists \vec{v}_e \in \mathbb{Z}^{n_e} : \begin{pmatrix} A & A_e \end{pmatrix} \begin{pmatrix} \vec{v} \\ \vec{v}_e \end{pmatrix} + \vec{c} \geq \vec{0} \right\} \\ &= \left\{ \vec{v} \;\middle|\; A\vec{v} + A_e\vec{v}_e + \vec{c} \geq \vec{0} \right\} \text{ with } A \in \mathbb{Z}^{m \times n}, A_e \in \mathbb{Z}^{m \times n_e}, \vec{c} \in \mathbb{Z}^m \end{aligned}$$

This vector set can also be understood as a polyhedron of $n + n_e$ dimensions where n_e coordinates of each vector in the polyhedron are projected out.

Such a vector set in general is not convex anymore in any sense of convexity that might be applicable to $\mathbb{Z}$-polyhedra. For instance, the set $\{i \in \mathbb{Z} \mid \exists j \in \mathbb{Z} : i = 2j\}$, the set of even numbers, contains the elements 0 and 2, but not 1. For polyhedra in real vector spaces it is still convex. In the example j may take the value $1/2$. We still refer to the set as a $\mathbb{Z}$-polyhedron.

7.1.1.3 Parametric Dimensions

We can add parameters to a polyhedron to represent a family of polyhedra. A parameter vector $\vec{p} \in \mathbb{Z}^{n_p}$ fixes some of the coordinates in the backing $\mathbb{Z}$-polyhedron while the remaining free coordinates describe the vector set. Feautrier [33] uses the term *structure parameters*.

The distinction between parameter coordinates and vector set coordinates are purely conventional by how they are used. For instance, a parameter p might describe the length of an interval $0 \leq v \leq p$. Here, v is the set of values in that parametrized interval, but the backing polyhedron $\{(p, v) \mid v \geq 0, p - v \geq 0\}$ just describes a set of possible tuples (p, v) that do not violate the condition.

Formally, the vector set is parametrized with a parameter vector $\vec{p}$ which is merged into the condition with an offset $A_p\vec{p}$:

$$\begin{aligned} P_{\mathbb{Z},\vec{p}}(A_p, A, \vec{c}) =& \left\{ \vec{v} \in \mathbb{Z}^n \;\middle|\; \begin{pmatrix} A_p & A \end{pmatrix} \begin{pmatrix} \vec{p} \\ \vec{v} \end{pmatrix} + \vec{c} \geq \vec{0} \right\} \\ =& \left\{ \vec{v} \in \mathbb{Z}^n \;\middle|\; A_p\vec{p} + A\vec{v} + \vec{c} \geq \vec{0} \right\} \subseteq \mathbb{Z}^n \\ & \text{with } A_p \in \mathbb{Z}^{m \times n_p}, A \in \mathbb{Z}^{m \times n}, \vec{c} \in \mathbb{Z}^m \end{aligned} \tag{7.3}$$

Coordinates of the parameter vector have names and/or identifiers in ISL. This is to distinguish and find the parameters with the same meaning in different polyhedra. This is in contrast to other dimensions which are identified by their position. For instance, let there be two polyhedra $P1_{n,m} \subseteq \mathbb{Z}^3$ and $P2_{m,n,o} \subseteq \mathbb{Z}^3$. The parameter m and n are matched to each other although their order is reversed. Only $P2$ is also dependent on a parameter o, but an operation combining both polyhedra (union, intersection, etc.) is probably also dependent on o. Both polyhedra are subsets of $\mathbb{Z}^3$, no realignment of coordinates takes place.

7.1.1.4 Relationships Dimensions

Instead of interpreting an element (x_0, x_1, y_0, y_1) in some vector set $\subseteq \mathbb{Z}^4$ as a single tuple/vector, it may also be interpreted as a pair vectors (x_0, x_1) and (y_0, y_1) which relate to each other in

some sense. For instance, one vector might be a reflection of the other. We want to represent such relations of vectors and tuples using polyhedra.

$$\begin{aligned} P_{\mathbb{Z}}(A_x, A_y, \vec{c}) &= \left\{ (\vec{v}_x, \vec{v}_y) \;\middle|\; \begin{pmatrix} A_x & A_y \end{pmatrix} \begin{pmatrix} \vec{v}_x \\ \vec{v}_y \end{pmatrix} + \vec{c} \geq \vec{0} \right\} \\ &= \{ (\vec{v}_x, \vec{v}_y) \mid A_x \vec{v}_x + A_y \vec{v}_y + \vec{c} \geq 0 \} \subseteq \mathbb{Z}^{n_x} \times \mathbb{Z}^{n_y} \\ &\quad \text{with } A_x \in \mathbb{Z}^{m \times n_x}, A_y \in \mathbb{Z}^{m \times n_y}, \vec{c} \in \mathbb{Z}^m \end{aligned} \tag{7.4}$$

The $\vec{v}_x$ part of the relation is called the *domain*, the $\vec{v}_y$ side is the range. A pair $(\vec{v}_x, \vec{v}_y)$ that is member of the set is said to be related to each other in the sense of an assigned semantic meaning. The individual coordinates of the domain are called the *input* dimensions. Likewise, the range coordinates are *output* dimensions.

If the domain has zero dimensions ($v_x \in \mathbb{Z}^0$) then there is either one element in the domain (which is "()") or otherwise the relation is empty; no element in the range that can be mapped from. The relation can then be interpreted as a vector set: all the elements that map from "()". The result is the same as if $\vec{v}_x$ and A_x did not exist in Equation (7.4).

Let R be the relation of vectors in $\mathbb{Z}^2$ that are mirrors of each other around the x_1-axis. Expressed as a function, this is

$$R : \mathbb{Z}^2 \to \mathbb{Z}^2, \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \mapsto \begin{pmatrix} -x_0 \\ x_1 \end{pmatrix}$$

In case that the relation is not uniquely defined, R can be modeled as a function to the set of elements that relate to the argument, i.e. $R : \mathbb{Z}^2 \to 2^{\mathbb{Z}^2}$.

The same function can be expressed as a relation

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} R \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \;\Leftrightarrow\; x_0 = -y_0 \wedge x_1 = y_1 \;,$$

as a set

$$R \in \mathbb{Z}^2 \times \mathbb{Z}^2, R = \{ (\vec{x}, \vec{y}) \mid x_0 = -y_0, x_1 = y_1 \} \;,$$

or as a polyhedron with 4 constraints:

$$R = P \left(\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}, \vec{0} \right) = \left\{ (\vec{x}, \vec{y}) \;\middle|\; \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \vec{y} \geq \vec{0} \right\}$$

We allow "abusing" the notation by not explicitly distinguishing between those representations and even use them alternatively depending on the context. For instance, the symbol $\mapsto$ is used for mappings in general, not just functional (one-to-one) mappings.

Note that this is merely a re-interpretation of vectors in a polyhedral set. In fact, we can reinterpret any relation as defined in Equation (7.4) as a set where we interpret the first n_x dimensions belonging to the mapping's domain, and the remaining n_y dimensions as belonging to the range.

The ISL type for polyhedral relations is `isl_basic_map`.

7.1.1.5 Nested Relations

Relations may relate more than two tuples to each other. One way to do this is to add another matrix A_z to the polyhedron's definition. Unfortunately, ISL does not support this, but it can nest relations, that is, we can build a relation of relations.

Let's say, we have a relation of Pythagorean triples – three numbers that match $x^2+y^2=z^2$. Unfortunately, the complete set cannot be represented as a $\mathbb{Z}$-polyhedron, so we use a subset

$$\{(x,y,z) \mid \exists w : w \geq 1, x = 3w, y = 4w, z = 5w\}$$

or in explicit notation

$$\{(3,4,5),(6,8,10),(9,12,15),\ldots\} .$$

This is, of course, already a polyhedral set, but we may want to express it as a relation. Using nested relations, the same set can be represented as

$$R : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}), \quad x \mapsto (y \mapsto z)$$

or

$$R : (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}, \quad (x \mapsto y) \mapsto z .$$

The first version maps an integer to a tuple that in itself can be interpreted as a mapping. The second version maps a tuple, again interpreted as a mapping, to an integer. We may not need to make a distinction between the two and treat the mapping operator associatively. Correspondingly, we may again "abuse" notation and just omit the parenthesis.

$$R : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}, \quad x \mapsto y \mapsto z$$

7.1.1.6 Combining all Dimension Types

Previously all the different types of dimensions have been defined independently, but can of course be put into a single definition.

$$\begin{aligned} P_{\mathbb{Z},\vec{p}}(\vec{c}, A_p, A_x, A_y, A_e) =& \left\{ (\vec{v}_x, \vec{v}_y) \;\middle|\; \exists \vec{v}_e \in \mathbb{Z}^{n_e} \begin{pmatrix} \vec{c} & A_p & A_x & A_y & A_e \end{pmatrix} \begin{pmatrix} 1 \\ \vec{p} \\ \vec{v}_x \\ \vec{v}_y \\ \vec{v}_e \end{pmatrix} \geq \vec{0} \right\} \\ =& \{(\vec{x},\vec{y}) \mid \exists \vec{v}_e \in \mathbb{Z}^{n_e} : \vec{c} + A_p\vec{p} + A_x\vec{x} + A_y\vec{y} + A_e\vec{v}_e \geq 0\} \\ & \text{with } \vec{c} \in \mathbb{Z}^m, A_p \in \mathbb{Z}^{m \times n_p}, A_x \in \mathbb{Z}^{m \times n_x}, A_y \in \mathbb{Z}^{m \times n_y}, A_e \in \mathbb{Z}^{m \times n_e} \end{aligned}$$

Any use-case of polyhedral sets and relations in this thesis is covered by this definition. It is also the internal representation used by ISL.

7.1.2 Affine Functions

In some contexts it is not necessary to have a many-to-many relation that polyhedral maps represents. One-to-one mappings (functions) are more efficiently represented by the rule that maps from the domain to the range value it maps to. A polyhedral one-to-one map consists of equalities only, i.e. a $\leq$ and a $\geq$ constraint.

Instead of

$$R = \left\{ (\vec{v}_x, \vec{v}_y) \;\middle|\; A_x\vec{v}_x + A_y\vec{v}_y + \vec{c} = \vec{0} \right\}$$

the same relation can be represented a

$$R : \mathbb{Z}^{n_x} \to \mathbb{Z}^{n_y}, \quad \vec{v}_x \mapsto A_x^{-1}(-\vec{c} - A_y\vec{v}_y) \ .$$

By the requirement that R is a one-to-one relation, A_x is left-invertable, i.e. has full rank. In ISL, such functions are supported using the data types `isl_aff` and `isl_multi_aff`.

7.1.3 Named Sets and Relations

ISL allows tuples to carry an identifier with a name. A tuple in ISL's sense are the vectors $\vec{v}_x$ and $\vec{v}_y$ in polyhedral relations and plain $\vec{v}$ if it is a polyhedral set. Tuples with different identifiers are considered incompatible even if the number of coordinates is equal.

This is useful if the tuples represent different spaces even if their share the same dimensionality. For instance, let $P1$ be a set of neutron numbers of carbon isotopes (e.g. 6 to 8)[1] and $P2$ the number of neutrons in oxygen isotopes (e.g. 8 to 10). Operations like intersection and union on the raw numbers does not make a lot of sense on $P1$ and $P2$, it is like comparing apples to oranges. It does, however, make sense to build a set that keep the element names. The intersection of the example $P1$ and $P2$ would be empty and the union would be

$$\{(C,6),(C,7),(C,8),(O,8),(O,9),(O,10)\} \ .$$

ISL uses the types `isl_union_set` and `isl_union_map` for this purpose. Here, the element identifiers (C for carbon and O for oxygen) have been added to the tuple as discriminator, but ISL does not consider them to be a separate dimension.

7.1.4 Operations on Polyhedral Sets and Relations

Just representing sets is useless without any operations applied on them. Some of them reveal interesting aspects of polyhedra.

7.1.4.1 Union

The union of two polyhedral sets generally is not representable by a single polyhedron. A polyhedron is always convex, but the union of two polyhedra is not necessarily convex and non-convex sets cannot be represented with a single polyhedron. In order to make the union operation possible, an implementation has to manage a list of polyhedra. The set it represents is the union of all polyhedra in this list.

The union operation is so common that ISL's main types are such lists. The ISL-implementation's data type is `isl_set` which is a list of `isl_basic_set`s. Analogously, `isl_map` is a list of `isl_basic_map`s. The naming suggests that `isl_basic_set` and `isl_basic_map` are more specialized vector sets, but actually `isl_set` and `isl_map` are the types with the extra feature of having a union operation defined on it. We follow this understanding by implicitly assuming lists of polyhedra when necessary even without mentioning it. As a result any finite vector set can be represented by polyhedral sets, in worst case with one $\mathbb{Z}$-polyhedron for each element.

The practical difference in lists of $\mathbb{Z}$-polyhedra and explicit lists of all vectors it contains is the computational cost. Most operations are performed much quicker on a polyhedron than on a list of all vectors it contains. This is not necessarily true anymore if a polyhedral set is itself built-up from many $\mathbb{Z}$-polyhedra.

[1] This is example; resemblance or non-resemblance to reality not intended

The union operation becomes a simple concatenation of the polyhedron lists. The cost is that any further operation must be applied on all polyhedra in the list. The polyhedron list can be optimized when considered worthwhile by removing empty polyhedra, coalescing them if their union is polyhedral and removing overlapping sections.

Also, $\mathbb{Z}$-polyhedra can represent sets of infinitely many vectors, which is not possible in an explicit representation. Within the Molly framework, only structure parameter can be unbounded and this may take infinitely many values.

Lets consider Pythagorean triples again. The set of triples can be closer approximated by adding more primitive cases the set; the example below contains two primitive sets. The complete set of Pythagorean triples cannot be represented using this construction because there are infinitely many Pythagorean primitives.

$$\begin{aligned} & \{(x,y,z) \mid \exists w : w \geq 1, x = 3w, y = 4w, z = 5w\} \\ \cup & \{(x,y,z) \mid \exists w : w \geq 1, x = 5w, y = 12w, z = 13w\} \\ = \quad & \{(x,y,z) \mid \exists w : w \geq 1 \wedge ((x = 3w \wedge y = 4w \wedge z = 5w) \vee (x = 5w \wedge y = 12w \wedge z = 13w))\} \end{aligned}$$

7.1.4.2 Intersection

Intersecting two polyhedra is as simple as concatenation of all constraints from both. Constraints are conjunctive, therefore the resulting polyhedra contains only the elements that are in both argument sets.

Problems start when both polyhedral sets are lists of multiple $\mathbb{Z}$-polyhedra. In this case the result is itself a list of polyhedra, one for each combination of two basic $\mathbb{Z}$-polyhedra from each argument. Say, $P1$ consists of a list of n_1 $\mathbb{Z}$-polyhedra and $P2$ has n_2 $\mathbb{Z}$-polyhedra. Then the intersection has up to $n_1 n_2$ $\mathbb{Z}$-polyhedra, i.e. a quadratic growth.

Unfortunately, many higher-level operations have to intersect vector sets for every dimension, like the lexicographical comparison. The result is an exponential blowup in the number of dimensions. Lexicographic comparison for instance is used to find statement dependencies, therefore essential to any optimization based on the polyhedral model.

Most often the number of $\mathbb{Z}$-polyhedra in the set can be reduced because many of them are empty. Still some care is necessary to avoid excessive computational cost.

7.1.4.3 Subtraction

Removing elements from a polyhedral set is another operation that can drastically increase the number of basic $\mathbb{Z}$-polyhedra in the set. For instance, if a single point from the middle of a square is removed, 4 surrounding $\mathbb{Z}$-polyhedra remain. In practice, fortunately, this is less of a concern than the intersection.

7.1.4.4 Projection

(Out-)projection is the removal of dimensions from the range. Constraints that involve coordinates of the dimensions projected must still be considered. An element remains in the set if the vector can be completed with coordinates for the removed dimensions such that it becomes element in the set before the projection. In other words, the out-projected dimensions become existentially quantified.

Therefore the set

$$\left\{ \begin{pmatrix} \vec{v}_x \\ \vec{v}_e \end{pmatrix} \;\middle|\; A \begin{pmatrix} \vec{v}_x \\ \vec{v}_e \end{pmatrix} + \vec{c} \geq \vec{0} \right\}$$

becomes

$$\left\{ \vec{v}_x \;\middle|\; \exists \vec{v}_e : A \begin{pmatrix} \vec{v}_x \\ \vec{v}_e \end{pmatrix} + \vec{c} \geq \vec{0} \right\}$$

when the dimensions $\vec{v}_e$ are projected out.

7.1.4.5 Applying Maps

One of the more common operation in the polyhedral model is to find the image of the mapping where the input is a vector set itself.

$$P \subset \mathbb{Z}^n, R : \mathbb{Z}^n \to \mathbb{Z}^m$$
$$R(P) \coloneqq \{v_x \mid \exists v_y \in P : (v_x, v_y) \in R\}$$

Another variant is to apply a relation on the domain or range of another relation, or concatenation of maps.

$$R_1 : \mathbb{Z}^{n_x} \to \mathbb{Z}^{n_y}, R_2 : \mathbb{Z}^{n_y} \to \mathbb{Z}^{n_z}$$
$$R_2 \circ R_1 \coloneqq \{(v_x, v_z) \mid \exists v_y : (v_x, v_y) \in R_1 \vee (v_y, v_z) \in R_2\} \subseteq \mathbb{Z}^{n_x} \times \mathbb{Z}^{n_z}$$

Both operations can easily be implemented by identifying dimensions of the first set or relation with dimensions of the second relation. Dimensions not appearing in the resulting relation are projected away.

7.2 Static Control Parts

The more properties of some source code is known at compile-time the more the compiler can do to optimize it. The control flow is crucial here, many optimizations cannot be applied if the compiler does not know what next piece of code is executed or what has been executed before. Functions adhere a calling convention such that both pieces of code, the function body and the function caller execute without the compiler knowing what the other code is. Because of this black box view, the compiler cannot apply some optimizations without the risk of breaking the program's semantics.

Easiest to optimize are parts of a function that always execute sequentially, without any branching, loops or conditionals. In compiler jargon such maximal sequential parts are called *basic blocks*.

Loops with known number of iterations, as in Listing 7.1, can be unrolled. This is unpractical in many cases. In the example this means 100 times the same line S1 and will probably execute slower than the original loop because the CPU needs to decode all the instructions, and thrashes other code from the instruction cache. However, such for-loops can be perfectly described using linear algebra, which is the core of the polyhedral model.

```
    for (int i = 0; i < 100; i+=1) {
S1:   array[i] = array[i] + 1;
    }
```

Listing 7.1: Example of a Static Control Part

Only the line marked with S1 has some effect. In the polyhedral model the for-statement itself is considered boilerplate code that describes the execution of its body. Only statements are objects of analysis and transformation. Let's say S1 is such a statement. What do we know about it?

- It executes 100 times, every time parametrized with a different i. An execution of a statement with an specific i is called an *instance*.
- i is a non-negative integer and at most 99, called the *domain*[1] of S1.
- The instances of S1 are executed in a specific order. Given to instances, knowing their loop counter values i_1 and i_2 is enough to know which of both instances is executed first, namely the instance with i_i is executed first iff $i_1 < i_2$ because the loop is executed with increasing counters. This is the "executes before" relation between statement instances.
- An instance with loop counter value i accesses the `array` at index i, for reading and writing.

The domain and the executes-before relation are both $\mathbb{Z}$-polyhedra. The domain is

$$\left\{ i \;\middle|\; \begin{pmatrix} 0 \\ 99 \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \end{pmatrix} i \geq 0 \right\} \subseteq \mathbb{Z}^1$$

and the "executes before" relation is

$$\left\{ (i_1, i_2) \;\middle|\; \begin{pmatrix} -1 & 1 \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} - 1 \geq 0 \right\} \subseteq \mathbb{Z}^1 \to \mathbb{Z}^1$$

In other cases, such as Listing 7.2, the code flow is highly predictable, but not strictly static because it depends on the value of the variable `N` whose value is only known at runtime. This case is more interesting because it is more common and complete unrolling is not possible.

```
  for (int i = 0; i < N; i+=1) {
S1:   array[i] = array[i] + 1;
  }
```

Listing 7.2: Example of a SCoP with structure parameter `N`

With the extensions from Equation (7.3) it is again possible to handle this: N is a parameter of the loop and used as a parameter of all involved polyhedra. As a result, the value of such captures may not change during the execution of the part, i.e. it is *Static Control Part* (SCoP)-invariant.

Many different symbols, letters, function names, etc. are use for the mathematical description of a SCoP. There is on overview of the notation used in this thesis in Appendix B on Page 175 which the reader may use to look up any symbol's meaning when working through this document.

7.2.1 Definition

We formally define a SCoP in two definitions. The first recursively defines code regions, the second the conditions a code region must fulfill to be called a SCoP.

Definition 7.1 (Code Region)
We call contiguous part of a function's source code a *code region* if it matches one of the following forms.

1. A sequence of sequentially executed instructions, called statements in the polyhedral model. similar to a basic block in compiler engineering, but without requirement to be maximal (see next rule).

[1] Not related to the domain of a polyhedral relation described in Section 7.1.1.4

2. The sequential concatenation of code regions is a code region again.

3. Conditionals of the form

```
if (C >= 0) {
  S
}
```

   are code regions if `S` is also a code region.

4. Any for-loop that has the form

```
for (int i = begin; i < end; i += step) {
  S
}
```

   where `S` is a code region; `step` is an integer constant. `i` is called the loop induction variable.

The set Ω is defined to contain all statements (Rule 1) of a code region.

Definition 7.2 (Static Control Part)
Let R be a code region. The structure parameters of R are the variables in R that never change their value in any execution of R. In any of R's sub-regions, the domain variables are the induction variables of the loops the sub-region is nested in. The set of domain variables if R itself is empty. A domain-affine expression is an affine expression (polynomial of degree one) that depends only on domain variables and structure parameters. Then R is a *Static Control Part* (SCoP) if it fulfills the following conditions.

1. Conditional expressions C (third form) are domain-affine expressions.

2. The placeholders `begin` and `end` in for-loops (point 4) are domain-affine expressions.

3. Statements are not allowed to have any side-effects, except memory accesses of the types below.

   (a) Read accesses to induction variables

   (b) Read accesses to memory that never changes during the SCoP's execution, i.e. the structure parameters

   (c) Read and write accesses to memory elements. A memory element is either a scalar variable or an array element. If it is an array element, its subscript(s) must be domain-affine expressions.

4. There is no aliasing, i.e. different variables and arrays always refer to disjoint memory regions.

This is not the only possible definition. Some code can also be transformed to match the definition. For instance, a conditional statement

```
if (C == 0) {
  S
}
```

can be transformed to

```
if (C >= 0) {
  if (-C >= 0) {
    S
  }
}
```

that matches rule 3 twice. Second, one can think of many extensions [39]. An extended SCoP definition may allow while-loop, conditionals with arbitrary conditions, array index expressions that are not affine, and much more. But such extensions either require more effort to handle them (computational and/or amount of code) or require pessimistic analysis approximation which restrict the transformations applicable. Neither kind is topic in this thesis.

7.2.2 Describing a SCoP

SCoPs are complex objects, but compared to general source code all the rules ensure that some of its properties are determinable. According to Rice's theorem, non-trivial properties of general programs are undecidable, for instance, whether it ever halts. SCoPs always terminate due to the fact that loop iteration counts are predetermined before the loop is entered. There is no halting problem on SCoPs. Welcome to the world of predictable program behavior.

There are quite some properties SCoPs have. This section tries to introduce the ones relevant to this thesis.

Every statement $\mathrm{S} \in \Omega$ has a domain $\mathcal{D}_{\mathrm{S}}$, a vector set. Every element of this set describes an iteration of the for-loops the statement is nested in. Since the statement may use the loop's counter variables, the statement's effect is different in every iteration. Such an iteration is an *instance* of the statement, mathematically described using a tuple $(S, \vec{i})$ associating a statement and with the values of the induction variables it depends on. The vectors of the domain contain the values of the loop counters at each iterations. An example.

```
for (int i = 0; i < 100; i += 2) {
  for (int j = -5; j < 6; j += 1) {
    S1
  }
}
```

The domain of `S1` is $\{(i,j) \mid \exists z \in \mathbb{Z} : 0 \leq i < 100, i = 2z, -5 \leq j < 6\}$, or using matrices:

$$\mathcal{D}_{S1} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \exists z \in \mathbb{Z}^1 : \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -2 \\ 2 \end{pmatrix} z + \begin{pmatrix} 0 \\ 99 \\ 5 \\ 5 \\ 0 \\ 0 \end{pmatrix} \geq \vec{0} \right\}$$

The total number of instances is the number of elements in the domain. S1 has $50 \cdot 11 = 550$ instances, i.e. is executed that many times during the execution of the example code. The set $\mathcal{D}$ (without subscript) contains all instances of all statements in Ω.

For a complete description of a SCoP one needs to define the order in which the statements execute. The order is relevant for every statement that has effects, like writing to memory. A statement without effect is useless, therefore it can be assumed that any statement has some effect. The order is defined by a non-strict total order relation $<_{\text{exec}}$ between instances. A tuple $((S, \vec{i}), (R, \vec{j}))$ is member of $<_{\text{exec}}$ if during the sequential execution of the SCoP, the instance $(S, \vec{i})$ is executed before $(R, \vec{j})$. The execution order of sequential code is uniquely defined, therefore is $<_{\text{exec}}$.

$<_{\text{exec}}$ can be derived from another representation of execution order, a scatter function θ_{S}. A scatter function maps a statement's instance to its time of execution in an abstract way. Instead of time units of seconds, minutes and hours, we use artificial integer time units. We use the symbol $\ominus$ for the space of times, and $N_{\ominus}$ for the number of time units. In contrast to seconds, minutes and hours, artificial time cannot converted between each other.

The scatter function can be directly derived from the loop hierarchy as follows. Every sequential region (rule 2) and loop (rule 4) is assigned a level according to their nesting. The elements of sequential regions are assigned a number matching the position in the region. The first element gets assigned the number zero, followed be the next element with number 1, etc. The elements with lower numbers are executed before the ones with higher numbers.

Without loss of generality we assume that loop induction variables begin with value zero and increase at every iteration. If it does not, the code can be transformed such that it does without changing the semantics (subtracting `begin` at all uses, including `end`; also negating if `step` is negative). The associated time of the loop's body is the value of the induction variable. Because the induction variable monotonously increases for iterations that execute later, this also fulfills the requirement that $<_{\text{exec}}$ matches the relation less-than relation ($<$) for the associated time.

The per-level-time value are put in a vector with the outermore levels first. If we assume that also a single loop or statement is a sequential region with one element, loop and sequential levels are alternating, beginning with an sequential region. Let l be the number of loops the most nested statement is nested in, then

$$N_{\circlearrowleft} = 2l + 1 \tag{7.5}$$

is the number of required time dimensions. If an element of one level is executed before another, this is necessarily also true for its children, i.e. previous dimensions take priority. This matches the lexicographical ordering the time vectors denoted using $\ll$.

The result is a scatter function that assigns a multidimensional time to each instance. The $<_{\text{exec}}$ relation derives from θ using the equation

$$<_{\text{exec}} = \left\{ ((S, \vec{i}), (R, \vec{j})) \;\middle|\; \theta_S(\vec{i}) \ll \theta_R(\vec{j}) \right\}$$

A scatter function θ that assigns a value to every instance of the SCoP is also called a *schedule*. The schedule that is derived from the source code according to the aforementioned rule is called the source-induced schedule. The source-induced schedule θ reflects the execution order of a sequential program before any transformation is applied.

7.2.2.1 Variables and Arrays

We define $\mathcal{V}$ to be the set of variables that are not structure parameters nor loop inductions variables, i.e. anything that can be written to in statements. Variables can be arrays with multiple elements whereas an element is an unit of memory that is processed atomically, that is, always read and written as a whole[1]. A variable that has just one element is a *scalar*, not an array.

The different elements of an array are referenced to (subscripted) by using an index. Every array A has a set of valid indices $\mathcal{I}_A$ that map to some memory for that element, the *indexset*. Accesses to indices outside the indexset are undefined. The indexset again is a $\mathbb{Z}$-polyhedron and usually, but not necessarily, rectangular. Arrays with indexsets with more than one dimension are multidimensional arrays. Scalars have indexsets with zero dimensions.

The set $\mathcal{E}$ refers to all elements of any variable (array or scalar).

7.2.2.2 Dependencies

In order to change the execution order of the instances, one could simply define a different schedule. The goal, however, should be to not change the semantics or otherwise the SCoP could return a wrong result. What are the conditions that preserve semantics?

[1] What is meant here is how an element in the language semantics, not atomicity in the context of multi-threading

The only effects statements can have are memory accesses (rule 3). For the result of a statement instance to be different, the input must have been different. For the input to be different, the instance (over-)writing the relevant memory location must have been moved behind, or a different write access must have been moved between the instances.

Let λ be the function that tells for every instance which memory location it accesses. We classify accesses into reading and writing accesses. λ_{read} returns the read accesses while λ_{write} returns the write accesses ($\lambda = \lambda_{\text{read}} \cup \lambda_{\text{write}}$). We assume that every memory location is write-accessed at most once in a statement instance, otherwise we have to handle cases of self-dependencies. A *generator* is a statement instance that writes a value to a memory location and a *consumer* is a memory access that reads the value from the same location.

There are three reasons for preserving the order of access (called dependencies or hazards).

Flow: An instance writes a value that is later read by another (read-after-write; RAW)
The reader cannot read the correct value before it has been written

Anti: An instance read a value that is later overwritten by another (write-after-read; WAR)
The value cannot be overwritten before the last flow-dependency of the previous value has executed

Output: Two instances write to the same memory location (write-after-write; WAW)
The content at the location when the SCoP terminates depends on which of the writers executed last

These dependencies are denoted by the relations $<_{\text{flow}}$, $<_{\text{anti}}$ and $<_{\text{output}}$, defined as

$$<_{\text{flow}} = \left\{ ((S,\vec{i}),(R,\vec{j})) \;\middle|\; \lambda_{\text{write}}(S,\vec{i}) \cap \lambda_{\text{read}}(R,\vec{j}) \neq \varnothing, (S,\vec{i}) <_{\text{exec}} (R,\vec{j}) \right\}$$
$$<_{\text{anti}} = \left\{ ((S,\vec{i}),(R,\vec{j})) \;\middle|\; \lambda_{\text{read}}(S,\vec{i}) \cap \lambda_{\text{write}}(R,\vec{j}) \neq \varnothing, (S,\vec{i}) <_{\text{exec}} (R,\vec{j}) \right\}$$
$$<_{\text{output}} = \left\{ ((S,\vec{i}),(R,\vec{j})) \;\middle|\; \lambda_{\text{write}}(S,\vec{i}) \cap \lambda_{\text{write}}(R,\vec{j}) \neq \varnothing, (S,\vec{i}) <_{\text{exec}} (R,\vec{j}) \right\} .$$

Read-after-Read is no dependency because values can be read in any order without affecting the result. The relation of two instances that have to keep the order for any reason is the union of all three types. In addition, it is transitive. Hence, whether two statement are dependent can be determined using the transitive closure of all dependencies.

$$<_{\text{dep}} := (<_{\text{flow}} \cup <_{\text{anti}} \cup <_{\text{output}})^*$$

Using this definition of unchanged semantics, a schedule is *valid* (or *legal*) if the source-induced $<_{\text{exec}}$ is a superset of $<_{\text{dep}}$ constructed using the original schedule. It is said that the schedule does not violate any dependencies.

Using a scatter function, the requirement can be expressed as

$$\forall((S,\vec{i}),(R,\vec{j})) \in <_{\text{dep}}: \theta(S,\vec{i}) \ll \theta(R,\vec{j}) \; . \tag{7.6}$$

Any schedule that fulfills the above condition can be considered valid. Generally, there is no single solution therefore there is a space of legal scatter functions. Out of all valid schedules, an optimizer can chose one according to other conditions that improve the execution speed. Fo name a few, *PLuTo* [40], *LooPo* [41] and *LeTSeE* [42] are such optimizers. They use the fact that the domains and scatter functions can be expressed using $\mathbb{Z}$-polyhedra, and therefore Equation (7.6) is expressible as (piecewise) $\mathbb{Z}$-polyhedron as well. Typical optimization goals are locality of memory access and minimization of the working set.

Every consumer has at least one flow dependence from an instance that wrote to the same memory element, but for each element an instance reads, there is at most one generator from

which the consumed value actually originates (Remember the atomicity condition for writing elements). It is the last executed writing instance of the flow dependencies for that element. The flow dependencies matching this condition are the *direct flow dependencies*. The other flow dependencies can be derived transitively using the output dependencies.

Hence, for every consumer and location, there is an unique generator (except the case when generators are executed in parallel, in which case the result is undefined). For every read element, let's define a *value source location*:

$$\sigma((C,\vec{i}),(A,\vec{k})) = \arg\max_{(G,\vec{j})}{}_{\text{lex}}\left\{\theta(G,\vec{j}) \;\middle|\; (A,\vec{k}) \in \lambda_{\text{write}}(G,\vec{c}) \cap \lambda_{\text{read}}(R,\vec{j}), (G,\vec{j}) <_{\text{exec}} (C,\vec{i})\right\}$$

This is the relevant relation for finding def-use chains in SCoPs, not the flow dependence relation which does not even contain information about which element causes the dependency.

7.2.2.3 Dependence Graphs

Dependencies are inherently transitive. It therefore makes sense to visualize them in a directed graph. There are actually two kinds of graphs for this purpose.

The *generalized dependence graph* [33, p. 6] uses statements (Ω) as nodes. For this reason we may also call it the statement dependence graph. Edges between statements are annotated with a relation between the statement's domain. Two instances are dependent in the direction of the edge if their domains are in the relation annotation. In case of the polyhedral model the annotation is a $\mathbb{Z}$-polyhedral relation. Standard dependence graphs show relations between instructions that are executed once only, not in loops. The generalization comes from the annotations that also describe which instances of statements are dependent.

Statements can be expanded to execute-once instructions by completely unrolling the surrounding loops. It results in the *detailed dependence graph* [33, p. 7] where every statement instance is a node. Nodes are connected according to the $<_{\text{dep}}$ relation between instances. The generalized dependence graph is usually a much more compact description of dependencies because the actual instructions are implicit. The explicit detailed graph (where every node and edge is drawn) may not even exist in case the polyhedral description has parameters or a statement domain is unbounded. The latter yields infinitely many graph nodes. However, even the detailed dependence graph can be described using implicit definitions. Families of graphs are described when using parameters.

7.2.2.4 Code Generation

Once a new schedule has been defined, the SCoP source code needs to be rewritten to reflect this change. The technique takes the existing code of the statements and puts them into new for-loops that obey the new instance ordering. Very often it is also necessary to copy the same statement source code into multiple locations with different loops.

Cloog [43], ISL and Omega [44] are the most used open source projects that are able to perform this task. The algorithm of choice is by Quilleré et al. [45].

7.2.2.5 Shared-Memory Parallelism

The previous section assumed that a SCoP is executed sequentially without parallelism. A few things change when parallelism is allowed. Firstly, $<_{\text{exec}}$ is may not be total order anymore, but a partial order. Instances can be executed at the same time. Another interpretation is that the order of execution is undefined.

For instance, iterations of a loop annotated with OpenMP (`#pragma omp parallel for`) can be executed in parallel, but do not need to. Indeed, a sequential execution is still valid and probably done on single core processors.

In terms of the scatter function this means that multiple instances can be mapped to the same scattering. If a complete loop is supposed to be executed in parallel (or with undefined order), the corresponding level either disappears from the scattering or is just set to zero for all instances. The execution-before relation follows accordingly and the code generator has to recognize this situation. Depending on whether the target compiler supports it, it may generate OpenMP *pragmas*, advise it to use SIMD instructions (`#pragma ivdep` for *Intel C++Compiler* (ICC)) or use some other notion of parallelism.

Dependencies must still be satisfied. Otherwise mapping every instance to a single scatter vector would always be valid. Instances that are dependent must still map to lexicographically positive time-vectors, i.e. the dependent vector is at least one greater in the first coordinate that is different.

7.2.2.6 Virtual Statements

The prologue of a SCoP is a virtual statement executed before any other statement that writes every element of every variable referenced in the SCoP. "Virtual" because it does not exist in the SCoP. It is however used during the computation of the data flow. Since it writes elements it comes up as a generator in the data flow computation. It "generates" a value by loading it from a node's local storage. The prologue's domain is the zero-dimensional point since it is not part of any sub-region, but its only instance is "executed before" ($<_{\text{exec}}$) all other statement instances.

The epilogue is an analogous virtual statement that is executed after all other statements and read-accesses all variables and all their elements. Therefore it will appear as a consumer in any data flow as the last user of a value. Its domain is the zero-dimensional point as well, but every other statement is "executed before" ($<_{\text{exec}}$) the SCoP's epilogue.

As shortcut notation, we use the top symbol $\top$ to denote the prologue and the bottom symbol $\bot$ for the epilogue. $\Omega' \coloneqq \Omega \cup \{\top, \bot\}$ is the set of statements including the virtual nodes.

Because the prologue is the very first and read-access statement in a SCoP, there are no Write-after-Read nor Write-after-Write hazards involving the prologue. Similarly, the epilogue is a write access and also never causes Write-after-Read or Write-after-Write dependencies. Only data flows (Read-after-Write) are possible.

A data flow from the prologue to a non-virtual instance is an *input flow*. It represents the data that is not generated in the SCoP itself, the input of the SCoP. Data flows to the epilogue are values that can be read by code after the SCoP's termination, i.e. the SCoP's output. We call it the SCoP's *output flow*. Direct data flow dependencies from the prologue to the epilogue represent data not changed inside the SCoP (but might be read by statement's in the SCoP). Such dependencies can be ignored.

Prologue and epilogue do not exist in the literature on the polyhedral model, but are introduced in this thesis to generalize the concept of generator and consumer. They ensure that every use of an element has a generator and every write has a consumer. In addition, it allows us to compute the interaction with code outside of the SCoP without introduction of special data flows.

7.3 Example: 2D Jacobi

A 2D Jacobi stencil will serve as a running example. It can also seen as a repetitive blur-filter that averages the 4 neighbor stencil points and itself. Listing 7.3 shows the source code in C++.

The statements S1 to S5 read the stencil points and add the values to a per-stencil sum. Stencil points that exceed the 9×9 volume are skipped. S6 computes the average, depending on how many stencil points have been skipped. The result is stored to the sink array by

```
  #define LX 9
  #define LY 9
  double source[LX][LY];
  double sink[LX][LY];

  for (int i = 0; i < ITERATIONS; i += 1) {
    for (int x = 0; x < LX; x += 1)
      for (int y = 0; y < LY; y += 1) {
S1:     double sum = source[x][y];

        if (x - 1 >= 0)
S2:       sum += source[x - 1][y];

        if (x + 1 < LX)
S3:       sum += source[x + 1][y];

        if (y - 1 >= 0)
S4:       sum += source[x][y - 1];

        if (y + 1 < LY)
S5:       sum += source[x][y + 1];

S6:     double avg = sum / (1 + (x - 1 >= 0 ? 1 : 0) + (x + 1 < LX ? 1 : 0)
                            + (y - 1 >= 0 ? 1 : 0) + (y + 1 < LY ? 1 : 0));

        // Writeback
S7:     sink[x][y] = avg;
      }

    if (i + 1 < ITERATIONS)
      for (coord_t x = 0; x < LX; x += 1)
        for (coord_t y = 0; y < LY; y += 1)
S8:       source[x][y] = sink[x][y];
  }
```

Listing 7.3: Jacobi example code

statement S7. This is repeated `ITERATIONS` times, where `ITERATIONS` is either a constant or a structure parameter.

To execute the stencil on the previous stencil's output, the arrays source and sink must be swapped. Unfortunately pointer swapping is not compatible with the polyhedral model, variables must always refer to the same memory to be predictable. One solution is to copy the stencil with the role of source and sink reverse. In the example code, the output data is moved to the source array value-by-value instead, which yields more compact code. Another solution is to use modulo-2 subscripts to alternate between subfields. It might be more efficient, but complicates the polyhedral sets, making it unsuitable for an example case.

Listing 7.3 fulfills all the requirements of being a SCoP code region starting with the first for-loop with loop counter `i`. We will manually compute all the sets once for this example. The statements are already labeled in the list. Ω' also includes the pro- and epilogue.

$$\Omega = \{\mathrm{S1}, \mathrm{S2}, \mathrm{S3}, \mathrm{S4}, \mathrm{S5}, \mathrm{S6}, \mathrm{S7}\}$$
$$\Omega' = \{\top, \mathrm{S1}, \mathrm{S2}, \mathrm{S3}, \mathrm{S4}, \mathrm{S5}, \mathrm{S6}, \mathrm{S7}, \bot\}$$

The statements' domains correspond to the iteration ranges of each for-loop, but without the conditionalized instances that do not match their conditions. For instance, S1 is not executed

if $x = 0$ and such instances are therefore removed from the instance set $\mathcal{D}_{\mathrm{S1}}$.

$$\begin{aligned}
\mathcal{D}_{\mathrm{S1}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x, y < 9\} \\
\mathcal{D}_{\mathrm{S2}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 1 \leq x < 9, 0 \leq y < 9\} \\
\mathcal{D}_{\mathrm{S3}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x < 8, 0 \leq y < 9\} \\
\mathcal{D}_{\mathrm{S4}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x < 9, 1 \leq y < 9\} \\
\mathcal{D}_{\mathrm{S5}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x < 9, 0 \leq y < 8\} \\
\mathcal{D}_{\mathrm{S6}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x, y < 9\} \\
\mathcal{D}_{\mathrm{S7}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS}, 0 \leq x, y < 9\} \\
\mathcal{D}_{\mathrm{S8}} =& \{(i,x,y) \mid 0 \leq i < \texttt{ITERATIONS} - 1, 0 \leq x, y < 9\}
\end{aligned}$$

Since all for-loops increase the induction variables after every iteration, the execution order corresponds to their lexicographic ordering. To order instances of S1 to S7 relative to each other, they are assigned a different last coordinate. The total number of coordinates of the scatter space is one plus two times of the deepest nesting level (cf. Equation (7.5)), i.e. seven.

$$\text{🕒} = \mathbb{Z}^7$$

The first coordinate is the order of the first sequential region, which contains only the for-loop with induction variable `i`. It can be any constant number, of which we chose 0. The instances of S8 of a specific `i`-counter value are executed before the instances of S1 to S7 because they follow in the sequential body of the outermost loop. The third coordinate therefore must be larger than the others' third coordinates.

The prologue can be assigned any scattering that is lexicographically smaller than every other instance's scattering. We simply assign -1 to the first coordinate, the remaining ones are irrelevant, but since they need to have a value, they are zero. Correspondingly, the epilogue must be assigned the lexicographic largest vector.

$$\begin{aligned}
\theta_{\mathrm{S1}} =& (i,x,y) \mapsto (0,i,0,x,0,y,0) \\
\theta_{\mathrm{S2}} =& (i,x,y) \mapsto (0,i,0,x,0,y,1) \\
\theta_{\mathrm{S3}} =& (i,x,y) \mapsto (0,i,0,x,0,y,3) \\
\theta_{\mathrm{S4}} =& (i,x,y) \mapsto (0,i,0,x,0,y,4) \\
\theta_{\mathrm{S5}} =& (i,x,y) \mapsto (0,i,0,x,0,y,5) \\
\theta_{\mathrm{S6}} =& (i,x,y) \mapsto (0,i,0,x,0,y,6) \\
\theta_{\mathrm{S7}} =& (i,x,y) \mapsto (0,i,0,x,0,y,7) \\
\theta_{\mathrm{S8}} =& (i,x,y) \mapsto (0,i,1,x,0,y,0)
\end{aligned}$$

An excerpt of the induced execution order relation is shown below. It can be represented by piecewise polyhedral maps (Section 7.1.1.4) with a relatively large number of $\mathbb{Z}$-polyhedra. Fortunately, this representation is never required explicitly.

$$\begin{aligned}
<_{\mathrm{exec}} =& \{((\mathrm{S1}, i_1, x_1, y_1), (\mathrm{S1}, i_2, x_2, y_2)) \mid \\
& \qquad (i_1 < i_2) \vee (i_1 = i_2 \wedge x_1 < x_2) \vee (i_1 = i_2 \wedge x_1 = x_2 \wedge y_1 < y_2)\} \\
& \cup \{((S1, i_1, x_1, y_1), (S2, i_2, x_2, y_2)) \mid \\
& \qquad (i_1 < i_2) \vee (i_1 = i_2 \wedge x_1 < x_2) \vee (i_1 = i_2 \wedge x_1 = x_2 \wedge y_1 < y_2) \\
& \qquad \vee (i_1 = i_2 \wedge x_1 = x_2 \wedge y_1 = y_2)\} \\
& \ldots \\
& \cup \{((S7, i_1, x_1, y_1), (S8, i_2, x_2, y_2)) \mid i_1 \leq i_2\}
\end{aligned}$$

In order to ensure that the prologue is executed before, and the epilogue is executed after all non-virtual statements, we may increase the number of scatter dimensions, prefix the prologue's scattering with -1, the epilogue's with $+1$ and everything else using 0. But this is just one possibility to do so.

$$
\begin{aligned}
\Theta' &= \mathbb{Z}^8 \\
\theta'_{\top} &= () \mapsto (-1, 0, 0, 0, 0, 0, 0, 0) \\
\theta'_{\mathrm{S1}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 0) \\
\theta'_{\mathrm{S2}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 1) \\
\theta'_{\mathrm{S3}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 3) \\
\theta'_{\mathrm{S4}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 4) \\
\theta'_{\mathrm{S5}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 5) \\
\theta'_{\mathrm{S6}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 6) \\
\theta'_{\mathrm{S7}} &= (i, x, y) \mapsto (0, 0, i, 0, x, 0, y, 7) \\
\theta'_{\mathrm{S8}} &= (i, x, y) \mapsto (0, 0, i, 1, x, 0, y, 0) \\
\theta'_{\bot} &= () \mapsto (1, 0, 0, 0, 0, 0, 0, 0)
\end{aligned}
$$

There are two array variables in the program (`source` and `sink`). There are also two scalar variables in the program, `sum` and `avg`.

$$\mathcal{V} = \{\texttt{source}, \texttt{sink}, \texttt{sum}, \texttt{avg}\}$$

Both arrays have the same number of dimensions and also the same indexset, which in this example corresponds to the iterations space of the two inner for-loops.

$$\mathcal{I}_{\texttt{source}} = \mathcal{I}_{\texttt{sink}} = \{(x, y) \mid 0 \leq x, y < 9\}$$

There are plenty of memory accesses the SCoP, shown below. In order to be significant, every instance should have at least one write access, otherwise it can be removed without changing the semantics. By definition, scalar variables are read and written as a unit and therefore an access index is not required.

$$
\begin{aligned}
\lambda_{\mathrm{read}}(S1, i, x, y) &\mapsto \{(\texttt{source}, x, y)\} & \lambda_{\mathrm{write}}(S1, i, x, y) &\mapsto \{\texttt{sum}\} \\
\lambda_{\mathrm{read}}(S2, i, x, y) &\mapsto \{(\texttt{source}, x-1, y), \texttt{sum}\} & \lambda_{\mathrm{write}}(S2, i, x, y) &\mapsto \{\texttt{sum}\} \\
\lambda_{\mathrm{read}}(S3, i, x, y) &\mapsto \{(\texttt{source}, x+1, y), \texttt{sum}\} & \lambda_{\mathrm{write}}(S3, i, x, y) &\mapsto \{\texttt{sum}\} \\
\lambda_{\mathrm{read}}(S4, i, x, y) &\mapsto \{(\texttt{source}, x, y-1), \texttt{sum}\} & \lambda_{\mathrm{write}}(S4, i, x, y) &\mapsto \{\texttt{sum}\} \\
\lambda_{\mathrm{read}}(S5, i, x, y) &\mapsto \{(\texttt{source}, x, y+1), \texttt{sum}\} & \lambda_{\mathrm{write}}(S5, i, x, y) &\mapsto \{\texttt{sum}\} \\
\lambda_{\mathrm{read}}(S6, i, x, y) &\mapsto \{\texttt{sum}\} & \lambda_{\mathrm{write}}(S6, i, x, y) &\mapsto \{\texttt{avg}\} \\
\lambda_{\mathrm{read}}(S7, i, x, y) &\mapsto \{\texttt{avg}\} & \lambda_{\mathrm{write}}(S7, i, x, y) &\mapsto \{(\texttt{sink}, x, y)\} \\
\lambda_{\mathrm{read}}(S8, i, x, y) &\mapsto \{(\texttt{sink}, x, y)\} & \lambda_{\mathrm{write}}(S8, i, x, y) &\mapsto \{(\texttt{source}, x, y)\}
\end{aligned}
$$

We only present the value source relation here. The data flow dependency relation can be derived easily from it.

In the first iteration the source array contains data from the region before the SCoP was entered. This is represented by using the prologue as the source. The values stored in the arrays after the SCoP executed are the ones the epilogue depends on. `sum` and `avg` do not

require such dependency, they are out of scope. Accesses to them after the SCoP terminated invoke undefined results.

$$\sigma((S1,i,x,y),(\texttt{source},x,y)) \mapsto \begin{cases} \top & \text{if } i=0 \\ (S8,i-1,x,y) & \text{otherwise} \end{cases}$$

$$\sigma((S2,i,x,y),(\texttt{source},x-1,y)) \mapsto \begin{cases} \top & \text{if } i=0 \\ (S8,i-1,x-1,y) & \text{otherwise} \end{cases}$$

$$\sigma((S3,i,x,y),(\texttt{source},x+1,y)) \mapsto \begin{cases} \top & \text{if } i=0 \\ (S8,i-1,x+1,y) & \text{otherwise} \end{cases}$$

$$\sigma((S4,i,x,y),(\texttt{source},x,y-1)) \mapsto \begin{cases} \top & \text{if } i=0 \\ (S8,i-1,x,y-1) & \text{otherwise} \end{cases}$$

$$\sigma((S5,i,x,y),(\texttt{source},x,y+1)) \mapsto \begin{cases} \top & \text{if } i=0 \\ (S8,i-1,x,y+1) & \text{otherwise} \end{cases}$$

$$\sigma((S8,i,x,y),(\texttt{sink},x,y)) \mapsto (S7,i,x,y)$$

$$\sigma(\bot,(\texttt{source},x,y)) \mapsto (S8,i,x,y)$$

$$\sigma(\bot,(\texttt{sink},x,y)) \mapsto (S7,i,x,y)$$

The scalars' scope is limited to the innermost loop body, the dependence relation therefore is much simpler:

$$\sigma((S2,i,x,y),\texttt{sum}) = (S1,i,x,y)$$
$$\sigma((S3,i,x,y),\texttt{sum}) = (S2,i,x,y)$$
$$\sigma((S4,i,x,y),\texttt{sum}) = (S3,i,x,y)$$
$$\sigma((S5,i,x,y),\texttt{sum}) = (S4,i,x,y)$$
$$\sigma((S6,i,x,y),\texttt{sum}) = (S5,i,x,y)$$
$$\sigma((S7,i,x,y),\texttt{avg}) = (S6,i,x,y)$$

There are also anti- and output-dependencies because variables are reused for multiple values. Only direct dependencies are shown here. For instance, there is only one variable `sum`, hence S1 cannot start until the previous iteration uses it the last time, i.e. an instance of statement S6. S6 depends on S1 to S5, therefore dependencies involving these are redundant.

$$\begin{aligned} <_{\text{anti}} = {} & \{(S1,i,x,y,S8,i,x,y)\} \\ & \cup \{(S2,i,x_1,y,S8,i,x_2,y) \mid x_1 - 1 = x_2\} \\ & \dots \\ & \cup \{(S8,i,x,y,S7,i,x,y)\} \\ & \cup \{(S6,i_1,x_1,y_1,S1,i_2,x_2,y_2) \mid i_1 + 1 = i_2\} \\ & \cup \{(S7,i_1,x_1,y_1,S6,i_2,x_2,y_2) \mid i_1 + 1 = i_2\} \end{aligned}$$

The output dependencies ensure that the last written value is stored in the arrays when the SCoP terminates. The effective last write of an element will also have a flow dependency to the epilogue.

$$<_{\text{output}} = \{(S7,i_1,x,y,S7,i_2,x,y) \mid i_1 + 1 = i_2\} \cup \{(S8,i_1,x,y,S8,i_2,x,y) \mid i_1 + 1 = i_2\}$$

8 Distributed Memory Parallelism and the Polyhedral Model

Everything in the previous chapter appears more or less similar in any textbook about the polyhedral model [37]. It is time to introduce some new definitions.

An *Single Program Multiple Data* (SPMD) scheme parallelism executes – as its name says – the same program multiple times, typically on different hardware nodes. The characteristic of a node is that it can access only its own memory, but not directly the memory of other (remote) nodes. Some programming models (PGAS, *High Performance Fortran* (HPF), Universal Parallel C) may make it appear as if one can directly use the memory, but involve a big overhead. Communication between the nodes can still happen using APIs such as MPI.

Such units of parallelism are also called processes, tasks or ranks. These terms usually refer to a hardware unit that can run multiple processes or ranks if configured to do so. In this case the node's memory is divided into (not necessarily disjoint) regions by the processors *Memory Management Unit* (MMU).

In contrast, in shared-memory parallelism (OpenMP, Pthreads), there is just one memory that is accessed by all threads at the same speed, or at least without additional programming in case of a *non-uniform memory architectures* (NUMA). By default only the stack is separate for each thread. Instead of all processes executing the same program (SPMD), only one thread is active at the beginning and needs to activate the other threads. Those other threads remain idle prior activation.

It is much harder to optimize for distributed-memory than for shared-memory architectures. While in the shared-memory case the data is just there and all the programmer has to do is to avoid race conditions if multiple threads write to the same memory, distributed memory platforms require the programmer to plan what data to send to whom and when.

The polyhedral model needs to be extended in order to model *Distributed Memory Machines* (DMMs). Such a cluster computer logically consists of multiple nodes. Let $\mathcal{P}$ be the set of nodes. We assume it is organized in a $N_{\mathcal{P}}$-dimensional grid that does not change after program startup. Nodes are therefore identified using $N_{\mathcal{P}}$-dimensional vectors representing the coordinates in grid. The coordinates should match the physical connections in the grid to reduce routing.

Also, we need to specify on which node a statement instance is executed and where the memory is stored. Both are discussed in the following two sections.

8.1 Data Distribution

We classify two different types of memory: Data that is distributable between nodes and data that is copied for every node (privatized). For instance, the sole value of a scalar variable cannot be split up such that parts of it are stored on different nodes. Big arrays with more elements than a single node can store are predestined to be distributed.

The previously defined set of variables $\mathcal{V}$ (Definition 7.2, Page 100) includes variables of both types. We add a subset of arrays that are distributed, $\mathcal{F}$. Any non-distributed variable in

$\mathcal{V} \setminus \mathcal{F}$ is available on every node, i.e. each node has its own private copy of it. The terminology in this thesis for distributed arrays is *field*, because in our main use case they store the spinor- and gauge fields.

Every element of a field must be assigned at least one node where it is stored by default. Multiple storage locations are permissible. The relation $\pi_F \subseteq \mathcal{I}_F \times \mathcal{P}$ specifies the set of nodes an element of field $F \in \mathcal{F}$ is stored on. Since every element must be available on at least one node (otherwise the actual value is forgotten), π_F is a right-total relation and therefore can be interpreted as a function $\pi_F : \mathcal{I}_F \to 2^{\mathcal{P}}$.

We establish the term *storage location* for a triple $(F, \vec{k}, \vec{p})$ of an element $\vec{k}$ of field F stored on node $\vec{p}$.

If a logical element $(F, \vec{k})$ is assigned a new value, it must be written to all nodes that store it. Redundancy pays off only when reading the element, in which case only one – the nearest available – needs to be fetched. Consistency between all privatized copied is guaranteed by updating all locations in case the logical element is overwritten. Ideally, the node itself is a storage location such that no remote communication is required. Otherwise, it may chose a location with the nearest distance. $\pi_{\mathcal{F}}$ is the set of all storage locations of any field, i.e. the union

$$\pi_{\mathcal{F}} = \bigcup_{F \in \mathcal{F}} = \left\{ (F, \vec{k}, \vec{p}) \;\middle|\; (\vec{k}, \vec{p}) \in \pi_F \right\} .$$

In our model storing the same element multiple times in the same local memory is useless because of the uniform memory assumption: Every address is accessed at the same speed and prefetch of data as seen in the first part of this thesis is not a concern. The relation π_F only specifies the nodes where it is stored, not a number of copies per node.

8.1.1 Data Alignment

Data alignment is the decision which elements are stored together on the same node. Some elements may be required in multiple computations that combine different elements. A computation with an ideal data distribution would require only locally available elements without duplication values. In the general case this is not possible.

Data alignment and distribution are different in that alignment only specifies location relative to other values. There is a degree of freedom where an element is placed. Some elements may be fixed to arbitrary nodes, but the location of other elements follows by the alignment rules.

For instance, the Lattice QCD Hopping Matrix requires every site in 8 different computations (stencils). Ideal alignment would imply that every element is on the same node. Which node it is does not matter. If just one site is assigned to a node, the ideal alignment rule would imply all other noes stored on the same node as well. But storing every value on the same node is not desirable for parallelism, therefore alignment has to make trade-offs. Alignment is a NP-hard problem [46].

This thesis does not cover the alignment problem. Instead, we assume that the data distribution is defined by the programmer. The article [47] contains an overview on heuristics found in literature.

8.1.2 Default Data Distribution

In case the programmer does not explicitly specify a distribution, we assume a simple block distribution by default. The definition of a block-distributed π_F is ($N_{F,i}$ and $N_{\mathcal{P},i}$ are the size of field F in the ith dimension, respectively the number of nodes in the ith dimension):

$$\pi_F : \vec{k} \mapsto \left\lfloor \frac{\vec{k}_i N_{\mathcal{P},i} + N_{F,i} - 1}{N_{F,i}} \right\rfloor_{i=1..N_F}$$

The number of cluster dimensions must be at most the number of field dimensions for this to work. If this is not the case, the node coordinates must be projected to coordinates of lower dimensions to avoid idle nodes. On the other side, more field dimensions than cluster dimension are no problem. The excess dimensions are just assumed to have length one, i.e. no data distribution in that dimension.

Note that, if the field's length is not a multiple of the number of nodes in that dimension, the block lengths are irregular. Typically the slowest nodes (the node with the most workload) dictates the speed of the computation. Imagine how tedious it is to special case the irregular nodes when programming by hand.

This is simple, but sufficient for all the stencil codes in this thesis, including Lattice QCD. More sophisticated approaches would analyze the access patters in every SCoP in the program and apply an alignment algorithm.

8.2 Work Distribution

There are two primary methods on how to organize which process or thread does which work. The first is the job queue at runtime, a list of work to be done (not necessarily centralized) which is distributed to the workers. If a worker has done its work it requests more work from the worklist. This type of job is good if the amount of work per job varies a lot or is unknown before the job starts. Its disadvantage is the overhead of job scheduling. On distributed memory architectures – since it is not known in advance what process will do a job – the job's input data must be transferred with the job to the executing node .

The second school is to assign jobs to nodes statically before execution starts. Every node knows in advance how much work is to be done and what data is required for its execution. For the organization of workload itself, no overhead is required. For Lattice QCD this is the only efficient option. Domain decomposition (Section 2.5, Page 30) distributes work between nodes equally and predictably, there is no need for extra overhead at runtime. Transfer of complete subvolume data between node would slow-down the complete processing considerably.

The relation π_Ω between instances and nodes defines which instances are executed where. A tuple $(S, \vec{i}, \vec{p})$ of an instance $(S, \vec{i})$ executed on node $\vec{p}$ is called an *execution*. $\mathcal{E} \subseteq \mathcal{D} \times \mathcal{P}$ is the set of all executions. To preserve semantics, every instance should be executed somewhere, i.e. π_Ω is left-total. Instances that have no observable effect can be optimized away by not being assigned to any node, but for simplicity we assume every statement contributes to the semantics of the SCoP. A statement can also be executed redundantly on multiple nodes. This might be faster than executing it on one node only and then submitting its result to the other. π_Ω can be interpreted as a function that maps every execution to the set of nodes it is executing on.

Examples for trivial statement placement relations for a statement S are

$$\pi_S = \mathcal{D} \times \mathcal{P}$$

which executes every instance on every node and

$$\pi_S = \mathcal{D} \times \{\vec{0}\}$$

which executes the SCoP on the master node ($\vec{0}$) only, while the other nodes are idle. Both placements are necessarily correct without any data transfer in the sense that the SCoP's result is available on at least one node. However, it is executed sequentially and therefore defeats the purpose of parallel computing.

8.2.1 Default Work Distribution

As with data distribution, the programmer could define a work distribution for every statement instance. However, this is more tedious to do that with a field distribution, programmers may

not even know what parts in their code is understood as a statement by the compiler. In most cases it is not even necessary as it can be inferred well by simple heuristics.

If no distribution has been specified explicitly, the compiler will choose one. The goal is locality: Select a location that requires the least amount of communication. There are three kinds of communication. From the storage location to a statement that reads (consumes) the element, from a statement that generates an element to the storage location, and direct transfer from a generator to a consumer.

Generally, consumers should be executed on the same node where the generator is executed and vice versa. If either one location is known we transitively infer the other's location. This yields a greedy fixpoint computation that assigns locations to statement instances until all statements are executed somewhere.

The compiler does not have a sophisticated algorithm in case that according to these rules the execution location conflicts. Only the first possible location encountered during the fixpoint computation is accepted (greedy algorithm).

The rules in detail follow below.

1. If a statement writes a non-field variable that is used after the SCoP's termination, the statement instance must be executed on all nodes. A non-field never invokes communication and by the nature of SPMD, every node will use the result.

2. If the statement instance generates a field value that is not overwritten before the end of the SCoP (i.e. a flow to the epilogue), it is computed on the node that owns the value ("owner computes") according to the data distribution.

3. If the instance computes a value that is the input another instance of which the executed location is known, the generator is also executed on that node ("dependence computes"). This rule is applied until a fixpoint is reached, following every def-use chain backwards.

4. The instances that are not mapped to any node yet are assigned to the master node at coordinate $\vec{0}$. The rationale is that we expect such instances to have no effect and therefore will be optimized away. Another possibility is that it prints output to stdout/stderr to report the progress of computation. It is a side effect we assumed not to happen, but it is useful for debugging purposes. Execution on one node (only) ensures preservation of order.

In any case, a non-field dependency forces a generator to be executed on at least all the nodes the consumer is executed like in the first rule. No transfer is never generated for non-field data.

Obviously much smarter algorithms are possible, but in practice, most other related works on automatic distribution choose the owner-computes rules which works well for many classes of problems such as stencils.

8.3 Data Transfers

Once it is known where data is stored and on which nodes it is needed for computations, the data transfers between those nodes can be organized. In contrast to data and work distribution, we do not expect the programmer to define anything explicitly. It solely up to the compiler to find a node to get the data from, assemble data in messages, and insert code for the transfer.

Data transfers are only necessary for flow-dependencies (read-after-write hazards) between nodes. Other types of dependencies (write-after-read and write-after-write) do not involve data movement but ensure that values are not overwritten prematurely. Data is duplicated in transfer buffers, so there is no danger of overwriting other values.

In general, a transfer is the flow of data from a statement that computes a value (the generator G) and a statement that needs that value for its computation (the consumer C). In the SCoP this means that the generator writes the value to some element of a field $\texttt{F}[\vec{k}]$ or array and the consumer reads it from there. We redirect this flow to go through a communication channel. That is, the generator writes the value into a communication buffer, invokes the transfer of the buffer's contents to the consumer's node which already waits for it, and finally the consumer statement reads the value from its received copy of the buffer contents.

As a consequence, accesses to the original array are removed and therefore the hazards involved. The new communication buffers are dedicated to that single flow of a value and therefore does not hazard with anything. The communication source and target node is independent of the element's distribution. It is possible to say a field's element has no dedicated location during a SCoP's execution.

Data flows from the prologue to the epilogue can be ignored. Such a dependency means that the element is not changed in the SCoP, therefore no transfer needs to be made.

8.4 Chunking

To avoid transferring single values with immense overhead, values are grouped into data *chunks*, such that all data of one chunk can be written to the same buffer (per destination), then sent and received. Intuitively, two values cannot be in the same chunk if there is a direct or indirect dependency between the statements that produce them.

We model the mapping from statement instances to chunks by grouping them in equivalence sets. Instances belong to the same chunk if a *chunking function* returns the same value for them. The challenge is to find a chunking function that minimizes the number of transfers and therefore pack as many values as possible into a communication buffer.

Chunking functions already appeared in the polyhedral model to find schedules that improve cache usage [48].

Definition 8.1 (Chunking Function)
A *chunking function* is a projection $\varphi : \sigma \to \mathcal{X}$ of flow-dependencies. The equivalence classes

$$[\varphi((G,\vec{i}),(F,\vec{k}),(C,\vec{j}))]$$
$$= \left\{((S,\vec{g}),(E,\vec{g}),(R,\vec{h})) \;\middle|\; \varphi((S,\vec{g}),(E,\vec{g}),(R,\vec{h})) = \varphi((G,\vec{i}),(F,\vec{k}),(C,\vec{j}))\right\}$$

are called *chunks*.

The chunking function's image $\mathcal{X}$ can be any set. Its only use is to distinguish different chunks. A practice-oriented choice is to use $\mathcal{X} = \delta$, i.e. the chunking function maps data flows to data flows. One may think of the dependency mapped to as the representative of its equivalence class, the chunk. By convention, φ then always maps a representatives to itself, i.e. is idempotent.

Not every chunking function preserves a SCoP's semantics. Some chunkings may produce circular dependencies and therefore cause a deadlock. This is the case, for instance, if a statements reads a value from a chunk and produces a value that belongs to the same chunk. The associated communication buffer therefore cannot be completely filled without the information it contains itself. An example:

Let the flows from S1 to S2 and S3 to S4 belong to the same chunk

$$\begin{aligned}\varphi(\text{S1},(\texttt{A},0),\text{S3}) &= \varphi(\text{S3},(\texttt{A},1),\text{S4})\\ &\neq \varphi(\text{S2},\texttt{x},\text{S3})\ .\end{aligned} \tag{8.1}$$

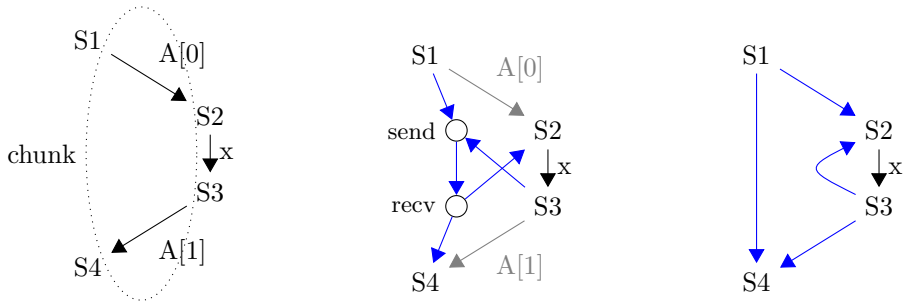
S1 generates A[0], S3 consumes `x` and generates A [1]. The communication buffer containing A[0] and A[1] (because they are generated in the same chunk) is not filled before the execution

```
S1:  A[0] = f();
S2:  x = A[0];
S3:  A[1] = x;
S4:  use(A[1]);
```

Listing 8.1: Program to illustrate invalid chunking

of S3 and therefore cannot be sent. However, S2 requires some content of this buffer and S3 depends on S2 through variable `x`. Hence this chunking function yields a non-executable SCoP.

The chunking function is also illustrated in Figure 8.1. The starting point is Figure 8.1a which show the dependency graph and the two data flows we want to apply the chunking function of Equation (8.1) on. We do this by adding communication code for sending and receiving data in Figure 8.1b. Since both elements of array `A` are put into the same message, the send operation has a dependency from both generators. Also, the two consumers depend on the receive operation because their consumed value does not arrive before. And of course, the receive depends on the send operations being issued. But this adds a loop to the graph (S2 → S3 → send → recv → S2), making it impossible to schedule it.



(a) Dependency graph of Listing 8.1 (b) Communication primitves inserted (c) Collapsed chunk with $Q = \{(\text{S1}, (\texttt{A}, 0), \text{S3}), (\text{S3}, (\texttt{A}, 1), \text{S4})\}$

Figure 8.1: Illustration of chunking function (8.1)

Therefore we employ the notion of *validity* of a function as defined in Definition 8.3. Instead of adding send and recv primitives as in Figure 8.1b, we may also just add the edges that cause the loop to appear (Figure 8.1c). Basically, a chunking function is valid if the collapsing of instances of a (non-circular) statement instance flow graph is still acyclic. We already used collapsing in the previous example, let us formalize it.

Definition 8.2 (Chunk Collapse)
Let (V, E) be a directed graph and $Q \subset V \times V$ a set of edges (not necessarily $\subseteq E$). Define $L = \{u \mid (u, v) \in Q\}$ and $R = \{v \mid (u, v) \in Q\}$. The *chunk Q-collapsed* graph of (V, E) is the graph $(V, E \cup \{(u, v) \mid u \in L, v \in R\})$.

Collapsing a chunk therefore means to add edges to a graph such that the involved vertices build a complete bipartite subgraph.

Definition 8.3 (Valid Chunking Function)
Let φ be a chunking function and $(\mathcal{D}, <_{\text{dep}})$ a valid dependence graph. φ is *valid* if $(\mathcal{D}, <_{\text{dep}})$ is still valid after chunk-collapsing all chunks

$$Q \in \left\{ [\varphi((G, \vec{i}), (F, \vec{k}), (C, \vec{j}))] \;\middle|\; ((G, \vec{i}), (F, \vec{k}), (C, \vec{j})) \in \sigma \right\} .$$

The validity of a chunking function therefore depends on the schedulability of the resulting dependence graph. Given this definition we show that a valid chunking always exists.

Definition 8.4 (Trivial Chunking Function)
The chunking function $\varphi_{\text{trivial}} = \text{id}$, the identity function over δ, is the *trivial* chunking function.

Every chunk induced by the trivial chunking function as exactly one element: The flow-dependency representing itself. A trivial chunking function is always valid. If a valid schedule already exists without applying the trivial chunking function then the same schedule is valid with that chunking function, but will cost a lot of overhead because any communication buffer contains just one value. A chunking function should therefore have some desirable properties.

Definition 8.5 (Semi-Trivial Chunking Function)
The chunking function $\varphi_{\text{semi-trivial}} : \sigma \to \mathcal{D}$, $((C, \vec{i}), (F, \vec{k})) \mapsto (C, \vec{i})$ is the *semi-trivial* chunking function.

The semi-trivial chunking function is also valid for any dependence graph. First, it is irrelevant for the schedulability of a dependence graph which and how many values are transferred between statements. That is, we can add multiple elements to the same message of they origin is the same. Second, collapsing does not add edges to the graph if their either just one node in L and that node was the target node of all edges in Q or there is just one node in R and that node was the source node of all edges in Q. We chose the former for the definition of semi-triviality because it is more intuitive: Such messaged contain values from one generator only, all its consumers depend on it anyway.

A chunking function we may wish to apply must be valid in any case. Besides validity, there are multiple desirable, but partially conflicting goals to optimize for. These include:

- reduce the number of chunks, so fewer communication events with longer messages happen
- maximize the size of chunk, so more data is transferred per communication event
- allow asynchronous communication such that communication latency overlaps with computations
- balance chunk sizes (in contrast to, for instance, an extremely large chunk and a small one)
- prefer data transfer between close nodes
- simple, easily computable

To maintain tractability, the chunking function must be a piecewise affine mapping as in Sections 7.1.2 and 7.1.4.1.

8.4.1 Antichain Chunking

There is an optimal and efficient algorithm applicable on the detailed dependence graph. Efficient here means polynomial in the number of nodes in the graph. It is optimal in the sense that it minimizes the number of chunks, but is defined only on the *detailed dependence graph*, i.e. the dependence graph with a node per statement instances.

When considering chunking, the detailed dependence graph has two different kinds of edges that make it difficult to apply graph theory on it. There are the flow dependencies that we wish to collapse, and all other dependency types that are not collapsed. Both types impose limitations on which flows can be collapsed. Therefore the first step in Algorithm 8.1 is to convert the dependence graph into a reduced form that captures only the required information.

Algorithm 8.1: Antichain chunking

Input : detailed dependence graph $G = (V, E)$, a direct flow relation $\sigma \subseteq E$
Result: chunking function φ
$V' = \sigma$;
$E' = \{(u, v) \mid u, v \in V', u = (u_1, u_2), v = (v_1, v_2), u_2 \Rightarrow_G^* v_1\}$;
$G' = (V', E')$;
$\varphi : V' \to \mathbb{N}$;
foreach $v \in V'$ **do**
    Find a longest chain in G' ending in v (e.g. depth-first search);
    $\varphi(v) \coloneqq$ length of that chain;
end
return φ;

The vertices in the new graph G' are the direct flows of the dependency graph. Edges are added between vertices if there is a path – consisting of edges of any type – between two flows. The result is similar to a transitive closure between flow dependencies. Dependencies are inherently transitive, therefore adding transitive edges impose no semantic change. In fact, only the reachability of any two flows using non-flow dependencies is required here, not the full transitive closure.

Chunking the flow dependencies of the detailed dependence graph is equivalent to partitioning the vertices of G' into pairwise incomparable sets. Such sets are also known as antichains. Finding the minimum number of chunks is the same as finding a minimum antichain decomposition in G'.

Mirsky's theorem [49] shows that the number of chains of a minimum antichain decomposition equals the length of a longest chain (path) in the graph. Every edge of a segment must be member of a different antichain, therefore they may build the set of representatives for each chunk.

The proof of Mirsky's theorem is constructive – it can be used directly to find a minimum antichain decomposition. Every node is assigned a depth, the longest distance to the root node. However, G' is a partial order, not a lattice, and there may not be a single root node. Instead, one takes the upstream node with the most hops as local "root". Nodes of the same depths are incomparable, therefore form an antichain. The greatest depth has the node at the end of the longest paths; the depth is the number of minimum antichains.

The antichain algorithm as presented is practically not usable for SCoPs because it requires an explicit representation of the detailed dependence graph. This graph has a node for every statement instance, therefore its execution time and memory requirement depends on the execution time of the SCoP itself, not the SCoP's description as set of statements with domains. If the SCoP has structural parameters, the detailed dependence graph does not even exist. It can only be constructed once the concrete values of these parameters is known.

8.4.2 Polyhedral Flow Dependence Graph Scheduling

The graph $G = (V', E')$ from Algorithm 8.1 is a graph of dependencies between direct flows – a flow dependence graph. Such graph can be scheduled using the know scheduling algorithms [33, 34] like any other generalized dependence graph. The algorithm would return a scatter function $\sigma \to \mathbb{Z}^N$ in one or multiple dimensions. We can use this function directly as chunking function. This technique is applicable to SCoPs with parameters and arbitrary large loop bounds.

In contrast to the antichain function, the resulting chunking is not guaranteed to be optimal in any sense. In fact, some algorithms might even be optimized for different goals that might not make sense for chunking, such as temporal locality [40, 48].

Even if we employ an scheduling algorithm that gives very dense schedules, there are other optimization goals we want to consider. First, it does not consider that transfer can be asynchronous and duration can overlap. Second, these algorithms typically are computationally intensive and can also return complex expressions for the scatter functions. Third, to the programmers there is no obvious connection between the source code they wrote and the resulting chunking. They have no direct influence on the output which makes it hard to modify the source code to get a chunking closer to the one the programmers may have in mind.

8.4.3 Schedule-Dependent Chunking Heuristic

Given the problems with the previous chunking methods, we need a heuristic as alternative. In order to also be predictable for the programmer, we select a chunking function guided by how the SCoP is written, i.e. following the SCoP's scatter function. In many cases the source code is already organized such that the chunking is obvious, such as any typical stencil code. In other cases a schedule optimizer may find a scatter function that reduces the amount of chunks. Receiving all data before a loop's execution or sending data after a loop terminates is a common communication pattern which we are going to exploit.

The idea behind the algorithm is the following. Instances whose schedule differ only in the innermost coordinate (meaning they are executed in the same outer code regions but different inner regions) are put into the same chunk. Then the algorithm checks whether any dependencies are violated. If it is legal one can recheck with larger chunks by involving ignoring more scattering coordinates. The most coarse chunking is applied.

The pseudocode is shown by Algorithm 8.2. Every generator statement is processed individually to allow different levels of chunking instead of just SCoP-wide level. It begins with the most coarse-grained chunking function possible, the one that puts everything into the same chunk and checks it for dependence violations. In succeeding iterations it adds more scattering coordinates to the chunking function's image, refining the chunks. When the number of levels of the scattering function is reached the algorithm aborts. This is the semi-trivial chunking function (Definition 8.5) where only generators that execute in parallel anyway are chunked. Since by definition the input dependence graph is loop-free, this semi-trivial chunking is loop-free as well.

Chunks obtained this way are statement-*continuous*. This means that if two instances belong to the same chunk, any instance of the same statement with a scattering lexicographically between the scattering of the two will also belong to the same chunk.

A legal schedule guarantees that instances ordered lexicographically according to their scattering only have dependencies in direction of this order. This property can be used to make the legality check more efficient: if within the same chunk the scatterings of all consumer instances lie after all the generator instances, then no loop is introduced by collapsing the data flows in this chunk because this adds edges only from generators to consumers which is compatible with any preexisting edges. What actually is needed is the scattering of the first consumer, $\min_{\text{lex}} \theta(C)$, and the one of the last generator ($\max_{\text{lex}} \theta(G)$). If the last generator lies before the first consumer, the chunking will be valid.

In this heuristic, different instances from different statements are never chunked together.

The algorithm is greedy in the sense that it takes the coarsest chunking to a statements as soon it is processed. Chunking of one statement can make chunking of another statement impossible as the example below shows. The order of processing the statements therefore is significant. Algorithm 8.2 does not define the order of iteration in the for-each loop making its result not uniquely defined.

Listing 8.2 shows an example program where the order of iteration matters. The domains

Algorithm 8.2: Schedule-dependent chunking heuristic

Input : parametric dependence graph $(\mathcal{D}, <_{\text{dep}})$, legal schedule θ
Result: chunking function $\varphi : \sigma \to \circlearrowright^{N_\circlearrowright}$
$\varphi((G,\vec{i}),(F,\vec{k}),(C,\vec{j})) \leftarrow (G, \theta(G,\vec{i}))$; `// Semi-trivial chunking function`
foreach $S \in \Omega$ **do**
  for $l \leftarrow 0$ **to** $N_\circlearrowright$ **do**
    if $l = N_\circlearrowright$ **then**
      break;
    end
    $\rho \leftarrow \left\{((S,\vec{i}),(F,\vec{k}),(C,\vec{j})) \mapsto (k_1,\ldots,k_l,\vec{0}) \;\middle|\; \vec{k} = \theta(G)\right\} \cup \varphi|_{G\neq S}$;
    $\mathcal{X} \leftarrow \rho(\sigma)$;
    $L = \left\{\chi \mapsto (G,\vec{i}) \;\middle|\; \chi \in \mathcal{X}, ((G,\vec{i}),(F,\vec{k}),(C,\vec{j})) \in \rho^{-1}(\chi)\right\}$;
    $R = \left\{\chi \mapsto (C,\vec{j}) \;\middle|\; \chi \in \mathcal{X}, ((G,\vec{i}),(F,\vec{k}),(C,\vec{j})) \in \rho^{-1}(\chi)\right\}$;
    $d \leftarrow \{c \mapsto \min_{\text{lex}} \theta(R(\chi)) - \max_{\text{lex}} \theta(L(\chi))) \mid \chi \in \mathcal{X}\}$;
    if $\forall \chi \in \mathcal{X} : d(\chi) \geq 1$ **then**
      $\varphi \leftarrow \rho$;
      break;
    end
  end
end
return φ;

```
   for(int i = 1; i <= 2; ++i) {
A:   A[i] = D[i-1];
B:   B[i] = A[i];
C:   C[i] = B[i-1];
D:   D[i] = C[i];
   }
```

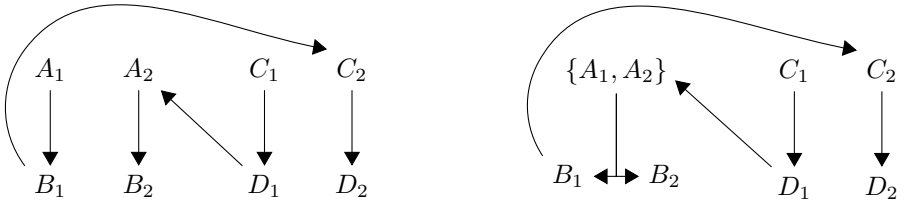
Listing 8.2: Example program showing the underdefined result of the chunking heuristic

and source-induced schedule are

$$\begin{aligned}
&\mathcal{D}_A = \mathcal{D}_B = \mathcal{D}_C = \mathcal{D}_D = \{1,2\}\\
&\theta_A(i) = (0,i,1)\\
&\theta_B(i) = (0,i,2)\\
&\theta_C(i) = (0,i,3)\\
&\theta_D(i) = (0,i,4)\ .
\end{aligned}$$

The detailed dependency graph is shown in Figure 8.2a. There are no loops but two dependency chains in this program, all of them are flow dependencies:

$$\begin{aligned}
&(\mathrm{A},1) \overset{\texttt{A[1]}}{\to} (B,1) \overset{\texttt{B[1]}}{\to} (C,2) \overset{\texttt{C[2]}}{\to} (D,2)\\
&(\mathrm{C},1) \overset{\texttt{A[1]}}{\to} (D,1) \overset{\texttt{D[1]}}{\to} (A,2) \overset{\texttt{A[2]}}{\to} (B,2)
\end{aligned}$$



(a) Before collapse (b) Chunked generator A

Figure 8.2: Counterexample

Using antichain chunking, there are three chunks.

$$\varphi((A,1),(A,1),(B,1)) = \varphi((C,1),(C,1),(D,1)) = 0$$
$$\varphi((B,1),(B,1),(C,2)) = \varphi((D,1),(D,1),(A,2)) = 1$$
$$\varphi((C,2),(C,2),(D,2)) = \varphi((A,2),(A,2),(B,2)) = 2$$

If the heuristic algorithm begins with processing statement A, the most coarse-grained chunking function is

$$\varphi((\mathrm{A},i),(A,i),(B,j)) = () \ ,$$

which yields the collapsed graph in Figure 8.2b. Chunking C is not possible anymore because it would add a dependency from C_2 to D_1 and therefore a loop. We end up with at least 4 chunks corresponding to the largest chain/antichain.

The example is symmetric in A,B and C,D. Hence, the algorithm could also have chunked $\{C_1, C_2\}$ first which would restrict A to be chunked.

8.4.4 Transfer Sets

Until now we did not use any information about the placement of data and work. This changes here. Once the chunking function is known, we can insert calls that trigger communication and redirect any accesses to remote data to the communication buffers. If a value is stored at multiple locations there is still a decision to be made from where to take it.

All required information can be gathered into a giant relation "Transfers" between the representative instance of the chunk, generator execution, consumer execution and the field element they access. Transfers is a $\mathbb{Z}$-polyhedron relation (Section 7.1.1.5) made up from multiple polyhedral sets.

$$\begin{aligned}\text{Transfers} \coloneqq \big\{((G,\vec{i}_G,\vec{p}_G),(F,\vec{k}_F),(C,\vec{i}_C,\vec{p}_C)) \mid \\ ((G,\vec{i}_G),(F,\vec{k}_F),(C,\vec{i}_C)) \in \sigma, \\ (G,\vec{i}_G,\vec{p}_G),(C,\vec{i}_C,\vec{p}_C) \in \pi_\Omega\big\}\end{aligned}$$

The execution $(G,\vec{i}_G,\vec{p}_G)$ is where the value is generated and therefore $\vec{p}_G$ specifies the source node of the transfer. The consumer execution $(C,\vec{i}_C,\vec{p}_C)$ uses this value and thus $\vec{p}_C$ is the target node of the transfer. $(F,\vec{k}_F)$ is the element that is transferred. Note that the storage placement π_F is irrelevant here. Values are transferred directly from generator to consumer. Transfers is something like a data flow relation that includes between which nodes the transfer happens.

Our model allows generator instances being executed on multiple nodes. This is useful if transferring a value is more costly than executing it redundantly on the same node. This also means that we need to select a source for every consumer execution. What is needed is a function that assigns a unique data source for every element $(F, \vec{k}_F)$ and consumer $(C, \vec{i}_C, \vec{p}_C)$ uses. One is interested to find the closest of these nodes. This is done by the relation MinDistance defined as follows.

$$\text{MinDistance} \coloneqq \left\{ ((C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F)) \mapsto \underset{(G, \vec{i}_G, \vec{p}_G)}{\arg\min_{\text{lex}}} \begin{pmatrix} \|\vec{p}_G - \vec{p}_C\|_1 \\ \vec{p}_G - \vec{p}_C \end{pmatrix} \;\middle|\; \ldots \right\}$$

MinDistance assigns a generator execution to every element $(F, \vec{k}_F)$ consumed by some execution $(C, \vec{i}_C, \vec{p}_C)$. It is structurally compatible to the Transfers-relation because we agreed to not make a difference between mapping types and Cartesian products (Sections 7.1.1.4 and 7.1.1.5).

The primary decision measure is the Manhattan distance ($\|x\|_1 = \sum_i |x_i|$) between two nodes. It corresponds to the number of hops in a cluster organized in a Cartesian mesh. The distance is 0 if, and only if, the generator is on the same node and therefore always preferred.

The term $\vec{p}_G - \vec{p}_C$ serves as a tiebreaker. If there are two nodes with the same distance, take the one to the "left" of the consumer in the first dimension that differs. MinDistance does not take link latency or bandwidth into account. For instance, traffic could be distributed between multiple nodes of the same hop-distance if the bandwidth is saturated. Another example are networks where performance is limited by connection latency. In this case values should be collected from as few nodes as possible, even if it is also available at closer nodes.

The single value transfers from MinDistance are then partitioned by the chunking function, which results into a new mapping relation "Chunks".

$$\begin{aligned} \text{Chunks} \coloneqq \big\{ \chi \mapsto ((C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \;\big|\; \chi = \varphi((G, \vec{i}_G), (C, \vec{i}_C)), \\ ((C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \in \text{MinDistance} \big\} \end{aligned}$$

Even more relevant for the implementation is the mapping that also depends on the source and target nodes. The source nodes must know which values it has to send and the receiving node must prepare a buffer to hold those elements. Every node may receive from/send to every other node in the cluster, potentially requiring a separate buffers for all of them. The relation "Events" determines which elements a message contains.

$$\begin{aligned} \text{Events} \coloneqq \big\{ (\chi, \vec{p}_G, \vec{p}_C) \mapsto ((C, \vec{i}_C), (F, \vec{k}_F), (G, \vec{i}_G)) \;\big|\; \chi = \varphi((G, \vec{i}_G), (C, \vec{i}_C)), \\ ((C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \in \text{MinDistance} \big\} \end{aligned} \tag{8.2}$$

Therefore Events maps a communication event consisting of the source node ($\vec{p}_G$), the destination node ($\vec{p}_C$) and the point in time to send it (χ) to the elements that the message contains $(F, \vec{k}_F)$. If Events maps a tuple to nothing, no corresponding message needs to be sent. In case of stencils the set of non-empty messages is constant, containing just the neighborhood nodes.

8.4.5 Transfer Primitives

We define four transfer statement primitives or events that represent the states of a transfer buffer. An alternative interpretation is that they are actions on buffers. They must be inserted explicitly into a SCoP at specific locations to make the data transfer between the nodes happen.

send_wait Reserves a send buffer of appropriate size. Typically waits until the previous send operation has finished and the next chunk can be written into that buffer. An MPI-based implementation would preallocate some memory for the buffer.

send Signals that all data has been written into a send buffer and hence can be transferred to its destination. Equivalent to `MPI_Isend` in an MPI-based implementation.

recv_wait Waits until all data of a communication event is received. Statements can then start reading from this receive buffer. Represents `MPI_Wait` on a receive buffer in MPI.

recv Notification that no more data is read from the receive buffer. The implementation can free resources that was allocated by send_wait or reinitialize them to be reused.

8.4.5.1 Primitives' Schedules

Given the Events-relation and a fixed schedule θ we can derive where these primitives must be placed into a SCoP. For preparation, we modify the schedule θ to a schedule θ' in order to be able to place statement instances between other instances by appending a coordinate zero to the scattering.

$$\theta'_S(\vec{i}) \coloneqq \begin{pmatrix} \theta_S(\vec{i}) \\ 0 \end{pmatrix}$$

The scatter space dimension becomes $N'_{\clockface} = N_{\clockface} + 1$. This simple change does not change the order of instances nor the legality of a schedule. In θ', a new statement R that shall execute before every instance $(S, \vec{i})$ can be scheduled at

$$\theta'_R(\vec{i}) = \begin{pmatrix} \theta_R(\vec{i}) \\ -1 \end{pmatrix} .$$

If R is supposed to be executed after it, scatter it at

$$\theta'_R(\vec{i}) = \begin{pmatrix} \theta_R(\vec{i}) \\ +1 \end{pmatrix} .$$

As utility functions we define two functions that return subsets if the scatter space $\clockface$, containing the execution time of either all generator instances on the source node or the consumer instances on the destination node per chunk. Both are projections of the Events-relation, the non-relevant parts projected away, and the schedule applied on the generator instance, or consumer instance respectively.

$$\text{GeneratorExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \coloneqq \left\{ \theta_G(\vec{i}_G) \;\middle|\; \exists C, \vec{i}_C, F, \vec{k}_F : ((\chi, \vec{p}_G, \vec{p}_C), (C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \in \text{Events} \right\} \subseteq \clockface$$

$$\text{ConsumerExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \coloneqq \left\{ \theta_C(\vec{i}_C) \;\middle|\; \exists G, \vec{i}_G, F, \vec{k}_F : ((\chi, \vec{p}_G, \vec{p}_C), (C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \in \text{Events} \right\} \subseteq \clockface$$

The event send_wait must be put before any of a node's execution that writes to this buffer. Therefore we put it before the first of these write accesses.

$$\theta_{\text{send\_wait}}(\chi, \vec{p}_G, \vec{p}_C) = \begin{pmatrix} \min_{\text{lex}} \text{GeneratorExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \\ -1 \end{pmatrix}$$

Certainly one wants to execute the wait primitives the last possible moment. Any statement between the first and the last generator instances potentially increases the total wait time by expanding the time the buffer is in use and therefore shrinking the window communication can happen asynchronously.

The send primitive is executed as soon as the last value has been written into the send buffer. Again, one does not want to waste any more time as it reduces the time available for the transfer without (busy) waiting.

$$\theta_{\text{send}}(\chi, \vec{p}_G, \vec{p}_C) = \begin{pmatrix} \max_{\text{lex}} \text{GeneratorExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \\ +1 \end{pmatrix}$$

The receive-operations on the destination node are analogous, but with the consumer statement instead.

$$\theta_{\text{recv\_wait}}(\chi, \vec{p}_G, \vec{p}_C) = \begin{pmatrix} \min_{\text{lex}} \text{ConsumerExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \\ -1 \end{pmatrix}$$

$$\theta_{\text{recv}}(\chi, \vec{p}_G, \vec{p}_C) = \begin{pmatrix} \max_{\text{lex}} \text{ConsumerExecTimes}\,(\chi, \vec{p}_G, \vec{p}_C) \\ +1 \end{pmatrix}$$

The domain space of these primitives are different than for normal SCoP statements, namely $\mathcal{D}_R \subseteq \mathcal{X} \times \mathcal{P}$. This is because they are executed once per chunk and destination on the source node (respectively chunk and source on the destination node). For send_wait and send, $\vec{p}_G$ equals the current node coordinate ($\pi_{\text{send\_wait}}((\chi, \vec{p}_G), \vec{p}_C) = \pi_{\text{send}}((\chi, \vec{p}_G), \vec{p}_C) = \vec{p}_G$) whereas it is $\vec{p}_C$ for recv_wait and recv.

Transfer sets that are empty should be pruned. No primitives are required. The lexical minimum and maximum are not defined for empty sets.

8.4.6 Message Sizes and Value Mapping

Given the transfer sets, how large does the buffer have to be? And what are the offsets for each value in the buffer? We present two approaches to these questions, the exact method and the bounding box hull.

Equation (8.2) ("Events") is the mapping of the elements $(F, \vec{k}_F)$ that a buffer ought to contain, the generator and consumer statements are not relevant here. Projecting them away yields a new mapping

$$\begin{aligned} \text{EventElements} \coloneqq \big\{ (\chi, \vec{p}_G, \vec{p}_C) \mapsto (F, \vec{k}_F) \;\big|\; \\ \exists C, \vec{i}_C, G, \vec{i}_C : ((\chi, \vec{p}_G, \vec{p}_C), (C, \vec{i}_C, \vec{p}_C), (F, \vec{k}_F), (G, \vec{i}_G, \vec{p}_G)) \in \text{Events} \big\} \end{aligned}$$

containing the elements $\subseteq \mathcal{E}$ to be transferred for each event.

8.4.6.1 Exact Method

The exact method finds the exact number of elements in a buffer and assigns an unique position for all of them. For simplicity, we assume that all elements occupy the same number of bytes. Alexander Barvinok's algorithm [50] find the the exact number of elements in a $\mathbb{Z}$-polyhedron as an expression of the structure parameters and input dimensions. The expression has the form of an Ehrhart quasi-polynomial, which is a set of N polynomials $p_i(x)$ selected by $i = x \mod N$ for some period N. Barvinok's algorithm has been implemented, for instance, in the barvinok library [51] of which newer versions are built on top of ISL.

Using this knowledge one can arithmetically find an Ehrhart quasi-polynomial that describes the number of elements in a buffer for each transfer set, namely

$$(\chi, \vec{p}_G, \vec{p}_C) \mapsto |\text{EventElements}\,(\chi, \vec{p}_G, \vec{p}_C)| \ .$$

This cardinality times the element's size in bytes is the minimum buffer size to be allocated and transferred per event.

The order of elements in the buffer can be determined by counting the number of lexicographical smaller elements – itself a (piecewise) $\mathbb{Z}$-polyhedron– in the transfer set an interpreting it as the offset in the buffer.

$$((\chi, \vec{p}_G, \vec{p}_C), (E, \vec{k}_E)) \mapsto \left| \left\{ (F, \vec{k}_F) \in \text{EventElements}\,(\chi, \vec{p}_G, \vec{p}_C) \;\middle|\; (F, \vec{k}_F) \ll (E, \vec{k}_E) \right\} \right| \quad (8.3)$$

The field names E and F can be compared by assigning them an arbitrary but unique ordinal number. For instance, `FieldA` gets 1 and `FieldB` the 2. This means that every element of `FieldA` is placed before those of `FieldB`. The lexicographically smallest element will be assigned an index of zero and therefore becomes the first element in the buffer. The element that only has the first element as smaller element is assigned an offset of one and therefore is put next into the buffer, etc.

A flaw of this method is that the quasi-polynomial (8.3) may be complicated. The most costly part probably is switching between the different pieces and polynomials because the processor may mispredict them, resulting in a performance penalty. If the number of pieces and polynomials is large, we also have a problem of code blowup. Computing the storage location is done often (for each buffer element) and SCoPs are likely performance-critical.

This method as already been tested for organizing scratchpads memories [52]. The goal of a scratchpad is speed-up of memory accesses. Ours is to compact messages, so code blowup is a smaller problem since it is dwarfed by the unnecessary transfer time of padding space like in the next method.

8.4.6.2 Bounding Box Method

The alternative assumes that the transfer set is a rectangular partition of the fields. Computing the offset is always simple but the trade-off is wasted memory and bandwidth if the transfer set is not rectangular.

Definition 8.6 (Axis-Aligned Box Hull)
The *bounding box hull* of a bounded subset $P \subset \mathbb{Z}^N$ is the superset

$$\{(x_i)_{i=1,\dots,N} \mid \forall i = 1, \dots, N : \min\{p_i \mid \vec{p} \in P\} \leq x_i \leq \max\{\vec{p} \mid \vec{p} \in P\}\} \ .$$

The minimum and maximum values of a coordinate in a bounded $\mathbb{Z}$-polyhedra can be computed by ISL. Those extrema cannot be dependent on other coordinates, but on structure parameters.

The "volume" of the box corresponds to the number of elements in the buffer can be computed by multiplying the box's side lengths. We define the following shortcuts.

$$\begin{aligned}
\text{MinCoord}_{F,i}\,(\chi, \vec{p}_G, \vec{p}_C) &= \min\left\{ k_{F,i} \;\middle|\; (F, \vec{k}_F) \in \text{EventElements}\,(\chi, \vec{p}_G, \vec{p}_C) \right\} \\
\text{MaxCoord}_{F,i}\,(\chi, \vec{p}_G, \vec{p}_C) &= \max\left\{ k_{F,i} \;\middle|\; (F, \vec{k}_F) \in \text{EventElements}\,(\chi, \vec{p}_G, \vec{p}_C) \right\} \\
\text{LenCoord}_{F,i}\,(\chi, \vec{p}_G, \vec{p}_C) &= 1 + \text{MaxCoord}_{F,i}\,(\chi, \vec{p}_G, \vec{p}_C) - \text{MinCoord}_{F,i}\,(\chi, \vec{p}_G, \vec{p}_C)
\end{aligned}$$

Offsets in the message are computed using either row-major or column-major order. The offset row-major order in a buffer of just one field variable is

$$\text{MessageOffset}_F\,(\chi, \vec{p}_G, \vec{p}_C, \vec{k}) = \sum_{i=1}^{N_F} k_i \prod_{j=i+1}^{N_F} \text{LenCoord}_{F,j}\,(\chi, \vec{p}_G, \vec{p}_C))$$

which results in a total size of the message data of

$$\text{MessageSize}_F\left(\chi, \vec{p}_G, \vec{p}_C\right) = \prod_{i=1}^{N_F} \text{LenCoord}_{F,i}\left(\chi, \vec{p}_G, \vec{p}_C\right))$$

elements.

If a message contains elements from different fields the elements are again appended to each other using an arbitrary but fixed order of fields. One also needs to consider that the size in bytes per element might be different for every field. The message size and offsets should therefore be specified in bytes instead of the number of elements.

$$\begin{aligned} \text{MessageByteOffset}\left(\chi, \vec{p}_G, \vec{p}_C, F, \vec{k}\right) = & \sum_{\substack{E \in \mathcal{F} \\ E < F}} \text{eltsizeof}\left(E\right) \cdot \text{MessageSize}_E\left(\chi, \vec{p}_G, \vec{p}_C\right) \\ & + \text{eltsizeof}\left(F\right) \cdot \text{MessageOffset}_F\left(\chi, \vec{p}_G, \vec{p}_C, \vec{k}\right) \\ \text{MessageByteSize}\left(\chi, \vec{p}_G, \vec{p}_C\right) = & \sum_{F \in \mathcal{F}} \text{eltsizeof}\left(F\right) \cdot \text{MessageSize}_F\left(\chi, \vec{p}_G, \vec{p}_C\right) \end{aligned} \tag{8.4}$$

Here, $\text{eltsizeof}\left(F\right)$ denotes the size in bytes of an element of field F. By definition of a field, every element has the same byte size.

The disadvantage of this scheme is that there might be unused offsets in the buffer. They occupy memory and are transferred between nodes, but the offset is fast to compute. In many cases, including all application use cases in this thesis, the transfer sets are rectangular anyway and therefore we currently use this scheme only.

8.5 Example: 2D Jacobi

We continue the example 2D Jacobi stencil in Listing 7.3 on Page 106. This time running on a cluster computer with 9 nodes in an 3×3 arrangement. There are only two arrays in the program of which both make sense to be distributed.

$$\mathcal{F} = \{\texttt{source}, \texttt{sink}\}$$

By default the fields use block-distribution. Both fields are of size 9×9 which yields a 3×3 block for every node.

$$\pi_{\texttt{source}} = \pi_{\texttt{sink}} = \{((p_1, p_2), (i_1, i_2)) \mid 3p_1 \leq i_1 < 3(p_1 + 1), 3p_2 \leq i_2 < 3(p_2 + 1)\} \tag{8.5}$$

We can distribute the statements between the nodes according to the rules in Section 8.2.1. Rule 1 says that if the SCoP leaks a non-distributed value, it must be computed on all the nodes. There is no such non-field variable in the example.

The variables `avg` and `sum` are not distributed meaning that their generators must be executed on all the nodes their values are read. This property is transitive. Statement S6 generates the non-field variable `avg` and reads `sum`, therefore the generator of `sum` must be executed on at least the nodes S6 executes. `S5` reads `sum` and assigns a new value to `sum` which again propagates the nodes to be executed on to statement S4, etc.

$$\pi_{\text{S1}} = \pi_{\text{S2}} = \pi_{\text{S2}} = \pi_{\text{S3}} = \pi_{\text{S4}} = \pi_{\text{S5}} = \pi_{\text{S6}} = \pi_{\text{S6}}$$

No execution has been fixed to concrete nodes yet. The remaining statements can be distributed arbitrarily and the necessary communication code will be generated between them, in contrast to uses of non-distributed variables.

Rule 2 is the corresponding rule for fields. The contents of `source` and `sink` are the ones from the last iteration. Therefore we execute their generator statements according to the distribution (8.5).

$$\pi_{\mathrm{S7}} = \pi_{\mathrm{S8}} = \{(i,x,y,p_1,p_2) \mid i = \texttt{ITERATIONS} - 1, 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1)\}$$

For the not yet distributed statements the dependence computes rule 3 states that statements are preferably executed where their generated value is going to be used. The statement S7 generates the elements used by statement S8. The first use of `source[x][y]` generated by S8 is S1 of which we already know has the same distribution as S7. We already fixed the distributions of S7 and S8 in the last iteration, therefore we can conclude by backtracking that these must be the same for all iterations.

$$\pi_{\mathrm{S7}} = \pi_{\mathrm{S8}} = \{(i,x,y,p_1,p_2) \mid 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1)\}$$

This outcome partially depends on the order of statements S1 to S4. If one of the other statements was first, the distribution relation would be considerably more complicated. It is sufficient for our purpose of showing the feasibility of the method.

Now we have all the information required to build the Transfers relation.

$$\begin{aligned}
&\text{Transfers} = \\
&\quad \{((\mathrm{S7},(i,x,y),(p_1,p_2)),(\texttt{sink},(x,y)),(\mathrm{S8},(i,x,y),(p_1,p_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1)\} \\
&\cup \{((\mathrm{S8},(i,x,y),(p_1,p_2)),(\texttt{source},(x,y)),(\mathrm{S1},(i+1,x+1,y),(q_1,q_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1), 3q_1 \leq x+1 < 3(q_1+1), p_2 = q_2\} \\
&\cup \{((\mathrm{S8},(i,x,y),(p_1,p_2)),(\texttt{source},(x,y)),(\mathrm{S2},(i+1,x-1,y),(q_1,q_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1), 3q_1 \leq x-1 < 3(q_1+1), p_2 = q_2\} \\
&\cup \{((\mathrm{S8},(i,x,y),(p_1,p_2)),(\texttt{source},(x,y)),(\mathrm{S3},(i+1,x,y+1),(q_1,q_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1), p_1 = q_1, 3p_2 \leq y+1 < 3(p_2+1)\} \\
&\cup \{((\mathrm{S8},(i,x,y),(p_1,p_2)),(\texttt{source},(x,y)),(\mathrm{S4},(i+1,x,y-1),(q_1,q_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1+1), 3p_2 \leq y < 3(p_2+1), p_1 = q_1, 3p_2 \leq y-1 < 3(p_2+1)\}
\end{aligned}$$

There are no statement instances that are executed on multiple nodes and no choice of value origins. The consequence is Transfers = MinDistance.

The following step is to find a chunking function. Following the heuristic of Algorithm 8.2 (Page 120) the maximal code regions that do not violate any dependencies are grouped with the first instances of the generator's chunk becoming an representative. In this example, there is no data flow withing the innermost x and y-loops so they become chunks. In contrast, data written to `sink` and `source` are used within respectively in the following iteration of the i-loop.

$$\begin{aligned}
\varphi : \sigma &\to \mathbb{Z}^7 \\
(S7,(i,x,y),S8,(i,x,y)) &\mapsto (0,i,0,\vec{0}) \\
(S8,(i,x,y),S1,(i+1,x,y)) &\mapsto (0,i,1,\vec{0}) \\
(S8,(i,x,y),S2,(i+1,x+1,y)) &\mapsto (0,i,1,\vec{0}) \\
(S8,(i,x,y),S3,(i+1,x-1,y)) &\mapsto (0,i,1,\vec{0}) \\
(S8,(i,x,y),S4,(i+1,x,y+1)) &\mapsto (0,i,1,\vec{0}) \\
(S8,(i,x,y),S5,(i+1,x,y-1)) &\mapsto (0,i,1,\vec{0})
\end{aligned}$$

As a result, the relation Chunks assigns the chunk equivalence classes $(0, i, 0, \vec{0})$ and $(0, i, 1, \vec{0})$ to the elements to be transferred in a message.

$$
\begin{aligned}
&\text{Chunks} = \\
&\quad \{(0, i, 0, \vec{0}) \mapsto ((\text{S7}, (i, x, y), (p_1, p_2)), (\texttt{sink}, (x, y)), (\text{S8}, (i, x, y), (p_1, p_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1)\} \\
&\cup \{(0, i, 1, \vec{0}) \mapsto ((\text{S8}, (i, x, y), (p_1, p_2)), (\texttt{sink}, (x, y)), (\text{S1}, (i, x, y), (p_1, p_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1)\} \\
&\cup \{(0, i, 1, \vec{0}) \mapsto ((\text{S8}, (i, x, y), (p_1, p_2)), (\texttt{sink}, (x, y)), (\text{S2}, (i, x + 1, y), (q_1, q_2))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1), 3q_1 \leq x + 1 < 3(q_1 + 1), p_2 = q_2\} \\
&\cup \ldots
\end{aligned}
$$

Reordered by sending and receiving node we have the mapping of messages to their contents.

$$
\begin{aligned}
&\text{Events} = \\
&\quad \{((\text{S1}, (i, 0, 0)), (p_1, p_2), (p_1, p_2)) \mapsto ((\text{S7}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S8}, (i, x, y))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1)\} \\
&\cup \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (p_1, p_2)) \mapsto ((\text{S8}, (i, x, y))), (\texttt{sink}, (x, y)), (\text{S1}, (i, x, y))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1)\} \\
&\cup \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2)) \mapsto ((\text{S8}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S2}, (i, x + 1, y))) \mid \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), 3p_2 \leq y < 3(p_2 + 1), 3q_1 \leq x + 1 < 3(q_1 + 1), p_2 = q_2\} \\
&\cup \ldots
\end{aligned}
$$

No messages are required when the source and destination node are equal. We can therefore remove those from the relation. They are handled more efficiently when written to a buffer in the node's local memory. The consumer statements then read the values from there. The local buffer therefore is a combined send- and receive-buffer. Alternatively, if the elements of this buffer are a subset of the locally stored field elements, the field's local storage may be used as well. The remaining inter-node transfer sets are

$$
\begin{aligned}
&\text{Events}' = \\
&\quad \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2)) \mapsto ((\text{S8}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S2}, (i, x + 1, y)))\} \\
&\qquad 3q_1 = x, p_1 = q_1 + 1, 3p_2 \leq y < 3(p_2 + 1), p_2 = q_2\} \\
&\cup \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2)) \mapsto ((\text{S8}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S3}, (i, x - 1, y)))\} \\
&\qquad 3p_1 = x, p_1 + 1 = q_1, 3p_2 \leq y < 3(p_2 + 1)p_2 = q_2\} \qquad (8.6) \\
&\cup \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2)) \mapsto ((\text{S8}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S4}, (i, x, y + 1)))\} \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), p_1 = q_1, 3q_1 = y, p_2 = q_2 + 1\} \\
&\cup \{((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2)) \mapsto ((\text{S8}, (i, x, y)), (\texttt{sink}, (x, y)), (\text{S4}, (i, x, y - 1)))\} \\
&\qquad 3p_1 \leq x < 3(p_1 + 1), p_1 = q_1, 3p_1 = y, p_2 + 1 = q_2\} \ .
\end{aligned}
$$

For the event $((\text{S8}, (i, 0, 0)), (p_1, p_2), (q_1, q_2))$, the first generator according to the schedule θ is the instance $(\text{S8}, (i, 3p_1, 3p_2))$ and the last one is $(\text{S8}, (i, 3p_1 + 2, 3p_2 + 2))$. Their scatterings are $(0, i, 1, 3p_1, 0, 3p_2, 0)$ respectively $(0, i, 1, 3p_1 + 2, 0, 3p_2 + 2, 0)$. In order to execute the send primitives before/after this, an 8th scatter coordinate is added. Similarly, the first consumer instance is $(\text{S2}, (i, 3q_1, 3q_2))$ and the last $(\text{S5}, (i, 3q_1 + 2, 3q_2 + 2))$. Their scatterings are $(0, i, 0, 3q_1, 0, 3q_2, 1)$ and $(0, i, 0, 3q_1 + 2, 0, 3q_2 + 2, 5)$. The communication primitives are

therefore scheduled at the following positions.

$$\begin{aligned}
\theta_{\text{send_wait}} &= (0, i, 1, 3p_1 \quad , 0, 3p_2 \quad , 0, -1) \\
\theta_{\text{send}} &= (0, i, 1, 3p_1 + 2, 0, 3p_2 + 2, 0, +1) \\
\theta_{\text{recv_wait}} &= (0, i, 1, 3p_1 \quad , 0, 3p_2 \quad , 0, -1) \\
\theta_{\text{recv}} &= (0, i, 1, 3p_1 + 2, 0, 3p_2 + 2, 0, +1)
\end{aligned}$$

The primitive's schedules are dependent on the nodes they are executed on. send_wait and send are executed on the transfer event's source nodes (p_1, p_2), respectively recv_wait and recv on the destination node (q_1, q_2).

How many elements are transferred in a message? In other words, what is the set of field indices (x, y) in Events' with fixed chunk, source and destination node? Let (p_1, p_2) a node sending data to a node $(p_1 - 1, p_2) = (q_1, q_2)$. Only the first part of (8.6) applies to this case. The index set is independent of i and x is fixed to $3p_1 = 3(q_1 + 1)$, the lowest x-index of the sending node. There is no data to be sent if $p_1 = 0$, it has been excluded by the if-statements of the example program. The y-index's range is $3p_2 \leq y < 3(p_2 + 1)$ with 3 elements.

$$\text{EventElements}\,(\chi, (p_1, p_2), (p_1 - 1, p_2)) = \{(3p_1, 3p_2), (3p_1, 3p_2 + 1), (3p_1, 3p_2 + 2)\}$$

Using the bounding box model from Section 8.4.6.2 we get

$$\begin{aligned}
\text{MinCoord}_F\,(\chi, (p_1, p_2), (p_1 - 1, p_2)) &= (3p_1, 3p_2) \\
\text{MaxCoord}_F\,(\chi, (p_1, p_2), (p_1 - 1, p_2)) &= (3p_1, 3p_2 + 1) \\
\text{LenCoord}_F\,(\chi, (p_1, p_2), (p_1 - 1, p_2)) &= (1, 3)
\end{aligned}$$

and an element ordering rule of

$$\begin{aligned}
&\text{MessageOffset}_F\,(\chi, (p_1, p_2), (p_1 - 1, p_2), (x, y)) = y - 3p_2 \\
&\text{MessageSize}_F\,(\chi, (p_1, p_2), (p_1 - 1, p_2)) = 3\ .
\end{aligned}$$

9 Molly

Molly is the implementation of the parallelization technique from the previous chapter as an extension to the toolchain consisting of Clang, *LLVM*, *Polly* and ISL. These existing components allow such an implementation without re-implementing a compiler from scratch; relatively few parts have to be added to create Molly.

Clang is a modern C, C++ and Objective-C compiler frontend. Its development has been initiated to compete with GCC which switched to GPL version 3 that some companies deem unacceptable. Clang takes source files of any of those three languages, or even a mix of them, as input and outputs lowered code in LLVM intermediate representation. By now it is a sub-project of LLVM.

LLVM is a compiler backend and code optimizer. Some of the target architectures are *x86*, ARM, PowerPC (including Blue Gene/Q subtarget), AMD and NVIDA GPUs. Initially developed as the project of a master's thesis [53], it now developed more into the direction of an industry-quality compiler backed.

Polly also originates from a master's thesis [54]. It is a plugin for LLVM for polyhedral optimizations the same way the GCC project *Graphite* [55] does. Recently it also became an official LLVM sub-project.

ISL is a library for $\mathbb{Z}$-polyhedra used by Polly. It can also analyze, optimize SCoPs and re-generate code. Section 7.1 already described this library.

Molly, a plugin like Polly, also an extension to LLVM, adds memory transformations to the LLVM toolchain. The emphasis of this thesis is to distribute multi-dimensional arrays to the nodes of a DMM, but other optimizations are possible as well.

9.1 C++ Language Extensions

Molly is not able to distribute arbitrary programs. Such transformations may slow down programs because of the overhead of inter-node communication or even result in wrong results because the program uses pointer access to data that Molly moved to another node. Therefore this section introduces ways for the programmer to tell Molly what it can optimize.

9.1.1 Declaring a Distributed Array

Automatic detection of fields has not been implemented in Molly and therefore the programmer is required to explicitly specify which arrays are to be distributed among the nodes. Fields also impose restrictions on array accesses. Therefore we introduce an API that passes on information to the Molly optimizer. The semantics should be preserved when Molly is deactivated or when compiling with a different compiler than Clang.

The Molly API is imported by #including the header file `molly.h`. Any of its declarations are put into the namespace "molly" or prefixed with `__molly`. Note that the double-underscore

is reserved by the C++ standard for compiler-internal use so user programs do not break because they already declared some symbol with the same name.

The primary API member is `molly::array`, a variadic-templated class for declaring distributed arrays. The first template arguments is the array's element type, followed by any number of dimension lengths. Such arrays are always rectangular with zero-based subscripts, just like standard C arrays.

For instance,

```
#include <molly.h>
molly::array<double,9,9> field;
```

declares two-dimensional array of 9^2 floating-point values. An equivalent array in C syntax is

```
double field[9][9];
```

As a result, fields declared this way always have a fixed number of dimensions of sizes determined at compile-time. Dynamically-sized fields are not supported and would require a separate type.

Both declarations of fields are used the same way using the subscript operator with angular brackets:

```
field[0][0] = 0.0;           // Write access
double result = field[0][0]; // Read access
```

An iteration over all elements of the field can be done as usual by nested for-loops as shown as method A shown in Listing 9.1.

```
// Method A (Element iteration)
for (int i = 0; i < 9; ++i)
  for (int j = 0; j < 9; ++j)
    field[i][j] = 0.0; // Initialize all elements to zero

// Method B (Pointer iteration)
for (bool *p = &field[0]; i <= &field[9*9-1]); ++p)
  *p = 0.0; // Initialize all elements to zero

// Method C (memset)
memset(&field[0], 0, 9*9*sizeof(bitmap[0]));
```

Listing 9.1: Different methods to clear arrays

The semantic difference between a C-style array and `molly::array` is that the C-standard fixed the order of the elements to row-major ([56, Section 6.5.2.1]). Therefore it is legal to iterate over all elements by incrementing a pointer 9^2 times, or by using memset to zero all the memory at once (Methods B and C in Listing 9.1).

These become illegal with `molly::array`. Elements may be ordered differently, not stored consecutively or with padding, or not even in local memory. Because of this change of the semantics, Molly will not promote arbitrary arrays to distributed ones. The programmer has to state that the stricter rules are followed by declaring arrays of type `molly::array`.

9.1.2 Pragmas

A pragma is an annotation within C/C++ source code that conveys additional information to the compiler. Probably best-know are the pragmas to switch off compiler warnings and those that belong to OpenMP. For instance,

```
#pragma omp parallel for
for (int i = 0; i < 128; ++i)
  ...
```

suggests to execute the loop in parallel. The code is supposed to have the same semantics when the pragma is removed or OpenMP is deactivated. Therefore it serves as a hint to the compiler what it can parallelize. It may also parallelize the loop without the pragma if it can ensure that the semantics are preserved. The XLc does this at higher optimization levels (`-qsmp=auto`).

9.1.2.1 Data Distribution Pragma

We use pragmas to suggest an element distribution to Molly. If omitted, Molly chooses a distribution by itself. Currently this is the block-distribution from Section 8.1.2, but future versions may select a distribution based on access pattern and alignment algorithms.

The syntax of this `#pragma molly transform` is shown in the example below which will use the second index j to identify the node at which the value is stored and the first index i to order the elements on that node.

```
#pragma molly transform("{ [i,j] -> [rank[j], local[i]] }")
molly::array<double,9,9> field;
```

The argument of the transform-clause describes a polyhedral map (Section 7.1.1.4) parsed by ISL. It consists of an input tuple (`[i,j]`) that corresponds to the field's logical domain and two named output tuples, `rank` and `local`. The tuple names themselves are not relevant, but their order. The first is the coordinate of a node and therefore must correspond to the shape of the cluster. In this example, the cluster must be a one-dimensional mesh of (at least) 9 nodes. The second tuple `local` is optional and specifies a coordinate in memory on the local node. If omitted, the identity mapping is assumed – `local[i,j]` in this case. The reserved local memory per node is compacted to contain just those elements that are stored on that node.

When combined, the first and second tuples must be injective, otherwise multiple values overlap in memory which for obvious reasons must not happen. But a single coordinate can have multiple locations. In this case, if the value is read, the local value will be read, or an undefined one if none of them is local. Though, this increases the cost if a new value is written at that coordinate as all the location must be kept up-to-date.

The current syntax is the syntax used by ISL to define polyhedral maps. This makes some patterns awkward to use as there is no possibility to use the field's shape in the definition. This problem might be addressed in a future version.

9.1.2.2 Statement Distribution Pragma

Programmers may also explicitly specify on which node a statement should be executed, like in the example below.

```
for (int i = 0; i < 9); ++i)
  for (int j = 0; j < 9; ++j)
#pragma molly where("[i,j] -> rank[i+j]")
    field[i,j] = 0.0;
```

This guarantees that `field[i,j] = 0.0;` will become at least one (or multiple, the compiler is allowed to split up further) separate statement that will be executed on the node with rank $i + j$. Some form of communication will be involved if the distribution of `field` is different.

The statement may also be executed on multiple nodes. In case it writes to a distributed field, all the instances should have the same result such that it does not matter which one of the results is actually written to the field element. Unused results may be removed by the compiler in a later optimization phase.

9.2 The Toolchain

Molly requires some additions to the standard compiler toolchain. First, Clang needs to create metadata for distributed arrays. Another optimization pass is added to LLVM. And there is another runtime library – *MollyRT*– that implements the functionality of transferring data between nodes.

Figure 9.1 shows the Molly toolchain with different optimization levels. The first step is the frontend Clang that translates a C++ program to LLVM *intermediate representation* (IR) which is then processed by LLVM itself.

What happens inside LLVM depends on the optimization level. Without any optimization (-O0) the IR is directly translated to machine assembler code. With optimizations enabled (-O1 to -O3, -Os or -Oz) the code goes into a normalization phase[1]. An example for a normalization is the one for induction variables: Every loop that resembles a Fortran for-do loop is normalized such that the loop counter starts at zero and increases by one each iteration. The old loop counter becomes a linear expression of the new loop counter. The main optimzations, like strength reduction of expressions that depend on the loop counter, use this normalized form before passing the code to the target-specific backends.

Polly intercepts the compilation after the normalization. More precise, it is an early pass in the pass pipeline and inserts some normalization passes before it runs itself. The analysis part searches maximal code regions that are valid SCoPs. If it found one, it extracts the statements and creates a separate data structure that represents the SCoP. Polly's optimization passes take the information about found SCoPs and try to improve them. Optimizations include parallelization, vectorization and improving locality of accesses.

After a SCoP has been modified, Polly's code generator synthesizes them back to IR. All other LLVM optimization passes do not understand Polly's SCoP representation and only process the IR.

Molly inserts itself as the first Polly optimization pass. It may analyze how the SCoPs access the memory of a field and change the field's memory layout accordingly or according to the metadata stored by Clang. In case of distributed memory, Molly knows the cluster's node arrangement and inserts communication primitives accordingly.

In this model the remaining part follows the standard C++ toolchain with the exception that the linker adds another runtime library, MollyRT, that implements the communication primitives.

9.2.1 The `molly.h` Header File

A very brief synopsis of the `molly.h` header is shown below. The class `LocalStore` will allocate the memory required for the current node at runtime on the heap. `length` is implemented using a recursive template such that it can be used in contexts where a constant is required (C++11 `constexpr`).

```
namespace molly {
  class LocalStore {
    void *data;
  };

  template<typename T, size_t... L>
  class array : LocalStore {
    array() {
      __builtin_molly_field_init(this, L...);
    }

    T &operator[](size_t idx) {
```

[1] This is an simplification; There are several optimization phases of which some can be seen as normalization such as dead code elimination

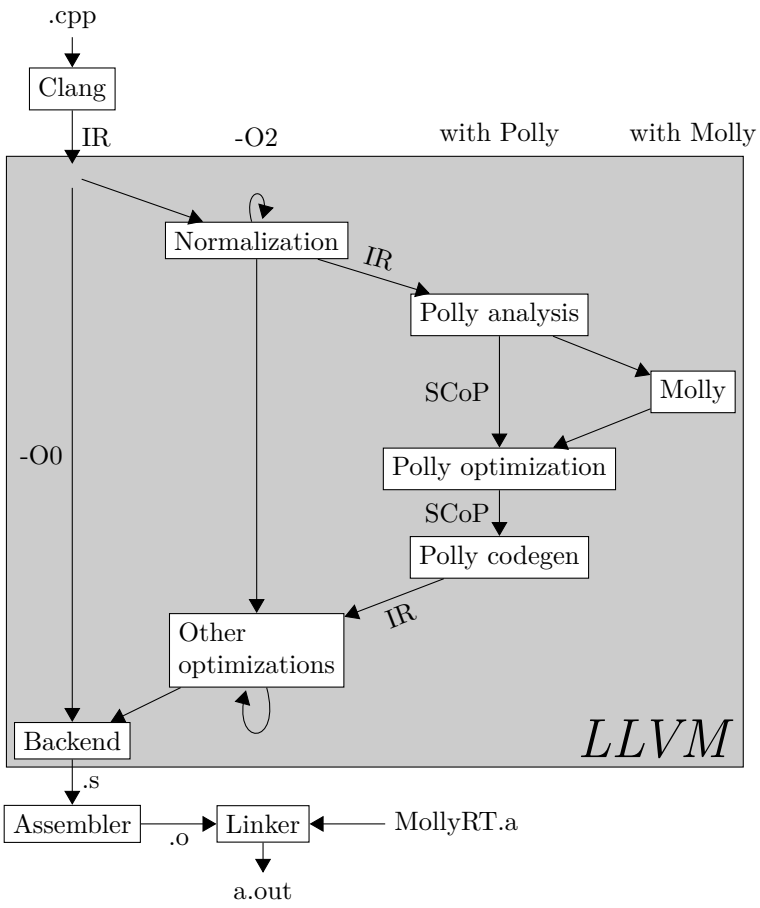


Figure 9.1: Molly toolchain interior

```
      return *__builtin_molly_ptr(this, idx);
    }

    constexpr size_t length(size_t t);
  };
}
```

The implementation of `operator[]` is actually more complicated than it looks here. Depending on the number of template parameters it either returns a reference to the element or a proxy object if the array is at least two-dimensional. The proxy stores the indices that have been appended so far and resolves to a reference to the actual element if as many subscripts as dimensions have been specified. If the proxy object is indexed before the last coordinate is reached, it returns a new proxy object with the new index appended to the stored indexes.

9.2.1.1 Builtins

Most of the compiler's magic[1] happens with the call to `__builtin_molly_ptr`. It is the logical equivalent to LLVM's `GetElementPtr` (GEP) [57]. `GetElementPtr` is the instruction that adds offsets to pointers. Applied to arrays and pointers, it evaluates to the address of a specific

[1]The term "compiler magic" refers to exceptional functionality and semantics not possible to recreate with the programming language's orthogonal rules. For instance, in Pascal, `SetLength` accepts any number of arguments, but there is no way to declare such a function.

element. Applied on structures, it returns the address to an element of the structure by adding its offset. `GetElementPtr` therefore can, and will be when hitting the backend, lowered to a combination of multiplications and additions. It never accesses memory by itself.

`__builtin_molly_ptr`, in contrast, is applied to fields instead of arrays and structures. It accepts an argument for the field to be referenced and arbitrary many indexes. Of course, the number of indexes should match the number of dimensions of the field.

The pointer returned is merely symbolic. The element in question may not be local to the node and therefore cannot be referred to using a pointer. It is therefore up to the Molly pass to fix this and add some transfer between nodes to read or write the value accessed.

The *builtin* `__builtin_molly_field_init` is called before any field is used. It ensures that each field is initialized for the right layout. It mainly allocates the memory required to hold the local data. The allocation size depends on how Molly decides to distribute the data and therefore cannot be known by the Clang frontend which runs before that.

9.2.1.2 Attributes

Most of `molly::array`'s methods are annotated using attributes. Attributes are a new feature of C++11. Alternatively, the syntax `__attribute__((...))` introduced by GCC is supported as well. One of the purpose of these attributes is to allow calls to internal methods in SCoPs. Usually, Polly rejects any SCoP that contains function calls because their potential side effects. However, `molly::array` is explicitly designed to work within SCoPs. The attribute `[[molly::pure]]` (alternatively: `__attribute__((molly_pure))` marks a function as free from unexpected side-effects. The qualifier `constexpr` is meant for the frontend only and not passed over to the IR by Clang. We did not just reuse the existing `__attribute__((pure))` because it already has semantics which we did not want to overload with a potentially slightly different meaning.

9.2.2 Clang: Parse

Only few changes had to be applied to Clang's C++ parser. It must, of course, recognize the new builtins and attributes. The clang build process has two central files that contain all builtins and attributes known to Clang: `Builtins.def` contains all the builtins and `Attr.td` all known attributes. These are just the two root files, both include other files that contain items for more specific purposes.

`Builtins.def` is a C-preprocessor file. It is `#include`d wherever required. Molly appends another line

```
#include "BuiltinsMolly.def"
```

and the new file `BuiltinsMolly.def` contains the Molly-specific builtins.

On the other side `Attr.td` is an input file for "tblgen", an LLVM- and Clang-specific code generator that is embedded into their build processes. This tool generates C++ classes as specified in `td` input files. Again, we append the line

```
include "clang/Basic/AttrMolly.td"
```

into the main `Attr.td` and insert the attribute descriptions into the included `AttrMolly.td`.

Unfortunately, there is no such file for the pragmas and therefore they had to be implemented manually. The additional code tries to mimic the mechanism of the OpenMP pragma parser.

9.2.3 Clang: IR Generation

Any call to `__builtin_molly_ptr` are translated to calls to the `llvm.molly.ptr` intrinsic. In the terminology used by Clang (and GCC), a builtin is a function that does not have an implementation in a library, but is transformed to something else by some compiler magic. An

example are target-specific assembly-instructions that have no equivalent expression in the frontend language like `__builtin_popcount` which counts the number of set bits in an integer.

In LLVM terms, its equivalent is an *intrinsic*, not to confuse with the same term as used in the sense of Clang builtins by XLc (Sections 4.2 and 4.3). The population count is translated to a call to the `llvm.ctpop` intrinsic. Depending on the target instruction set this either is translated to an assembler instruction or to a call to a runtime library that implements the semantics. Intrinsics are used to avoid introducing new operations to the intermediate representation that every pass must cope with and instead are modeled using a function call to a set of predefined functions.

The Molly pass has to transform the `llvm.molly.ptr` intrinsic to something else before the instruction selection phase in the backend. Indeed, the significant difference between distributed- and shared memory computer architectures is that the distributed architecture has no dereferencable pointers to remote memory[1].

In addition, Clang is required to pass on some metadata like the shape of the field, the `#pragma molly transform` annotation and the emitted symbol names of methods from `molly::array`. The template parameters and other meta-information are not preserved in the intermediate representation.

The field shape is associated with the class type `molly::array`. Unfortunately LLVM IR has not syntax to add metadata information to types. Additional information can be passed using function attributes and `MDNode`s (metadata nodes) that can be attached to instructions, modules[2] and used as arguments when calling intrinsics.

The modified Clang appends an `MDNode` to the call of an intrinsic `llvm.molly.field.init`. The metadata node contains information about the field's shape and the requested layout of the transform pragma. It is inserted in place of the `__builtin_molly_field_init` builtin call. In addition, Clang adds a list of all fields and their metadata to the module under the global metadata node `molly.fields`.

Information about `#pragma molly where` is attached to any IR instruction in the pragma's scope. The `MDNode` instance also identifies to which statement an instruction belongs, such that a reconstruction of the statement is possible.

9.2.4 Polly: SCoP Analysis

Polly by its nature is pretty rigid about what it accepts in a SCoP. This is understandable since any non-predictable behavior has the power to change the semantics after transformation. For instance, if a function call is found, this part cannot be part of a SCoP because the order of the calls might be significant.

Polly inserts some code preparation passes before its own analysis. For instance, the IndVarSimplify pass transforms loops into a canonical representation to allow easy detection and handling. It ensures that the loop counter (also called induction variable) starts at zero and is increased by one in every iteration. The original loop variable becomes a linear expression in terms of the canonical loop counter.

Together with some more passes required by Polly this is the normalization phase that enables Polly to detect SCoPs. The only change required here is to make Polly accept `llvm.molly.ptr` with affine indices as part of a SCoP, the same way it already accepts `GetElementPtr` for array index calculation. Molly will handle the semantic differences.

Normalization passes such as *SROA* (Scalar Replacement of Aggregates) and LLVM's inliner also run when standard optimization mode is activated, even when Polly or Molly are disabled.

[1] Except PGAS and similar technologies. Transfers must still be invoked explicitly by the programmer.

[2] An LLVM module is the result of compiling a C++ translation unit; the representation of an object file (.o)

9.2.5 Polly: Optimization

Polly's main task is the optimization of the SCoP. The SCoP information may either originate directly from the analysis pass or be the the result of a modification by Molly. Optimization here means to change the execution order of the statements which potentially includes to parallelize them. Of course, optimization can also be skipped.

Figure 9.2 shows a refinement of the "Polly Optimization" pass in Figure 9.1. It consists of a dependence analysis and one of three optimization engines: PlutoOptimizer, IslScheduleOptimizer and PoccOptimizer. The dependence analysis determines $<_{\text{flow}}$, $<_{\text{anti}}$ and $<_{\text{output}}$ dependencies between statement instances that serve as input to the optimizer algorithms.

All three actually use the algorithm described in the article [40], just the implementation is different. PlutoOptimizer uses its libpluto library implementation written by the authors of [40]. PoccOptimizer calls the standalone executable from the *Polyhedral Compiler Collection* (PoCC, [58]). The distinction is made because either libpluto or PoCC may not be installed on the system.

IslScheduleOptimizer on the other side uses the more recent re-implementation that is part of ISL. ISL is a mandatory for Polly, therefore IslScheduleOptimizer is always available.

The optimizers return a potentially non-injective schedule function θ. Instances with the same scattering can be executed in parallel.

Statements inserted by Molly are handled like any other statement. Such statements have additional dependencies such that any access to a receive-buffer is dependent on the corresponding recv_wait statement. This is already handled correctly and therefore no changes to are Polly required.

9.2.6 Polly: IR Generation

Polly's final part is to generate IR code again from the statement instances. Again, Polly supports two engines: Cloog (Chunky Loop Generator) and ISL. Both return abstract syntax trees when given the schedule function θ. Polly converts these syntax trees to LLVM IR code. Thereafter the SCoP information can be released and optimization can continue with any LLVM pass.

Cloog's syntax tree representation is called "Clast". Polly's converter to IR supports three kinds of parallelism: Vectorization of the innermost loop, OpenMP parallelization and kernel generation for PTX targets. PTX is the abstract ISA for NVIDIA GPUs. Unfortunately, the OpenMP generation is in conflict with the implementation of OpenMP in newer versions of Clang. Polly targets the libgomp library (GNU OpenMP from GCC) while Clang links to libomp ("OpenMP Runtime", formerly an Intel library, now part of the LLVM project).

The ISL version also consists of two passes. The IslAstInfo pass asks ISL to generate the abstract syntax tree. The second pass, IslCodeGeneration converts it to IR code. It only supports vectorization, but no OpenMP parallelization nor GPU kernel generation. Again, ISL is required by Polly itself and therefore always available, but Cloog is optional.

The old code is wrapped into a `if (false)` statement such that it will be removed by the dead code elimination pass. Ideally, if neither Polly nor Molly did not make any changes or did not run at all, the generated IR should be the same as before the SCoP detection. Some normalization might be necessary to undo statement isolation like fusing adjacent basic block etc.

As for the optimizers, no adaptation for Molly is required. Both IR generators copy the statement basic blocks from the old code to the new part while adapting it to the new loop environment. For the Molly-generated statements these blocks did not previously exist in the source code. Instead, they are generated directly in the main Molly pass. This code will also be wrapped as dead code and therefore never executed. Their only purpose is to be copied by the IR generators. An alternative implementation would detect the non-existing old blocks and

generate the required code on-the-fly. This is less overhead, but requires two implementations, one for each generator.

9.3 The Molly Pass

Molly is the new component of the LLVM toolchain that converts sequential code with random access to memory to an SPMD program, i.e. code that runs in parallel on DMM. If nothing else is required then this is a trivial task. Such sequential code is already a program that runs on cluster, but uses one node only. This is, of course, not the idea of DMMs. What we actually want is a program that runs as fast as possible or in other words, an optimized program.

Molly is actually not a single pass, but a collection of passes, most of them shown in Figure 9.2, a more detailed version of Figure 9.1. Here we assume that Molly is activated and follows the control flow.

The core of Molly is a global pass that runs on a module as a whole. The prototype also assumes that it is the only module of the program that uses fields. For this to be true in a larger program the Molly pass would need to be run by the linker.

The pass manager of LLVM is not flexible enough to organize the phases of Molly by itself. For instance, coarse grained passes cannot access analysis results of more fine-grained passes[1]. A module-level pass cannot get the analysis results of all SCoPs in the module. This is required if the module pass tries to heuristically determine a data distribution for fields that matches how the field is accessed. Moreover, each pass has to specify which analysis results are still valid after running the pass. Any non-Molly pass is not aware on the existence of Molly's field analysis and therefore cannot ask the pass manager to preserve it. The pass manager therefore will happily throw away any of Molly's results if a non-Molly pass runs in between. This is, for instance, the case for the Polly passes than run in between. Molly therefore uses its own pass manager for better control of passes.

9.3.1 Preparation Passes

LLVM receives the intermediate representation from Clang with the metadata as described in Section 9.2.3. The pipeline begins with normalizing the code generated by Clang. It is required because later phases do not recognize all variations of a specific code pattern.

For example, the type `molly::array` is mostly implemented using method calls and proxy objects, like the call to `llvm.molly.ptr` which is hidden behind a chain of functions. These must be inlined into the SCoP until `llvm.molly.ptr` is called directly. Molly requires a direct reference to the variable, not the pointer to it passed as a function argument that might point to an arbitrary location. Therefore a special pass "MollyInliner" unconditionally inlines any function that has the attribute `[[molly::inline]]`. These proxy functions are very simple and extremely optimizable.

This is followed by an selection of standard LLVM and Polly canonicalization phases. These include control flow graph simplification, expression reassociation, algebraic simplification, SROA, memory-to-register promotion and tail call elimination. This is to remove the boilerplate from various implementation tricks and inlining.

SROA for instance converts the proxy object returned by `molly::array::operator[]` into its components such that in the end, the indices passed to `operator[]` are direct arguments of `llvm.molly.ptr`. Non-inlined methods are still valid with the `[[molly::pure]]` already presented in Section 9.2.1.

[1] With the exception that module-level passes may run function-level passed "on-the-fly", but for just one function at a time.

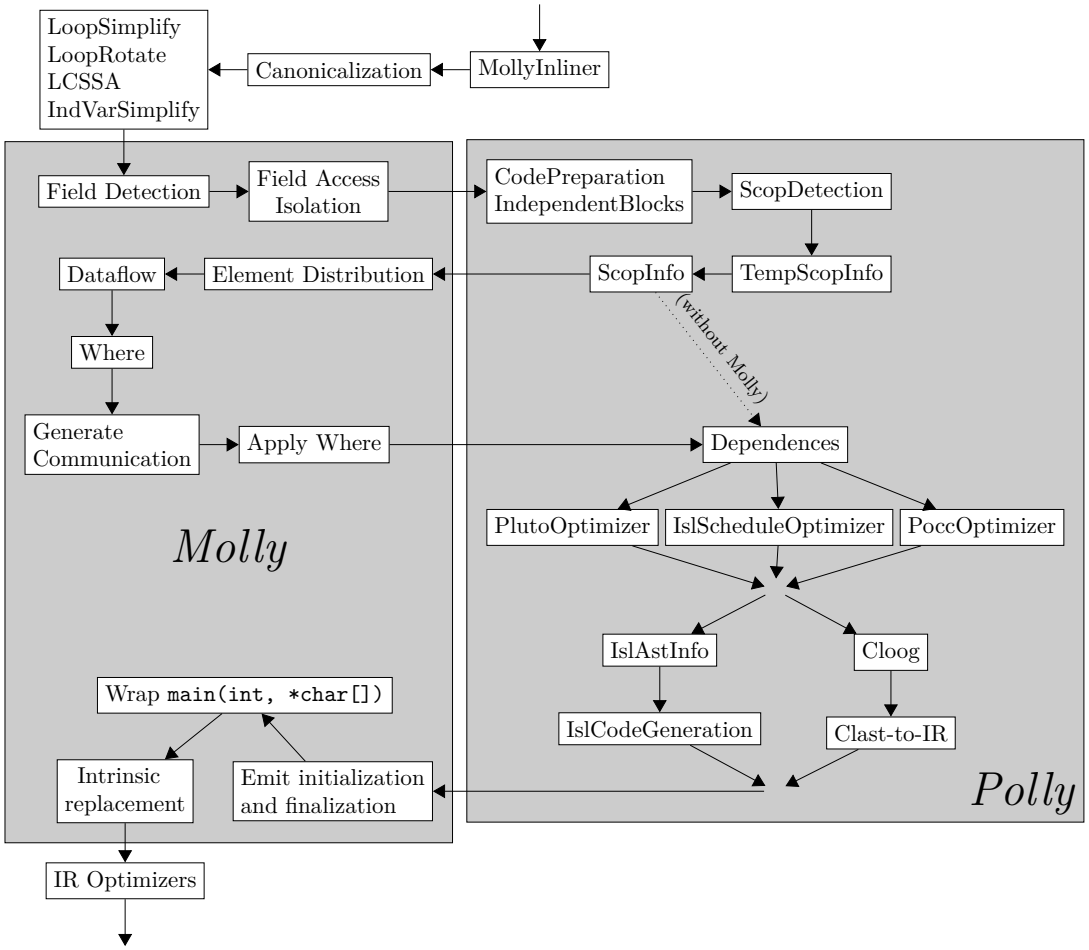


Figure 9.2: Molly pass internals

9.3.2 Field Detection

This pass searches the module for field declarations. Information is taken from the metadata attached to `llvm.molly.field.init` intrinsic calls and the list found in the module's `molly.fields` named metadata. It creates objects representing the fields and their user-defined layouts (`#pragma molly transform`).

9.3.3 Field Access Isolation

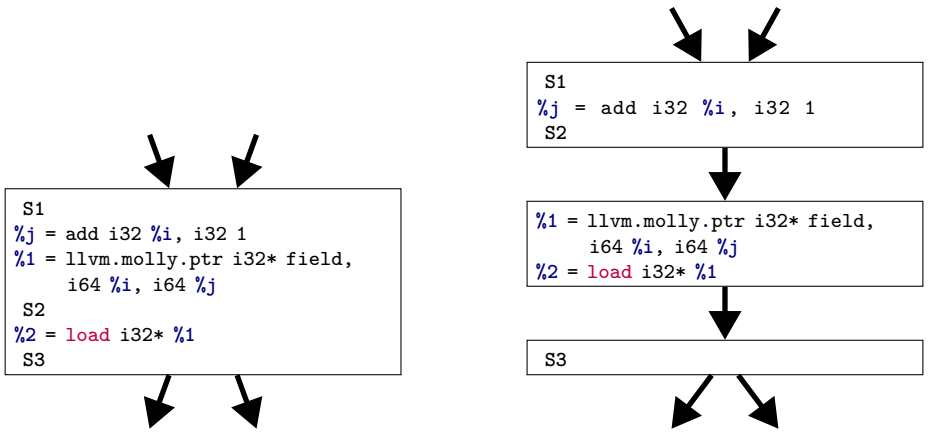
When a statement instance accesses a field element, the access may either be to the node's location storage, or to a buffer received from/to be send to another node, depending on the loop induction variable's value. The parallelized code must be prepared for both cases; the latter may also involve multiple buffers because the element in question is related to multiple remote nodes. The statement instance must specialize for all these cases. If the statement accesses even multiple elements, the code must be prepared for all combinations of how an the different accesses are carried out, i.e. an exponential blowup in the number of accesses.

For this reason we ensure that field accesses are always embedded into their own statements. The statements can be manipulated independently, for instance creating multiple versions for different kinds of accesses each with disjoint domains $\mathcal{D}_S$, but same scatter function. What

we get is three types of statements: Statements that read from a single field element, those that write to a single field element, and those with only non-field accesses. Of course, one cannot assume that existing code adheres this requirement. This pass isolates any field access by splitting basic blocks.

From Molly's point of view a field access is a call to `llvm.molly.ptr` followed by a load or store of the returned pointer. The "access" here is the load or store; to find out which field element it accessed, one has to look at the pointer's origin. If it is an `llvm.molly.ptr`, then it is a field access. Otherwise it is an access to the node's local memory. Figure 9.3a shows an example of a read access to an element `field[i, i+1]`.

Once a field access has been identified, the basic block is split into three parts: The block before the access, the block containing the load or store, and everything following the access (see Figure 9.3b). The `llvm.molly.ptr` is also moved into the middle basic block as it virtually belongs to that access. Other instructions (In the figure S1, `add`, S2 and S3) stay in either the before-block or the after-block.



(a) Before field access isolation

(b) After field access isolation

Figure 9.3: Field access isolation

The basic blocks will be further processed later by Polly's IndependentBlocks pass. Its main task is to make any reorderings of basic block syntactically valid. For instance, an optimizer may decide to move a basic block that contains a use of a value before the value's definition. In *Static Single Assignment* (SSA) form this is invalid because the value does not exist before it has been defined.

IndependentBlocks undoes SSA for values that span between basic blocks. For each such value it allocates memory for a variable on the stack. If the basic block contains a use of the value then a load instruction for that variable is issued. The basic block containing the value's definition stores the value to that location. If, as in the example, the load is moved before the store, then it uses uninitialized memory but it is syntactically valid. In Figure 9.4 the result of the field access `%2` is stored into a local variable `lvar` to be used by other blocks.

The IndependentBlocks pass also moves instructions without side-effects between basic blocks if doing so avoids memory accesses. Figure 9.4 shows that the `add` instruction has been moved into the isolated field access to avoid a variable being created for the value `%j`. `%i` might be the loop counter of a for loop. These do not need to be transformed because they are recreated by the SCoP-to-IR passes anyway.

The Polly SCoP detection passes create a statement for each basic block. The field accesses have been isolated into their own blocks, therefore every field access gets its own statement as required. Blocks that do not contain code with effects do not result in a SCoP statement. In

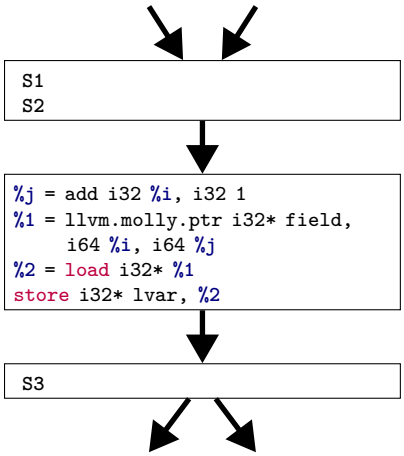


Figure 9.4: After IndependentBlocks

Figure 9.4 the two outer basic blocks may not contain any instruction and no SCoP statements are created for them if this is the case.

Any transformation done by IndependentBlocks and field access isolation can be undone by LLVM's optimization passes. Basic block are merged by control flow simplification and the SSA passes; mem2reg and SROA also remove the additional local variables.

9.3.4 Element Distribution

The task of the element distribution pass is to find a σ function that assign one (or more) nodes to field elements (Section 8.1).

In the simplest case, the relation has been defined in `#pragma molly transform` metadata. The relation must be checked for soundness, i.e. every element is assigned to at least one node and only existing nodes. Furthermore, the local part of every node must be injective. Two logically different indices must map to different physical locations. In case the layout is not defined by the user, this pass applies the standard block-distribution of Section 8.1.2.

9.3.5 Dataflow

The dataflow pass determines $<_{\text{flow}}$, the direct data flow of values computed by one SCoP statement and used by another. Any such dependency also means that a value is passed from the dependency source instance to the target instance. If those instances are executed on different nodes, an explicit transfer is necessary.

$<_{\text{anti}}$, $<_{\text{output}}$ (and $<_{\text{flow}}$) are not required by Molly because it does not by itself reorder statements. Also, Molly is only interested in data flows that are carried out through fields; only these do potentially require a data transfer. Since field accessed are always isolated into their own statements, any flow source is a field write statement whereas any flow target is a field read statement.

The prologue and epilogue statements are added in this phase. The prologue "executes" before every statement in the SCoP and virtually writes all elements of every field. The result is that dependency analysis will attribute the source of a value to the prologue when there is no other possible source. Such a flow dependency means that the value is not computed in the SCoP, but must be read from memory. The data is partitioned between multiple nodes, therefore a transfer might be necessary in this case.

The epilogue has a scattering that is after all statement in the SCoP. It virtually reads every element of every field. The intend is to find the last written values to these fields. After

dependency analysis there will be a flow dependency from the last definition of the elements to the epilogue. These are the values that the fields must contain when the SCoP's execution has finished.

ISL computes the direct data flow. For every field element use – including the prologue's – it finds its definition. From ISL's point of view this is nothing else than determining the Read-after-Write dependencies with two strange statements. Molly currently assumes that this computation is exact, which it is if all access indices are affine and branching is always conditioned on affine expressions.

9.3.6 Statement Distribution

This pass decides on which node a statement will be executed, i.e. find a placement function π_Ω. If any of the instructions in the statement is annotated with metadata from a `#pragma molly where`, the statement will be executed according to that metadata.

In case the executing node is undefined, the rules from Section 8.2.1 are applied until all statements are executed on at least one node.

9.3.7 Generate Communication

This is the main phase of the Molly framework. It finds a chunking function and inserts communication primitives into the SCoPs as described in Section 8.4.

All the dataflow dependencies are categorized into three sets (see Table 9.1): transfers between non-virtual statement instances (dependence flow), transfers from the prologue to non-virtual instances (input flow), and transfers from non-virtual instances to the epilogue (output flow).

Flow type	Flow source	Flow target	Represents
Dependency	Generator instance	Consumer instance	Flow dependency between non-virtual instances
Input	Prologue	Consumer instance	Read of field data
Output	Generator instance	Epilogue	Writeback of field data
unmodified	Prologue	Epilogue	Data not written in SCoP

Table 9.1: Data flow categories

Molly uses the schedule-dependent chunking heuristic from Section 8.4.3 (see Page 119) with the exception that chunks are computed independently for every combination of field, generator- and consumer statements. Algorithm 8.2 has different chunkings only for generator instances while it could be consumed by instances of different consumer statements. With this change flows where either the generator or the consumer (or the field) are different statements cannot be chunked into the same message. Only the domain vectors (and field indices) can be different in a single transfer event's message. This has been done for the sake of simplifying the implementation. For instance, messages that do not contain elements from different fields and indexation with different element byte lengths according to Equation (8.4) are not necessary.

Dependence flows are processed using the Events-relation from Equation (8.2). A new *combuf* (communication buffer) is created per event. A combuf is the ensemble of all the send buffers on nodes that receive data in that event and the receive buffers on receiving nodes. One property of a combuf is the (piecewise polyhedral) relation between nodes that have non-empty transfer sets. Also, four new statements are added to the SCoP that represent the communication primitives (Section 8.4.5).

send_wait is represented by a call to the intrinsic `llvm.molly.combuf.send_wait`. Its domain is the chunk space and the destination node. The source node is obviously the node it is executing on. The domain is limited to the cases where transfer sets are non-empty. Its schedule ($\theta_{\text{send\_wait}}$) is one least-significant unit before the first write to the buffer (one send buffer per chunk and destination node).

send is represented by a call to `llvm.molly.combuf.send`. Its domain is the same as for send_wait. The schedule is set one least-significant unit after the last write to the send buffer.

recv_wait is represented by a call to the intrinsic `llvm.molly.combuf.recv_wait`. Its domain consists of the chunk space and the source node, restricted to those with non-empty transfer sets. It is scheduled one unit before the first read to the send buffer (one receive buffer per chunk and source node).

recv is represented by a call to the intrinsic `llvm.molly.combuf.recv`. Analogously, it inherits most properties from the previous statement, but its schedule function is set to one least-significant unit after the last read from the receive buffer.

Input flows are the data transfers into the SCoP. There is just a single chunk "()", the zero-element tuple. That is, they are all executed at once.

All input data transfers are invoked at the beginning of the SCoP with a scattering that is below the lexicographic minimum of all other scatterings in the SCoP. In addition to the four communication primitives there is a fifth statement called readin that loads data from the node-local field partition and writes it into the send buffer. There is no source code statement that does that and therefore a new one has to be created. This statement's domain is $\mathcal{D}_{\text{readin}} \subseteq \mathcal{P} \times \mathcal{I}_F$, the destination node coordinates and the field's elements, restricted by the elements that are actually sent according to the Events-relation. A single field element can therefore be written into multiple send buffers if it is used on multiple nodes. The schedule function is set to $\theta(\text{readin}, \vec{p}_{\text{dest}}, \vec{k}) = \theta(\top)$, i.e. it is executed before all other statements in the SCoP, potentially in parallel. The placement function π_Ω is set to the placement of that field, $\pi_{\mathcal{F}}$; of course a value can only be read on the node where the value is local.

Output flows are handled similarly to input flows. Again, there is just a single chunk such that the recv_wait for all data is executed at the SCoP's termination. The statement that writes the data to the nodes' local memory is called writeback, which is also executed per field element and source node.

Primitive	Executes	Domain $\mathcal{D}_S$	Schedule θ_S	Placement
send_wait	`llvm.molly.combuf.send_wait`	$\mathcal{X} \times \mathcal{P}_C$	Before first write	$\pi_G(\vec{i}_G)$
readin	$\texttt{F}_{\text{buffer}}\texttt{[}\vec{k}\texttt{]} = \texttt{F}_{\text{local}}\texttt{[}\vec{k}\texttt{]}$	$() \times \mathcal{P}_C \times \mathcal{I}_F$	$\theta(\top)$	$\pi_F(\vec{k})$
send	`llvm.molly.combuf.send`	$\mathcal{X} \times \mathcal{P}_C$	After last write	$\pi_G(\vec{i}_G)$
recv_wait	`llvm.molly.combuf.recv_wait`	$\mathcal{X} \times \mathcal{P}_C$	Before first read	$\pi_G(\vec{i}_C)$
writeback	$\texttt{F}_{\text{local}}\texttt{[}\vec{k}\texttt{]} = \texttt{F}_{\text{buffer}}\texttt{[}\vec{k}\texttt{]}$	$() \times \mathcal{P}_G \times \mathcal{I}_F$	$\theta(\bot)$	$\pi_F(\vec{k})$
recv	`llvm.molly.combuf.recv`	$\mathcal{X} \times \mathcal{P}_G$	After last read	$\pi_G(\vec{i}_C)$

Table 9.2: Overview on communication primitives

Table 9.2 shows all the types of statements that are added by Molly. The statements readin and writeback move field elements between local storage and combufs. The others surround chunks of generator and consumer statements. In the table, $\mathcal{P}_G$ is the set of clusters where the generators execute, respectively $\mathcal{P}_C$ is where the consumer statements execute. In fact, both run on the same DMM therefore both sets are equal to (or at least subsets of) $\mathcal{P}$. The

different symbols are used to distinguish the uses of the domain. Likewise, $\vec{i}_G$ is the domain value of the generator instances and $\vec{i}_C$ that of the consumer.

The field access statements themselves are duplicated and replaced. They contain the `llvm.molly.ptr` intrinsic that must be removed. The duplicated statements replace the result of `llvm.molly.ptr` with the designated location into the combufs or local storages. The original statement is not removed but the domain of the new statement is removed from the original's domain. At the end, the original domain should be empty, therefore never executed and finally removed by dead code elimination.

The properties of the object that represents the a field access statement are: The field variable they access, the affine expressions for every index dimension and a variable on the stack created by the field isolation pass. For field read accesses this is the location the read value is stored into and in case of field write accessed the value is taken from there.

The case where $\pi_G(\vec{i}_G)$ is equal to $\pi_C(\vec{i}_C)$ is special cased. No combuf is created, but a local buffer. The generator writes its value into that buffer and the consumers read it from there. The size of the local buffer is determined by the bounding box method from Section 8.4.6.2 (Page 125). In case of input- and output flow the data is written to/read from the field's local storage in the readin and writeback statements.

9.3.8 Apply Where

Since Polly does not understand placements π_Ω of statement instances, it has to be lowered to a restriction of statement domains before being passes back to Polly. This is done in the "Apply Where" pass.

First, the code has to know the node coordinate it is executing on. This pass adds a call to the intrinsic `llvm.molly.cluster.pos` which returns the coordinate of current node to the beginning of any function that contains at least one SCoP. The values containing the coordinate is then made an structural parameter of the SCoP. Now the domains can be modified such that they depend on this coordinate. If the old domain of a statement S is $\mathcal{D}_S$, then the new domain $\mathcal{D}'_S$ that depends on a the node coordinate $\vec{p}$ is

$$\mathcal{D}'_S(\vec{p}) = \mathcal{D}_S \cap \pi_S^{-1}(\vec{p}) \ .$$

As an example, take the simple for loop

```
for (int i = 0; i < N; ++i) {
  S1;
}
```

whose statement S1's domain $\mathcal{D}_{\mathrm{S1},N} = \{i \mid 0 \leq i < N\}$ already depends on the parameter N. As placement function, set $\pi_{\mathrm{S1},N}(i) = \lfloor i/\lceil N/4 \rceil \rfloor$ such that S1 is distributed among four nodes. The new domain becomes $\mathcal{D}'_{\mathrm{S1},N,p} = \{i | 0 \leq i < N, \pi_{\mathrm{S1},N}(i) = p\}$. The equivalent code is

```
int p = llvm.molly.cluster.pos(0);
for (int i = p*((N+3)/4); i < min(N, (p+1)*((N+3)/4)); ++i) {
  S1;
}
```

9.3.9 Emit Communication Buffer Initialization and Finalization

Combufs are created when compiling a program, but MollyRT is not compiled specifically for each program. Therefore MollyRT cannot know how many combufs there are nor their properties. Molly compiles calls to initialization library functions into the optimized programs.

Two new functions are emitted: `__molly_generated_init` and `__molly_generated_release`. The former contains function calls that create new combufs at runtime, while the latter calls functions that free their memory.

For each combuf created during the communication generator phase, Molly adds a call to the MollyRT functions `__molly_combuf_send_alloc` and `__molly_combuf_recv_alloc` into `__molly_generated_init`, and corresponding free functions into `__molly_generated_release`. They accept arguments about how many communication destinations respectively sources there are with non-empty transfer sets.

For each of the destinations, a call to `__molly_combuf_send_dst_init` is added. The size of the buffer to that destination is passed as an argument so MollyRT can allocate a buffer of that size. The transfer set size might be different for different chunks, therefore the maximum over all chunks is used. Actually, this can fail if the number of chunks is unbounded and the transfer set grows for every chunk. The only possible solution for this case is to allocate the buffer again for every chunk, something that Molly cannot do in its current implementation. Compilation therefore will fail in this particular case.

Correspondingly, calls to `__molly_combuf_recv_src_init` are added, passing the receive-buffer size. The allocated size of memory to store the receive- and send combuf objects is again determined using the bounding box method (Section 8.4.6.2).

Molly does not create a call instruction for every possible destination or source. Instead, it creates a new SCoP with statements that call these functions. Their domains are the set of destination/source nodes with non-empty transfer sets ($\mathcal{D}_S \subseteq \mathcal{P}$). Polly's code generator then generates IR code out of these. Polly previously did not support creating SCoPs from scratch, i.e. without references to LLVM IR code. This feature had to be added.

9.3.10 Replace Remaining Intrinsics

The communication generation pass removed all Molly-specific intrinsics from any SCoPs, but those not in a SCoP remain in the code. These have to be removed eventually which happens in this pass. Otherwise LLVM's machine code generators fail because it they do not know how to translate such intrinsics to machine code.

Table 9.3 shows what the intrinsics are replaced with. Most of them are replaced by a MollyRT library functions (those that begin with two double underscores) that will handle these cases.

Intrinsic	Replaced by
`%ptr = llvm.molly.ptr` `%val = load %ptr`	`__molly_value_load`
`%ptr = llvm.molly.ptr` `store %val, %ptr`	`__molly_value_store`
`llvm.molly.rankof`	Generated code
`llvm.molly.islocal`	Generated code
`llvm.molly.indexof`	Generated code
`llvm.molly.cluster.pos`	`__molly_cluster_pos`
`llvm.molly.field.init`	`__molly_local_free`
`llvm.molly.field.free`	`__molly_local_init`
`llvm.molly.combuf.send_wait`	`__molly_combuf_send_wait`
`llvm.molly.combuf.send`	`__molly_combuf_send`
`llvm.molly.combuf.recv_wait`	`__molly_combuf_recv_wait`
`llvm.molly.combuf.recv`	`__molly_combuf_recv`
`llvm.molly.combuf.send_ptr`	`__molly_combuf_send_ptr`
`llvm.molly.combuf.recv_ptr`	`__molly_combuf_recv_ptr`

Table 9.3: Selected intrinsics and their replacements

The handling of `llvm.molly.ptr` intrinsics not part of a SCoP depends on whether they are a read or write access to a field. Eventually, they are converted to calls to either `__molly_value_load` or `__molly_value_store`. The MollyRT implementation then has to execute the requested information. It may either just access the element in the node's local memory or invoke a communication to the node that actually stores the value. This has to be done per access and the library therefore is unable to coalesce communication. This overhead is immense so programmers better access `molly::array` fields inside SCoPs. However, handling such cases is necessary for the optimization's correctness.

The intrinsics `llvm.molly.rankof`, `llvm.molly.indexof` and `llvm.molly.islocal` are required by the `molly.h` header file. They allow queries for the physical memory location. The functionality is also used during the communication generation phase for SCoPs, but these intrinsics make access outside of SCoPs possible. In particular, MollyRT will call functions in `molly.h` containing these intrinsics using virtual dispatch to implement `__molly_value_load` and `__molly_value_store`.

`llvm.molly.rankof` returns the rank of a node that stores a particular element. This node has to be contacted to access that element. `llvm.molly.islocal` is a faster implementation of "`llvm.molly.rankof == llvm.molly.cluster.pos`". If it is local, `llvm.molly.indexof` can be used to get the offset into the local storage for that element. All three depend on decisions made in element distribution pass for each field and therefore cannot be implemented in a library. Multiple storage locations for the same logical elements are currently not supported in the mechanism.

The function `llvm.molly.cluster.pos` returns the coordinates of the node it is executed on. Node coordinates are handled by MollyRT, therefore this is also a call to `__molly_cluster_pos`. The intrinsics `llvm.molly.combuf.send_wait`, `llvm.molly.combuf.send`, `llvm.molly.combuf.recv_wait` and `llvm.molly.combuf.recv` have been inserted by the communication generation pass. They are also handled by Molly's runtime library.

9.3.11 Wrap Main

Finally, the `main` function is renamed to `__molly_orig_main` and replaced by a Molly-specific main-function. The new main calls `__molly_main` from MollyRT. This is necessary because `MPI_Init` requires the argument passed to the program and the `argc` and `argv` arguments of the main function is the only platform-independent source. An MPI implementation of mpirun may choose to run MPI program instances with command line arguments that tell them how to connect to each other[1]. Without this information such MPI programs assume they run on a single node only. The wrapping of main takes away from the programmer the responsibility to call designated initialization and finalization functions. Leaving it to the compiler removes a possible source of error.

In addition, `__molly_main` has additional parameters about the expected cluster shape (passed to the compiler using the `--molly-shape=` switch). The runtime should error-exit if a smaller/incompatible physical shape was found.

`__molly_main` itself initializes the communication API, e.g. `MPI_Init`, calls `__molly_generated_init` and then `__molly_orig_main`, the program itself. When the original `main` returns, `__molly_generated_release` is called, the communication API is released (e.g. `MPI_Finalize`) and then `__molly_main` returns to the `main` mock which in turn returns to the C-runtime that will eventually terminate the application.

[1] MPICH and OpenMPI both pass this information using environment variables

9.4 MollyRT

MollyRT is the runtime library for applications compiled with Molly. It provides implementations of functions that are called during the code generation as mentioned in the previous section. It currently uses MPI to implement its functionality, but other backends such as Blue Gene/Q MUSPI, PAMI, shared memory, etc. are possible as well.

The library is organized into "communicators". A communicator has basically the same meaning as in MPI, i.e. a set of nodes that communicate with each other. There might be additional communicators that use other means of communication than MPI, like through shared memory. Molly-generated programs use just the main communicator, which – in principle (because currently there is just one implementation) – can be configured using different configurations and backends.

The entry point is `__molly_main` which must be called by the host application before using any other library call (see previous section) and initializes the default communicator. It is passed information about program start-up (command line arguments, environment variables) and the cluster configuration the program was compiled for.

MollyRT is interwoven with the `molly.h` header file (Section 9.2.1). `molly.h` contains Clang builtins, translated to LLVM intrinsics that are processed by Molly (see Figure 9.5). The intrinsics are finally translated to MollyRT functions as in Section 9.3.10. Molly is compiled independently from the optimized program and therefore does not know about the fields' shapes and distributions. Only functions implemented in `molly.h`, with **inline** and **static** keywords, are compiled with information about the field's distribution and layout.

Therefore `molly::array`'s base type `LocalStore` has virtual functions that can be called by MollyRT. For example, see the interaction of the field initialization between the layers in Figure 9.5. The `molly::array` type has a constructor that uses `__builtin_molly_field_init`, a Clang builtin. Note that the constructor is declared with a `[[molly::inline]]` attribute that will cause the MollyInline pass from Figure 9.2 to inline the constructor for each declared field, such that `this` becomes a direct reference to the declared field. Builtins are translated to intrinsics in LLVM IR. Molly will find these and deduce that the first argument is a field with the logical shape and dimensions passed as arguments (The `L...` variadic template expansion). The intrinsic is then translated to a call to `__molly_local_init` (Section 9.3.10) passing the number of values local to this node. This MollyRT function call dispatches to an init-function in `molly.h` that finally allocates the memory.

The virtual function cannot be called directly because virtual method calling convention and name mangling is done in Clang, not LLVM, and specific to the platform it compiles to: Itanium (GCC) and Microsoft (Visual C++). The function names cannot be reliably unmangled. Moreover, the function may not appear at all if all calls to it have been inlined or never called at all.

The allocation cannot be implemented in `__molly_local_init` because the memory allocation depends on the type template argument. The byte size could be passed as an argument, but the type (`T` in the figure) might have a constructor.

9.4.1 MPI Communicator Implementation

The communicator calls the usual `MPI_Init_thread` in its implementation, then it calls `MPI_Cart_create` to create a Cartesian grid using all the node in the shape the program was compiled for. The number of nodes must be the same as the program was compiled for, otherwise it aborts. It queries the cluster configuration and the coordinate on which the current program is running on.

For each combuf created by Molly two objects are created: A send combuf and a receive combuf. Each may contain multiple destinations/sources. Each destination/source in the combuf is set up with MPI persistent communication using `MPI_Send_init` (respectively

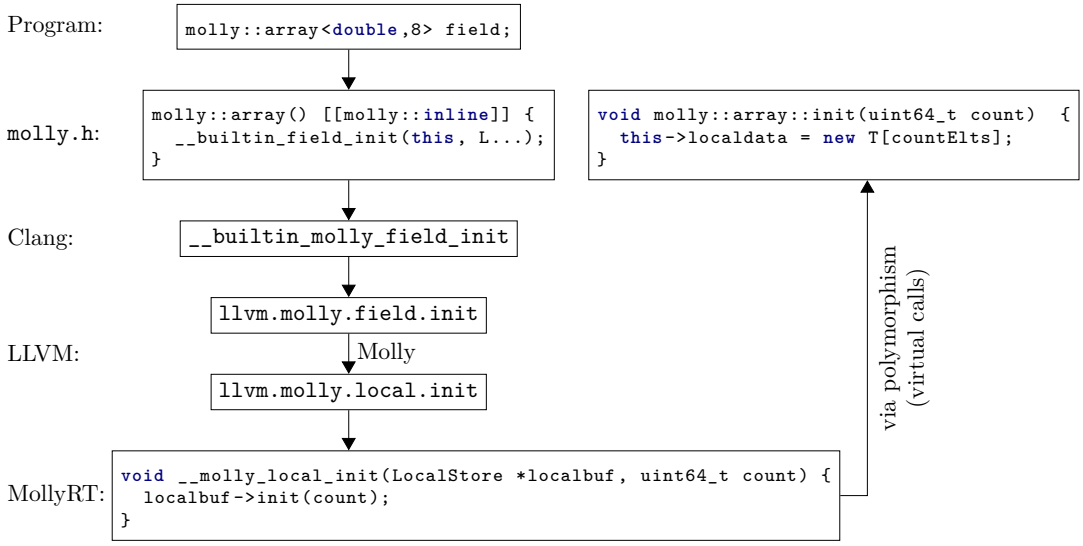


Figure 9.5: Simplified transformation/call sequence for local storage initialization

`MPI_Recv_init`) and allocates the required buffer sizes. For receive buffers, `MPI_Start` is called immediately such that the node is ready to receive data even before entering any SCoP.

With persistent communication the primitive `__molly_combuf_send_wait` becomes a call to `MPI_Wait` unless it was the first send_wait. In this case, there is nothing to be done. The data is sent in `__molly_combuf_send` using `MPI_Start`.

Analogously, `__molly_combuf_recv_wait` calls `MPI_Wait` to ensure any data has been received. The buffer is put back in ready state in `__molly_combuf_recv` using `MPI_Start` just as if the buffer has been freshly initialized.

10 Results

In this chapter the results of three programs compiled with Molly are presented. The first is Jacobi, the running example used in Sections 7.3 and 8.5. The second is the Dslash stencil the thesis' first part was all about. The third is Conway's Game of Life.

For benchmarking, the same remarks as in Chapter 5 apply. The compiled program also runs on the Blue Gene/Q cluster in Jülich. Hal Finkel, contributor to the LLVM project at Argonne National Laboratory, provided the modification necessary to make LLVM compile to the Blue Gene/Q target [59]. It is highly unofficial by both, IBM and the LLVM project, and does not have had a non-development release yet.

All three programs are kinds of stencils, so again we can use million lattice updated per second (mlup/s) as measurement unit.

Clang's effort to support OpenMP are still in development using a runtime library donated by Intel. Polly can generate code that uses gcc's runtime *libgomp*. The thesis' main topic is distributed-memory parallelism, so SMP has not been implemented. In order to be able to use all the cores of a node, multiple MPI processes ("ranks") can be executed on one node. We use this mechanism instead of threads. The abbreviation RpN stands for "Ranks per Node", the number of processes running on the node.

10.1 Jacobi 2D

This example program has only few floating-point operations, in contrast to Lattice QCD. The source code from Listing 7.3 is compiled using Molly. The preprocessor definitions of `LX` and `LY` are modified accordingly. `ITERATIONS` is set to 3. The program has to be compiled separately for every configuration, although double and single precision versions can be put into the same executable.

10.1.1 Ranks per Node

Table 10.1 and Figure 10.1 show the results for some configurations on 32 nodes. The number of ranks on a node varies between one and 64. The number of OpenMP threads can be any integer in this range, but ranks per node only allows powers of two. 48 threads were tested in Sections 5.1.1 and 5.2.1, but there is no 48 ranks per node setting for Blue Gene/Q nodes. No special attention has been drawn to how the ranks are distributed between the nodes.

We can observe that the more ranks run on the node the better the performance due to improved core utilization. Like in the manually optimized programs, (weak) scaling is nearly perfect up to 32 ranks, but still improves significantly with 64 ranks when communication is disabled ("nocom" column). Communication is switched off by not issuing the calls to MPI in Molly's runtime, there is no difference for the compiler itself. Interestingly, there is no difference in performance between single and double precision, indicating that memory

MPI		nocom			
Time	Stencils/node	Time	Stencils/node	RpN	Precision
34.49 ms	4.35 mlup/s	2.69 ms	55.93 mlup/s	64	64 bit
27.22 ms	7.69 mlup/s	2.76 ms	54.58 mlup/s	64	32 bit
11.80 ms	6.38 mlup/s	1.83 ms	41.14 mlup/s	32	64 bit
8.40 ms	8.95 mlup/s	1.84 ms	40.94 mlup/s	32	32 bit
6.77 ms	5.56 mlup/s	1.87 ms	20.12 mlup/s	16	64 bit
4.93 ms	7.68 mlup/s	1.73 ms	21.72 mlup/s	16	32 bit
4.49 ms	4.19 mlup/s	1.62 ms	11.64 mlup/s	8	64 bit
3.24 ms	5.81 mlup/s	1.52 ms	12.37 mlup/s	8	32 bit
3.53 ms	2.67 mlup/s	1.62 ms	5.80 mlup/s	4	64 bit
2.63 ms	3.58 mlup/s	1.64 ms	5.73 mlup/s	4	32 bit
3.17 ms	1.49 mlup/s	1.79 ms	2.62 mlup/s	2	64 bit
2.30 ms	2.05 mlup/s	1.72 ms	2.73 mlup/s	2	32 bit
2.28 ms	1.03 mlup/s	1.87 ms	1.26 mlup/s	1	64 bit
2.08 ms	1.13 mlup/s	1.72 ms	1.36 mlup/s	1	32 bit

Table 10.1: Execution times, 28x28 volume per rank, 32 nodes

bandwidth is no issue. The volumes processed in these test are also relatively low indicating that the issue is a large overhead and/or inefficient code.

When calls to the MPI functions are not just skipped we see that communication was dominating the total runtime, especially with more ranks. We already know this phenomenon from Section 5.1.3: MPI executes data transfers within the same node using a software-based `memcpy` while data to remote destination are moved by the hardware. Hence, the more communication within the node, the higher the communication overhead. It also becomes visible by the difference between 4 and 8 byte elements. The latter requires twice large buffers, i.e. more data to transfer by `memcpy`.

10.1.2 Matrix Size

Here wo modify the size of the lattice (for Jacobi: a matrix) without changing the other parameters. The execution speeds are shown in Table 10.2 and Figure 10.2.

Generally it can be said that the performance improves with larger matrices because of the reduced relative overhead. The time taken by communication is still dominating in all cases shown here. The matrices all fit into the L2 cache so there is no slowdown visible for the largest sizes.

10.1.3 Weak Scaling

Here we observe the performance behavior when adding more nodes to the system, but the work to do for every rank remains the same. The data is shown in Table 10.3 and Figure 10.3.

The performance without communication is approximately constant, as expected since other nodes should have no influence on the work done of unrelated nodes.

The situation is worse with enabled communication. The execution speed decreases dramatically with more nodes in the system. We assume that the reason for this behavior is a sub-optimal mapping of the 2-dimensional matrix to the rank space (5 node dimensions and 1 representing the processes within one node) with multiple hops between ranks that communicate with each other. The more hops there are the slower the communication. The mapping looks fine for 4 nodes or less.

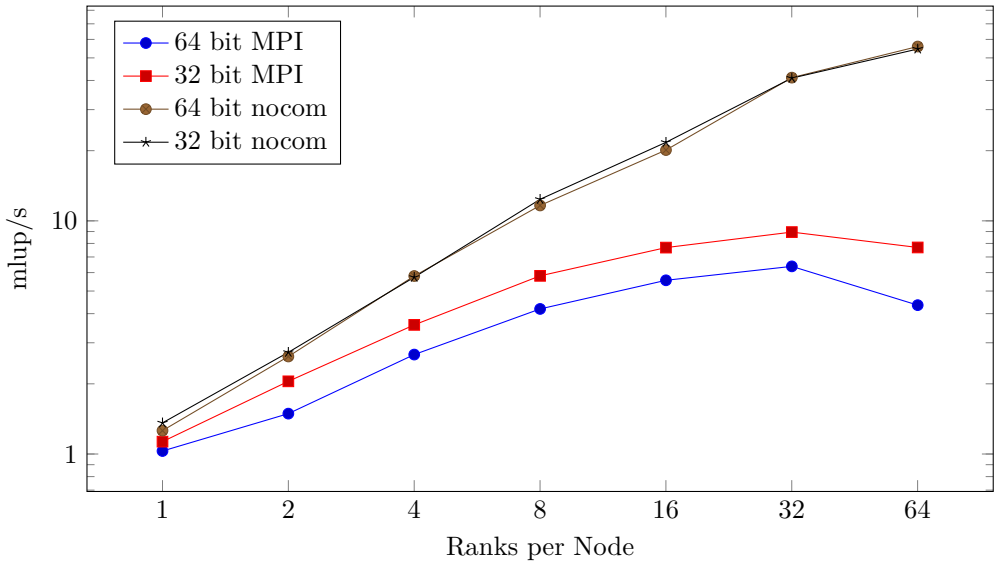


Figure 10.1: Visualization of Table 10.1

MPI		nocom		
Time	Stencils/node	Time	Stencils/node	Volume per rank
3.32 ms	0.18 mlup/s	0.07 ms	20.60 mflop/s	4x4
8.36 ms	0.41 mlup/s	0.11 ms	30.06 mflop/s	6x6
9.65 ms	0.64 mlup/s	0.20 ms	30.29 mflop/s	8x8
9.70 ms	0.99 mlup/s	0.26 ms	37.49 mflop/s	10x10
9.82 ms	1.41 mlup/s	0.36 ms	38.87 mflop/s	12x12
10.07 ms	1.87 mlup/s	0.49 ms	38.71 mflop/s	14x14
10.60 ms	2.32 mlup/s	0.65 ms	37.53 mflop/s	16x16
10.92 ms	2.85 mlup/s	0.79 ms	39.14 mflop/s	18x18
11.01 ms	3.49 mlup/s	1.01 ms	38.00 mflop/s	20x20
11.18 ms	4.16 mlup/s	1.15 ms	20.28 mflop/s	22x22
11.39 ms	4.85 mlup/s	1.30 ms	42.42 mflop/s	24x24
11.61 ms	5.59 mlup/s	1.65 ms	39.45 mflop/s	26x26
11.83 ms	6.36 mlup/s	1.83 ms	41.14 mflop/s	28x28
11.85 ms	7.29 mlup/s	1.93 ms	44.87 mflop/s	30x30
13.08 ms	7.52 mlup/s	2.67 ms	36.87 mflop/s	32x32
12.96 ms	8.56 mlup/s	2.94 ms	37.78 mflop/s	34x34
13.39 ms	9.29 mlup/s	3.30 ms	37.68 mflop/s	36x36
13.01 ms	10.58 mlup/s	2.97 ms	46.61 mflop/s	38x38
13.36 ms	11.50 mlup/s	3.30 ms	46.56 mflop/s	40x40
13.65 ms	12.41 mlup/s	3.62 ms	46.80 mflop/s	42x42
14.10 ms	13.19 mlup/s	3.99 ms	46.54 mflop/s	44x44
14.55 ms	13.96 mlup/s	4.29 ms	47.32 mflop/s	46x46
15.89 ms	13.92 mlup/s	5.36 ms	41.28 mflop/s	48x48

Table 10.2: Varying subvolumes, 32 ranks per node, 32 nodes, 64 bit precision

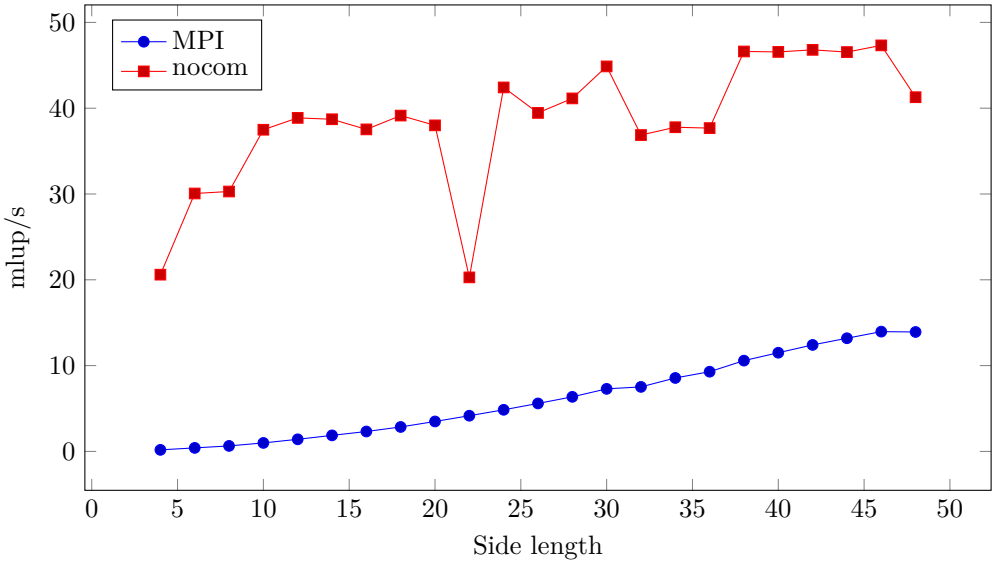


Figure 10.2: Visualization of Table 10.2

MPI		nocom			
Time	Stencils/node	Time	Stencils/node	Shape	Nodes
4.18 ms	17.99 mlup/s	2.10 ms	35.76 mlup/s	8x4	1
4.43 ms	16.97 mlup/s	2.04 ms	36.85 mlup/s	8x8	2
4.31 ms	17.47 mlup/s	1.84 ms	40.82 mlup/s	16x8	4
9.09 ms	8.28 mlup/s	2.15 ms	35.05 mlup/s	32x16	16
11.80 ms	6.38 mlup/s	1.83 ms	41.14 mlup/s	32x32	32

Table 10.3: Weak scaling, 28x28 volume per rank, 32 ranks per node, 64 bit precision

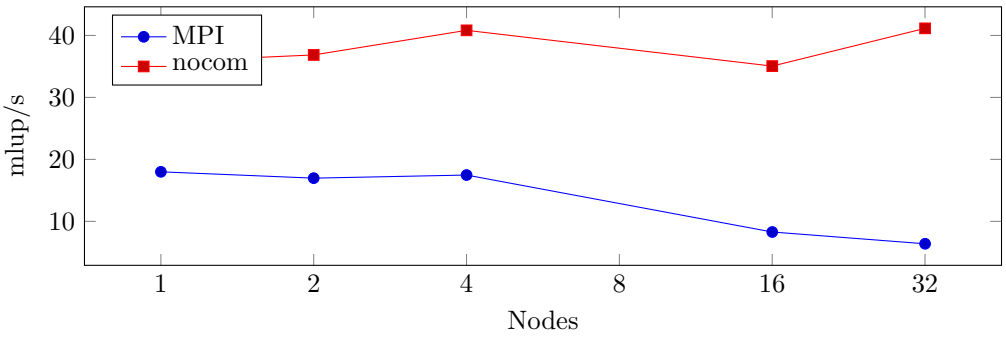


Figure 10.3: Visualization of Table 10.3

10.2 Lattice QCD

We finally come to the results of the automatically distributed Lattice QCD kernel which are comparable to the manually-optimized program. The source code of the program with 1320 flop per stencil is shown below. It has been instantiated for 32 nodes in a 4x2x2x2-shape and a volume of 48x24x24x24 (i.e. a volume of 12x12x12x12 per node). The instantiation happens by defining symbols using the compiler's `-D` command line arguments and the preprocessor.

In total, there are three fields (distributed arrays) in this program. Data is read from the source and gauge fields and written to the sink spinor field. The spinor fields are torus-shaped, meaning that the rightmost point in one dimension is a neighbor of the leftmost point with otherwise same coordinates. The data type `molly::array` does not enforce this rule by itself (although there could be a data type that does this), but must be enforced manually using the `molly::mod` function. The internal remainder operator `%` does not work for this purpose because by definition in *C99* its result is negative when the left operand is negative. This happens with the left neighbor of coordinate 0. Molly recognizes the `molly::mod` operation and will convert either into a call to the runtime library that implements this function or to the corresponding modulus operation supported by ISL, depending on context.

The gauge could also be handled this way, but there is a practical obstacle. The bounding box method for reserving memory (the bounding box method from Section 8.4.6.2) is unaware of the modulus operation. For instance, consider the rank that processes the element `sink[0][0][0][23]`. To update the element at that coordinate, it needs the elements `gauge[0][0][0][23]` and the right neighbor `gauge[0][0][0][24]`. If the latter actually maps to `gauge[0][0][0][0]`, this means that the bounding box spans over the complete gauge field (from element 0 to 23), although only two of its elements are actually used. If the lattice is large and normally spans many nodes, a single memory allocation for all the gauge field variables may exhaust a single node's memory.

The workaround for this case is to append another element to the right of the rightmost elements and not using the modulo operator. The 0th element has to be mirrored manually to the 24th; they logically they must contain the same value.

The `#pragma` annotation of the gauge field is a near-block decomposition but with an one element overlap which is used by two (logical) nodes, i.e. some SU(3) matrices are stored on two nodes. Because of this its definition is so long as it cannot be expressed as a function.

The spinor fields are block-decomposed between the logical nodes, but in the node's local memory the elements are permuted. The t-dimension is split up into two dimensions. The information whether the t coordinate is even or odd is appended as the innermost dimension. This should simplify the compiler's task to vectorize values that are neighbors in t-direction, instead of those in z direction. The rationale is that in practical uses the t-dimension is twice as large as the other dimensions so the vectorized (physical in the terms of the first part) geometry is rectangular. A vectorized element also does not interfere with itself because the innermost loop iterates over the z-coordinate. Alternatively, this mechanism could be used to separate odd from even coordinates as it has been done in the manual optimization Section 4.1. It also serves the purpose of demonstration that Molly can also modify the data layout within a node.

The types `su3matrix_t` and `spinor_t` have been defined in the `lqcd.h` header file including the expected operator overloads. The header also defines the functions `project_`DIR and `expand_`DIR to convert between full- and halfspinors. It is best comparable with the fullspinor layout from Section 4.4.1 because halfspinors are not stored persistently.

10.2.1 Featured Experiments

Finally, we present the timings of the code above in Table 10.4 and Figure 10.4.

Compared to the manually optimized version, this is about five times slower than the

Time	Stencil/node	Flop/Node	Peak %	Variant
422.64 ms	3.14 mlup/s	5049 mflops/s	2.5 %	1608 flop/stencil
418.58 ms	3.17 mlup/s	5098 mflops/s	2.5 %	1320 flop/stencil
399.02 ms	3.33 mlup/s	5348 mflops/s	2.6 %	1608 flop/stencil, nocom
395.71 ms	3.35 mlup/s	5393 mflops/s	2.6 %	1320 flop/stencil, nocom

Table 10.4: Execution time of a 96x96x96x48 lattice on 32 nodes, 64 threads each, MPI, 64 bit precision

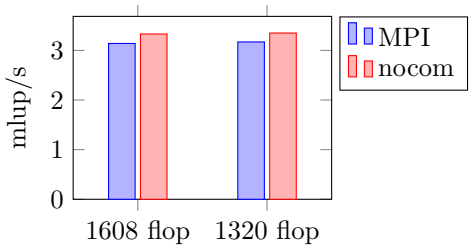


Figure 10.4: Visualization of Table 10.4

fullspinor implementation (6 for nocom) and about 12 times slower than the halfspinor version (a whopping 18 times slower when comparing the nocom versions).

When the disassembly of the executable are compared it becomes clear why. The manually optimized versions contain a large portion of QPX instructions in the Hopping Matrix kernel. In contrast the LLVM output was not able to vectorize any instruction. Also, the ration of floating point instructions to general purpose instructions (integer arithmetic, etc.) is much lower.

At least the numbers show that the total execution is dominated by the computation of the stencil, in contrast to the Jacobi examples with much smaller lattice sizes where the communication dominates the total runtime.

The difference between the 1320 flop version and the 1608 flops version is almost unnoticeable but the tendency that the 1608 flop version is expected.

10.2.2 Ranks Per Node

The effect of changing the number of processes executing on a single node is shown in Table 10.5. The characteristic is the same as with the Jacobi example. There is an almost perfect scaling up to 16 ranks executing on a single node. 32 and 64 ranks still show a performance improvement, but not a doubling of performance. This is a difference to the Jacobi behavior where the 32 ranks case still almost doubled the performance.

10.2.3 Lattice Size

In this step we modify the volume that is local to a rank without considering global effect, i.e. communication is switched off.

We can deduce from Table 10.6 that the performance is not dependent on the amount of work to be done per rank, we even do not see the impact of the L2 cache not being large enough as we did in Sections 5.1.2 and 5.2.2. The only exception the case there every rank only does one stencil, which is slower due to the impact of the relatively large overhead.

The explanation of this behavior is the code's inefficiency: The execution is just not fast enough such that memory bandwidth could be a limiting factor. The increased latency his

Time	Stencils/node	RpN
195.58 ms	0.11 mlup/s	1
187.43 ms	0.22 mlup/s	2
187.96 ms	0.44 mlup/s	4
189.49 ms	0.88 mlup/s	8
193.92 ms	1.71 mlup/s	16
247.14 ms	2.68 mlup/s	32
396.51 ms	3.35 mlup/s	64

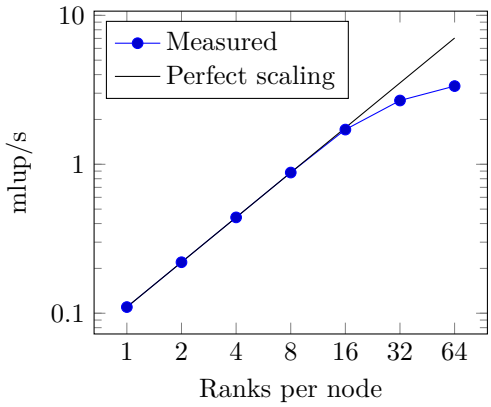


Table 10.5: Varying ranks per node, 128 nodes, 12x12x12x12 subvolume, 1320 flops/stencil, nocom

Time	Stencils/node	Volume per rank
0.03 ms	2.14 mlup/s	1x1x1x1
5.11 ms	3.21 mlup/s	4x4x4x4
24.90 ms	3.33 mlup/s	6x6x6x6
80.08 ms	3.27 mlup/s	8x8x8x8
190.73 ms	3.36 mlup/s	10x10x10x10
396.62 ms	3.35 mlup/s	12x12x12x12
732.73 ms	3.36 mlup/s	14x14x14x14
1244.56 ms	3.37 mlup/s	16x16x16x16
1979.28 ms	3.39 mlup/s	18x18x18x18

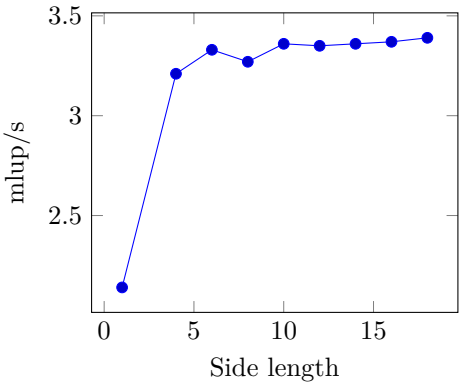


Table 10.6: Varying subvolumes, 1320 flops/stencil, 128 nodes, 64 ranks per node, nocom

well hidden by the threading.

10.2.4 Weak Scaling

Of course we also study the weak scaling behavior of the compiler-distributed Lattice QCD program. The performance data is shown in Table 10.7.

From the data we cannot observe any impact of the cluster size on the performance of the individual nodes. This is as it should be and what we call perfect (weak) scaling.

Time	Stencils/node	Shape	Nodes
399.70 ms	3.32 mlup/s	4x4x2x2	1
409.32 ms	3.24 mlup/s	4x4x4x2	2
413.11 ms	3.21 mlup/s	4x4x4x4	4
417.02 ms	3.18 mlup/s	8x4x4x4	8
418.73 ms	3.17 mlup/s	8x8x4x4	16
425.92 ms	3.12 mlup/s	8x8x8x4	32
424.74 ms	3.12 mlup/s	8x8x8x8	64
423.46 ms	3.13 mlup/s	16x8x8x8	128
418.33 ms	3.17 mlup/s	16x16x8x8	256

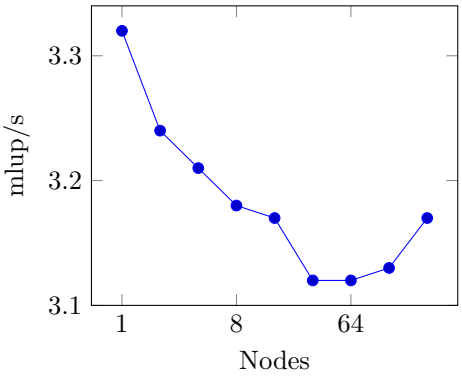


Table 10.7: Weak scaling, 12x12x12x12 volume per rank, 64 ranks per node, 1320 flops/stencil

```
#include <molly.h>
#include <lqcd.h>

#pragma molly transform("{ [t,x,y,z] -> [node[floor(t/12),floor(x/12),floor(y/12),
    floor(z/12)] -> local[floor(t/2),x,y,z,t%2]] }")
molly::array<spinor_t, 48, 24, 24, 24> source, sink;

#pragma molly transform("{ [t,x,y,z,d] -> [node[pt,px,py,pz] -> local[t,x,y,z,d]] :
    0<=pt<4 and 0<=px<2 and 0<=py<2 and 0<=pz<2 and 12pt<=t<=12*(pt+1) and 12px<=x
    <=12*(px+1) and 12py<=y<=12*(py+1) and 12pz<=z<12*(pz+1) }")
molly::array<su3matrix_t, 48 + 1, 24 + 1, 24 + 1, 24 + 1, 4> gauge;

void HoppingMatrix() {
  for (int t = 0; t < source.length(0); t += 1)
    for (int x = 0; x < source.length(1); x += 1)
      for (int y = 0; y < source.length(2); y += 1)
        for (int z = 0; z < source.length(3); z += 1) {
          auto halfspinor = project_TUP(source[molly::mod(t + 1, LT)][x][y][z]);
          halfspinor = gauge[t + 1][x][y][z][DIM_T] * halfspinor;
          auto spinor_t result = expand_TUP(halfspinor);

          halfspinor = project_TDN(source[molly::mod(t - 1, LT)][x][y][z]);
          halfspinor = gauge[t][x][y][z][DIM_T] * halfspinor;
          result += expand_TDN(halfspinor);

          halfspinor = project_XUP(source[t][molly::mod(x + 1, LX)][y][z]);
          halfspinor = gauge[t][x + 1][y][z][DIM_X] * halfspinor;
          result += expand_XUP(halfspinor);

          halfspinor = project_XDN(source[t][molly::mod(x - 1, LX)][y][z]);
          halfspinor = gauge[t][x][y][z][DIM_X] * halfspinor;
          result += expand_XDN(halfspinor);

          halfspinor = project_YUP(source[t][x][molly::mod(y + 1, LY)][z]);
          halfspinor = gauge[t][x][y + 1][z][DIM_Y] * halfspinor;
          result += expand_YUP(halfspinor);

          halfspinor = project_YDN(source[t][x][molly::mod(y - 1, LY)][z]);
          halfspinor = gauge[t][x][y][z][DIM_Y] * halfspinor;
          result += expand_YDN(halfspinor);

          halfspinor = project_ZUP(source[t][x][y][molly::mod(z + 1, LZ)]);
          halfspinor = gauge[t][x][y][z + 1][DIM_Z] * halfspinor;
          result += expand_ZUP(halfspinor);

          halfspinor = project_ZDN(source[t][x][y][molly::mod(z - 1, LZ)]);
          halfspinor = gauge[t][x][y][z][DIM_Z] * halfspinor;
          result += expand_ZDN(halfspinor);

          sink[t][x][y][z] = result;
        }
}
```

Listing 10.1: Lattice QCD program for Molly

10.3 Game of Life

Here we present of a simplified version of Conway's Game of Life performs when compiled with Molly. The code is shown in Listing 10.2 and has just 5 stencils point instead of 9 in the original Game of Life. Structurally, it similar to the Jacobi example, but has no branching and no floating-point math. Because the fields are not torus-shaped, the stencils at the borders are not computed.

```
#include <molly.h>

molly::array<bool, LX, LY> source,sink;

[[molly::pure]] bool hasLife(bool hadLife, int neighbors) {
  if (hadLife)
    return 2 <= neighbors && neighbors <= 3;
  else
    return neighbors == 3;
}

void GameOfLife() {
  for (int i = 0; i < 100; i+=1) {
    for (int x = 1, w = source.length(0); x < w-1; x+=1)
      for (int y = 1, h = source.length(1); y < h-1; y+=1) {
        auto neighbors = source[x-1][y]
        neighbors += source[x][y+1];
        neighbors += source[x+1][y];
        neighbors += source[x][y-1];

        auto hadLife = front[x][y];
        auto living = hasLife(hadLife, neighbors);

        sink[x][y] = living;
      }
    for (int x = 1, w = sink.length(0); x < w-1; x+=1)
      for (int y = 1, h = sink.length(1); y < h-1; y+=1) {
        source[x][y] = sink[x][y];
      }
  }
}
```

Listing 10.2: Conway's Game of Life

The experiments in Table 10.8 have a wider variety than what has been tested in the previous benchmarks, but the configuration are not chosen systematically. The time unit in this table is not milliseconds, but seconds, and the number of stencils per second is specified per ranks, not per node. Depending on the "ranks-per-node" setting, up to 64 ranks are executed on the same node. Accordingly, the performance per rank declines with 32 or even 64 ranks on a node.

The experiments show the approximately perfect weak scaling of the program and on this hardware. If the tile size per node is 128^2, the 100 iterations take almost exactly one second, whatever the total number of nodes in the system is. The nodes on the mesh surface have less work to do, therefore the execution with just 4 nodes takes just 0.9 seconds. With 64 ranks per node the execution time increases to just 1.7 seconds, but gets even a little faster with one rank per core due to in-memory transfers on the node.

The program also scales as expected if the size of the tile per rank increases. For a 4-times more lattice points, the program takes 4 times longer to execute. The number of stencil executions per seconds remains approximately the same.

Of course, this serves more as a demonstration of the capabilities of Molly than to run

Time	Stencils/rank	Rank shape	RpN	Global volume	Volume per rank
0.9 s	1.79 mlup/s	2x2	1	256^2	128^2
1.0 s	1.63 mlup/s	4x4	1	512^2	128^2
63.3 s	1.65 mlup/s	4x4	1	4096^2	1024^2
0.3 s	1.35 mlup/s	8x8	1	512^2	64^2
1.0 s	1.63 mlup/s	8x8	1	1024^2	128^2
4.0 s	1.64 mlup/s	8x8	1	2048^2	256^2
15.8 s	1.66 mlup/s	8x8	1	4096^2	512^2
63.0 s	1.66 mlup/s	8x8	1	8192^2	1024^2
0.3 s	1.35 mlup/s	8x8	2	512^2	64^2
62.9 s	1.67 mlup/s	8x8	2	8192^2	1024^2
1.0 s	1.63 mlup/s	8x8	2	1024^2	128^2
4.0 s	1.64 mlup/s	8x8	2	2048^2	256^2
1.0 s	1.64 mlup/s	16x16	1	2048^2	128^2
1.0 s	1.64 mlup/s	16x16	8	2048^2	128^2
1.0 s	1.64 mlup/s	16x16	1	2048^2	128^2
1.0 s	1.64 mlup/s	32x32	1	4096^2	128^2
1.0 s	1.64 mlup/s	32x32	16	4096^2	128^2
1.2 s	1.36 mlup/s	32x32	32	4096^2	128^2
1.7 s	0.96 mlup/s	32x32	64	4096^2	128^2
1.7 s	0.96 mlup/s	64x64	64	8192^2	128^2
1.7 s	0.96 mlup/s	128x128	64	16384^2	128^2

Table 10.8: Runtimes of the reduced Conway's Game of Life on Blue Gene/Q

the Game of Life. Any optimized Game of Life program reaches this performance without thousands of nodes with techniques that are not in Molly's scope. In addition, the code generated by Molly is not as optimal as it could. For instance, it generates a lot of conversions between 32- and 64-bit integers, as one can see in the disassembly.

11 Discussion

While the previous chapters presented the implementation of Molly and experimental results, this chapter is about the evaluation of the work done. We begin with a summary of the previous chapters of this part of this thesis and jump over to conclusions we can draw from the experience of implementing Molly and the benchmarks. The future work section suggests how to overcome known deficiencies, and how to make Molly more useful. Finally, we briefly go over what other researchers and have done in the field of compiler-based optimization of distributed memory parallelism.

11.1 Summary

Molly is a compiler extension intended to take away the complexity of writing scientific programs for distributed memory machines by generating the necessary communication code. Such communication code often has the same patterns but is difficult to write because a lot of index computations, allocations, node synchronizations etc. are involved. The goal is not to take away control from the programmer who can still decide where to place and execute computations and data. Quite the contrary, the developer can try out different configurations to see which runs faster on a particular machine.

Molly was developed with stencil-like code in mind, but is not restricted to those. Non-stencil codes are parallelized as well, but less effectively. The minimal requirement is that the code to parallelize must be region of arbitrarily nested for-loops (*Static Control Part* (SCoP)). In contrast to many previous attempts on automatic parallelization on *Distributed Memory Machines* (DMMs), Molly does it introduce a new programming language that forces the developer to only use high-level constructs. The programmer does not need to write explicitly for a specific architecture, but still has full control if needed for optimization. The integration into the general purpose compiler *Clang* [6] hopefully simplifies its use. It also has the advantage of taking away the necessity of reinventing the wheel for every language feature or optimization.

This thesis also establishes an extension to the polyhedral model that describes the distribution of data and work in a cluster computer, as well as the data order in local memory. Default policies are described for both, the distribution of data and executions of statements around the nodes in the cluster. The antichain chunking algorithm combines transfers of single values to messages containing multiple values with minimal number of total messages, but it is only applicable to non-loop codes or SCoPs that can be unrolled completely. Its alternative is a chunking heuristic that tries to follow the source code's structure and therefore the intention of the programmer.

Given such a chunking, we showed how to insert communication code that encodes and decodes the value data into/from messages and how to determine where to insert statements that start the transfer and ensure they are completed at the right time.

The first experiments give a promising outlook on what is possible for a large class of

scientific programs. These include stencil-computations, image processing, linear algebra, differential equations, quantum chemistry, quantum physics and generally anything that can be expressed as a static control part (SCoP). It is not intended for other things like data(base) lookups or operations on sparse graphs.

11.2 Conclusions

Compared to manual optimization, the Molly-compiled programs do not perform well. Their disassembly shows that a large portion of instructions does not directly contribute to the final result. Such instructions are conversions between different integer types, spilling code (store and restore register values because the number of registers is limited), condition checks and branching, strength reduction temporaries of loop counter-derived variables[1], and others. Molly (and Polly) generates a lot of boilerplate code that is meant to be deleted in optimization passes further down the compilation pipeline. The pipeline has not been designed for this kind of input code. Furthermore, LLVM has a vectorization pass which unfortunately fails to do any vectorization in this kind of code.

However, Molly's main goal was archived: Provide automatic SPMD-based parallelization of stencil codes with perfect weak scaling. The generated code provides the structure for a good performance but has to undergo further processing to reach better performance.

Motivated by the Lattice QCD stencil – although Molly is able to distribute any code representable by SCoPs– many non-stencil codes do not work out that well. For instance, a matrix-matrix multiplication code does not scale well, because this inner loop using the current heuristic will fetch data from all the other nodes. Better methods exist that make use of the associativity of additions. The current Molly does not take this into account.

For a complete support of Lattice QCD codes with Molly, scalar products are also not well supported. At the moment reduction operations compile to very inefficient code which basically computes the reduction on every node in the system. We need explicit detection of such reduction operations, similar to what the ScalarEvolution pass does for strength reduction.

Molly might be the beginning of a more extensive framework for memory organization-based optimizations based on the polyhedral model. It can already distribute data in DMMs and reorder data in local memory with ideal scaling properties. What is still missing are transformations for typical scientific programs.

It is worth mentioning that the time to compile using Molly can be extensive. The Lattice QCD code (Section 10.2 on Page 159) takes about a minute to compile with Molly, but only a few seconds with an unmodified Clang-compiler or GCC. When the polyhedra become complicated with many disjoint pieces, the compilation can take hours and finally abort because of memory exhaustion. It is usually a sign that there is a bug in the code that add unusual dependencies to the code. Such problems may be a restriction to the general applicability of Molly's techniques. A production-level compiler may resort to a fall-back strategy with lower optimization levels.

Instead of processing directly source code written by a programmer, Molly can also be useful for code generators that cannot generate for DMMs because the data flow analysis and logic for efficient code is too complicated. One example is QIRAL [60], a high-level language embedded into LaTeX for mathematical algorithms such as conjugate gradient with a compiler that generates C source code with OpenMP annotations. Distributed memory targets are not supported by that compiler although typical Lattice QCD sizes are too large to run on a single node. Instead of adding such capabilities to the source generator, Molly could do the distribution and parallelization. The result would be a complete toolchain from an algorithmic problem formulation to scalable executable to run on machines such as Blue Gene/Q.

[1] The result of LLVM's ScalarEvolution pass

11.3 Future Work

In the current state Molly is more a prototype than a production-grade product. This partly because it is developed with a single use case in mind, Lattice QCD stencils. For potential everyday use some additional features are essential, some are nice to have, and others are future research projects. Some have already been mentioned.

11.3.1 Technical Aspects

To begin with, there is the insufficient testing of the current code which has only been tested with a couple of programs, all of which are stencils. Compilation failures with other programs are expected.

The current program output is remarkably inefficient on the assembly level. The main issues identified are the missing vectorization and boilerplate code. A big part comes from the strength reduction pass (ScalarEvolution) which creates many temporary registers to replace multiplications by additions in loops. In Lattice QCD there is a huge amount of multiplication in the loop kernel. If there are too many temporary registers, these are spilled to stack memory which adds additional load and store instructions. Unfortunately, the ScalarEvolution pass cannot just be skipped, it is interwoven with the other passes, but it needs to made aware about the number of available registers.

On the Blue Gene/Q machine and many other architectures, prefetching is very important. Thanks to the polyhedral model, the compiler has a good idea about which elements are accessed next, it should be no big problem to let it insert explicit prefetch instruction (such as **dcbt**). But inserting explicit prefetches must not make the rest of the code more complicated.

The prototype assumes that there is one module only. It is designed to work as a link-time optimization pass which assumes that all fields are discoverable and not hidden in, for instance, dynamically or statically linked libraries. Link-time compilation is not always feasible, therefore one needs to develop an ABI for `molly::array` fields and their layout descriptions.

The Molly programming interface is a C++ header, yet it assumes that the array's element types are PODs. A POD is a data type that does not require initialization and finalization using a constructor or destructor. There are open question like when to call the constructor. If a node copies data from a buffer, does it need to call an overloaded assignment operator to do this? Are the data transfers considered copy operations such that the items in the receive buffer need to be initialized using the copy constructor? If so, what is the source item passed to the copy constructor? Such question must be answered or non-POD fields declared non-conformant.

The pointer returned by `__builtin_molly_ptr` is not a regular pointer that can be used in any context. The compiler will replace the pointer by another pointer to a local buffer. The compiler cannot do this if the pointer "escapes" the compiler's tracking. For instance, it might be passed as a function argument or stored into a global variable. Such uses must be detected and forbidden by the compiler or replaced by a "fat pointer" that also contains runtime information to what the pointer is supposed to point at.

At the moment only globally-declared fields are supported. A `molly::array` cannot be declared locally in a function nor as a class member or passed to a function as an argument. Additional compiler logical is required to promote fields to first-class types that can be passed around.

This also involves some kind of runtime type information. Two fields of the same type may have different layouts at runtime. By the language rules a reference to the field type may point to one of the fields, but the reference itself does not "know" which layout the target has. Some kind of polymorphism is needed to call the functions for the correct layout. Resolving this is not trivial. When processing a SCoP the compiler needs to know the layout of the fields that are involved. The compiler may do a global data-flow analysis and force all fields that may

alias to use the same layout or determine a set of possible layouts and compile for all of them. This reduces the flexibility and still requires all the code to be known in advance. A fall-back compilation for the SCoPs that work with any layout may resolve the issue. Alternatively the programmer declares all permissible field layouts in a header file that is included by every code accessing the field.

We can develop this even further to allow switching the layouts at runtime, as it is done for spinor fields in the manually optimized Lattice QCD program (Section 4.6, Page 60). Out of a fixed set of layouts, the functions working on the current layout are selected, for instance, from a virtual method table.

All the issues of not knowing the current layout at runtime can be resolved by also optimizing the SCoP at runtime. This requires the compiler to be included into the executable, or linked in as a shared library. In fact, LLVM was started as a Just-In-Time compiler and still features the mechanisms to do so. This also resolves the issue that the cluster's node shape must be known at compile-time: The compilation of the code requiring this information is deferred to when the program runs.

This opens all the possibilities of virtual machines and JIT compilers. The compiled code may include performance monitoring. If the performance is not satisfactory it recompiles it with different optimization parameters. A multi-armed bandit algorithm may decide whether it is worth to try a different compilation or continue with the current. Libraries such as fftw [61] and Spiral use such adaptive techniques [62].

Currently, also the size of the arrays must be fixed. For one, this is a syntactical requirement of `molly::array`'s declaration as a variadic template, but is also needed because otherwise non-affine terms would appear in the polyhedral description of the array's layout. The limitation vanishes if using a Just-In-Time compiler so a dynamically-sized array type can be introduced. The types' relation is analogous to the relation between STL's `std::array` and `std::dynarray` (C++14).

More advanced constructions allow arbitrary nesting of fields and structures. Internally, a structure would be an array with fixed numbers of elements, but the following indices and the actual element type depend on it. For example, the following could be a valid declaration.

```
struct {
  molly::array<double, 8> a;
  molly::array<struct {
      char b;
      int c;
    }, 32, 32> b;
} SoA;
```

The internal representation of the element `SoA.a[0]` then is `SoA[0][0]`. Another example is `SoA.b[19][14].c` with a representation that looks like `SoA[1][19][14][1]`. Note the different number of subscripts. The byte offset of a particular element can be described by the lengths of all lexicographical elements. Something the Barvinok library [51] can do when given an polyhedron describing the size of every element.

The goal is more flexibility in memory layout transformations. Molly could transform the Structure-of-Arrays declaration to an Array-of-Structures equivalent to

```
molly::array<struct {
  double a;
  char b;
  int c;
}, 32, 32> AoS;
```

This is the famous Structure-of-Arrays to Array-of-Structures transformation used to improve spatial locality. The internal representation should be more efficient because only the `a`-elements `AoS[$i$][0].a` for $0 \geq i < 8$ existed in the `AoS` declaration and the others can be marked as occupying zero bytes in the element-lengths polyhedron. The reverse transformation, Array-of-Structures to Structure-of-Arrays, is possible as well. Which of the transformations

is the better might be automatically determined by the hardware description and the access pattern in SCoPs.

The syntax of Molly-specific pragmas is not very user-friendly. Instead of exposing the syntax used by ISL, they may instead use higher-level primitives like specifying a block-cyclic layout, tiling, etc.

We may support a broader definition of what a SCoP is. There are techniques that allow while loops, conditionals and array subscripts with non-affine expressions [39]. With the cost of lower efficiency, a less strict model may allow approximate data accesses. For instance, depending on some condition only evaluable at runtime, the code may either overwrite an array element A or element B. In this case the compile must be pessimistic and assume that both of them have been overwritten, but one of them with the old value.

Another idea is to support indirect array accesses. Instead of calculating which array element index is accessed, a second array stores the index for each loop counter value. We already used this technique for the halfspinor layout's Hopping Matrix implementation (Section 4.4.2.1) where the target pointers were stored in the array `target_ptr`. It costs an additional memory access, but the function computing the index can be complicated, representable only with affine functions of many pieces. In such cases even the compilation time may be lowered because ISL has just one case left to cope with. The index array would be automatically computed by the compiler. Either as a constant array, or – to reduce the size of the executable – as an initialization function that runs at program startup. Such index array may also make more complicated data reorderings feasible such as space filling curves, like Z-curves or Hilbert curves.

11.3.2 Algorithmic Improvements

As mentioned earlier more kinds of algorithms need to be supported than just stencil, even though stencils already cover many use cases. Especially reduction operations (scalar product, norm, checksums, etc.) are another important class of codes, often used in combination with others. For instance, to implement a CG solver (Algorithm 2.1 on Page 25), norm and scalar product are both required. Elementwise addition and subtraction can be seen as a special case of 1-points stencil operations with two input arrays.

The default data and statement execution distribution rules used by Molly are deliberately kept simple and predictable. Other researchers already looked into the topic of data alignment [46, 47], some of which might be implemented in Molly.

All three chunking algorithms try to reduce the number of messages only, without taking in account other optimization goals like improving asynchronous communication. Techniques such as index-set splitting in order to partition the statement instance that can be executed immediately from those that require data from receive buffers could be useful.

There might be better heuristics as the one presented and implemented in Molly. It is based on the idea that the programmer writes codes that are already well-structured. Values generated in a for loop are not often used in the same innermost loop but on a different node. Of course, this is not always the case. Better heuristics may relieve the programmer from the burden of restructuring the code.

The modified SCoPs passed to Polly for code generation contain many special cases that are inefficient to represent as polyhedra. This is what makes compilation take so long and also all this special cases appear as mode code with branching in the final executable. It would be nice if we could drastically reduce these for smaller and faster codes as well as compilation.

11.4 Related Work

In the past, efforts to shift the responsibility of parallelization and data distribution to the compiler were not very successful. Probably the most noteworthy project is HPF. Today, HPF

plays a minor role because of many reasons [63]. One of them is that compilers promised to optimize code, but could only do so for specifically structured code. Every compiler could only optimize a different set of code patterns.

Probably the most similar work is [64]. The authors also build polyhedral subsets of array indexes that have to be transferred to different nodes. However, their set operations are very different and the implementation is capable of more advanced techniques like index set splitting and merge of communication events. Their tool dHPF is a HPF to Fortran77-with-MPI source-to-source compiler and uses the Omega [44] library for the set operations. It only allows block-cyclic data distribution and uses the owner-computes policy for statements. More work of this kind are the Last Write Tree [65] and [66].

A different kind of optimization partitions the work into tiles which are then distributed to the nodes [67, 68, 69]. The tiles' size is fixed at compile-time, but the number of nodes doesn't need to be fixed until runtime. Therefore most of these works uses a notion of "virtual" processors. Virtual processors are mapped to physical ones at runtime.

A novel technique has been presented in [70]. It combines syntactical code analysis with a dynamic part that allows non-static code behavior such as indirect array accesses. These can normally not be represented in a SCoP.

Again another approach is to invent a language that has high-level operators on multi-dimensional arrays. ZPL [71] belongs to this category. Instead of "accessing the element to the left" there is a shift operator that moves the entire field one index to the right. Such languages require rethinking of algorithms, but high-level operators can be pre-implemented and algebraically optimized efficiently. Other languages of this category are Fortress [72], Chapel [73] and X10 [74].

The middle way are extensions to existing languages. To name a few, there is OpenACC, OmpSs, C++ AMP, Universal Parallel C (UPC), etc. The first three are language extensions that target accelerators like GPUs to offload computation from the main CPU. The programmer has to state explicitly the data region that must be available on the device memory. UPC and X10 are representative of languages using PGAS. They create an address space window for non-local data. Code flow still has to be explicitly organized and synchronized.

A Original Hopping Matrix Source

```
typedef double _Complex complex;

typedef struct {
   complex c0,c1,c2;
} su3_vector;

typedef struct {
   su3_vector s0,s1,s2,s3;
} spinor;

typedef struct {
   complex c00,c01,c02,c10,c11,c12,c20,c21,c22;
} su3;

#define _vector_add(r,s1,s2)  \
  (r).c0 = (s1).c0 + (s2).c0; \
  (r).c1 = (s1).c1 + (s2).c1; \
  (r).c2 = (s1).c2 + (s2).c2;

#define _vector_sub(r,s1,s2)  \
  (r).c0 = (s1).c0 - (s2).c0; \
  (r).c1 = (s1).c1 - (s2).c1; \
  (r).c2 = (s1).c2 - (s2).c2;

#define _vector_i_add(r,s1,s2)  \
  (r).c0 = (s1).c0 + I*(s2).c0; \
  (r).c1 = (s1).c1 + I*(s2).c1; \
  (r).c2 = (s1).c2 + I*(s2).c2;

#define _vector_i_sub(r,s1,s2)  \
  (r).c0 = (s1).c0 - I*(s2).c0; \
  (r).c1 = (s1).c1 - I*(s2).c1; \
  (r).c2 = (s1).c2 - I*(s2).c2;

#define _su3_multiply(r,u,s)                                  \
  (r).c0 = (u).c00*(s).c0 + (u).c01*(s).c1 + (u).c02*(s).c2; \
  (r).c1 = (u).c10*(s).c0 + (u).c11*(s).c1 + (u).c12*(s).c2; \
  (r).c2 = (u).c20*(s).c0 + (u).c21*(s).c1 + (u).c22*(s).c2;

#define _su3_inverse_multiply(r,u,s)                                                    \
  (r).c0 = conj((u).c00)*(s).c0 + conj((u).c10)*(s).c1 + conj((u).c20)*(s).c2; \
  (r).c1 = conj((u).c01)*(s).c0 + conj((u).c11)*(s).c1 + conj((u).c21)*(s).c2; \
  (r).c2 = conj((u).c02)*(s).c0 + conj((u).c12)*(s).c1 + conj((u).c22)*(s).c2;

#define _complex_times_vector(r,c,s) \
  (r).c0 = (c)*(s).c0;               \
  (r).c1 = (c)*(s).c1;               \
  (r).c2 = (c)*(s).c2;
```

```
#define _complexcjg_times_vector(r,c,s) \
  (r).c0 = conj(c)*(s).c0;              \
  (r).c1 = conj(c)*(s).c1;              \
  (r).c2 = conj(c)*(s).c2;

#define _vector_assign(r,s) \
  (r).c0 = (s).c0;          \
  (r).c1 = (s).c1;          \
  (r).c2 = (s).c2;

#define _vector_add_assign(r,s) \
  (r).c0 += (s).c0;             \
  (r).c1 += (s).c1;             \
  (r).c2 += (s).c2;

#define _vector_sub_assign(r,s) \
  (r).c0 -= (s).c0;             \
  (r).c1 -= (s).c1;             \
  (r).c2 -= (s).c2;

#define _vector_i_add_assign(r,s) \
  (r).c0 += (I*(s).c0);           \
  (r).c1 += (I*(s).c1);           \
  (r).c2 += (I*(s).c2);

#define _vector_i_sub_assign(r,s) \
  (r).c0 -= (I*(s).c0);           \
  (r).c1 -= (I*(s).c1);           \
  (r).c2 -= (I*(s).c2);


static su3_vector psi1, psi2, psi, chi, phi1, phi3;

/* 8. */
/* l output , k input*/
/* for ieo=0, k resides on  odd sites and l on even sites */
void Hopping_Matrix(int ieo, spinor * const l, spinor * const k){
  int ix,iy;
  int ioff,ioff2,icx,icy;
  su3 * restrict up, * restrict um;
  spinor * restrict r, * restrict sp, * restrict sm;
  spinor temp;

#ifdef _GAUGE_COPY
  if(g_update_gauge_copy) {
    update_backward_gauge();
  }
#endif

  /* for parallelization */
# if (defined MPI && !(defined _NO_COMM))
  xchange_field(k, ieo);
#endif

  if(k == l) {
    printf("Error in H_psi (simple.c):\n");
    printf("Arguments k and l must be different\n");
    printf("Program aborted\n");
    exit(1);
  }
  if(ieo == 0) {
    ioff = 0;
  } else {
    ioff = (VOLUME+RAND)/2;
  }
```

```
  ioff2 = (VOLUME+RAND)/2-ioff;

  /*************** loop over all lattice sites ***************/
  for (icx = ioff; icx < (VOLUME/2 + ioff); icx++){
    ix = g_eo2lexic[icx];
    r = l+(icx-ioff);

    /********************** direction +0 **********************/
    iy = g_iup[ix][0]; icy = g_lexic2eosub[iy];

    sp = k+icy;
#if ((defined _GAUGE_COPY))
    up = &g_gauge_field_copy[icx][0];
#else
    up = &g_gauge_field[ix][0];
#endif

    _vector_add(psi,(*sp).s0,(*sp).s2);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka0,chi);

    _vector_assign(temp.s0,psi);
    _vector_assign(temp.s2,psi);

    _vector_add(psi,(*sp).s1,(*sp).s3);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka0,chi);

    _vector_assign(temp.s1,psi);
    _vector_assign(temp.s3,psi);

    /********************** direction -0 **********************/
    iy = g_idn[ix][0]; icy = g_lexic2eosub[iy];

    sm=k+icy;
#if ((defined _GAUGE_COPY))
    um = up+1;
#else
    um = &g_gauge_field[iy][0];
#endif

    _vector_sub(psi,(*sm).s0,(*sm).s2);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka0,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_sub_assign(temp.s2,psi);

    _vector_sub(psi,(*sm).s1,(*sm).s3);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka0,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_sub_assign(temp.s3,psi);

    /********************** direction +1 **********************/
    iy = g_iup[ix][1]; icy = g_lexic2eosub[iy];

    sp = k+icy;
#if ((defined _GAUGE_COPY))
    up = um+1;
```

```
#else
    up += 1;
#endif

    _vector_i_add(psi,(*sp).s0,(*sp).s3);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka1,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_i_sub_assign(temp.s3,psi);

    _vector_i_add(psi,(*sp).s1,(*sp).s2);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka1,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_i_sub_assign(temp.s2,psi);

    /********************** direction -1 ************************/
    iy = g_idn[ix][1]; icy = g_lexic2eosub[iy];

    sm = k+icy;
#ifndef _GAUGE_COPY
    um = &g_gauge_field[iy][1];
#else
    um = up+1;
#endif

    _vector_i_sub(psi,(*sm).s0,(*sm).s3);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka1,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_i_add_assign(temp.s3,psi);

    _vector_i_sub(psi,(*sm).s1,(*sm).s2);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka1,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_i_add_assign(temp.s2,psi);

    /********************** direction +2 ************************/
    iy = g_iup[ix][2]; icy = g_lexic2eosub[iy];

    sp = k+icy;
#if ((defined _GAUGE_COPY))
    up = um+1;
#else
    up += 1;
#endif
    _vector_add(psi,(*sp).s0,(*sp).s3);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka2,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_add_assign(temp.s3,psi);

    _vector_sub(psi,(*sp).s1,(*sp).s2);
```

```
    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka2,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_sub_assign(temp.s2,psi);


    /********************** direction -2 ***********************/
    iy = g_idn[ix][2]; icy = g_lexic2eosub[iy];

    sm = k+icy;
#ifndef _GAUGE_COPY
    um = &g_gauge_field[iy][2];
#else
    um = up +1;
#endif

    _vector_sub(psi,(*sm).s0,(*sm).s3);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka2,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_sub_assign(temp.s3,psi);

    _vector_add(psi,(*sm).s1,(*sm).s2);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka2,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_add_assign(temp.s2,psi);

    /********************** direction +3 ***********************/
    iy = g_iup[ix][3]; icy = g_lexic2eosub[iy];

    sp = k+icy;
#if ((defined _GAUGE_COPY))
    up = um+1;
#else
    up += 1;
#endif
    _vector_i_add(psi,(*sp).s0,(*sp).s2);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka3,chi);

    _vector_add_assign(temp.s0,psi);
    _vector_i_sub_assign(temp.s2,psi);

    _vector_i_sub(psi,(*sp).s1,(*sp).s3);

    _su3_multiply(chi,(*up),psi);
    _complex_times_vector(psi,ka3,chi);

    _vector_add_assign(temp.s1,psi);
    _vector_i_add_assign(temp.s3,psi);

    /********************** direction -3 ***********************/
    iy = g_idn[ix][3]; icy = g_lexic2eosub[iy];

    sm = k+icy;
#ifndef _GAUGE_COPY
    um = &g_gauge_field[iy][3];
#else
```

```
    um = up+1;
#endif

    _vector_i_sub(psi,(*sm).s0,(*sm).s2);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka3,chi);

    _vector_add((*r).s0, temp.s0, psi);
    _vector_i_add((*r).s2, temp.s2, psi);

    _vector_i_add(psi,(*sm).s1,(*sm).s3);

    _su3_inverse_multiply(chi,(*um),psi);
    _complexcjg_times_vector(psi,ka3,chi);

    _vector_add((*r).s1, temp.s1, psi);
    _vector_i_sub((*r).s3, temp.s3, psi);
    /********************** end of loop **********************/
  }
}
```

B Symbol Summary

B.1 Sets, Relations and Functions

B.1.1 General

Symbol	Superset	Description
$<$	$\mathbb{R} \times \mathbb{R}$	less-than relation for real numbers
$\ll$	$\mathbb{R}^n \times \mathbb{R}^n$ $= \{(u, v) \mid \exists i = 1..n : u_i < v_i \wedge \forall j = 1..i-1 : u_j = v_j\}$	*lexicographic* less-than relation for vectors

B.1.2 Cluster Machine

Symbol	Superset	Description
$\mathcal{P}$	$\mathbb{Z}^{N_\mathcal{P}}$	all processes/ranks of a DMM

B.1.3 Statements

Symbol	Superset	Description
Ω		all *statements* in the SCoP
Ω'	$= \Omega \cup \{\top, \bot\}$	all statements including prologue and epilogue
$\mathcal{D}_S$	$\mathbb{Z}^{N_S}$	iteration space/*domain* of S
$\mathcal{D}$	$= \left\{(S, \vec{i}) \;\middle|\; S \in \Omega, \vec{i} \in \mathcal{D}_S\right\}$	all statement *instances*
π_S	$\mathcal{D}_S \times \mathcal{P}$	*work distribution* for instances of S
π_Ω	$\mathcal{D} \times \mathcal{P}$ $= \left\{(S, \vec{i}, \vec{p}) \;\middle|\; S \in \Omega, \vec{i} \in \mathcal{D}_S, (\vec{i}, \vec{p}) \in \pi_S\right\}$	all statement instance *executions*

B.1.4 Scheduling

Symbol	Superset	Description
$\boxdot$	$\mathbb{Z}^{N_\boxdot}$	scatter space
θ_S	$\mathcal{D}_S \to \boxdot$	*scatter function* of statement S
θ	$\mathcal{D} \to \boxdot$	*schedule*, all scatter functions in a SCoP
	$= \left\{ (S, \vec{i}) \mapsto \vec{t} \;\middle|\; (S, \vec{i}) \in \mathcal{D}, \vec{t} = \theta_S(\vec{i}) \right\}$	
$<_{\text{exec}}$	$\mathcal{D} \times \mathcal{D}$	"executes before" relation
	$= \left\{ ((S, \vec{i}), (R, \vec{j})) \;\middle|\; \theta_S(\vec{i}) \ll \theta_R(\vec{j}) \right\}$	if derived from schedule

B.1.5 Data

Symbol	Superset	Description
$\mathcal{V}$		all used *variables* (scalars or arrays)
$\mathcal{F}$	$\mathcal{V}$	all *fields* (distributed arrays)
$\mathcal{V} \setminus \mathcal{F}$	$\mathcal{V}$	non-distributed arrays and scalars
$\mathcal{I}_A$	$\mathbb{Z}^{N_A}$	*indexset* of A, set of valid subscripts
$\mathcal{E}$		all (array) *elements*
	$= \left\{ (A, \vec{k}) \;\middle|\; A \in \mathcal{V}, \vec{k} \in \mathcal{I}_A \right\}$	
π_F	$\mathcal{I}_F \times \mathcal{P}$	*data distribution* for elements of field F
$\pi_\mathcal{F}$	$\mathcal{E} \times \mathcal{P}$	all element *storage locations*
	$= \left\{ (F, \vec{k}, \vec{p}) \;\middle|\; F \in \mathcal{F}, \vec{k} \in \mathcal{I}_F, (\vec{k}, \vec{p}) \in \pi_F \right\}$	

B.1.6 Data Access

Symbol	Superset	Description
λ_{read}	$\mathcal{D} \to \mathcal{E}$	read access relation
λ_{write}	$\mathcal{D} \to \mathcal{E}$	write access relation
λ	$\mathcal{D} \to \mathcal{E}$	access relation
	$= \lambda_{\text{read}} \cup \lambda_{\text{write}}$	

B.1.7 Dependencies

Symbol	Superset	Description
$<_{\text{flow}}$	$<_{\text{exec}}$	Read-After-Write dependency
$<_{\text{anti}}$	$<_{\text{exec}}$	Write-After-Read dependency
$<_{\text{output}}$	$<_{\text{exec}}$	Write-After-Write dependency
$<_{\text{dep}}$	$<_{\text{exec}}$ $= (<_{\text{flow}} \cup <_{\text{anti}} \cup <_{\text{output}})^*$	dependence relation
σ	$\mathcal{D} \times \mathcal{E} \times \mathcal{D}$	direct data flow, transfer of a value from generator to consumer through an element

B.1.8 Chunking

Symbol	Superset	Description
$\mathcal{X}$		*chunk space*, can be chosen arbitrarily
φ	$\sigma \to \mathcal{X}$	*chunking function* (Definition 8.1, Page 115)

B.2 Typical Placeholder and Constant Names

Throughout this thesis, if a symbol is not explicitly defined to be an element of some set, it belongs to the set mentioned in the table below.

Set	Elmt. Description	Placeholder variables	Typical elements
$\mathcal{P}$	Node	$\vec{p}$	(p, q), $\vec{0}$ (master node)
Ω	Statement	S, R, G (for Generator), C (for Consumer)	S1, S2,... $\top$,$\bot$, send_wait, send, recv_wait, recv
$\mathcal{D}_S$	Loop context	$\vec{i}$, $\vec{j}$	(i, j)
$\mathcal{D}$	Instance		$(S, \vec{i})$
π_Ω	Execution		$(S, \vec{i}, \vec{p})$
$\circlearrowright$	Scattering	$\vec{t}$	
$\mathcal{V}$	Variable	A	`result`, `sum`, `avg`, `neighbors`
$\mathcal{F}$	Field	F, E	`field`, `source`, `sink`, `gauge`
$\mathcal{I}_A$	Index	$\vec{k}$, $\vec{g}$	(t, x, y, x)
$\mathcal{E}$	Element		$(A, \vec{k})$
$\pi_\mathcal{F}$	Storage location		$(F, \vec{k}, \vec{p})$
λ_{read}	Read access		$(C, \vec{i}) \mapsto (A, \vec{k})$
λ_{write}	Write access		$(G, \vec{i}) \mapsto (A, \vec{k})$
$\mathcal{X}$	Chunk	χ	
σ	Flow		$((G, \vec{i}_G), (A, \vec{k}), (C, \vec{i}_C))$
$\sigma \to \mathcal{X}$	Chunking function	φ, ρ	id (trivial chunking function)

Bibliography

[1] Kenneth G. Wilson. "Confinement of quarks". In: *Physical Review D* 10.8 (Oct. 1974), pp. 2445–2459. DOI: 10.1103/PhysRevD.10.2445.

[2] Jack Laiho, Enrico Lunghi, and Ruth S. Van de Water. "Flavor Physics in the LHC era: the role of the lattice". In: *Proceedings of Science*. 29th International Symposium on Lattice Field Theory. Lattice 2011. (Squaw Valley, USA, July 10–16, 2011). Trieste, Italy: International School for Advanced Studies (SISSA), PoS(Lattice 2011)018. arXiv: 1204.0791 [hep-ph].

[3] Carsten Urbach. *tmLQCD Package Documentation*. included in the tmLQCD software distribution. 2009.

[4] Donald E. Knuth. "Structured Programming with go to Statements". In: *ACM Computing Surveys (CSUR)* 6.4 (1974), pp. 261–301.

[5] IBM. *C and C++ Compilers family*. URL: http://www-03.ibm.com/software/products/en/ccompfami (visited on 08/11/2014).

[6] *clang: a C language family frontend for LLVM*. Version 3.5. URL: http://clang.llvm.org.

[7] Christof Gattringer and Christian B. Lang. "QCD on the lattice – a first look". In: *Quantum Chromodynamics on the Lattice*. Vol. 788. Lecture Notes in Physics. Springer, 2010, pp. 25–41. DOI: 10.1007/978-3-642-01850-3_2.

[8] Rajan Gupta. "Introduction to Lattice QCD". In: *Probing the Standard Model of Particle Interactions*. LXVIII Les Houches Summer School. (Les Houches, July 28–Oct. 5, 1997), pp. 83–219. arXiv: hep-lat/9807028 [hep-lat].

[9] Karl Jansen and Carsten Urbach. "tmLQCD: a program suite to simulate Wilson Twisted mass Lattice QCD". In: *Computer Physics Communications* 180.12 (Dec. 2009), pp. 2717–2738. DOI: 10.1016/j.cpc.2009.05.016. arXiv: 0905.3331 [hep-lat].

[10] Peter A. Boyle and et. al. *QCDOC Architecture*. URL: http://phys.columbia.edu/~cqft/qcdoc/qcdoc.htm (visited on 08/13/2014).

[11] *A2 Processor User's Manual for Blue Gene/Q*. Version 1.3. IBM, Aug. 23, 2012.

[12] *POWER ISA Version 2.07*. IBM, May 3, 2013. URL: http://www.power.org/documentation/power-isa-version-2-07/ (visited on 06/27/2013).

[13] Megan Gilge. *IBM System Blue Gene Solution Blue Gene/Q Application Development*. IBM Redbooks, June 12, 2013. ISBN: 0-738-43823-5.

[14] Carlos Sosa and Brant Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM Redbooks, Nov. 11, 2009. ISBN: 0-738-43333-0.

[15] Gary L. Mullen-Schultz and Carlos Sosa. *IBM System Blue Gene Solution: Application Development*. IBM Redbooks, June 11, 2008. ISBN: 0-738-48939-5.

[16] Peter A. Boyle, Chulwoo Jung, and Tilo Wettig. "The QCDOC supercomputer: Hardware, software, and performance". In: *eConf*. Computing in High Energy and Nuclear Physics. CHEP 2003. (La Jolla, USA, Mar. 24–28, 2003). Vol. C0303241, THIT001–THIT003. arXiv: hep-lat/0306023 [hep-lat].

[17] Thomas Fox, Michael Gschwind, and Jaime Moreno. *QPX Architecture: Quad Processing eXtension to the Power ISA*. Reference Manual. Yorktown Heights, USA: IBM Research Division, June 13, 2012. URL: http://domino.research.ibm.com/library/cyberdig.nsf/papers/BBD8BB0F809822E885257A2F00511C69 (visited on 08/13/2014).

[18] Alexandre E. Eichenberger and K. O'Brien. "Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q". In: *IBM Journal of Research and Development* 57.1/2 (Jan.–Mar. 2013), 8:1–8:8. DOI: 10.1147/JRD.2012.2228769.

[19] Amy Wang et al. "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT 2012. (Minneapolis, USA, Sept. 19–23, 2012). ACM, pp. 127–136. DOI: 10.1145/2370816.2370836.

[20] Xing Zhou et al. "Hierarchical Overlapped Tiling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization. CGO 2012*. (San Jose, USA). ACM, pp. 207–218. DOI: 10.1145/2259016.2259044.

[21] Peter A. Boyle. "The BAGEL assembler generation library". In: *Computer Physics Communications* 180.12 (Dec. 2009), pp. 2739–2748. DOI: 10.1016/j.cpc.2009.08.010. URL: http://www.ph.ed.ac.uk/~paboyle/bagel/Bagel.html (visited on 08/13/2014).

[22] Peter O. Boyle. "The BlueGene/Q Supercomputer". In: *Proceedings of Science*. 30th International Symposium on Lattice Field Theory. Lattice 2012. (Cairns, Australia, June 23–29, 2012). Trieste, Italy: International School for Advanced Studies (SISSA), PoS(Lattice 2012)020.

[23] *Columbia Physics System (CPS)*. URL: http://qcdoc.phys.columbia.edu/cps.html (visited on 08/13/2014).

[24] Robert G. Edwards and Balint Joo. "The Chroma Software System for Lattice QCD". In: *Nuclear Physics B – Proceedings Supplements* 140 (2005), pp. 832–834. DOI: 10.1016/j.nuclphysbps.2004.11.254. arXiv: hep-lat/0409003 [hep-lat].

[25] Jun Doi. "Peta-Scale Lattice Quantum Chromodynamics on a Blue Gene/Q Supercomputer". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC 2012. (Salt Lake City, USA, Nov. 11–15, 2012). IEEE Computer Society.

[26] Guido Cossu et al. "JLQCD IroIro++ lattice code on BG/Q". In: (), PoS(LATTICE 2013)482. arXiv: 1311.0084 [hep-lat].

[27] Bálint Joó et al. "Lattice QCD on Intel Xeon Phi coprocessors". In: *Supercomputing*. 28th International Supercomputing Conference. ISC 2013. (Leibzig, Germany, June 15–20, 2013). Vol. 7905. Lecture Notes in Computer Science. Springer, 2013, pp. 40–54. DOI: 10.1007/978-3-642-38750-0_4.

[28] M. A. Clark et al. "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs". In: *Computer Physics Communications* 181.9 (Sept. 2010), pp. 1517–1528. DOI: 10.1016/j.cpc.2010.05.002. arXiv: 0911.3191 [hep-lat].

[29] Leslie Lamport and David Presberg. *The Parallel Execution of FORTRAN DO Loops*. Final Report. Applied Data Research, Inc., 1972.

[30] Leslie Lamport. "The Parallel Execution of DO Loops". In: *Communications of the ACM* 17.2 (Feb. 1974), pp. 83–93. DOI: 10.1145/360827.360844.

[31] Leslie Lamport. *My Writings*. URL: `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#coordinate` (visited on 12/02/2013).

[32] Paul Feautrier. "Parametric Integer Programming". In: *RAIRO Recherche opérationnelle* 22.3 (Sept. 1988). Ed. by A. Ridha Mahjoub, pp. 243–268.

[33] Paul Feautrier. "Some efficient solutions to the affine scheduling problem – Part I. One-dimensional time". In: *International Journal of Parallel Programming* 21.6 (Oct. 1992), pp. 313–347. DOI: `10.1007/BF01407835`.

[34] Paul Feautrier. "Some efficient solutions to the affine scheduling problem – Part II. Multidimensional time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. DOI: `10.1007/BF01379404`.

[35] Paul Feautrier. "Scalable and Structured Scheduling". In: *International Journal of Parallel Programming* 34.5 (Oct. 2006), pp. 459–487. DOI: `10.1007/s10766-006-0011-4`.

[36] Alexander Schrijver. *Theory of linear and integer programming*. Discrete mathematics and optimization. Wiley-Interscience, July 1994. ISBN: 0-471-90854-1.

[37] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and automatic parallelization*. Birkhäuser Boston, Apr. 2000. ISBN: 0-817-64149-1.

[38] Sven Verdoolaege. *Integer Set Library*. Version 0.13. URL: `http://isl.gforge.inria.fr`.

[39] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Compiler Construction*. 19th International Conference. CC 2010. (Paphos, Cyprus, Mar. 20–28, 2010). Vol. 6011. Lecture Notes in Computer Science. Springer, pp. 283–303. DOI: `10.1007/978-3-642-11970-5_16`.

[40] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2008. (Tucson, USA, June 7–17, 2008). ACM, pp. 101–113. DOI: `10.1145/1375581.1375595`. URL: `http://pluto-compiler.sourceforge.net`.

[41] Martin Griebl and Christian Lengauer. "The loop parallelizer LooPo". In: *Languages and Compilers for Parallel Computing*. 9th International Workshop of Languages and Compilers for Parallel Computing. LPLC 1996. (San Jose, USA, Aug. 8–10, 1996). Vol. 1239. Lecture Notes in Computer Science. Springer, pp. 311–320.

[42] Louis-Noël Pouchet, Cédric Bastoul, and Albert Cohen. "LeTSee: the LEgal Transformation SpacE Explorator". In: *Proceedings of the 3rd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*. ACACES 2007. (L'Aquila, Italy, July 15–20, 2007). 2007, pp. 247–251.

[43] Cédric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*. PACT 2004. (Antibes Juan-les-Pins, France, Sept. 29–Oct. 3, 2004). IEEE Computer Society, pp. 7–16. DOI: `10.1109/PACT.2004.1342537`.

[44] Wayne Kelly and William Pugh. "A framework for Unifying Reordering Transformations". In: IEEE First International Conference on Algorithms and Architectures for Parallel Processing. ICAPP 1995. (Brisbane, Australia, Apr. 19–21, 1995). Vol. 1. IEEE Computer Society, pp. 153–162. DOI: `10.1109/ICAPP.1995.472180`.

[45] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. "Generation Of Efficient Nested Loops From Polyhedra". In: *International Journal of Parallel Programming* 28.5 (Oct. 2000), pp. 469–498. DOI: `10.1023/A:1007554627716`.

[46] Jingke Li and Marina Chen. "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays". In: 3rd Symposium on the Frontiers of Massively Parallel Computation. Frontiers 1990. (College Park, USA, Oct. 8–10, 1990). IEEE Computer Society, pp. 424–433.

[47] Eduard Ayguadé, Jordi Garcia, and Ulrich Kremer. "Tools and techniques for automatic data layout: A case study". In: *Parallel Computing* 24.3-4 (May 1998), pp. 557–578. DOI: 10.1016/S0167-8191(98)00025-8.

[48] Cédric Bastoul and Paul Feautrier. "Improving Data Locality by Chunking". In: *Compiler Construction*. 12th International Conference. CC 2003. (Warsaw, Poland, Apr. 7–11, 2003). Vol. 2622. Lecture Notes in Computer Science. Springer, 2003, pp. 320–334.

[49] Leon Mirsky. "A Dual of Dilworth's Decomposition Theorem". In: *American Mathematical Monthly* 78.8 (Oct. 1971), pp. 876–877. DOI: 10.2307/2316481.

[50] Alexander I. Barvinok. "A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension Is Fixed". In: 34th Annual Symposium on Foundations of Computer Science. FOCS 1993. (Palo Alto, USA, Nov. 3–5, 1993). IEEE Computer Society, pp. 566–572. DOI: 10.1109/SFCS.1993.366830.

[51] Sven Verdoolaege et al. "Counting Integer Points in Parametric Polytopes using Barvinok's Rational Functions". In: *Algorithmica* 48.1 (Feb. 9, 2007), pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

[52] Armin Größlinger. "Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes". In: *Compiler Construction*. 18th International Conference. CC 2009. (York, UK, Mar. 22–29, 2009). Vol. 5501. Lecture Notes in Computer Science. Springer, 2009, pp. 236–250. DOI: 10.1007/978-3-642-00722-4_17.

[53] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". Master's Thesis. Urbana-Champaign, USA: Computer Science Dept., University of Illinois, Dec. 2002. URL: http://llvm.org.

[54] Tobias Grosser. "Enabling Polyhedral Optimizations in LLVM". Diploma Thesis. Department of Informatics and Mathematics, University of Passau, Apr. 2011.

[55] Sebastian Pop et al. "GRAPHITE: Polyhedral Analyses and Optimizations for GCC". In: *Proceedings of the 2006 GCC Developers Summit*. (Ottawa, Canada, June 28–30, 2006).

[56] *ISO/IEC 9899:2011. Information technology – Programming languages – C*. Geneva, Switzerland: International Organization for Standardization, Dec. 8, 2011. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=57853 (visited on 08/16/2014).

[57] *The Often Misunderstood GEP Instruction*. URL: http://llvm.org/docs/GetElementPtr.html (visited on 12/07/2013).

[58] *PoCC: the Polyhedral Compiler Collection*. URL: http://www.cs.ucla.edu/~pouchet/software/pocc/ (visited on 02/24/2014).

[59] Hal Finkel. *bgclang (LLVM/clang on the BG/Q)*. URL: http://trac.alcf.anl.gov/projects/llvm-bgq (visited on 05/24/2014).

[60] Denis Barthou et al. "QIRAL: A High Level Language for Lattice QCD Code Generation". In: *Proceedings Fifth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software*. PLACES 2012. (Tallinn, Estonia, Mar. 31, 2012). DOI: 10.4204/EPTCS.109. arXiv: 1208.4035 [cs.PL].

[61] Matteo Frigo and Steven G. Johnson. "FFTW: An Adaptive Software Architecture for the FFT". In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*. (Seattle, USA, May 15, 1998). Vol. 3. IEEE Computer Society, pp. 1381–1384. DOI: 10.1109/ICASSP.1998.681704. URL: http://fftw.org.

[62] Fréderic de Mesmay. "On the Computer Generation of Adaptive Numerical Libraries". PhD Thesis. Pittsburgh, USA: Carnegie-Mellon University, May 2010.

[63] Ken Kennedy, Charles Koelbel, and Hans Zima. "The Rise and Fall of High Performance Fortran: An Historical Object Lesson". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL-III. (San Diego, USA, June 9–10, 2007). ACM, 2007, pp. 7.1–7.22. DOI: 10.1145/1238844.1238851.

[64] Vikram S. Adve and John M. Mellor-Crummey. "Using Integer Sets for Data-Parallel Program Analysis and Optimization". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. PLDI 1998. (Montreal, Canada, June 17–19, 1998). ACM, pp. 186–198. DOI: 10.1145/277650.277721.

[65] Saman P. Amarasinghe and Monica S. Lam. "Communication optimization and code generation for distributed memory machines". In: *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. PLDI 1993. (Albuquerque, USA, June 21–25, 1993). ACM, pp. 126–138. DOI: 10.1145/155090.155102.

[66] Roshan Dathathri et al. "Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-Memory". In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. PACT 2013. (Edinburgh, UK, Sept. 7–11, 2013). IEEE Computer Society, pp. 375–386.

[67] Uday Bondhugula. *Automatic Distributed-Memory Parallelization and Code Generation using the Polyhedral Framework*. Research Report IISc-CSA-TR-2011-3. Bangalore: Indian Institute of Science, Sept. 2011.

[68] Michael Claßen and Martin Griebl. "Automatic Code Generation for Distributed Memory Architectures in the Polytope Model". In: *Proceedings of the 20th International Symposium on Parallel and Distributed Processing*. IPDPS 2006. (Rhodes Island, Greece, Apr. 25–29, 2006). IEEE Computer Society, 7pp. DOI: 10.1109/IPDPS.2006.1639500.

[69] Tomofumi Yuki and Sanjay Rajopadhye. "Memory Allocations for Tiled Uniform Dependence Programs". In: *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*. IMPACT 2013. (Berlin, Germany, Jan. 13, 2013), pp. 13–22. URL: http://nbn-resolving.de/urn:nbn:de:bvb:739-opus-26930.

[70] Mahesh Ravishankar et al. "Code Generation for Parallel Execution of a Class of Irregular Loops on Distributed Memory Systems". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC 2012. (Salt Lake City, USA, Nov. 11–15, 2012). IEEE Computer Society. DOI: 10.1109/SC.2012.30.

[71] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, 1999. ISBN: 0-262-69217-1.

[72] Guy L. Steele. "Parallel Programming and Parallel Abstractions in Fortress". In: *Functional and Logic Programming*. 8th International Symposium,. FLOPS 2006. (Fuji-Susono, Japan, Apr. 24–26, 2006). Vol. 3945. Lecture Notes in Computer Science. Springer, 2006. DOI: 10.1007/11737414_1.

[73] B. L. Chamberlain, D. Callahan, and H. P. Zima. "Parallel Programmability and the Chapel Language". In: *International Journal of High Performance Computing Applications* 21.3 (Aug. 2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[74] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. "X10: Programming for hierarchical parallelism and non-uniform data access". In: *Proceedings of the International Workshop on Language Runtimes, OOPSLA*. Vol. 30. 2004.