Technische Universität Dortmund Studienjahr 2007/08

Endbericht der Projektgruppe 512

SmartCell: Clevere Algorithmen für den Cell-Prozessor

Till Barz Thorsten Deinert Danny van Dyk
Markus Geveler David Gies Nina Harmuth
Volker Jung Mathias Kadolsky Sven Mallach
André Matuschek Joachim Messer Dirk Ribbrock

14. April 2008

Veranstalter der Projektgruppe 512:
Carsten Gutwenger, Prof. Petra Mutzel
(Fakultät für Informatik, LS 11: Algorithm Engineering)
Dominik Göddeke, Prof. Stefan Turek
(Fakultät Mathematik, Mathematik III: Angewandte Mathematik)

Inhaltsverzeichnis

1. Beschreibung der Projektgruppe

1.1. Veranstalter und Teilnehmer der Projektgruppe

Die Veranstalter der Projektgruppe 512: „SmartCell: Clevere Algorithmen für den Cell-Prozessor“ waren Prof. Dr. Petra Mutzel und Carsten Gutwenger von der Fakultät für Informatik der Technischen Universität Dortmund, Lehrstuhl 11 (*Algorithm Engineering*), sowie Prof. Dr. Stefan Turek und Dominik Göddeke von der Fakultät Mathematik, Lehrstuhl III (*Angewandte Mathematik*). Veranstaltungszeitraum waren das Sommersemester 2007 und das Wintersemester 2007/08.

Das Thema der Projektgruppe ist der Entwurf und die Implementierung von Algorithmen für die Architektur der IBM *Cell Broadband Engine*[1] zur Berechnung physikalisch korrekter Wellenausbreitungen sowie die Visualisierung dynamischer Graphen mit Hilfe kräftebasierter Verfahren. Details zu den Aufgaben und Zielen der Projektgruppe werden im folgenden Abschnitt gegeben.

Die Teilnehmer der Projektgruppe sind in alphabetischer Reihenfolge:

Till	Barz
Thorsten	Deinert
Danny	van Dyk
Markus	Geveler
David	Gies
Nina	Harmuth
Volker	Jung
Mathias	Kadolsky
Sven	Mallach
André	Matuschek
Joachim	Messer
Dirk	Ribbrock

1.2. Aufgabenstellung und Ziele der Projektgruppe

Im Mittelpunkt der Projektgruppenarbeit stand die Frage, welcher Anteil der theoretisch möglichen 230 GFlop/s Rechenleistung der Cell BE in realen Anwendungen überhaupt abgerufen werden kann. Das erste Ziel war es dabei, zunächst ein gutes Verständnis für die Cell-Architektur aufzubauen, um dann existierende Algorithmen, die für das klassische Programmierparadigma bereits gut verstanden sind, auf das neue Modell zu

[1] Im folgenden wird statt dessen auch kurz *Cell BE* oder *Cell*(-Prozessor) geschrieben.

übertragen. Die Anwendungsfälle waren beide so gewählt, dass sich die laufzeitkritischen Komponenten in eine Cell-Bibliothek (die sogenannte *Basisbibliothek*) auslagern lassen, während weniger kritische Komponenten klassisch auf einer normalen Power-CPU implementiert werden konnten. Der Schwerpunkt lag also auf der Grundlagenarbeit, und die Anwendungen sollten den Erfolg und die Übertragbarkeit in die Praxis demonstrieren.

Ein Beispiel für die notwendige algorithmische Grundlagenarbeit sind *Matrix-Vektor-Operationen* (mit nichttrivialen Datenabhängigkeiten), die häufig in numerischen Simulationen auftreten. Die Größe der beteiligten Vektoren überschreitet hier die Speicherkapazität der SPEs um mehrere Größenordnungen. Klassische Implementierungen, bei denen in einer Schleife linear alle Werte mittels Speicherindirektionen berechnet werden, eignen sich also nicht für den Cell-Prozessor. Vielmehr war es nötig, geschickte Parallelisierungs- und Blockungsstrategien anzuwenden und so einen komplett neuen Algorithmus für das Problem zu entwickeln.

Erste Aufgabenstellung: Simulation von Wellenausbreitung

Das erste Ziel, das als Aufgabenstellung gewählt wurde, kommt aus dem Gebiet der numerischen Strömungssimulation. Die Aufgabe bestand darin, die physikalisch korrekte Ausbreitung von Wellen zu simulieren.

Der grundlegende Ansatz dieser Aufgabenstellung kann durch ein System *partieller Differentialgleichungen* – die sogenannten Flachwassergleichungen – beschrieben werden; dieser Schritt wird als *Modellierung* bezeichnet.

Im folgenden Schritt, der *Diskretisierung*, wird ein (reguläres) Gitter über das Simulationsgebiet gelegt, und man ist nicht mehr an den kontinuierlichen Größen Druck, Geschwindigkeit und Wasserhöhe interessiert, sondern beschränkt sich auf die endlich vielen Berechnungspunkte im Gitter. Wendet man mathematische Methoden wie *Finite Differenzen* oder *Finite Elemente* bzw. *Finite Volumen* an, so ergibt sich am Ende ein lineares Gleichungssystem, das die unbekannten Größen in einem Gitterpunkt mit denen in den anderen Gitterpunkten koppelt.

Eine entsprechende Diskretisierung wird auf der Zeitachse durchgeführt, so dass am Ende ein Gleichungssystem pro Schritt gelöst werden muss. Natürlich ist die berechnete Lösung um so genauer, je feiner die Diskretisierung ist. In praktischen Fällen haben daher diese Gleichungssysteme häufig viele Millionen Unbekannte pro Zeitschritt. Bei einer Zeitschrittweite, die deutlich unter einer Sekunde liegt, ergibt sich daher ein immenser Rechenaufwand.

Als algorithmischer Ansatz für dieses numerisch anspruchsvolle Problem wurde ein sogenanntes Relaxationsverfahren gewählt, dessen clevere Implementierung von Grund auf analysiert und neu entworfen wurde.

Zweite Aufgabenstellung: Visualisierung von dynamischen Graphen

Die zweite Anwendung behandelt die Visualisierung von Graphen, die sich über die Zeit verändern, d.h. Knoten und Kanten kommen nach und nach hinzu oder verschwinden.

Beispiele hierzu finden sich bei der Modellierung von Software-Systemen, der Darstellung sozialer Netzwerke oder in der Kriminalistik.

Das konkrete Ziel der Projektgruppenarbeit war es, für jeden (diskreten) Zeitpunkt ein *Layout des Graphen* in der Ebene zu bestimmen, so dass sich aus den einzelnen Darstellungen der Layouts in ihrer zeitlichen Entwicklung insgesamt eine flüssige Animationssequenz ergibt. Knoten, die über einen Zeitschritt hinweg erhalten bleiben, sind dabei mit ihren Vorgänger- bzw. Nachfolge-Instanzen durch spezielle Kanten verbunden.

Die Projektgruppenteilnehmer haben als Ergebnis eine Graphenbibliothek entwickelt, die Layouts mit Hilfe *kräftebasierter Layoutverfahren* berechnet. Diese Verfahren simulieren ein physikalisches Kräftemodell, bei dem sich adjazente Knoten anziehen und nicht-adjazente Knoten voneinander abstoßen. Auch diese Bibliothek stützt sich wie der Löser für die Flachwassergleichungen auf die Funktionalität der Basisbibliothek.

2. Ergebnisse der Entwurfsphase

In der Seminarphase der Projektgruppe wurden die theoretischen Grundlagen für das Verständnis der Cell BE-Hardware sowie der beiden anwendungsorientierten Aufgabenstellungen erarbeitet. Die sich daran anschließende Entwurfsphase wurde durch eine Aufteilung in Arbeitsgruppen strukturiert, in denen Lösungsansätze für die Erstellung der verschiedenen Bibliotheks-Komponenten analysiert und diskutiert wurden.

Ausgangspunkt für diese Arbeit war eine gründliche Recherche, in der die Literatur der Seminarphase noch einmal im Hinblick auf die Qualität und Umsetzbarkeit der dort dargestellten Lösungsansätze untersucht und durch weitere Quellen ergänzt wurde. Die Entwurfstätigkeit für die praktischen Anwendungen, die sich beide auf die Cell-optimierte Basisbibliothek stützen sollten, läßt sich im wesentlichen dadurch charakterisieren, dass sie sich sehr verschieden schwierig gestaltete. Während die Aufgabenstellung des automatisierten Zeichnens von dynamischen Graphen auf gut bekannte und weitgehend ausgearbeitete Verfahren zurückgreifen konnte, mussten für das Problem des Lösens der 2D-Flachwassergleichungen neue wissenschaftliche Ansätze gefunden und auf ihre Eignung geprüft werden. Es hatte sich dabei herausgestellt, dass die in der Seminarphase vorstellten Verfahren, nämlich sowohl das in der Diplomarbeit von Becker [Bec06] vorgestellte semi-implizite Verfahren als auch das explizite Verfahren, das in der Arbeit von Hagen und Henriksen [HHHL06] dargelegt wird, nicht den Qualitätsansprüchen der Teilnehmer an Genauigkeit, Flexibilität bzgl. der Simulation von verschiedenen Szenarien und der Geschwindigkeit für ein gesuchtes Lösungsverfahren genügte.

Bevor in diesem Kapitel die Ergebnisse der Enwurfsphase für die algorithmischen Grundlagen der beiden Anwendungsprobleme präsentiert werden, sollen zunächst die architektonischen Besonderheiten der Cell-Architektur vorgestellt werden, die im Hinblick auf die praktischen Möglichkeiten und Grenzen der Cell-Programmierung die weiteren Entscheidungen über die Implementierung der Bibliothekskomponenten wesentlich mitbestimmt haben. Als weiterführende Lektüre sei an dieser Stelle auf die Dokumentation des *IBM Cell Broadband Engine Software Development Kits* [IBM05] und der einführenden Darstellung von James Kahle [KDH+05] verwiesen.

2.1. Der Cell Prozessor

Die *Cell-Broadband Engine* wurde im Jahr 2001 vom STI Design Center, einer Kooperation der Unternehmen Sony, Toshiba und IBM, angekündigt [KDH+05] und findet sich inzwischen nicht nur in der aktuellen Spielekonsole *PlayStation 3* von Sony wieder, sondern wird aufgrund seiner außergewöhnlichen Architektur und theoretischer Leistungsfähigkeit auch in der Welt des wissenschaftlichen Rechnens intensiv diskutiert oder bereits praktisch eingesetzt.

Im Gegensatz zu herkömmlichen Mehrkern-Prozessoren besteht der Cell-Prozessor nicht aus mehreren gleichartigen Kernen, sondern aus einer zweikernigen 64bit-*PowerPC*-CPU, dem sogenannten *Power Processor Element* (PPE), und bis zu 8 sehr leistungsfähigen Vektorprozessoren, den so genannten *Synergistic Processing Elements* (SPE). Letztere sind nicht zu Steuerungsaufgaben, wie z.B. der Ausführung eines Betriebssystems fähig, aber insbesondere für Fließkommaberechnungen hocheffizient. Abbildung 2.1 zeigt den schematischen Aufbau des Cell-Prozessors.

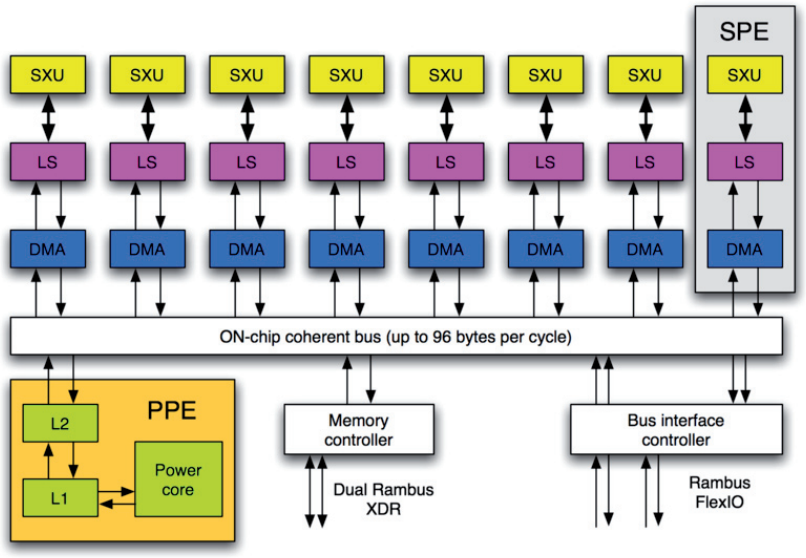


Abbildung 2.1.: Schema des Cell-Prozessors (Quelle: Wikipedia)

In konventionellen Mehrkern-Prozessoren haben alle Kerne Zugriff auf schnellen Cache-Speicher. Dabei unterscheiden sich die Latenzen und Bandbreiten der verschiedenen Ebenen in der Speicherhierarchie jeweils um mehrere Zehnerpotenzen voneinander. Dagegen verfügt jede SPE im Cell-Prozessor nur über sehr wenig lokalen Speicher (genannt *Local Store (LS)*) und kann nicht direkt auf den Hauptspeicher zugreifen. Die Kommunikation zwischen dem Hauptspeicher und den lokalen Speichern der SPEs über einen gemeinsamen Speicherbus muss vom Programmierer explizit vorgegeben werden. Dies bedeutet einerseits einen sehr hohen Freiheitsgrad des Programmierers, andererseits starke Einschränkungen auf Grund des verhältnismäßig kleinen direkt adressierbaren Speichers.

Durch die 8 SPEs summiert sich die maximale (theoretische) Fließkommaleistung des Cell-Prozessors auf 230 GFlop/s [1] bei einfacher Genauigkeit. Im Vergleich dazu erreichen

[1] 1 GFlop/s = 10^9 Flop/s (*Floating point operations per second*), bezeichnet 1 Milliarde Fließkommaoperationen pro Sekunde. Flop/s ist die Maßeinheit für die Geschwindigkeit von wissenschaftlich genutzten Computersystemen oder Prozessoren und bezeichnet die Anzahl der elementaren Gleitkommazahl-Operationen (Additionen oder Multiplikationen), die von ihnen pro Sekunde ausgeführt werden können.

die zweikernigen Vertreter der Intel *Core 2 Duo* Prozessorfamilie weniger als 20 GFlop/s, das aktuelle Topmodell mit 4 Kernen bis zu 12 MB L2 Cache und Codenamen Penryn maximal theoretische 96 GFlop/s.

Die in Kapitel 4 vorgestellten Benchmarks versuchen eine Antwort auf die Frage zu geben, welcher Anteil der 230 GFlop/s in realen Anwendungen tatsächlich erreicht werden kann. Interessant ist insbesondere, ob sich die Optimierung von numerischen Basisoperationen lohnt, aus denen dann komplexe Anwendungsoperationen zusammen gebaut werden können, oder ob die Optimierung für spezielle Anwendungsfälle selbst die bessere Alternative ist.

Die folgende tiefergehende Beschreibung der wesentlichen Komponenten des Cell-Prozessors nimmt Bezug auf die Darstellung von Kahle [KDH+05], in der auch weitere Details nachzulesen sind.

2.1.1. Das Power Processor Element

Das *Power Processor Element* (kurz: PPE) verfügt über einen 32 kB L1-Instruktions- und Daten-Cache, sowie eine Taktfrequenz von 3,2 GHz. 2 Threads werden nach dem *Round-Robin Prinzip* zur gleichen Zeit verarbeitet (*dual-threaded*) und bis zu 2 Instruktionen können sich gleichzeitig in der Ausführungs-Phase befinden (*dual-issued*), solange sie nicht die gleiche Ausführungseinheit belegen.

Das Hauptmerkmal des Designs ist eine reduzierte Komplexität um eine hohe Taktrate und eine geringe Verlustleistung zu gewährleisten. So wurde zum Beispiel auf *Out-of-Order* Ausführung und eine komplexe Sprung-Vorhersage (*Branch-Prediction*) verzichtet. Die SIMD-Erweiterungen (*Altivec*) der CPU sind auf einfache Genauigkeit beschränkt.

Speichertransfers per DMA

Wie alle Recheneinheiten des Cell-Prozessors verfügt das PPE über einen Speichercontroller namens *Memory Flow Controller (MFC)*. Über ihn werden alle Transfers per DMA (*Direct Memory Access*) oder die im Verhältnis dazu langsameren MMIO (*Memory-Mapped-I/O*)-Register abgewickelt. Dabei ist es sowohl möglich, dass das PPE Daten zu den SPEs sendet, als auch, dass die SPEs ihrerseits mit ihren *Memory Flow Controllern* Daten vom Hauptspeicher des PPE anfordern. DMA-Speichertransfers sind grundsätzlich an 16-Byte Grenzen ausgerichtet und haben eine maximale Größe von 16 kB. Sie dienen insbesondere für die Übertragung von kontinuierlichen Datenbereichen aus dem Hauptspeicher in die lokalen Speicher der SPEs.

Sollen nicht-kontinuierliche Bereiche übertragen werden, bietet die Cell-Architektur das Konzept der Listentransfers. In diesem Fall werden bis zu 2048 DMA-Transfers mit jeweils 16 kB Größe über ein vorgegebenes Schema zu einer DMA-Liste zusammengefasst. Alle Adressen einer DMA-Liste müssen dabei in den höherwertigen 32 Bit übereinstimmen. So sind theoretisch mit einer solchen Liste bis zu 32 MB Übertragung möglich, was der 128-fachen Größe des *Local Stores* entspricht. SPEs können ihrerseits solche DMA-Listen im *Local Store* anlegen, typischerweise werden die Listen aber vom PPE

(auf Grund der Kenntnis von Hauptspeicheradressen) vorbereitet und dann entweder zum SPE übertragen, oder vom SPE vor dem eigentlichen DMA-Listentransfer per herkömmlichem DMA-Transfer geladen.

Eine andere Möglichkeit zur Kommunikation zwischen dem PPE und den SPEs ist das Mailbox-System, welches ebenfalls über den MFC und die MMIO-Register abgewickelt wird. Jede Recheneinheit besitzt einen Maileingang und -ausgang. Mit diesem System ist es möglich 32-bit Nachrichten auszutauschen, die auch unterbrechenden Charakter haben können (*Interrupt-Mails*).

Im Rahmen der Projektgruppe wurde das PPE in erster Linie als Verwaltungs- und Steuereinheit für die SPEs verwendet. Eine detaillierte Beschreibung dieser Aufgaben folgt in Abschnitt 3.1.5.

2.1.2. Die Synergistic Processing Elements

Die SPEs verfügen über 128 128-bit Vektorregister und einen 256 kB großen *Local Store* anstelle eines Caches. Die Ausführung und Zugriffszeiten sind dadurch vollständig vorhersagbar und nicht von Trefferraten im Cache abhängig. Die Anzahl der vorhandenen Register erlaubt eine hohe Ausnutzung der SIMD-Vektoreinheit für große Probleme. Dies ist vor allem für die Anwendungsoptimierung selbst interessant, während bei grundlegenden Operationen wie Additionen oder Multiplikationen eher wenige Register ausgenutzt werden können.

Auch die SPEs nutzen *dual issue*: Es können Berechnungen und verschiedene Speicheroperationen gleichzeitig durchgeführt werden. Ebenso wurde wie beim PPE auf *In-Order-Execution* gesetzt und Sprünge werden nicht in Hardware vorhergesagt. Mit Hilfe von *Branch Hints* kann der Programmierer oder Compiler hohe Kosten für *branch misses* verhindern. In so einem Fall werden 17 Instruktionen ab der Sprungadresse vorgehalten.

Da die SPEs in der Lage sind ein 3-Operanden-*multiply add* in einem Befehl auszuführen, also Multiplikation zweier Operanden inklusive Addition des Ergebnisses auf einen dritten Operanden, können vektorisiert bis zu vier solche *multiply-add*-Instruktionen pro SPE gleichzeitig ausgeführt werden. Insgesamt handelt es sich dabei also um acht Operationen, wodurch bei 3,2 GHz theoretisch bis zu 25,6 GFlop/s erreicht werden. Aufsummiert für 8 SPEs bedeutet dies eine maximale theoretische Performanz von 204,8 GFlop/s. Dieser Wert wurde bei Tests von IBM auch annähernd erreicht.

Der *Local Store* erfüllt ähnlich wie ein Cache die Aufgaben eines sehr schnellen Zwischenspeichers zwischen RAM und Prozessor. Er dient sowohl für den schnellen Zugriff auf per DMA-Transfer geholte Daten, wie auch als Ablageort für Zwischenergebnisse. Zum Ende einer typischen Berechnung werden die Ergebnisse dann erneut über DMA-Transfers in den Hauptspeicher des PPE zurückgeschrieben.

Wie bereits erwähnt besitzt jedes SPE dazu einen eigenen Speicher-Controller, der die Zugriffe von außen auf den *Local Store* sowie auch umgekehrt steuert, also zum Beispiel Datenanfragen an den Hauptspeicher oder andere *Local Stores*.

2.1.3. Bus und Speicheranbindung des Cell

Der sogenannte *Element Interconnect Bus* (kurz: EIB) verbindet die einzelnen Elemente des Cell Prozessors (PPE, 8 SPEs, E/A-Geräte) miteinander. Es handelt sich dabei um 4 unidirektionale 16-Byte-Kanäle, die paarweise in entgegengesetzte Richtungen verlaufen. Die Daten werden pro Bustakt (halber Prozessor-Takt) einen Schritt weitergegeben, wodurch bis zu 3 Übertragungen gleichzeitig je Kanal durchgeführt werden können. Somit sind bis zu 12 parallele Übertragungen und sehr hohe Datenraten möglich, was erforderlich ist, um die SPEs kontinuierlich mit Daten zu versorgen. Von IBM wurden über 200 GB/s Datenrate bei 1,6 GHz Bustakt erreicht.

Auch die Anbindung an den Hauptspeicher muss genug Bandbreite bieten, um die 9 Prozessoreinheiten mit Daten zu versorgen. Dafür sorgt das *XIO Interface von Rambus*, das im Dual-Channel-Betrieb Datenraten von bis zu 25,6 GB/s erreicht. Im Vergleich dazu erreicht zum Beispiel *DDR2-800 RAM* im Dual-Channel-Betrieb 12,8 GB/s. Die Kommunikation mit anderen Hardwarekomponenten geschieht über das *FlexIO-Interface* der Firma Rambus. Insgesamt sind bei 3,2 GHz Datenraten von 44,8 GB/s ausgehend und 32 GB/s eingehend möglich. Neben Verbindungen zur *North-* und *Southbridge* können so auch 2 Cell-Prozessoren direkt miteinander verbunden werden.

2.2. Flachwassersimulation

2.2.1. Einführung

Die Flachwassergleichungen[2] (*Shallow Water Equations*) sind ein System von physikalischen Gleichungen, die, unter Berücksichtigung der Erhaltung von Masse, Energie und Impuls, ein Modell liefern, um die Oberfläche von *inkompressiblen* Flüssigkeiten (*Fluiden*) zu berechnen. Dabei ist die (Wasser-) Oberfläche gegeben als Höhenwert, was dementsprechend auch für das Bodenprofil gilt, wie in Abbildung 2.2 dargestellt wird. Die Flachwassergleichungen sind eine Vereinfachung der *Navier Stokes* Gleichungen, die entsteht, wenn man bei letzteren die Viskosität des Mediums und die Geschwindigkeit in z-Richtung außer Acht läßt. Für eine detailierte Beschreibung der Navier Stokes Gleichungen und für Betrachtungen zu Erhaltungsgleichungen im Allgemeinen sei auf die Arbeit von Becker [Bec06] verwiesen.

Eine Lösung für die Flachwassergleichungen besteht darin, eine Funktion (in diesem Fall ein Höhenfeld) zu finden, die dem Gleichungssystem genügt. Dazu ist eine räumliche *Diskretisierung* nötig, wodurch das kontinuierliche Rechengebiet mit einem Gitter überzogen wird. Hierbei gibt es verschiedene Ansätze, wobei sich hier auf den im unten beschriebenen Verfahren verwendeten Finite Volumen (FV) Ansatz beschränkt wird. Anschaulich korrespondiert der Mittelpunkt einer *Gitterzelle* dabei mit einem Höhen- oder Geschwindigkeitswert, wie in Abbildung 2.3 gezeigt. Die Zeit ist eine weitere kontinuierliche Größe und daher muss auch sie diskretisiert werden, damit das Problem mit endlichen Ressourcen und in endlicher Zeit numerisch gelöst werden kann. Für die zeitliche Diskretisierung gibt es wiederum einige Verfahren, von denen im folgenden auch

[2] Im folgenden sind stets die zweidimensionalen Flachwassergleichungen gemeint.

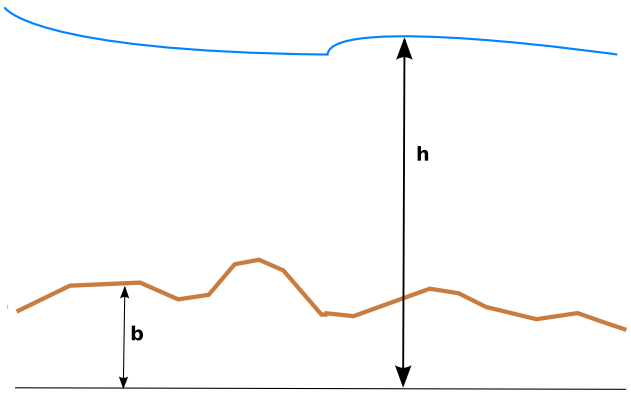


Abbildung 2.2.: Höhenfelder der Flachwassersimulation

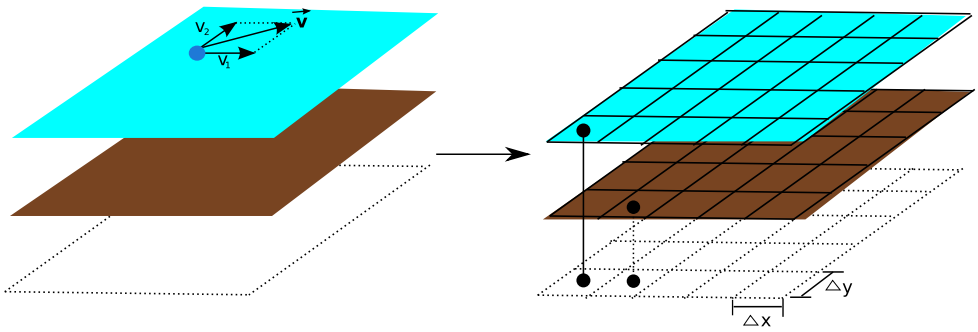


Abbildung 2.3.: Finite Volumen Diskretisierung

lediglich dasjenige genauer betrachtet werden soll, das auch implementiert wurde. Dieses ist stark mit dem Gesamtverfahren verknüpft, das im nächsten Abschnitt beschrieben wird. Zusammenfassend und stark vereinfacht besteht der Prozess der Diskretisierung damit insgesamt aus der Etablierung von Schrittweiten. Diese sind die Gitterschrittweiten bei der räumlichen und die Zeitschrittweite bei der zeitlichen Diskretisierung.

Das physikalische Modell, welches allen weiteren Betrachtungen zugrunde liegt, besteht aus den folgenden Komponenten.

- Schrittweiten: Die Gitterschrittweiten in x- und y-Richtung, sowie die Zeitschrittweite.
- Höhenfelder für die Wasserhöhe und das Bodenprofil
- Geschwindigkeiten für jeden Gitterpunkt: Der Geschwindigkeitsvektor besteht aus je einer Komponente für die x und die y Richtung (siehe auch Abbildung 2.3 links).

Bei numerischen Lösungsverfahren für die Flachwassergleichungen unterscheidet man generell zwischen impliziten und expliziten Verfahren. Der Unterschied läßt sich dadurch charakterisieren, dass bei expliziten Verfahren eine Lösung ausschließlich aus Koeffizienten des aktuellen Zeitschrittes berechenbar ist, während bei impliziten Verfahren zusätzlich Beziehungen zwischen den Werten des aktuellen Zeitschrittes eine Rolle spielen, was dazu führt, dass auf unterster Ebene ein lineares Gleichungssystem gelöst werden muss.

2.2.2. Relaxationsverfahren für das Lösen der Flachwassergleichungen

Die Arbeitsgruppe zum Themengebiet Flachwassergleichungen hat sich mit den Schwächen und Stärken verschiedener Verfahren zum Lösen dieser speziellen physikalischen Erhaltungsgleichungen auseinandergesetzt.

Begonnen wurde mit einer eingehenden Literaturrecherche und der mathematischen Untersuchung alternativer Lösungsansätze. Bereits in den Vorträgen der Seminarphase hatte sich angedeutet, dass die dort vorgestellten beiden Ansätze [Lv02, HHHL06] den Ansprüchen des Projektgruppenziels für dieses Anwendungsbeispiel nicht genügen würden. Das gesuchte alternative Verfahren sollte Modell-Szenarien wie zum Beispiel völlig trockenen Boden (*dry-states*) und eine beruhigte Wasseroberfläche (*steady-states*) simulieren können und dadurch ein visuell ansprechendes Ergebnis liefern. Damit wird der Anwender in die Lage versetzt, auch Flüsse über eine unebene Oberfläche, wie beispielsweise die Wasseroberfläche eines Sees oder Ozeans in einer Küstenregion zu simulieren. Als Ergebnis der Recherchephase der Flachwasser-Gruppe wurde zu Gunsten eines dritten Lösungsansatzes entschieden, der als aussichtsreich bewertet wurde, möglichst viele interessante Probleminstanzen für die Simulation von Wellenbewegungen abzudecken und dabei hinreichend genau und schnell Ergebnisse zu liefern. Bei diesem gewählten Lösungsansatz handelt es sich um ein sogenanntes Relaxationsverfahren, das von Delis und Katsaounis im Detail beschrieben wird [DK05].

Delis und Katsaounis kombinieren ein Relaxationsschema mit dem *Runge-Kutta-Mechanismus* zur zeitlichen Entwicklung, d.h. dem Addieren der Änderung eines Wasservolumens $Q_{i,j}$ vom letzten zum nächsten Zeitschritt. Die Grundidee des Relaxationssche-

mas besteht, grob gesagt, darin, dass vor der eigentlichen Diskretisierung das System der Erhaltungsgleichungen durch ein anderes ersetzt wird, welches das ursprüngliche System approximiert. Auf dieses neue System wird dann die räumliche und zeitliche Diskretisierung angewandt. Mit der Relaxation werden zusätzliche (Relaxations-)Vektoren und eine Relaxationsvariable ϵ eingeführt. Bei dem Runge-Kutta-Mechanismus handelt sich um ein sogenanntes Vorhersage-Korrektur (*predictor-corrector*) Schema.

Mit Blick auf die in der Arbeit von Delis and Katsaounis belegten numerischen Tests und Ergebnisse, wurde die Entscheidung für die Implementierung des Verfahrens von der Einschätzung getragen, dass das Relaxationsverfahren ein hinreichend schnelles und gutes Lösungsverfahren darstellt, das den beiden anderen Verfahren, die im Seminar vorgestellt wurden, insofern überlegen ist, als dass die Berechnungskraft größer ist, d.h. es kann auch die gewünschten Szenarien mit trockenem und unebenem Untergrund stabil bewältigen. Inwieweit diese Einschätzung korrekt war, wird in Abschnitt 4.2 analysiert.

2.2.3. Das explizite Verfahren nach Delis und Katsaounis

Die Erhaltungsgleichungen können (ohne Berücksichtigung der Viskosität des Mediums und ohne Berücksichtigung von Verwirbelungen sowie Einflüssen durch Wind und Coriolis-Kräfte) als eine einzelne Vektorgleichung wie folgt formuliert werden:

$$U_t + F(U)_x + G(U)_y = S(U) \text{ mit } (x, y) \in \Omega, t \geq 0, \tag{2.1}$$

wobei

$$U = \begin{pmatrix} h \\ hu_1 \\ hu_2 \end{pmatrix} = \begin{pmatrix} h \\ q_1 \\ q_2 \end{pmatrix},$$

$$S(U) = \begin{pmatrix} 0 \\ -gh\frac{\partial Z}{\partial x}(x,y) - ghS_f^x \\ -gh\frac{\partial Z}{\partial y}(x,y) - ghS_f^y \end{pmatrix},$$

$$F(U) = \begin{pmatrix} q_1 \\ \frac{q_1^2}{h} + \frac{1}{2}gh^2 \\ \frac{q_1 q_2}{h} \end{pmatrix},$$

$$G(U) = \begin{pmatrix} q_2 \\ \frac{q_1 q_2}{h} \\ \frac{q_2^2}{h} + \frac{1}{2}gh^2 \end{pmatrix}.$$

Dieses System beschreibt den Fluss zum Zeitpunkt t in einem Punkt $(x, y) \in \Omega$. Dabei ist $h(x, y, t)$ die Wasserhöhe in Punkt (x, y) zum Zeitpunkt t, Ω bezeichnet das Gebiet der mit Wasser bedeckten x-y-Ebene, und $Z(x, y)$ ist eine Funktion, die die Bodentopographie in die Rechnung einbezieht. (u_1, u_2) sind die Komponenten der durchschnittlichen Geschwindigkeit des Fluids, g die Gravitationskonstante. Die Variable $\mathbf{q}$ stellt die zu erhaltene Gesamtenergie dar und ist gegeben durch $(q_1, q_2) = (hu_1, hu_2)$. Die Flussterme $\mathbf{G}$ und $\mathbf{F}$ repräsentieren anschaulich den gesamten Fluss über eine Zelle bezüglich der

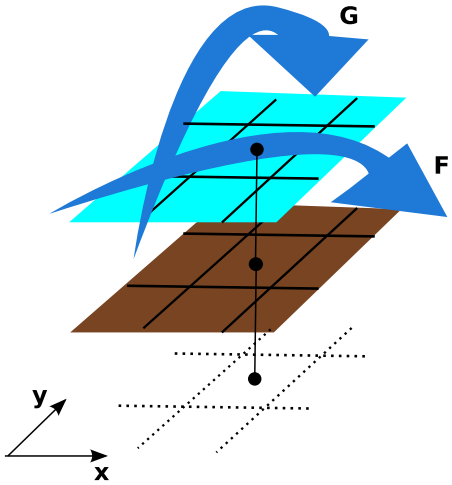


Abbildung 2.4.: Der Fluss über ein Zellenvolumen

Richtungen x und y im aktuellen Zeitschritt. Quellterme sind notwendig, um Gefälle und Reibung bei unebenem Boden in Flachwassersimulationen zu internalisieren. Als Quellterme werden die sogenannten Bodenreibungssterme mit $-\partial Z/\partial x$ sowie $-\partial Z/\partial y$ (für die x-Richtung bzw. die y-Richtung) von den sogenannten Reibungstermen (*friction-slopes*) S_f^x bzw. S_f^y unterschieden. Für die Modellierung der zu simulierenden Szenarien können die Reibungsterme als materialabhängig auf Grund empirischer Zusammenhänge durch die folgende *Manningdarstellung* abgeschätzt werden mit

$$S_f^x = n_m^2 u_1 h^{-4/3} \sqrt{u_1^2 + u_2^2} \tag{2.2}$$

und

$$S_f^y = n_m^2 u_2 h^{-4/3} \sqrt{u_1^2 + u_2^2} \tag{2.3}$$

wobei n_m den *Manning-Koeffizienten* bezeichnet. Einige empirische Werte für den Manning-Koeffizienten werden in der folgenden Tabelle aufgeführt.

Material	Größe (mm)	n_m
Sand	0.2	0.012
Grober Sand	1 - 2	0.026 - 0.035
Kies	2 - 64	0.028 - 0.035
Geröll	64 - 256	0.038 - 0.050
Brocken	> 256	0.050 - 0.070

Das folgende Relaxationsschema ersetzt die obige Erhaltungsgleichung 2.1 mit Hilfe der Relaxationsvektoren v, w:

$$\begin{aligned}
h_t + v_{1x} + w_{1y} &= 0, \\
q_{1t} + v_{2x} + w_{2y} &= -ghZ_x, \\
q_{2t} + v_{3x} + w_{3y} &= -ghZ_y, \\
v_{1t} + c_1^2 h_x &= -\frac{1}{\epsilon}(v_1 - q_1) \\
v_{2t} + c_2^2 q_{1x} &= -\frac{1}{\epsilon}\left(v_2 - \left(\frac{q_1^2}{h} + \frac{g}{2}h^2\right)\right) \\
v_{3t} + c_3^2 q_{2x} &= -\frac{1}{\epsilon}\left(v_3 - \left(\frac{q_1 q_2}{h}\right)\right) \\
w_{1t} + d_1^2 h_y &= -\frac{1}{\epsilon}(w_1 - q_2) \\
w_{2t} + d_2^2 q_{1y} &= -\frac{1}{\epsilon}\left(w_2 - \left(\frac{q_1 q_2}{h}\right)\right) \\
w_{3t} + d_3^2 q_{2y} &= -\frac{1}{\epsilon}\left(w_3 - \left(\frac{q_2^2}{h} + \frac{g}{2}h^2\right)\right).
\end{aligned} \tag{2.4}$$

Setzt man

$$\mathbf{u} = \begin{bmatrix} h \\ q_1 \\ q_2 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix},$$

so läßt sich das System als Vektorgleichung umschreiben zu:

$$\begin{aligned}
\mathbf{u}_t + \mathbf{v}_x + \mathbf{w}_y &= \mathbf{S}(\mathbf{u}), \\
\mathbf{v}_t + C^2 u_x &= -\frac{1}{\epsilon}\mathbf{v} - \mathbf{F}(\mathbf{u}), \\
\mathbf{w}_t + D^2 u_y &= -\frac{1}{\epsilon}\mathbf{w} - \mathbf{G}(\mathbf{u}),
\end{aligned} \tag{2.5}$$

wobei $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^3$ und $C^2, D^2 \in \mathbb{R}^{3\times 3}$ konstante Diagonalmatrizen mit positiven Eigenwerten sind. Die Relaxation wird durch die Relaxationsrate (den Relaxationsparameter) $0 < \epsilon < 1$ bestimmt. Für $\epsilon \to 0$ wird das ursprüngliche System angenähert.

Die Diskretisierung des Schemas erfolgt zunächst in räumlicher und danach in zeitlicher Hinsicht. Für das räumliche (semidiskrete) Relaxationsschema wird ein Schema zweiter Ordnung, MUSCL (*Monotone Upstream-centered Scheme for Conservation Laws*), verwendet. Dieses ersetzt die stückweise konstante Approximation durch eine stückweise lineare Interpolation, um die Genauigkeit zweiter Ordnung zu erreichen. Die zeitliche Diskretisierung zu einem volldiskretem Schema erfolgt durch Anwendung einer impliziten Runge-Kutte-Methode, das die Lösung in Zeitschritten von Δt fortführt.

Das Relaxationsschema berechnet die Werte von $\mathbf{u}_{n+1}$, $\mathbf{v}_{n+1}$ und $\mathbf{w}_{n+1}$ für einen Zeitschritt mit Hilfe von Vorhersage- und Korrekturschritten aus den alten Werten $\mathbf{u}_n$, $\mathbf{v}_n$

und $\mathbf{w}_n$:

$$\begin{aligned}
\mathbf{u}^{n,1} &= \mathbf{u}^{\mathbf{n}} \\
\mathbf{v}^{n,1} &= \mathbf{v}^n + \frac{\Delta t}{\epsilon}(\mathbf{v}^{n,1} - F(\mathbf{u}^{n,1})) \qquad (2.6)\\
\mathbf{w}^{n,1} &= \mathbf{w}^n + \frac{\Delta t}{\epsilon}(\mathbf{w}^{n,1} - G(\mathbf{u}^{n,1})) \\
\mathbf{u}^{(1)} &= \mathbf{u}^{n,1} - \Delta t(\Delta_+^x \mathbf{v}^{n,1} + \Delta_+^y \mathbf{w}^{n,1}) + \Delta t \mathbf{S}(\mathbf{u}^{n,1}) \\
\mathbf{v}^{(1)} &= \mathbf{v}^{n,1} - \Delta t C^2 \Delta_+^x \mathbf{u}^{n,1} \qquad (2.7)\\
\mathbf{w}^{(1)} &= \mathbf{w}^{n,1} - \Delta t D^2 \Delta_+^y \mathbf{u}^{n,1} \\
\mathbf{u}^{n,2} &= \mathbf{u}^{(1)} \\
\mathbf{v}^{n,2} &= \mathbf{v}^{(1)} - \frac{\Delta t}{\epsilon}(\mathbf{v}^{n,2} - F(\mathbf{u}^{n,2})) - \frac{2\Delta t}{\epsilon}(\mathbf{v}^{n,1} - F(\mathbf{u}^{n,1})) \qquad (2.8)\\
\mathbf{w}^{n,2} &= \mathbf{w}^{(1)} - \frac{\Delta t}{\epsilon}(\mathbf{w}^{n,2} - G(\mathbf{u}^{n,2})) - \frac{2\Delta t}{\epsilon}(\mathbf{w}^{n,1} - G(\mathbf{u}^{n,1})) \\
\mathbf{u}^{(2)} &= \mathbf{u}^{n,2} - \Delta t(\Delta_+^x \mathbf{v}^{n,2} + \Delta_+^y \mathbf{w}^{n,2}) + \Delta t \mathbf{S}(\mathbf{u}^{n,2}) \\
\mathbf{v}^{(2)} &= \mathbf{v}^{n,2} - \Delta t C^2 \Delta_+^x \mathbf{u}^{n,2} \qquad (2.9)\\
\mathbf{w}^{(2)} &= \mathbf{w}^{n,2} - \Delta t D^2 \Delta_+^y \mathbf{u}^{n,2} \\
\mathbf{u}^{n+1} &= \frac{1}{2}(\mathbf{u}^n + u^{(2)}) \\
\mathbf{v}^{n+1} &= \frac{1}{2}(\mathbf{v}^n + v^{(2)}) \qquad (2.10)\\
\mathbf{w}^{n+1} &= \frac{1}{2}(\mathbf{w}^n + w^{(2)})
\end{aligned}$$

Hierbei bezeichnet

$$\Delta_+^x \mathbf{p}_{ij} = \frac{1}{\Delta x}(\mathbf{p}_{i+\frac{1}{2},j} - \mathbf{p}_{i-\frac{1}{2},j})$$

$$\Delta_+^y \mathbf{p}_{ij} = \frac{1}{\Delta y}(\mathbf{p}_{i,j+\frac{1}{2}} - \mathbf{p}_{i,j-\frac{1}{2}}).$$

Für $\mathbf{p} \in \{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ gilt

$$\mathbf{u}_{\mathbf{i+\frac{1}{2},j}} = \frac{1}{2}(\mathbf{u}_{ij} + \mathbf{u}_{i+1,j}) - \frac{1}{2c_k}(\mathbf{v}_{i+1,j} - \mathbf{v}_{ij}) + \frac{\Delta x}{4c_k}\left(\mathbf{s}_{ij}^{x,+} - \mathbf{s}_{i+1,j}^{x,-}\right)$$

$$\mathbf{v}_{i+\frac{1}{2},j} = \frac{1}{2}(\mathbf{v}_{ij} + \mathbf{v}_{i+1,j}) - \frac{c_k}{2}(\mathbf{u}_{i+1,j} - \mathbf{u}_{ij}) + \frac{\Delta x}{4}\left(\mathbf{s}_{ij}^{x,+} - \mathbf{s}_{i+1,j}^{x,-}\right)$$

$$\mathbf{u}_{i,j+\frac{1}{2}} = \frac{1}{2}(\mathbf{u}_{ij} + \mathbf{u}_{i,j+1}) - \frac{1}{2d_k}(\mathbf{w}_{i,j+1} - \mathbf{w}_{ij}) + \frac{\Delta y}{4d_k}\left(\mathbf{s}_{ij}^{y,+} - \mathbf{s}_{i,j+1}^{y,-}\right)$$

$$\mathbf{w}_{i,j+\frac{1}{2}} = \frac{1}{2}(\mathbf{w}_{ij} + \mathbf{w}_{i,j+1}) - \frac{d_k}{2}(\mathbf{u}_{i,j+1} - \mathbf{u}_{ij}) + \frac{\Delta y}{4}\left(\mathbf{s}_{ij}^{y,+} - \mathbf{s}_{i,j+1}^{y,-}\right)$$

Des weiteren seien die Reibungsterme komponentenweise gegeben durch

$$s_{ij}^{x,\pm} = \frac{1}{\Delta x}(v_{i+1,j} \pm c_k u_{i+1,j} - v_{ij} \mp c_k u_{ij})\phi(\theta_{ij}^{x,\pm})$$

und

$$s_{ij}^{y,\pm} = \frac{1}{\Delta y}(w_{i,j+1} \pm d_k u_{i,j+1} - w_{ij} \mp d_k u_{ij})\phi(\theta_{ij}^{y,\pm})$$

Außerdem ist

$$\theta_{ij}^{x,\pm} = \frac{v_{ij} \pm c_k u_{ij} - v_{i-1,j} \mp c_k u_{i-1,j}}{v_{i+1,j} \pm c_k u_{i+1,j} - v_{ij} \mp c_k u_{ij}}$$

sowie

$$\theta_{ij}^{y,\pm} = \frac{w_{ij} \pm d_k u_{ij} - w_{i,j-1} \mp d_k u_{i,j-1}}{w_{i,j+1} \pm d_k u_{i,j+1} - w_{ij} \mp d_k u_{ij}}$$

wobei $\phi(\mathbf{x})$ eine Limiterfunktion ist. Eine solche Funktion ist beispielsweise der *MinMod Limiter*, der sich komponentenweise ergibt zu

$$\phi(x) = max(0, min(1, x)) \tag{2.11}$$

Damit kann man nun den Algorithmus wie folgt herleiten.

2.2.4. Herleitung des Algorithmus

Ein wesentliches Problem für die Implementierung dieses expliziten Ansatzes ergibt sich aus der Tatsache, dass es neben den offensichtlich zu bewerkstelligenden Linearkombinationen kaum Rechnungen gibt, die auf wiederverwendbaren Komponenten einer Bibliothek für Lineare Algebra (LA) aufsetzen. Das Ziel der PG ist es, eine möglichst *allgemeingültige* Bibliothek von Operationen bereitzustellen. Eine Löserimplementierung muss dem Rechnung tragen, was hier bewerkstelligt wird, indem die Nachbarschaftsbeziehungen der finiten Volumen (FV) Gitterpunkte ausgenutzt werden. Dazu werden die Terme derart neu angeordnet, dass ein System von Vektorgleichungen entstand, das die Relaxationsvektoren auf Linearkombinationen von Matrix-Vektor Produkten zurückführt. Dazu wird das Ursprungsgleichungssystem wie folgt umgeformt. Betrachtet man die Berechnung der Zwischenergebnisse für die $\mathbf{u}_{ij}^{(1)}$,

$$\mathbf{u}_{ij}^{(1)} = \mathbf{u}_{ij}^{n,1} - \Delta t \left(\Delta_+^x \mathbf{v}_{ij}^{n,1} + \Delta_+^y \mathbf{w}_{ij}^{n,1}\right) + \Delta t \mathbf{S}\left(\mathbf{u}_{ij}^{n,1}\right)$$

so expandiert man zunächst die Raumdiskretisierungsoperatoren Δ_+^x und Δ_+^y :

$$\mathbf{u}_{ij}^{(1)} = \mathbf{u}_{ij}^{n,1} \underbrace{-\frac{\Delta t}{\Delta x}\mathbf{v}_{i+\frac{1}{2},j}^{n,1} + \frac{\Delta t}{\Delta x}\mathbf{v}_{i-\frac{1}{2},j}^{n,1}}_{\mathbf{a}} - \frac{\Delta t}{\Delta y}\mathbf{w}_{i+\frac{1}{2},j}^{n,1} + \frac{\Delta t}{\Delta y}\mathbf{w}_{i-\frac{1}{2},j}^{n,1} + \Delta t \mathbf{S}\left(\mathbf{u}_{ij}^{n,1}\right)$$

Bevor man nun das MUSCL Schema anwendet, um die Zwischenwerte aufzulösen, also jene mit nicht-ganzzahligem Index, kann man die Gleichung in zwei Teile aufteilen, derart, dass sich ein Teil auf den Vektor $\mathbf{v}$ und einer auf den Vektor $\mathbf{w}$ bezieht. Der

Quellterm wird separat betrachtet. Damit ergibt sich für $\mathbf{a}$ bei gleichzeitigem Einsetzen der Reibungsterme:

$$-\frac{\Delta t}{\Delta x}\left(\frac{1}{2}\mathbf{v}_{ij}^{n,1}+\frac{1}{2}\mathbf{v}_{i+1,j}^{n,1}-\frac{c_k}{2}\mathbf{u}_{i+1,j}^{n,1}+\frac{c_k}{2}\mathbf{u}_{i,j}^{n,1}+\frac{1}{4}\left(\mathbf{e}_1\phi\left(\mathbf{\Theta}_{ij}^{x,+}\right)-\mathbf{e}_2\phi\left(\mathbf{\Theta}_{i+1,j}^{x,-}\right)\right)\right)$$
$$+\frac{\Delta t}{\Delta x}\left(\frac{1}{2}\mathbf{v}_{i-1,j}^{n,1}+\frac{1}{2}\mathbf{v}_{i,j}^{n,1}-\frac{c_k}{2}\mathbf{u}_{i,j}^{n,1}+\frac{c_k}{2}\mathbf{u}_{i-1,j}^{n,1}+\frac{1}{4}\left(\mathbf{e}_3\phi\left(\mathbf{\Theta}_{i-1,j}^{x,+}\right)-\mathbf{e}_4\phi\left(\mathbf{\Theta}_{ij}^{x,-}\right)\right)\right)$$

wobei

$$\begin{aligned}
\mathbf{e}_1 &= \mathbf{v}_{i+1,j}^{n,1}+c_k\mathbf{u}_{i+1,j}^{n,1}-\mathbf{v}_{ij}^{n,1}+c_k\mathbf{u}_{ij}^{n,1}\\
\mathbf{e}_2 &= \mathbf{v}_{i+2,j}^{n,1}-c_k\mathbf{u}_{i+2,j}^{n,1}-\mathbf{v}_{i+1,j}^{n,1}+c_k\mathbf{u}_{i+1,j}^{n,1}\\
\mathbf{e}_3 &= \mathbf{v}_{ij}^{n,1}-c_k\mathbf{u}_{ij}^{n,1}-\mathbf{v}_{i-1,j}^{n,1}+c_k\mathbf{u}_{i-1,j}^{n,1}\\
\mathbf{e}_4 &= \mathbf{v}_{i+1,j}^{n,1}-c_k\mathbf{u}_{i+1,j}^{n,1}-\mathbf{v}_{ij}^{n,1}+c_k\mathbf{u}_{ij}^{n,1}
\end{aligned} \tag{2.12}$$

Klammert man nun geeignet aus und rearrangiert die Terme nach den Vorkommen von $\mathbf{u}_{ij}$, $\mathbf{u}_{i-1,j}$, $\mathbf{u}_{i+1,j}$, $\mathbf{u}_{i+2,j}$ bzw. $\mathbf{v}_{ij}$, $\mathbf{v}_{i-1,j}$, $\mathbf{v}_{i+1,j}$, $\mathbf{v}_{i+2,j}$ so ergeben sich die Koeffizienten α_{ij} der Matrix M_1:

$$\begin{aligned}
\alpha_{i-1,j} &= \left[\frac{\Delta t}{4\Delta x}\left(2-\phi\left(\mathbf{\Theta}_{i-1,j}^{x,+}\right)\right)\right]\\
\alpha_{ij} &= \left[\frac{\Delta t}{4\Delta x}\left(\phi\left(\mathbf{\Theta}_{ij}^{x,+}\right)+\phi\left(\mathbf{\Theta}_{i-1,j}^{x,+}\right)+\phi\left(\mathbf{\Theta}_{ij}^{x,-}\right)\right)\right]\\
\alpha_{i+1,j} &= \left[-\frac{\Delta t}{4\Delta x}\left(2+\phi\left(\mathbf{\Theta}_{ij}^{x,+}\right)+\phi\left(\mathbf{\Theta}_{i+1,j}^{x,-}\right)+\phi\left(\mathbf{\Theta}_{ij}^{x,-}\right)\right)\right]\\
\alpha_{i+2,j} &= \left[\frac{\Delta t}{4\Delta x}\left(\phi\left(\mathbf{\Theta}_{i+1,j}^{x,-}\right)\right)\right]
\end{aligned} \tag{2.13}$$

Das heißt, dass M_1 eine nicht-symmetrische 4-Bandmatrix ist. Für die Koeffizienten β_{ij} von M_3 gilt:

$$\begin{aligned}
\beta_{i-1,j} &= \left[\frac{c_k\Delta t}{4\Delta x}\left(2-\phi\left(\mathbf{\Theta}_{i-1,j}^{x,+}\right)\right)\right]\\
\beta_{ij} &= \left[\frac{-c_k\Delta t}{4\Delta x}\left(4-\phi\left(\mathbf{\Theta}_{ij}^{x,+}\right)-\phi\left(\mathbf{\Theta}_{i-1,j}^{x,+}\right)+\phi\left(\mathbf{\Theta}_{ij}^{x,-}\right)\right)\right]\\
\beta_{i+1,j} &= \left[\frac{c_k\Delta t}{4\Delta x}\left(2-\phi\left(\mathbf{\Theta}_{ij}^{x,+}\right)+\phi\left(\mathbf{\Theta}_{i+1,j}^{x,-}\right)+\phi\left(\mathbf{\Theta}_{ij}^{x,-}\right)\right)\right]
\end{aligned} \tag{2.14}$$

$$\beta_{i+2,j} = \left[\frac{-c_k \Delta t}{4\Delta x} \left(\phi \left(\mathbf{\Theta}^{x,-}_{i+1,j} \right) \right) \right]$$

Bei der Herleitung der Matrizen aus den Gleichungen für die Geschwindigkeits-Relaxationsvektoren müssen zusätzlich noch die Matrizen C und D beachtet werden. Aus

$$\mathbf{v}^{(1)}_{ij} = \mathbf{v}^{n,1}_{ij} - \Delta t C^2 \Delta^x_+ \mathbf{u}^{n,1}_{ij}$$

wird dabei

$$\begin{aligned} \mathbf{v}^{(1)}_{ij} &= \mathbf{v}^{n,1}_{ij} - \Delta t c_k^2 \Delta^x_+ \mathbf{u}^{n,1}_{ij} \\ &= \mathbf{v}^{n,1}_{ij} \underbrace{- \frac{\Delta t c_k^2}{\Delta x} \left(\mathbf{u}^{n,1}_{i+\frac{1}{2},j} - \mathbf{u}^{n,1}_{i-\frac{1}{2},j} \right)}_{\mathbf{b}} \end{aligned}$$

Damit ist

$$\begin{aligned} \mathbf{b} = &\frac{-\Delta t c_k^2}{\Delta x} \left(\frac{1}{2} \mathbf{u}^{n,1}_{ij} + \frac{1}{2} \mathbf{u}^{n,1}_{i+1,j} - \frac{1}{2c_k} \mathbf{v}^{n,1}_{i+1,j} + \frac{1}{2c_k} \mathbf{v}^{n,1}_{ij} + \frac{1}{4c_k} \left(\mathbf{e}_1 \phi \left(\mathbf{\Theta}^{x,+}_{ij} \right) + \mathbf{e}_2 \phi \left(\mathbf{\Theta}^{x,-}_{i+1,j} \right) \right) \right) \\ &+ \frac{\Delta t c_k^2}{\Delta x} \left(\frac{1}{2} \mathbf{u}^{n,1}_{i-1,j} + \frac{1}{2} \mathbf{u}^{n,1}_{ij} - \frac{1}{2c_k} \mathbf{v}^{n,1}_{ij} + \frac{1}{2c_k} \mathbf{v}^{n,1}_{i-1,j} + \frac{1}{4c_k} \left(\mathbf{e}_3 \phi \left(\mathbf{\Theta}^{x,+}_{i-1,j} \right) + \mathbf{e}_4 \phi \left(\mathbf{\Theta}^{x,-}_{i,j} \right) \right) \right) \end{aligned}$$

Seien nun γ_{ij} bzw. δ_{ij} die Koeffizienten von M_5 bzw. M_6 dann gilt:

$$\gamma_{i-1,j} = \left[\frac{c_k \Delta t}{4\Delta x} \left(2 - \phi \left(\mathbf{\Theta}^{x,+}_{i-1,j} \right) \right) \right]$$

$$\gamma_{ij} = \left[\frac{-c_k \Delta t}{4\Delta x} \left(4 - \phi \left(\mathbf{\Theta}^{x,+}_{ij} \right) - \phi \left(\mathbf{\Theta}^{x,+}_{i-1,j} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{ij} \right) \right) \right] \tag{2.15}$$

$$\gamma_{i+1,j} = \left[\frac{c_k \Delta t}{4\Delta x} \left(2 - \phi \left(\mathbf{\Theta}^{x,+}_{ij} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{i+1,j} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{ij} \right) \right) \right]$$

$$\gamma_{i+2,j} = \left[\frac{-c_k \Delta t}{4\Delta x} \left(\phi \left(\mathbf{\Theta}^{x,-}_{i+1,j} \right) \right) \right]$$

und

$$\delta_{i-1,j} = \left[\frac{c_k \Delta t}{4\Delta x} \left(2 - \phi \left(\mathbf{\Theta}^{x,+}_{i-1,j} \right) \right) \right]$$

$$\delta_{ij} = \left[\frac{c_k^2 \Delta t}{4\Delta x} \left(\phi \left(\mathbf{\Theta}^{x,+}_{ij} \right) + \phi \left(\mathbf{\Theta}^{x,+}_{i-1,j} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{ij} \right) \right) \right] \tag{2.16}$$

$$\delta_{i+1,j} = \left[-\frac{c_k^2 \Delta t}{4\Delta x} \left(2 + \phi \left(\mathbf{\Theta}^{x,+}_{ij} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{i+1,j} \right) + \phi \left(\mathbf{\Theta}^{x,-}_{ij} \right) \right) \right]$$

$$\delta_{i+2,j} = \left[\frac{c_k^2 \Delta t}{4\Delta x} \left(\phi \left(\mathbf{\Theta}_{i+1,j}^{x,-} \right) \right) \right]$$

Die Matrix M_2 ist symmetrisch zu M_1 in y, d.h. die Indexverschiebung bezieht sich auf die j und die Θ -Werte auf y. Des weiteren ist die Schrittweite in y -Richtung relevant. M_4 ist dann auf dieselbe Weise symmetrisch zu M_3wobei aber zusätzlich noch c_k durch d_k ersetzt wird. Die Herleitung für die Matrizen M_7 und M_8, welche für die Berechnung von $\mathbf{w}_{ij}^{(1)}$ benötigt werden, erfolgt analog.

Damit erhält man

$$\begin{aligned} \mathbf{u}_{ij}^{(1)} &= \mathbf{u}_{ij}^{n,1} + M_1 \mathbf{v}_{ij}^{n,1} + M_2 \mathbf{w}_{ij}^{n,1} + M_3 \mathbf{u}_{ij}^{n,1} + M_4 \mathbf{u}_{ij}^{n,1} + \mathbf{S}\left(\mathbf{u}_{ij}^{n,1} \right) \\ \mathbf{v}_{ij}^{(1)} &= \mathbf{v}_{ij}^{n,1} + M_5 \mathbf{v}_{ij}^{n,1} + M_6 \mathbf{u}_{ij}^{n,1} \\ \mathbf{w}_{ij}^{(1)} &= \mathbf{w}_{ij}^{n,1} + M_7 \mathbf{w}_{ij}^{n,1} + M_8 \mathbf{u}_{ij}^{n,1} \end{aligned} \tag{2.17}$$

und hat so die Hauptberechnungslast des Verfahrens in Aufrufe von LA-Operationen gegliedert, respektive in Bandmatrix-Vektor Produkte. Einzig der Quellterm wird hier noch separat behandelt. Die drei obigen Gleichungen zusammen bilden die Vorhersagephase (*prediction stage*).

Zu betrachten bleiben nun noch die restlichen Linearkombinationen des Verfahrens. Hier werden zwei Initialisierungsphasen (im folgenden auch *setup stage*) unterschieden. Die erste wird durch die folgenden Gleichungen beschrieben:

$$\begin{aligned} \mathbf{u}_{ij}^{n,1} &= \mathbf{u}_{ij}^{n} \\ \mathbf{v}_{ij}^{n,1} &= \frac{1}{\epsilon - \Delta t} \left(\epsilon \mathbf{v}_{ij}^{n} - \Delta t \mathbf{F}\left(\mathbf{u}_{ij}^{n} \right) \right) \\ \mathbf{w}_{ij}^{n,1} &= \frac{1}{\epsilon - \Delta t} \left(\epsilon \mathbf{w}_{ij}^{n} - \Delta t \mathbf{G}\left(\mathbf{u}_{ij}^{n} \right) \right) \end{aligned} \tag{2.18}$$

Der zweite Initialisierungsschritt findet zwischen den beiden Vorhersageschritten statt und berechnet

$$\begin{aligned} \mathbf{u}_{ij}^{n,2} &= \mathbf{u}_{ij}^{(1)} \\ \mathbf{v}_{ij}^{n,2} &= \frac{1}{\epsilon + \Delta t} \left(\mathbf{v}_{ij}^{(1)} + \Delta t \mathbf{F}\left(\mathbf{u}_{ij}^{n,2} \right) - 2\Delta t \left(\mathbf{v}_{ij}^{n,1} - \mathbf{F}\left(\mathbf{u}_{ij}^{n,1} \right) \right) \right) \\ \mathbf{w}_{ij}^{n,2} &= \frac{1}{\epsilon + \Delta t} \left(\mathbf{w}_{ij}^{(1)} + \Delta t \mathbf{G}\left(\mathbf{u}_{ij}^{n,2} \right) - 2\Delta t \left(\mathbf{w}_{ij}^{n,1} - \mathbf{G}\left(\mathbf{u}_{ij}^{n,1} \right) \right) \right) \end{aligned} \tag{2.19}$$

Man beachte dabei, dass die Flussterme ganz rechts in den beiden letzten Gleichungen aus den Werten des ersten Vorhersageschrittes berechnet werden. Diese Terme sind also bereits bekannt, müssen aber an den entsprechenden Stellen gespeichert werden.

Abb.2.5 zeigt die Abfolge des Verfahrens.

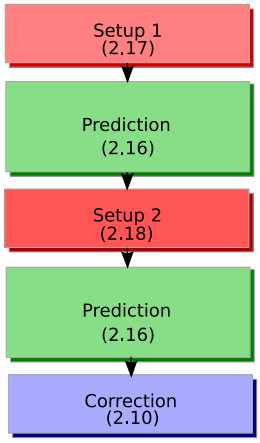


Abbildung 2.5.: Das Relaxationsverfahren: Grobe Abfolge ohne Vorbehandlung und Nachbehandlung. Die einzelnen Schritte sind mit den jeweiligen Formeln assoziiert.

2.3. Kräftebasierte Layout-Algorithmen für Graphen

Das Zeichnen von Graphen kann auf verschiedene Weise mit unterschiedlichen Verfahren vorgenommen werden, jedoch ist es unerlässlich, gewisse ästhetische Kriterien bei der Auswahl der Zeichenverfahren zu berücksichtigen. So ist es sinnvoll, Symmetrieeigenschaften in dem Layout zu berücksichtigen und adjazente Knoten in der späteren Zeichnung räumlich näher zueinander darzustellen als nicht adjazente Knoten, die über einen gewissen Mindestabstand verfügen sollten. Weitere Grundlagen und Informationen zu diesem Thema findet man z.B. in den Vorlesungsskripten von Jünger [JBHP06] und Brandenburg [Bra02].

Kräftebasierte Layout-Verfahren berechnen Layout-Lösungen, welche diese Kriterien erfüllen, indem sie das Ausgangsproblem als ein physikalisches Kräftemodell formulieren. Knoten werden hierbei als elektrisch geladene Partikel interpretiert, die sich gegenseitig abstoßen, während Kanten als Federn betrachtet werden, die verhindern, dass die Knoten unkontrolliert auseinander driften, indem sie eine anziehende Kraft in den Kräftemodell ausüben. Ziel der Verfahren ist die Berechnung eines Zustandes mit minimaler Energie, d.h. dass ein Gleichgewichtszustand des Kräftesystems approximiert wird. Besonders geeignet ist diese Klasse von Zeichenverfahren für dünne Graphen, also für Graphen, deren Kantenanzahl im Verhältnis zu der Anzahl der Knoten klein ist.

Das zugrunde liegende Kräftemodell soll hier kurz erläutert werden. An einem Knoten $v \in V$ eines Graphen $G = (V, E)$ wirken abstoßende Kräfte F_{rep} und zudem anziehende Kräfte F_{attr}, falls er mit einem anderen Knoten über eine Kante verbunden ist. Mit der *Federkraft* $f_{u,v}$ zwischen den Knoten u und v, sowie der *elektrischen Abstoßung* zwischen u und v ergibt sich die Gesamtkraft am Knoten v also durch

$$F(v) \quad = \quad F_{\text{attr}}(v) + F_{\text{rep}}(v) = \sum_{(u,v)\in E} f_{u,v} + \sum_{(u,v)\in V\times V} g_{u,v}. \tag{2.20}$$

Die *Distanz* zwischen zwei Knoten u und v mit den Positionen $p_u = (x_u, y_u)$ und $p_v = (x_v, y_v)$ ergibt sich durch die euklidische Norm zu

$$d(p_u, p_v) \quad = \quad \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}.$$

Bezeichnet weiter k die *Federsteifigkeit*, dann gilt nach dem *Hooke'schen Gesetz* zur Berechnung der Federkraft $f_{u,v}$ für eine natürliche Federlänge $l_{u,v}$ und die Federelongation Δl:

$$f_{u,v} \quad = \quad k^{(1)}_{u,v} \Delta l = k^{(1)}_{u,v}(d(p_u, p_v) - l_{u,v}).$$

Es handelt sich bei dem Modell um eine Mischform mit der Vorstellung von Federn und elektrischen Kräften. Wird die elektrische Abstoßung $g_{u,v}$ mit

$$g_{u,v} \quad = \quad \frac{k^{(2)}_{u,v}}{d(p_u, p_v)^2}$$

definiert, wobei $k^{(2)}_{u,v}$ die Stärke der elektrischen Abstoßung bezeichnet, so erhält man insgesamt

$$F_{\text{attr},x}(v) \quad = \quad \sum_{(u,v)\in E} k^{(1)}_{u,v}(d(p_u, p_v) - l_{u,v}) \frac{(x_v - x_u)}{d(p_u, p_v)} \tag{2.21}$$

bzw.

$$F_{\text{rep},x}(v) \quad = \quad \sum_{(u,v)\in V\times V, u\neq v} \frac{k^{(2)}_{u,v}}{d(p_u, p_v)^3}(x_v - x_u) \tag{2.22}$$

sowie

$$F_{\text{attr},y}(v) \quad = \quad \sum_{(u,v)\in E} k^{(1)}_{u,v}(d(p_u, p_v) - l_{u,v}) \frac{(y_v - y_u)}{d(p_u, p_v)} \tag{2.23}$$

bzw.

$$F_{\text{rep},y}(v) \quad = \quad \sum_{(u,v)\in V\times V, u\neq v} \frac{k^{(2)}_{u,v}}{d(p_u, p_v)^3}(y_v - y_u). \tag{2.24}$$

Die Federkraft $f_{u,v}$ wirkt einer zu großen Streckung der Kanten entgegen, während die elektrische Abstoßung der Knoten verhindert, dass die Knoten zu dicht beieinander liegen. Über die Parameter $l_{u,v}$, $k^{(1)}_{u,v}$ und $k^{(2)}_{u,v}$ lässt sich die Darstellung des Graphen entsprechend beeinflussen.

Zwei kräftebasierte Verfahren, die sich auf verschiedene Weisen diese Modellvorstellung zu Nutze machen und die im Rahmen der Projektgruppe implementiert wurden, sollen nun genauer betrachtet werden.

2.3.1. Kamada-Kawai

Sei $G = (V, E)$ ein zusammenhängender Graph mit Knoten $u, v \in V$, und sei $\delta(u, v)$ die Anzahl der Kanten auf einem kürzesten (u, v)-Weg. In dem ungewichteten Verfahren von Kamada-Kawai [KK89] werden statt elektrischer Kräfte Federn verwendet. Daher lässt sich der *Kräfteparameter* k hier als *Steifigkeitsparameter* einer Feder interpretieren, so dass nun Federn zwischen allen Paaren von Knoten definiert werden können. Der kürzeste Weg zwischen zwei Knoten u und v sowie die Anzahl der Kanten auf ihm werden mittels Breitensuche berechnet. Die Kräftefunktionen der Knoten werden über die potentielle Energie des Gesamtsystems bestimmt (siehe Abbildung 2.6).

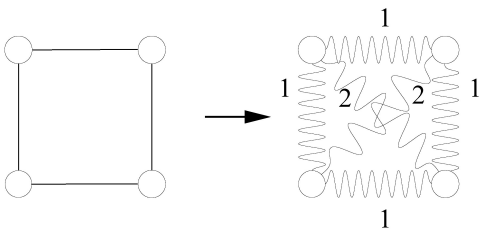


Abbildung 2.6.: Veranschaulichung der Kräftefunktion

Das Kraft-Weg-Diagramm in Abbildung 2.7 stellt die Kraft $F(x)$ zweier Federn mit unterschiedlichen Federsteifigkeitsparametern dar. Die *potentielle Energie* berechnet sich

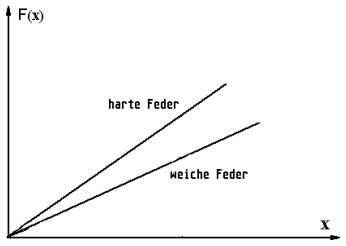


Abbildung 2.7.: Zum Begriff der potentiellen Energie

durch:

$$\nu(x) \quad = \quad \frac{1}{2}F(x)x = \frac{1}{2}kx^2. \tag{2.25}$$

Im Verfahren von Kamada-Kawai wird die potentielle Energie der (u, v)-Feder beschrieben durch die Funktion

$$\frac{1}{2}k_{uv}(d(p_u, p_v) - \delta(u, v))^2.$$

Es ist zu beachten, dass die potentielle Energie der (u, v)-Feder gleich Null wird, wenn die Distanz zwischen u und v der *graphentheoretischen Distanz* $\delta(u, v)$ der beiden Punkte entspricht. Zwei Knoten, die über eine große graphentheoretische Distanz zueinander

verfügen, üben nur eine geringe Anziehung aufeinander aus; dies spiegelt sich in der resultierenden Zeichnung dadurch wider, dass diese Knoten weiter voneinander entfernt liegen.

Für den Steifigkeitsparameter k_{uv} wählt man:

$$k_{uv} = \frac{k}{\delta(u,v)^2}.$$

Die *potentielle Energie* der (u,v)-Feder ergibt sich zu:

$$\frac{k}{2}\left(\frac{d(p_u,p_v)}{\delta(u,v)} - 1\right)^2.$$

Für die potentielle Energie der Zeichnung erhält man dann insgesamt:

$$\nu = \frac{k}{2}\sum_{(u,v)\in V, u\neq v}\left(\frac{d(p_u,p_v)}{\delta(u,v)} - 1\right)^2.$$

Die notwendigen Bedingungen für eine minimale potentielle Energie ν ergeben sich durch die partiellen Ableitungen

$$\frac{\delta\nu}{\delta x_v} = 0 \text{ und} \qquad \frac{\delta\nu}{\delta y_v} = 0 \text{ für alle } v \in V.$$

In dem Verfahren von Kamada-Kawai wird die minimale potentielle Energie ν über einen iterativen Ansatz bestimmt und lässt sich mit dem folgenden Algorithmus darstellen:

Algorithm: Kamada-Kawai

Input: Graph $G = (V, E)$
Output: geradlinige Zeichnung $\Gamma(G)$
starte mit einer beliebigen Verteilung der Knoten
berechne k_{uv} und $\delta(u,v)$
while *($\nu \geq \epsilon$)* **do**
  berechne Knoten v: Knoten mit größter Kraft mit $\sqrt{\left(\frac{\delta\nu}{\delta x_v}\right)^2 + \left(\frac{\delta\nu}{\delta y_v}\right)^2}$
  verschiebe v in eine energieminimale Position (bei festen Positionen der anderen Knoten)
end

Die Gesamtkomplexität des Verfahrens läßt sich mit $O(n^3) + O(Tn)$ in der *Landau Notation* ausdrücken, wobei T die Anzahl der Iterationen ist. Ein Nachteil dieses Verfahrens besteht darin, dass ein Verfangen in einem *lokalen Energieminimum* möglich ist. Abhilfe leisten hier beispielsweise *Swap-Heuristiken* (siehe Brandenburg [Bra02]).

Gewichtetes Verfahren

Für einen Knoten $v \in V$ eines Graphen $G = (V, E)$ wird die Verschiebung des Knotens durch

$$F_{\mathrm{KK}}(v) = \sum_{u \in N_i(v)} \left(\frac{d(u,v)^2 \delta(u,v)^2}{k^2} - 1 \right) (p_u - p_v)$$

berechnet, wobei $N_i(v)$ die Menge der freien Nachbarn von v ist. Da das Verfahren von Kamada-Kawai die Distanz zwischen Paaren von Knoten berechnet, ist es wegen der hohen Rechen- und Speicherplatzanforderungen bei der Berechnung aller Pfade zwischen zwei Knoten und der Ermittlung des kürzesten gewichteten Pfades zwischen ihnen praktisch unerlässlich, statt dessen eine *approximierte optimale Distanz* für gewichtete Graphen zu bestimmen.

Sei dazu $p_1, p_2, ..., p_n$ die Knotensequenz in dem kürzesten Pfad zwischen zwei Knoten u und v im Graphen G, dann ist der gewichtete Kräftevektor durch

$$F_{\mathrm{KKg}}(v) = \sum_{u \in N_i(v)} \left(\frac{2d(u,v)^2}{\mathrm{optDist}_G(u,v)^2 + d(u,v)^2} - 1 \right) (p_u - p_v) \tag{2.26}$$

mit

$$\mathrm{optDist}_G(u,v) = \sum_{i=2}^{n} \frac{\sqrt{w_{p_i} w_{p_{i-1}}}}{w_{e_{p_i p_{i-1}}}}$$

gegeben.

2.3.2. Fruchterman-Reingold

Beim Verfahren von Fruchterman-Reingold [FR91] werden hingegen Kräfte zwischen allen Knoten berechnet. Das Verfahren enthält nur noch einen Parameter, den globalen *Kräfteparameter* k, so dass die Parameter $l_{u,v}, k^{(1)}_{u,v}$ und $k^{(2)}_{u,v}$ entsprechend ersetzt werden:

$$\begin{aligned} k^{(1)}_{u,v} &= d(p_u, p_v)^2/k^2, \text{ für alle } (u,v) \in E \\ k^{(2)}_{u,v} &= d(p_u, p_v)k^2, \text{ für alle } u, v \in V \\ l_{u,v} &= 0, \text{ für alle } (u,v) \in E. \end{aligned}$$

Man erhält für die anziehenden und abstoßenden Kräfte F_{attr} und F_{rep} im Zusammenhang mit

$$\begin{aligned} F(v) &= F_{\mathrm{attr}}(v) + F_{\mathrm{rep}}(v) \\ &= \sum_{(u,v) \in E} f_{u,v} + \sum_{(u,v) \in V \times V} g_{u,v} \end{aligned}$$

die folgenden Darstellungen:

$$F_{\text{attr},x}(v) = \sum_{(u,v)\in E} \frac{d(p_u,p_v)^2(x_v - x_u)}{k^2} \tag{2.27}$$

$$F_{\text{rep},x}(v) = \sum_{(u,v)\in V\times v} \frac{k^2 d(x_u - x_v)}{d(p_u,p_v)^2} \frac{(x_u - x_v)}{d(p_u,p_v)}. \tag{2.28}$$

Ist die Differenz zwischen anziehender und abstoßender Kraft von zwei adjazenten Knoten u und v gleich Null, so entspricht ihr Abstand $d(p_u, p_v)$ dem Parameter k. Daher wird k üblicherweise als *Ideallänge* bezeichnet. Weitere Details zum Algorithmus findet man in der Arbeit von Forrester [FKN+04], in der Dokumentation zum Graphenlayout-System Graphael [FKN+03] sowie im Vorlesungsskript von Brandenburg [Bra02].

Gewichtetes Verfahren

Bei dem ungewichteten Verfahren von Fruchterman-Reingold wird lediglich die Adjazenz zwischen Knoten berücksichtigt und die anziehenden und abstoßenden Kräfte an einem Knoten v des Graphen G mit Hilfe folgender Kräftevektoren berechnet, wobei $Adj.(v)$ die Menge aller zu v adjazenten Knoten ist (vgl. Darstellung in [FKN+03]):

$$F_{\text{attr}}(v) = \sum_{u\in Adj.(v)} \frac{d(u,v)^2}{k^2}(p_u - p_v) \tag{2.29}$$

$$F_{\text{rep}}(v) = \sum_{u\in N_i(v)} \frac{k^2}{d(u,v)^2}(p_v - p_u). \tag{2.30}$$

Um nun Kanten- und Knotengewichte eines Graphen mit in die Berechnung einbeziehen zu können, muss das Verfahren derart modifiziert werden, dass es Gewichte und ideale Kantenlängen berücksichtigt. Dabei ergeben sich folgende Modifikationen der Kräftevektoren:

$$F_{\text{attr}}(v) = \sum_{u\in Adj.(v)} w_e^4 d(u,v)^2(p_u - p_v)$$

$$F_{\text{rep}}(v) = \sum_{u\in N_i(v)} \frac{(w_u w_v)^2}{d(u,v)^2}(p_u - p_v).$$

2.3.3. Evolving Graphs

Als Anwendungsgebiet der Projektgruppenarbeit wurden insbesondere sich dynamisch über die Zeit ändernde Graphen betrachtet. Dabei wird ein Zeitintervall $I = [0; t]$ mit $t \in \mathbb{N}$ definiert, dessen (diskrete) Zeitabschnnitte i als *Timeslices*[3] bezeichnet werden. Dem Graphen G liegt eine Folge von Subgraphen $S_G = G_0, G_1, ..., G_t$ zugrunde, wobei G_i der zum Zeitabschnitt i gehörige Subgraph ist. Das gesamte System $G' = (G, S_G)$ kann nun als ein zeitabhängiges diskretes dynamisches System betrachten werden, welches sich über das Zeitintervall I entwickelt. Systeme dieses Typs heißen *Evolving Graphs*.

Jeder *Timeslice* enthält alle Knoten und Kanten des zugehörigen Subgraphen des entsprechenden Zeitpunktes sowie weitere *Timeslice*-Attribute. Zwischen verbundenen Knoten desselben *Timeslice* wirken nur abstoßende Kräfte. Korrespondierende Knoten aus verschiedenen *Timeslices* werden über sogenannte gewichtete *Inter-Timeslice*-Kanten miteinander verbunden. Die Ideallänge der *Inter-Timeslice*-Kante e zwischen zwei korrespondierenden Knoten u und v wird an Hand der Gewichte der Kante $e = (u, v)$, sowie der Knoten u und v durch

$$\text{IdealLength}(e) = \sqrt{w_u w_v}/w_e$$

berechnet. Die optimale Distanz zwischen zwei korrespondierenden Knoten aus verschiedenen *Timeslices* ist gleich Null, da zwischen diesen beiden Knoten nur anziehende Kräfte bestehen. Man geht in der Praxis davon aus, dass alle Kanten mit einem Gewicht von $w_e = 0$ aus dem Graphen entfernt werden.

Da es sich bei *Evolving Graphs* um ein gewichtetes Verfahren handelt, verfügen sowohl Knoten als auch Kanten über Gewichte w_i. Dabei sollen folgende Ziele erreicht werden:

- Zwei Knoten u und v, die durch eine Kante e mit dem Kantengewicht $w_e = 0$ verbunden sind, sollten sich so verhalten, als wenn sie gar nicht miteinander verbunden wären.
- Eine Kante e, die zwei Knoten u und v mit den Knotengewichten $w_u = w_v = 0$ verbindet, sollte eine Länge von Null haben und verbundene Knoten mit kleinen Gewichten ziehen einander an, während sich Knoten mit großen Gewichten weit voneinander entfernt befinden sollen.
- Das Gewicht einer *Inter-Timeslice*-Kante zwischen zwei Knoten u und v mit den Gewichten w_u und w_v wird als Durchschnitt angenommen und mit $(w_u + w_v)/2$ berechnet.
- *Inter-Timeslice Kanten* mit hohem Gewicht sorgen dafür, dass die Knotenpositionen in jeder Grapheninstanz fixiert werden, während leichtgewichtete *Inter-Timeslice*-Kanten zu einem nahezu unabhängigen Layout in jeder Grapheninstanz führen.

[3] Hier und im folgenden wird der englische Begriff verwendet, um ihn in als graphentheoretischen Terminus zu betonen.

So wird die Lesbarkeit des Graphen durch Veränderung der Gewichte der *Inter-Timeslice*-Kanten erleichtert. Um das Verfahren dynamisch zu halten, sollte es nun möglich sein, in jedem *Timesclice* eine beliebige Kante oder einen beliebigen Knoten hinzuzufügen oder zu entfernen und darüber hinaus beliebig viele Gewichte der Kanten und Knoten zu ändern. Um die Entwicklung des Graphen über die Zeit darstellen zu können, müssen die *Timeslice*-Attribute jedes Knotens in die Rechnung einbezogen werden. Dabei sind einige Modifikationen der Layout-Algorithmen notwendig. Abstoßende Kräfte sollen nur zwischen Knoten desselben *Timeslices* bestehen, folglich ist die optimale Distanz zwischen Knoten verschiedener *Timeslices* Null. Für zwei Knoten u und v mit den *Timeslice*-Indizes t_u und t_v wird nun die Funktion $\text{optDist}_G(u, v)$ so geändert, dass sie die *Timeslices* der betrachteten Knoten berücksichtigt:

$$\text{optDist}_G(u, v) \quad = \quad \sum_{i=2}^{n} \tau_{t_u t_v} \frac{\sqrt{w_{p_i} w_{p_{i-1}}}}{w_{e_{p_i p_{i-1}}}} \tag{2.31}$$

mit $\tau_{t_u t_v} = 1$ wenn $t_u = t_v$, sonst 0. Für weitere Details sei noch einmal auf Forrester [FKN+04] sowie die Arbeit von Kaufmann und Wagner [KW01] verwiesen.

Algorithmus-Komponenten

Der im folgenden dargestellte Algorithmus arbeitet im wesentlichen auf Vektor- und Matrizen-Datenstrukturen, in denen die Adjazenzeigenschaften bzw. die Abstände und Positionen der Knoten des Eingangsgraphen repräsentiert sowie Zwischenergebnisse der Berechnung der Kraftzusammenhänge berechnet werden. Seine Komponenten wurden als eigenständige Methoden der Graphen-Bibliothek (s. Kapitel 3) zugeordnet, da sie außerhalb des speziellen Anwendungskontextes automatischer Layoutverfahren keine allgemeine Anwendung finden würden. Zum besseren Verständnis des folgenden Algorithmus kann die Bedeutung der einzelnen Variablen und Komponenten in folgender Tabelle abgelesen werden.

Variablen (Bezeichnung)	Bedeutung
F_{ges}	Summe von anziehenden und abstoßenden Kräften
$\max F_{\text{res}}$	maximaler Wert in der Matrix der Gesamtkräfte
timeout	Anzahl der gewünschten Iterationen
D	Matrix mit Distanzen aller Knoten zueinander
D^{-1}	Matrix mit dem Kehrwert der Distanzen aller Knoten zueinander
D_{mask}	Matrix mit Distanzen aller *adjazenten* Knoten zueinander
VD_{mask}	Zeilensummenvektor von D_{mask}
VD^{-1}	Zeilensummenvektor von D^{-1}

Operation (Bezeichnung)	Bedeutung
kehr(D)	elementweise Berechnung des Kehrwertes der Distanzmatrix D mit dem Ergebnis D^{-1}
maskieren(D, E)	Matrix D wird an den Stellen (i, j) auf 0 gesetzt, an denen Adjazenzmatrix E eine 0 als Eintrag enthält
diag(VD)	Vektor D wird in die Diagonale einer Einheitsmatrix geschrieben
zsv(D)	Der Zeilensummenvektor von D wird gebildet, sodass D für jede Zeile der Matrix D die Summen ihrer Einträge enthält

Algorithm: Fruchterman-Reingold

Input: p, E, k, ϵ, timeout
Output: p
$\max F_{\text{res}} \leftarrow 3\epsilon$;
iter $\leftarrow 1$;
while $(\max F_{\text{res}} > \epsilon) \wedge (\text{iter} \leq \text{timeout})$ **do**
  for $i \in 0 \dots cols(p) - 1$ **do**
    for $j \in 0 \dots cols(p) - 1$ **do**
      $M_{\text{i,j}} = \begin{cases} \frac{1}{d(p_i,p_j)^2} & \text{falls } d(p_i, p_j) \neq 0 \\ 0 & \text{sonst} \end{cases}$
    end
  end
  $D^{-1} \leftarrow M$;
  $D_{\text{mask}} \leftarrow \text{kehr}(\text{maskieren}(D^{-1}, E))$;
  $VD_{\text{mask}} \leftarrow \text{zsv}(D_{\text{mask}})$;
  $VD^{-1} \leftarrow \text{zsv}(D^{-1})$;
  $F_{\text{attr}} \leftarrow \frac{p}{k^2}(VD_{\text{mask}} - \text{diag}(VD_{\text{mask}}))$;
  $F_{\text{rep}} \leftarrow k^2 p(\text{diag}(VD^{-1}) - (VD^{-1}))$;
  $F_{\text{ges}} \leftarrow F_{\text{attr}} + F_{\text{rep}}$;
  for $i \in 0 \dots cols(F_{\text{ges}}) - 1$ **do**
    $F_{0,i} \leftarrow |F^{(i)}_{\text{ges}}|$;
  end
  $F_{\text{res}} \leftarrow F$;
  $\max F_{\text{res}} \leftarrow \max(F_{\text{ges}})$ **for** $i \in 0 \dots cols(p) - 1$ **do**
    $p^{(i)} \leftarrow p^{(i)} + \frac{1}{10|F^{<i>}_{\text{res}}|} F^{<i>}_{\text{ges}}$
  end
  iter $\leftarrow$ iter $+ 1$;
end

2.3.4. Beispiel

Zum bessseren Verständnis werden die Schritte des Fruchterman-Reingold-Algorithmus anhand eines Beispiels illustriert. Die Eingabeparameter Positionsmatrix p, Adjazenzmatrix E und der Steifigkeit k werden dabei, wie in Abbildung 2.8 dargestellt, gewählt. Die Variable *timeout* wird dabei auf den Wert 70 gesetzt, so dass das Verfahren nach 70 Iterationen endet. Ein weiteres Abbruchkriterium ist die Variable ϵ. Zunächst wird zeilenweise

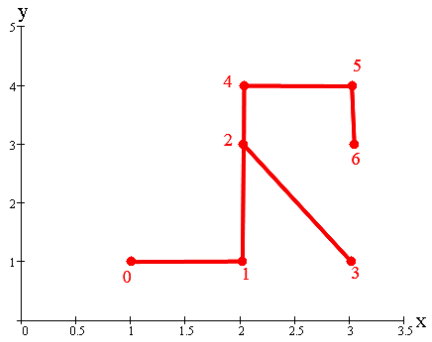
Positionsmatrix

$$p := \begin{pmatrix} 1 & 2 & 2 & 3 & 2 & 3 & 3 \\ 1 & 1 & 3 & 1 & 4 & 4 & 3 \end{pmatrix}$$

Adjazenzmatrix

$$E := \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(a) Eingabeparameter des Beispiels



(b) Graphische Darstellung der Eingabe

Abbildung 2.8.: Beispiel für die Anwendung des Verfahrens von Fruchterman-Reingold

über die Positionsmatrix iteriert und, wie in Abbildung 2.9 dargestellt, elementweise der Kehrwert der Distanzen aller Knoten zueinander berechnet und an entsprechende Stelle der Matrix D^{-1} geschrieben. Dabei bezeichnet $cols(p)$ die Spalten in der Positionsmatrix p. Die Distanz zwischen nicht adjazenten Knoten beträgt dabei jeweils Null.

Input: p
Output: D^{-1}
for $i \in 0 \dots cols(p) - 1$ **do**
  for $j \in 0 \dots cols(p) - 1$ **do**
    $D^{-1}_{i,j} =$
    $\begin{cases} \frac{1}{d(p_i, p_j)^2} & \text{falls } d(p_i, p_j) \neq 0 \\ 0 & \text{sonst} \end{cases}$
  end
end

(a) Berechnung der Kehrwerte

$$D^{-1} = \begin{pmatrix} 0 & 1 & 0.2 & 0.25 & 0.1 & 0.077 & 0.125 \\ 1 & 0 & 0.25 & 1 & 0.111 & 0.1 & 0.2 \\ 0.2 & 0.25 & 0 & 0.2 & 1 & 0.5 & 1 \\ 0.25 & 1 & 0.2 & 0 & 0.1 & 0.111 & 0.25 \\ 0.1 & 0.111 & 1 & 0.1 & 0 & 1 & 0.5 \\ 0.077 & 0.1 & 0.5 & 0.111 & 1 & 0 & 1 \\ 0.125 & 0.2 & 1 & 0.25 & 0.5 & 1 & 0 \end{pmatrix}$$

(b) Matrix D^{-1} des Beispiels

Abbildung 2.9.: Kehrwerte der Distanzen aller Knoten zueinander

Anschließend wird elementweise über die Matrix D^{-1} iteriert und der Kehrwert aller Einträge berechnet, so dass die Matrix D mit den eigentlichen Distanzen aller Knoten zueinander gebildet wird. Die Distanzmatrix D wird nun als Eingabe weiterverwendet und mit der Adjazenzmatrix E maskiert. Abbildung 2.10 zeigt die Methode *maskieren*, in der alle Distanzen zwischen nicht adjazenten Knoten auf Null gesetzt werden, so dass die resultierende Matrix D_{mask} nur noch Einträge für *adjazente* Knoten enthält.

Das Maskieren ist beim Verfahren von Fruchterman-Reingold von besonderer Bedeutung, da für die Berechnung der Kräfte zwischen adjazenten und nicht-adjazenten Knoten unterschieden werden muss.

Input: D, E
Output: D_{mask}
for $i \in 0 \dots cols(p) - 1$ **do**
  for $j \in 0 \dots cols(p) - 1$ **do**
    $D_{\text{mask},i,j} = \begin{cases} D_{i,j} & \text{falls } E_{i,j} \neq 0 \\ 0 & \text{sonst} \end{cases}$
  end
end

(a) Maskieren der Matrix D

$$D_{\text{mask}} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 5 & 1 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b) Matrix D_{mask} des Beispiels

Abbildung 2.10.: Maskieren der Distanzmatrix mit der Adjazenzmatrix

Sowohl für die Distanzmatrix D, als auch für die Kehrwerte der Distanzmatrix D^{-1} werden die Zeilensummenvektoren VD und VD^{-1} berechnet, indem über die Spalten iteriert und die jeweiligen Inhalte aufaddiert werden (vgl. Abbildung 2.11).

Input: D_{mask}
Output: VD
for $i \in 0 \dots cols(D_{\text{mask}}) - 1$ **do**
  $VD_i = 0$
  for $j \in 0 \dots cols(D_{\text{mask}}) - 1$ **do**
    $VD_i = VD_i + D_{\text{mask},i,j}$
  end
end

(a) Zeilensummenvekor

$$VD = \begin{pmatrix} 1 & 5 & 10 & 5 & 2 & 2 & 1 \end{pmatrix}$$

(b) Vektor VD des Beispiels

Abbildung 2.11.: Zeilensummenvektor der maskierten Distanz-Matrix

Die Methoden diag(VD_{mask}) bzw. diag(VD^{-1}) schreiben die zugehörigen Zeilensummenvektoren nun in die Diagonale einer Einheitsmatrix. Dies wurde hier allerdings nicht dargestellt.

Für die anziehenden Kräfte werden dabei nur die maskierten Zeilensummenvektoren der Distanzmatrizen verwendet, da in dem Verfahren von Fruchterman-Reingold keine anziehenden Kräfte zwischen nicht-adjazenten Knoten bestehen (vgl. Abbildung 2.12).
Abstoßende Kräfte hingegen wirken auch zwischen Knoten, die nicht adjazent zueinander sind, so dass sich F_{rep}, wie ebenfalls in Abbildung 2.12 dargestellt, ergibt.

$$F_{\text{attr}} = \frac{p}{k^2}(VD_{\text{mask}} - VD_{\text{mask}})$$

(a) anziehende Kräfte

$$F_{\text{attr}} = \begin{pmatrix} 0.25 & -0.25 & 1.25 & -1.25 & 0.25 & -0.25 & 0 \\ 0 & 2 & -4.25 & 2.5 & -0.25 & -0.25 & 0.25 \end{pmatrix}$$

(b) F_{attr} des Beispiels

$$F_{\text{rep}} = pk^2(\text{diag}(VD^{-1}) - VD^{-1})$$

(c) abstoßende Kräfte

$$F_{\text{rep}} = \begin{pmatrix} -8.815 & -1.2 & -6 & 7.2 & -6 & 7.015 & 7.8 \\ -4.723 & -6.133 & -0.8 & -6.133 & 9.733 & 9.456 & -1.4 \end{pmatrix}$$

(d) F_{rep} des Beispiels

Abbildung 2.12.: anziehende und abstoßende Kräfte

Die Matrix F_{ges} der Gesamtkräfte aller Knoten ergibt sich aus elementweiser Addition der Matrizen für anziehende F_{attr} und abstoßende F_{rep} Kräfte (vgl. Abbildung 2.13):

$$F_{\text{rep}} = \begin{pmatrix} -8.815 & -1.2 & -6 & 7.2 & -6 & 7.015 & 7.8 \\ -4.723 & -6.133 & -0.8 & -6.133 & 9.733 & 9.456 & -1.4 \end{pmatrix}$$

(a) Matrix F_{ges} des Beispiels

p:= **for** $i \in 0 \ldots cols(p) - 1$ **do**
  $p^{(i)} \leftarrow p^{(i)} + \frac{1}{10 \cdot |F_{\text{res}}^{(i)}|} \cdot F_{\text{ges}}^{(i)}$
end

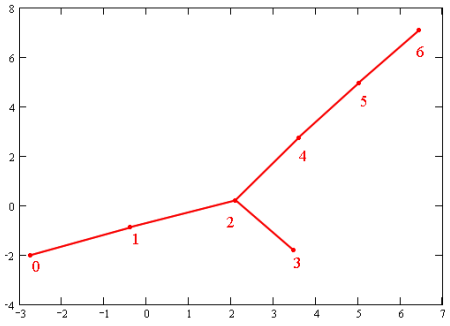
(b) Berechnung der neuen Positionen

Abbildung 2.13.: Berechnung der neuen Positionen mit Hilfe der Gesamtkräfte

In jedem Iterationsschritt werden alle Knoten gemäß der Gesamtkraft F_{ges} verschoben. Für diese Verschiebung wird die Gesamtkraft der entsprechenden Knoten auf ihre alten Knotenpositionen aufaddiert und somit eine neue Knotenposition in Form einer Positionsmatrix p errechnet. Abbildung 2.14 illustriert neben der neuen Positionsmatrix auch die graphische Darstellung des Beispiels.

$$\begin{pmatrix} 0.912 & 1.967 & 1.931 & 3.085 & 1.948 & 3.059 & 3.099 \\ 0.952 & 0.906 & 2.927 & 0.948 & 4.086 & 4.081 & 2.985 \end{pmatrix}$$

(a) neue Positionsmatrix p



(b) graphische Darstellung

Abbildung 2.14.: Layout nach Anwendung des Verfahrens von Fruchterman-Reingold

3. Die HONEI-Bibliothek

Die Programme, die Infrastruktur und die Algorithmen, welche von der Projektgruppe entwickelt wurden, sind Bestandteile der sogenannten HONEI-Bibliothek. Das Akronym HONEI steht hier abkürzend für *Hardware Oriented Numerics Efficently Implemented* und unterstreicht den Grundgedanken der Projektgruppenarbeit, nämlich die Bereitstellung einer Bibliothek, die für verschiedene Prozessor-Architekturen – insbesondere aber für die *Cell Broadband Engine* – das wissenschaftliche Rechnen mit großen Datenmengen durch effizient implementierte Basisoperationen unterstützt.

Neben der Cell-Architektur wurden die Bibliothekskomponenten für Standard-CPUs, für Prozessoren mit SSE-Erweiterungen sowie für MultiCore-Prozessoren mit SSE-Erweiterung implementiert. Es wurde dabei besonders Wert auf eine transparente Nutzung gelegt. Das wurde durch konsequente Anwendung von generischer Programmierung, d.h. dem Einsatz von C++-*Templates* verwirklicht.

Dieses Kapitel stellt die Bestandteile der HONEI-Bibliothek vor. Neben der Basisbibliothek umfasst sie die Beispielanwendungen zur Berechnung von 2D-Flachwassergleichungen und Layoutverfahren für Graphen, welche auf die Basisbibliothek zurückgreifen und sie ergänzen. Auch die Anwendungskomponenten wurden mit Blick auf eine leichtere Wiederverwendbarkeit möglichst modular konzipiert. Sicherlich sind die verschiedenen numerischen Verfahren für die Lösung linearer Gleichungssysteme, die in diesem Zusammenhang implementiert wurden, von besonderem Interesse, weil sie als relativ unabhängige Bestandteile einer Löserbibliothek leicht für unterschiedliche Zwecke eingesetzt und miteinander kombiniert werden können.

Für die Fehleranaylse der Bibliothekskomponenten wurde zudem eine eigene Test-Umgebung entwickelt, die eine weitgehende Automatisierung beim Ausführen von Funktionalitätstests erlaubt und somit die Kontrolle und Pflege der Code-Qualität erleichtert.

Außerdem wurde in HONEI eine Benchmark-Infrastruktur integriert, die als Grundlage für die folgende Evaluation der HONEI-Bibliothek diente. Die Ergebnisse der Evaluation der Bibliothek auf den verschiedenen Architekturen folgen im nächsten Kapitel.

3.1. Basisbibliotheken

In der Basisbibliothek wurden effiziente Datenstrukturen implementiert, die durch Containerklassen und die grundlegenden algebraischen Operationen, die auf ihnen durchgeführt werden können, definiert werden. Daher sollen zuerst die Container in ihrer Realisierung erläutert werden und anschließend die unterschiedlichen Vorgehensweisen bei der Implementierung der Operationen für die verschiedenen Architekturplattformen. Diese sogenannten *Backends* bezeichnen in diesem Sinn die spezifischen Realisierungen

und Optimierungen der Basisbibliothek für die verschiedenen Architekturen bzw. Erweiterungen (Cell BE, SSE, MultiCore + SSE). Sie werden im Anschluss an die Darstellung der Container genauer vorgestellt.

3.1.1. HONEI Container

Funktionaler Kern der Bibliothek sind effiziente Repräsentationen von und Operationen für Vektoren und Matrizen. Aus diesem Grund wurden verschiedene Matrix- und Vektorrepräsentationen entwickelt, um den verschiedenen Anforderungen der Anwendungsfälle gerecht zu werden. Im Detail sind dies folgende:

Vektoren

- `DenseVector`: Dicht besetzte Vektoren, die als Feld der Elemente des Vektors abgespeichert sind.
- `SparseVector`: Dünn besetzte Vektoren, die ebenfalls als Feld abgelegt werden, allerdings werden hier nur die Nicht-Null-Elemente und deren Indizes gespeichert.

Matrizen

- `DenseMatrix`: Dicht besetzte Matrizen, die als eindimensionales Array aller Elemente der Matrix (zeilenweise) abgespeichert sind.
- `SparseMatrix`: Dünn besetzte Matrizen, deren Zeilen als eigenständige Instanzen von `SparseVector` abgespeichert werden.
- `BandedMatrix`: Quadratische Bandmatrizen. Hier werden die Daten in Form von Bändern abgespeichert, die durch `DenseVector` realisiert werden. Für jedes Band existiert ein Zeiger, jedoch wird für nicht exisitierende Bänder kein Speicherplatz alloziert.

Die Unterscheidung dieser Typen ist in der Mathematik und anderen numerischen Bibliotheken gängig, da beispielsweise das Ausnutzen der Kenntnis, dass eine Matrix nur dünn oder in Form von wenigen Bandvektoren besetzt ist, bei der Ausführung von Operationen erhebliche Leistungszuwächse bringen kann. Bibliotheksintern wird das Abspeichern der Elemente in speziellen Feldern, sog. `SharedArrays` realisiert. [1]

Der Hintergrund ist folgender: Im Laufe eines Programmes entstehen sehr viele Verweise auf ein und denselben Datenbereich. Gleichzeitig soll eine optimale Ausnutzung des Speicherplatzes erzielt und nicht mehr benötigte Speicherbereiche so schnell wie möglich per wieder freigegeben werden. Das Problem ist, dass man innerhalb eines typischen Programmes normalerweise nicht feststellen kann, wann die letzte Referenz auf den Speicherbereich seine Gültigkeit verloren hat. An dieser Stelle hilft das `SharedArray`,

[1] Die in HONEI verwendete Implementation von `SharedArray` basiert auf der Entwicklung desselben aus der *Boost C++ Library*.

dessen generisch implementierte Klasse zusätzlich zum Zeiger auf das eigentliche dynamische Feld auch einen sogenannten *Reference Counter* besitzt. Dieser zählt alle erstellten und gelöschten Referenzen auf das entsprechende Feld und gibt nach Ende der Gültigkeit der letzten Referenz automatisch den entsprechenden Speicherbereich wieder frei.

Zugriff auf Container

Grundsätzlich werden drei Arten des Zugriffs auf Container ermöglicht:

- Zugriffsoperatoren
- Zugriff über Iteratoren
- Zugriff per Zeiger

`DenseMatrix` und `SparseMatrix` erlauben den direkten Zugriff auf Zeilen per `operator[]`. Durch die Benutzung eines zweifachen Aufrufes oder Aufruf von `operator(x,y)` kann direkt auf ein Element in Zeile `x` und Spalte `y` zugegriffen werden. Für `DenseMatrix` wurde der spaltenweise Zugriff per Methode `column()` ebenfalls ermöglicht. Bei `SparseMatrix` ist diese Methode nicht effizient implementierbar und steht daher nicht zur Verfügung.

HONEI Bandmatrizen erlauben weder direkten Zeilen- noch direkten Spaltenzugriff, aber einen Zugriff auf die einzelnen Bänder der Matrix. Die Methode `band()` bietet diese Art des Zugriffs und liefert eine Referenz auf den entsprechenden `DenseVector` zurück. Wichtig ist hier zu beachten, dass der Index vorzeichenbehaftet ist. Index 0 repräsentiert die Diagonale, das „unterste“ Band erhält Index `-size + 1`, wenn `size` die Anzahl der Zeilen bzw. Spalten der Bandmatrix bezeichnet. Analog hat das „oberste“ Band den Index `size - 1`.

Eine neue Bandmatrix kann mit Hilfe zweier Konstruktoren entweder leer oder mit einem `DenseVector` als übergebener Diagonale erzeugt werden. Zum anschließenden Hinzufügen weiterer Bänder dient die Methode `insert_band()`.

Der Zugriff per Iterator erfolgt ausnahmslos für alle Container mit Hilfe des sogenannten `ElementIterator`. Dieser erlaubt das vollständige Durchlaufen des Containers, entweder Element für Element, oder in größeren Schritten.

Der `ElementIterator` basiert auf `std::iterator` und ist ein reiner `ForwardIterator`. Wie alle anderen Iteratoren, die nachfolgend vorgestellt werden, existieren zwei Varianten des `ElementIterator`, die einerseits eine veränderbare Referenz und andererseits eine Referenz auf einen konstanten Wert liefern. Letztere Version heißt `ConstElementIterator`. Der Zugriff per Iterator ist besonders sicher, da sichergestellt ist, dass nur auf gültige Elemente des Containers zugegriffen wird, aber auch sehr langsam im Vergleich zum direkten Zugriff per Zeiger.

Für `SparseVector` und `SparseMatrix` wird noch ein weiterer nützlicher Iterator namens `NonZeroElementIterator` zur Verfügung gestellt, welcher nur über die Nicht-Null-Elemente des Containers iteriert. Analog besitzt die `SparseMatrix` einen `NonZeroRowIterator` um leere Zeile aussparen zu können.

Für `BandedMatrix` stehen ein `BandIterator` (ermöglicht auch Zugriff auf bisher nicht existierende Bänder) und ein `NonZeroBandIterator` zur Verfügung. Bei den Band-Iteratoren unterscheiden sich die Indizes von den bei der Funktion `band()` verwendeten. Die Iteration beginnt mit dem „untersten“ Band und Index 0 und wird aufsteigend bis zum Band mit Index `size - 1` fortgesetzt.

Für die Operationen auf dem Cell-Prozessor und der SSE-Vektoreinheit wird zusätzlich der Zugriff auf die Zeiger der `SharedArrays` benötigt, um direkt auf den Daten operieren zu können. Dieser Zugriff wird bei dicht besetzten Containern über die Methode `elements()` ermöglicht. Dies gilt auch für `SparseVector`, nicht jedoch für `SparseMatrix`, da diese wie beschrieben nicht kontinuierlich im Speicher abgelegt ist.

3.1.2. CPU-Backend

Die CPU-Spezialisierung von HONEI *CPU-Backend* stellt die grundlegende Implementierung aller von HONEI zur Verfügung gestellten Operationen dar. Durch die Nutzung von generischen Klassen (*Templates*) und Iteratoren ist eine möglichst hohe Wiederverwendbarkeit sichergestellt. Ziel des Backends ist es, eine Referenz- und Vergleichsimplementierung für alle weiteren Backends zur Verfügung zu stellen. Das CPU-Backend ist als einziges auf allen unterstützten Plattformen lauffähig, durch den Verzicht auf Optimierungen auf Hardwareebene sind die darin enthaltenen Operationen allerdings langsamer als die für eine Plattform spezialisierten Operationen. Im Gegensatz zu den optimierten Varianten werden seitens des CPU-Backends die Operationen für nahezu alle Kombinationen von Containern implementiert. Dies schließt beispielsweise auch solche mit ein, deren Implementierung auf Vektoreinheiten wenig Optimierungspotential erwarten lassen würde. Gleichzeitig dient es aufgrund seiner Fülle an Sicherheitsmechanismen als Werkzeug für die Anwendungsentwicklung.

Implementierung

Zur Unterscheidung des jeweiligen Backends erhalten alle Operationen ein so genanntes *Architektur-Tag*. Die Implementierungen der Operationen (statische `value`-Methoden) des CPU-Backends sind daher mit dem Tag `tags::CPU` versehen und befinden sich direkt in den Header-Dateien, also zum Beispiel in `sum.hh`. Ruft man eine HONEI Operation ohne Angabe des *Architekturtags* auf, so wird automatisch das CPU-Backend ausgewählt.

Umsetzung ausgewählter Operationen

Im folgenden sollen die Ideen zur Umsetzung einiger Operationen des CPU-Backends näher erläutert werden:

Sum(DenseVector, DenseVector) Um zwei Vektoren beliebiger Länge zu addieren, werden die durch die Container zur Verfügung gestellten Iteratoren genutzt. Der folgende Codeausschnitt macht nahezu keinerlei Annahmen über die Art der Daten oder die Semantik der Operation und ist somit portabel und wiederverwendbar.

```
template <typename DT1_, typename DT2_>
static DenseVector<DT1_> &
    value(DenseVector<DT1_> & a, const DenseVector<DT2_> & b)
{
    if (a.size() != b.size())
        throw VectorSizeDoesNotMatch(b.size(), a.size());

    typename Vector<DT2_>::ConstElementIterator
        r(b.begin_elements());
    for (typename Vector<DT1_>::ElementIterator
        l(a.begin_elements()), l_end(a.end_elements()) ;
            l != l_end ; ++l)
    {
        *l += *r;
        ++r;
    }

    return a;
}
```

3.1.3. SSE-Backend

Das SSE-Backend ist ein direkter Ersatz für das in Abschnitt 3.1.2 vorgestellte CPU-Backend. Es besitzt ebenfalls keinerlei Funktionalität, um mehrere Prozessoren, Kerne oder SPEs zu nutzen. Im Gegensatz zum CPU-Backend verzichtet es auf den Einsatz von Iteratoren und wurde im Hinblick auf eine hohe Geschwindigkeit entwickelt. Hierzu bedient es sich der in modernen x86 Prozessoren vorhandenen Vektoreinheiten.

Technische Grundlagen

MMX, SSE, SSE2 und diverse Nachfolger sind Erweiterungen des x86 Befehlssatzes um nach dem SIMD (*Single Instruction Multiple Data*)-Prinzip denselben Befehl gleichzeitig auf einer Menge von Daten ausführen zu können. Hierzu stehen laut Intel [Int07] auf aktuellen x86 Architekturen acht Register zur Verfügung, die 128 Bit groß sind, also zum Beispiel vier Fließkommazahlen vom Typ `float` oder zwei vom Typ `double` beinhalten können. Da auf kontinuierlichen Folgen von Daten gearbeitet wird, ist es notwendig, dass diese Daten auch kontinuierlich im Speicher vorliegen. Aus diesem Grund konnten nur Operationen auf Daten dicht besetzter Container oder elementweise Operationen auf Daten dünn besetzter Container implementiert werden. Bei anderen Containern kann ohne größeren Aufwand immer nur ein Skalar als nächster Operand von einer Speicherstelle geladen werden, so dass eine vektorisierte Bearbeitung keinen Sinn ergibt. Moderne C++ Compiler sind zwar in der Lage, die SSE Einheiten einer CPU selbstständig zu nutzen, dies kann eigenen Untersuchungen zur Folge bisher eine manuelle Optimierung allerdings nicht ersetzen.

Implementierung

Die Operationen des SSE-Backends sind mit dem Architektur-Tag `tags::CPU::SSE` versehen und befinden sich in Dateien wie `sum-sse.cc`. Sie werden nur zu HONEI hinzu kompiliert, wenn das Konfigurationprogramm der verwendeten *GNU Autotools* `configure` mit dem Parameter `enable-sse` ausgeführt wurde. Um die Entwicklungszeit für ein zusätzliches Backend in einem vertretbaren Rahmen zu halten, wurde auf die Verwendung von Assembler und den damit verbundenen Möglichkeiten noch tiefergehender Optimierungsmöglichkeiten verzichtet. Die SSE Einheit wird stattdessen im Programmcode nicht direkt über die Maschinenbefehle in Assembler angesprochen sondern über korrespondierende Intrinsics, die selbige in C++ kapseln und somit deutlich einfacher zu verwenden sind.

Umsetzung ausgewählter Operationen

Im folgenden sollen die Ideen zur Umsetzung einiger Operationen in SSE näher erläutert werden:

Sum(DenseVector, DenseVector) Bevor die Operation einer elementweisen Addition von zwei kontinuierlich im Speicher liegender Datenfelder vorgenommen werden kann, sind einige Vorarbeiten notwendig, die so in beinahe allen SSE-Operationen vorkommen.

```
DenseVector<float> & Sum<tags::CPU::SSE>::
    value(DenseVector<float> & a, const DenseVector<float> & b)
{
    if (a.size() != b.size())
        throw VectorSizeDoesNotMatch(b.size(), a.size());

    intern::sse::sum(a.elements(), b.elements(), a.size());

    return a;
}
```

In der von einer Applikation aufgerufenen `value`-Methode passiert wenig mehr als das Überprüfen der Vektorgrößen und die Extraktion der Zeiger auf die eigentlichen Daten. Diese werden an die interne Methode `intern::sse::sum` übergeben, die im folgenden näher betrachtet werden soll.

```
inline void sum(float * a, const float * b, unsigned long size)
{
    __m128 m1, m2, m3, m4, m5, m6, m7, m8;

    unsigned long a_address = reinterpret_cast<unsigned long>(a);
    unsigned long a_offset = a_address % 16;
    unsigned long b_address = reinterpret_cast<unsigned long>(b);
    unsigned long b_offset = b_address % 16;

    unsigned long x_offset(a_offset / 4);
    x_offset = (4 - x_offset) % 4;

    unsigned long quad_start = x_offset;
    unsigned long quad_end(size - ((size - quad_start) % 16));

    if (size < 24)
    {
        quad_end = 0;
        quad_start = 0;
    }
}
```

Zunächst werden acht Variablen deklariert, die vom Compiler auf die acht SSE Register gelegt werden sollen. Danach werden die Adressen der Operanden `a` und `b` bezüglich ihrer Position im Speicher untersucht, da in SSE Register ohne Mehraufwand nur an 16-Byte Grenzen ausgerichtete Daten geladen werden können. Das Zurückschreiben nicht ausgerichteter Daten ist mit noch größerem Mehraufwand verbunden. Da das Ergebnis der Addition später wieder in den Operanden `a` geschrieben werden soll, wird ein Versatz (*Offset*) errechnet und in der Variablen `quad_start` die nächste 16-Byte Grenze gespeichert. Die Variable `quad_end` kennzeichnet das Ende der SSE-Phase, da danach nur noch weniger Elemente zu berechnen sind als in ein SSE Register passen. Ist der gesamte Vektor kleiner als 24 Elemente, ist von der Verwendung von SSE kein Geschwindigkeitsvorteil zu erwarten und die Operation wird vollständig skalar berechnet.

```
if (a_offset == b_offset)
{
    for (unsigned long index(quad_start) ;
        index < quad_end ; index += 16)
    {
        m1 = _mm_load_ps(a + index);
        m3 = _mm_load_ps(a + index + 4);
        m5 = _mm_load_ps(a + index + 8);
        m7 = _mm_load_ps(a + index + 12);
        m2 = _mm_load_ps(b + index);
        m4 = _mm_load_ps(b + index + 4);
        m6 = _mm_load_ps(b + index + 8);
        m8 = _mm_load_ps(b + index + 12);

        m1 = _mm_add_ps(m1, m2);
        m3 = _mm_add_ps(m3, m4);
        m5 = _mm_add_ps(m5, m6);
        m7 = _mm_add_ps(m7, m8);

        _mm_store_ps(a + index, m1);
        _mm_store_ps(a + index + 4, m3);
        _mm_store_ps(a + index + 8, m5);
        _mm_store_ps(a + index + 12, m7);
    }
}
```

Nun wird die eigentliche Berechnung durchgeführt. Falls die Operanden `a` und `b` denselben Offset besitzen, ihre Addressen also einen identischen Abstand von der jeweils nächsten 16-Byte Speichergrenze besitzen, können beide regulär in die SSE Register geladen werden, darin addiert und danach das Ergebnis wieder in `a` gespeichert werden. Die Schleife ist vierfach entrollt, um die acht SSE Register vollständig auszunutzen.

```
for (unsigned long index(quad_start) ;
    index < quad_end ; index += 16)
{
    m1 = _mm_load_ps(a + index);
    m3 = _mm_load_ps(a + index + 4);
    m5 = _mm_load_ps(a + index + 8);
    m7 = _mm_load_ps(a + index + 12);
    m2 = _mm_loadu_ps(b + index);
    m4 = _mm_loadu_ps(b + index + 4);
    m6 = _mm_loadu_ps(b + index + 8);
    m8 = _mm_loadu_ps(b + index + 12);

    m1 = _mm_add_ps(m1, m2);
    m3 = _mm_add_ps(m3, m4);
    m5 = _mm_add_ps(m5, m6);
    m7 = _mm_add_ps(m7, m8);

    _mm_store_ps(a + index, m1);
    _mm_store_ps(a + index + 4, m3);
    _mm_store_ps(a + index + 8, m5);
    _mm_store_ps(a + index + 12, m7);
}
```

Falls die Operanden `a` und `b` nicht mit demselben Offset im Speicher liegen, werden die Daten von `b` mit Hilfe des Befehls `_mm_loadu_ps` geladen, um dem Prozessor anzuzeigen, dass sie nicht ausgerichtet im Speicher liegen. Die Daten des Operanden `a` können so weiterhin ausgerichtet geladen und das Ergebnis wieder in `a` ausgerichtet geschrieben werden.

```
for (unsigned long index(0) ; index < quad_start ; index++)
{
    a[index] += b[index];
}

for (unsigned long index(quad_end) ; index < size ; index++)
{
    a[index] += b[index];
}
}
```

Zuletzt werden die Daten am Anfang und am Ende der Vektoren skalar addiert, die nicht per SSE verarbeitet wurden.

Product(DenseMatrix, DenseMatrix) Um möglichst viele Operationen auf Daten im Cache ausführen zu können, werden die Matrizen zunächst so lange zerteilt, bis sie vollständig in den Cache der CPU passen. Erst im Anschluss wird tatsächlich auf diesen Teilstücken gerechnet. Hierzu wird eine Matrizenzerlegung verwendet, wie sie von Prokop [Pro99] vorgeschlagen wird. Ziel ist es, bei dem Produkt einer $m \times n$ Matrix A mit einer

$n \times p$ Matrix B immer die größere der beiden in zwei Matrizen zu teilen. Es sind drei Fälle zu unterscheiden:

$$AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}, \tag{3.1}$$

$$AB = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2, \tag{3.2}$$

$$AB = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}. \tag{3.3}$$

Im Fall 3.1 gilt $m \geq n$ und $m \geq p$. Matrix A wird horizontal halbiert und beide Hälften werden mit der Matrix B multipliziert. Im Fall 3.2 gilt $n \geq m$ und $n \geq p$. Beide Matrizen werden geteilt, jeweils multipliziert und danach werden die Teilergebnise aufaddiert. Der letzte Fall 3.3 tritt ein, wenn $p \geq m$ und $p \geq n$ gilt. Die Matrix B wird vertikal halbiert und im Anschluss wird die Matrix A mit beide Hälften multipliziert.

Einen Sonderfall stellt das Produkt einer $n \times n$ Matrix mit einer $n \times 2$ Matrix dar. Dieses kommt häufig in Layout-Verfahren für Graphen vor, bei denen $n \times 2$ Matrizen in sogenannte Positionsmatrizen verwendet werden. Hier wäre es nicht sinnvoll, wie oben beschrieben vorzugehen, da jede Zeile nur zwei Elemente lang ist. Da die Spaltenzahl der $n \times 2$ Matrix bekannt ist, wird hier wiederum auf das bekannte Schema der Form „Zeile mal Spalte“ zurück gegriffen. Hierzu wird intern für jede zwei-elementige Zeile der Ergebnismatrix, jeweils das Skalarprodukt einer Zeile der $n \times n$ Matrix mit beiden Spalten der $n \times 2$ Matrix berechnet.

Product(BandedMatrix, DenseVector) Um den Programm- und Datenfluss möglichst stetig und unterbrechungsfrei zu halten, wurde entschieden, dass die Operation den Bändern der Matrix folgt und das Teilergebnis der Multiplikation jedes Bandes mit dem Operandenvektor auf den Ergebnisvektor aufaddiert wird. Da die Bänder intern als `DenseVector` mit identischer Länge gespeichert und zum Ende hin mit Nullen aufgefüllt sind, berechnet man anhand der Position des Bandes in der Matrix den Einstiegs- und Endpunkt der Berechnung. Der sogenannte *Offset* des Bandes gibt auch an, welcher Bereich des Operandenvektors mit dem Band verrechnet werden muss.

ElementInverse Da die Operation `ElementInverse` selbstständig eine Division durch Null und die damit verbundene Entstehung ungültiger Werte vermeiden muss, ist hierfür ein Algorithmus ohne teure Verzweigung im Programmablauf notwendig. Die Lösung für das SSE-Backend besteht darin, zunächst einmal ohne Abfrage auf Nullelemente jedes Element zu invertieren und erst im Anschluss an jeder Stelle, an der vorher eine Null in der Eingabe stand wieder eine Null einzufügen

3.1.4. MultiCore-Backend

Während sowohl das CPU-Backend als auch das SSE-Backend lediglich einen einzelnen CPU-Kern ausnutzen (im folgenden werden diese auch als *SingleCore-Backends* bezeich-

net), ermöglicht es das MultiCore-Backend die Rechenlast auf alle CPU-Kerne eines Systems zu verteilen, um dadurch Mehrkern- bzw Mehrprozessor-Architekturen auszunutzen. Es teilt die Probleme dabei in Teilprobleme auf und bedient sich zur Lösung dieser der Implementierungen der CPU-Referenzimplementierung und des SSE-Backends.

Technische Grundlagen

Als Basis für paralleles Programmieren gibt es verschiedene Ansätze. Im Rahmen der Projektgruppe wurden *OpenMP* und *POSIX Threads* in Erwägung gezogen. Die Wahl fiel schließlich auf POSIX Threads (oder kurz: *Pthreads*), da man sich hierbei die größte Flexibilität versprach. Weiterhin musste entschieden werden, ob leichtgewichtige Prozesse (im weiteren als *Threads* bezeichnet) nach Bedarf dynamisch erzeugt werden sollten oder ob ein Pool von Threads bestehen soll. Man entschied sich für Letzteres, da auf diese Weise die Zusatzlast des Erzeugens und Zerstörens von Threads vermieden wird. Zudem können Threads, welche keine Berechnungen ausführen, in einen Wartezustand versetzt werden, so dass sie das System nicht unnötig belasten.

Um die Schnittstelle des Pools einfach und einheitlich zu halten nimmt dieser nur Funktionen entgegen, welche keine Parameter und den Rückgabetyp `void` besitzen. Eine beliebige Funktion kann in diese Form unter Verwendung zweier Hilfsmittel gebracht werden: zum einen durch Funktionalitäten des *TR1*[2], zum anderen mittels einer Menge von Kapselungsklassen (*Wrapper*-Klassen).

Der TR1 bietet unter anderem drei Funktionalitäten, welche intensiv im MultiCore-Backend genutzt werden: `std::tr1::bind` ermöglicht es, eine Funktion an feste Argumente zu binden. Im Falle des Pools wird eine Funktion an alle Argumente gebunden, so dass eine neue Funktion entsteht, welche den gleichen Rückgabetyp jedoch keinerlei Parameter mehr besitzt. Mit `std::tr1::function` lassen sich Container für Funktionen definieren, welche in diesem Fall dazu genutzt werden, die durch den `bind`-Aufruf erzeugten Funktionen aufzunehmen. Die dritte Funktionalität ist `std::tr1::shared_ptr`, welche sogenannte intelligente Zeiger (*smart pointer*) umsetzt. Durch die Verwendung dieser intelligenten Zeiger vereinfacht sich der Umgang mit dem Backend abermals.

Um den Rückgabetyp einer Funktion zu ändern musste eine eigene Lösung gefunden werden: eine Menge von *Wrapper*-Klassen. Mit Hilfe dieser Klassen ist es nicht nur möglich, eine Funktion bezüglich ihres Rückgabewertes an die Schnittstelle des Pools anzupassen, sondern auch bezüglich der Argumente. Dies erwies sich als vorteilhaft, da `std::tr1::bind` Probleme im Umgang mit generisch implementierten Containern zeigte.

Implementierung

Sämtliche Implementierungen der Operationen des MultiCore-Backends sind in Dateien mit der Endung `-mc`, also beispielsweise `sum-mc.hh` zu finden. Die Implementierungen, welche auf dem CPU-Backend aufsetzen, werden über das Tag `tags::CPU::MultiCore` angesprochen, während die auf dem SSE-Backend aufsetzenden Implementierungen über

[2] Der TR1 (*Technical Report 1*) ist eine Spezifikation des C++-Standardisierungskomitees, in dem Erweiterungsvorschläge zur Standard-Bibliothek gesammelt und getestet werden ([The05]).

das Tag `tags::CPU::MultiCore::SSE` aufgerufen werden. Letztere müssen, wie bereits die Operationen des SingleCore-SSE-Backends, gesondert hinzukompiliert werden.

Die Infrastruktur des MultiCore-Backends setzt sich insgesamt aus 4 Klassen zusammen, welche im weiteren näher erläutert werden. Zunächst eine kurze Übersicht:

- `PoolTask`: Ein Objekt der Klasse `PoolTask` kapselt eine Arbeitsanfrage (im weiteren nur noch als *Job* bezeichnet), welche an den `ThreadPool` übergeben wurde. Mit Hilfe eines solchen Objektes ist es möglich, auf die Abarbeitung eines dem `ThreadPool` übergebenen Jobs zu warten.
- `PoolThread`: Objekte der Klasse `PoolThread` realisieren jeweils einen Thread, welcher von der Klasse `ThreadPool` erzeugt und verwaltet wird.
- `ThreadPool`: Diese Klasse stellt die zentrale Instanz des MultiCore-Backends dar. Erzeugte Jobs werden an den `ThreadPool` geschickt, welcher diese dann entweder an einen verfügbaren `PoolThread` übergibt oder zur Ausführung vormerkt, sollte kein `PoolThread` verfügbar sein. Für jeden Job wird ein `PoolTask` erzeugt.
- Wrapper: Ein Wrapper wird benötigt um einen Job vorzubereiten, damit er dem `ThreadPool` übergeben werden kann.

Im folgenden wird auf die Komponenten im Detail eingegangen.

Die Wrapper-Klassen Um einen Job mit Hilfe des MultiCore-Backends abarbeiten zu können, muss dieser zunächst geeignet vorbereitet werden. Dies realisieren die *Wrapper*-Klassen, wobei zu beachten ist, dass für jede Anzahl von Argumenten (derzeit bis zu 5) eine eigene Wrapperklasse existiert. Diese Klassen gibt es jeweils in zwei Ausführungen, wobei der Name der Klasse auf die Variante schließen läßt: zum einen für Funktionen, welche als Ergebnis mindestens eines der übergebenen Argumente ändern, wie z.B. in der Operation `Sum`, bei der die Klasse `TwoArgWrapper` benutzt wird. Zum anderen für Funktionen, die ein neues Objekt als Resultat liefern, wie z.B. die Operation `Reduction` welche u.a. die Klasse `ResultOneArgWrapper` nutzt. Da sämtliche Funktionen der SingleCore-Backends `value` genannt wurden beschränkte man sich bei den *Wrapper*-Klassen darauf, dass nur `value` genannte Funktionen gekapselt werden können, wobei dies die Komplexität der *Wrapper*-Klassen etwas verminderte.

Die Klasse `PoolTask` Ein Objekt der Klasse `PoolTask` repräsentiert einen Job, der an den `ThreadPool` übergeben wurde. Es kapselt die übergebene Funktion und bietet die Möglichkeit auf die Abarbeitung der Aufgabe zu warten. Dieses ist ein blockierendes Warten, so dass, wenn ein Nutzer in seinem Programm auf die Abarbeitung des Jobs wartet, sein Programm an dieser Stelle erst fortfährt, wenn der Job fertig bearbeitet wurde. Objekte dieser Klasse werden direkt vom `ThreadPool` erzeugt, wenn diesem ein Job übergeben wurde.

Die Klasse `ThreadPool` Diese Klasse ist nach dem sog. *Singleton*-Entwurfsmuster implementiert, d.h. dass von dieser Klasse maximal ein Objekt erzeugt werden kann. Der `ThreadPool` ist die zentrale Instanz des MultiCore-Backends. Jegliche Jobs, welche durch das MultiCore-Backend ausgeführt werden sollen, werden zunächst dem `ThreadPool` übergeben. Dieser verwaltet eine Liste von wartenden Threads. Gibt es einen wartenden Thread, so wird diesem der neue Job direkt zur Ausführung übergeben. Ist diese Liste leer, wird der Job in die Warteschlange eingefügt und ausgeführt, sobald ein Thread die Ausführung eines Jobs beendet hat. Die Anzahl der Threads im `ThreadPool` kann nach der Erstellung desselben nicht mehr verändert werden, jedoch ist es möglich die Anzahl bei der Erstellung vorzugeben.

Die Klasse `PoolThread` Objekte der Klasse `PoolThread` realisieren jeweils einen Thread, welcher vom `ThreadPool` erzeugt und verwaltet wird. Ein `PoolThread` bearbeitet immer nur einen Job und meldet sich dann wieder dem `ThreadPool` als bereit zurück. Sollte kein Job zur Ausführung bereit stehen, wird der `PoolThread` in einen Wartezustand versetzt und bei Bedarf reaktiviert.

Neben diesen Klassen wird Gebrauch von zwei weiteren Klassen gemacht, der Klasse `Configuration` und der Klasse `Partitioner`. Beide sind nicht unbedingt notwendig, erleichtern die Implementierung von Operationen für das MultiCore-Backend allerdings erheblich. Die Klasse `Configuration` bietet eine zentrale Anlaufstelle für Parameter, welche vom Backend benötigt werden. Die Parameter werden dabei in einer zentralen Datei abgelegt. So bezieht zum Beispiel der `ThreadPool` die Information über die Anzahl der zu erstellenden Threads über die `Configuration`. Die Klasse `Partitioner` wird dazu genutzt, Größen aufzuteilen. Da dies in allen Operationen des Backends nötig ist um Teilprobleme zu erzeugen, wurde diese Routine zentralisiert. Auf die Einstellungsmöglichkeiten wird hier nicht eingegangen, da es nicht zum Verständnis des Backends beiträgt.

Umsetzung ausgewählter Operationen

Im folgenden sollen die Ideen zur Umsetzung einiger Operationen für das MultiCore-Backend näher erläutert werden:

Sum(DenseVector, DenseVector) Diese Operation ist trivial zu implementieren, eignet sich jedoch dafür, den allgemeinen Ablauf eines Aufrufes des MultiCore-Backends exemplarisch nachzuvollziehen:

```
DenseVector<DT1_> & Sum<Tag_>::
    value(DenseVector<DT1_> & a,
    const DenseVector<DT2_> & b)
{
    if (a.size() != b.size())
        throw VectorSizeDoesNotMatch(b.size(), a.size());

    unsigned long min_part_size(
        Configuration::instance()->get_value(
            "mc::sum[DV,DV]::min-part-size", 1024));
    unsigned long overall_size(a.size());

    if (overall_size < 2 * min_part_size)
    {
        return Sum<typename Tag_::DelegateTo>::value(a,b);
    }
```

Im ersten Abschnitt der Operation werden zunächst die Vektorgrößen überprüft. Des weiteren wird sich der Klasse `Configuration` bedient, um eine für die Operation spezifische Mindestgröße für Teilprobleme auszulesen (im Beispiel ist dies `min-part-size`, der zweite Parameter des Aufrufes beschreibt den Standardwert, falls der angegebene Eintrag nicht existiert). Ist das Problem an sich bereits zu klein, um es lohnend in zwei Teilprobleme aufzuteilen, wird es komplett an das entsprechende SingleCore-Backend weitergereicht, um den Aufwand des MultiCore-Backends zu vermeiden.

```
    else
    {
        unsigned long num_threads(
            Configuration::instance()->get_value(
                "number-of-threads", 2));
        unsigned long max_count(
            Configuration::instance()->get_value(
                "mc::sum[DV,DV]::max-count",
                num_threads));

        PartitionList partitions;
        Partitioner<tags::CPU::MultiCore>(
            max_count, min_part_size, 16,
            overall_size,
            PartitionList::Filler(partitions));
```

Ist die Problemgröße derart, dass ein Aufteilen lohnt, so werden weitere Parameter herangezogen, wie z.B. eine Maximalzahl von Teilproblemen (im Code-Beispiel ist dies `max_count`) in die unterteilt werden soll. Diese Informationen werden für den Aufruf der Partitionierungsroutine genutzt. Das Resultat ist eine Liste mit Startindizes und Größen der Teilprobleme, welche im weiteren abgearbeitet wird.

```
        std::list< std::tr1::shared_ptr<PoolTask> >
            dispatched_tasks;
        unsigned long offset, part_size;

        for (PartitionList::ConstIterator p(partitions.begin()),
                p_end(partitions.end()); p != p_end; ++p)
        {
            offset = p->offset;
            part_size = p->size;
            DenseVectorRange<DT1_>
                range_1(a.range(part_size, offset));
            DenseVectorRange<DT2_>
                range_2(b.range(part_size, offset));
            TwoArgWrapper<Sum<typename Tag_::DelegateTo>,
                DenseVectorRange<DT1_>,
                DenseVectorRange<DT2_> >
                    mywrapper(range_1, range_2);
            std::tr1::shared_ptr<PoolTask>
                ptr(ThreadPool::instance()->dispatch(mywrapper));
            dispatched_tasks.push_back(ptr);
        }
```

Wie bereits erwähnt werden bei der Übergabe von Jobs an den Pool Objekte der Klasse `PoolTask` zurückgegeben. Daher wird zunächst eine Liste erzeugt, welche diese Objekte aufnimmt, da sie am Ende der Operation noch benötigt werden. Auf Basis der Liste der Teilproblemgrößen wird das Problem nun aufgeteilt. Die Klasse `DenseVectorRange` realisiert einen speziellen Container, welcher ein Teilstück eines `DenseVector` beschreibt dessen Einträge kontinuierlich im Speicher liegen. Nun wird ein Objekt der Klasse `TwoArgWrapper` erstellt. Der erste Template-Parameter bezeichnet die Operation, welche ausgeführt werden soll, alle weiteren beschreiben die Typen der Argumente. Der Template-Parameter `Tag_::DelegateTo` sorgt dafür, dass die Implementierung des entsprechenden SingleCore-Backends aufgerufen wird. Als nächstes wird das *Wrapper*-Objekt, bzw genauer der `operator()` des *Wrapper*-Objektes, übergeben. Mit dem durch diesen Aufruf erhaltenen Objekt der Klasse `PoolTask` wird ein *smart pointer* initialisiert, welcher anschliessend in die entsprechende Liste eingetragen wird.

```
        while(! dispatched_tasks.empty())
        {
            dispatched_tasks.front()->wait_on();
            dispatched_tasks.pop_front();
        }
    }
    return a;
}
```

Abschließend wird sukzessive auf die Abarbeitung der Teilprobleme gewartet, wobei der erzeugende Thread inaktiv wird und die so frei werdende Rechenkapazität für die Bearbeitung der Teilprobleme zur Verfügung steht.

Product(BandedMatrix, DenseVector) Eine direkte Abarbeitung entlang der Bänder der Matrix würde jeweils exklusives Schreibrecht der erzeugten `PoolTask`-Objekte beim Aufaddieren der Teilergebnisse auf den Ergebnisvektor erfordern. Um diesen Mehraufwand zu vermeiden und dennoch die Struktur des Problems vor allem in Bezug auf die Kontinuität der Daten auszunutzen, arbeiten die Teilprobleme auf unabhängigen Bereichen des Ergebnisvektors. Dies geschieht durch die Verwendung einer internen Methode der folgenden Form:

```
MCProduct<Tag_>::value(
    DenseVectorRange<DT1_> & res_range,
    const BandedMatrix<DT1_> & a,
    const DenseVectorRange<DT2_> & b,
    unsigned long offset,
    unsigned long size)
```

Sie realisiert die zeilenweise Unterteilung der Matrix, indem nur Abschnitte der Bänder betrachtet werden, die zu dem jeweiligen Teilstück des Ergebnisvektors korrespondieren und durch die übergebenen Parameter `offset` und `size` beschränkt sind. Hierbei wird mit Hilfe von `NonZeroBandIterator` über alle Bänder iteriert, jedoch nur mit denjenigen gerechnet, die zumindest partiell im zugehörigen Abschnitt der Matrix liegen und entsprechend nur auf den relevanten Stücken des Bandes. Dieses Vorgehen erweist sich auch im Sinne der Unterteilung des Gesamtproblems als günstiger, da den Teilaufrufen nicht explizit eine Menge von zu bearbeitenden Bändern bzw. Stücke von Bändern mit geeigneter Größe übergeben werden muss.

Product(DenseMatrix, DenseMatrix) MultiCore nutzt die gleiche Matrizenzerlegung, wie sie bereits im SSE-Backend beschrieben wurde, um Teilprobleme einer vorher einstellbaren Maximalgröße zu erhalten. Erst diese werden dann entsprechend der Anzahl an Threads im `ThreadPool` weiter unterteilt, um schließlich parallel abgearbeitet zu werden. Die bei der Matrizenzerlegung entstandenen Teilprobleme werden allerdings sequentiell bearbeitet, d.h. es wird jeweils auf die vollständige Abarbeitung eines Teilproblems gewartet bis mit dem nächsten begonnen wird, um Cache-Koherenz zu gewährleisten. Hieraus folgt eine von der Maximalgröße abhängige Synchronisation der Threads auf den Daten, die aber zwangsläufig auch Leerlaufzeiten einzelner Threads mit sich bringen kann.

3.1.5. Cell-Backend

Die SPEs sind als reine Recheneinheiten konzipiert und nicht in der Lage komplexere Steuerungsaufgaben, wie z.B. das Ausführen eines Betriebssystems, zu übernehmen. Daher ist es nötig, die Ausführung von Programmen auf den SPEs durch das PPE zu steuern und zu synchronisieren. Das Cell-Backend ist ein komplexes *Framework*, welches einerseits alle nötigen Komponenten bereitstellt, um ein Programm auf einer oder mehreren SPEs zur Ausführung zu bringen. Andererseits ermöglicht es die vollständige Verwaltung und Kontrolle des Ablaufes und der Kommunikation, sowie die Nutzung ei-

niger architektur-spezifischer Konzepte, wie z.B. den in Abschnitt 2.1 angesprochenen DMA-Listentransfers.

Überblick über die wesentlichen Elemente des Cell Backends

Operationenkernel Eine wichtige Aufgabe ist zu ermöglichen, dass ein SPE mehrere Funktionen nacheinander ausführen kann, ohne jedes Mal den kompletten Inhalt des lokalen Speichers austauschen zu müssen. Aus diesem Grund wurden im Rahmen der Projektgruppe einige Operationenkernel entwickelt. Ein Operationenkernel beinhaltet eine Auswahl implementierter Operationen und wird im lokalen Speicher der einzelnen SPEs abgelegt, um deren Ausführung zu erlauben. Er entspricht also sozusagen dem Hauptprogramm, welches auf einem SPE läuft und von dem aus einige der unterstützten Operationen aufgerufen werden können. Einem Kernel werden zusätzlich alle wesentlichen technischen Argumente übergeben, die zum Start eines SPE-Programms benötigt werden.

Im Rahmen der Projektgruppe wurden mehrere solche Operationenkernel vorimplementiert, insbesondere ein `float`- und ein `double`-Referenzkernel mit allen für den Cell-Prozessor implementierten Operationen sowie Kernel, die die Operationen enthalten, die für die Anwendungsfälle `libSWE` und `libGraph` erforderlich sind.

Ein wesentlicher Grund für die Nutzung von Kerneln ist das Einsparen von Speicherplatz im auf 256 kB beschränkten lokalen Speicher, um mehr Platz für Nutzdaten zu haben, wenn nur einige wenige Operationen benötigt werden. Außerdem muss nicht für jede Operation ein neues SPE-Programm geladen werden, wodurch unnötiger Zeitverlust vermieden wird. Dies bedeutet darüber hinaus keine wesentliche Einschränkung, da der Nutzer zur Entwicklungs-/Kompilierzeit weiß, welche Operationen später aufgerufen werden sollen.

SPEInstruction Wie bereits beschrieben, kapseln Operationenkernel mehrere Operationen. Eine einzelne Anwendungsoperation hat im Rahmen der Projektgruppe den Namen `SPEInstruction`. Eine `SPEInstruction` besitzt einen `Opcode` und einige anwendungsspezifische Funktionsparameter, mit denen der Aufruf der entsprechenden Operation innerhalb des SPE-Hauptprogramms realisiert wird. Da viele Operationen sich nicht in ihrem Aufbau, sondern nur im Aufruf eines oder mehrerer *Intrinsics* unterscheiden, wurde zusätzlich das Konzept der `SPEFrameworkInstruction` implementiert. Es bietet eine Standardschnittstelle für sich ähnelnde Operationen. Ein Beispiel ist die Klasse `SPEFrameworkInstruction<2, float, rtm_dma>`, das den typischen PPE-Teil einer Operation darstellt, welche `2` Container verwendet, deren Datentyp `float` ist, und die das Ergebnis per DMA-Transfer zurück in den PPE-Hauptspeicher schreibt, wie etwa eine Vektorsumme.

SPEManager Der SPEManager ist die zentrale Verwaltungseinheit für die SPEs. Er verwaltet Objekte für jedes SPE auf dem PPE und erlaubt das *Dispatching* übergebener

Instanzen von `SPEInstruction`.

SPEKernel Der PPE-seitige `SPEKernel` übernimmt die komplette Kommunikation mit den SPEs. Zur Programmlaufzeit existiert also für jedes SPE ein entsprechendes Objekt auf dem PPE. Zunächst wird dem `SPEKernel` ein Operationenkernel übergeben, welcher dann durch Aufruf der Funktion `load()` in den lokalen Speicher des SPEs geschrieben wird. Somit sind die unterstützten `Opcodes` seitens des PPEs bekannt und es können bei Bedarf andere Kernel in ein SPE geladen werden, falls eine Operation angefordert wird, die vom aktuellen Operationenkernel nicht unterstützt wird.

Der erzeugte `kernel_thread` wartet auf ankommende Mailbox-Nachrichten und Signale der SPEs, verarbeitet sie und gibt sie (im Debug-Modus) aus. Dies gilt z.B. auch für skalare Ergebnisse einiger Reduktionen, die nicht per DMA-Transfer übertragen, sondern vom SPE über das Mailbox-System zum PPE versandt werden, ebenso wie für die Nachricht des SPEs, dass die Ausführung des SPE-Programms abgeschlossen ist.

SPETransferList Wie in Abschnitt 2.3 beschrieben wurde, erlaubt der Cell Prozessor neben einfachen DMA-Transfers auch die Nutzung so genannter DMA-Listentransfers. `SPETransferList` bietet die notwendige Funktionalität, um solche Transferlisten zu erstellen. Jedes Element einer Liste kann dabei ein normaler DMA-Transfer mit den üblichen Nebenbedingungen (d.h. max. 16 kB Transfergröße, Ausrichtung im Speicher) sein. Mit Hilfe dieses Konstrukts wird es erst möglich, nicht-kontinuierliche Speicherbereiche aus dem PPE-Hauptspeicher in einen kontinuierlichen Speicherbereich des lokalen Speichers des SPEs zu übertragen. Dies ist inbesondere für Container wie `DenseMatrix`, deren Daten kontinuierlich im Speicher abgelegt sind, sehr nützlich, wenn man etwa nur einige Spalten der Matrix übertragen möchte.

Technische Grundlagen

Das Cell-Backend übernimmt alle notwendigen Aufgaben, um ein Programm auf den SPEs zur Ausführung zu bringen und von deren Geschwindigkeit sowie von Vektorisierung profitieren zu können.

Das SPE-*Instruction Set* wurde vom existierenden Altivec-Befehlsatz früherer PowerPC-Prozessoren abgeleitet, weist jedoch diverse Unterschiede auf. Die wichtigsten Assembler-Befehle sind für den GCC-Compiler bereits in Form von *Intrinsics* nutzbar. Zum Teil fanden sich an dieser Stelle aber auch Fehler, d.h. *GCC-Intrinsics* oder deren Operanden stimmten nicht mit denen im aktuellen *SPE Instruction Set Architecture*-Handbuch überein. Allgemein werden beim Vergleich des Cell-Prozessors mit x86-Prozessoren Unterschiede zwischen RISC- und CISC-Befehlsätzen deutlich. Beispielsweise erlaubt das SPE eine *intrinsic*-basierte Ausführung von Multiplikationen lediglich für die Datentypen `float` und `double`. Die Integer-Multiplikation muss hingegen per manuellen Assemblerbefehlen aus mehreren Vektoroperationen zusammengesetzt werden.

Ein weiterer wesentlicher Unterschied in der Programmierung beider Architekturen ist die dem Programmierer erlaubte Flexibilität bei der Verwendung von Speichertransfers.

Während selbst die SSE-Einheit der x86-Prozessoren, wenn auch mit geringer Verzögerung, das Laden von nicht am Speicher ausgerichteten Datenpaketen erlaubt, verweigert der *MFC-Controller* der SPEs jegliche Ausführung von Speichertransfers, die nicht:

- an 16 Byte Grenzen ausgerichtete Adressen sowohl auf Hauptspeicher- als auch *Local Store*-Seite besitzen
- eine Transfergröße haben, die ein Vielfaches von 16 Bytes ist und höchstens 16 kB beträgt.

Lediglich bei sehr kleinen Transfers von weniger als 16 Byte reicht ein natürliches *Alignment*. Dies bedeutet beispielsweise die Ausrichtung an einer 8-Byte Adresse für einen Transfer von 8 Bytes.

Mit Hilfe von `SPETransferList` und häufiger Nutzung von *Shuffle-Intrinsics* (vgl. die HONEI-Funktionen `extract` und `insert`) konnte bei *Dense-Containern* auch im Falle von Containergrößen, die kein Vielfaches von 16 Byte sind eine sehr performante Implementierung entwickelt werden. Wenn beispielsweise eine Zeile einer `DenseMatrix` an einer nicht an 16-Byte ausgerichteten Speicheradresse liegt, wird die nächstniedrigere ausgerichtete Adresse für den Datentransfer in den lokalen Speicher des SPEs verwendet. Anschließend wird bei der vektorisierten Iteration durch den Datenblock pro Iterationsschritt eine Verschiebung der Daten des aktuellen und nachfolgenden SIMD-Vektors vorgenommen.

Ein Beispiel: Die angesprochene Zeile beginne erst zwei Elemente nach einer ausgerichteten Adresse. Nach dem Transfer enthält der erste SIMD-Vektor im Falle von einfacher Genauigkeit also zwei Elemente, die nicht zur Zeile gehören. Der darauffolgende Vektor enthält aber nur gültige Elemente. Daher wird nun eine Verschiebeoperation um zwei Elemente angewandt, so dass der später verarbeitete SIMD-Vektor vier gültige Elemente enthält. Dieses Vorgehen kann man bis zum Ende des Datenblockes fortsetzen, wenn man die Größe des Transfers um einen weiteren SIMD-Vektor erhöht. Die Kosten entsprechen einer zusätzlichen Vektoroperation. Insbesondere in den Fällen, in denen die Laufzeit durch die Geschwindigkeit der Speichertransfers dominiert wird, ist dieser minimale *Overhead* vernachlässigbar.

Für die Kombination von verschiedenen Containertypen, also z.B. das Produkt einer `BandedMatrix` und einer `SparseMatrix`, sind die beschriebenen Einschränkungen aber zu hoch, um Vektoroperationen anwenden zu können, insbesondere weil der lokale Speicher und die Größe eines DMA-Transfers stark beschränkt sind. Somit ist ständig ein hoher Speichertransferaufwand erforderlich, um zu jedem Zeitpunkt die richtigen Daten verfügbar zu haben. Dies ist ein wesentlicher weiterer Unterschied zum SSE-Backend, welches Daten aus einem verhältnismäßig sehr großen Hauptspeicher direkt adressieren kann, während beim Cell-Prozessor für jedes Tupel von zu verarbeitenden Daten erst Transfers zwischen Hauptspeicher und lokalem Speicher erforderlich sind.

Es folgt ein Beispiel, welches den hohen Aufwand an Speichertransfers und das Verhältnis zum eigentlich funktional rechnenden Code verdeutlichen soll. `x[y].vectorised[i]` beschreibt dabei den Zugriff auf den `i`-ten SIMD-Vektor des `y`-ten Datenblocks des Containers `x`. `counter` repräsentiert die Anzahl der notwendigen Transfers um die volle Größe

des zu verarbeitenden Containers zu erreichen. Da innerhalb der *while-Schleife* bereits die Speichertransfers für die nachfolgenden Daten angefordert werden während auf den aktuellen Daten gerechnet wird, zeigt dieses Beispiel auch eine einfache Implementierung von *double buffering*.

```
// Laden des jeweils ersten Blocks
mfc_get(a[current - 1].untyped, ea_a, size, current, 0, 0);
mfc_get(b[current - 1].untyped, ea_b, size, current, 0, 0);
ea_a += size;
ea_b += size;

while (counter > 1)
{
    nextsize = (counter == 2 ? lastsize : inst.size);
    // Laden des jeweils folgenden Blocks i + 1
    mfc_get(a[next].untyped, ea_a, nextsize, next, 0, 0);
    mfc_get(b[next].untyped, ea_b, nextsize, next, 0, 0);
    ea_a += nextsize;
    ea_b += nextsize;
    mfc_write_tag_mask(1 << current);
    mfc_read_tag_status_all();

    // Rechnen auf dem aktuellen Block i
    for (unsigned i(0) ; i < size / sizeof(vector float) ; ++i)
    {
        a[current].vectorised[i] =
            spu_add(a[current].vectorised[i],
                b[current].vectorised[i]);
    }

    // Schreiben des aktuellen Blocks i
    mfc_putb(a[current].untyped, ea_result, size, current, 0, 0);
    ea_result += size;
    --counter;
    unsigned temp(next);
    next = current;
    current = temp;
    size = nextsize;
}

mfc_write_tag_mask(1 << current);
mfc_read_tag_status_all();

// Rechnen auf dem letzten Block
for (unsigned i(0) ; i < size / sizeof(vector float) ; ++i)
{
    a[current].vectorised[i] =
    spu_add(a[current].vectorised[i], b[current].vectorised[i]);
}
```

```
// Schreiben des letzten Blocks.
mfc_putb(a[current].untyped, ea_result, size, current, 0, 0);
mfc_write_tag_mask(1 << current);
mfc_read_tag_status_all();
```

Für eine komplexere Operation, bei der der Zugriff auf Daten nicht kontinuierlich linear erfolgt, sondern zu jedem Zeitpunkt dynamisch auf unterschiedliche Bereiche des Datenraumes zugeriffen wird, ist der Aufwand für die Verwaltung der Speichertransfers noch wesentlich höher. Des weiteren müssen beispielsweise beim Produkt zweier `DenseMatrix`-Objekte ein und dieselben Daten gegebenenfalls mehrfach geladen werden, weil nicht alle Berechnungen auf diesem Teilstück zu einem Zeitpunkt möglich sind, da die korrespondierenden Daten ebenfalls an unterschiedlichen Stellen liegen oder nicht in einen 16 kB-Transfer passen. Auch an dieser Stelle hat z.B. die SSE-Einheit durch die Adressierung eines wesentlich größeren Speicherraumes deutliche Vorteile.

Implementierung

Die Realisierung des Cell-Backends findet sich in erster Linie in den Dateien, die mit `spe_` beginnen. Diese Dateien, wie z.B. `spe_kernel.cc` stellen die PPE-Seite des Backends dar. Die Operationen für den Cell-Prozessor sind mit dem Tag `tags::Cell` versehen und befinden sich in Dateien wie `sum-cell.cc`. Sie werden nur zu HONEI hinzu kompiliert, wenn *configure* mit `-enable-cell` ausgeführt wird. Das korrespondierende SPE-Programm für die jeweilige Operation heißt in diesem Fall z.B.
`sum_dense_dense_float.cc`. Der Name beschreibt immer die Operation, die Art der Container (*dicht-, dünn-, mit Bändern besetzt*) sowie deren Datentyp (*float, double*).

Umsetzung ausgewählter Operationen

Im folgenden sollen die Ideen zur Umsetzung einiger der implementierten Operationen für den Cell-Prozessor näher erläutert werden:

Product(DenseMatrix, DenseMatrix) Da in HONEI eine `DenseMatrix` als eine kontinuierliche Folge ihrer Zeilen abgespeichert wird, ist kein effizienter Zugriff auf Werte einer Matrixspalte möglich. Daher ist es, ähnlich wie im SSE-Backend, nicht möglich, den naiven Ansatz eines Skalarproduktes aus Zeilen- und Spaltenvektoren auf die Cell-Version zu übertragen.

Aus diesem Grund musste ein zeilenbasierter Algorithmus entwickelt werden. Eine weitere Schwierigkeit liegt darin begründet, dass Übergänge zwischen Zeilen bei nicht durch 4 teilbaren Spaltenanzahlen mitten in einem SIMD-Vektor liegen, was die vektorisierte Abarbeitung und auch die Partionierung des Problems erschwert.

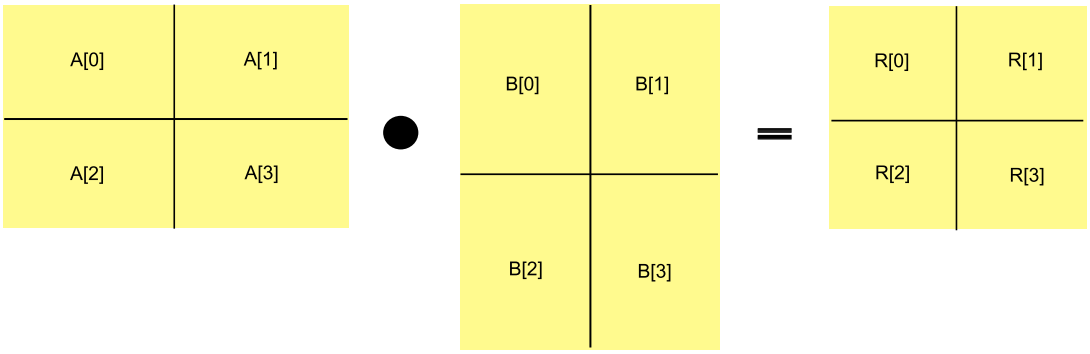


Abbildung 3.1.: Partitionierung der Matrizen im DenseMatrix-Produkt

Mit Hilfe der Transferlisten wird das Problem folgendermaßen aufgeteilt.

$$R[0] = A[0] * B[0] + A[1] * B[2]$$
$$R[1] = A[0] * B[1] + A[1] * B[3]$$
$$R[2] = A[2] * B[0] + A[3] * B[2]$$
$$R[3] = A[2] * B[1] + A[3] * B[3]$$

Während man die Teile $A[0]$ und $A[1]$, bzw. $A[2]$ und $A[3]$ der Matrix A in diesem Falle per *double buffering* übertragen kann, müssen die Matrizen B und R mit Hilfe von DMA-Listentransfers in hohem Maße zerteilt werden. Auf Grund der bereits beschriebenen Problematik von Zeilengrenzen kommt es bei diesem Vorgehen aber zu Interferenzen, insbesondere beim Zurückschreiben eines Ergebnisses $R[i]$. So schreibt z.B. das SPE, welches $R[0]$ berechnet auch in den Bereich von $R[1]$, da die SIMD-Vektoren, die den Beginn der Zeile enthalten, auch Elemente aus der darüberliegenden Zeile enthalten können, falls die Spaltenanzahl nicht ohne Rest durch die Anzahl der Elemente eines SIMD-Vektors teilbar ist. Aus diesem Grund wird auch das PPE mit in die Berechnung eingebunden. Sie berechnet die Ergebnisse für die Spalten der Matrix R, in denen Interferenzen zwischen den SPEs auftreten würden. Die Berechnungen des PPEs erfolgt simultan zu den Berechnungen der 4 SPEs. Die Ergebnisse des PPEs werden jedoch erst in die Matrix R geschrieben, wenn die SPEs ihre Berechnungen beendet haben. Auf diese Weise wird die Konsistenz der Daten sichergestellt. Da nur zwei solche Interferenzräume von jeweils 4 Spalten (einfache Genauigkeit) existieren, können $n-(2*4+n \mod 4)$ Spalten auf SPEs gerechnet werden, wobei n die Anzahl der Spalten der Matrix R darstellt. Insbesondere für große Eingaben bleibt damit die Wartezeit für die Schreibzugriffe des PPEs am Ende klein, während zusätzlich die große Last aufgeteilt wird. Die vier SPEs benötigten relativ viel Speicherbandbreite, da für jede Zeile eines Blockes von A das Durchlaufen des kompletten Blocks $B[i]$ erforderlich ist. An diesen Stellen wird der hohe Aufwand ersichtlich, der gegenüber hochperformanten Beispielimplementierungen erforderlich ist, wenn das Produkt für beliebige Matrizen berechenbar sein soll und nicht auf für die Hardware zugeschnittene Beispiele wie z.B. quadratische Matrizen, deren Zeilen- bzw.

Spaltenanzahl ein Vielfaches von 128 Elementen ist und welche sich daher optimal in 64x64 Blöcke (entsprechend in einfacher Genauigkeit genau einen 16 kB DMA-Transfer) zerlegen lassen, beschränkt sein darf.

Eine letzte Besonderheit betrifft nochmals das Zurückschreiben von R. Da die Anzahl der Zeilen eines R-Blocks von der Anzahl der Zeilen eines korrespondierenden A-Blocks abhängt, muss, da A ja per *double buffering* übertragen wird, auch entsprechend oft ein (Teil-)Ergebnis von R zurückgeschrieben werden. Man kann allerdings nicht erwarten, dass, wenn n Zeilen aus A mit einer `SPETransferList` übertragen werden können, die korrespondierenden n Zeilen aus R ebenfalls mit nur einem `SPETransferList` übertragen werden können. Dies ist zum einen in der möglichen unterschiedlichen Anzahl der Spalten begründet, andererseits – und wesentlich einschränkender – aber auch technisch bedingt, da für alle Elemente einer `SPETransferList` verlangt wird, dass ihre Hauptspeicher-Adressen die gleichen höherwertigen 32 Bit besitzen. Letzteres ist insbesondere bei großen Matrizen nicht trivial zu garantieren. Es ist nicht möglich, eine `SPETransferList` nur teilweise zu übertragen und anschließend den Transfer, mit manipulierter Local-Store Adresse, fortzusetzen. Aus diesem Grund wurde der PPE-Algorithmus so entwickelt, dass er bei einer Zeile in R, die in Matrix A gerade die Grenze eines *double buffering* Schrittes darstellt, eine neue Liste beginnt und die Zeile dort ablegt. Auf diese Weise wird sichergestellt, dass sich die Zeilenanzahlen, die zu einer *double buffering* Iteration gehören, immer auf die Zeile genau aus einer oder mehreren Listen zusammensetzen lassen.

Product(BandedMatrix, DenseVector) Die generelle Vorgehensweise ähnelt derjenigen des SSE-Backends. Um konfliktfrei auf mehreren SPEs arbeiten zu können, wird der Ergebnisvektor in verschiedene zusammenhängende Bereiche geteilt. Auf diese Weise können mehrere SPEs lesend auf die Bandmatrix und den Operandenvektor zugreifen und konfliktfrei auf ihrem Teil des Ergebnisvektors lesen und schreiben. Das PPE-Programm durchläuft die `BandedMatrix` also Band für Band und erstellt jeweils für jedes SPE eine Operation für ihren Teil des Ergebnisvektors. Diese werden in einer `SPEInstructionQueue` gesammelt und erst nach der kompletten Assemblierung der Teilaufgaben an die SPEs geschickt. Dies ist im aktuellen Cell-Backend schneller, da das Starten einer `SPEInstruction` oder einer `SPEInstructionQueue` im Vergleich zur Berechnung sehr lange dauert.

3.2. Test- und Benchmark-Framework

3.2.1. Das Test-Framework

Ziele

Gesetztes Ziel der Entwicklung der Test-Bibliothek sollte es sein, die geschriebenen Klassen und Funktionen von HONEI effizient testen zu können. Der Entwickler einer Klasse soll mit möglichst geringem Aufwand ein Testcase für seine Klasse instanzieren können. Als Software-Werkzeug, das in die Infrastrukur des Build-Systems integriert ist, wurde

dafür `Make` gewählt. Dem Entwickler sollte es damit insbesondere möglich sein, (halb-) automatisch einzelne oder alle verfügbaren Tests auszuführen.

Schreiben von Testcases

Wenn ein Entwickler eine Klasse oder eine Bibliothekskomponente fertig gestellt hat, so kann er direkt ein *Testcase* anlegen. Dazu erstellt er eine Datei mit demselben Namen wie die zu testende Datei der entsprechenden Komponente, fügt aber noch `_TEST` am Ende in den Dateinamen ein.

Zum Erstellen des *Testcases* leitet der Entwickler eine Klasse von der Klasse `BaseTest` ab. Dem Konstruktor übergibt er einen String mit dem Namen des Tests. So kann später beim Testlauf ein aussagekräftiger Name ausgegeben werden. In der Methode `virtual void run() const` schreibt er den Testcode. Dieser kann dann eines oder mehrere der folgenden Makros benutzen um die Rückgabewerte der zu testenden Funktion auszuwerten.

- `TEST_CHECK(a)` – Prüft, ob ein Ausdruck `a` gleich `true` ist.
- `TEST_CHECK_EQUAL(a,b)` – Prüft, ob die Ausdrücke `a` und `b` gleich sind.
- `TEST_CHECK_NOT_EQUAL(a,b)` – Prüft, ob die Ausdrücke `a` und `b` nicht gleich sind.
- `TEST_CHECK_STRINGIFY_EQUAL(a,b)` – Prüft, ob die Strings der Ausdrücke `a` und `b` gleich sind.
- `TEST_CHECK_EQUAL_WITHIN_EPS(a,b,eps)` – Prüft, ob die Differenz von `a` und `b` kleiner `eps` ist.
- `TEST_CHECK_TROWS(m,e)` – Prüft, ob die Methode `m` eine Ausnahme der Klasse `e` wirft.

Beispiel

```
using namespace std;
using namespace tests;

class UnitTest : public BaseTest
{
    /// Konstruktor
    UnitTest(const std::string & id) :
    BaseTest(id)
        {
        }

    virtual void run() const
    {
        TEST_CHECK(true);
        TEST_CHECK_EQUAL(1,1);
        TEST_CHECK_STRINGIFY_EQUAL(4711, 4711);
        TEST_CHECK_EQUAL_WITHIN_EPS(25,23,2.2);
        TEST_CHECK_THROWS(string("0").at(10), exception);
    }
}
UnitTest metatest("UnitTest-test");
```

Durchführung der Tests

Um alle Tests nacheinander auszuführen, startet man auf der Kommandozeile den Befehl `make check`. Am Ende des Testdurchlaufs erhält man eine Zusammenfassung der erfolgreichen und gescheiterten Tests. Ein solcher kompletter Testlauf kann für die in HONEI definierten Tests auf Hardware mit geringer Speicher- und Rechenkapazität allerdings sehr lange dauern. Als Lösung bietet sich an, entweder nur einen einzelnen Test zu starten oder mit `make quickcheck` eine gekürzte Auswahl der Testcases zu verwenden.

Einzelne Tests werden mit `make check tests=test1,test2,...` ausgeführt. Will man alle Tests eines Unterordners ausführen, startete man `make check` einfach in diesem Ordner statt im Verzeichnis `trunk`.

Um `make quickcheck` verwenden zu können, müssen die Entwickler neben der eigentlichen Testklasse noch von einer Klasse `QuickTest` ableiten. Die Syntax ist identisch, nur sollten hier die Tests auf Probleminstanzen beschränkt werden, die absehbar eine relativ geringe Ausführungszeit erwarten lassen. Auch `make quickcheck` kann in einzelnen Unterordnern statt im Verzeichnis `trunk` ausgeführt werden.

Realisierung

Im folgenden wird beschrieben, wie das `unittest`-Framework realisiert wurde. Einige Teile des Quelltexten werden hier näher betrachtet.

Zunächst müssen nach dem Aufruf von `make check` alle in Frage kommenden Tests in einer Liste gesammelt werden. Dann muss diese Liste abgearbeitet werden. Dies bewerkstelligt eine zentrale Instanz von `TestList`.

Jede Instanz von `BaseTest` trägt sich selbst durch einen Aufruf im Konstruktor in diese Liste ein. Diese Liste wird in der `main`-Methode einmal durchlaufen. Dabei wird jeder Test ausgeführt. Bei `make quickcheck` wird zusätzlich noch die Methode `bool is_quick_test()` ausgeführt. Gibt diese `false` zurück, wird der Test ausgelassen.

Der Programmierer verwendet nur die anfangs aufgelisteten Makros, beispielsweise `TEST_CHECK_EQUAL(a,b)`.

3.2.2. Das Benchmark-Framework

Ziele

Zur Optimierung der Implementierung sowie zum Vergleich der Laufzeit der Algorithmen auf verschiedener Hardware wurde ein Benchmark-Framework erstellt, mit dessen Hilfe ohne großen Aufwand Benchmarks erstellt werden können.

Realisierung

Die Realisierung des Benchmark-Frameworks basiert stark auf der Architektur des zuvor beschriebenen Test-Frameworks. Die Benchmarks werden ebenfalls in einer zentralen Liste gesammelt und die entsprechende Auswahl dann nacheinander ausgeführt.

Schreiben von Benchmarks

Um einen Benchmark zu implementieren, wird dieses von der Klasse `Benchmark` abgeleitet. Zur Zeitmessung und Auswertung stehen einige Hilfsmittel zur Verfügung:

- `BENCHMARK(a)` – Misst die Zeit, die für die Ausführung von `a` verwendet wird und fügt die gemessene Zeit an eine Liste an.
- `evaluate()` – Die Funktion `evaluate` wertet die Liste der gemessenen Zeiten aus und gibt die Ergebnisse in der Konsole und in eine Textdatei `BenchmarkOut.txt` aus. Die Ausgabe beinhaltet die gesamte, die kürzeste, die längste und durchschnittliche Laufzeit (Arithmetisches Mittel und Median) sowie Datum und Uhrzeit des entsprechenden Benchmarks.
- `evaluate(BenchmarkInfo info)` – Zusätzlich zu der Ausgabe von `evaluate()` werden hier noch die erreichten FLOPS und die Transferrate ausgegeben.
- `evaluate_to_plotfile(list<BenchmarkInfo> info, list<int> cores, int count)` – Erzeugt eine Ausgabe in Tabellenform in der Datei `BenchmarkPlotData` sowie eine `.plt`-Datei die mit GnuPlot geöffnet werden kann um die Benchmarkergebnisse graphisch darzustellen.

Die benötigten `BenchmarkInfo` für die ausführlicheren Auswertungen erhält man mit `get_benchmark_info` von der jeweiligen Operation. Dieser Methode müssen die benutzten Operanden übergeben werden, damit eine möglichst genaue Auswertung erfolgen kann.

Soll eine individuelle Ausgabe erstellt werden, so kann man auch direkt auf die durch `BENCHMARK(a)` gemessenen Zeiten in der Liste `_benchlist` zugreifen.

Nach Fertigstellung der Klasse werden Benchmarks mit den gewünschten Spezifikationen wie zum Beispiel Präzision, Name und Größe der Operanden instanziiert und damit – wie im vorigen Abschnitt beschrieben – in eine Benchmark-Liste eingetragen. Der gewählte Name, der dem Konstruktor übergeben wird, sollte möglichst alle nötigen Informationen enthalten, da dieser dem Anwender als Auswahl in einem *Interface* angezeigt wird. Wenn die neuen Benchmarks auch direkt mit `make bench` ausgeführt werden sollen, muss die neue Datei noch per `include` in die Datei `bench.cc` eingefügt werden. Um eine doppelte Instanziierung der Benchmark-Liste zu verhindern wird ein Symbol `ALLBENCH` in `bench.cc` definiert. In der neu geschriebenen Datei muss das Einbinden des Benchmark-Frameworks davon abhängig gemacht werden (siehe Beispiel).

Beispiel

```
#ifndef ALLBENCH
#include <benchmark/benchmark.cc>
#include <tr1/memory>
#include <string>
#endif
#include <honei/libla/sum.hh>

using namespace honei;

template <typename Tag_, typename DataType_>
class DenseVectorSumBench :
    public Benchmark
{
    private:
        unsigned long _size;
        int _count;
    public:
        DenseVectorSumBench(const std::string & id,
        unsigned long size, int count) :
            Benchmark(id)
        {
            register_tag(Tag_::name);
            _size = size;
            _count = count;
        }

        virtual void run()
        {
            DenseVector<DataType_> dv0(_size, DataType_(rand()));
            DenseVector<DataType_> dv1(_size, DataType_(rand()));
            for(int i(0) ; i < _count; ++i)
            {
                BENCHMARK(Sum<Tag_>::value(dv0, dv1));
```

```
            }
            BenchmarkInfo info(Sum<>::get_benchmark_info(dv0, dv1));
            evaluate(info);
        }
};
DenseVectorSumBench<tags::CPU, float>
    DVSBenchfloat1("Dense Vector Sum Benchmark -
    vector size: 64^4, float", 64ul*64*64*64, 10);
DenseVectorSumBench<tags::CPU, double>
    DVSBenchdouble1("Dense Vector Sum Benchmark -
    vector size: 64^4, double", 64ul*64*64*64, 10);
```

Durchführung der Benchmarks

Um die umfangreichen Funktionen für den Benutzer leichter zugänglich zu machen, existieren verschiedene `make`-Targets. So kann im Voraus die Menge der auszuführenden Benchmarks sowie die Art der Auswertung bestimmt werden.

Nach dem Kompilieren können die nun existierenden ausführbaren Dateien auch direkt gestartet werden. Neben dem Hauptbenchmark `benchmark` existiert für jede Operation eine eigene Datei. Über Argumente wie `sc, mc, sse` und `cell` kann auch hier die Auswahl der Benchmarks noch verfeinert werden, oder mit `i` und `plot` Funktionen wie ein Benutzerinterface und das automatische Erstellen von `GNUPlot`-Dateien zugeschaltet werden.

`./benchmark mc sse i` bietet zum Beispiel im Benutzerinterface alle MultiCore- sowie SSE-Benchmarks zur Auswahl an.

3.3. Flachwassersimulation

In diesem Abschnitt sollen alle für die Implementierung des Lösers für die Flachwassergleichungen (im folgenden gemäß des Namens der Hauptlöserklasse auch `RelaxSolver` genannt) relevanten Überlegungen dargestellt werden.

3.3.1. Implementierung

Diskretisierung

Die Wahl der Gitterdatenstruktur folgt intuitiv aus der Beschreibung des Verfahrens. Behält man die von Delis und Katsaounis [DK05] vorgeschlagene reguläre Rechtecksform der Diskretisierung bei und verzichtet man außerdem auf adaptive Verfeinerung bei globaler Wahl von Δx und Δy oder auch auf Bereichszerlegungen (die im Rahmen der PG von vornherein nicht vorgesehen waren), so erfolgen daraus zwei wesentliche Ersparnisse. Zum einen können recht einfache Randbedingungen gewählt werden (s.u.) und zum anderen kann die Speicherung der Skalarfelder in Matrix-Containern der Basisbibliothek erfolgen, so dass keine separate (geometrische) Gitterdatenstruktur nötig ist.

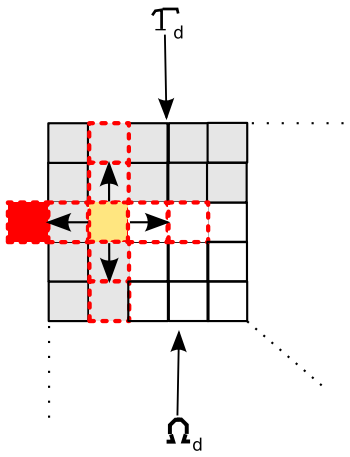


Abbildung 3.2.: Das Problem des ungültigen Zugriffs auf Nachbarwerte von Randzellen.

Randbehandlung

Das erste wesentliche Problem bei der Implementierung ist die Wahl der Randbedingungen (*boundary conditions*). Die Entscheidung für oder gegen eine spezielle Art von Randbedingungen hängt dabei auch von der Art der Szenarien[3] ab, die von dem Löser bewältigt werden müssen. Die denkbar einfachste Art von Randbedingungen sind sogenannte reflektierende Ränder (*reflective boundaries*), bei denen die Randwerte stets ihren direkten Nachbarn entsprechen. Diese Art von Randbedingung ist für ernstzunehmende technische Anwendungen sicherlich oftmals unzureichend. Sie wurde dennoch gewählt, da nur insofern starke Anforderungen an den Löser gestellt worden sind, als dass er trockenen Boden und ruhiges Wasser (*dry- und steady-states*) simulieren können muss. Dies wird jedoch durch die Wahl der Randbedingungen nicht wesentlich beeinflusst. Anders ausgedrückt zählt der visuelle Effekt hier mehr, als beispielsweise die Möglichkeit, das Rechengebiet in Untergebiete zerteilen zu können. Dennoch wurde bei der Implementierung wie in den Operationsbibliotheken auf Erweiterbarkeit geachtet. Um dem Löser ein anderes Randverhalten vorzugeben, ist keine vollständige Reimplementierung nötig, vielmehr muss nur an wenigen Stellen im Code eine Veränderung vorgenommen werden, was durch generische Programmierung ermöglicht wird.

Im Falle von `RelaxSolver` liegt ein Verfahren vor, dass zweistufige Nachbarschaftsbeziehungen bei der Matrixassemblierung mit einbezieht, wie in Abbildung 3.2 gezeigt. Das Problem, das daraus entsteht ist, dass man für zwei Reihen Gitterzellen entlang des Randes eine Ausnahmebehandlung braucht, was zu tendenziell vielen Verzweigungen (*branches*) führen würde. Gerade im Hinblick auf eine SIMD-Implementierung sollte man solche von vorneherein eliminieren. Dies geschieht folgendermaßen: In einer Vorberechnungs- (*preprocessing*-) Phase werden die Eingabeskalarfelder auf größere Matrizen zentriert übertragen (*mapping*), wie in Abbildung 3.2 dargestellt, wobei das Ein-

[3] Unter dem Begriff Szenario wird im folgenden stets ein Datensatz als Eingabe für den Flachwasserlöser verstanden. Dieser besteht im wesentlichen aus den Skalarfeldern für die initiale Wasserhöhe, das Bodenprofil und die Geschwindigkeiten sowie aus den Gitterschrittweiten und der Zeitschrittgröße.

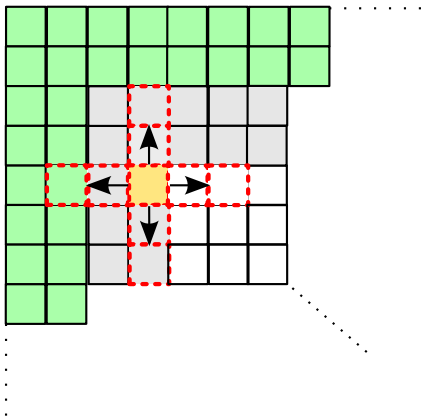


Abbildung 3.3.: Randübertragung: Durch das Übertragen auf eine entsprechend dimensionierte Matrix, kann der Zugriff ohne Sonderbehandlung erfolgen.

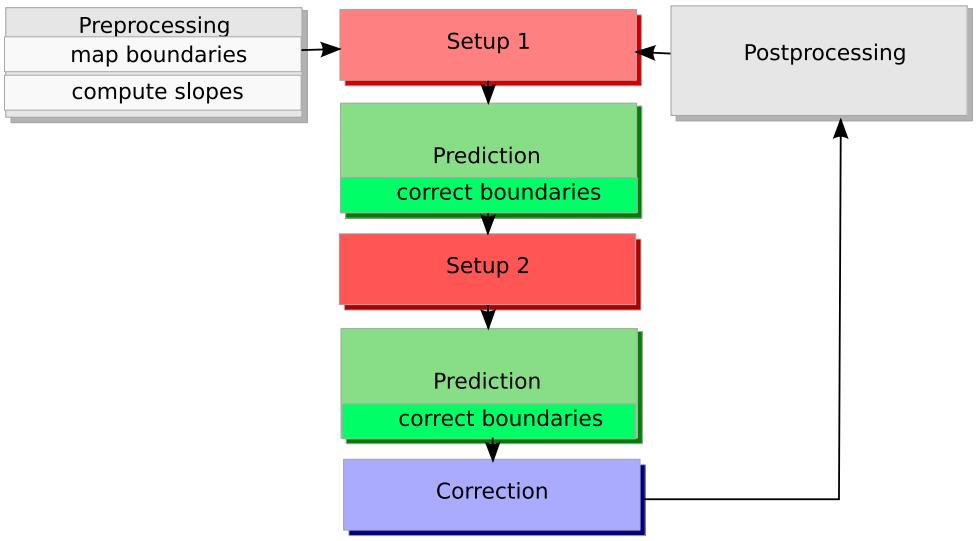


Abbildung 3.4.: Das Relaxationsverfahren: Abfolge mit Preprocessing, Postprocessing und Randbehandlung

gabefeld von zwei Reihen Zellen (Geistzellen, *ghost cells*) umschlossen wird. Dadurch ist sichergestellt, dass jeder Zugriff von Randzellen auf ihre direkten und indirekten Nachbarn gültig ist, ohne zusätzliche Informationen über eine Zelle zu benötigen. Gleichzeitig wird so sichergestellt, dass für den Rand keine Sonderfallbehandlung nötig wird. Die Geistzellen werden in der Berechnung stets übersprungen.

RelaxSolver

Mit dem oben beschriebenen Vorberechnungsverfahren für die Randwerte und einer Nachbehandlung sieht das Verfahren nun wie in Abbildung 3.4 dargestellt aus. Während der Vorbehandlung werden zusätzlich noch die Bodendifferentiale $\frac{\partial Z(x,y)}{\partial x}$ bzw. $\frac{\partial Z(x,y)}{\partial y}$ berechnet. In der Nachbehandlung (*postprocessing*) wird entweder eine Echtzeitdarstellung durch eine OpenGL basierte Darstellungsapplikation durchgeführt oder ein Plot im GnuPlot-Format erstellt.

Zunächst als monolithische Kontrollklasse implementiert, wiesen erste Benchmarks bereits darauf hin, dass der Löser einen wesentlichen Anteil der Rechenzeit für die nicht in Basisoperationen weggekapselten Module aufwendete. Obwohl die Unterprozeduren keinen echten Wiederverwendungswert haben, wurde der Löser im Hinblick auf eine Applikationsoptimierung reimplementiert. Ein wesentlicher Punkt bei der Evaluierung wird es sein, zu untersuchen, inwieweit eine allgemeingültige Bibliothek von algebraischen Operationen überhaupt sinnvoll ist, d.h. welcher Anteil der Anwendungen durch solche Operationen abgedeckt wird. Durch die Modularisierung wird dies wesentlich vereinfacht.

Operationen der Bibliothek

Damit bleibt nur noch der Code im Löser selbst gekapselt, der entweder *vollständig* auf Basisoperationen aufsetzt, oder für die Randbehandlung, respektive das Nachkorrigieren der Geistzellen verantwortlich ist. Die monolithische Variante wurde für Demonstrationszwecke weiter gepflegt, um Aussagen über die Rentabilität von solchen Optimierungen auf Applikationsebene zuzulassen.

Die modulare Version des Lösers verwendet die folgenden Operationen.

`FlowProcessing` Diese Operation Berechnet die Flüsse in x- und y-Richtung für einen Vektor, d.h. $\mathbf{F}(\mathbf{x}_{ij})$ bzw. $\mathbf{G}(\mathbf{x}_{ij})$.

`SourceProcessing` Die Berechnung des Quellterms $\mathbf{S}(x_{ij})$ für einen Vektor wird durch diese Operation bewerkstelligt. Zu beachten ist, dass die Berechnung des Quellterms für viele Szenarios nahezu entfallen kann. Für eine Manningkonstante $n_m = 0$ wird die Quelltermberechnung zu

$$\mathbf{S}(x_{ij}) = \mathbf{S}\begin{pmatrix} h_{ij} \\ q^1_{ij} \\ q^2_{ij} \end{pmatrix} = \begin{pmatrix} 0 \\ h_{ij} b^x_{ij} \\ h_{ij} b^y_{ij} \end{pmatrix} \tag{3.4}$$

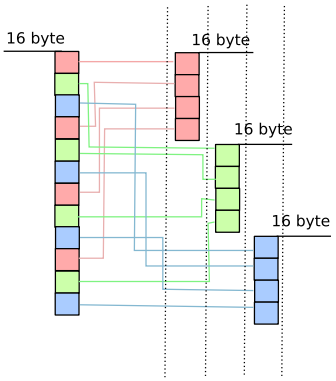


Abbildung 3.5.: Extrahieren von Gleitkommadaten aus den in Tripeln angeordneten Vektoren aus der Flachwassersimulation: Vor der Extrahierung würde ein SIMD-Vektor nicht zueinander gehörige Werte enthalten. Es werden zusätzliche, an 16-Byte Grenzen ausgerichtete Hilfsvektoren angelegt, so dass die Daten nach der Extrahierung stets geeignet im Speicher ausgerichtet sind.

`CorrectionProcessing` Hier wird lediglich das gewichtete Mittel aus den beiden Vorhersageschritten (siehe Gleichung (2.10)) berechnet.

`AssemblyProcessing` Das Assemblieren der Matrizen zerfällt in zwei Hauptschritte, in denen die Matrizen M_1, M_3, M_2 und M_4 aufgebaut werden sowie das Kopieren der Matrizen M_3 und M_4 bzw. das Assemblieren von M_5 bis M_8, welches auf einfache elementweise Multiplikationen der Bänder von M_1 und M_2 zurückgeführt werden kann.

Sonstige

Neben den unbedingt notwendigen Klassen der Bibliothek wurde des weiteren Funktionalität zum Erstellen und Verwalten von Datensätzen als Eingabe (*Szenarios*) für den Löser dieser Bibliothek bereitgestellt.

3.3.2. SIMD-Implementierung - Ideen für Cell und SSE

Für eine performante Implementierung der Operationen in der Flachwasserbibliothek ergeben sich in erster Linie drei Probleme:

Sowohl in `FlowProcessing` als auch in `SourceProcessing` werden Tripel verwendet, was zur Folge hat, dass man keine kompakte Speicherung für die Elemente zur Verfügung hat, da zwischen je zwei im SIMD-Sinne zusammenhängenden Elementen der Operanden wiederum zwei andere Elemente stehen. Dies bedeutet, dass man, um die Berechnung auf der SSE Einheit bzw. durch die SPEs der Cell BE überhaupt möglich zu machen, die entsprechenden Elemente aus den Operandenvektoren extrahieren muss, siehe Abbildung 3.5. Dies kann durch zwei Techniken geschehen. Die erste Möglichkeit besteht

im Mischen der Elemente zweier SIMD-Vektoren, dem so genannten *shuffle*. Die zweite Möglichkeit ist, dass man die entsprechenden Daten in einer Vorberechnungsphase kompakt abspeichert, was man geschickt mit den skalaren Teilen der Rechnung verknüpfen kann. Dies führt zu der Notwendigkeit, die dabei nötigen Hilfsvektoren an 16-Byte Speichergrenzen ausgerichtet zu speichern, was bei größeren Problemen zu einem sehr großen Speicherplatzbedarf führen kann, da man zusätzliche, an 16-Byte-Grenzen ausgerichtete Hilfsfelder benötigt. Dennoch hat sich diese Technik für SSE bewährt, da das *shuffle* zu Problemen mit der Kontrollierbarkeit der Registerinhalte geführt hat. Auf der Cell BE ist dies aber ohne Probleme möglich, und daher ist es dort der obigen Technik vorzuziehen. Im Fall von SSE hat sich gezeigt, dass trotz der Kosten für die zusätzlichen Kopieroperationen der SIMD-Code performant ist, wie im nächsten Kapitel demonstriert. Nicht alle für die Rechnung in `SourceProcessing` nötigen arithmetischen Operationen stehen intrinsisch für SSE und SPEs zur Verfügung. So gibt es beispielsweise keinen atomaren Befehl für ein SPE-Programm für das Potenzieren oder das Radizieren. Daher muss man die entsprechenden Teile der Rechnung auf die Fließkommaeinheit (FPU) bzw. das PPE auslagern, was zu Performanzverlust gegenüber der quelltermunabhängigen Lösung führt, oder es muss eine hardwarespezifische Implementierung bereitgestellt werden. Das dritte Problem hängt mit den Sprüngen in den Operanden zusammen, die sich aus der Randbehandlung ergeben und die für nicht an 16-Byte Grenzen ausgerichtete SIMD-Quadrupel sorgen würden. Dies kann weitgehend dadurch umgangen werden, dass man die Geistzellen schlicht ignoriert, d.h. einfach als Normalzelle betrachtet, was ja eine wesentliche Idee dieser Randbehandlung ist. Für das `CorrectionProcessing` sorgt dies allerdings für Fehler, da der Rand gewissermaßen überkorrigiert wird. Dies äußert sich dadurch, dass das Wasser einen bereits erreichten Pegel am Rand nicht mehr verlieren würde, d.h., es würde nicht der Schwerkraft nachgeben, was zu einem falschen Ergebnis führt. Daher muss hier eine Sondermaßnahme ergriffen werden. Die Geistzellen werden wie immer ignoriert, es wird jedoch zu Beginn der Rechnung, ähnlich wie beim Vorberechnen für in Tripeln angeordnete Operationen eine Kopie der Daten vorgehalten, mit deren Hilfe am Schluss die Randzellen zurückkorrigiert werden. Mit Hilfe dieser Technik konnte die Korrekturphase wesentlich beschleunigt werden.

3.3.3. Gemischte Genauigkeit

Die Idee gemischt genauer Verfahren, d.h. die Verwendung mehrerer (in der Regel: zwei verschiedener) Genauigkeitsformate für die Gleitpunktdarstellung von reellen Zahlen in einer Rechnung, zielt darauf ab, möglichst große Teile der Rechnung in geringer Genauigkeit (typischerweise: *float*-Repräsentation) statt in hoher Genauigkeit (typischerweise: *double*-Repräsentation) durchzuführen. Dadurch kann zum einen Hardware eingespart werden, zum Anderen die Rechenzeit verkürzt werden. Die *Cell BE* verfügt mit ihren SPEs über eine hohe Rechenleistung in geringer Genauigkeit. Es bietet sich also an, einen großen Teil der Rechnung auf parallel (in den SPEs) arbeitende FPUs auzulagern. Hier liegt die Leistung in geringer Genauigkeit theoretisch etwa um den Faktor 14 höher als in hoher. So können auch wissenschaftliche Anwendungen von der höheren Anzahl einfach genauer FPUs auf solchen Architekturen profitieren. Allerdings haben gemischt

genaue Verfahren auch praktische Nachteile. Immer dann, wenn man mit geringer Genauigkeit weiterrechnen möchte, wird eine *Konversion* der Zahlenrepräsentation nötig. Wie (effizient) diese Konversion realisiert wird, hängt dabei von der Hardware ab: Entweder wird ein Bitvektor durch eine *Arraycopy*-Operation überführt, oder, wenn die zugrundeliegende *FPU* des Prozessors beide Formate unterstützt, *gecastet*.

Für das prototypische Beispiel der Berechnung einer möglichst genauen Lösung von linearen Gleichungssystemen gilt, dass mehr als 90 Prozent der Rechnung in geringer Genauigkeit vollzogen werden können. Ein solches Verfahren, ein sogenanntes *mixed precision iterative refinement*, wurde im Rahmen der PG ebenfalls implementiert, siehe Kapitel 3.5.1. Da es sich bei dem implementierten Verfahren um ein explizites Verfahren handelt, kann dieses hier nicht zum Einsatz kommen. Die Struktur des Lösers, wie sie im letzten Abschnitt beschrieben wurde, bietet jedoch die Möglichkeit, gemischt genaue Methoden auch hier anzuwenden. Eine implementierte Konfiguration ist, nur den ersten *oder* den zweiten Vorhersageschritt oder nur den Korrekturschritt in hoher Genauigkeit durchzuführen. Andererseits könnte man auch nur jeden k-ten Zeitschritt mit hoher Genauigkeit rechnen. Dabei muss selbstverständlich annähernd dieselbe Ergebnisgenauigkeit sichergestellt sein. Im Falle des Flachwasserlösers, bei einem rein subjektiven Qualitäts- (oder Genauigkeits-) Maß ist dies jedoch nicht trivial. Daher ist, neben einem visuell ansprechenden Ergebnis, die Löserstabilität das stärkste Qualitätsmaß, das hier angelegt werden soll. Zusätzlich wurde auch eine Möglichkeit zur numerischen Volumenberechnung unter Höhenfeldern implementiert, um die Masseerhaltungsfähigkeit des Verfahrens zu verifizieren. Damit kann die numerische Qualität der Lösungen durch verschiedene Löserkonfigurationen verglichen werden. Abschließend werden also die folgenden Fragen zu beantworten sein:

1. Kann mit einer gemischt genauen Konfiguration des Lösers dasselbe visuelle Ergebnis wie mit einem ausschließlich in hoher Genauigkeit rechnenden Löser bei gleicher Stabilität überhaupt erreicht werden?

2. Wie groß ist der Anteil des Verfahrens, der in geringer Genauigkeit gerechnet werden kann, so dass der Rechenzeitgewinn durch Konvertierungsoperationen nicht aufgehoben wird?

3. Muss das Verfahren innerhalb eines Zeitschrittes in verschiedene Genauigkeitsformate aufgeteilt werden, oder reicht es jeden k-ten Zeitschritt doppelt genau zu rechnen? (In letzterem Fall wäre k natürlich möglichst genau zu bestimmen.)

Experimente zum gemischt genauen Lösen der Flachwassergleichungen mit `RelaxSolver` finden sich in Kapitel 4.2.

3.4. Kräftebasierte Layout-Verfahren

3.4.1. Terminierung der kräftebasierten Verfahren

Die in Abschnitt 2.3 vorgestellten Verfahren lassen noch die Frage nach der Terminierung offen. Ideal wäre, wenn ein Verfahren im Optimum, also bei Gleichheit der anziehenden

und der abstoßenden Kräfte, terminieren würde. Der Idealfall ist für ein approximatives Verfahren jedoch schwer zu realisieren, so dass man die Berechnung abbricht, wenn beispielsweise die Summe der Kräfte oder die maximale Knotenkraft kleiner einem ϵ-Wert ist. Ideen hinsichtlich der Terminierung werden nachfolgend dargelegt und durch Beispiele beschrieben.

Terminierung von Fruchterman-Reingold

Ein Problem bei der praktischen Anwendung des Algorithmus von Fruchterman-Reingold tritt dann auf, wenn das Abbruchkriterium aufgrund von Oszillationen der Zeichnung nie erfüllt werden kann. Gegeben seien zwei Knoten, die über eine Kante miteinander verbunden sind und deren Abstand voneinander eine Ideallänge 1.9 haben soll. Beträgt nun der aktuelle Abstand der beiden Knoten voneinander zwei, und das Verfahren verschiebt die Knoten konstant mit 0.1, bewegen sich die Knoten infolge der anziehenden Kräfte aufeinander zu, und der neue Abstand beträgt dann 1.8. Dieser Wert liegt nun unter der Ideallänge. In Folge dessen werden die Knoten durch die abstoßenden Kräfte im nächsten Schritt in ihre Ausgangslage zurück bewegt. Das System oszilliert.

Um dem entgegenzuwirken, lässt sich die Verschiebung als eine abnehmende Funktion von der Anzahl der Iterationen definieren (*cooling schedule*):

$$d(v) = \min(W_t, \|F(v)\|)$$

mit

$$W_{t+1} := \text{cool}(W_t) = \alpha^{t+1} * W_0$$

wobei t die Anzahl der Iterationen sind, W_0 der Startwert ist und mit $0.95 \leq \alpha \leq 0.995$ der empfohlene Abkühlungsfaktor als Parameter festgelegt wird. Für eine detaillierte Darstellung sei an dieser Stelle wieder auf das Vorlesungsskript von Brandenburg verwiesen [Bra02].

Ist der Betrag der Knotenkraft kleiner als der Wert aus dem *cooling schedule*, bedarf es keiner Verschiebung mehr in Größenordnung von W_t, daher wird das Minimum aus beiden Werten gewählt. Ist α kleiner 0.95, so kann das Verfahren sehr schnell in einem lokalen Minimum stecken bleiben, da die Verschiebung dann sehr schnell sehr klein wird. Wie sich die Wahl des Abkühlungsfaktors auswirkt, soll an der Graphendarstellung 3.6 und an den Diagrammen 3.7 und 3.8 demonstriert werden. Als Eingabegraph wurde eine Binärbaumstruktur gewählt.

Abbildung 3.6 zeigt das resultierende Layout des Graphen nach 100 Iterationen bei einem gewählten Wert für $\alpha = 0.8$. In Diagramm 3.7 ist zu erkennen, dass die maximale Knotenkraft ab ca. 15 Iterationen nicht mehr abnimmt. Der Abkühlungsfaktor wurde zu klein gewählt, das Verfahren ist in einem lokalen Minimum steckengeblieben.

Wählt man dagegen ein α von 0.995, konvergiert das Verfahren nur sehr langsam, aber stetig gegen den Idealwert Null, wie Diagramm 3.8 bei diesem Beispiel zeigt.

Anzumerken bleibt noch, dass das Verfahren auch bei einem empfohlenen Abkühlungsfaktor von 0.95 bei 250 Iterationen nicht mehr weiter gegen Null konvergiert. Die maximale Knotenkraft beträgt dann jedoch nur 2.5. Für Werte mit $\alpha > 0.995$ zeigen sich keine Verbesserungen mehr hinsichtlich der Konvergenz.

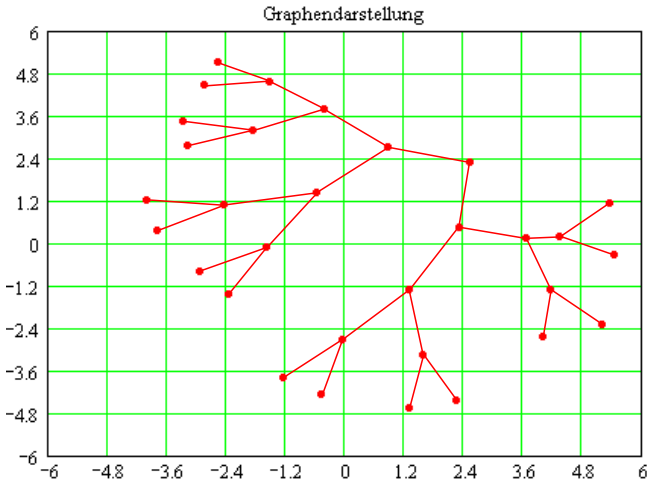


Abbildung 3.6.: Anwendung des Verfahrens von Fruchterman-Reingold nach 100 Iterationen mit einem Abkühlungsfaktor von 0.8

Abbildung 3.7.: Größe der maximalen Knotenkraft über 100 Iterationen bei einem Abkühlungsfaktor von 0.8

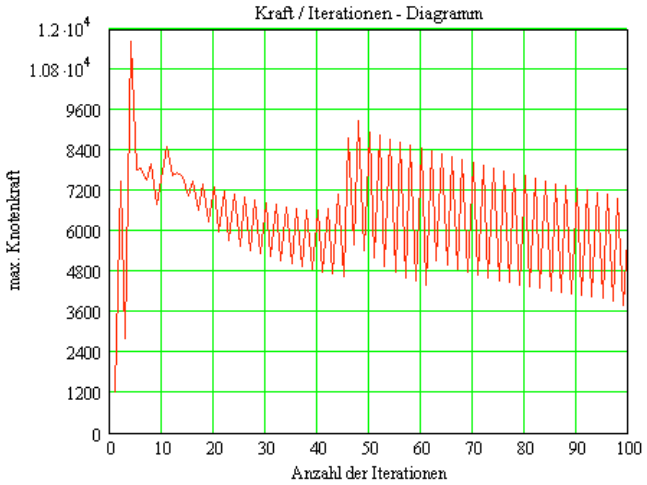


Abbildung 3.8.: Größe der maximalen Knotenkraft über 100 Iterationen bei einem Abkühlungsfaktor von 0.995

Eine weitere Methode für eine Verschiebungsfunktion beschreibt das Verfahren nach Frick [FLM95]. Es definiert keine globale, sondern eine knotenspezifische Verschiebung. Für jeden Knoten wird in jeder Iteration die aktuelle Knotenkraftrichtung mit der vorherigen verglichen. Sind die beiden Knotenkraftvektoren entgegengesetzt gerichtet, oszilliert der Knoten und die knotenspezifische Verschiebung wird reduziert. Zeigen die beiden Kraftvektoren in die gleiche Richtung, wird der Knoten in die korrekte Richtung verschoben und die Knotenverschiebung wird vergrößert. Das Verfahren kann aber nicht nur Oszillation detektieren, sondern auch Rotation. Stehen die beiden Kraftvektoren im rechten Winkel zueinander, bedeutet dies, dass der Knoten um eine Position rotiert. Auch hier wird dann die Knotenverschiebung verringert. Für dieses Verfahren muß für jeden Knoten der vorherige Knotenkraftvektor und die Knotenverschiebung gespeichert werden.

Auch wenn gegen das Oszillieren oder Rotieren Vorkehrungen getroffen wurden, können die Verfahren dennoch lokale Minima erreichen, die sie dann nicht mehr verlassen können. Addiert man eine kleine zufällige Kraft (*noise*) auf die einzelnen Knotenkräfte, erhöht man die Robustheit gegen solche Effekte.

Implementiert wurde die Verschiebungsmethode nach Frick, wobei hier auf die einzelnen Knotenkräfte ein *noise* aufaddiert wurde. In Kapitel 4.3.2 wird diese Wahl begründet.

Terminierung Kamada-Kawai

Ähnlich dem Verfahren von Fruchterman-Reingold muss auch beim Verfahren von Kamada-Kawai die Verschiebung reduziert werden, um ein Oszillieren oder Rotieren zu vermeiden. Dazu wurde auch hier die Methode nach Frick implementiert. Ein Aufaddie-

ren einer zufälligen Kraft auf die Knotenkräfte zeigte hier keine Vorteile.

3.4.2. Implementierung der Grundkomponenten

Ursprünglich wurden Implementierungen sowohl für die ungewichteten als auch für die gewichteten Graphen-Layout-Algorithmen von Kamada-Kawai und Fruchtermann-Reingold geplant. Schließlich entschloss man sich aber auf eigene Implementierung der ungewichteten Verfahren zu verzichten, da sie als Spezialfälle der gewichteten Version leicht durch geeignete Parameterwahl mit dem gewichteten Verfahren behandelt werden können. Offensichtlicher Nachteil dieser Vorgehensweise ist zwar, dass die Boolesche Adjazenzmatrix des Eingabegraphen in Gleitkommadarstellung repäsentiert wird und somit mehr Speicherplatz als nötig belegt ist, aber angesichts der zu erwartenden Graphenkomplexität der Eingabeinstanzen mit maximal 1000 Knoten wurde zu Gunsten einer universellen Implementierung entschieden.

Als wesentliche Datenstrukur für die Darstellung und Verarbeitung der Graphenstrukturen kommt die `DenseMatrix`-Containerklasse zum Einsatz. Mit ihrer Hilfe werden sowohl Adjazenzmatrizen, Koordinaten der Graphen-Knoten im Raum, Kantengewichte sowie die Zwischenergebnisse und Endresultate für die iterative Berechnung der Kräfte und Distanzen der Knoten untereinander repräsentiert. Diese Entscheidung gegen die Implementierung mit Hilfe der Containerklasse `SparseMatrix` wurde getroffen, weil sie in der vorliegenden Version keinen Vorteil für die Anwendung erbrachte.

Fruchterman-Reingold

Die vollständige Berechnung der quadratischen Differenzen, die zu Beginnn des Verfahrens benötigt werden (vgl. Abschnitt 2.3), erfolgt in der Implementierung der Klasse `NodeDistance`. Mit `repulsive_force_range` kann für diese Klasse ein bestimmter Wert festgelegt werden, der die maximale Wirkungsreichweite der abstoßenden Kräfte bestimmt. Damit kann verhindert werden, dass im Falle eines nichtzusammmenhängenden Graphen die Zusammenhangskomponenten immer weiter auseinanderdriften. Die Neuberechnung der Knotenpositionen wird im wesentlichen in der Klasse `Positions` realisiert.

Kamada-Kawai

Während bei dem gewichteten Verfahren von Fruchterman-Reingold für n Knoten jeweils alle n^2 vielen quadratischen Differenzen berechnet werden, ist die vollständige Bestimmung der Differenzen beim gewichteten Kamada-Kawai nur einmal zu Beginn des Verfahrens erforderlich, weil in den Iterationen jeweils nur ein Knoten verschoben wird und sich daher nur $n - 1$ Distanzen der Knoten zueinander ändern.

Da bei dem gewichteten Kamada-Kawai die optimale Distanz (vgl. Abschnitt 2.3.1) benötigt wird, wird sie zu Beginn des Verfahrens mit Hilfe der Klasse `BreadthFirstSearch` berechnet. Zugleich wird damit geprüft, ob der Eingabegraph zusammenhängend ist. Falls nicht, kann das Verfahren auch nicht angwandt werden und eine entsprechende Fehlermeldung wird ausgegeben.

Genau wie beim Fruchterman-Reingold Verfahren wird die Neuberechnung der Knotenpositionen in der Klasse `Positions` realisiert. Da beim Kamada-Kawai Verfahren nur ein Knoten verschoben wird, sind dort vorwiegend Operationen auf Vektoren der Länge zwei bis drei anzutreffen und Optimierungen über Basisoperationen der linearen Algebra Bibliothek kaum möglich. Daher wurden zwei zeitintensive Funktionen ausgelagert, um sie dann wie auch im Fall der Klassen `NodeDistance` und `BreadthFirstSearch` gesondert zu optimieren.

3.4.3. Implementierung der Evolving Graphs

Im Abschnitt 2.3.3 auf Seite 28 wurde die Funktionsweise von *Evolving Graphs* dargestellt und gezeigt, dass nur geringfügige Eingriffe in die Layoutverfahren nötig sind, um *Evolving Graphs* zu unterstützen. Hierzu werden einige Datenstrukturen benötigt, die die Daten sowohl für *Evolving Graphs* als auch für gewöhnliche Graphen kapseln und dann beschrieben, welche Änderungen in den Layoutverfahren gemacht werden, um Graphen zu layouten, die durch diese Struktur gegeben sind.

Grundlegende Datenstrukturen

Im folgenden wird die Impementierung der benötigten Klassen beschrieben. Alle vorgestellten Klassen sind generisch und werden mit dem Typ der enthaltenen Daten, z.B. `EvolvingGraph<double>` spezialisiert. Die Klasse `Node` kapselt die Daten zu einem Knoten. So ist es leicht möglich die Knoten um weitere Eigenschaften wie beispielsweise eine Beschriftung zu ergänzen, wenn dies erforderlich wird. Wichtig ist die Eigenschaft `id`, die den Schlüssel eines Knotens angibt.[4]

Die Klasse `AbstractGraph` bildet die Grundlage für `Graph` und `EvolvingGraph` und beinhaltet gemeinsame Attribute und Methoden. Alle diese Methoden sind virtuell und dienen dazu, für verschiedene Arten von Graphen polymorphes Verhalten bereitzustellen. Folgende Methoden bieten einen Zeiger auf die Container, die Zugriff auf die drei Komponenten eines gewichteten Graphen bieten:

`coordinates` Eine `DenseMatrix` mit den Koordinaten der Knoten,

`node_weights` Ein `DenseVector` mit den Knotengewichten,

`edges` Eine quadratische `SparseMatrix` mit den Kantengewichten.

Diese Methoden sind virtuell, da für bestimmte Arten von Graphen mehr nötig ist als nur die internen Container zu übergeben. Nur *Evolving Graphs* benötigen Informationen über *Timeslices*. Damit jedoch in den Erweiterungen[5] für die Verfahren alle `AbstractGraph`-Unterklassen polymorph behandelt werden können, definiert `AbstractGraph` die Methoden `timeslice_index` und `same_timeslice`.

[4] Bei *Evolving Graphs* repräsentieren Knoten mit gleichem Schlüssel denselben Knoten in unterschiedlichen *Timeslices*

[5] Die Visualisierung benötigt zum Zeichnen der Graphen auch die Information, in welchem *Timeslice* ein Knoten liegt und ob eine Kante zwischen zwei verschiedenen *Timeslices* verläuft.

Die Klasse `Graph` erbt von `AbstractGraph` und kapselt somit die Knoten mit ihren Koordinaten und den Knotengewichten sowie die Kantengewichte eines *ungerichteten Graphen*. Sie ist primär für die Konstruktion der einzelnen *Timeslices* konzipiert, aber nicht auf diese beschränkt. So können unabhängig von *Evolving Graphs* auch gewöhnliche Graphen gekapselt und mit den Verfahren gelayoutet werden. Die Erzeugung der Graphen wird durch `Graph` dann einfacher als das Erzeugen und Befüllen der Datencontainer per Hand.

`Graph` erstellt die für die Daten benötigten Container in der richtigen Größe: Für einen Graphen mit n Knoten in d Dimensionen werden also folgende Datencontainer erstellt:

- Eine $n \times d$ `DenseMatrix` für die Koordinaten der Knoten,
- ein n-elementiger `DenseVector` für die Knotengewichte und
- eine $n \times n$ `SparseMatrix` für die Kantengewichte der vorhandenen Kanten.

Da die erzeugten Container nun in der Größe festgelegt sind, muss die Anzahl der Knoten des Graphen a priori bekannt sein. `Graph` verwaltet eine Liste der Knoten, um festzustellen, welcher Knoten welchen Index in den internen Matrizen hat. Diese Funktionalität stellt, wie unter Assemblierung des Gesamtgraphen beschrieben, die Verbindung zwischen zwei Indizes der gesamten Kantenmatrix des Evolving Graph her. `Graph` bietet die Möglichkeit, die Knoten und Kanten des Graphen zu definieren.

`Graph` benutzt und überschreibt die von `AbstractGraph` geerbten Methoden `same_timeslice` nicht. Dies ist sinnvoll, da `Graph` ja einen einzelnen *Timeslice* repräsentiert oder aber einen gewöhnlichen Graphen, der *Timeslices* gar nicht kennt. Der Abschnitt „Nötige Eingriffe in die Verfahren“ auf Seite 74 beschreibt, wie die Verfahren erweitert werden, um sowohl `Graph` als auch `EvolvingGraph` zu unterstützen.

Die Klasse `EvolvingGraph` bildet das Kernstück der Infrastruktur für *Evolving Graphs*. Hier werden die *Timeslices* angelegt, die dann mit Knoten und Kanten zu füllen sind. Anschließend müssen die Informationen aus den einzelnen *Timeslices* in einen großen Gesamtgraphen assembliert werden. Dieser Gesamtgraph beschreibt das gesamte Evolving-Problem. Seine Komponenten werden über die von `AbstractGraph` geerbten und hier überschriebenen Methoden `coordinates`, `node_weights` und `edges` zur Verfügung gestellt. Somit kann man den `EvolvingGraph` einem der Verfahren übergeben und ein Layout berechnen lassen. Über die Methode `update_timeslice_coordinates` werden die neu berechneten Koordinaten aus dem Gesamtgraphen in die einzelnen *Timeslices* zurückgeschrieben. Nach dieser kurzen Übersicht folgt nun eine Beschreibung einzelner Methoden im Detail.

Die Methode `add_timeslice` fügt dem `EvolvingGraph` einen weiteren *Timeslice* am Ende hinzu. `EvolvingGraph` speichert dabei, wie viele Knoten vor dem neuen *Timeslice* schon vorhanden sind. Dies wird benötigt, um effizient festzustellen, in welchem *Timeslice* ein Index der Matrix ist. `add_timeslice` liefert außerdem eine Referenz auf den neu angelegten *Timeslice* zurück, so dass er direkt bearbeitet werden kann.

Die bisher vernachlässigte Methode `same_timeslice` von `AbstractGraph` wird für `EvolvingGraph` bedeutsam und deshalb überschrieben. Hier gibt `timeslice_index(x)`

den *Timeslice*-Index des übergebenen (Zeilen-)Index x der Knoten- oder Kantenmatrix zurück. Die Methode `same_timeslice(x,y)` gibt an, ob Knoten im selben *Timeslice* liegen, also $\tau_{t_x,t_y} = 1$ gilt (vgl. die Erläuterungen in Kapitel 2.3.3 auf Seite 28).

Assemblierung des Gesamtgraphen Bei der Assemblierung werden für die Koordinaten, die Knoten- und die Kantengewichte neue Container erstellt, die der Gesamtgröße des *Evolving Graph* entsprechen. Besteht der Evolving Graph aus m *Timeslices* und hat der *Timeslice* i mit $0 \leq i < m$ und n_i Knoten, dann ist $N_{\text{ges}} = \sum_{i=0}^{m-1} n_i$ die Zahl der Knoten des Gesamtgraphen.

Die Assemblierung des Gesamtgraphen erfolgt, wenn zum ersten Mal auf eine der Komponenten des Gesamtgraphen zugegriffen wird oder aber durch Aufruf der Methode `reassemble_graph`. Intern werden dabei die folgenden drei Methoden aufgerufen.

Die Methode `assemble_nodes` erzeugt eine `DenseMatrix` der Größe $N_{\text{ges}} \times d$ für die Knotenkoordinaten, wenn d die Anzahl der Dimensionen beschreibt. Danach werden die Zeilen der Matrizen der einzelnen *Timeslices* in die große Matrix zusammengefügt.

`assemble_node_weights` erzeugt einen `DenseVector` der Größe N_{ges} und füllt diesen dann mit den Knotengewichten der einzelnen *Timeslices*. Das Vorgehen ist analog zur Assemblierung der Koordinaten der Knoten.

`assemble_edges` ist komplexer, denn hier werden nicht nur die Kantengewichte der einzelnen *Timeslices* übernommen, sondern auch die benötigten *Intertimeslice*-Kanten hinzugefügt. Zunächst wird eine `SparseMatrix` der Größe $N_{\text{ges}} \times N_{\text{ges}}$ erzeugt. Danach wird jeder *Timeslice* betrachtet und die Kantengewichte in die Gesamtmatrix übertragen. Die einzelnen *Timeslices* bilden Blöcke entlang der Hauptdiagonalen.

Intertimeslice-Kanten können nur zwischen zwei benachbarten *Timeslices* verlaufen. Für einen *Timeslice* t_i mit $i > 0$ muss also für jeden Knoten v aus t_i geprüft werden, ob in t_{i-1} ein Knoten w mit $v.id = w.id$ vorhanden ist. Wenn ja, wird in der Gesamtmatrix die *Intertimeslice*-Kante mit dem vorgegebenen Gewicht eingefügt. *Intertimeslice*-Kanten können dabei in der Matrix nur innerhalb eines kleinen Bereichs auftreten: über dem Block von t_i ein Bereich der Größe $n_{i-1} \times n_i$ und (aufgrund der Symmetrie) links von t_i der gespiegelte Bereich.

Nötige Eingriffe in die Verfahren

Wie in der Beschreibung der *Evolving Graphs* in Kapitel 2.3.3 auf Seite 28 gezeigt, sind die zur Berechnung notwendigen Änderungen in den Verfahren minimal. Für beide Verfahren muss lediglich in der Initialisierungsphase berücksichtigt werden, dass zwei Knoten in unterschiedlichen *Timeslices* liegen können. `AbstractGraph` und seine Unterklassen stellen die nötigen Daten bereit und kann somit von `Position` unterstützt werden.

Daher stellt `Position` einen Konstruktor zur Unterstützung von `AbstractGraph` zur Verfügung, der die Arbeit dem Konstruktor der Implementierung des jeweiligen Verfahrens überlässt.

```
Positions(AbstractGraph<DataType_> & graph, DataType_ edge_length)
    : _imp(new methods::Implementation<Tag_, DataType_, GraphTag_>
      (graph, edge_length))
{ }
```

Der Parameter `edge_length` wird aktuell nicht verwendet.

Evolving Graphs für Fruchterman-Reingold In der Implementierung von Fruchterman-Reingold ist lediglich der Konstruktor zur Unterstützung von `AbstractGraph` anzupassen. Der übergebene Graph wird in seine Bestandteile zerlegt und die Koordinaten, die Knotengewichte und die Kantengewichte intern gespeichert.

Die Berechnung der Parameter für die anziehenden Kräfte erfolgt wie bereits gesehen. Daraus folgt, dass zwischen zwei durch eine *Intertimeslice*-Kante verbundenen Knoten anziehende Kräfte wirken. In die Berechnung der Parameter für die abstoßenden Kräfte geht aber die Information ein, ob die betrachteten Knoten im selben *Timeslice* liegen. Falls nicht, wird der Parameter für die abstoßenden Kräfte auf 0 gesetzt, sie werden also zwischen den Knoten eliminiert.

Evolving Graphs für Kamada-Kawai Die Anpassungen für die Implementierung Kamada-Kawai sind den eben beschriebenen sehr ähnlich. Zunächst wird auch hier der Konstruktor so abgeändert, dass die Koordinaten, Knoten- und Kantengewichte des `AbstractGraph` übernommen werden.

Bei Kamada-Kawai liegt der Unterschied für *Evolving Graphs* in der Berechnung der optimalen Graphdistanz der Knoten. Da diese in `BreadthFirstSearch` ausgelagert ist, bietet diese Klasse eine für `AbstractGraph` angepasste `value`-Methode. Diese bekommt den `AbstractGraph` übergeben. Die Besonderheit von *Evolving Graphs* geht bei der Addition des Gewichts der betrachteten Kante auf dem Pfad ein. Verläuft die Kante zwischen zwei *Timeslices*, so wird ihr Gewicht nicht berücksichtigt.

3.4.4. Architekturabhängige Optimierungen

Ideen für die Multicore-Implementierung

NodeDistance Für die MC-Version wird die Menge der Knoten in Teilmengen aufgeteilt. Die Berechnungen der Distanzen für diese Teilmengen wird dann in Threads gekapselt:

```
unsigned long rows(result.rows() / num_parts);
unsigned long rest_of_rows(result.rows() % num_parts);
unsigned long matrix_row(0);
unsigned long i(0);
for ( ; i < num_parts - 1; ++i)
{
    DenseMatrixTile<DataType_> tile(result, rows, result.columns(),
    matrix_row, 0);
    ThreeArgWrapper< MCNodeDistance<Tag_>,
    DenseMatrixTile<DataType_>,
```

```
    const DenseMatrix<DataType_>, unsigned long>
    wrapper(tile, pos_matrix, matrix_row);
    dispatched_tasks.push_back(tp->dispatch(wrapper));
    matrix_row += rows;
}
```

Die Größe der Teilmengen, in die das Gesamtproblem dabei aufgeteilt wird, bestimmt der *Partitioner*, der bereits im Abschnitt zum MultiCore-Backend erwähnt wurde.

BreadthFirstSearch Auch hier wird die Menge der Knoten in Teilmengen aufgeteilt und die Berechnungen der optimalen Distanzen für diese Teilmengen dann in Threads gekapselt.

SIMD Implementierung - Ideen für Cell und SSE

NodeDistance(SSE) Die Entfernung zwischen zwei Knoten a und b entspricht

$$d(a,b) = (a_x - b_x)^2 + (a_y - b_y)^2$$

Da die Koordinaten a_x und a_y jeweils in einer Zeile einer $N \times 2$ Matrix gespeichert sind, lassen sie sich leicht hintereinander in SSE Register laden, voneinander abziehen und quadrieren. Nach diesen Schritten liegen in einem Register die Ergebnisse von $(a_x - b_x)^2$ und $(a_y - b_y)^2$ vor. Bis SSE2 gibt es keinen Befehl um diese horizontal (d.h. in einem Register) zu addieren. Hierzu wurde es nötig, das SSE3 Intrinsic `_mm_hadd` zu verwenden. Hiermit kann die komplette Berechnung `NodeDistance` innerhalb der SSE Register geschehen. Falls die verwendete CPU kein SSE3 unterstützt steht ein alternativer SSE2 Pfad ohne horizontale Addition zur Verfügung.

NodeDistance(Cell) Auch auf dem Cell Prozessor wird die Tatsache, dass die Koordinaten a_x und a_y jeweils in einer Zeile einer $N \times 2$ Matrix gespeichert sind, ausgenutzt und entsprechend vektorisiert gearbeitet. Im Falle des Datentyps `float` werden genau genommen die beiden Skalare in einen SIMD-Vektor repliziert und somit immer mit zwei Koordinatenpaaren der Positionsmatrix in einem Schritt verarbeitet. Das Ergebnis sind zwei skalare Werte der zu berechnenden Distanzmatrix. Ähnlich wie beim normalen Matrizenprodukt ist das Vorhalten der richtigen Daten gegenüber der SSE-Variante hier aber schwieriger, da für ein a_x bzw. a_y die zwei-spaltige Positions-Matrix vollständig abgearbeitet werden muss, um alle Distanzen (und damit eine Zeile der Ergebnismatrix) berechnen und schreiben zu können. Außerdem bietet die Vektoreinheit der SPEs keinerlei horizontale Addition, weshalb am Ende das Ergebnis eines Vektors skalar addiert werden muss.

Aus diesen Gründen werden Teile der Positionsmatrix zweimal zu den SPEs übertragen. Als Erstes werden soviele zweispaltige Zeilen an ein SPE übertragen, dass das dafür berechnete n-spaltige Ergebnis gerade in einen DMA-Transfer-Block passt. Entsprechend viele Aufrufe des SPE-Programms kann dies für sehr große Matrizen zur Folge haben. Außerdem ist auf Grund dieser Implementierung die maximale berechnenbare Eingabegröße auf eine 4096×2-Positionsmatrix beschränkt, da eine Zeile des Ergebnisses dann

4096 Elemente hat, die im Falle einfacher Genauigkeit genau 16 KB entsprechen. Als Zweites wird die Positionsmatrix nochmals *double buffered* vollständig übertragen, um die Zeile(n) vollständig berechnen zu können. Das PPE berechnet einen Rest an Zeilen, die auf dem SPE keine vollständigen SIMD Vektoren mehr zur Folge gehabt hätten.

```
for (unsigned i(0) ; i < a_size / sizeof(float) ; i += 2)
{
    // Repliziere ein x,y-Paar in den xy-vektor
    vector float xy = spu_splats(a.typed[i]);
    xy = spu_insert(a.typed[i + 1], xy, 1);
    xy = spu_sub_insert(a.typed[i + 1], xy, 3);

    // Berechne die Distanzen des x,y-Paares
    // zu anderen x,y-Paaren vektorisiert
    unsigned j(0);
    for ( ; j < b_calc_size / sizeof(vector float) ; j++)
    {
        Subscriptable<float> temp =
            { spu_sub(xy, b[b_current].vectorised[j]) };
        temp.value = spu_mul(temp.value, temp.value);
        d.typed[ctr] = temp.array[0] + temp.array[1];
        d.typed[ctr + 1] = temp.array[2] + temp.array[3];
        ctr += 2;
    }

    // Berechne eventuellen Rest von einem x,y-Paar skalar.
    // unsigned rest = b_calc_size mod sizeof(vector float);
    for(unsigned k(0) ; k < rest / sizeof(float) ; k += 2)
    {
        float temp_x =
            a.typed[i] - b[b_current].typed[(j * 4) + k];
        float temp_y =
            a.typed[i + 1] - b[b_current].typed[(j * 4) + k + 1];
        temp_x *= temp_x;
        temp_y *= temp_y;
        d.typed[ctr] = temp_x + temp_y;
        ctr++;
    }
}
```

In der Variante für Fruchterman-Reingold werden die jeweils passenden Zeilen, bzw. Speicheradressen der zusätzlichen Matrizen ((inverse) Quadratische Distanzen, Kantengewichte) ebenfalls zum SPE übertragen und ebenso manipuliert und zurückgeschrieben.

An dieser Stelle folgt ein Code-Beispiel für den angesprochenen Fruchterman-Reingold-Teil. In `d` sind hier die im 1. Schritt berechneten Distanzen abgelegt. In `e` sollen die inversen quadratischen Distanzen gespeichert werden, in `f` die normalen quadratischen Distanzen und in `g` wurden die Kantengewichte übergeben. `squares` ist ein Vektor der an allen 4 Stellen das Quadrat eines Kraftparameters enthält, welcher der Funktion `NodeDistance` zuvor übergeben wurde und entspricht `square_force_range` aus der SingleCore-Variante.

```
(d < square_force_range) &&
(d > std::numeric_limits<DataType_>::epsilon()) ?
*e = 1 / d : *e = 0;
if (*g > std::numeric_limits<DataType_>::epsilon()) *f = d;
```

Der nun folgende Cell-Code entspricht daher dem vorangehenden skalaren Vorgehen aus der SingleCore-Variante:

```
for (unsigned i(0) ; i < (4 * result_part_size) /
    sizeof(vector float) ; ++i)
{
    // Berechne die if-Bedingungen vektorisiert.
    vector unsigned greater_than =
    spu_cmpgt(d.vectorised[i], num_limits);
    vector unsigned smaller(spu_cmpgt(squares, d.vectorised[i]));
    vector unsigned equal(spu_cmpeq(squares, d.vectorised[i]));
    vector unsigned smaller_than(spu_or(smaller, equal));
    vector unsigned if_and(spu_and(greater_than, smaller_than));

    // Berechne 1 / d und vollziehe den 1. Teil
    // eines Newton-Raphson Schrittes.
    Vector<float> v = { d.vectorised[i] };
    Vector<float> k = { spu_re(v.vf) };
    Vector<float> t = { spu_nmsub(v.vf, k.vf, one) };
    v.vf = spu_madd(t.vf, k.vf, k.vf);

    // Selektiere die korrekten Werte der SIMD-Vektor-Elemente.
    e.vectorised[i] = spu_sel(zeros, v.vf, if_and);
    vector unsigned greater_than2 =
        spu_cmpgt(g.vectorised[i], num_limits);
    f.vectorised[i] = spu_sel(f.vectorised[i],
        d.vectorised[i], greater_than2);
}
```

3.5. Löser für lineare Gleichungssysteme

3.5.1. Komponenten

Bei der Implementierung der Löser für lineare Gleichungssysteme (LGS) wurde auf eine einheitliche Schnittstelle geachtet. So können beispielsweise ein nicht-vorkonditioniertes Konjugierte Gradienten Verfahren und einem, das mit dem Jacobi Verfahren vorkonditioniert wird, beide über dieselbe Schnittstelle angesprochen werden, wie im folgenden am Beispiel eines CPU Aufrufs gezeigt wird:

```
ConjugateGradients<tags::CPU, methods::NONE>
```

bzw.

```
ConjugateGradients<tags::CPU, methods::JAC>
```

Es folgt eine Beschreibung der implementierten Löser.

Das Jacobi Verfahren

Als Repräsentant der Klasse von stationären, iterativen Lösern (d.h. solche, die über das zuletzt berechnete Glied $\mathbf{x}_{k-1}$ der Lösungsfolge hinaus keine weiteren Daten aus der letzten Iteration benötigen) für LGS wurde das Jacobi-Verfahren implementiert: Sei $A\mathbf{x} = \mathbf{b}$ das zu lösende System und D der Diagonalteil von A sowie $-L$ und $-U$ der strikte untere bzw. strikte obere Dreiecksanteil von A, dann ist die Jacobi Iteration gegeben durch

$$\mathbf{x}_k = D^{-1}\left((L+U)\,\mathbf{x}_{k-1} + \mathbf{b}\right) \tag{3.5}$$

Das gesamte Verfahren besteht also (in einer Iteration) aus der Hintereinanderausführung einer Bandmatrix-Vektor Operation, einer Vektor-Vektor Addition und der anschließenden elementweisen Multiplikation zweier Vektoren. Hierbei zeigt sich bereits das Problem einer allgemein wiederverwendbaren lineare Algebra (LA) -Bibliothek, die zwar auf Operationsebene optimiert, jedoch nicht das Zusammenspiel mehrerer Basisoperationen. Betrachtet man die Iteration genauer, so wird schnell klar, dass man weit bessere Ergebnisse erzielt, wenn man für eine solche Operation, im folgenden `JacobiKernel` genannt, eine eigenständige Implementierung bereitstellt. Der `JacobiKernel` wird damit zu einer Basisoperation der Numerikbibliothek, bei der um die im Bandmatrix-Vektor Produkt enthaltenen elementweisen Produkte herum noch die notwendigen zusätzlichen elementweisen Multiplikationen mit dem invertierten Diagonalvektor eingesetzt werden. D.h., dass die Operation *bandweise* multipliziert und sich $\mathbf{b}$ damit wie eines der Bänder verhält und zuletzt mit den anderen Bändern akkumuliert wird, was besonders im Hinblick auf eine SIMD-Implementierung performant ist.

Das Konjugierte Gradienten Verfahren

Für das Problem der Lösung eines LGS ist das Verfahren der Konjugierten Gradienten (*Conjugate Gradients*, CG) gegeben durch die Initialisierung

$$\begin{aligned} \mathbf{r}_0 &= \mathbf{b} - A\mathbf{x}_0 \\ \mathbf{z}_0 &= \mathbf{r}_0 \\ \mathbf{d}_0 &= \mathbf{z}_0 \end{aligned} \tag{3.6}$$

und die Iteration

$$\begin{aligned} \alpha_k &= \frac{\mathbf{z}_k^T r_k}{\mathbf{d}_k^T A \mathbf{d}_k} \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{d}_k \\ \mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k A \mathbf{d}_k \end{aligned}$$

$$\mathbf{z}_{k+1} = \mathbf{r}_{k+1} \tag{3.7}$$

$$\beta_{k+1} = \frac{\mathbf{z}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{z}_k^T} \mathbf{r}_k$$

$$\mathbf{d}_{k+1} = \mathbf{z}_{k+1} + \beta_{k+1} \mathbf{d}_k$$

Damit kann der CG Löser vollständig auf Basisoperationen aufsetzend implementiert werden. Für spezialisierte Implementierungen des CG-Verfahrens sei auf die Veröffentlichung von Strzodka und Göddeke verwiesen [SG06].

Löser mit Vorkonditionierung

Es ist manchmal sinnvoll, einen iterativen Löser für LGS *vorzukonditionieren*, d.h. man löst das Problem $C^{-1}A\mathbf{x} = C^{-1}\mathbf{b} \Leftrightarrow A\mathbf{x} = \mathbf{b}$. Dabei sollte C möglichst einfach zu invertieren sein. Vorkonditionierung *kann* die Iterationszahl verringern, d.h. die Konvergenzrate erhöhen. Multipliziert man in Gleichung (3.6) die rechte Seite mit C^{-1}, so erhält man

$$\mathbf{z}_0 = C^{-1}\mathbf{r}_0$$

und setzt man ferner in Gleichung (3.7)

$$\mathbf{z}_{k+1} = C^{-1}\mathbf{r}_{k+1}$$

so erhält man ein vorkonditioniertes CG Verfahren. Die Beschaffenheit von C hängt nun vom gewählten Verfahren als Vorkonditionierer ab. Setzt man

$$C = D \tag{3.8}$$

wobei D wieder der Diagonalanteil von A ist, so spricht man von einem Jacobi-Vorkonditionierer. Interessant hierbei ist sicherlich die Gegenüberstellung der Performanz dieses Verfahrens mit Basisoperationen auf der einen Seite und mit dem oben beschriebenen `JacobiKernel`. Das CG Gerüst wird wieder vollständig mit Basiskomponenten implementiert.

Gemischt genaue Löser

Wie bei der Flachwassersimulation, wurden auch hier gemischt genaue Löserkonfigurationen untersucht. Eine davon ist das gemischt genaue Verfeinern von Linearen Gleichungssystemen (*mixed precision iterative refinement*). Für weitere Details zu dieser Implementierung des Verfeinerungsverfahrens sei auf Göddeke et al. verwiesen [GST07]. Der Algorithmus läßt sich am Besten durch ein Flussdiagramm beschreiben, wie in Abbildung 3.9 gezeigt.

Die durch Computer errechneten Ergebnisse für Lösungen linearer Gleichungssysteme (LGS), also das Lösen von Problemen der Form $A\mathbf{x} = \mathbf{b}$ mit numerischen Verfahren,

können auf Grund von Rundungsfehlern, numerischer Instabilität des Algorithmus sowie der Fehlerfortpflanzung beliebig schlecht, d.h. mit beliebig großen Fehlern behaftet sein. Iterative Verfeinerungsverfahren sollen bereits berechnete Lösungen für $\mathbf{x}$ verbessern. Das hier vorgestellte *Mixed Precision Iterative Refinement* (MPIR) basiert zusätzlich auf der Überlegung, dass ein möglichst großer Teil der Rechnung in einfacher und nur ein kleiner Teil in hoher Genauigkeit erfolgen soll. Die Idee der iterativen Verfeinerung besteht darin, das Residuum (den Defekt)

$$\mathbf{d}_k = \mathbf{b} - A\mathbf{x}_k$$

einer bereits errechneten Näherungslösung $\mathbf{x}_k$ als rechte Seite einer Korrekturgleichung

$$A\mathbf{c}_k = \mathbf{d}_k$$

zu benutzen. Dabei wird das Residuum in hoher Genauigkeit berechnet und die Korrekturgleichung (das Gleichungssystem) in geringer Genauigkeit gelöst. Die Verfeinerung der Lösung, also $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{c}_k$ erfolgt dann wieder in hoher Genauigkeit. Die Folge der Lösungen lässt sich also berechnen aus:

$$\begin{aligned}
\mathbf{x}_0 &= 0 \\
\mathbf{d}_k &= \mathbf{b} - A\mathbf{x}_k \quad \text{(Berechnung in hoher (doppelter) Genauigkeit)} \\
A\mathbf{c}_k &= \mathbf{d}_k \qquad \text{(Lösung in geringer (einfacher) Genauigkeit)} \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{c}_k \quad \text{(Berechnung in hoher (doppelter) Genauigkeit)}
\end{aligned}$$

wobei $\mathbf{b}, \mathbf{d}_k, \mathbf{c}_k, \mathbf{x}_0, \mathbf{x}_k \in R^n, A \in R^{n \times n}$ sind. Der initiale Wert für den zu verfeinernden Lösungsvektor $\mathbf{x}_0$ wird im klassischen MPIR-Schema durch den Nullvektor vorgegeben. Oft wird aber auch ein randomisiert erzeugter Vektor als Startwert verwendet. Die Lösung der Korrekturgleichung kann prinzipiell durch alle Lösungsverfahren erfolgen. Im Hinblick auf schwach besetzte Systeme (oder genauer solche, die sich durch eine Bandstruktur auszeichnen), wurden als innere Löser iterative Verfahren verwendet, da sie sich für große, dünn besetzte Systeme anbieten. Für Näheres zu *mixed precision iterative refinement* und gemischt genauen Methoden wird auf Göddeke et al. verwiesen [GST07]. Auf eine Implementierung eines Verfahrens, dass einen direkten inneren Löser verwendet, wie etwa von Langou vorgeschlagen ([LLL+06]), wurde im Hinblick auf die beiden Applikationen verzichtet.

Eine weitere Möglichkeit, ein lineares Gleichungssystem vornehmlich in einfacher Genauigkeit zu lösen besteht darin, mit einem CG Löser nur jede k-te Iteration in doppelter Genauigkeit durchzuführen. Die besondere Beschaffenheit des CG Verfahrens sorgt in der Regel dafür, dass die Lösungen in etwa eben so gut sind, wie bei einem vollständig in hoher Genauigkeit rechnenden CG Löser. Auch hierfür wurden Tests durchgeführt (siehe Kapitel 4.4).

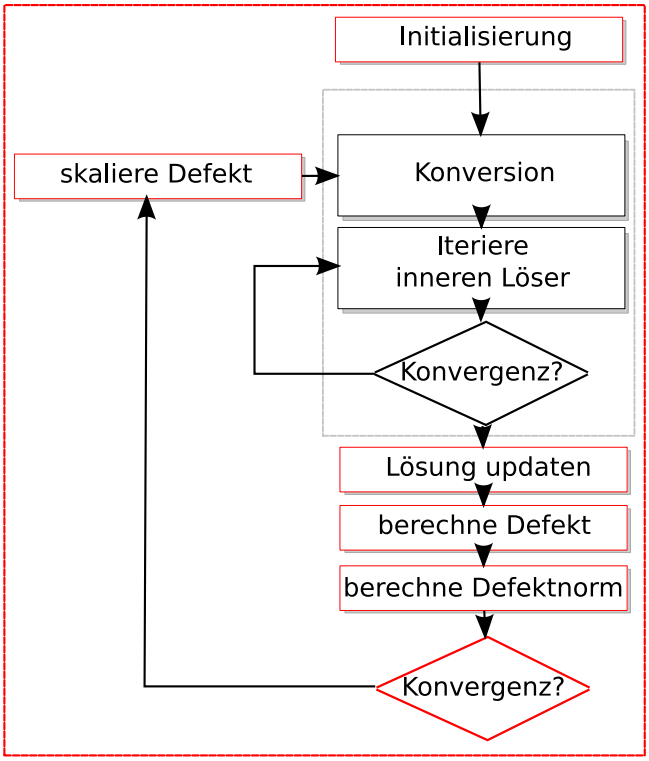


Abbildung 3.9.: Gemischt genaues Verfeinern von Lösungen linearer Gleichungssysteme: Die Defektkorrektur wird in hoher (doppelter) Genauigkeit durchgeführt, während der innere Löser in geringer (einfacher) Genauigkeit läuft.

3.5.2. SIMD-Implementierung - Ideen für SSE und Cell

Die vorgestellten Löser für lineare Gleichungssysteme sind aus Basisoperationen modular zusammengesetzt. Zusätzlich wurde für das Jacobi Verfahren eine SSE Implementierung bereitgestellt, die von den Basisoperationen unabhängig ist. Dabei wurde die bereits vorhandene Implementierung des Bandmatrix-Vektor Produktes entsprechend erweitert. Die zentrale Idee dabei ist, anstelle der durch die modulare Verwendung von Basisoperationen vorgegebene Reihenfolge der Aufrufe die zusätzlichen elementweisen Multiplikationen und Additionen direkt in derselben Schleife zu vollführen, in der das entsprechende Band abgearbeitet wird. Betrachtet man Gleichung (3.5), so sieht man, dass die Iteration im wesentlichen aus dem Bandmatrix-Vektor Produkt besteht. Die Matrixmultiplikation mit D^{-1} ist eine elementweise Multiplikation mit einem Vektor und die Addition mit dem Vektor $\mathbf{b}$ verhält sich wie ein weiteres Band, das auch mit D^{-1} elementweise skaliert wird.

3.6. Visualisierung

3.6.1. Flachwassersimulation

GnuPlot

Um schnell erste Visualisierungen der durch die Anwendung errechneten Ergebnisse zu erhalten, wurde zunächst eine einfache Nachbehandlung (`PostProcessing`) implementiert. Dieses wandelt die errechneten Höhenfeldmatrizen in ein für GnuPlot lesbares Format um und speichert sie in einer Datei. Im Anschluss stehen diese Einzelbilder zur weiteren Analyse persistent zur Verfügung.

OpenGL

Um in Echtzeit die tatsächliche Entwicklung der Berechnungen beobachten zu können, wurde eine auf OpenGL/Glut basierte Visualisierung der in jedem Zeitschritt errechneten Daten realisiert. Hierzu wird für jedes gezeichnete Bild der nächste Zeitschritt der Anwendung errechnet und die so erhaltene Höhenfeldmatrix gezeichnet. Der Benutzer kann zur Laufzeit das aktuelle Szenario neu starten, ein anderes Szenario auswählen und einige Parameter bezüglich der grafischen Ausgabe ändern.

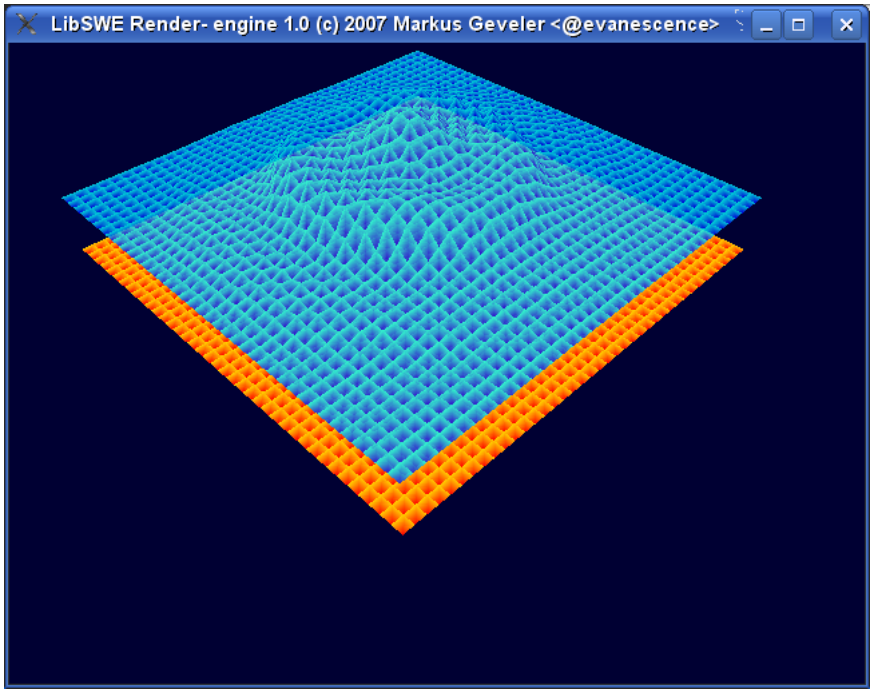


Abbildung 3.10.: Screenshot der `RelaxSolver` Visualisierung

Da auf Systemen wie zum Beispiel der PS3 Ressourcen nur eingeschränkt zur Verfügung stehen wurde weiterhin eine TCP/IP basierte Netzwerklösung entwickelt, die es ermöglicht die Berechnung auf einem Client in Echtzeit visuell zu analysieren und zu steuern.

Hierzu wird das Höhenfeld nach jedem Zeitschritt zeilenweise in TCP-Paketen an den Client verschickt. Da das Bodenprofil sich zur Laufzeit nicht ändert ist es nicht notwendig, dieses ebenfalls in jedem Zeitschritt zu verschicken. Der Client ist hier ebenfalls in der Lage, durch das Verschicken von Steuerungssignalen an den Server das Szenario neu zu starten oder ein anderes Szenario auszuwählen.

3.6.2. Graphenzeichnen

Die Visualisierung der Graphen basiert auch auf OpenGL und stellt das aktuelle Layout des Graphen dar. So ist es möglich, die Verfahren während der Ausführung zu betrachten. *Evolving Graphs* werden in Schichten gemäß den *Timeslices* zerlegt, deren Knoten auch unterschiedlich gefärbt sind.

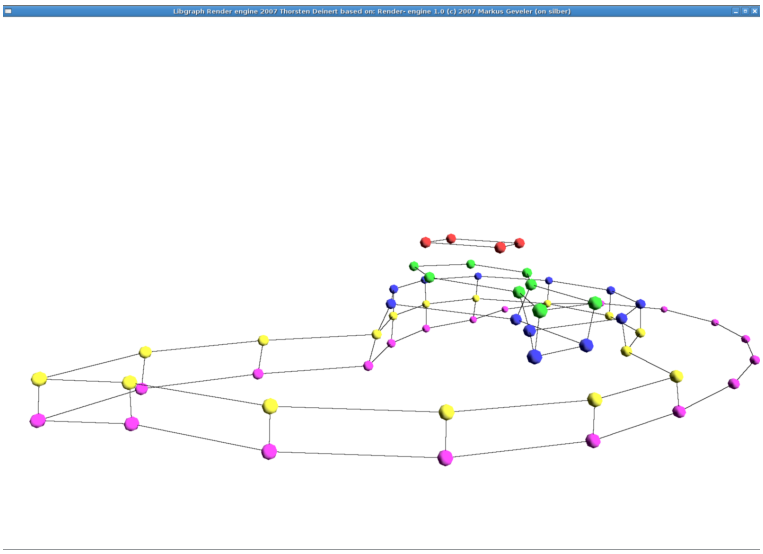


Abbildung 3.11.: Die Graph-Visualisierung

Evolving Animator

In Kapitel 3.4.3 wurden Datenstrukturen und Veränderungen im Verfahren beschrieben, die das Layout von *Evolving Graphs* ermöglichen und deren Visualisierung unterstützen. *Evolving Graphs* beschreiben jedoch nicht lediglich geschichtete Graphen, sondern den Zusammenhang zwischen Knoten über einen Zeitabschnitt. Sind m Zeitpunkte gegeben, bildet jeder Zeitpunkt einen *Timeslices*, der beschreibt, wie sich der Zusammenhang zum Zeitpunkt t_i mit $i \in \mathbb{N}, 0 \leq i \leq m-1$ darstellt.

Diese Zeitpunkte lassen sich animiert als Abfolge der *Timeslices* darstellen. Hierbei reicht es offenslichtlich nicht aus, die *Timeslices* übergangslos aneinanderzureihen, denn die Knoten würden ohne ersichtlichen Zusammenhang umherspringen, erscheinen oder verschwinden. Der Zusammenhang zwischen zwei Knoten aufeinanderfolgender *Timeslices* wird durch eine *Intertimeslice*-Kante zum Ausdruck gebracht. Besteht also zwischen

zwei Knoten v und w eine solche Kante, stellen v und w genaugenommen die Position desselben Knotens zu unterschiedlichen Zeitpunkten dar. Die (*Intertimeslice*-)Kante zwischen v und u ist somit nichts anderes als die *lineare Interpolation* dieser beiden Positionen. Man könnte also die einzelnen *Timeslices* als *keyframes* ansehen und die *Intertimeslice*-Kanten als lineare Interpolation zwischen diesen Zeitpunkten.

Ist ein Knoten im folgenden *Timeslice* nicht mehr enthalten, ist seine Position fest und muss nicht interpoliert werden. Die Klasse `EvolvingAnimator` stellt genau diese Funktionalität zur Verfügung. Aus einem gegebenen Evolving Graph mit m *Timeslices* kann `EvolvingAnimator` eine Animation vorbereiten. Danach können die interpolierten Koordinaten der Knoten für einen beliebigen Zeitpunkt t mit $t \in \mathbf{R}, t \geq 0$ dargestellt werden. Dabei werden für $t \geq m - 1$ immer die Koordinaten des letzten *Timeslice* zurückgegeben. `EvolvingAnimator` ist templatisiert und erwartet als Parameter ein Prozessor-Tag und den verwendeten Datentyp: `EvolvingAnimator<tags::CPU, double>`. Die Daten werden aus dem übergebenen `EvolvingGraph` übernommen und die Schrittweite wird festgelegt.

Die Metode `prepare_interpolation` bereitet die Interpolation der Koordinaten vor. Anschaulich gesehen wird die Animation zurückgespult und danach wird für jeden *Timeslice* eine `DenseMatrix` angelegt, die der Koordinatenmatrix des *Timeslice* entspricht. Diese wird mit den Koordinaten gefüllt, die die Knoten am Ende des *Timeslices* haben. Ein Knoten v im *Timeslice* t_i hat folgende Endkoordinaten:

- Ist $i < m - 2$ und gibt es in t_{i+1} einen Knoten w mit $(v, w) \in E$, so sind die Koordinaten von w Endkoordinaten von v. E ist hierbei die Menge der Kanten des *Evolving Graphs*.
- Andernfalls sind die Koordinaten von v auch seine Endkoordinaten.

Nach der Vorbereitung ist die Interpolation nun leicht und wird durch die HONEI-Bibliothek unterstützt. Zentral ist hier die Funktion `interpolate_timeslice`, die den gewüschten Zeitpunkt erwartet und die interpolierten Koordinaten für diesen Zeitpunkt zurückliefert. Darüber hinaus kann man zurückspulen, einen Schritt weiter gehen und überprüfen, ob das Ende der Animation erreicht ist. `EvolvingAnimator` bietet Zugriff auf die interpolierten Koordinaten und die Knoten- und Kantengewichte des aktuellen *Timeslice*.

4. Evaluation

Im folgenden soll die Korrektheit und Leistungsfähigkeit sowohl der einzelnen Operationen als auch der Anwendungen untersucht werden. Hierzu wird zunächst die Korrektheit der Operationen behandelt und es werden einige Erläuterungen zu den für die Leistungsevaluierung wichtigen Begriffen geliefert. Der Evaluierungsteil selbst gliedert sich auf unterster Ebene jeweils in die Abschnitte *Vektoroperationen*, *Matrix-Vektoroperationen* und *Matrixoperationen*. Es sollen die Themen *Numerische Korrektheit*, *Skalierbarkeit*, *Ausnutzung der Speicherbandbreite* und *Vergleich der Architekturen* behandelt werden. Zusätzlich wird in den Abschnitten zu den einzelnen Applikationen auf anwendungspezifische Fragestellungen eingegangen, wie beispielsweise auf *gemischt genaue Methoden*.

Numerische Korrektheit Auf Seite der einzelnen Operationen wird dies durch die in Kapitel 3.2.1 vorgestellte Testinfrastruktur sichergestellt. Auf Seite der Anwendungen wird dies durch diverse numerische Testverfahren sichergestellt, die in den jeweiligen Abschnitten näher erläutert werden.

Erreichen der theoretischen Leistungsfähigkeit der Architekturen Ein generelles Problem von Operationen der linearen Algebra (zumindest bei Vektoroperationen und Matrix-Vektoroperationen) ist die geringe arithmetische Intensität. Dies bedeutet, dass deutlich weniger Fließkommaoperationen als Lade- und Speicheroperationen benötigt werden, um ein Datum zu berechnen. Zum Beispiel erfordert das elementweise Addieren von zwei Vektoren für jedes Vektorelement zwei Lade- und eine Speicheroperation aber nur eine Fließkommaoperation. Der Quotient I_{ar} aus benötigten Fließkommaoperationen

Operation	Fließkommaoperationen	Ladeoperationen	Speicheroperationen
`Reduction`	n	n	1
`ScaledSum`	$2n$	$2n+1$	n
`DotProduct`	$2n$	$2n$	1
`Product(BM, DV)`	$2n$ (pro Band)	$3n$ (pro Band)	n (pro Band)

Tabelle 4.1.: Anzahl der notwendigen Fließkomma-, Lade- und Speicheroperationen für eine Eingabegröße von n. Alle Angaben beziehen sich auf die aktuelle Implementierung.

F und benötigten Transfers T, $I_{ar} = F/T$, wird als arithmetische Intensität bezeichnet. Im obigen Beispiel ist demnach die arithmetische Intensität $I_{ar} = 1/3$. Operationen, bei denen I_{ar} klein ist, werden als durch den Speichertransfer in ihrer Geschwindigkeit

begrenzt (*memory bound*) bezeichnet. Dies hängt im Einzelfall von der betrachteten Architektur ab. Im Gegensatz dazu werden Operationen mit großem I_{ar} als durch die Rechenleistung in ihrer Geschwindigkeit begrenzt (*compute bound*) bezeichnet. Des weiteren ist die theoretisch mögliche Anzahl von Fließkommaoperationen einer modernen CPU deutlich größer als das Datenvolumen, das im selben Zeitraum zwischen Hauptspeicher und CPU transferiert werden kann. Da hier die Speichertransferrate also den größten Flaschenhals darstellt, wird dieses Phänomen auch als *memory wall* bezeichnet. Für Erläuterungen zum memory wall-Problem siehe Wilkes [Wil00]. Dies gilt in gleichem Maße auch für die Cell BE.

Skalierbarkeit Man unterscheidet hierbei zwischen *starker* und *schwacher* Skalierbarkeit, wobei *starke Skalierbarkeit* den Umstand bezeichnet, dass bei gleicher Problemgröße und doppelten Ressourcen (z.B. doppelter CPU-Anzahl) die Laufzeit halbiert wird und man unter *schwacher Skalierbarkeit* versteht, dass bei doppelter Problemgröße und doppeltem Ressourceneinsatz die Laufzeit unverändert bleibt. Dabei besteht die Erwartung, dass bei Vektoroperationen auf der Cell BE bei steigender Anzahl von SPEs eine Vollauslastung der Speicherbandbreite erreicht wird und demnach auch keine weitere Skalierbarkeit darüber hinaus zu verzeichnen ist. Für die MultiCore Implementierung wird bei Vektoroperationen keine Skalierbarkeit erwartet, da diese typischerweise bereits bei Benutzung eines Threads memory bound sind. Bei Matrix-Vektoroperationen wird für die Cell BE dasselbe erwartet, da diese Operationen in geringerem Maße memory bound sind als Vektoroperationen. Für MultiCore Implementierungen ist hier ebenfalls kaum Skalierbarkeit zu erwarten, da die wiederverwendbaren Datenmengen gering sind. Matrixoperationen sollen schließlich besonders gut skalieren, da sie als compute bound gelten. Dies gilt für alle Architekturen.

Testsysteme Zur Evaluation der CELL BE wurden die Playstation 3 und ein Cellblade verwendet (siehe Tabelle 4.3). Für die Evaluation der x86 Backends wurden die in Tabelle 4.2 dargestellten Systeme herangezogen[1]. Die Auswahl dieser Systeme deckt einerseits aktuelle x86-CPUs und andererseits sowohl symmetrische Multiprozessorsysteme (SMP) als auch MultiCore-Maschinen ab. Dabei bedeutet SMP, dass mehrere eigenständige CPUs mit ihren Recheneinheiten und Caches auf einem Mainboard eingesetzt werden, während bei MultiCore-Prozessoren eine CPU aus mehreren Prozessorkernen besteht.

[1]Die Transferraten wurden mit Hilfe von RAMspeed/SMP (Linux) v3.4.1 [Hol07] ermittelt.

Architektur	Taktfrequenz	Cache	max. Transferrate	Prozessoren, Kerne je Prozessor	
Intel Core2Duo 6320	1.86 GHz	4096 kB geteilt	3054.53 MB/s	1	2
Intel Xeon E5345	2.33 GHz	2x 4096 kB pro Prozessor	2349.65 MB/s	2	4
AMD Opteron 844	1.80 GHz	1024 kB pro Prozessor	4065.41 MB/s	4	1
AMD Dual Core-Opteron 2214	2.20 GHz	1024 kB pro Kern	5350.79 MB/s	2	2

Tabelle 4.2.: Kenndaten der x86 Testsysteme

Architektur	Taktfrequenz	LS pro SPE	max. Transferrate	SPEs
Playstation 3	3.2 GHz	256 KByte	25,6 GB/s	6
Cellblade	3.2 GHz	256 KByte	$2 \cdot 25{,}6$ GB/s	$2 \cdot 8$

Tabelle 4.3.: Kenndaten der Cell Testsysteme

4.1. Operationen

4.1.1. Numerische Korrektheit

Die numerische Korrektheit wurde mit der zuvor beschriebenen Testinfrastruktur sichergestellt. Da die Fließkommaeinheiten der x86 CPUs nicht in der gleichen Genauigkeit arbeiten wie die SSE-Einheiten und die SPEs der Cell BE, ist eine Gegenüberstellung der numerischen Ergebnisse auf den jeweiligen Architekturen nötig. Es muss sichergestellt werden, dass die in einer 32-bit bzw. 64-bit Repräsentation rechnenden SSE Einheiten und SPEs annähernd dieselbe numerische Ergebnisgenauigkeit bewerkstelligen können. Da die Fehlerfortpflanzung nicht bei allen Operationen gleich stark auf die Ergebnisgenauigkeit durchschlägt, muss man dies gesondert für jede Operation tun. Es wird also explizit überprüft, ob die SSE bzw. Cell BE -Implementierung einer Operation dieselben Fehlerschranken respektiert, wie die CPU-Referenzimplementierung, welche im Rahmen von Berechnungen auf FPU-Registern von einer 80 bit Genauigkeit profitieren können.

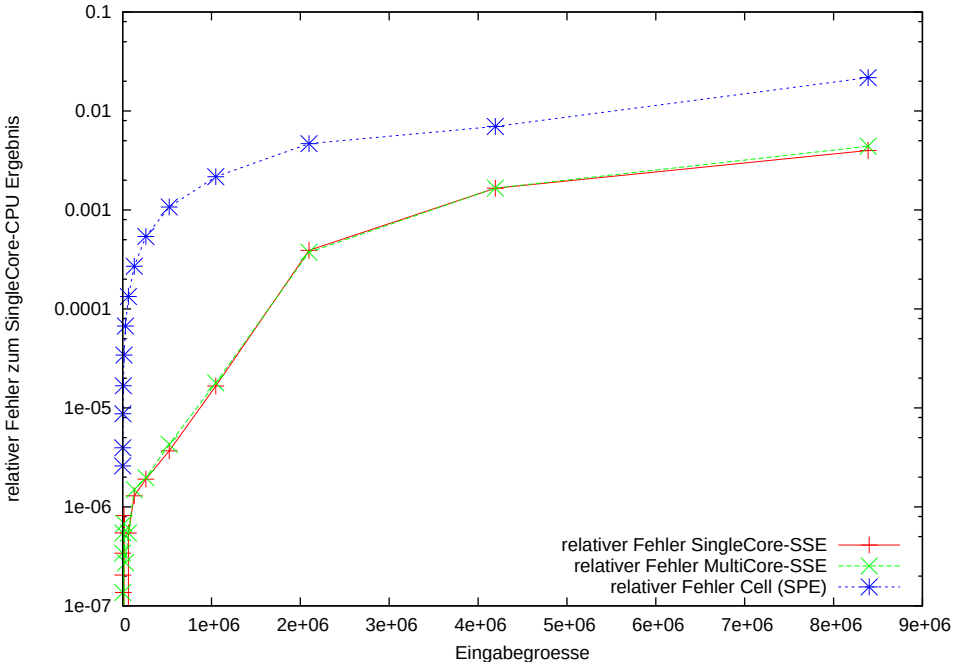


Abbildung 4.1.: Relativer Fehler gegenüber CPU-Referenzergebnissen beim Skalarprodukt (einfache Genauigkeit).

Ein weiterer wichtiger Aspekt ist hier, dass die SPEs des Cell Prozessors den IEEE 754 Standard für den Umgang mit Fließkommawerten lediglich für doppelte Genauigkeit exakt einhalten. Wie stark der Effekt der Fehlerfortpflanzung bei unterschiedlicher Berechnungsgenauigkeit sein kann, zeigt der Vergleich des numerischen Fehlers bei einem Skalarprodukt in Abbildung 4.1. Generell ist anzumerken, dass die einzelnen Implementierungen erst durch eine Applikation einem Praxistest unterzogen werden können, wie in den Abschnitten 4.2.1, 4.3.1 und 4.4.1 gezeigt wird.

4.1.2. Erreichen der theoretischen Leistungsfähigkeit der Architekturen

Starke und Schwache Skalierbarkeit

Vektoroperationen Als Vertreter dieser Klasse von Operationen sei die Operation `ScaledSum`, d.h. $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{y}$ herangezogen, bei welcher, im Fall der Cell Implementierung, ein Faktor von zwei beim Übergang von einem auf zwei bzw. von zwei auf vier SPEs bei der Leistung zu verzeichnen ist, wie in Abbildung 4.2 gezeigt. Unabhängig von der Anzahl der SPEs zeigt sich deutlich, dass für kleinere Problemgrößen der Mehraufwand durch das Cell-Backend einen wesentlichen Teil der Rechenzeit einnimmt. Erst für größere Probleme wird eine konstante Leistung erreicht. Verwendet man jedoch mehr als vier SPEs, so wird diese Leistungssteigerung nicht mehr erreicht. Es kann im Gegenteil

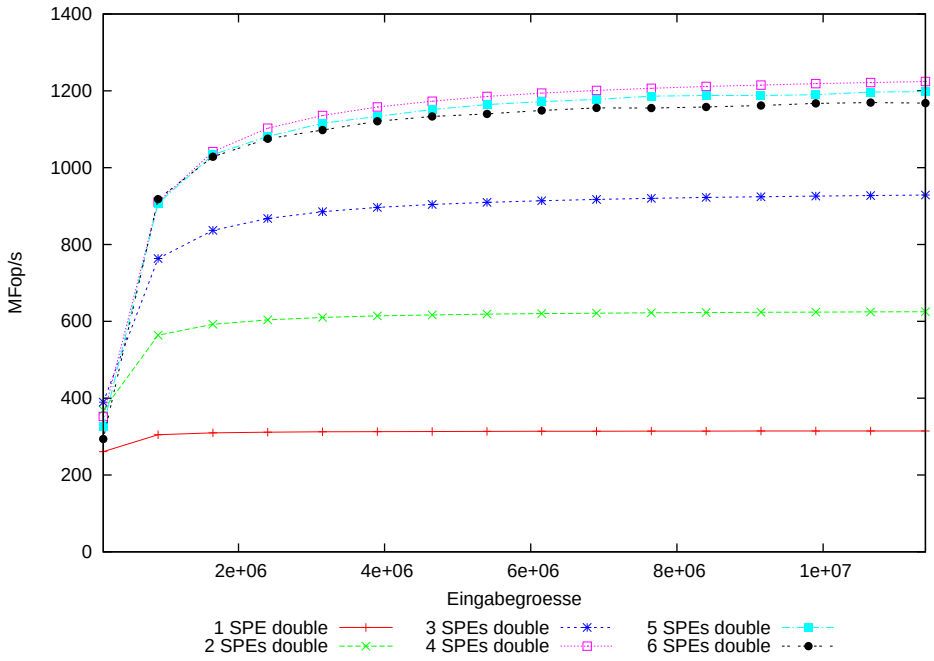


Abbildung 4.2.: Skalierung ScaledSum auf der Playstation 3.

sogar beobachtet werden, dass kleine Leistungseinbußen durch Überlastung des Bussystems hingenommen werden müssen. Diese Ergebnisse gelten für alle Vektor-Operationen, wobei die maximale Anzahl von SPEs, für die starke Skalierbarkeit noch erreicht wird, von Operation zu Operation differieren kann, siehe dazu auch Tabelle 4.4. Dabei wird deutlich, dass nur Ladeoperationen bis zu sechs SPEs den EIB gewinnbringend auslasten können und Operationen die zusätzlich Speichertransfers ausführen, den EIB schon beim Einsatz von vier SPEs auslasten. Da das Cellblade physikalisch aus zwei Cell Prozessoren besteht, ist hier die doppelte Anzahl von SPEs notwendig um beide EIBs auszulasten. Es ist allerdings noch nicht möglich zu kontrollieren, ob die genutzten SPEs wirklich zu beiden Cell Prozessoren gehören oder in welchem Hauptspeicher die Daten abgelegt sind. Dies kontrolliert allein das Betriebssystem. Verdoppelt man die Problemgröße, so gelten dieselben Aussagen: Es liegt schwache Skalierbarkeit vor und es gelten dafür dieselben Schranken bei der Anzahl der eingesetzten SPEs.

Im Fall des MultiCore-Backends erreicht man für die Operation `ScaledSum` schon mit einem Kern die maximale Rechenleistung. Das Hinzunehmen von weiteren Kernen verringert die Rechenleistung sogar geringfügig. Der Grund hierfür ist die Speicherbandbreite, die bereits ein Kern voll ausschöpfen kann. Einzig bei kleinen Problemen, die gerade noch im L2-Cache ablaufen können ist bei MultiCore ein Leistungsvorsprung zu SingleCore zu beobachten. Bei größeren Problemen kann der Cache nicht mehr genutzt werden und die Leistung wird durch die Speicherbandbreite beschränkt. Siehe hierzu Abbildung 4.3. Bei Vektoroperationen liegt also keine Form der Skalierbarkeit vor. Positiv zu bemerken

	SPEs PS3		SPEs Cellblade	
Operation	float	double	float	double
`Reduction`	6	6	12	12
`ScaledSum`	4	4	8	8
`DotProduct`	4	4	8	8
`Product(BM, DV)`	4	4	8	8

Tabelle 4.4.: Experimentell ermittelte Schranken für die Anzahl der SPEs hinsichtlich starker Skalierbarkeit.

ist jedoch, dass der Mehraufwand zur Verwendung von Threads vollständig kompensiert wird.

Matrix-Vektoroperationen An dieser Stelle soll das Bandmatrix-Vektor Produkt betrachtet werden. Die Eingaben sind Vierbandmatrizen, wie sie in der Flachwasseranwendung eingesetzt und in Kapitel 2.2.4 beschrieben werden. Die Leistungsdaten bei Einsatz verschiedener Anzahlen von SPEs sind in der Abbildung 4.4 dargestellt. Es bietet sich dasselbe Bild wie bei den Vektoroperationen, d.h. es liegt sowohl schwache als auch starke Skalierbarkeit für bis zu vier SPEs vor. Der Einsatz von sechs SPEs führt in den meisten Fällen zu derart vielen Konflikten auf dem EIB, dass lediglich eine mit drei SPEs vergleichbare Leistung erzielt wird. In seltenen Fällen scheinen sich Berechnung und Transfer der einzelnen SPEs aber so gut zu verschränken, dass eine ähnliche Leistung wie mit vier SPEs erreicht wird; was gleichzeitig der angenommenen maximalen Leistung des Backends entspricht.

Auch beim Vergleich zwischen Multicore und Singlecore sind die Ergebnisse die gleichen, wie bei den Vektoroperationen. Es liegt keine Skalierbarkeit vor, da bereits ein Kern die Speicherbandbreite vollständig auslasten kann. Die Abbildungen 4.5 stellen diese Ergebnisse dar. Dies überrascht nicht, da es sich bei dieser Operation prinzipiell auch um eine Vektor-Vektor-Implementierung handelt, da sie bandweise arbeitet.

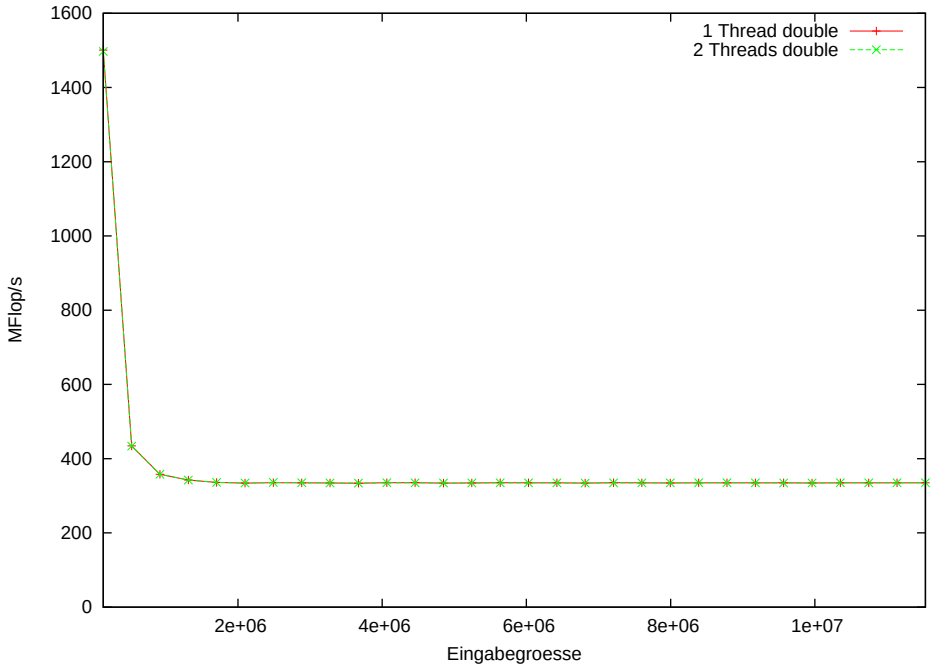


Abbildung 4.3.: Skalierung ScaledSum MultiCore auf dem Core2Duo.

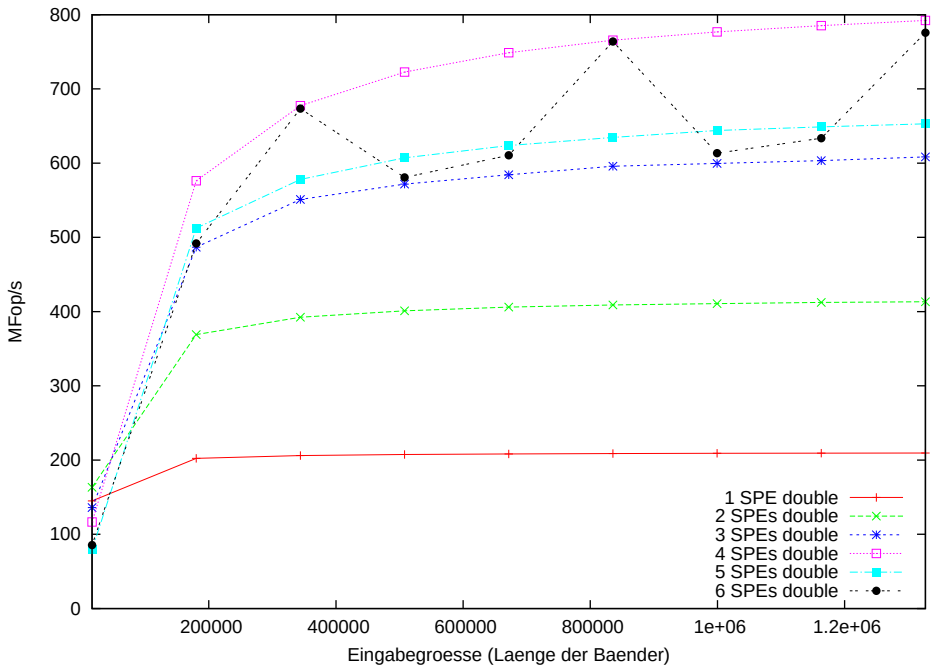


Abbildung 4.4.: Skalierung Bandmatrix-Vektor Produkt auf der Playstation 3.

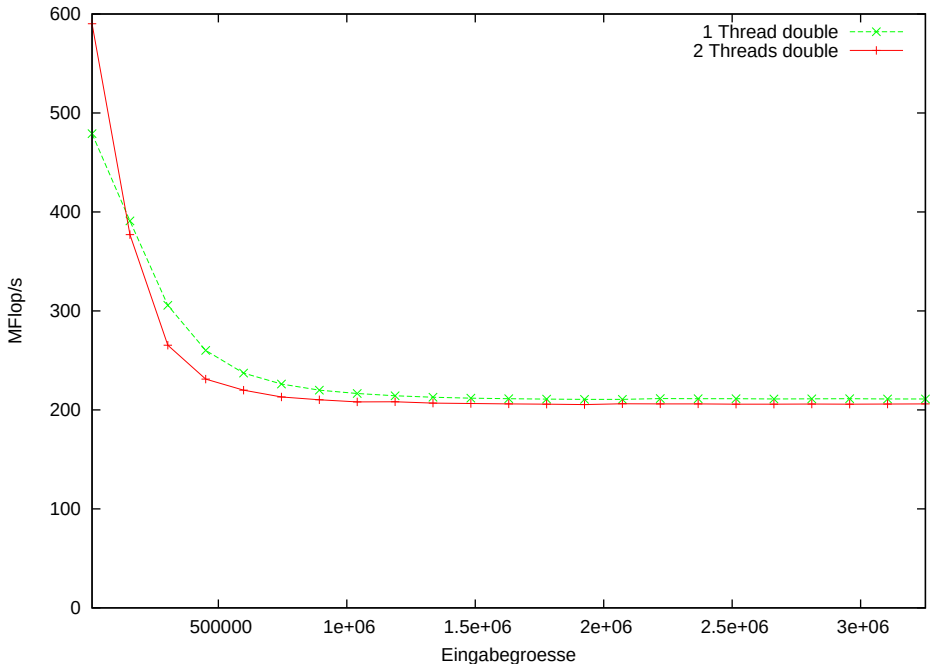


Abbildung 4.5.: Skalierung Bandmatrix-Vektor Produkt MultiCore auf dem Core2Duo.

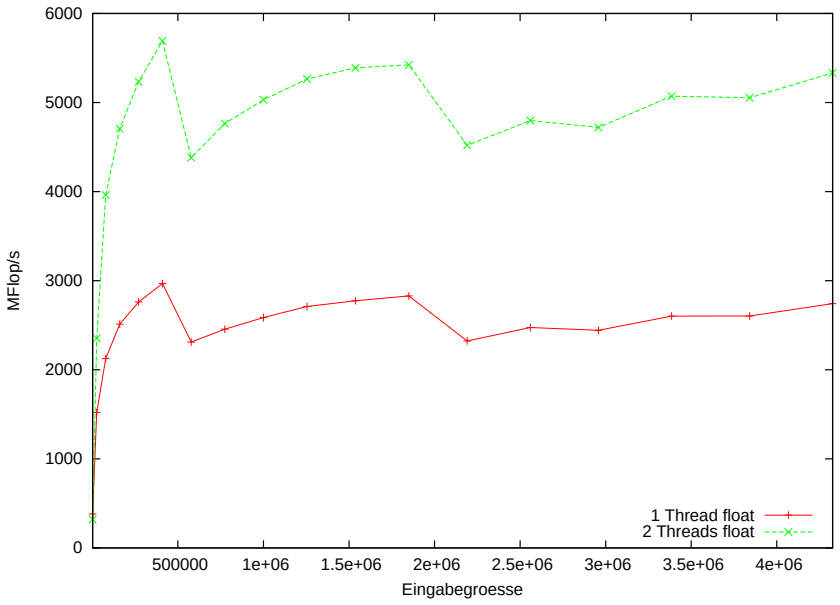


Abbildung 4.6.: Skalierung Matrix-Matrix Produkt MultiCore auf dem Core2Duo.

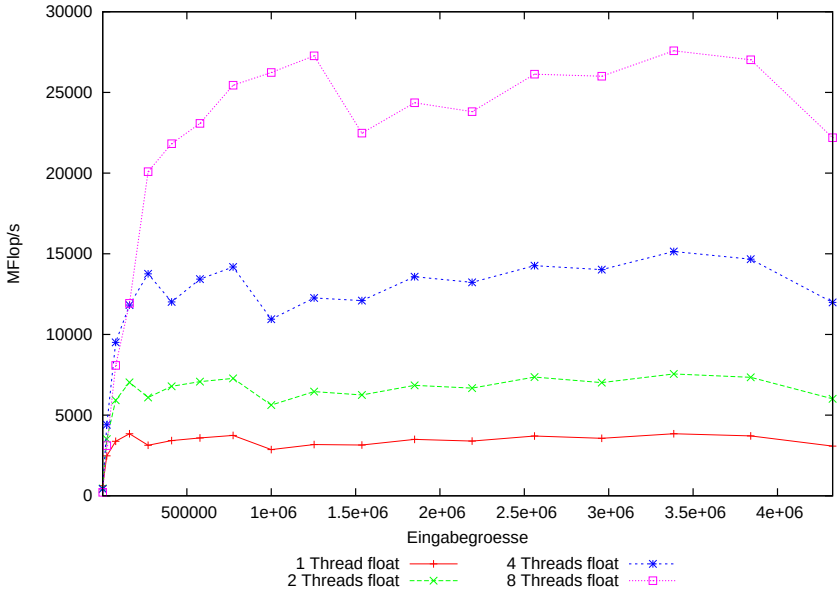


Abbildung 4.7.: Skalierung Matrix-Matrix Produkt MultiCore auf dem Intel Xeon.

Matrixoperationen In der aktuellen Implementierung für die Cell BE kann die SPE-Anzahl bei einem Matrixprodukt nicht verändert werden. Somit sind bezüglich der Skalierbarkeit keine fundierten Erkenntnisse möglich. Es hat sich jedoch gezeigt, dass die aktuelle Variante mit 4 SPEs deutlich performanter ist, als eine ehemalige Implementierung, die nur ein SPE ausnutzte. Zumindest in der aktuellen Form des Algorithmus ist auf Grund der in Abschnitt 3.1.5 und den folgenden Abschnitten beschriebenen Probleme zu vermuten, dass die Verwendung von 8 oder mehr SPEs zwar einen Laufzeitgewinn zur Folge hätte, dieser aber auf Grund der Probleme mit der Speicherbandbreite klein ausfallen würde.

Das MultiCore Backend hingegen ist beim Matrixprodukt auf Grund der hohen Wiederverwendbarkeit an Daten, die durch die verhältnismäßig großen Caches gut ausgenutzt werden kann, wesentlich weniger an die Speicherbandbreite gebunden. Es hat sich allerdings gezeigt, dass eine zu granulare Unterteilung des Problems, die stärkere Synchronisation zwischen den Threads zur Folge hat, zu schlechteren Ergebnissen führt als eine, wie hier, grob gewählte Aufteilung: Es wurden so viele Teilprobleme erzeugt, wie Threads im `ThreadPool` existieren. Hierzu wurden die linke Operanden- und entsprechend die Ergebnismatrix zeilenweise unterteilt.

Es wird also ausgenutzt, dass die Teilprobleme innerhalb des jeweiligen SSE-Aufrufes weiter zerlegt werden. Die Abbildungen 4.6 und 4.7 zeigen stellvertretend für die übrigen x86 Referenzarchitekturen das Skalierungsverhalten am Beispiel des Core2Duo und des Intel Xeon. Die Schwankungen im Kurvenverlauf lassen sich durch die, beim SSE Backend vorgestellte, Unterteilungsstrategie erklären (vgl. zugehörigen Abschnitt in Kapitel 3.1.3), die für beliebige Seitenlängen der Matrizen konzipiert ist. Es liegt starke

Skalierbarkeit vor.

Bezüglich schwacher Skalierbarkeit wird im Schnitt ca. ein Faktor von 1,6 erreicht, was hauptsächlich auf die kubische Laufzeit des Matrixproduktes aber auch auf den Mehraufwand bei der Unterteilung der doppelten Eingabegröße (Verdoppelung der Anzahl an Matrixelementen) zurückzuführen ist.

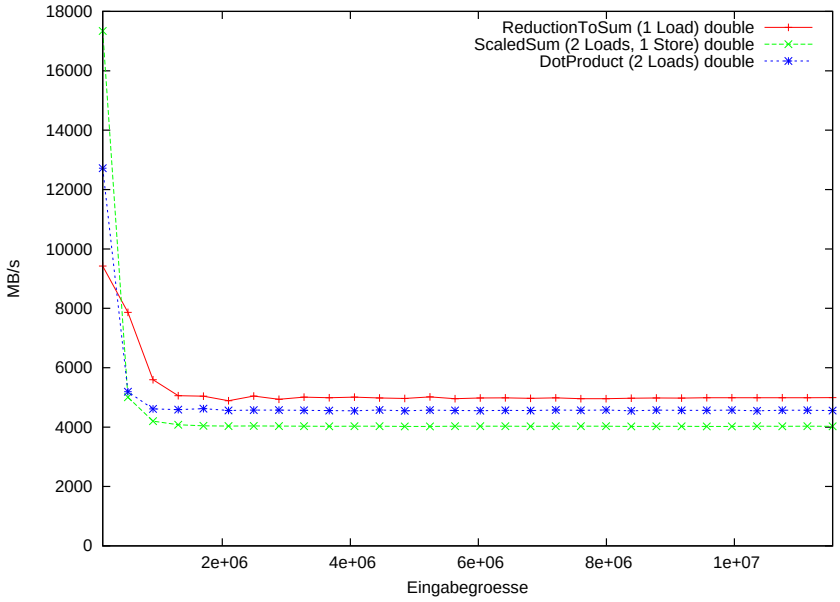


Abbildung 4.8.: Vergleich der Transferraten für Operationen mit unterschiedlicher Anzahl von Loads und Stores auf dem Core2Duo.

Auslastung der Speicherbandbreite

Vektoroperationen Die tatsächliche Auslastung der Speicherbandbreite ohne Ausnutzung der Caches kann am besten bei Operationen ohne Datenwiederverwendung beurteilt werden, da der Einsatz von Caches die Ergebnisse sonst überdeckt. Für SSE kann leider keine konkrete Aussage getroffen werden, da, wie Abbildungen 4.8 - 4.11 verdeutlichen, kein erkennbarer Bezug zu den, in der Architekturübersicht (vgl. Tabelle 4.2) ermittelten, Transferraten hergestellt werden kann. Ebenso ist keine allgemeine, architekturübergreifende Tendenz in den genannten Abbildungen erkennbar, die das Verhältnis der drei betrachteten Vektoroperationen bezüglich der Transferraten beschreiben könnte. Es ist allerdings festzuhalten, dass SSE für Vektoroperationen die gleiche Leistung wie die später vorgestellte GotoBLAS Implementierung erreicht, wie am Beispiel von `ScaledSum` gezeigt wird (vgl. Kapitel 4.1.2 Abb. 4.17). Da die MultiCore-Implementierungen auf dem SSE-Backend aufsetzen, gilt für diese Entsprechendes.

Für die Cell-Implementierung gilt, dass die Transferrate bei allen oben genannten Operationen mit wachsender Problemgröße etwa gegen 14 GB/s konvergiert. Die Er-

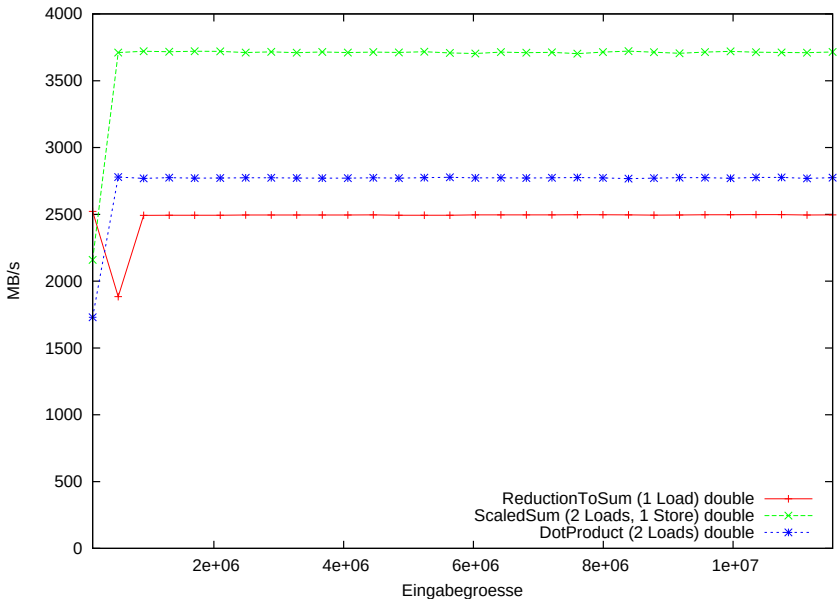


Abbildung 4.9.: Vergleich der Transferraten für Operationen mit unterschiedlicher Anzahl von Loads und Stores auf dem AMD DualCore.

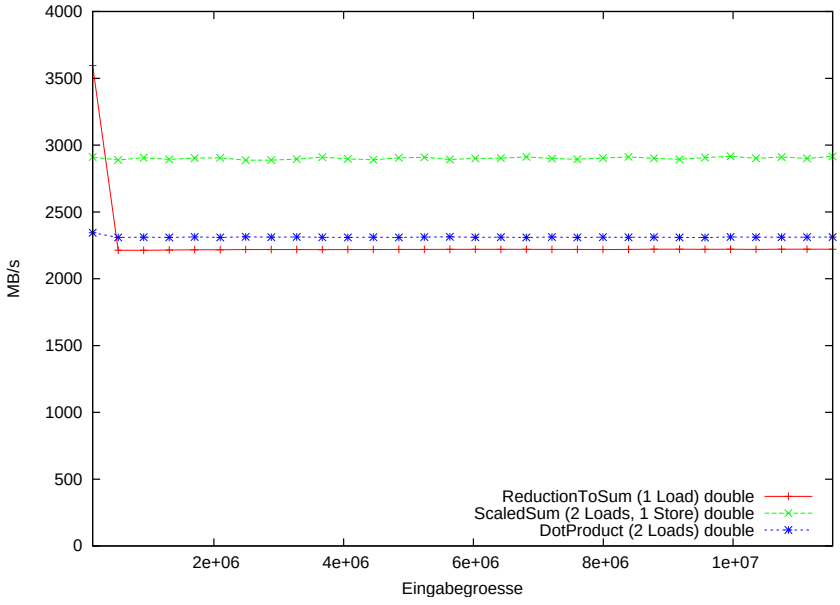


Abbildung 4.10.: Vergleich der Transferraten für Operationen mit unterschiedlicher Anzahl von Loads und Stores auf dem AMD Opteron.

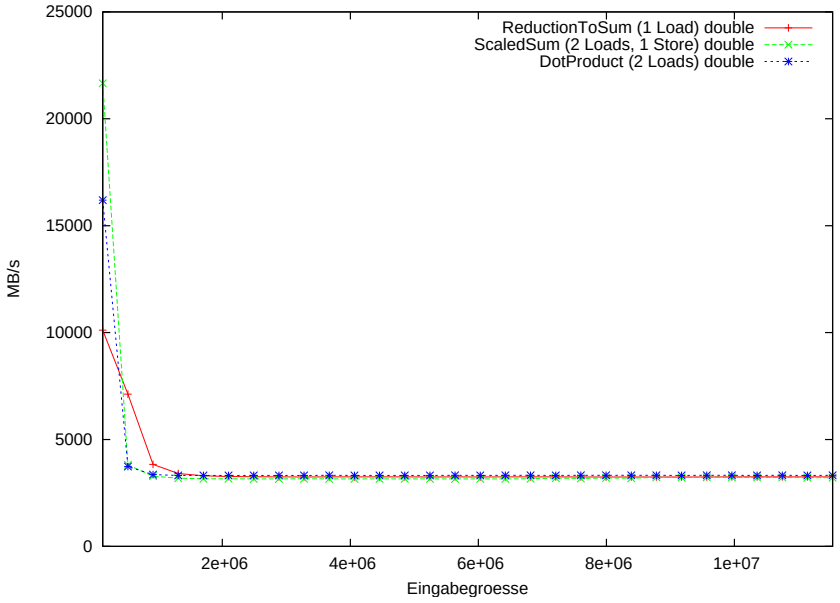


Abbildung 4.11.: Vergleich der Transferraten für Operationen mit unterschiedlicher Anzahl von Loads und Stores auf dem Intel Xeon

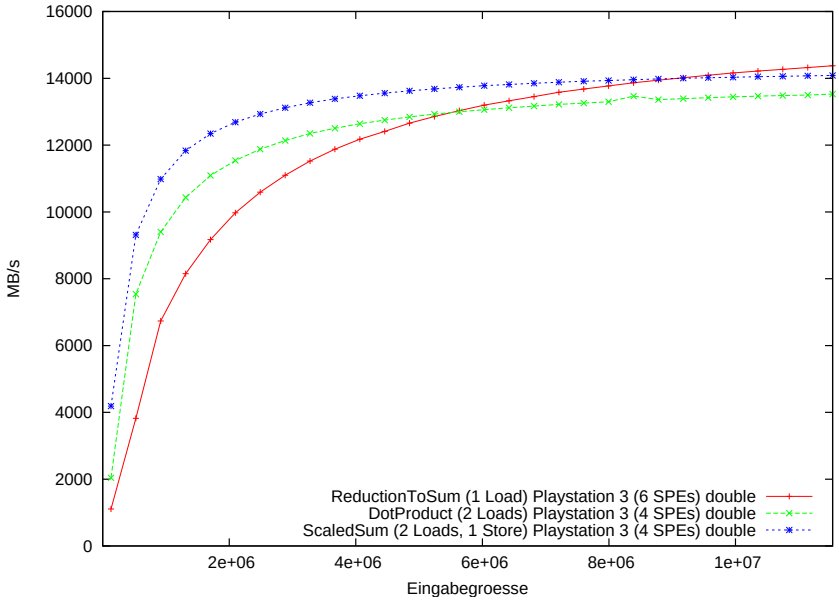


Abbildung 4.12.: Vergleich der Transferraten für Operationen mit unterschiedlicher Anzahl von Loads und Stores auf der Playstation 3.

gebnisse sind in den Abbildungen 4.12 dargestellt. Dies entspricht in etwa 54 % der theoretisch möglichen 25,6 GB/s. Die Leistungsgrenze der Cell BE kann mit der aktuellen Implementierung also nicht erreicht werden. Tests, die durchgeführt wurden, um diesen Umstand genauer und ausserhalb von HONEI zu untersuchen, zeigten, dass unter gewissen Umständen die theoretisch mögliche Transferrate erreicht werden kann. Dabei wurde festgestellt, dass ein wesentlicher Anteil der Laufzeit auf das Anstoßen der jeweiligen Operationen entfällt. In dem zugrundeliegenden Programmiermodell werden in einer Anwendung tendenziell viele einzelne Operationen aufgerufen, so dass dieser initiale Aufwand sehr oft zu Buche schlägt. Aus Zeitgründen konnte die Portierung dieser Implementierung nach HONEI im Rahmen der PG nicht mehr erfolgen.

Matrix-Vektoroperationen Da sowohl für das SSE-, das MultiCore- als auch für das Cell-Backend das Bandmatrix-Vektorprodukt wie eine bandweise Vektoroperation implementiert ist, gelten hier dieselben Aussagen wie bei Vektoroperationen.

Matrixoperationen Da die Datenwiederverwendung bei diesen Operationen typischerweise hoch ist, kann hier ohne eine genaue Analyse des Cache-Verhaltens keine gültige Aussage über die Bandbreitenausnutzung getroffen werden.

Vergleich der Architekturen

Obwohl die derzeitige Implementierung des Cell-Backends das Hardwarepotential nicht voll ausnutzen kann, zeigen die in den folgenden Abschnitten dargelegten Leistungswerte deutlich, dass auch damit eine wesentlich höhere Leistung erzielt wird, als auf herkömmlichen Systemen.

Vektoroperationen Stellvertreter ist hier wieder die Operation `ScaledSum`. Ein Direktvergleich der x86 Architekturen wird in Abbildung 4.13 bereitgestellt, wobei der Core2Duo die beste Leistung erzielt. Abbildung 4.14 zeigt, dass bereits die Playstation deutlich schneller ist, als die schnellste hier behandelte x86 Architektur. Da das Cellblade physikalisch aus zwei Cell Prozessoren mit ebenfalls getrennten Hauptspeichern besteht, sieht man deutliche Schwankungen in der Leistung. Dies liegt daran, dass bisher nicht beeinflusst werden kann, in welchem der beiden Hauptspeicher die jeweiligen Daten liegen und ob eine SPE auf diesen Daten arbeitet, zu der der jeweilige Hauptspeicher gehört. Die Verwaltung der Ressourcen obliegt also völlig dem Betriebssystem, welches auf dem Cellblade in einer deutlich älteren Version vorliegt.

Matrix-Vektor Operationen Für die SSE Implementierung als auch für die MultiCore sowie die Cell Implementierung gelten hier dieselben Aussagen wie bei Vektoroperationen. Vergleicht man direkt die Leistung einer Vektoroperation mit einer Matrix-Vektor Operation, so stellt man fest, dass diese in etwa identisch sind - mit leichtem Vorteil für die Vektoroperation, da diese keinen zusätzlichen Verwaltungsinformationen benötigen.

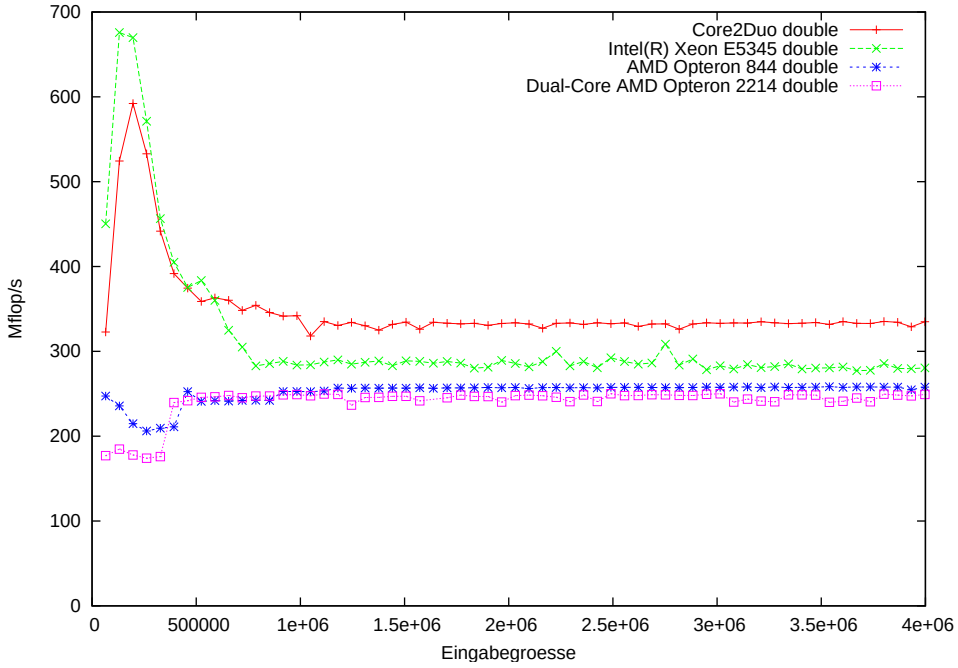


Abbildung 4.13.: ScaledSum - Vergleich der x86-Architekturen.

Matrixoperationen Für die x86 Implementierungen gilt hier wiederum, dass der Intel Xeon sich gegen die anderen Vertreter aufgrund seiner höheren Anzahl an Kernen bei schnellerer Taktfrequenz klar durchsetzt, wie in Abbildung 4.15 gezeigt.

Wie in Abbildung 4.16 zu sehen ist, steigt die Leistung der Cell Implementierung bis zu mittleren Eingabegrößen stark an und ist auf ihrem Höhepunkt bei einer 1000x1000 Matrix sogar etwas schneller als die MultiCore-SSE-Implementierung auf dem Core2Duo. Danach konvergiert die Leistung gegen etwa 4 GFlop/s, während die MultiCore-Implementierung beim Core2Duo 5 GFlop/s erreicht. Auf Grund der erforderlichen Unterstützung aller möglichen Formen der Matrizen (d.h. nicht nur quadratische Matritzen oder Beschränkung auf gewisse Eigenschaften wie Spalten- oder Zeilenanzahlen) sowie den bereits in Abschnitt 3.1.5 beschriebenen Herausforderungen bezüglich Speichertransfers und Ausrichtung von Daten am Speicher, wurde eine kachel-orientierte fest auf vier SPEs zugeschnittene Implementierung gewählt. Es ist zu vermuten, dass eine Variante mit mehr SPEs an die Grenzen des Bussystems stoßen wird, da bereits bei den einfachen Vektoroperationen eine gute Skalierung nur bis zu 4 SPEs festgestellt werden konnte. Dass nicht die bestmöglichen Ergebnisse erreicht werden, kann darüber hinaus noch mit zwei weiteren Aspekten begründet werden. Wie bereits beschrieben ist diese Operation sehr speicherintensiv. Selbst für die Vektoroperationen konnte allerdings nur etwa die Hälfte der möglichen Speicherbandbreite erreicht werden. Das Matrixprodukt leidet besonders unter dieser Einschränkung, weil sehr viel Speichertransferaufwand betrieben werden muss um für jede verarbeitete Zeile der in Abbildung 3.1 als A bezeichneten Ma-

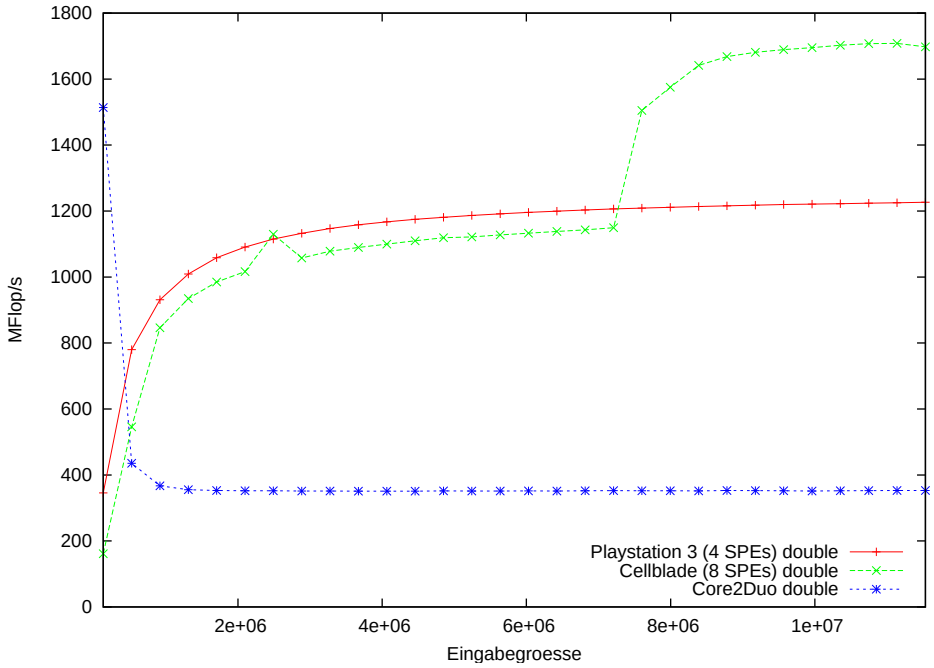


Abbildung 4.14.: ScaledSum - Vergleich der Architekturen Playstation 3/Cellblade/Core2Duo.

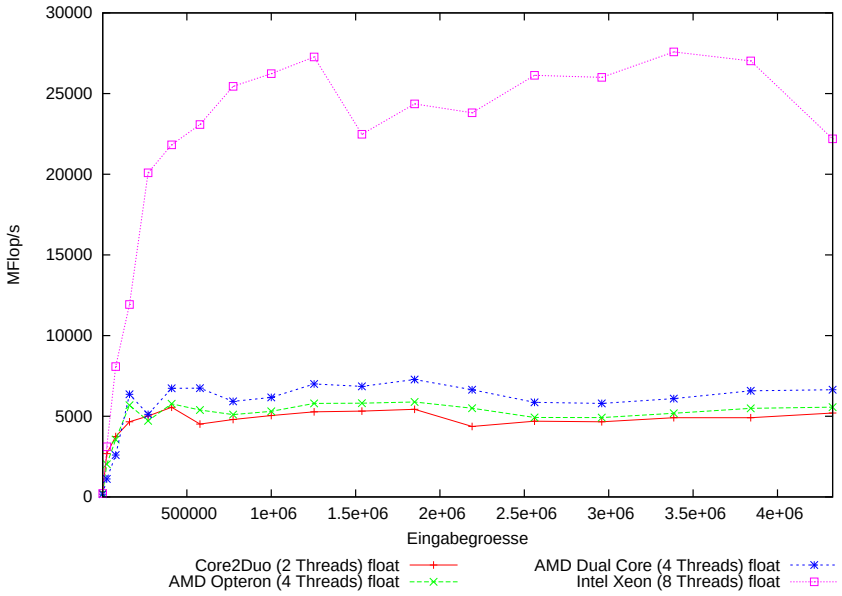


Abbildung 4.15.: Vergleich der x86 Architekturen für das Matrixprodukt.

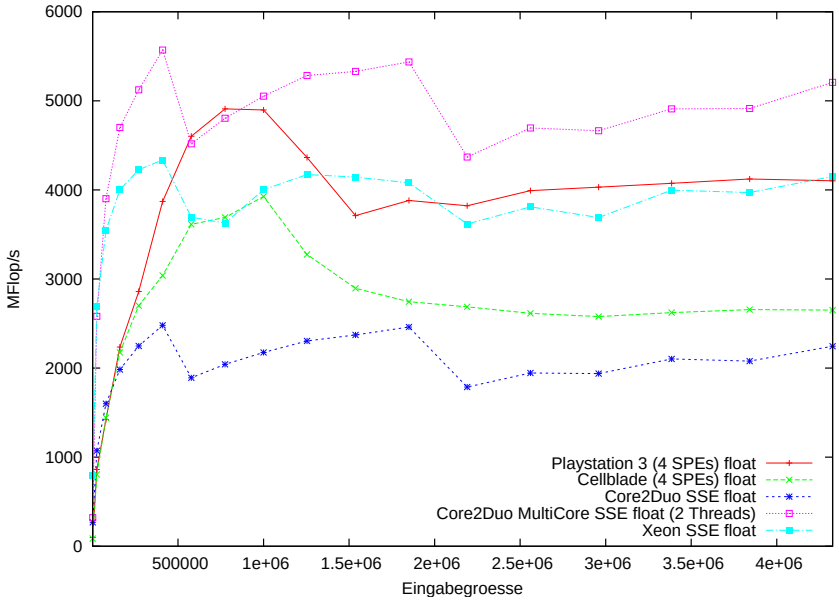


Abbildung 4.16.: Vergleich der Architekturen für das Matrixprodukt.

trix die als B bezeichnete Matrix vollständig durchlaufen zu können. Zum anderen kann beim Matrixprodukt der lokale Speicher der SPEs, im Gegensatz zu den Caches der x86-Prozessoren, in keinster Weise für die Wiederverwendung von Daten genutzt werden. Auf Grund seiner sehr kleinen Ausmaße und der Probleme mit der Speicherbandbreite wird er eher zu einer Art Flaschenhals, da jegliche Daten, auf die zugegriffen werden sollen, in ihn hineingeladen werden müssen, weil kein direkter Zugriff auf den Hauptspeicher möglich ist. Somit ist, auf Grund der beschriebenen Tatsache, dass immer die ganze rechtsseitge Matrix durchlaufen werden muss, keine Wiederverwendung möglich, da der Inhalt des lokalen Speichers ständig mit anderen Teilen der Matrix ersetzt werden muss. An dieser Stelle könnten sich andere Matrizen-*Layouts* möglicherweise bezahlt machen, wenn man beispielsweise eine Matrix gleich blockweise im Speicher ablegt. Auf diese Weise können Algorithmen entworfen werden, die mehr Nutzen aus Wiederverwendung ziehen können.

Vergleiche mit der GotoBLAS

BLAS (Basic Linear Algebra Subprograms) ist eine Programmbibliothek, die Matrix- und Vektoroperationen bereitstellt. Es gibt eine Reihe verschiedener Implementierungen, die für unterschiedliche Hardware optimiert sind. Zum Vergleich mit HONEI wurde die GotoBLAS [Kaza] von Kazushige Goto gewählt. Sie wird als die derzeit schnellste BLAS-Implementierung auf x86-Architekturen bezeichnet[Kazb].

Der Vergleich bestätigt bei den Vektoroperationen zunächst die Beobachtungen, dass bereits ein Kern die Speicherbandbreite vollständig auslastet. Wie in den Abbildungen

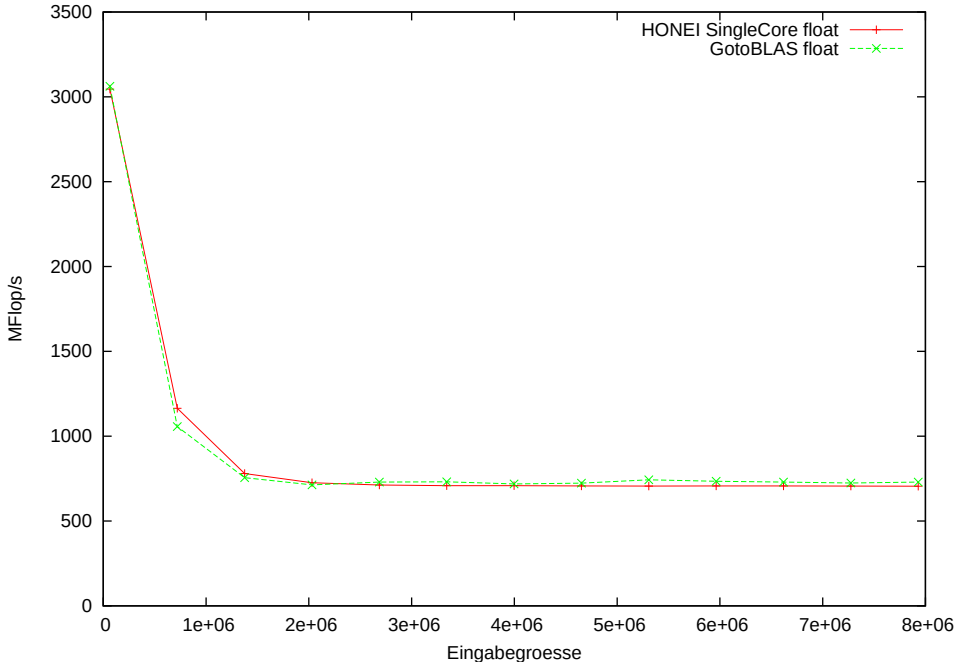


Abbildung 4.17.: GotoBLAS und HONEI ScaledSum auf dem Core2Duo.

4.17 zu sehen ist, bewegen sich die GotoBLAS- und die HONEI-Implementierung auf dem gleichen Niveau.

Für eine Matrix-Verktor Operation (in diesem Fall das DenseMatrix-Vector Produkt) liegen ähnliche Ergebnisse vor siehe Abbildung (4.18), allerdings ist die HONEI-Implementierung hier etwas schneller als die GotoBLAS, die mit $\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$ eine aufwändigere Variante als HONEI ($\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$) berechnet.

Für das Matrixprodukt zeigen die Messergebnisse (4.19) hingegen ein eindeutiges Bild. Die GotoBLAS-Leistung ist deutlich höher. Im Gegensatz zu den Vektor- und Matrix-Vektoroperationen kann hier auch ein deutlicher Leistungszuwachs bei mehreren Threads beobachtet werden.

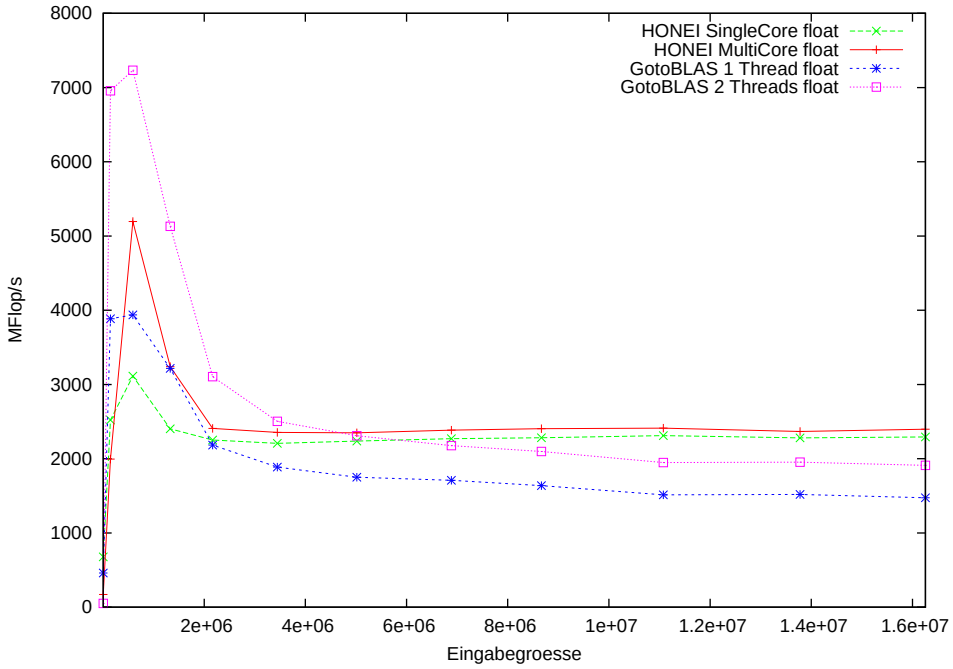


Abbildung 4.18.: GotoBLAS und HONEI Matrix-Vektor Produkt auf dem Core2Duo.

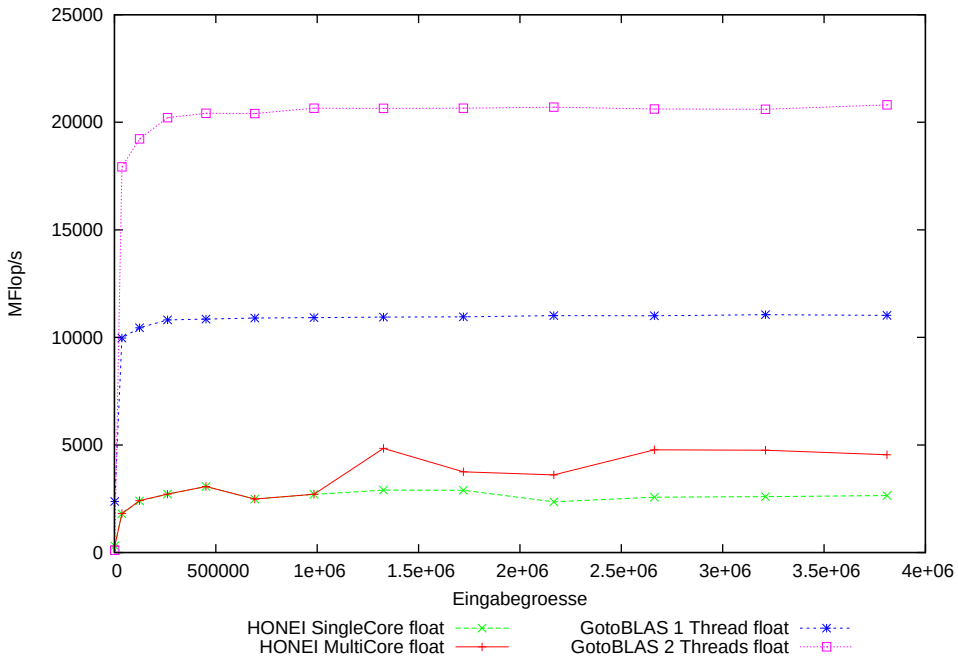


Abbildung 4.19.: GotoBLAS und HONEI Matrixprodukt auf dem Core2Duo.

4.2. Flachwassergleichungen

4.2.1. Numerische Ergebnisse

Um die Korrektheit des `RelaxSolver` und dessen Leistungsfähigkeit zu demonstrieren, werden hier nun einige Beispielszenarien beschrieben und die entsprechenden Ergebnisse präsentiert. Die folgenden Betrachtungen erfolgen stets für den in Kapitel 3.3.1 beschriebenen `RelaxSolver`. Neben dem in einer fest vorgegebenen Genauigkeit rechnenden Löser werden zusätzlich die zwei in Kapitel 3.3.3 beschriebenen gemischt genauen Löserkonfigurationen betrachtet. Als Prototyp für einen gemischt genauen Löser, der innerhalb eines Zeitschrittes in einer festen Genauigkeit rechnet, wird dabei die Implementierung betrachtet, bei der jeder k-te Rechenschritt in doppelter Genauigkeit und ansonsten in einfacher Genauigkeit gerechnet wird. Prototypisch für den Fall, dass *innerhalb* eines Zeitschrittes eine Konversion stattfindet, wird eine Löserkonfiguration evaluiert, die den zweiten Vorhersageschritt in doppelter Genauigkeit berechnet. Für alle folgenden Szenarios wurde ein Höhenfeld mit den Dimensionen 41×41 verwendet. Des weiteren wurden alle Szenarien in einfacher Genauigkeit gerechnet.

Subkritisches Volldammbruchszenario

Das erste Szenario, mit dem die grundlegensten Kapazitäten des `RelaxSolver`s getestet werden sollen, ist ein Volldammbruch mit subkritischen Wasserpegeln. Eine Welle propagiert dabei vom höher gelegenen Reservoir in ein Tal, in welchem der Wasserpegel jedoch ebenfalls wesentlich größer als Null ist. Die Ränder des Rechengebietes können dabei als solide und undurchlässig angesehen werden. Ergebnisse dieser Simulation sind in Abbildung 4.20 dargestellt. Dabei ist auf der linken Seite das Höhenfeld in einer dreidimensionalen Darstellung abgebildet, während die rechte Seite das Höhenfeld in Form einer Höhenkarte zeigt, da gerade die Erhaltung von Symmetrieeigenschaften damit gut beobachtet werden können.

Kritisches Volldammbruchszenario: Dry-states

Eine Herausforderung für einen Flachwasserlöser stellt ein Dammbruchszenario dar, bei welchem der Untergrund, auf den die Welle aus dem Reservoir prallt, trocken ist. Für alle simulierten Szenarios mit trockenem Becken war der `RelaxSolver` stets stabil. Ergebnisse sind in Abbildung 4.21 dargestellt. Dabei ist ein Quellterm vonnöten, d.h. eine Manning Konstante größer als Null zu wählen.

Interferierende Wellenfronten, steady-states

Bei dieser Simulation geht es darum, festzustellen, ob der Löser zu einem stabilen Zustand zurückkehren kann, d.h., ob sich die Wasseroberfläche in natürlicher Weise beruhigt, oder ob das Verfahren eventuell durch Randbehandlung gestört wird. Dabei werden zylinderförmige Wassersäulen über der ansonsten initial ebenen Wasseroberfläche positioniert, siehe Abbildungen 4.22 und 4.23

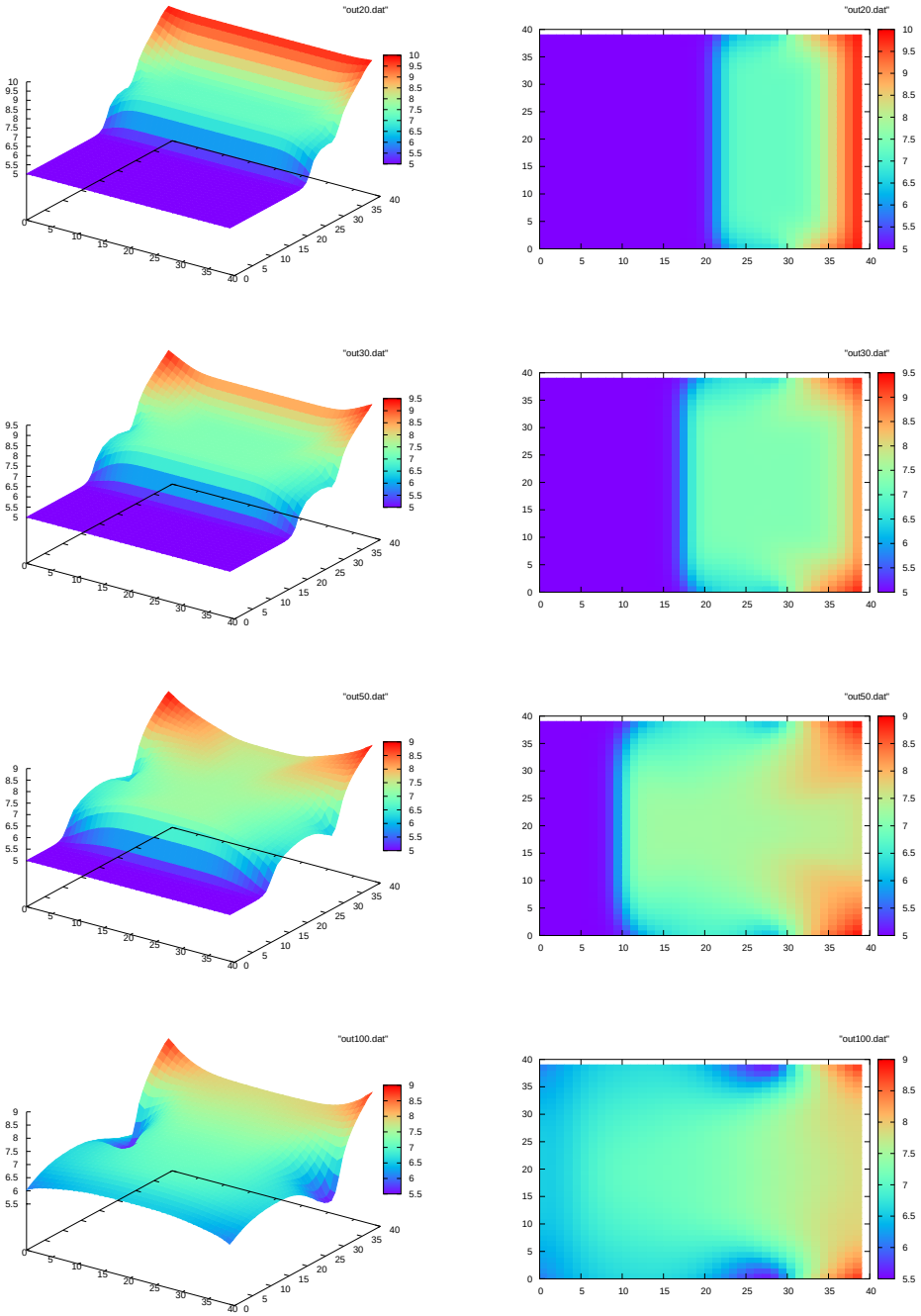


Abbildung 4.20.: Volldammbruch mit subkritischem Wasserpegel für $t_{20} = 4.5s$, $t_{30} = 6.8s$, $t_{50} = 11.36s$, $t_{100} = 22.7s$, $\Delta x = \Delta y = 5m$.

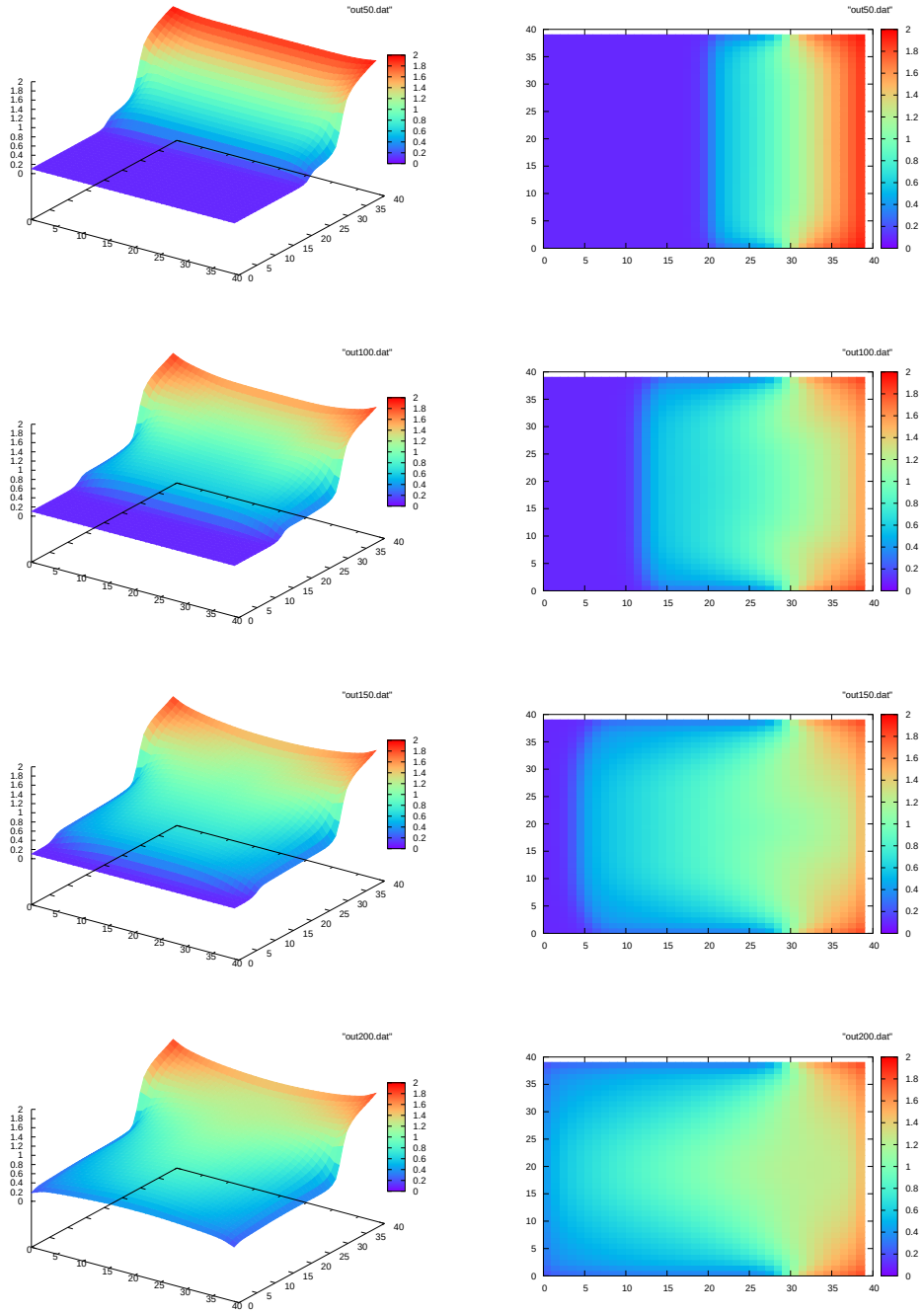


Abbildung 4.21.: Volldammbruch mit kritischem Wasserpegel für $t_{50} = 11.36s$, $t_{100} = 22.7s$, $t_{150} = 34.1s$, $t_{200} = 45.45s$, $\Delta x = \Delta y = 5m$, $n_m = 0.01$.

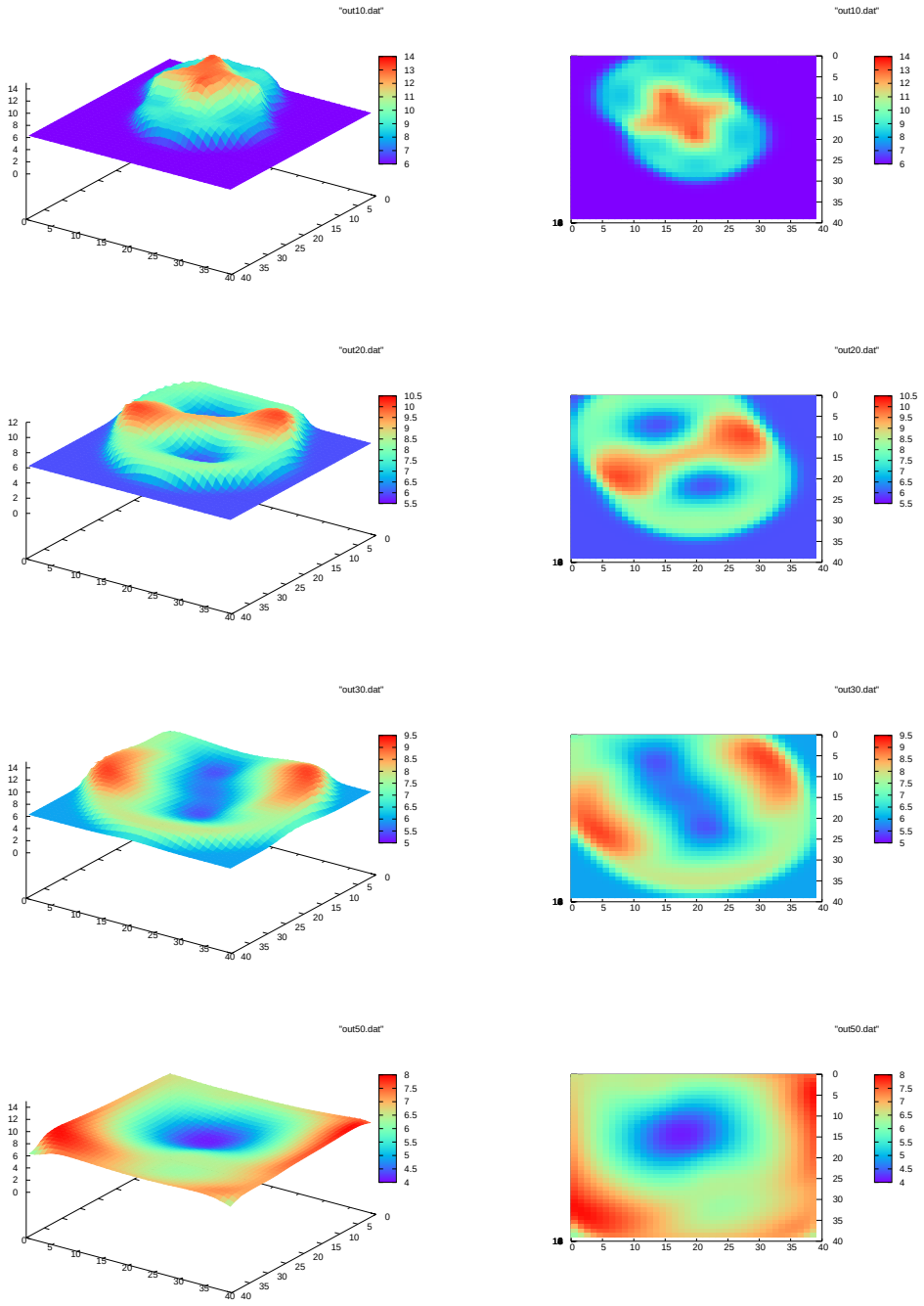


Abbildung 4.22.: Interferierende Wellenfronten: Die radialsymmetrische Ausbreitung beider Wellen wird durch die jeweils andere beeinflusst, aber nicht gestört. $\Delta t = 0.208s, \Delta x = \Delta y = 5m$

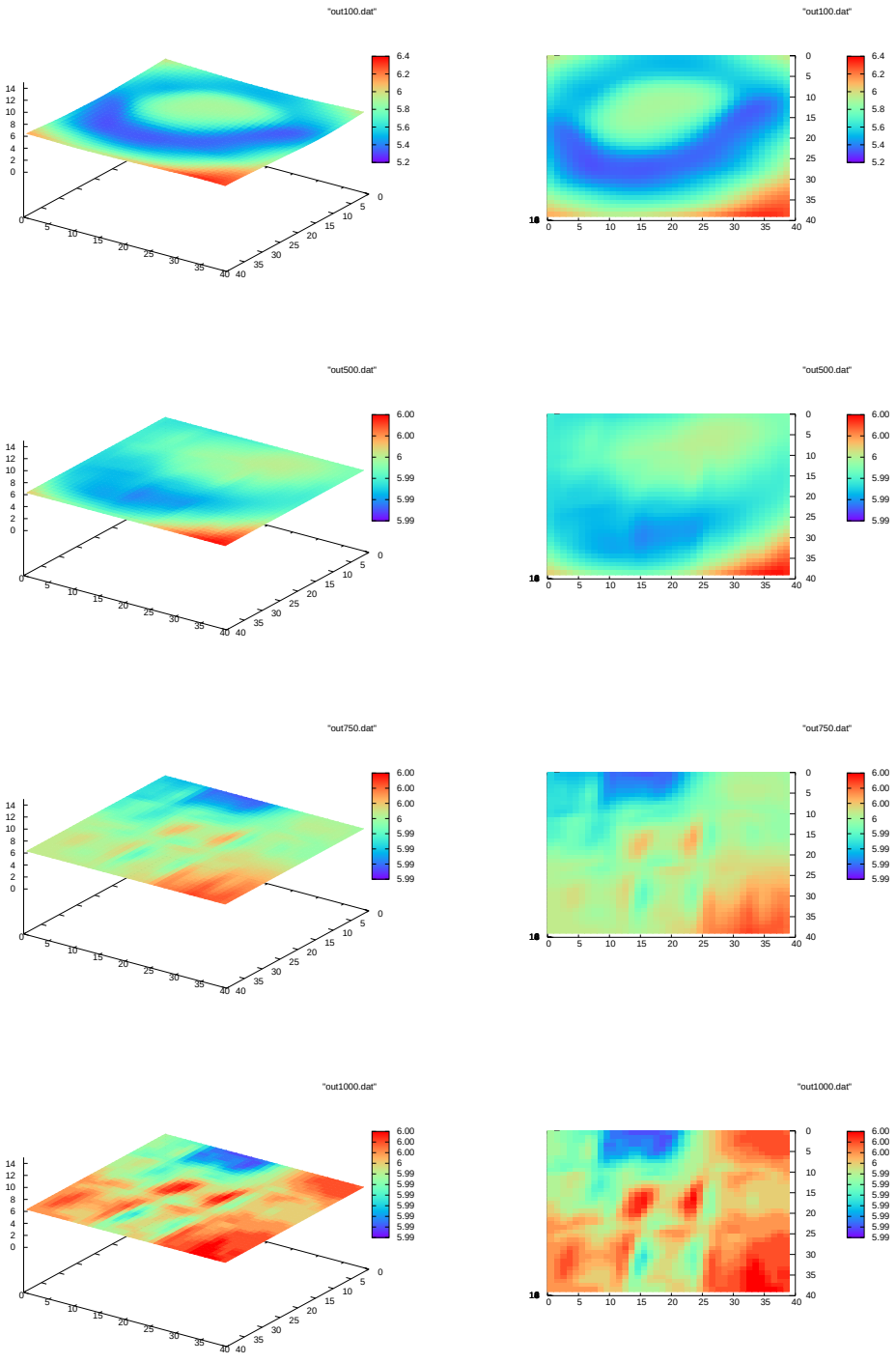


Abbildung 4.23.: Steady-state: Der Wasserstand konvergiert gegen eine konstante Höhe, nachdem die aus den Wassertropfen resultierenden Wellen das Rechengebiet passiert haben. $\Delta t = 0.208s, \Delta x = \Delta y = 5m$

Wassersturz auf unebenes Bodenprofil, stetiger Fluss über eine Bodenwelle

Wesentlich problematischer in numerischer Hinsicht ist die Störung des Wasserflusses durch eine Unebenheit im Bodenprofil. Für solche Simulationen sind im Regelfall komplizierte Quellterme unablässig. Der Löser ist mit seiner einfachen Quelltermimplementierung jedoch sehr wohl in der Lage, mit variablem Bodenprofil umzugehen. Einen ersten Eindruck von dessen Kapazitäten bezüglich eines hügeligen Untergrundes gibt die Simulation, die in Abbildung 4.24 dargestellt wird. Ein solches Szenario wird im Allgemeinen als numerisch problematisch betrachtet, da die Wellenfront zu Beginn der Simulation auf das Hindernis trifft. Ein korrektes Ergebnis beinhaltet die Tatsache, dass die Lösung wegen der erhöhten Position des Reservoirs und dem damit einhergehenden Wasserstrom gegen einen stetigen Fluss konvergiert. In allen durchgeführten Experimenten wurde dieses Kriterium erfüllt. Dennoch ist es schwierig, solche Szenarien zu erzeugen, da die Abschätzung der Relaxationsparameter hinreichend gut sein muss, so dass die Simulation stabil läuft. Zu hoch eingestellte Parameter sorgen für eine zu hohe Viskosität, während zu niedrig eingestellte Relaxationsvektoreinträge das Ergebnis erheblich verfälschen oder sogar den Löser destabilisieren.

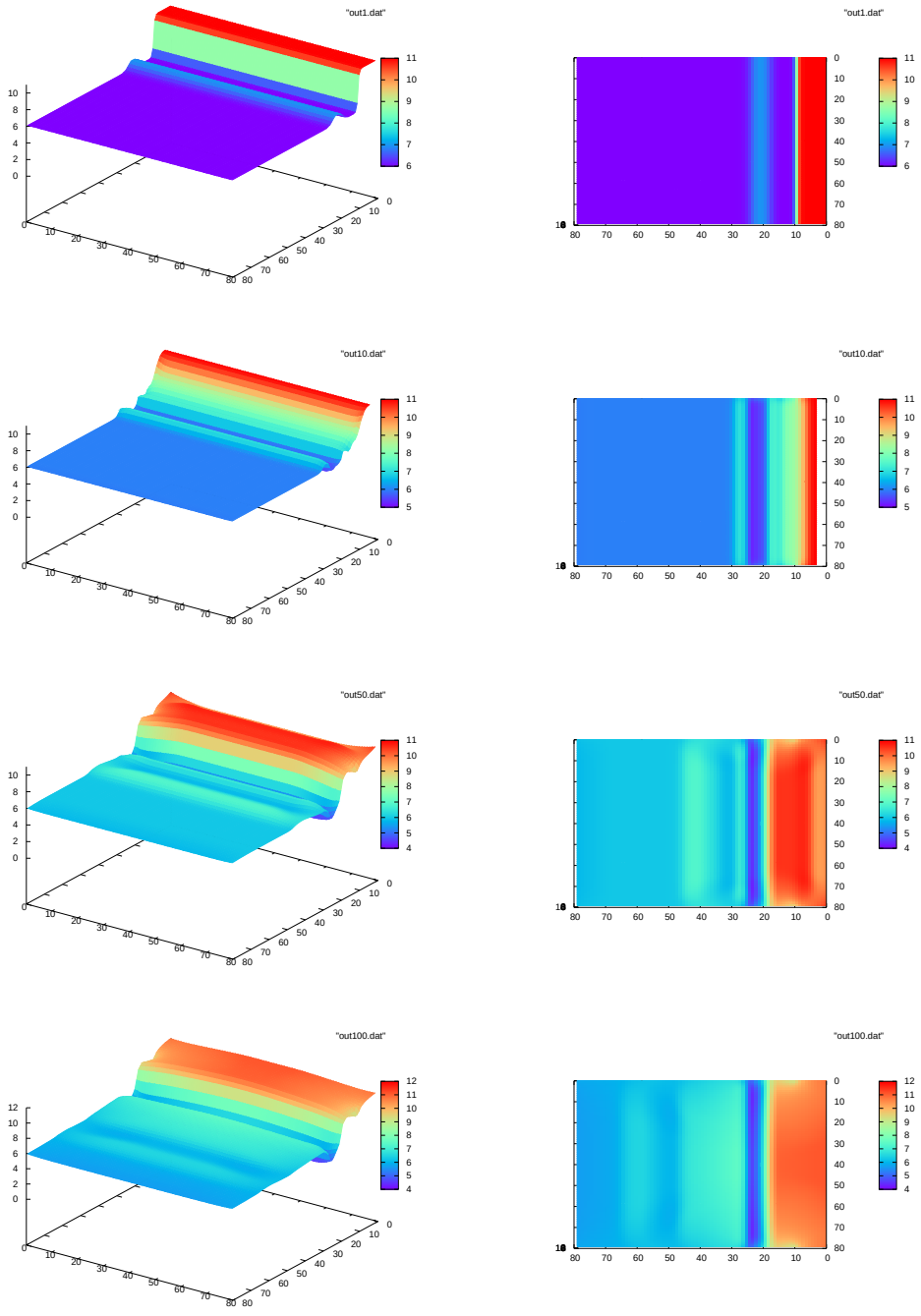


Abbildung 4.24.: Wassersturz auf unebenes Bodenprofil, stetiger Fluss über eine Bodenwelle $t_1 = 0.208s$, $t_{10} = 10.4s$, $t_{50} = 52s$, $t_{100} = 104s$, $\Delta x = \Delta y = 5m$, $n_m = 0.01$.

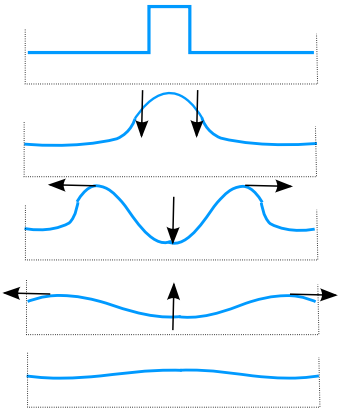


Abbildung 4.25.: Das Prinzip des Masseerhaltungstests: Nachdem das Wasser über die Grenzen des Rechengebietes transmittiert worden ist, kommt es zum *steady state*.

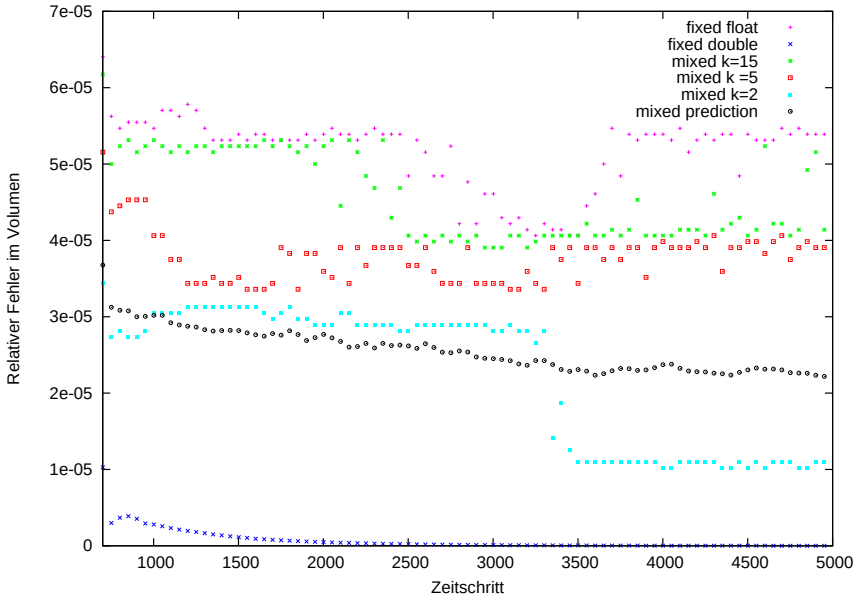


Abbildung 4.26.: Numerischer Volumenfehler im *steady state*.

Masseerhaltung - konstantes Volumen, gemischte Genauigkeit

Neben der visuellen Evaluierung soll hier der Nachweis angebracht werden, dass der vorgestellte Löser für die Flachwassergleichungen masseerhaltend ist. Dabei wird das Volumen im *steady state* nach einer sinnvollen Simulation bei sehr langer Laufzeit und

für die verschiedenen, in Abschnitt 3.3.3 beschriebenen Löserkonfigurationen (in Bezug auf die Berechnungsgenauigkeit) gemessen. Dabei wird prinzipiell so vorgegangen, wie in der in Abbildung 4.23 dargestellten Simulation. Ab einem Zeitpunkt in der Simulation wird das durch die Tropfen hinzugefügte Volumen über die Ränder aus dem Rechengebiet transmittiert. Das Volumen, das im *steady state* herrschen muss, kann so einfach analytisch berechnet werden. Für die numerische Volumenberechnung werden Gaussquadraturen und bilineare Interpolation angewendet. Für Details zur Volumenberechnung sei hier auf die von Becker [Bec06] beschriebene Methode verwiesen. Das Prinzip des Versuchs ist in Abbildung 4.25 visualisiert. Abbildung 4.26 zeigt den Verlauf der Differenz des Volumens unter dem Ergebnishöhenfeld mit dem analytischen Volumen bei Simulation mit der Playstation 3. Alle Löserkonfigurationen zeigen im *steady state* eine Abweichung vom Referenzergebnis von der Größenordnung 10^{-1}, was einem Anteil von 0.001% am Gesamtvolumen gleichkommt. Gleichzeitig erkennt man, dass beide gemischt genauen Löser im wesentlichen dieselbe numerische Genauigkeit im Ergebnis liefern, wie die doppelt genaue Konfiguration. Dennoch sind kleine numerische Unterschiede zwischen den einzelnen Konfigurationen feststellbar. Etwa dieselbe Ergebnisgenauigkeit erreichen die beiden Varianten für $k = 2$, wobei die Variante mit doppelt genauer Vorhersagephase einen wesentlich stabileren Kurvenverlauf aufweist. Zurückzuführen ist dies auf die Tatsache, dass bei dieser Konfiguration lediglich die für die Vorhersagephase nötigen Daten konvertiert werden müssen, wohingegen bei einer vollständigen Iteration in doppelter Genauigkeit eine Konversion aller Szenariodaten erfolgen muss, wobei ein größeres Potential von Informnationsverlust besteht. Des weiteren haben Experimente gezeigt, dass die Wahl des zweiten Vorhersageschrittes dabei durchaus eine Rolle spielt. Für den implementierten Flachwasserlöser ist die Wahl der Berechnung dieses Moduls an genau dieser Stelle optimal. Damit ist eine Iteration des Lösers bei dieser Variante dem in Abschnitt 3.5.1 beschriebenen *Mixed Precision Iterative Refinement* schematisch recht ähnlich, da auch dort die doppelt genauen Rechnungen zu einem geringen Anteil und zu einem späten Zeitpunkt innerhalb einer Iteration stattfinden. Ein Vergleich mit der anderen Variante zeigt, dass diese selbst für das kleinste k keine stabile Verbesserung des Ergebnisses erreichen kann. Wegen der geringen Datenmenge, die konvertiert werden muss ist zudem zu erwarten, dass diese auch die schnellste gemischt genaue Variante ist.

Auf allen Testsystemen konnte in der Größenordnung dieselbe Berechnungsgenauigkeit für diesen Test nachgewiesen werden. Lediglich bei Berechnung durch die SPEs der Cell BE konnten leichte Unterschiede festgestellt werden, was auf den nicht IEEE-standardkonformen Rundungsmodus zurückzuführen ist. Es konnte jedoch bei den Flachwassersimulationen kein Szenario erstellt werden, bei dem dies praktisch von Bedeutung gewesen wäre.

4.2.2. Leistungsevaluierung

Skalierbarkeit

Die Abbildungen 4.27, 4.28 und 4.29 zeigen die Ausführungszeiten der in fester Genauigkeit rechnenden Löserkonfigurationen mit den verschiedenen Backends und zeigen des

weiteren den wegen fehlender Parallelisierung auf Applikationsebene erwarteten parabelförmigen Verlauf der Rechenzeiten bei wachsender Problemgröße. Für die MultiCore- und die Cell-Implementierung sind dabei Kurven für verschiedene Anzahlen von Threads bzw. SPEs gegeben, respektive für 1, 2 und 4 SPEs bzw. 1, 2, 4 und 8 Threads dargestellt. Für die Simulationen auf der Playstation 3 gilt, dass sich die für Vektoroperationen beobachteten Skalierbarkeitseigenschaften für relevante Problemgrößen nicht übertragen lassen. Für die echtzeitnahen Simulationen sind die Eingabegrößen noch zu klein, um dieselben Leistungssteigerungen zu sehen, wie auf Operationsebene. Des weiteren sind Skalierbarkeitseffekte ohnehin durch die Tatsache gedämpft, dass nur ein gewisser Anteil des Lösercodes für die SPEs optimiert wurde. Dennoch kann man zeigen, dass sich die Zuschaltung weiterer SPEs lohnt. Bei einer Eingabegröße von etwa 350x350 (das entspricht einer Operandengröße für die Operationen von etwa 140000) sieht man dabei eine klare Leistungssteigerung. Dies passt zu den in Abbildung 4.4 in Abschnitt 4.1.2 gezeigten Skalierbarkeitstests für das Bandmatrix-Vektorprodukt, von welchem die Hauptberechnungslast des Lösers getragen wird. Für die Berechnung von größeren Problemen fehlt es der Playstation 3 jedoch an Arbeitsspeicher, bzw. die Benchmark-Ergebnisse werden durch Auslagerung von Daten auf die Festplatte verfälscht, wodurch der Hauptnachteil dieses Systems zu Tage tritt.

Für das MultiCore-Backend gilt, dass eine Verwendung von mehr als einem Thread bestenfalls für gleiche Ausführungszeiten sorgt, wie die Verwendung von einem Thread (vergleiche Abbildung 4.28). Auf dem Xeon-Testsystem wird schließlich der für Vektoroperationen beobachtete administrative Mehraufwand deutlich, der durch das MultiCore-Backend verursacht wird. Die sprunghafte Leistungssteigerung beim Übergang von einem auf zwei Threads ist dabei durch die Verwendung des zweiten Prozessors zu erklären. Obwohl der Löser nicht mit der Anzahl der CPUs skalieren dürfte, da das System nur über einen Speichercontroller verfügt, bedeutet die Hinzunahme der zweiten CPU bei den 5000p Chipsätzen, die über Intels DIB (*Dual Independent Bus*) Technologie verfügen, eine Weitung der Kapazität des Speichercontrollers (und damit eine Verringerung des Flaschenhalses). Generell kann die MultiCore-Implementierung bei Verwendung des Xeon-Systems jedoch keine wesentliche Verkürzung der Rechenzeit gegenüber dem Core2Duo-System bewirken.

Gemischt genaue Methoden

Für beide in Abschnitt 4.2.1 numerisch evaluierten gemischt genauen Löserkonfigurationen gilt, dass die Ausführungszeiten kleiner sind, als die der in hoher Genauigkeit rechnenden Variante. Dabei ist die numerisch genaueste Konfiguration, respektive die, welche innerhalb eines Zeitschrittes die zweite Vorhersagephase in hoher Genauigkeit rechnet, gleichzeitig auch die performanteste. Die Rechenzeiten für die verschiedenen Varianten auf der Playstation 3 sind in Abbildung 4.30 dargestellt, wobei die Kurven mit den Verläufen des Volumenfehlers aus Abbildung 4.26 korrespondieren.

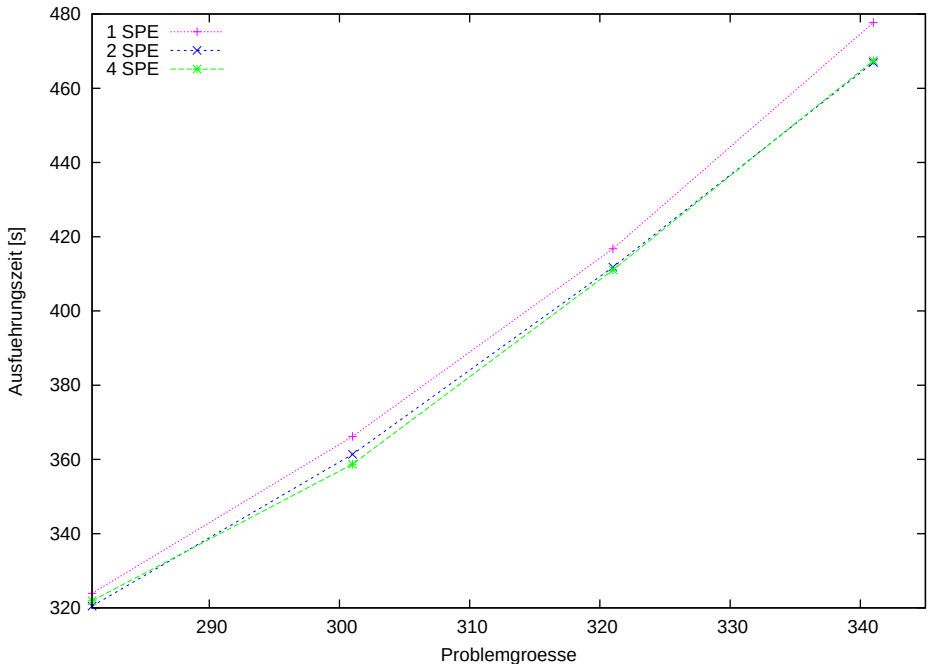


Abbildung 4.27.: Ausführungszeiten des Lösers mit 1, 2 und 4 SPEs auf der Playstation 3, einfache Genauigkeit.

Echtzeitsimulation

Eine wesentliche Frage bei der Evaluierung des Flachwasserlösers ist die nach der Fähigkeit, eine Simulation in Echtzeit durchzuführen. Dazu soll hier zunächst der Echtzeitbegriff geklärt werden, da er auf zwei verschiedene Arten verstanden werden kann. Die erste Sichtweise ist, dass die Simulationszeit im Zeitschritt t_k, T_k^{sim}, nicht kleiner sein darf, als die in Wirklichkeit verstrichene Zeit T^{real} d.h. $T_k^{sim} = t_k \Delta t \leq T^{real}$. Da diese Bedingung zwar für technische Anwendungen interessant ist, aber nicht für ein visuell ansprechendes Ergebnis ausreicht (man könnte ihr mit einem Zeitschritt genügen), benötigt man eine zweite Sichtweise. Dabei geht man von der Bildwiederholrate aus, die das menschliche Auge gerade als nicht unterbrochen wahrnimmt. Danach ist die Erzeugung von mindestens 24 Bildern pro Sekunde vonnöten, um das Auge zufrieden zu stellen, was eine Zeitschrittgröße von etwa $\Delta t = 0.0417$ vorraussetzt. Die Abbildung 4.31 visualisiert die Grenze bei der Eingabegröße, so dass gerade noch ein Echtzeitergebnis möglich ist für die SSE-, Cell- und MultiCore-Backends. Diese soll den derzeitigen Entwicklungsstand des Projektes im Hinblick auf die Leistungsfähigkeit der Flachwasseranwendung zeigen. Der Vergleich der Cell-Implementierung mit den x86-Backends ist insofern unzulässig, als dass die Entwicklung der beiden verschiedenen Programmiermodellen folgt, worauf im folgenden Abschnitt näher eingegangen wird. Mit der Cell-Implementierung kann die Portierung der Anwendung auf die Cell BE als gelungen angesehen werden,

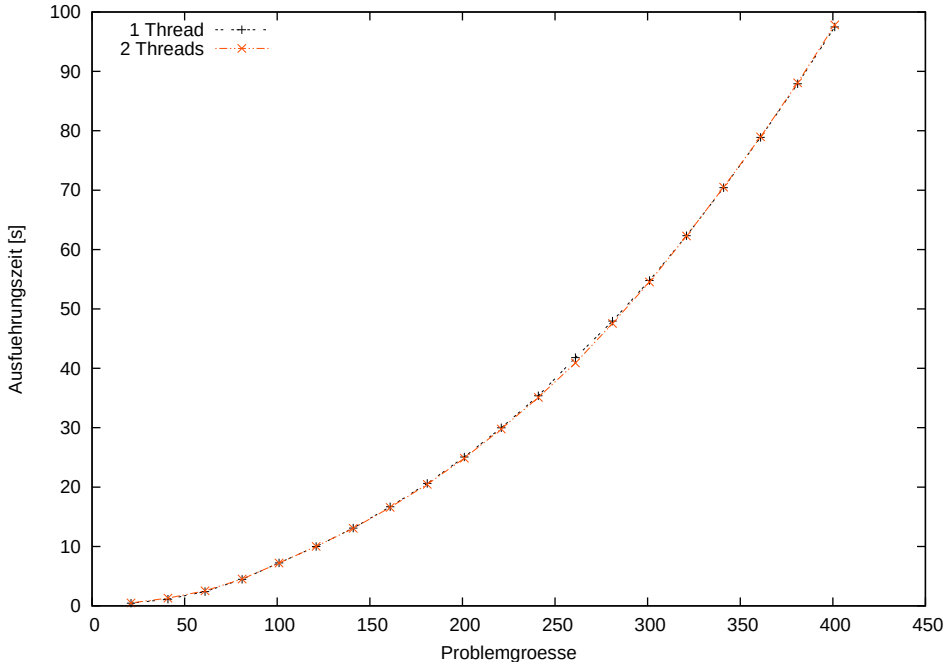


Abbildung 4.28.: Ausführungszeiten des Lösers mit 1 und 2 Threads auf dem Core2Duo-System, einfache Genauigkeit.

wobei das Optimierungspotential nicht voll ausgeschöpft ist, da einerseits nur der Teil der Applikation optimiert worden ist, der auf Basiskomponenten beruht und des weiteren das Cell-Backend nicht voll ausgereift ist, wie bereits in Abschnitt 4.1 erläutert. Der derzeitige Spitzenwert für die Eingabegröße, so dass die Simulation gerade noch in Echtzeit läuft wird von dem Core2Duo-System mit dem SSE-Backend erreicht und liegt bei 90×90. Dies entspricht auf Operationsebene in etwa einer Operandengröße von 25000. Dabei muss beachtet werden, dass ein Zeitschritt des Lösers den Aufruf von 16 Matrix-Vektor Produkten und fast 20 Vektor-Vektor-Operationen, wenn man nur die Basisoperationen betrachtet, beinhaltet. Wie im nächsten Abschnitt dargelegt, stellen diese nur einen Teil der Gesamtrechenzeit dar. Für die Playstation 3 liegt der ermittelte Spitzenwert in etwa bei30×30. Die Werte beziehen sich dabei auf einfache Genauigkeit, wobei man für doppelte Genauigkeit einen Faktor von etwa 0.5 wie erwartet anlegen kann. Durch gemischt genaue Implementierungen kann dieser jedoch wesentlich größer sein, wie Abbildung 4.30 deutlich macht.

Anteile der Basisoperationen an der Rechenzeit

Wie bereits im Implementierungskapitel zu `RelaxSolver` angemerkt wurde, stellte sich schnell heraus, dass der Anteil der Rechenzeit, der von der Ausführung von Basisoperationen eingenommen wird kleiner war, als erwartet. Als Konsequenz davon wurde der

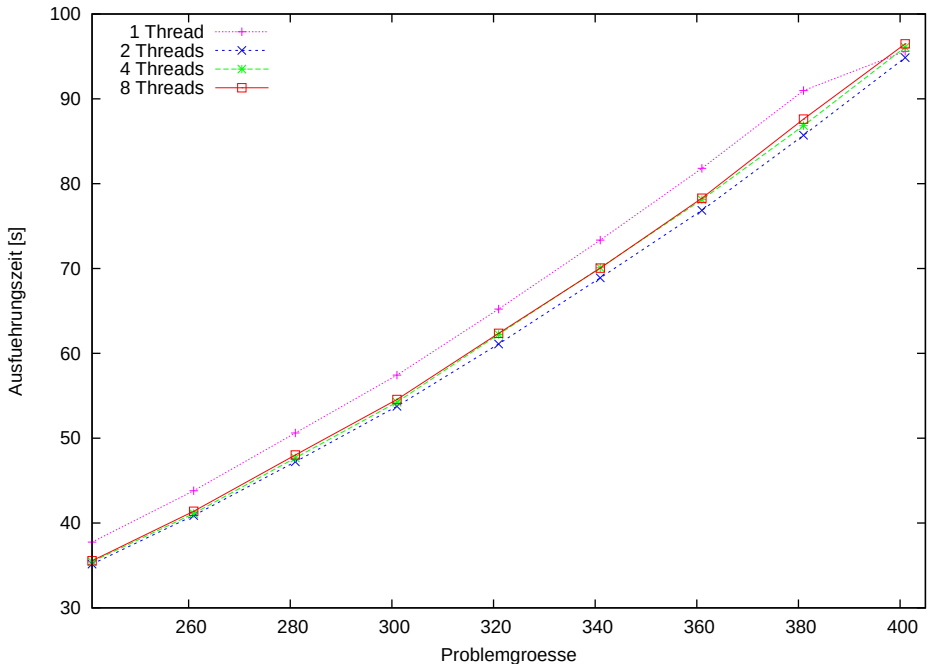


Abbildung 4.29.: Ausführungszeiten des Lösers mit 1, 2, 4 und 8 Threads auf dem Xeon-System, einfache Genauigkeit.

Löser reimplementiert, indem die für die Anwendung spezifischen Rechnungen als eigene Operationen gekapselt und später optimiert wurden. Zunächst sei anzumerken, dass dieses Vorgehen dem in der PG zugrunde gelegten Programmierparadigma einer allgemeingültigen Bibliothek von Operationen widerspricht. Die im folgenden dargelegten Ergebnisse zeigen aber die Schwächen einer solchen Herangehensweise auf. Idealerweise sollte eine Bibliothek von Basiskomponenten zwei Kriterien erfüllen. Zum einen sollte die Bibliothek möglichst alle laufzeitkritischen Komponenten der anvisierten Applikationen enthalten und zum anderen sollte sie einen Grad von Wiederverwendbarkeit haben, der wesentlich über die Beispielanwendungen hinausgeht. Dabei wird implizit davon ausgegangen, dass der Compiler den nicht in Operationen gekapselten Code hinreichend gut optimiert. Die Tatsache, dass die Entwicklung von Hardware der Entwicklung von Compilern im Allgemeinen wesentlich voraus ist, ist nicht nur ein bekanntes Problem, sondern ergibt sich ganz natürlich. Versuche dahingehend, die Optimierung durch den Compiler mit handoptimiertem Code zu schlagen, wurden ebenfalls von Teilnehmern der PG durchgeführt und ergaben, dass gerade im Hinblick auf SIMD-Code auch mit einfachen Mitteln Leistungsgewinne gegenüber compileroptimierten Operationen erzielt werden können. Des weiteren ist der Anteil der durch algebraische Operationen abgedeckten Applikationsmodule nicht so groß, wie es ein solches Programmierparadigma erfordern würde. Dies soll im folgenden näher erläutert werden.

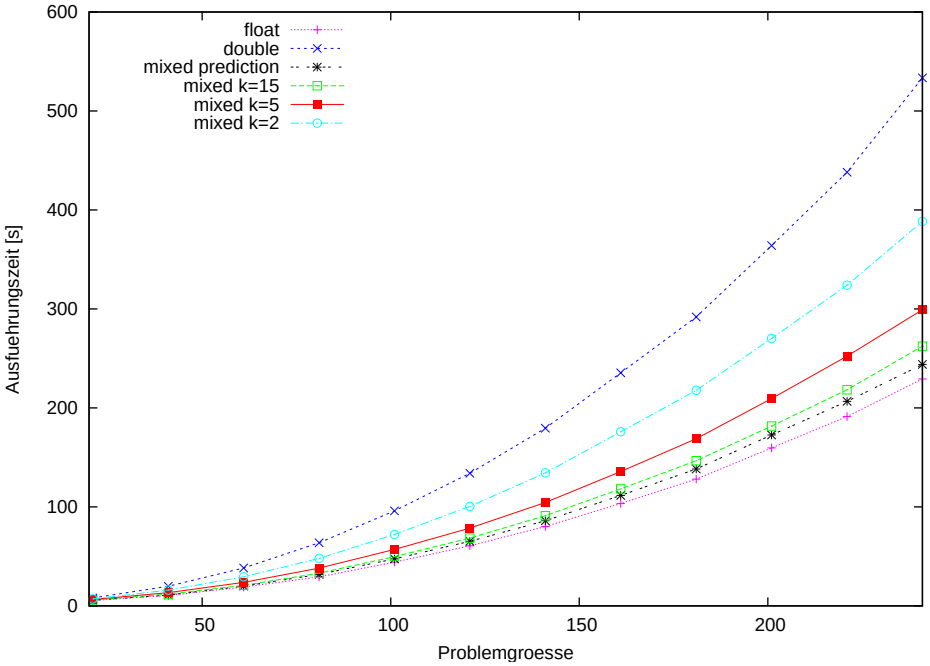


Abbildung 4.30.: Gesamtlaufzeit der Löservarianten auf der Playstation 3 für 100 Zeitschritte.

Durch die SSE Implementierung des monolithischen Lösers einerseits und die weiter optimierte SSE Variante, in der zusätzlich noch die eigenen Operationen des Lösers optimiert wurden, ist eine direkte Analyse der oben genannten Anteile an der Rechenzeit möglich. In Abbildung 4.32 werden die Rechenzeitanteile der verschiedenen Komponenten an der Gesamtrechenzeit des Lösers nach Optimierung der Module, die speziell für diese Anwendung implementiert wurden, dargestellt. Dabei ist ein Anteil von etwa einem Drittel der Ausführungszeit nicht den Basiskomponenten zuzuordnen, wobei der größte Beitrag dabei erwartungsgemäß den Matrix- und Vektorassemblierungen zufällt, bei denen die Anzahl der Speichertransfers sehr groß ist. Bei der monolithischen Implementierung ist ein Anteil von mehr als 60% gemessen worden. Die Optimierung der Module ergibt dann auch eine entsprechende Leistungssteigerung, wie in Abbildung 4.33 zu sehen ist. Man beachte, dass die implementierten Module wenig bis keinen Wiederverwendungswert haben.

Deutlich wird dabei, wie stark der Rechenzeitanteil von Nicht-Basisoperationen an der Rechenzeit sinkt, wenn man die modulare Variante betrachtet. Gleichzeitig ist damit der Aufwand abschätzbar, der entsteht, wenn man eine Applikation *nach* Implementierung mit der Basisbibibliothek weiter optimieren will. Mit diesen Erkenntnissen spätestens steht die erste Herangehensweise der PG zur Disposition. Das anfänglich angestrebte Programmiermodell kann den Widerspruch zwischen Allgemeingültigkeit und Perfor-

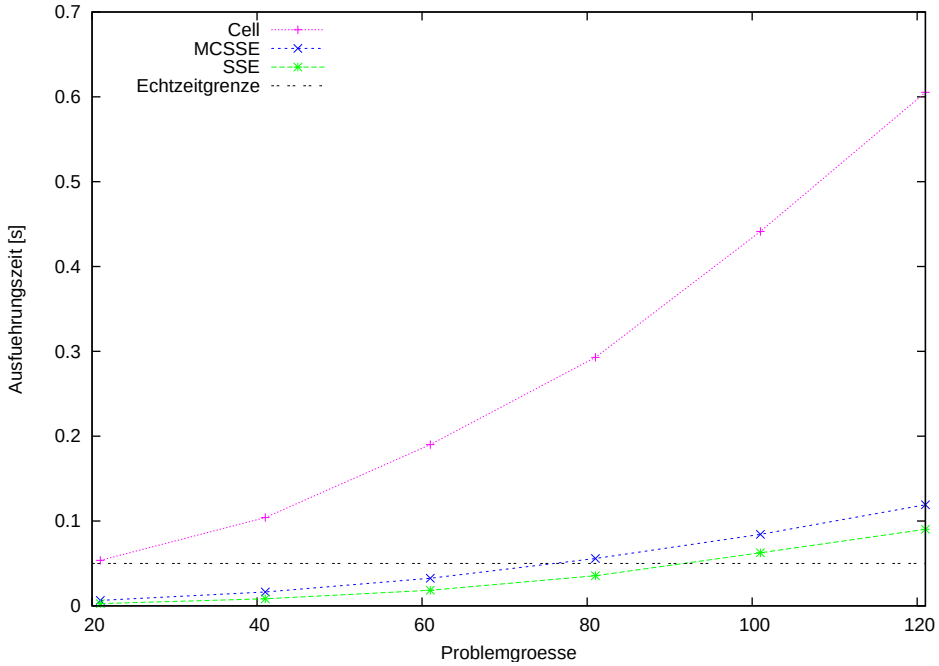


Abbildung 4.31.: Echtzeitgrenze für einfache Genauigkeit über die verschiedenen Backends: Intel Core2Duo Single- und MultiCore mit 2 Threads und Playstation 3 mit 4 SPEs.

manz nicht auflösen, vielmehr ist eine applikationsspezifische Optimierung für diese (und, wie man später sehen wird, auch für eine andere) optimal, wie in Abbildung 4.34 dargestellt. Dabei sieht man gleichzeitig die verschiedenen Stadien des Programmiermodells, das der Entwicklung des Lösers zugrunde lag, von links nach rechts.

Vergleich der Leistung auf den Testsystemen

Die Ausführungszeiten des Lösers auf der Playstation 3 sind nur direkt mit denen der x86 Systeme vergleichbar, wenn man die monolithische Löserimplementierung betrachtet, da das derzeitige Cell-Backend keine spezialisierten Module für die Operationen enthält, wie sie für das SSE-Backend zur Verfügung gestellt wurden. Vergleicht man die Leistung der SSE-Spezialisierung der monolithischen Variante mit der Cell-Implementierung, so erkennt man die Überlegenheit der Playstation 3 in einfacher Genauigkeit (vergleiche Abbildung 4.35). In doppelter Genauigkeit ist das Vergleichssystem erwartungsgemäß schneller. Man kann über die normalisierte Rechenzeit somit eine Leistungssteigerung von einem Faktor von 2 bei der Flachwasseranwendung und beim Vergleich von Playstation 3 und Core2Duo für einfache Genauigkeit prognostizieren, wenn das Cell-Backend ebenso anwendungsspezifisch optimiert wird, wie das SSE-Backend.

Zuletzt soll an dieser Stelle ein Vergleich der beiden zur Verfügung stehenden Test-

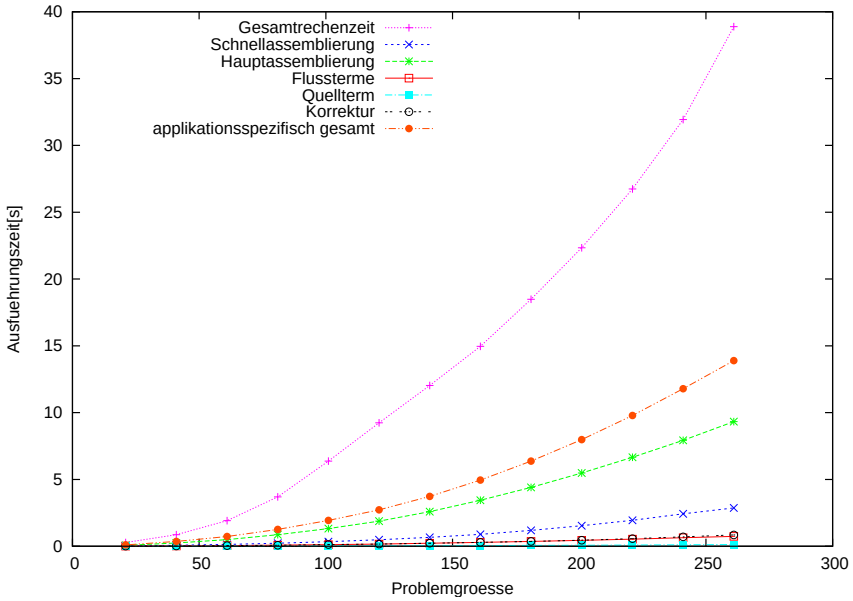


Abbildung 4.32.: Anteile der applikationsspezifischen Operationen an der Gesamtrechenzeit nach Optimierung für SSE auf dem Core2Duo-System.

systeme mit Cell Prozessor(en) angestellt werden. Die bereits bei den Basisoperationen festgestellten Abweichungen vom erwarteten Verhalten der Leistung des Cellblade können auch auf Applikationsebene verifiziert werden, wie Abbildung 4.36 zeigt.

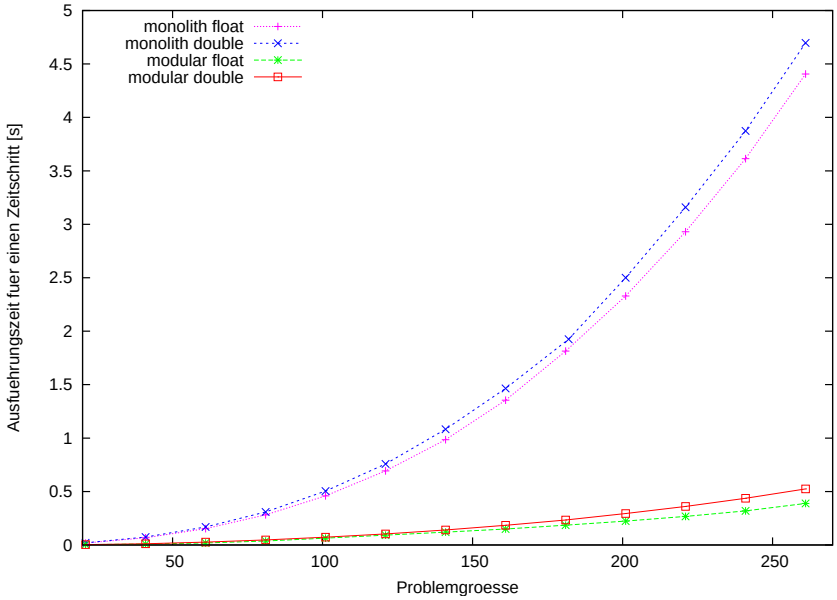


Abbildung 4.33.: Monolithische gegen modulare Implementierung des Flachwasserlösers: Ausführungszeiten für einen Zeitschritt mit SSE auf dem Core2Duo-System.

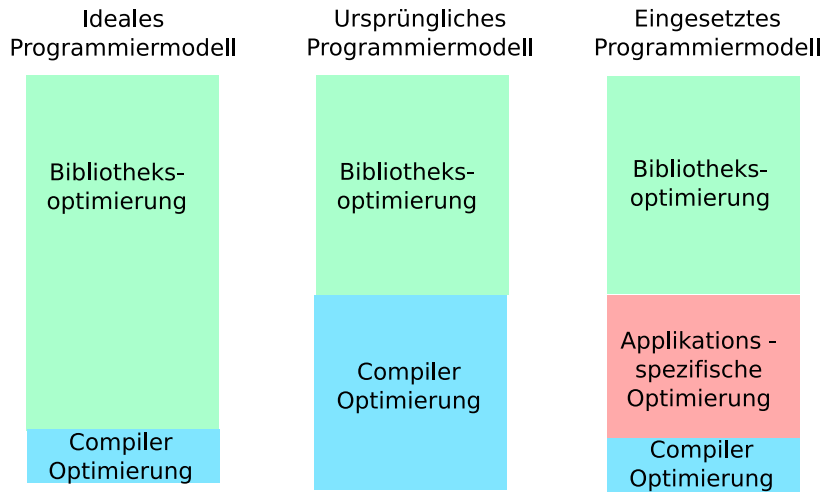


Abbildung 4.34.: Verteilung des Optimierungspotentials bei der Flachwasseranwendung.

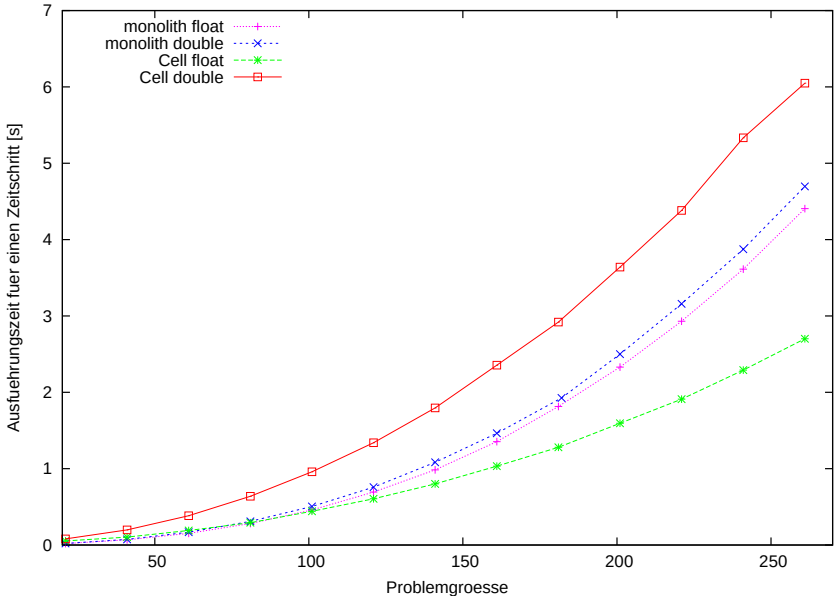


Abbildung 4.35.: Vergleich der Ausführungszeiten der nicht-modularen Implementierung des Flachwasserlösers mit SSE auf dem Core2Duo mit der aktuellen Cell-Implementierung

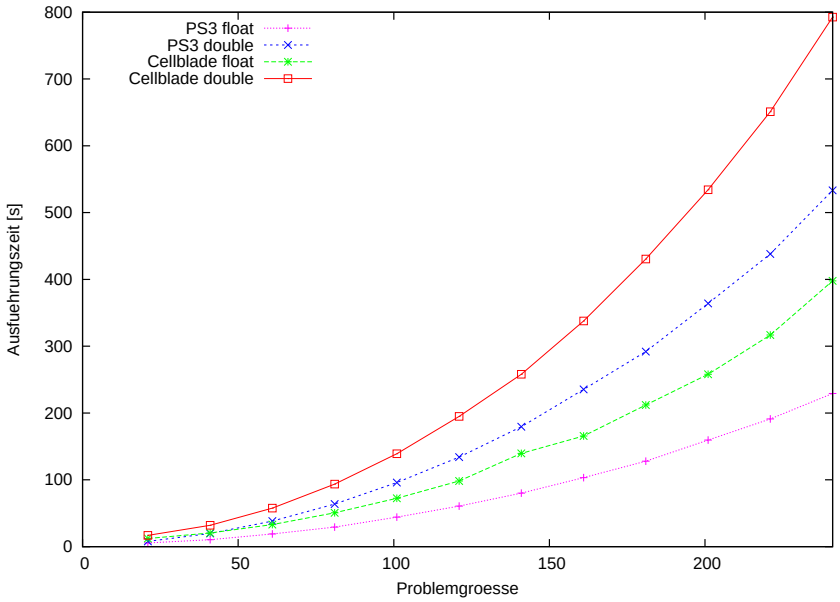


Abbildung 4.36.: Vergleich der Cell-Hardware.

4.3. Layout-Verfahren für Graphen

4.3.1. Numerische Ergebnisse

Im folgenden wird die Qualität und die Effizienz der in Kapitel 3.4 vorgestellten Implementierung der Layoutverfahren für Graphen betrachtet. Für die in Abschnitt 3.4.1 vorgestellte Verschiebungsmethode von Frick soll demonstriert werden, wie viele Iterationen nötig sind, um ein optisch ansprechendes Layout zu erzeugen. Die folgenden Beispiele sollen die Ergebnisse demonstrieren.

Regelmäßiges rechteckiges Gitter

Dieses Beispiel zeigt, ob das Verfahren überhaupt vernünftige Ergebnisse liefert. Hierzu wird ein Graph mit $m \times n$ Knoten erzeugt, die mit $2nm - n - m$ Kanten verbunden sind. Die Knoten werden so verbunden, dass sie ein rechteckiges Gitter ergeben. Damit dieses Gitter gleichmäßig ist, haben alle Knoten das Gewicht w_n und alle Kanten das Gewicht w_e. Abbildung 4.37 zeigt die Ergebnisse für ein Gitter der Größe 16×16 nach unterschiedlicher Anzahl von Iterationen mit dem Verfahren von Fruchterman-Reingold. Die Knoten wurden zunächst zufällig in einem bestimmten Bereich angeordnet. Danach wird der Algorithmus auf den Graph angewendet. Die Struktur des Graphen ist bereits nach weniger als $2nm$ Iterationen im Layout gut erkennbar. Nach etwa $5nm$ Iterationen sind praktisch keine Unregelmäßigkeiten zu beobachten. Weitere Iterationen beeinflussen die Positionierung der Knoten nur noch minimal, da die verbliebenen Kräfte zwischen den Knoten sehr klein sind.

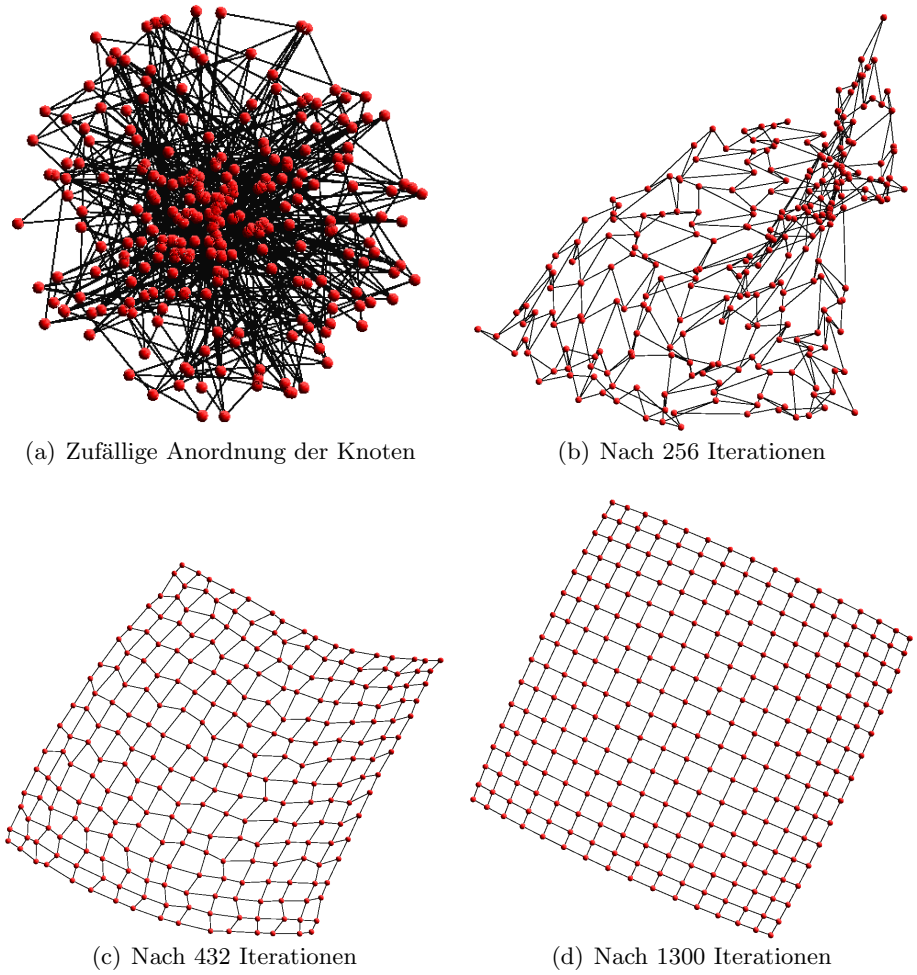
Etwas anders sieht das Layout bei Anwendung von Kamada-Kawai aus. Da hier nur jeweils ein Knoten pro Iteration bewegt wird, werden mehr Iterationen benötigt. Die Implementierung von Kamada-Kawai liefert auch visuell ein etwas anderes Ergebnis, wie Abbildung 4.38 zeigt. Die Ecken des Gitters werden nach Außen gezogen, wodurch die erwartete rechteckige Regelmäßigkeit nicht erreicht wird.

Binärer Baum

Für dieses Beispiel wird ein vollständiger binärer Baum der Tiefe t mit $2^{t+1} - 1$ Knoten erzeugt. Man ist gewohnt, Bäume mit der Wurzel oben und den Kindern auf ihren jeweiligen Ebenen darunter zu zeichnen um die Semantik der Hierarchie zu unterstreichen. Ein kräftebasiertes Layout-Verfahren ohne zusätzliche Randbedinungen wird jedoch die Wurzel in der Mitte des Graphen platzieren und die direkten Nachfolger möglichst weit voneinander entfernen, um so die Kräfte zu optimieren. Abbildung 4.39 stellt das Ergebnis von Fruchterman-Reingold für einen vollständigen binären Baum der Tiefe 5, also mit 63 Knoten dar. Knoten mit derselben Tiefe im Baum liegen in konzentrischen Gebieten: Je tiefer ein Knoten im Baum angeordnet ist, um so weiter außen wird er gezeichnet. Dieses Beispiel zeigt auch den Einfluss von unterschiedlichen Knotengewichten auf das Layout. Das Gewicht eines Knotens ist hier umso größer, je geringer seine Tiefe im Baum ist. In Abbildung 4.39 wird das Knotengewicht durch die Größe des Knotens dargestellt. Knoten mit großem Gewicht werden weiter von einander entfernt.

Evolving Graphs

Abbildung 4.40 zeigt exemplarisch das Layout von *Evolving Graphs*. Es besteht aus fünf *Timeslices* mit wachsender Knotenzahl, die teilweise miteinander verbunden sind. Die einzelnen *Timeslices* sind zur besseren Übersicht unterschiedlich gefärbt und in Ebenen angeordnet. Die Berechnung der Kräfte erfolgt hingegen in einer Ebene. Zwischen den Ebenen verlaufen die *Intertimeslice*-Kanten korrespondierender Knoten benachbarter Ebenen. Die Gewichtung dieser Kanten ist global vorgegeben und beeinflusst, ob das Layout dazu tendiert, korrespondierende Knoten unterschiedlicher *Timeslices* eng zusammen zu halten oder die Knoten der einzelnen *Timeslices* unabhängig von einander zu optimieren.



(a) Zufällige Anordnung der Knoten

(b) Nach 256 Iterationen

(c) Nach 432 Iterationen

(d) Nach 1300 Iterationen

Abbildung 4.37.: 16 × 16-Gitter: Layout mit Fruchterman-Reingold.

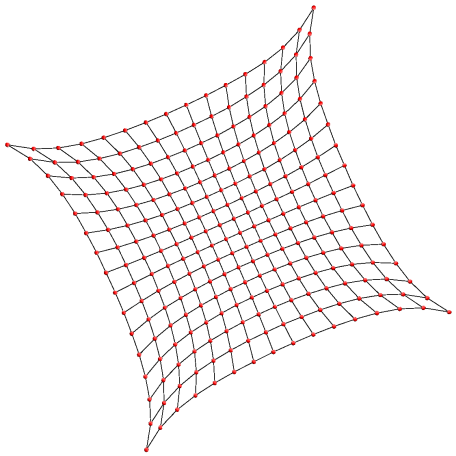


Abbildung 4.38.: 16×16-Gitter: Layout mit Kamada-Kawai nach 6000 Iterationen.

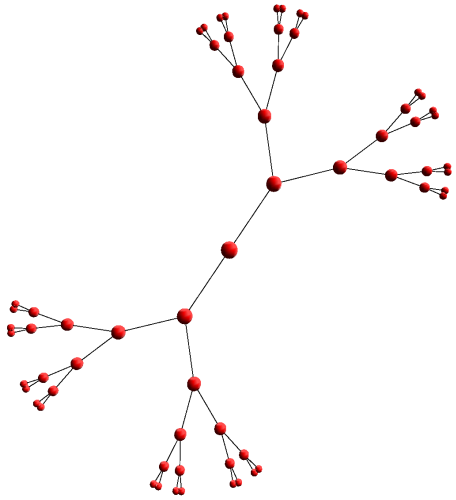


Abbildung 4.39.: Vollständiger binärer Baum mit 5 Ebenen nach 1532 Iterationen Fruchterman-Reingold.

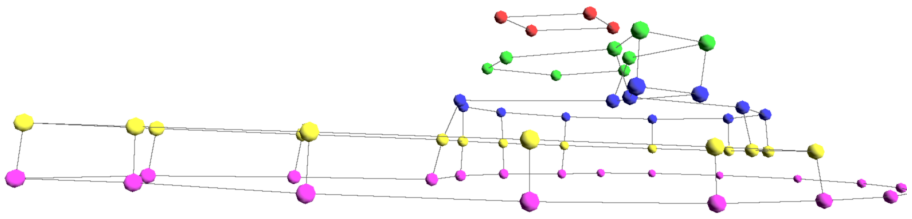


Abbildung 4.40.: *Evolving Graph* mit 5 *Timeslices* und einem *Intertimeslice*-Kantengewicht von 7. Die *Timeslices* sind in der Visualisierung in Ebenen übereinander gelegt und gefärbt.

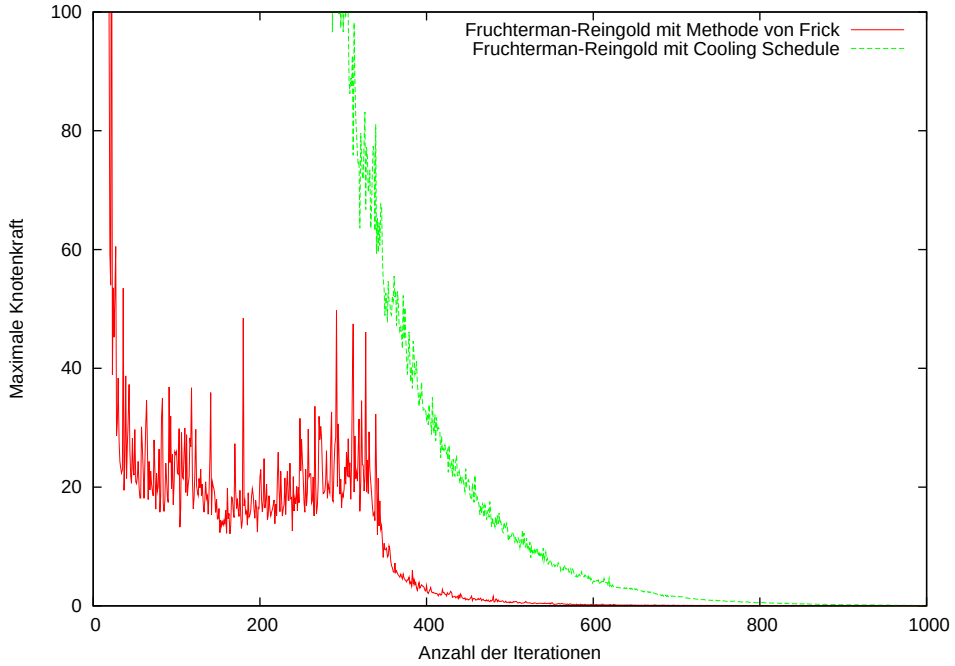


Abbildung 4.41.: Konvergenz der maximalen Knotenkräfte bei einem 16×16-Gitter.

4.3.2. Leistungsevaluierung

Konvergenz und Konvergenzgeschwindigkeit

Entscheidend für die korrekte Arbeitsweise der beiden Layoutmethoden ist die Konvergenz der maximalen Knotenkraft gegen Null. Wie in Abschnitt 3.4.1 beschrieben, hat die Verschiebungsfunktion Einfluss auf die Konvergenz. Für die Methode von Fruchterman-Reingold wurden zwei Varianten einer Verschiebungsfunktion vorgestellt, einmal eine globale Verschiebungsfunktion mit *cooling schedule* und zum anderen die Methode nach Frick. Anhand der oben vorgestellten Beispiele soll nun die Konvergenz und die Konvergenzgeschwindigkeit für beide Verschiebungsfunktionen aufgezeigt werden. Dazu wird die maximale Knotenkraft über die Anzahl der Iterationen dargestellt.

In Abbildung 4.41 wie auch in Abbildung 4.42 ist zu erkennen, dass bei beiden Verschiebungsfunktionsvarianten das Verfahren gegen Null konvergiert. Die Methode nach Frick weist jedoch bei beiden Beispielen die höhere Konvergenzgeschwindigkeit auf, so dass diese Variante bevorzugt wird.

Für regelmäßige rechteckige Gitter und binäre Bäume mit maximal 1000 Knoten ist die Konvergenzgeschwindigkeit ausgedrückt durch die Anzahl der Iterationen, die benötigt werden, um ein ansprechendes Layout zu erreichen, beim Verfahren von Fruchterman-Reingold mit der Verschiebungsmethode von Frick etwa zweieinhalbmal so groß wie die Anzahl der Knoten.

Diagramm 4.43 zeigt das Konvergenzverhalten beim Layoutverfahren von Kamada-Kawai. Auch hier ist zu erkennen, dass das Verfahren gegen Null konvergiert.

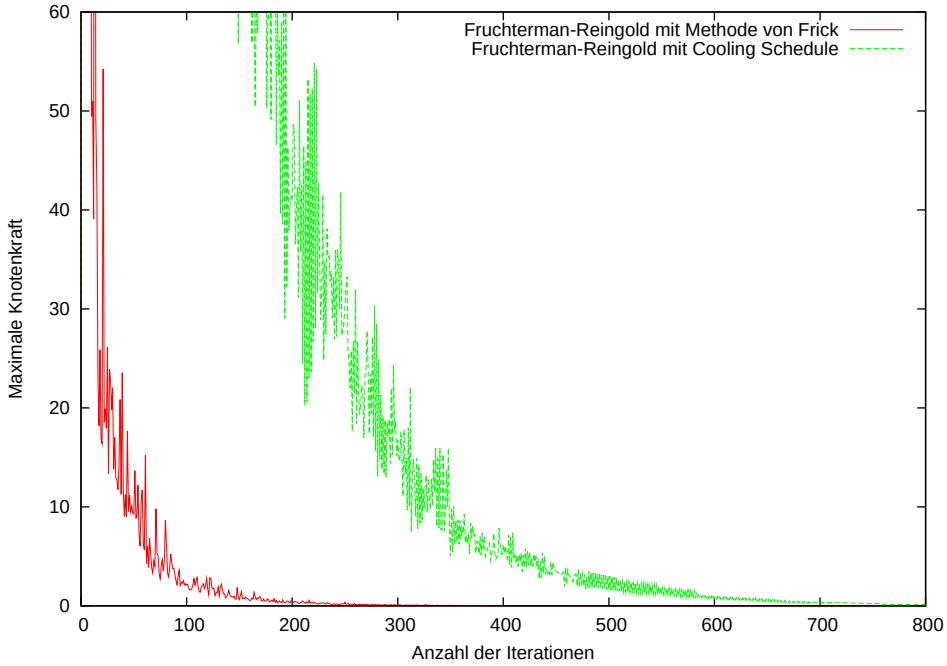


Abbildung 4.42.: Konvergenz der maximalen Knotenkräfte bei einem vollständigen binären Baum mit 5 Ebenen.

Für eine gut lesbare Zeichnung benötigt die Layoutmethode von Kamada-Kawai eine Anzahl an Iterationen, die etwa fünfzehnmal größer ist als die Knotenmenge. Die Konvergenzgeschwindigkeit bezüglich der Iterationsanzahl liegt damit erwartungsgemäß weit über der des Verfahrens von Fruchterman-Reingold.

Vergleich der Leistung auf den verschiedenen Architekturen

Die Abbildungen 4.44 und 4.45 zeigen die Laufzeiten des Fruchterman-Reingold Verfahrens für verschiedene Problemgrößen auf unterschiedlichen Architekturen. Die Anzahl der Iterationen ist für alle Größen und Architekturen konstant bei 400. Dadurch ist vergleichbar, inwieweit die Problemgröße und die Leistung des Systems die Rechenzeit beeinflusst. Anzumerken ist jedoch, dass nicht nur die Rechenzeit pro Iterationsschritt mit der Problemgröße ansteigt, sondern ein größerer Graph auch mehr Iterationen benötigt, bevor das Layout den ästhetischen Ansprüchen genügt.

Die CPU- und die Multicore-Implementierung nutzt weitgehend nur die von der HONEI-Bibliothek zur Verfügung gestellten Optimierungsmöglichkeiten. Anwendungsspezifische Optimierungen haben somit in dieser Konfiguration keine Einflüsse. Dies resultiert in den hohen Laufzeiten. Man kann erkennen, dass bereits für kleine Problemgrößen die Multicore-Implementierung einen deutlichen Leistungsgewinn mit sich bringt. Die Cell-Implementierung hingegen ist enttäuschend langsam. Es zeigt sich, dass für das Layouting von Graphen Optimierungen für Cell auf Ebene der Basisoperationen der linearen

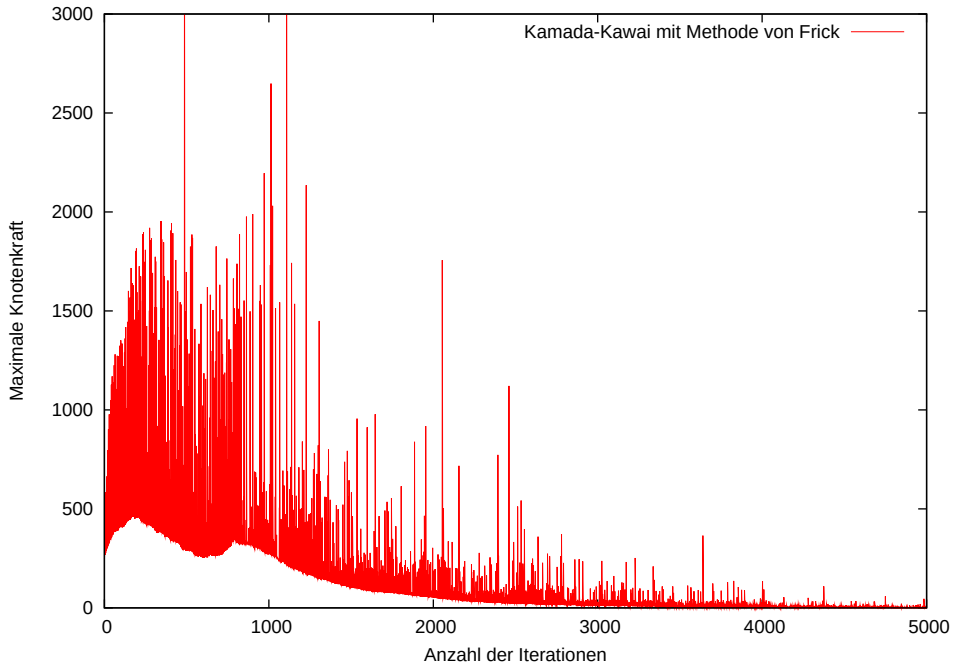


Abbildung 4.43.: Konvergenz der maximalen Knotenkräfte bei einem 16×16-Gitter mit Kamada-Kawai.

Algebra Bibliothek nicht ausreichen. Da die Matrizen sehr speziell sind (die Koordinatenmatrix hat nur zwei Spalten), müsste hier mehr auf Anwendungsebene optimiert werden, um die Leistung des Cell-Prozessors auszunutzen.

Im Gegensatz dazu sind die SSE-Implementierungen über die HONEI-Bibliothek hinaus speziell an die Berechnung der Fruchterman-Reingold Methode mit SSE-Unterstützung angepasst. Der erzielte Leistungsgewinn ist beachtlich und führt direkt zu der These, dass für das Fruchterman-Reingold Verfahren arichtekturspezifische Optimierungen auf Anwendungsebene der bessere Optimierungsansatz sind. Erstaunlich ist, dass die (ebenfalls optimierte) Multicore-Version der SSE-Implementierung schlechtere Ergebnisse liefert als die Singlecore-Variante.

Es bleibt nun die Frage offen, inwieweit der Leistungsgewinn der SSE- und Cell-Implementierungen auf die Optimierungen auf Anwendungsebene zurückzuführen sind und wie groß der Anteil der Spezialisierung der HONEI-Bibliothek ist. Hierzu wird für SSE und Cell gegenübergestellt, wie sich die Laufzeit für ein Gitter der Größe 16×16 und 400 Iterationen verhält, wenn weder HONEI- noch anwendungsspezifische Optimierungen benutzt werden, wenn nur die architekturspezifischen Optimierungen der HONEI-Bibliothek genutzt werden und wenn beides genutzt wird. Abbildung 4.46 zeigt die Ergebnisse im Vergleich.

Es wird deutlich, dass die anwendungsspezifischen Optimierungen in der Tat die Ursache für den Leistungsgewinn sind. Außerdem zeigt sich, dass es sich lohnt, die SSE-

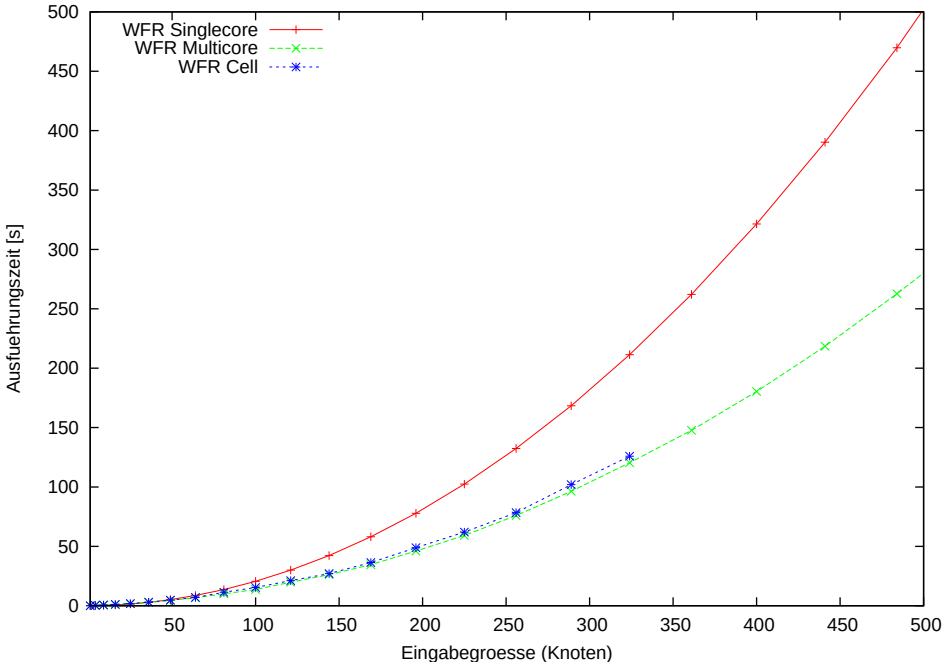


Abbildung 4.44.: Verschiedene Problemgrößen für Fruchterman-Reingold auf dem Core2Duo ohne Anwendungsoptimierungen und auf dem Cell

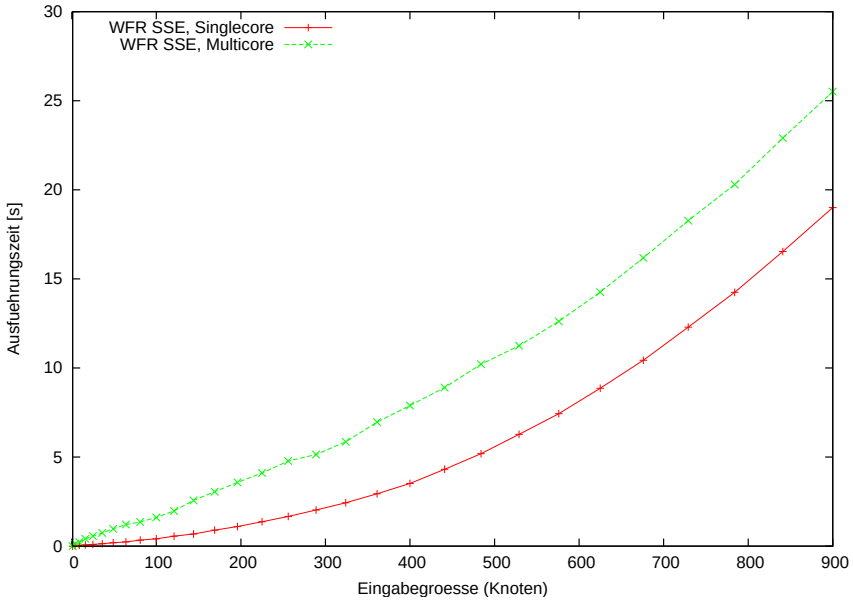


Abbildung 4.45.: Verschiedene Problemgrößen für Fruchterman-Reingold mit Anwendungsoptimierungen für SSE auf dem Core2Duo

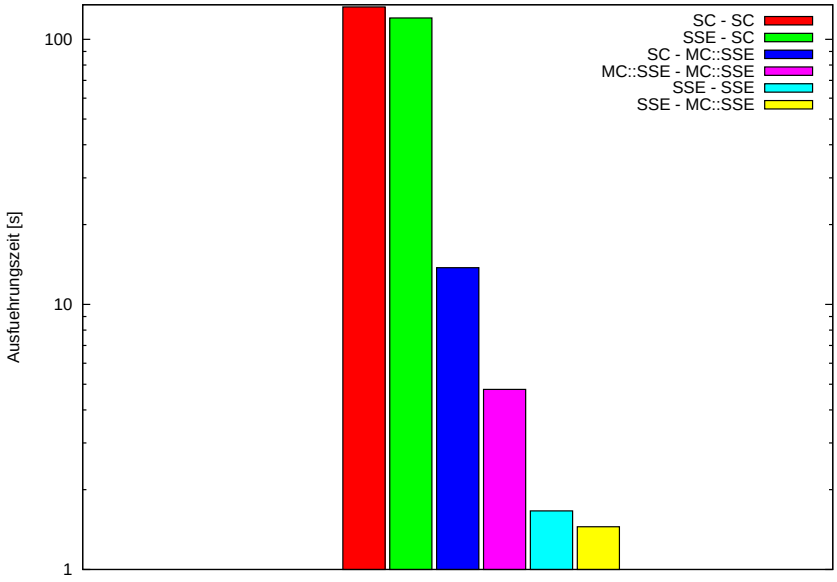


Abbildung 4.46.: Laufzeiten für 16×16 Gitter mit 400 Iterationen Fruchterman-Reingold mit unterschiedlicher Ausnutzung der Optimierung auf dem Core2Duo.

Multicore-Implementierung für die anwendungsspezifischen Optimierungen zu nutzen, obwohl die Multicore-SSE-Implementierung langsamer ist, sobald sie für die gesamte Berechnung des Fruchterman-Reingold Verfahrens eingesetzt wird. In Abbildung 4.47 und 4.48 wird diese Beobachtung für eine unterschiedliche Anzahl an Threads aufgegriffen und bestätigt. Abbildung 4.47 zeigt, dass auch auf dem Xeon die SSE-Variante schneller ist als die Multicore-SSE-Variante. Die Anzahl der Threads ist hierbei relativ unerheblich. Setzt man hingegen Multithreading selektiv nur für die optimierte Klasse `NodeDistance` ein, erreicht man einen Geschwindigkeitsvorteil gegenüber der Singlecore-Variante. 4.48 zeigt die Auswirkung der Anzahl der Threads für die Klasse `NodeDistance`.

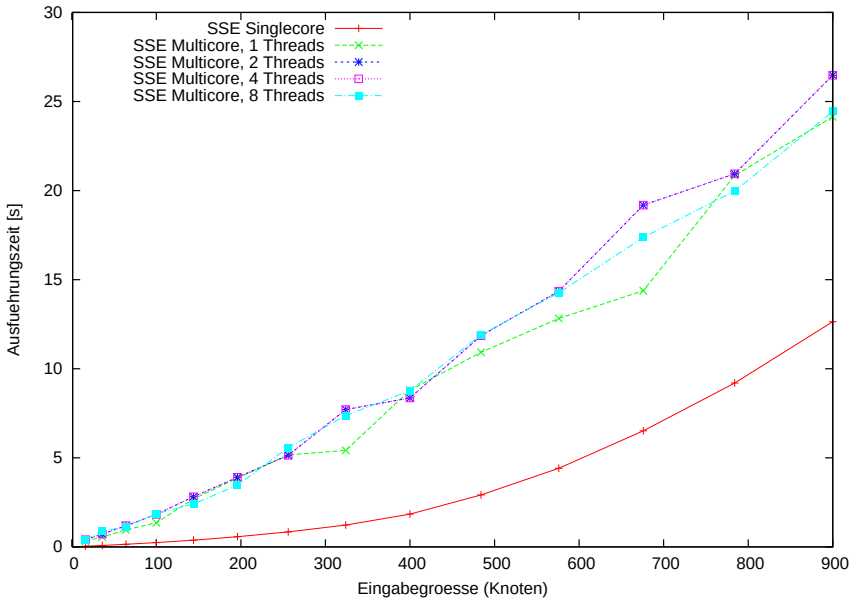


Abbildung 4.47.: Laufzeiten für verschiedene Gitter mit unterschiedlicher Anzahl an Threads auf dem Xeon-System

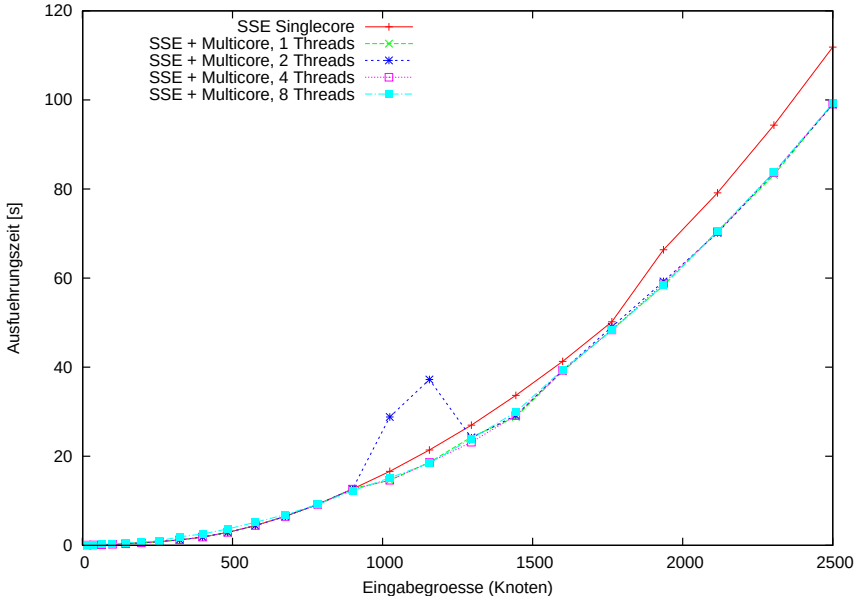


Abbildung 4.48.: Laufzeiten für verschiedene Gitter mit unterschiedlicher Anzahl an Threads für NodeDistance auf dem Xeon-System

4.4. Löser für lineare Gleichungssysteme

4.4.1. Korrektheit der Löser: Das Poisson-Problem

Um die Korrektheit der Löser für LGS zu testen, wurde das allgemeine Poisson Problem $-\Delta u = f$ mit einem Dirichlet Nullrand, d.h. $u = 0\ \forall u \in \Gamma, \Omega_d = \Omega \cup \Gamma$ herangezogen. Dieses wird i.A. als typisches Beispiel einer elliptischen Differentialgleichung herangezogen, die bei Problemen der *Computational Fluid Dynamics (CFD)* angetroffen wird, siehe hierzu beispielsweise Göddeke und Strzodka [SG06]. Bei den Tests erfolgt die Diskretisierung auf einem regulären Gitter mittels bilinearen, konformen Finiten Elementen. Bei diesem Problem sollte die L_2 Norm der Differenz zwischen berechneter und analytischer Lösung, $\|\mathbf{x}^* - \mathbf{x}_a\|_2$ bei Rechnung in ausreichender Genauigkeit um einen konstanten Faktor von 4 kleiner werden.[2] Alle Löser erfüllen dieses Kriterium. Für den nicht-vorkonditionierten Konjugierte Gradienten Löser ist der Verlauf des Diskretisierungsfehlers bei steigender Problemgröße in Abbildung 4.49 visualisiert. Bei diesen und allen folgenden Ergebnissen (inklusive der Poission-Benchmarks) wird der Löser stets so lange iteriert, bis keine Verbesserung in der Genauigkeit des Ergebnisses mehr zu verzeichnen ist. Wie man beobachten kann, sinkt der Diskretisierungsfehler bei doppelter Genauigkeit linear, während dies ab einem Detailgrad von L6 in einfacher Genauigkeit nicht mehr der Fall ist. Werte für die Fehlerreduktion finden sich in den Tabellen 4.5 und 4.6. Es wird der Diskretisierungsfehler für einfache und doppelte Genauigkeit und der Faktor, um den dieser im Verhältnis zum Fehler bei einem um eine Stufe kleineren Detailgrad gesunken ist, gezeigt. Für beide Implementierungen wurde ein Faktor von 4 in doppelter Genauigkeit verifiziert. Die SSE-Implementierung erreicht dieselbe Genauigkeit, wie die CPU-Referenzimplementierung. Auch der Cell Prozessor kann mit geringfügig schlechteren Ergebnissen dieses Kriterium erfüllen (siehe Abbildung 4.50, wo dies für den CG Löser dargestellt ist). Dazu ist jedoch anzumerken, dass die bereits in Abschnitt 4.1 erläuterten schlechteren numerischen Ergebnisse bei Ausführung eines Skalarproduktes mit den SPEs den Diskretisierungsfehler bei höheren Detailgraden stark (bis zu einem Faktor von zwei) vergrößert. Die hier präsentierten Ergebnisse wurden mit einer hybriden Rechnung, bei der ein Teil der Skalarprodukte auf der PPE berechnet werden, erzielt.

[2] Die Fehlerreduktion wird auf diese Weise in der einschlägigen Literatur untersucht. Für ein Beispiel sei an dieser Stelle auf Göddeke et al. [GST05] verwiesen

Detailgrad	N	Fehler float	Faktor	Fehler double	Faktor
L2	$\left(2^2+1\right)^2$	5.1845E-5	-	5.1842E-5	-
L3	$\left(2^3+1\right)^2$	1.2121E-5	4.28	1.2121E-5	4.28
L4	$\left(2^4+1\right)^2$	2.9944E-6	4.05	2.9748E-6	4.08
L5	$\left(2^5+1\right)^2$	1.0683E-6	2.80	7.4013E-7	4.00
L6	$\left(2^6+1\right)^2$	1.5614E-6	0.68	1.8480E-7	4.00
L7	$\left(2^7+1\right)^2$	5.7570E-6	0.27	4.6180E-8	4.00
L8	$\left(2^8+1\right)^2$	2.7530E-2	0.00	1.1546E-8	4.00
L9	$\left(2^9+1\right)^2$	3.8110E-1	0.07	2.8871E-9	4.00
L10	$\left(2^{10}+1\right)^2$	6.2065	0.06	7.2570E-10	4.00

Tabelle 4.5.: Fehlerreduktion bei CG , CPU Variante: Es wird jeweils die L2-Norm des Diskretisierungsfehlers für einfache und doppelte Genauigkeit und der Faktor, um den dieser Fehler beim Übergang auf eine höhere Anzahl von Unbekannten gesunken ist. Die Datenreihen für die Fehler enstprechen den Kurven float(CPU) und double(CPU) in Abbildung 4.49

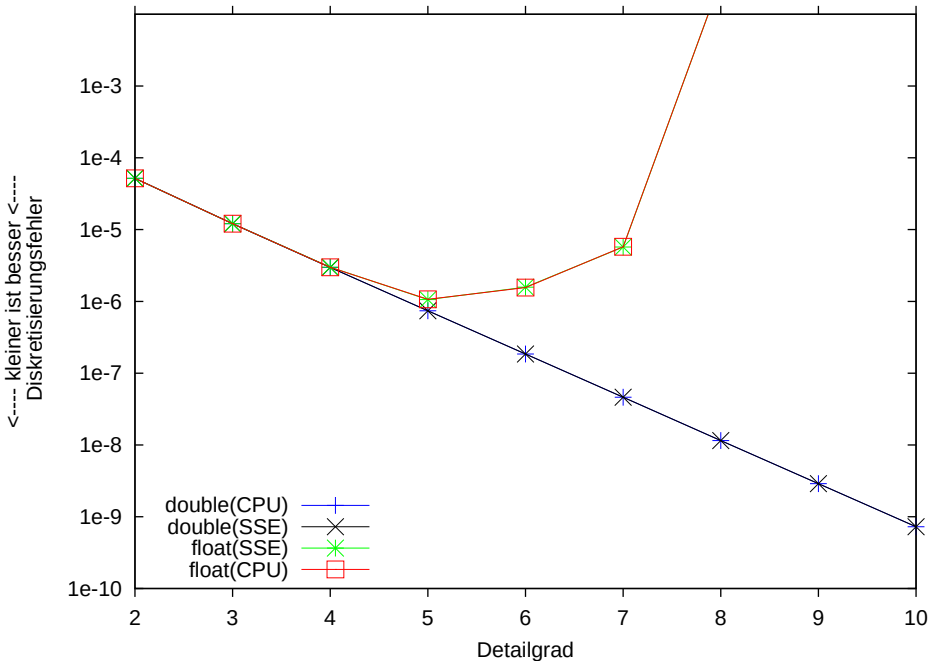


Abbildung 4.49.: Diskretisierungsfehler bei Lösung des Poisson Problems für den Konjugierte Gradienten Löser ohne Vorkonditionierung auf einem Core2Duo 6320.

Detailgrad	N	Fehler float	Faktor	Fehler double	Faktor
L2	$\left(2^2+1\right)^2$	5.1845E-5	-	5.1842E-5	-
L3	$\left(2^3+1\right)^2$	1.2123E-5	4.28	1.2123E-5	4.28
L4	$\left(2^4+1\right)^2$	2.9602E-6	4.04	2.9748E-6	4.08
L5	$\left(2^5+1\right)^2$	1.0591E-6	2.83	7.4013E-7	4.00
L6	$\left(2^6+1\right)^2$	1.5830E-6	0.67	1.8480E-7	4.00
L7	$\left(2^7+1\right)^2$	5.7112E-6	0.28	4.6188E-8	4.00
L8	$\left(2^8+1\right)^2$	2.6572E-1	0.00	1.1546E-8	4.00
L9	$\left(2^9+1\right)^2$	3.6886E-1	0.07	2.8869E-9	4.00
L10	$\left(2^{10}+1\right)^2$	4.3928	0.08	7.2518E-10	3.98

Tabelle 4.6.: Fehlerreduktion bei CG: Die handoptimierte SSE-Implementierung ist im Ergebnis so genau, wie die Referenzimplementierung.

4.4.2. Leistungsevaluierung

Die einfach und doppelt genauen CG und Jacobi Löser zeigen das bereits bei den Flachwassersimulationen beobachtete Verhalten bezüglich der Ausführungszeiten mit den verschiedenen Backends, wobei auch hier die Playstation 3 für realistische Problemgrößen hinter dem Core2Duo zurückbleibt. Für wesentlich größere Probleme schmilzt dieser Vorsprung jedoch zusammen. Auch diese Aussagen lassen sich auf den Jacobi Löser übertragen, der dieselben Verhältnisse zwischen Cell- und SSE-Implementierung aufweist. Die Rechenzeiten bis zur Konvergenz und die benötigte Anzahl von Iterationen sind in den Abbildungen 4.51 und 4.52 dargestellt. Dabei wird als Abbruchkriterium stets der Vergleich des Abstandes zweier aufeinander folgend berechneten Lösungen mit der kleinsten Zahl, die man zu 1.0 addieren kann um noch eine von eins verschiedene Zahl zu erhalten, verwendet. Der bei einem Detailgrad von acht zu verzeichnende Sprunghafte Anstieg der Iterationszahl in einfacher Genauigkeit ist bei allen Lösern zu verzeichnen.

Skalierbarkeit

Sowohl der Konjugierte Gradienten Löser, als auch der Jacobi Löser erreichen für realistische Problemgrößen für das Poisson Problem so gut wie keine Laufzeitgewinne bei der Nutzung mehr als einer SPE auf der Playstation 3. Erst ab einem Detailgrad von 9 kann für den CG Löser eine Verringerung der Rechenzeit verzeichnet werden, für den Jacobi Löser liegt dies jedoch bereits in einem Bereich, in dem ihn die hohe Iterationszahl unbrauchbar macht. Für den CG Löser ist dies in Abbildung 4.53 visualisiert. Der Einsatz von mehr als einer SPE ist im Gegenteil mit einem erheblichen Mehraufwand verbunden. Dennoch kann man durchaus die bei den Vektoroperationen verzeichneten Leistungsgewinne bei Hinzunahme von SPEs auf die Applikationsebene übertragen, wenn die Probleme hinreichend groß sind.

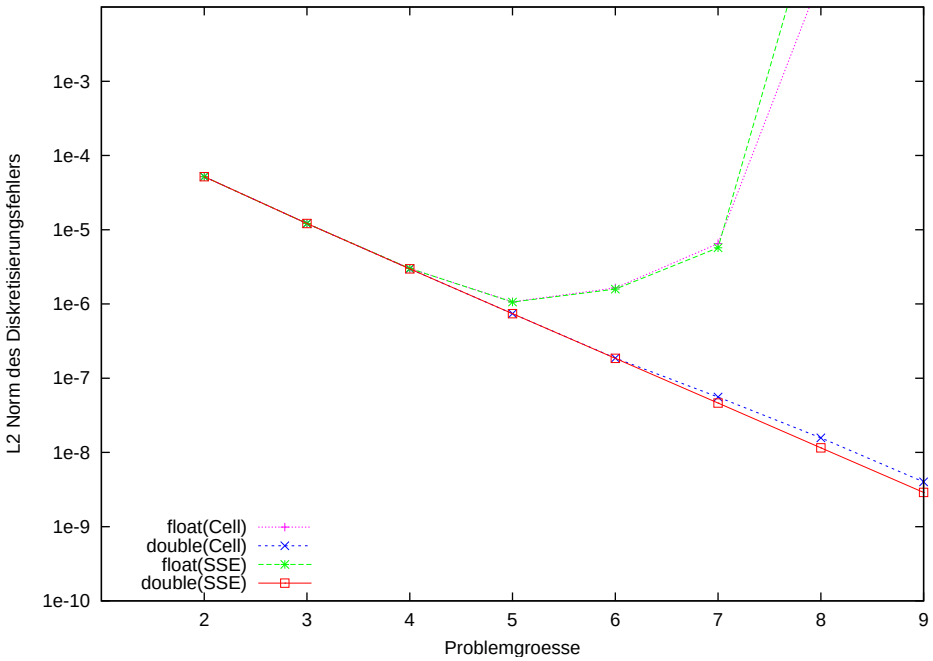


Abbildung 4.50.: Genauigkeit des CG Lösers: SSE vs Cell (Playstation 3)

Gemischte Genauigkeit

Für das SSE-Backend gilt, dass die gemischt genauen Konfigurationen dieselbe numerische Genauigkeit aufweisen, wie die doppelt genaue Rechnung, wie man in Abbildung 4.54 sieht. Dabei kann erst ab einem Detailgrad von 7 ein Unterschied in der Genauigkeit festgestellt werden. Für das Cell-Backend kann diese Beobachtung auch gemacht werden, jedoch ist dazu die bereits oben erwähnte hybride Rechnung notwendig, bei der die Skalarprodukte auf dem PPE gerechnet werden müssen.

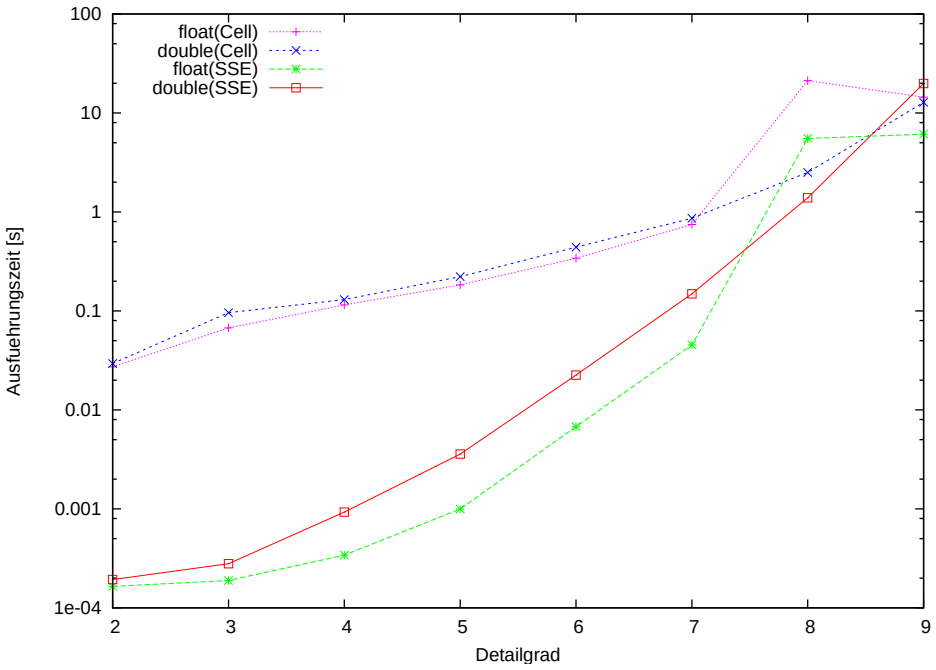


Abbildung 4.51.: Gesamtausführungszeiten des CG Lösers: Cell (Playstation 3) vs. SSE (Core2Duo)

Applikationsoptimierung

Wie bereits in den Abschnitten 4.2 und 4.3 festgestellt, ist das reine modulare Zusammensetzen von Applikationen aus Basiskomponenten problematisch. Hier soll nun für die Bibliothek von Gleichungssystemlösern ein analoger Nachweis erbracht werden, dafür, dass in solchen zusammengesetzten Anwendungen noch Optimierungspotential vorhanden ist. In Abbildung 4.55 ist die Rechenzeit für verschiedene Problemgrößen des in Abschnitt 4.4.1 dargestellten Poisson Problems und für die Jacobi Implementierung, die nur Basisoperationen benutzt, sowie die handoptimierte Variante dargestellt. Auch hier bestätigt sich die Überlegenheit letzterer Variante im Vergleich zu einer basisbibliothekgestützten Implementierung. In diesem Fall hat man es mit einem recht simplen iterativen Verfahren zu tun, das als Beispiel für eine applikationsspezifische Operation angesehen werden kann, wenn man die Lösung des Poisson Problems als Applikation ansieht.

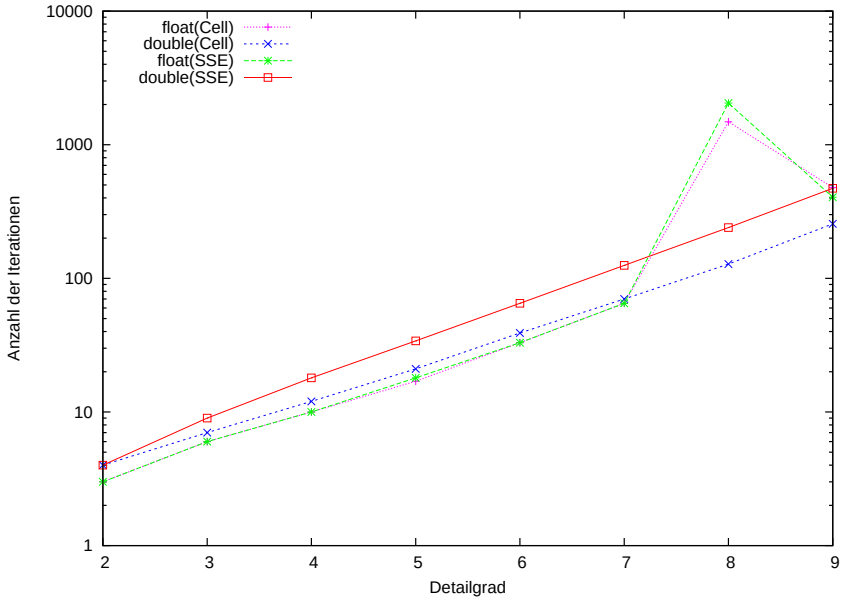


Abbildung 4.52.: Anzahl der Iterationen bis zur Konvergenz: CG: Cell (Playstation 3) vs. SSE (Core2Duo)

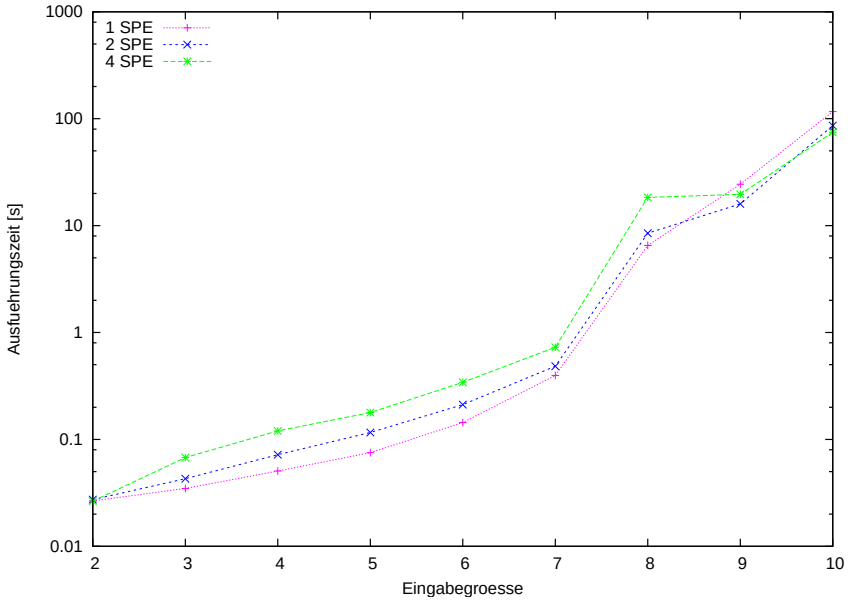


Abbildung 4.53.: Ausführungszeiten des CG Lösers für 1, 2, 4 SPEs auf der Playstation 3

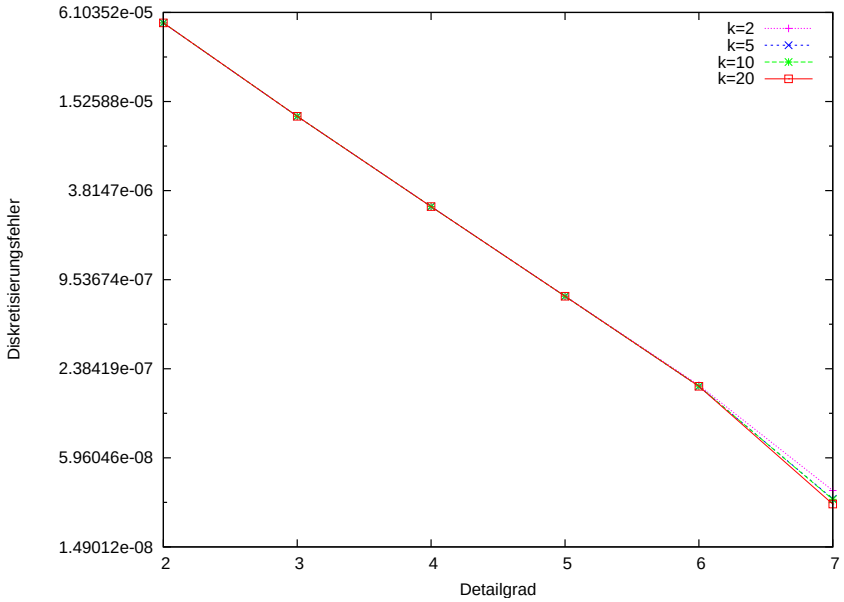


Abbildung 4.54.: Diskretisierungsfehler mit SSE für verschiedene gemischt genaue Konfigurationen.

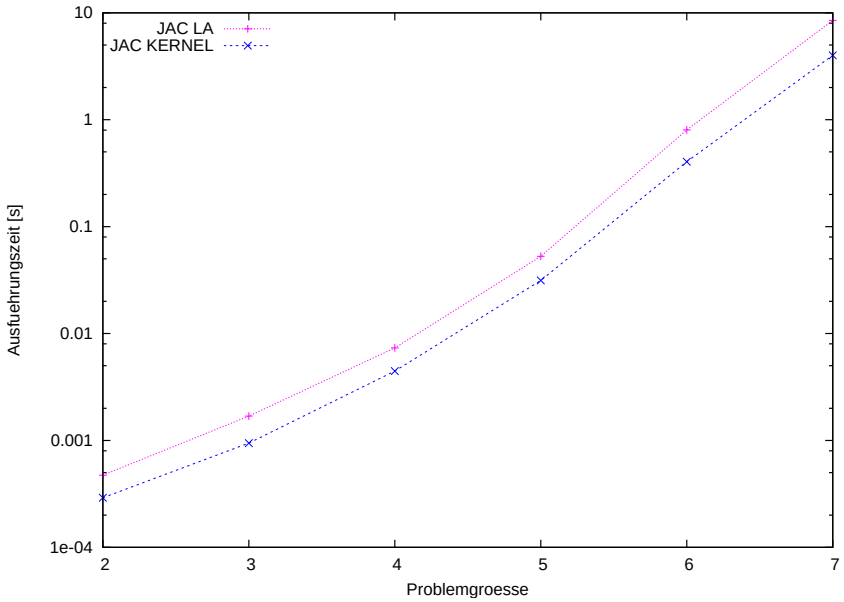


Abbildung 4.55.: Ausführungszeiten für `Jacobi` und `JacobiKernel` auf dem Core2Duo System. Die Y-Achse ist logarithmisch skaliert.

5. Abschlussbetrachtung

Im vorhergegangenen Kapitel wurden in Abschnitt 4.1 im Bereich der Operationen die folgenden Ergebnisse demonstriert:

- Die Operationen rechnen innerhalb akzeptabler Fehlerschranken korrekt.
- Die Skalierbarkeit auf der Cell BE ist bis zu den in Kapitel 4 angegebenen Schranken vorhanden.
- Die Skalierbarkeit beim MultiCore-Backend ist erst für Operationen mit hohem Grad an Wiederverwendbarkeit der Daten vorhanden.
- Für die Vektoroperationen wird auf den x86 Architekturen dieselbe Leistung erreicht wie durch die GotoBLAS.
- Das aktuelle Cell-Backend erreicht nicht die theoretisch maximale Speichertransferrate.

Bei den Anwendungen ergaben sich die in den Abschnitten 4.2, 4.3 und 4.4 erarbeiteten Erkenntnisse, die im folgenden noch einmal kurz zusammengefasst werden sollen:

- Die Flachwasseranwendung und die Löser für lineare Gleichungssysteme erreichen die gewünschten Ergebnisgenauigkeiten in doppelter Genauigkeit. Für die Cell BE kann dies jedoch nur mit einer Hybridlösung erreicht werden. Dabei müssen die Skalarprodukte auf dem PPE gerechnet werden, was mit einem erheblichen Geschwindigkeitsdefizit verbunden ist.
- Die gemischt genauen Konfigurationen des Flachwasserlösers erreichen annähernd dieselbe Ergebnisgenauigkeit, wie die ausschließlich in doppelter Genauigkeit rechnenden Konfigurationen. Auch bei den Gleichungssystemlösern kann dies erreicht werden.
- Die Flachwassersimulationsanwendung und die Löser für lineare Gleichungssysteme weisen bei größeren SPE Anzahlen und für hinreichend große Probleme die Leistungssteigerungen auf, die durch die Operationsbenchmarks prognostiziert wurden.
- Alle gemischt genauen Konfigurationen des Flachwasserlösers erreichen eine höhere Leistung als die fest genauen Varianten, insbesondere ist die Variante eins (jeder k-te Schritt wird doppelt genau gerechnet) bereits für $k = 2$ deutlich schneller als die doppelt genaue Variante. Bewährt hat sich die Variante, die innerhalb einer Iteration eine Konversion vornimmt und die zweite Vorhersagephase doppelt genau rechnet - sowohl in numerischer Hinsicht, als auch in Bezug auf die Performanz.

- Der Löser für die Flachwassergleichungen kann eine Problemgröße von 90x90 (SSE) auf einem gängigen PC-System bzw. 30x30 (Cell) auf der Playstation 3 in Echtzeit bewältigen. Der Leistungsunterschied der Architekturen ist auf ein unterschiedliches Optimierungsniveau zurückzuführen. Das SSE Backend erreicht mit dieser Anwendung eine Leistung von etwa 170 MFlop/s auf dem Core2Duo-Testsystem.
- Sowohl durch den Vergleich des monolithischen Lösers mit der modularen Implementierung, als auch durch die Gegenüberstellung von `Jacobi` und `JacobiKernel` konnte gezeigt werden, dass die jeweiligen Spezialisierungen schneller sind.
- Sowohl für die Flachwasser- als auch für die Poissonanwendung konnte das MultiCore Backend keine Leistungssteigerungen erreichen, da es sich in beiden Fällen um Folgen von Aufrufen von Vektoroperationen handelt.
- Die Playstation 3 kann im Vergleich mit einem gängigen PC-System bei einfacher Genauigkeit die Flachwassersimulationen unter der Prämisse desselben Optimierungsgrades des jeweiligen Backends mit einem deutlichen Leistungsgewinn durchführen.
- Bei den Layoutverfahren für Graphen zeigt sich, dass man mit Hilfe der Verschiebungsfunktion die Anzahl der Iterationen, die benötigt werden, um eine gut lesbare Zeichnung zu erhalten, wesentlich reduzieren kann. So ist die Konvergenzgeschwindigkeit beim Verfahren von Fruchterman-Reingold mit der Verschiebungsfunktion nach Frick deutlich höher als mit einer Verschiebungsfunktion mit *cooling schedule*.
- Das Layout von Graphen zeigt ferner, dass es nicht immer genügt, nur allgemeine Operationen für verschiedene Architekturen zu optimieren um das Rechenpotential auszureizen. Anwendungsspezifische Optimierungen sind für das Layout von Graphen unerlässlich. Diese wurde für Fruchterman-Reingold an der spezialisierten Operation `NodeDistance` vorgenommen und ermöglicht mit SSE-Unterstützung einen deutlichen Leistungsgewinn.

Die für die PG vorgegebenen Ziele, eine Bibliothek von Basisoperationen für den Cell Prozessor bereitzustellen und diese mit Hilfe von zwei Anwendungen auf ihre Praxistauglichkeit hin zu untersuchen sowie Referenzimplementierungen auf gängigen PC-Systemen zu entwickeln und zu optimieren, wurden erreicht. Zusätzlich dazu wurden mit den Lösern für lineare Gleichungssysteme weitere Applikationen implementiert, optimiert und evaluiert. Das zugrundeliege Programmiermodell basierte demnach auf der Annahme, dass die Applikationen, die von der Basisbibliothek Gebrauch machen, über die Nutzung der dort bereitgestellten Komponenten hinaus keine oder nur wenige eigene laufzeitkritische Komponenten benötigen. Das heißt, dass die Basisoperationen unter dem Gesichtspunkt der Wiederverwendbarkeit entwickelt wurden. Es konnte jedoch anhand des SSE-Backends gezeigt werden, dass dies im Allgemeinen schwierig ist, da selbst bei einfachen Applikationen Optimierungspotential verbleibt, wenn man sie modular aus Basiskomponenten zusammensetzt. Eine solche Bibliothek muss also stets auch applikationsspezifische Module enthalten um auch problemspezifische Optimierungsmöglichkeiten auszuschöpfen bzw. die Optimierung nicht dem Compiler zu überlassen. Es ist zu

erwarten, dass eine aus Zeitgründen nicht mehr durchgeführte Applikationsoptimierung für die Cell- und Multicore-Backends die Leistung erheblich steigern kann, wie am Beispiel der SSE-Implementierung der Flachwassersimulation und auch des Jacobi-Lösers exemplarisch gezeigt werden konnte. Da die Flachwasser- und Löseranwendungen durch die Verwendung von Bandmatrizen prinzipiell eine Folge von Vektoroperationsaufrufen darstellen und aufgrund der in Abschnitt 4.1.2 herausgestellten Überlegenheit des Cell-Backends bei Vektoroperationen, kann dieselbe Leistungssteigerung erwartet werden, wenn man die Applikationsoptimierung in gleicher Weise für das Cell-Backend durchführt. Des weiteren hat die Evaluierung der derzeitigen Cell-Implementierung ergeben, dass ein erheblicher Teil der Laufzeit von Operationen den Aufrufen der Basisoperationen zuzuschreiben ist. Durch Verwendung von Anwendungsoperationen ist zu erwarten, dass dieser Mehraufwand vermieden werden kann, da die Anzahl der Aufrufe damit tendenziell verringert wird.

Insgesamt ist die Playstation tatsächlich für wissenschaftliche Anwendungen interessant, wobei die numerischen Ergebnisse stets unter Vorbehalt betrachtet werden müssen, wie in Abschnitt 4.4.2 gezeigt wurde. Des weiteren erschwert die geringe Größe des Arbeitsspeichers die Berechnung größerer Probleme. Die Portierung von beliebigen Programmen auf die Cell BE stellt außerdem einen nicht zu unterschätzenden Aufwand dar.

Da die im engen Zeitrahmen der Projektgruppe erzielten Ergebnisse erhoffen lassen, dass mit mehr Aufwand und Optimierung noch bessere Leistungsdaten erzielt werden können, wird das HONEI-Projekt über die Dauer der Projektgruppe hinaus existieren. Die Entwicklung des Quellcodes wird fortan über die Webseite `http://www.honei.org` koordiniert und aktuelle Versionen werden zum Download zur Verfügung gestellt.

A. Anhang Bibliothekskomponenten

Containername	Kurzschreibweise	Beschreibung
`BandedMatrix<DataType_>`	BM	Bandmatrix
`DenseMatrix<DataType_>`	DM	dicht besetzte Matrix
`DenseMatrixTile<DataType_>`	DMT	Kachel einer DM
`DenseVectorBase<DataType_>`	DVB	Basis für dicht besetzte Vektoren
`DenseVectorContinuousBase<DataType_>`	DVCB	Basis für kontinuierlich im Speicher liegende Vektoren
`DenseVector<DataType_>`	DV	dicht besetzter Vektor
`DenseVectorRange<DataType_>`	DVR	kontinuierlicher Ausschnitt eines DV
`DenseVectorSlice<DataType_>`	DVS	nicht kontinuierlicher Ausschnitt eines DV
`SparseMatrix<DataType_>`	SM	dünn besetzte Matrix
`SparseVector<DataType_>`	SV	dünn besetzter Vektor

Tabelle A.1.: HONEI-Container

Operation	Funktionale Beschreibung	Beschreibung
`Absolute<Tag_>`	$Absolute(a) : a[i] \leftarrow \lvert a[i] \rvert$	Betragsfunktion
`Difference<Tag_>`	$Difference(a, b) : r \leftarrow a - b$	Differenz
`DotProduct<Tag_>`	$DotProduct(x, y) :\leftarrow x \cdot y$	Skalarprodukt
`ElementInverse<Tag_>`	$ElementInverse(a) : a[i] \leftarrow \begin{cases} 0, & falls\ a[i]{=}0 \\ (a[i])^{-1}, & sonst \end{cases}$	Elementweise Inversenbildung (Skalare)

`ElementProduct<Tag_>`	$ElementProduct(a, b) : a \leftarrow a[i] \cdot b[i]$	Elementweises Produkt
`Norm<vnt_1_one,Tag_>`	$Norm(x) : r \leftarrow \sum(\lvert x_0\rvert, \ldots, \lvert x_{size-1}\rvert)$	Betragsnorm
`Norm<vnt_2_one,Tag_>`	$Norm(x) : r \leftarrow \sqrt{\sum(\lvert x_0\rvert^2, \ldots, \lvert x_{size-1}\rvert^2)}$	Euklidische Norm (mit und ohne Quadratwurzel)
`Norm<vnt_max,Tag_>`	$Norm(x) : r \leftarrow \max(\lvert x_0\rvert, \ldots, \lvert x_{size-1}\rvert)$	Maximumsnorm
`Product<Tag_>`	$Product(a, b) : c \leftarrow a * b$	(nicht elementweises) Produkt
`Reduction<rt_max, Tag_>`	$Reduction(a) : x \leftarrow \max(a_0, \ldots, a_{size-1})$	Reduktion einer Entität auf ihr Element mit maximalem Wert
`Reduction<rt_min, Tag_>`	$Reduction(a) : x \leftarrow \min(a_0, \ldots, a_{size-1})$	Reduktion einer Entität auf ihr Element mit minimalem Wert
`Reduction<rt_sum, Tag_>`	$Reduction(a) : x \leftarrow \sum(a_0, \ldots, a_{size-1})$	Reduktion einer Entität auf die Summe ihrer Elemente
`Residual<Tag_>`	$Residual(b, A, x) : r \leftarrow b - A \cdot x$	Defekt/Residuum einer gegebenen Näherungslösung
`Scale<Tag_>`	$Scale(a, x); x[i] \leftarrow a \cdot x[i]$	Skalierung
`ScaledSum<Tag_>`	$ScaledSum(x, a, y) : x \leftarrow x + a \cdot y$	Addition eines Vektors und eines skalierten Vektors
`ScaledSum<Tag_>`	$ScaledSum(x, y, b) : x \leftarrow a \cdot x + b \cdot y$	Addition zweier skalierter Vektoren
`Sum<Tag_>`	$Sum(a, b) \leftarrow a + b$	Summe

Tabelle A.2.: Operationen der `libLA`

Operation	**Beschreibung**
`ConjugateGradients<Tag_,PreCon_>`	Berechnet Näherungslösung für ein gegebenes Gleichungssystem mit Hilfe des Konjugierten Gradientenverfahrens unter Verwendung eines Vorkonditionieres `PreCon_`

`Interpolation<Tag_, Method_>`	Interpolation (2D), Methoden: linear oder *Nearest Neighbour*
`IterativeRefinement<Method_, Tag_>`	Berechnet Näherungslösung durch Iterative Verfeinerung. Als Methoden `Method_` stehen das Konjugierte Gradientenverfahren `methods::CG` sowie das Jacobiverfahren `methods::JAC` zur Auswahl
`Jacobi<Tag_>`	Berechnet iterativ eine Näherungslösung für ein gegebenes Gleichungssystem, mit Hilfe des Jacobi-Verfahrens
`Quadrature<tags::Trapezoid, F_>`	Numerische Integration mit dem Trapezoid-Verfahren, mit Argumenttyp `F_`
`Sqrt<Tag_>`	Quadratwurzelberechnung für Containerklasse DVCB, DV, DVR: $Sqrt(a) : a[i] \leftarrow \sqrt{a[i]}$

Tabelle A.3.: Operationen der `libMath`

Operation	**Beschreibung**
`BreadthFirstSearch<Tag_>`	Bestimmung der Distanz für einen Graphen mit gewichteten Kanten mittels Tiefensuche (BFS)
`NodeDistance<Tag_>`	Algorithmus-Komponente für das Verfahren von Kamada-Kawai: *NodeDistance* berechnet die Distanzen zwischen Knoten, deren Lage in einer Positionsmatrix gegeben sind
`Positions<Tag_, DataType_, GraphTag_>`	Berechnet iterativ die Knotenpositionen eines Graphen mittels kräftebasierter Verfahren neu; die Eingabeparameter für `GraphTag_`, mit dem das Verfahren ausgewählt wird, sind `WeightedKamadaKawai` und `WeightedFruchtermanReingold`

Tabelle A.4.: Operationen der Bibliothekskomponente `libGraph`

Operation	**Beschreibung**
`AssemblyProcessing<Tag_,Type_>`	Assemblierung der Matrizen M_1 bis M_8 für das Relaxationsverfahren

`CorrectionProcessing<BoundaryType_,Tag_>`	Korrekturschritt des Runge-Kutta-Schemas mit `BoundaryType_`-Parameter für die Festlegung der Randbedingungen; zur Auswahl steht der Paramter `boundaries::REFLECT`
`FlowProcessing<Direction_, Tag_>`	Flussberechnung in `Direction`-Richtun.g
`PostProcessing<Type_>`	Ausgabe der berechneten Daten für verschiedene Ausgabetypen, zur Auswahl stehen: `output_types::GNUPLOT` für die Ausgabe im Gnuplot-Format und `output_types::HDF5` für Ausgabe in HDF5-definierte binäre Datendateien
`RelaxSolver<...>`	Aufruf des Relaxationslösers für 2D-SWE
`SourceProcessing<Type_,Tag_>`	Quelltermberechnung, wobei für `Type_` die Parameter `source_types::SIMPLE` und `source_types::INTEGRAL` zur Auswahl stehen

Tabelle A.5.: Operationen der Flachwasserbibliothek `libSWE`

B. Anhang Benchmarks

B.1. Operationsbenchmarks

Architektur	g++ Version	CXXFLAGS
Intel Core2Duo 6320	4.1.2 pre	-O2 -msse2 -mfpmath=sse -march=pentium-m
Intel Xeon E5345	4.1.2 pre	-O2 -msse2 -mfpmath=sse -march=pentium-m
AMD Opteron 844	4.1.2 pre	-O2 -msse2 -mfpmath=sse -march=opteron
AMD Dual Core Opteron 2214	4.1.2 pre	-O2 -msse2 -mfpmath=sse -march=opteron
Playstation 3	4.1.1	-O2 -fno-inline
Cellblade	4.1.1	-O2 -fno-inline

Tabelle B.1.: Verwendete Compiler

SPEs	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	900000	304.907	3489.39	0.0295029
2	900000	563.944	6453.83	0.01594
4	900000	910.376	10418.4	0.00983405
6	900000	917.992	10505.6	0.00974607
1	3900000	312.964	3581.59	0.124576
2	3900000	614.126	7028.11	0.063484
4	3900000	1158.1	13253.4	0.033648
6	3900000	1120.92	12827.9	0.0347211
1	6900000	314.052	3594.04	0.219646
2	6900000	621.33	7110.56	0.111013
4	6900000	1201.04	13744.8	0.0574169
6	6900000	1155.7	13225.9	0.059623
1	10650000	314.581	3600.09	0.338466
2	10650000	624.674	7148.83	0.170452
4	10650000	1221.4	13977.8	0.087132
6	10650000	1169.34	13382	0.090966
1	11400000	314.62	3600.54	0.362244
2	11400000	625.065	7153.31	0.182328
4	11400000	1224.31	14011.1	0.093071
6	11400000	1168.21	13369.1	0.0974541

Tabelle B.2.: Messdaten (Median) der Operation ScaledSum (double) auf der Playstation 3

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	900000	363.475	4159.65	0.0495219
2	900000	360.28	4123.08	0.0499611
1	3900000	350.726	4013.74	0.222396
2	3900000	334.853	3832.09	0.232938
1	6900000	349.763	4002.72	0.394553
2	6900000	334.918	3832.83	0.412041
1	10650000	350.861	4015.29	0.607078
2	10650000	335.465	3839.09	0.63494
1	11400000	349.618	4001.06	0.652141
2	11400000	335.473	3839.19	0.679637

Tabelle B.3.: Messdaten (Median) der Operation ScaledSum (MultiCore-SSE, double) auf dem Core2Duo

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	900000	287.141	3286.07	0.0626869
2	900000	287.801	3293.63	0.0625432
4	900000	287.798	3293.59	0.0625439
8	900000	287.872	3294.43	0.0625279
1	3900000	275.512	3152.99	0.283109
2	3900000	275.912	3157.56	0.282699
4	3900000	275.836	3156.69	0.282777
8	3900000	275.768	3155.92	0.282846
1	6900000	276.055	3159.2	0.4999
2	6900000	275.879	3157.19	0.500219
4	6900000	276.274	3161.71	0.499504
8	6900000	275.721	3155.38	0.500505
1	10650000	281.378	3220.11	0.756989
2	10650000	281.299	3219.21	0.757202
4	10650000	281.004	3215.83	0.757997
8	10650000	281.048	3216.34	0.757877
1	11400000	280.764	3213.09	0.812069
2	11400000	281.008	3215.88	0.811366
4	11400000	280.494	3210	0.812852
8	11400000	280.707	3212.44	0.812235

Tabelle B.4.: Messdaten (Median) der Operation ScaledSum (MultiCore-SSE, double) auf dem Intel Xeon

SPEs	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	180224	202.3	3086.85	0.0356112
2	180224	369.151	5632.8	0.0194881
4	180224	576.236	8792.67	0.0124769
6	180224	491.829	7504.72	0.0145719
1	507904	207.617	3167.99	0.0978408
2	507904	401.16	6121.22	0.050591
4	507904	722.681	11027.2	0.0280769
6	507904	580.704	8860.83	0.0348661
1	835584	208.717	3184.77	0.160044
2	835584	408.961	6240.25	0.0817001
4	835584	765.689	11683.5	0.043617
6	835584	763.799	11654.6	0.043644
1	1163264	209.315	3193.9	0.222241
2	1163264	412.331	6291.67	0.112803
4	1163264	785.242	11981.8	0.0591998
6	1163264	633.574	9667.58	0.0732331
1	1327104	209.484	3196.48	0.253345
2	1327104	413.382	6307.7	0.128357
4	1327104	792.262	12089	0.0669658
6	1327104	775.831	11838.2	0.068346

Tabelle B.5.: Messdaten (Median) der Operation Bandmatrix-Vektor Produkt (double) auf der Playstation 3

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	180224	373.163	5694.01	0.038636
2	180224	346.507	5287.28	0.0416081
1	507904	248.695	3794.79	0.16338
2	507904	226.087	3449.81	0.179718
1	835584	222.278	3391.69	0.300733
2	835584	211.395	3225.63	0.316215
1	1163264	214.595	3274.46	0.433658
2	1163264	208.234	3177.39	0.446905
1	1327104	212.938	3249.17	0.498587
2	1327104	207.236	3162.18	0.512303

Tabelle B.6.: Messdaten (Median) der Operation Bandmatrix-Vektor Produkt (MultiCore-SSE, double) auf dem Core2Duo

Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1600	97.3086	375.843	0.00657701
25600	861.773	3297.68	0.0475299
78400	1421.83	5433.52	0.154393
160000	2235.75	8539.35	0.286258
270400	2859.66	10919.2	0.491695
409600	3870.86	14777.7	0.677224
577600	4602.25	17567.7	0.953829
774400	4909.65	18739.5	1.38803
1000000	4897.72	18692.7	2.04177
1254400	4363.32	16652.2	3.21986
1537600	3711.45	14163.8	5.13714
1849600	3881.96	14813.9	6.47986
2190400	3821.84	14584.1	8.48228
2560000	3992.42	15234.6	10.2594
2958400	4030.6	15380	12.6245
3385600	4074.22	15546.1	15.2901
3841600	4122.67	15730.7	18.2637
4326400	4103	15655.5	21.9325

Tabelle B.7.: Messdaten (Median) der Operation Matrixprodukt (float) auf der Playstation 3 (4 SPEs)

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	25600	1528.99	5850.86	0.0267889
2	25600	2052.82	7855.37	0.019953
1	160000	2524.6	9642.61	0.253506
2	160000	4656.75	17786.3	0.137435
1	409600	2969.09	11335	0.882911
2	409600	5698.79	21756.2	0.459999
1	577600	2321.45	8861.46	1.89096
2	577600	4513.52	17229	0.972581
1	1000000	2589.34	9882.49	3.86199
2	1000000	5047.34	19263.7	1.98124
1	1537600	2780.16	10609.7	6.85797
2	1537600	5395.76	20591.5	3.53356
1	2190400	2323.42	8866.12	13.9527
2	2190400	4540.39	17326.1	7.1399
1	2958400	2463.58	9400.53	20.6547
2	2958400	4768.46	18195.5	10.6711
1	3841600	2605.46	9941.57	28.8991
2	3841600	5076.57	19370.5	14.8319
1	4326400	2749.08	10489.4	32.7343
2	4326400	5325.24	20319	16.8986

Tabelle B.8.: Messdaten (Median) der Operation Matrixprodukt (MultiCore-SSE, float) auf dem Core2Duo

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	25600	2481.24	9494.76	0.0165079
2	25600	3514.63	13449.2	0.0116541
4	25600	4408.15	16868.3	0.00929189
8	25600	3108.41	11894.7	0.0131772
1	160000	3849.53	14703.1	0.166254
2	160000	7028.8	26846.3	0.091054
4	160000	11822.3	45154.8	0.0541351
8	160000	11926.5	45553	0.0536618
1	409600	3424.58	13073.9	0.765479
2	409600	6783.3	25896.4	0.386455
4	409600	12015.7	45872.2	0.218167
8	409600	21820.6	83304	0.120136
1	577600	3585.36	13686.1	1.22436
2	577600	7072.83	26998.5	0.620651
4	577600	13433.7	51279.3	0.326772
8	577600	23078.2	88094.4	0.190212
1	1000000	2863.62	10929.3	3.49208
2	1000000	5624.06	21464.8	1.77808
4	1000000	10946.3	41777.7	0.913551
8	1000000	26232.3	100118	0.38121
1	1537600	3147.43	12011.3	6.05772
2	1537600	6248.51	23845.8	3.05133
4	1537600	12098.8	46171.9	1.57588
8	1537600	22479.4	85786.6	0.848166
1	2190400	3390.8	12939.3	9.56055
2	2190400	6673.66	25466.6	4.85759
4	2190400	13226.4	50471.8	2.451
8	2190400	23811.8	90865.7	1.36142
1	2958400	3561.37	13589.5	14.2879
2	2958400	7013.94	26763.8	7.25477
4	2958400	14019.6	53496	3.62953
8	2958400	26006.8	99236.9	1.95658
1	3841600	3716.39	14180.5	20.2604
2	3841600	7343.56	28020.6	10.2532
4	3841600	14662.7	55948.1	5.13516
8	3841600	27026.7	103125	2.78596
1	4326400	3080.24	11753	29.215
2	4326400	6011.22	22936.5	14.9702
4	4326400	11990.3	45750.5	7.50513
8	4326400	22196.1	84691.8	4.05428

Tabelle B.9.: Messdaten (Median) der Operation Matrixprodukt (MultiCore-SSE, float) auf dem Intel Xeon

Threads	Eingabegröße	MFlop/s	MB/s	Ausführungszeit
1	25600	632.734	2421.23	0.0647349
2	25600	1152.57	4410.45	0.035538
4	25600	1117.05	4274.52	0.0366681
1	160000	1939.88	7409.29	0.329918
2	160000	3830.18	14629.2	0.167094
4	160000	6358.92	24287.7	0.100646
1	409600	1875.15	7158.74	1.39799
2	409600	3683.22	14061.3	0.711726
4	409600	6732	25700.6	0.3894
1	577600	1882.02	7184.07	2.33247
2	577600	3488.32	13315.6	1.25842
4	577600	6742.99	25739.4	0.651011
1	1000000	1665.87	6357.97	6.00286
2	1000000	3238.44	12359.8	3.08791
4	1000000	6170.56	23550.6	1.6206
1	1537600	1805.49	6890.18	10.5601
2	1537600	3549.77	13546.8	5.37112
4	1537600	6850.45	26142.9	2.78321
1	2190400	1773.3	6766.9	18.2811
2	2190400	3465.44	13224.1	9.35464
4	2190400	6647.84	25368.1	4.87646
1	2958400	1507.48	5752.26	33.7546
2	2958400	2992.71	11419.6	17.0028
4	2958400	5797.67	22122.8	8.77671
1	3841600	1708.7	6519.85	44.0658
2	3841600	3366.47	12845.4	22.3662
4	3841600	6578.24	25100.4	11.4461
1	4326400	1719.86	6562.33	52.3235
2	4326400	3413.59	13025	26.362
4	4326400	6645.76	25357.7	13.5408

Tabelle B.10.: Messdaten (Median) der Operation Matrixprodukt (MultiCore-SSE, float) auf dem AMD DualCore

B.2. Anwendungsbenchmarks

B.2.1. Flachwassersimulation

Eingabegröße	N (Op.)	Cell	SSE	MCSSE
21	1875	0.0536389	0.00269294	0.006382
41	6075	0.104029	0.0083189	0.016341
61	12675	0.190148	0.018429	0.032513
81	21675	0.292866	0.035562	0.055801
101	33075	0.441415	0.062618	0.08428
121	46875	0.605326	0.0901792	0.119126
141	63075	0.800082	0.118159	0.16075
161	81675	1.03255	0.147824	0.209441
181	102675	1.2802	0.180974	0.263622
201	126075	1.595	0.218315	0.325129
221	151875	1.91099	0.262098	0.395579
241	180075	2.2904	0.309282	0.478779
261	210675	2.70113	0.358679	0.56472

Tabelle B.11.: Ausführungszeiten für einen Zeitschritt des Flachwasserlösers auf der Playstation 3 und dem Core2Duo, einfache Genauigkeit

Eingabegröße	N (Op.)	1 SPE	2 SPEs	4 SPEs
21	1875	3.26762	3.9757	5.0851
41	6075	8.5515	9.42212	10.1289
61	12675	16.6857	17.677	19.3276
81	21675	28.081	28.9079	30.463
101	33075	42.5884	43.0079	44.3073
121	46875	59.5778	60.0736	60.8315
141	63075	79.1343	79.2909	80.2096
161	81675	103.318	102.913	103.829
181	102675	128.539	127.306	128.041
201	126075	159.39	160.551	160.437
221	151875	193.301	191.573	190.516
241	180075	231.864	231.488	229.185
261	210675	272.396	272.114	267.584
281	243675	323.929	320.461	321.91
301	279075	366.172	361.359	358.71
321	316875	416.731	411.779	411.043
341	357075	477.695	466.962	467.411

Tabelle B.12.: Ausführungszeiten des Flachwasserlösers auf der Playstation 3 für 100 Zeitschritte, einfache Genauigkeit für 1, 2 und 4 SPEs.

Eingabegröße	N(Op.)	1 Thread	2 Threads
21	1875	0.454062	0.504605
41	6075	1.12906	1.29708
61	12675	2.40871	2.53284
81	21675	4.48008	4.5047
101	33075	7.23728	7.21444
121	46875	10.005	10.0073
141	63075	13.0922	13.0527
161	81675	16.6993	16.5704
181	102675	20.6137	20.4724
201	126075	25.0703	24.8694
221	151875	29.9974	29.7912
241	180075	35.3713	35.1185
261	210675	41.8064	40.8947

Tabelle B.13.: Ausführungszeiten des Flachwasserlösers auf dem Core2Duo für 100 Zeitschritte, einfache Genauigkeit, 1 und 2 Threads

Eingabegröße	N(Op.)	k=15	k=5	K=2	mixed pred.
21	1875	5.98196	6.40446	7.06831	5.62081
41	6075	11.8446	13.2916	15.8956	10.9753
61	12675	20.8151	23.6529	29.3442	19.8915
81	21675	32.9565	38.1591	47.9618	31.746
101	33075	49.7786	57.0371	72.1498	47.4609
121	46875	68.4803	78.5536	100.57	65.2668
141	63075	91.043	104.483	134.415	85.9969
161	81675	118.244	135.689	175.873	111.576
181	102675	146.597	168.893	217.817	138.278
201	126075	181.735	209.369	270.195	172.651
221	151875	218.3	252.31	323.961	206.655
241	180075	262.412	299.176	388.376	243.914
261	210675	309.903	351.859	455.349	287.103

Tabelle B.14.: Ausführungszeiten des Flachwasserlösers auf der Playstation 3 für 100 Zeitschritte, gemischte Genauigkeit, k=15

B.2.2. Kräftebasiertes Layout von Graphen

Eingabegröße	CPU 1 Thread	CPU 2 Threads	Cellblade
1	9.6e-05	0.00013	0.000416994
4	0.076618	0.26114	0.378653
9	0.244512	0.514011	0.655735
16	0.639546	0.955676	1.10119
25	1.4274	1.74249	1.84361
36	2.84368	2.67305	2.97966
49	5.13652	4.51423	4.69596
64	8.59523	6.72192	6.99468
81	13.624	9.95668	11.3648
100	20.5713	13.6535	15.3478
121	29.9971	19.7445	21.1534
144	42.2412	26.1224	27.0845
169	58.0779	34.6424	36.3818
196	77.7903	46.0384	48.8682
225	102.397	59.181	62.0482
256	132.447	75.8667	78.3911
289	168.406	96.3343	101.985
324	211.359	120.367	125.872
361	262.135	147.718	n.V.
400	321.48	180.4	n.V.
441	390.218	218.446	n.V.
484	469.78	262.72	n.V.
529	561.221	311.726	n.V.
576	665.903	367.118	n.V.
625	783.774	n.V.	n.V.
676	915.13	n.V.	n.V.
729	1065.02	n.V.	n.V.

Tabelle B.15.: Ausführungszeiten für 400 Iterationen Fruchterman-Reingold auf dem Cellblade und auf dem Core2Duo ohne Anwendungsoptimierungen

Eingabegröße	SSE	SSE Multicore
1	0.00013	0.000281
4	0.026752	0.124861
9	0.044452	0.226613
16	0.066262	0.407761
25	0.09291	0.561473
36	0.135312	0.738631
49	0.188992	0.970861
64	0.238127	1.21005
81	0.334989	1.35575
100	0.410685	1.61027
121	0.557561	1.97367
144	0.682169	2.56997
169	0.892901	3.05588
196	1.09946	3.57707
225	1.3694	4.10431
256	1.66427	4.77983
289	2.02938	5.14239
324	2.43342	5.85356
361	2.93695	6.96242
400	3.51977	7.89216
441	4.31093	8.9023
484	5.18568	10.2068
529	6.2747	11.2476
576	7.43038	12.6173
625	8.85794	14.262
676	10.4331	16.1821
729	12.2878	18.2659
784	14.2421	20.2989
841	16.534	22.9041
900	19.0057	25.5092

Tabelle B.16.: Ausführungszeiten für 400 Iterationen Fruchterman-Reingold auf dem Core2Duo mit Anwendungsoptimierungen

Bibliothek optimiert für	Anwendung optimiert für	Laufzeit
CPU Singlecore	CPU Singlecore	132.447
CPU SSE	CPU Singlecore	120.377
CPU Singlecore	CPU SSE Multicore	13.7622
CPU SSE Multicore	CPU SSE Multicore	4.77983
CPU SSE	CPU SSE	1.66427
CPU SSE	CPU SSE Multicore	1.45016

Tabelle B.17.: Ausführungszeiten für 400 Iterationen Fruchterman-Reingold bei einem 16×16 Gitter auf dem Core2Duo mit unterschiedlichen Anwendungsoptimierungen

Eingabegröße	1 Thread	2 Threads	4 Threads	8 Threads
16	0.315939	0.401331	0.401331	0.404697
36	0.559575	0.719098	0.719098	0.894332
64	0.956317	1.19481	1.19481	1.11626
100	1.35628	1.82843	1.82843	1.85783
144	2.6973	2.81777	2.81777	2.41502
196	3.85305	3.91107	3.91107	3.462
256	5.16292	5.14459	5.14459	5.57334
324	5.41559	7.71206	7.71206	7.37282
400	8.79834	8.35918	8.35918	8.76515
484	10.9201	11.8504	11.8504	11.9093
576	12.8267	14.3499	14.3499	14.2651
676	14.3805	19.182	19.182	17.3728
784	20.8475	20.9464	20.9464	19.9722
900	24.1426	26.4727	26.4727	24.4398

Tabelle B.18.: Ausführungszeiten für 400 Iterationen Fruchterman-Reingold auf dem Xeon mit Multicore::SSE und unterschiedlicher Threadanzahl

Eingabegröße	SSE ohne Multicore	Multicore 1 Thread	2 Threads	4 Threads	8 Threads
16	0.0411708	0.0412009	0.0412068	0.0411689	0.054601
36	0.0817218	0.0816212	0.0812519	0.081327	0.108377
64	0.14576	0.146029	0.145484	0.145504	0.193648
100	0.242743	0.243695	0.243472	0.242257	0.344858
144	0.379638	0.380539	0.380608	0.379274	0.524204
196	0.575862	0.578058	0.575952	0.575789	0.749907
256	0.84236	0.843711	0.843439	0.84226	1.06316
324	1.22504	1.22601	1.22858	1.23308	1.87933
400	1.84211	1.83576	1.85018	1.85067	2.56647
484	2.9194	2.87898	2.89584	2.91488	3.66383
576	4.41531	4.45772	4.42368	4.44282	5.18865
676	6.5153	6.54126	6.49414	6.46078	6.89502
784	9.19954	9.21676	9.20046	9.20277	9.18929
900	12.6416	12.6927	12.6499	12.6066	12.2245
1024	16.6122	14.6044	28.8166	14.6216	15.1449
1156	21.4029	18.6265	37.2193	18.5882	18.4684
1296	26.9704	24.3092	24.155	23.1559	23.9042
1444	33.6678	28.8356	29.1568	29.1384	29.9409
1600	41.2953	39.0659	39.2764	39.2965	39.414
1764	50.1835	48.3969	49.0446	48.3686	48.2508
1936	66.3593	58.293	59.2419	58.7977	58.4559
2116	79.1437	70.3936	70.1652	70.4172	70.5627
2304	94.3315	83.1893	83.6356	83.7464	83.7359
2500	111.857	99.142	98.9508	99.0588	99.245

Tabelle B.19.: Ausführungszeiten für 400 Iterationen Fruchterman-Reingold auf dem Xeon mit SSE für die Bibliothek und (wenn nicht anders angegeben) Multicore-SSE mit unterschiedlicher Threadanzahl für die Anwendungsoptimierungen

B.2.3. Löser für Lineare Gleichungssysteme

Detailgrad	# Iter.(float)	Zeit(float)	# Iter.(double)	Zeit(double)
L2	3	0.0272858	4	0.0294271
L3	6	0.0673759	7	0.0959949
L4	10	0.11487	12	0.130094
L5	17	0.183027	21	0.221797
L6	33	0.34088	39	0.440746
L7	65	0.746127	70	0.863249
L8	1484	21.2504	128	2.50507
L9	478	14.4821	256	12.7992
L10	677	76.1425	508	121.947

Tabelle B.20.: Ausführungszeiten und Iterationszahl des CG auf der Playstation 3.

Detailgrad	1 SPE	2 SPEs	4 SPEs
L2	0.026551	0.0273149	0.0264561
L3	0.0347521	0.0429361	0.0676291
L4	0.050627	0.071928	0.120076
L5	0.0750949	0.116034	0.178359
L6	0.143947	0.211518	0.342464
L7	0.396258	0.483283	0.728814
L8	6.52154	8.48788	8.3676
L9	24.4198	15.9573	19.6109
L10	116.633	86.2628	74.6521

Tabelle B.21.: Ausführungszeiten des CG auf der Playstation 3, einfache Genauigkeit für 1, 2 und 4 SPEs

Detailgrad	`Jacobi`	`JacobiKernel`
L2	0.000473	0.000291
L3	0.001687	0.000942999
L4	0.00733	0.00445
L5	0.052705	0.031364
L6	0.803086	0.404489
L7	8.49831	4.00836

Tabelle B.22.: Ausführungszeiten des Jacobi Lösers auf dem Core2Duo, einfache Genauigkeit, optimiert und unoptimiert

C. Anhang Korrektheit

C.1. Flachwassersimulation

T	float	double	mixed k=15	mixed k=5	mixed k=2	mixed pred.
550	5.0781E-05	3.517395E-08	4.609375E-05	3.870735E-05	2.375625E-05	2.63067E-05
5000	5.3906E-05	8.98987E-05	5.000005E-05	4.306665E-05	1.65692E-05	2.221435E-05

Tabelle C.1.: Ausgewählte Werte für den relativen Volumenfehler auf der Playstation 3.

C.2. Löser für lineare Gleichungssysteme

Detailgrad	N	Fehler float	Faktor	Fehler double	Faktor
L2	$\left(2^2+1\right)^2$	5.1845E-5	-	5.1842E-5	-
L3	$\left(2^3+1\right)^2$	1.2121E-5	4.28	1.2121E-5	4.28
L4	$\left(2^4+1\right)^2$	2.9944E-6	4.05	2.9748E-6	4.08
L5	$\left(2^5+1\right)^2$	1.0683E-6	2.80	7.4013E-7	4.00
L6	$\left(2^6+1\right)^2$	1.5614E-6	0.68	1.8480E-7	4.00
L7	$\left(2^7+1\right)^2$	5.7570E-6	0.27	4.6180E-8	4.00
L8	$\left(2^8+1\right)^2$	2.7530E-2	0.00	1.1546E-8	4.00
L9	$\left(2^9+1\right)^2$	3.8110E-1	0.07	2.8871E-9	4.00
L10	$\left(2^{10}+1\right)^2$	6.2065E-0	0.06	7.2570E-10	4.00

Tabelle C.2.: Fehlerreduktion bei CG , CPU Variante

Detailgrad	N	Fehler float	Faktor	Fehler double	Faktor
L2	$\left(2^2+1\right)^2$	5.1845E-5	-	5.1842E-5	-
L3	$\left(2^3+1\right)^2$	1.2123E-5	4.28	1.2123E-5	4.28
L4	$\left(2^4+1\right)^2$	2.9602E-6	4.04	2.9748E-6	4.08
L5	$\left(2^5+1\right)^2$	1.0591E-6	2.83	7.4013E-7	4.00
L6	$\left(2^6+1\right)^2$	1.5830E-6	0.67	1.8480E-7	4.00
L7	$\left(2^7+1\right)^2$	5.7112E-6	0.28	4.6188E-8	4.00
L8	$\left(2^8+1\right)^2$	2.6572E-1	0.00	1.1546E-8	4.00
L9	$\left(2^9+1\right)^2$	3.6886E-1	0.07	2.8869E-9	4.00
L10	$\left(2^{10}+1\right)^2$	4.3928E-0	0.08	7.2518E-10	3.98

Tabelle C.3.: Fehlerreduktion bei CG , SSE Variante

Literaturverzeichnis

[Bec06] Becker, Hendrik: *GPU-basierte Echzeit-Simulation und Visualisierung von Shallow Water Equations*, Universität Dortmund, Diplomarbeit, 2006

[Bra02] Brandenburg, Franz J. Zeichnen von Graphen oder Visualisierung von Graphen (Vorlesungsskript). `http://www.infosun.fim.uni-passau.de/br/lehrstuhl/Kurse/gd_02/`. 2002

[DK05] Delis, Anargiros I. und Katsaounis, Theodoros D.: Numerical solution of the two-dimensional shallow water equations by the application of relaxation methods. In: *Applied Mathematical Modelling* 29 (2005), Nr. 8, S. 754–783

[FKN+03] Forrester, David und Kobourov, Stephen G. und Navabi, Armand und Wampler, Kevin und Yee, Gary V. GraphAEL. `http://graphael.cs.arizona.edu/`. 2003

[FKN+04] Forrester, David und Kobourov, Stephen G. und Navabi, Armand und Wampler, Kevin und Yee, Gary V.: GraphAEL: A System for Generalized Force-Directed Layouts. In: *Graph Drawing*, Springer, 2004, S. 454–464

[FLM95] Frick, Arne und Ludwig, Andreas und Mehldau, Heiko: A Fast Adaptive Layout Algorithm for Undirected Graphs. In: *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing* Bd. 894, 1995, S. 388–403

[FR91] Fruchterman, Thomas M. J. und Reingold, Edward M.: Graph Drawing by Force-directed Placement. In: *Softw. Pract. Exper.* 21 (1991), Nr. 11, S. 1129–1164

[GST05] Göddeke, Dominik und Strzodka, Robert und Turek, Stefan: Accelarating Double Precision FEM Simulations with GPUS. Frontiers in Simulation (2005), S. 139–144

[GST07] Göddeke, Dominik und Strzodka, Robert und Turek, Stefan: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. In: *International Journal of Parallel, Emergent and Distributed Systems* 22 (2007), August, Nr. 4, S. 221–256

[HHHL06] Hagen, Trond und Henriksen, Martin und Hjelmervik, Jon und Lie, Knut-Andreas: How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine. In:

Geometrical Modeling, Numerical Simulation, and Optimisation: Industrial Mathematics at SINTEF, Springer, 2006, S. 211–264

[Hol07] Hollander, Rhett M. RAMspeed. `http://www.alasir.com/software/ramspeed/`. 2007

[IBM05] IBM. IBM Cell Broadband Engine Software Development Kit. `http://www.alphaworks.ibm.com/tech/cellsw`. 2005

[Int07] Intel Corporation. P.O. Box 5937, Denver, CO 80217-9808, USA: *Intel 64 and IA-32 Architectures Software Developer's Manual*. May 2007

[JBHP06] Jünger, M. und Buchheim, C. und Hachula und Percan, M. Automatisches Zeichnen von Graphen (Vorlesungsskript). `http://www.informatik.uni-koeln.de/ls_juenger/teaching/ss_06/gd/`. 2006

[Kaza] Kazushige Goto. GotoBLAS. `http://www.tacc.utexas.edu/resources/software/#blas`

[Kazb] Kazushige Goto. GotoBLAS FAQ. `http://www.tacc.utexas.edu/resources/software/gotoblasfaq.php`

[KDH+05] Kahle, James A. und Day, Michael N. und Hofstee, H. P. und Johns, Charles R. und Maeurer, Theodore R. und Shippy, David: Introduction to the Cell multiprocessor. In: *IBM Journal of Research and Development* 49 (2005), Nr. 4-5, S. 589–604

[KK89] Kamada, Tomihisa und Kawai, Satoru: An algorithm for drawing general undirected graphs. In: *Inf. Process. Lett.* 31 (1989), Nr. 1, S. 7–15

[KW01] Kaufmann, Michael (Hrsg.) und Wagner, Dorothea (Hrsg.): *Lecture Notes in Computer Science*. Bd. 2025 : Drawing Graphs: Methods and Models. Springer, 2001

[LLL+06] Langou, Julie und Langou, Julien und Luszczek, Piotr und Kurzak, Jakub und Buttari, Alfredo und Dongarra, Jack: Tools and techniques for performance - Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems), ACM Press, 2006

[Lv02] Layton, Anita T. und van de Panne, Michiel: A numerically efficient and stable algorithm for animating water waves. In: *The Visual Computer* 18 (2002), S. 41–53

[Pro99] Prokop, H.: *Cache-Oblivious Algorithms*, MIT Department of Electical Engineering and Computer Science, Master's thesis, June 1999

[SG06] Strzodka, Robert und Göddeke, Dominik: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, 2006, S. 259–270

[The05] The C++ Standards Committee. Draft Technical Report on C++ Library Extensions. `http://www.open-std.org/JTC1/SC22/WG21`. June 2005

[Wil00] Wilkes, Maurice: The Memory Gap (keynote). (2000). – `http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf`