

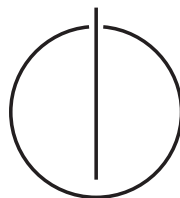
TECHNISCHE UNIVERSITÄT MÜNCHEN

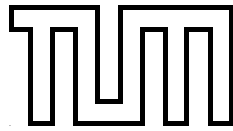
FAKULTÄT FÜR INFORMATIK

Bachelorarbeit in Informatik

**Shared-Memory-Parallelisierung
eines Multiskalenlösers
für Partielle Differentialgleichungen**

Svetlana Nogina





TECHNISCHE UNIVERSITÄT MÜNCHEN

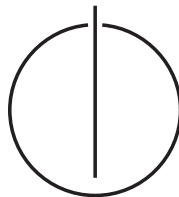
FAKULTÄT FÜR INFORMATIK

Bachelorarbeit in Informatik

**Shared-Memory-Parallelisierung
eines Multiskalenlösers
für Partielle Differentialgleichungen**

**Shared Memory Parallelisation
of a Multiskale Solver
for Partial Differential Equations**

Bearbeiter: Svetlana Nogina
Aufgabensteller: Univ.-Prof. Dr. Hans-Joachim Bungartz
Betreuer: Dr. Tobias Weinzierl, Dipl.-Inf. Kristof Unterwiesing
Abgabedatum: 4. Oktober 2010



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 4. Oktober 2010

.....
(Unterschrift des Kandidaten)

Inhaltsverzeichnis

1	Einleitung	1
2	Peano Framework	2
2.1	Funktionsweise	2
2.1.1	Gitter	2
2.1.2	Spacetree	3
2.1.3	Peano Ereignisse	3
3	Parallelisierung der Gittertraversierung	5
3.1	Rekursioneliminierung	5
3.2	OpenMP Ansatz	5
3.2.1	Erstellen von Task-Graphen	6
3.2.2	Ansätze	7
4	Abhängigkeiten	9
4.1	Abhängigkeiten bei der Gittertraversierung	10
4.2	Aufbau eines Datenabhängigkeitsgraphen	10
4.3	Implementierung	11
4.3.1	Datenstruktur	11
4.3.2	Besonderheiten der Implementierung	12
4.4	Laufzeit und der Speicherverbrauch	13
5	Task-Graph	16
5.1	Ziele	17
5.2	Algorithmen zur Graphenfärbung	17
5.3	Idee	17
5.4	Algorithmus Beschreibung	18
5.4.1	Implementierung	20
6	Implementierung mit OpenMP	23
6.1	Shared Workqueue	23
6.1.1	Busy Waiting	24
6.2	Parallele For-Schleife	24
6.2.1	Optimierungen	25
7	Experimente und Vergleich der Ansätze	26
7.1	Vergleich der OpenMP-Ansätze	26
7.1.1	Konstante Bearbeitungszeiten	26
7.1.2	Variable Bearbeitungszeiten	28
7.2	Shared Workqueue	28

Inhaltsverzeichnis

7.3	Parallele For-Schleife	29
8	Integration in Peano Framework	31
8.1	Identifizieren von regulären Gitterstrukturen	31
8.2	Traversierung	31
9	Ergebnisse und weitere Arbeitsrichtungen	33
	Literaturverzeichnis	35

1 Einleitung

Heute ist es besonders wichtig, Berechnungen für Mehrkernarchitekturen auszulegen. In dieser Arbeit wird ein Konzept untersucht, das die parallele Traversierung von Mehrgittern ermöglicht, die beim Lösen von partiellen Differentialgleichungen (*PDE*) Einsatz finden. Dabei wird versucht, die komplette Abarbeitung dieser Gitter parallel zu gestalten. Die wichtigsten Ziele sind Skalierbarkeit, verbesserte Lastverteilung und geringer Aufwand. Für die Umsetzung wird das Shared-Memory-Programmiermodell verwendet, und die Implementierung erfolgt mit der C++-Erweiterung OpenMP, mit der man mit einfachen Mitteln und minimalen Veränderungen ein Programm parallelisieren kann.

Die Zielsetzung dieser Arbeit ist die Parallelisierung der Gittertraversierung im Peano-Framework, das in den letzten Jahren am Lehrstuhl für Wissenschaftliches Rechnen der Technischen Universität München entstanden ist. Peano ist ein Framework zum Lösen von partiellen Differentialgleichungen, das besonderen Wert auf die Unterstützung von Multiskalenalgorithmen, dynamische Adaptivität, Erweiterbarkeit und Parallelität, jedoch bisher nur auf Distributed-Memory-Maschinen mittels einer Domain Decomposition, legt [Wei09]. Diese Arbeit soll Anhaltspunkte für die weitere Parallelisierung des Frameworks mittels Shared-Memory-Techniken geben.

Die Arbeit ist wie folgt aufgebaut: Kapitel 2 gibt einen Überblick über einige Peano Konzepte, wie Spacetrees, kartesische Gitter und Ereignissenmechanismen. Im zweiten Kapitel wird die Vorgehensweise bei der Parallelisierung betrachtet. In den nächsten drei Kapiteln werden die einzelnen Schritte dieser Parallelisierung ausführlich beschrieben. Kapitel 3 beschäftigt sich mit dem Modellieren der Abhängigkeiten. Im vierten Kapitel wird aus den Abhängigkeiten der für die parallele Ausführung notwendige Task-Graph hergeleitet. Die Parallelisierungsmethoden werden im fünften Kapitel beschrieben. Im darauf folgenden Kapitel wird der Vergleich der Parallelisierungsansätze aus dem fünften Kapitel durchgeführt. Die letzten zwei Kapiteln beschäftigen sich mit der Integration des Ansatzes in das Peano-Framework, Analyse der Ergebnisse und der weiteren Arbeitsrichtungen.

2 Peano Framework

Mit dem Peano-Framework ist dem Entwickler eine C++-Umgebung zum Lösen von partiellen Differentialgleichungen zur Verfügung gestellt worden, die adaptiv auf Mehrgittern funktioniert, cache-effizient ist und gute Voraussetzungen für die Skalierung auf Mehrkernarchitekturen mitbringt. Wichtig ist vor allem die Möglichkeit, einfach zusätzliche Komponenten im Framework zu implementieren, so dass das gesamte System dadurch erweitert werden kann, ohne dass der Erweiternde sich selbst um Parallelisierung und Gitterverwaltung kümmern muss. In folgenden Abschnitten möchte ich einige grundlegende Konzepte vom Peano erläutern, die für meine Arbeit relevant sind.

2.1 Funktionsweise

Heutige numerische Verfahren auf Mehrgittern benötigen dynamische Adaptivität, um präzise Ergebnisse zu erzielen. Viele Implementierungen haben einen hohen Speicherverbrauch und komplexe Datenstrukturen, die zur Verwaltung der Rechengitter notwendig sind. Peano benutzt für die Verwaltung von adaptiven Gittern die *Spacetrees*. Die Traversierung derselben erfolgt mithilfe von raumfüllenden Kurven, was die effiziente Abspeicherung von Gitterinformationen ermöglicht.

2.1.1 Gitter

Um auf einem kontinuierlichen Gebiet numerische Verfahren zum Lösen von PDE anzuwenden, wird dieses Gebiets diskretisiert. Das daraus resultierende Rechengitter kann in Abhängigkeit von der Zerlegungsmethode strukturiert oder unstrukturiert sein. In Peano werden alle Berechnungen auf einem kartesischen Gitter durchgeführt, in dem die Elementenkanten parallel zu den Koordinatenachsen verlaufen und alle Seiten eines Elements gleiche Länge haben. Gitterelemente sind ein Quadrat für die zweite Dimension, ein Würfel für die

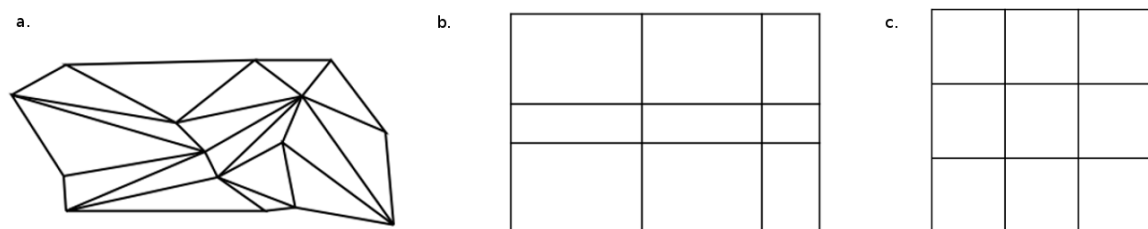


Abbildung 2.1: a.) Ein unstrukturiertes Dreieck-Gitter. b.) Ein rechwinkliges ungleichmäßiges Gitter. c.) Ein kartesisches Gitter.

dritte Dimension und ein Hyperwürfel für größere Dimensionen. Der einfachste Fall eines kartesischen Gitters ist ein reguläres Gitter, in dem alle Elemente gleich groß sind. Jedes

Gitterelement wird durch die kartesischen Koordinaten eines Eckpunktes und die Schrittweite identifiziert. In den Fällen, wo eine unregelmäßige Verfeinerung der Rechendomäne erwünscht ist, kommt ein *adaptives kartesisches Gitter* zum Einsatz. Ein adaptives Gitter wird rekursiv erstellt, indem jedes zu verfeinernde Element durch ein reguläres kartesisches Gitter mit kleinerer Schrittweite ersetzt wird [Wei09].

2.1.2 Spacetree

Ein *k-Spacetree* ist eine Baumstruktur, mit der die Mehrgitterrechendomäne dargestellt wird. Die Anzahl der Kinder ist durch k festgelegt: Jedes Volumenelement aus den Gittern wird auf einen Knoten des k -Spacetrees abgebildet. Dabei wird der Hyperwürfel, der das gesamte Rechengebiet umfasst, zum Wurzelknoten. Falls ein Gitterelement e verfeinert wird, definiert man die von ihm enthaltenen Hyperwürfel $\{e_1, \dots, e_n\}$ als seine *Kinderknoten* und den Würfel e selbst als den *Vaterknoten*. Ich verwende für diese Relation die Schreibweise aus [Wei09] : $\forall 1 \leq i \leq n, e_i \sqsubseteq_{child} e$. Jedem Paar $e_i \sqsubseteq_{child} e$ entspricht in dem k -Spacetree eine gerichtete Kante vom Vaterknoten zum Kind. Die Anzahl der Kinderknoten in einem k -Spacetree der Dimension d ist $n = k^d$. Jedem Element wird ein Level zugewiesen. Dieser Wert bedeutet die Anzahl der Verfeinerungen, die man braucht, um den Hyperwürfel zu erzeugen. Falls ein Element den Level l hat, erhalten seine Kinderknoten den Level $l + 1$. Die Nummerierung der Levels beginnt vom Wurzelknoten, der den Level 0 besitzt [Wei09].

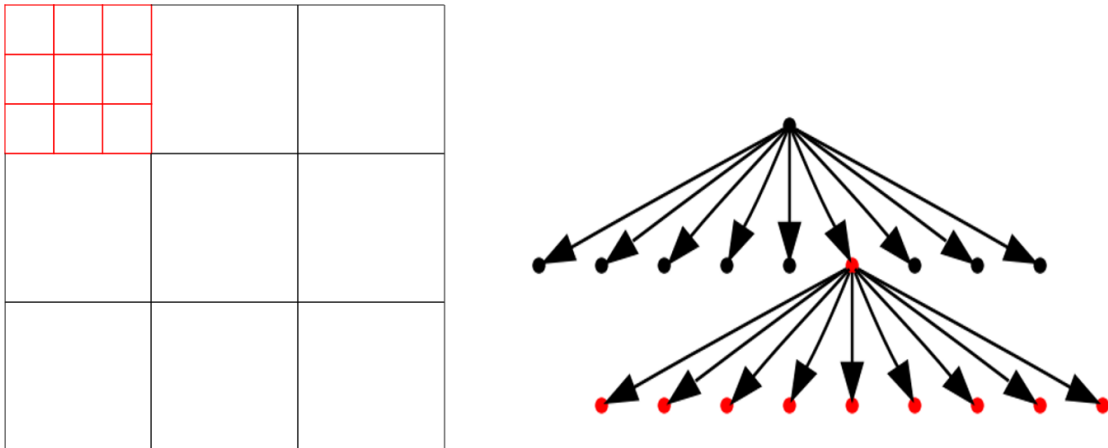


Abbildung 2.2: Ein adaptiv verfeinertes Gitter und dazugehöriger ($k=3$)-Spacetree.

2.1.3 Peano Ereignisse

Im Peano-Framework wird ein Adapter-Mechanismus verwendet, um die Einführung zusätzlicher Komponenten zu erleichtern. Die Gittertraversierung findet in der Basiskomponente *Grid* statt, wobei verschiedene Ereignisse erzeugt werden. Beim Ereignisaufruf wird eine entsprechende Methode im Adapter gestartet, in dem die Bearbeitung stattfindet. Die Methodenparameter unterscheiden sich in der Abhängigkeit vom Ereignistyp und der benötigten Funktionalität. Die wichtigsten Klassen sind die Ereignisse für die Gitterverwaltung und

Ereignis	Beschreibung
<code>beginTraversal()</code>	Iteration starten.
<code>endTraversal()</code>	Iteration beenden.
<code>enterElement()</code>	Alle Knoten einer Zelle sind geladen.
<code>leaveElement()</code>	Die abgearbeitete Zelle verlassen.
<code>touchVertexFirstTime()</code>	Zum ersten mal wurde eine der Nachbarzellen eines Knotens besucht.
<code>touchVertexLastTime()</code>	Die letzte aus den zum Knoten benachbarten Zellen wurde besucht.
<code>loadSubElement()</code>	Ausgelöst vor dem Übergang zum nächsten verfeinerten Level beim Abstieg (top-bottom transition).
<code>storeSubElement()</code>	Ausgelöst vor dem Übergang zum nächsten gröberen Level während des Aufstiegs (bottom-top transition).
<code>startStepsUp()</code>	Die Kindzellen sind abgearbeitet und der Aufstieg kann starten.
<code>startStepsDown()</code>	Ausgelöst vor der Traversierung der Kinderzellen.
<code>createPersistentVertex()</code>	Erstelle Randknoten bei Verfeinerung eines Knotens.
<code>destroyPersistentVertex()</code>	Zerstöre Randknoten beim Aufstieg zum gröberen Level.
<code>createTemporaryVertex()</code>	Randknoten erstellen.
<code>destroyTemporaryVertex()</code>	Zerstöre Randknoten

Tabelle 2.1: Ereignisse der Traversierungsgruppe

für die Traversierung. Die Ereignisse, die der Verwaltung dienen, werden für die Erstellung oder Verfeinerung neuer Zellen verwendet und sind vor allem für die Gittermodifikationen wichtig. Die Traversierungsereignisse sind wiederum in kleinere Gruppen unterteilt, die in der Tabelle 2.1 aufgelistet sind ¹ [But09, Wei09].

Die sequentielle Traversierung eines Spacetrees erfolgt rekursiv. Wenn ein verfeinertes Element durchlaufen wird, werden zuerst seine Kinder bearbeitet, bevor die Traversierung auf dem Level des Vaterknotens fortgesetzt wird.

¹Die Ereignismenge kann sich in den nächsten Peano-Realisierungen ändern, da die Methoden umbenannt werden oder auf einige Events verzichtet wird.

3 Parallelisierung der Gittertraversierung

In mehreren Arbeiten am Lehrstuhl für Wissenschaftliches Rechnen wurde eine Reihe von Ansätzen erforscht und implementiert, um das Peano-Framework zu parallelisieren. In einer der früheren Arbeiten wurde die Parallelisierung der Berechnungen mithilfe von Intel Threading Building Blocks (*TBB*) untersucht. Dabei wurde die Idee beschrieben, wie man in einem Spacetree Teilbäume mit einem bestimmten Muster identifiziert. Da diese Technik auch für meine Arbeit relevant ist, möchte ich sie kurz im nächsten Abschnitt ansprechen. Für die vollständige Ausarbeitung verweise ich jedoch auf [Eck09].

3.1 Rekursionseeliminierung

Um Vorteile der parallelen Traversierung zu nutzen, würde man das Wissen über die Struktur des Spacetrees benötigen. Falls der konkrete Aufbau eines Teilbaumes bekannt ist, kann man die Abhängigkeiten zwischen seinen Knoten feststellen und die sequentielle Traversierung parallelisieren. Die Rekursion-Unrolling Technik bietet die Möglichkeit, ausgewählte Teilbäume-Muster im Spacetree zu finden. Die Idee besteht darin, die Traversierung bestimmter Abschnitte im adaptiven Gitter ohne Rekursion zu erledigen, indem das gesamte Muster in einem Aufruf abgearbeitet wird. Nachdem die Rekursionseeliminierung durchgeführt wurde, ist die Struktur des Spacetrees bekannt und die rekursive Traversierung ist nicht mehr in dem ganzen Gitter notwendig. Jedes Element wird dabei durch einen Schlüsselwert gekennzeichnet, der die maximale Tiefe des Baumes mit dem Wurzel in diesem Element bedeutet.

3.2 OpenMP Ansatz

In meiner Arbeit möchte ich die Informationen über reguläre Teilbäume im Spacetree wiederverwenden, um Gitterstrukturen zu erkennen, die für die Parallelisierung geeignet sind. Die Regularität ist für die Parallelisierung besonders erwünscht, weil ein regulärer Baum einerseits die maximale Anzahl von Knoten unter den Graphen gleicher Tiefe besitzt, andererseits durch seine Dimension und Tiefe eindeutig definiert ist. In einem k -Spacetree besitzen alle verfeinerte Knoten genau k^d Kinder. Dadurch kann man die genaue Struktur eines regulären Baumes anhand seiner Tiefe bestimmen. Das weitere Vorgehen besteht darin, die Datenabhängigkeiten zwischen den Baumknoten zu bestimmen, um bei der parallelen Traversierung Probleme wie Lost Updates und Race Conditions zu vermeiden. Mithilfe von diesen Datenabhängigkeiten wird anschließend ein *Task-Graph* für entsprechenden Spacetree erstellt. Für die eigentliche Traversierung werden Parallelisierungs-Mechanismen von OpenMP eingesetzt. An die Implementierung werden dabei folgende Anforderungen gestellt:

- **Skalierbarkeit** - es soll ermöglicht werden, die Mehrkernarchitekturen effizient für die Traversierung zu nutzen.

- **Lastbalancierung** - die einzelnen Programmteile sollen gleichmäßig auf Prozessore verteilt werden. Ein wichtiger Punkt ist die dynamische Anpassung der Verteilung während des Programmablaufs.
- **Minimaler Overhead** - die Leistung sollte bereits bei kleineren Problemgrößen verbessert werden, da sie in der realen Anwendung viel häufiger auftreten.

Im nächsten Abschnitt möchte ich die untersuchten Konzepte zuerst abstrakt betrachten. In den folgenden drei Kapiteln werde ich dann noch mal ausführlich auf die einzelnen Schritte der Parallelisierung eingehen und dabei die konkrete Implementierung dieser Konzepte beschreiben.

3.2.1 Erstellen von Task-Graphen

Ein Programm besteht immer aus einer Reihe von Instruktionen, die Zwischenergebnisse der Berechnungen untereinander austauschen oder gemeinsame Daten benutzen können. Bei sequentieller Ausführung werden die Abhängigkeiten zwischen den einzelnen Aufgaben meistens automatisch eingehalten, da der Programmfluß deterministisch ist. Im Allgemeinen muss bei Parallelisierung die Synchronisation der Datenzugriffe vorgenommen werden und die Abarbeitung der Instruktionen in einer zulässigen Reihenfolge gewährleistet sein. Für die Lösung dieses Problems existieren mehrere Ansätze, wie zum Beispiel die Verwendung von Locks für gemeinsame Daten oder das Einführen von atomaren, d.h. nicht unterbrechbaren Operationen. In Peano erfolgen die Datenzugriffe innerhalb der Ereignisse und sind von dem Entwickler eines Löser-Plugins verdeckt. Aus diesem Grund muss die Synchronisation erst auf der Ereignissebene stattfinden. Ereignisse interpretiert man dabei als einzelne Aufgaben. Für die Verwaltung mehrerer voneinander abhängigen Aufgaben ist das *Task-Graph-Pattern* sehr gut geeignet [GMBM08]. Eine wichtige Voraussetzung für die Implementierung dieses Patterns ist das Wissen über alle Tasks und deren Abhängigkeiten. Die Programmausführung wird als eine Menge von Aufgaben dargestellt, die die Knotenmenge des Task- Graphen bilden. Die Abhängigkeiten werden durch gerichtete Kanten repräsentiert. Der Aufbau eines Task-Graphen gliedert sich in drei Schritte:

1. **Task Decomposition** - dabei wird das Problem in einzelne kleine Stücke zerlegt. Diese Instruktionengruppen sollten so zusammengestellt werden, damit die Anzahl der Abhängigkeiten von den anderen Tasks minimiert wird.
2. **Group Tasks** - kleine Instruktionengruppen werden zu den endgültigen Tasks vereinigt.
3. **Order Tasks** - die gültige Reihenfolge der Tasksabarbeitung wird festgelegt, indem alle Abhängigkeiten berücksichtigt werden.

Der erste und der zweite Schritt werden im Kapitel 4 beschrieben, wo es um den Aufbau eines Datenabhängigkeitsgraphen geht. Der dritte Schritt kommt im Kapitel 5 zur Betracht, in dem die Tasks anhand der Datenabhängigkeiten geordnet werden.

3.2.2 Ansätze

Die Abarbeitung der vorhandenen Tasks kann auf unterschiedliche Weisen erfolgen. Für rekursive Probleme, bei denen jede Aufgabe in Teilaufgaben zerlegt wird, ist z.B. das *Teile und Hersche* Pattern gut geeignet. Das *Fork/Join* Pattern sieht so genannte Parent Threads vor, die bei Bedarf andere Threads erzeugen können, denen verbliebene Tasks zugewiesen werden. Ein weiteres Pattern ist *Loop Parallelism*, mit dem die unabhängigen Tasks in einer Schleife durchlaufen werden, dabei werden einzelne Iterationen an unterschiedliche Threads verteilt. Dieses Pattern ist besonders für datenparallele Probleme bevorzuglich, die auf unterschiedlichen Daten gleiche Operationen durchführen. Für die Abarbeitung der Graphen wird häufig das *Shared Queue Design* Pattern eingesetzt, bei dem alle Tasks in einer Queue abgelegt sind, die in dem nächsten Schritt ausgeführt werden können.

Für die Traversierung des Task-Graphen wurden zwei Methoden ausgewählt: das Loop Parallelism Pattern und das Shared Queue Pattern. Die für die Gittertraversierung notwendigen Tasks unterscheiden sich grundsätzlich durch die zur Bearbeitung vorliegenden Knoten. Das entspricht sehr gut den Anforderungen des Schleifen-Patterns, weil alle Knoten somit in einer Schleife durchlaufen werden könnten. Die Parallelisierung der Schleifen erfordert außerdem weniger Aufwand, als drei weitere beschriebene Ansätze. Mit dem Shared Queue Ansatz wird versucht, eine möglichst optimale Lastbalancierung zu erreichen, da dieser Ansatz die dynamische Verteilung von Tasks bietet.

Shared Workqueue

Der erste Ansatz besteht darin, eine Queue mit Knoten zu erstellen, die als nächste traversiert werden können, weil alle seine Vorgänger bereits bearbeitet wurden. Diese Queue wird von mehreren Threads parallel durchlaufen. Nach dem Ausführen eines Knotens werden die Zahlen der Vorgänger bei allen Kindknoten um eins verkleinert. Falls einige Kindknoten keine Vorgänger mehr haben, die noch nicht traversiert wurden, werden sie in die Queue eingefügt.

Parallele For-Schleife

In dem zweiten Ansatz wird die Traversierung der Task-Graphen mithilfe zweier geschachtelten For-Schleifen durchgeführt. Dafür wird für jeden Knoten der maximale gerichtete Pfad berechnet, der ihn mit der Wurzel verbindet. Da der Task-Graph azyklisch sein muss, ist der maximale Pfad für jeden Knoten endlich. Die Kanten sind nicht gewichtet, so dass die Länge des maximalen Pfades gleich der Anzahl der Kanten im Pfad ist. Für zwei Knoten v und w gilt dabei, dass wenn v von w abhängig ist, die Ungleichung $\text{maximalPath}(v) \geq \text{maximalPath}(w) + 1$ gelten muss. Allgemein gilt für zwei Knoten v und w folgendes: falls ein gerichteter Pfad p_{w-v} der Länge n existiert, gilt für v und w $\text{maximalPath}(v) \geq \text{maximalPath}(w) + n$. Aus der Aussage kann man folgern, dass falls $\text{maximalPath}(v) = \text{maximalPath}(w)$ gilt, dann gibt es im Task-Graph keinen gerichteten Pfad von v nach w oder von w nach v und die Knoten v und w sind unabhängig voneinander. In diesem Fall können v und w parallel bearbeitet werden. Auf der Basis dieser Eigenschaft kann die Traversierung des Task-Graphen parallelisiert werden. Die Idee ist dabei, die Knoten in Gruppen mit gleichen Pfadlängen einzuteilen, so dass die einzelnen Gruppen in einer For-Schleife parallel durchlaufen werden. Man muss allerdings beachten, dass die Abarbeitung mehreren Gruppen nicht parallel geschehen kann, weil Datenabhängigkeiten zwischen den

3 Parallelisierung der Gittertraversierung

Knoten unterschiedlicher Gruppen existieren, d.h. die Knoten mit der Pfadlänge n dürfen erst bearbeitet werden, wenn die Knoten mit Pfadlängen 0 bis $n-1$ fertig sind.

4 Abhängigkeiten

In diesem Kapitel möchte ich das Modellieren der Datenabhängigkeiten für die Gittertraversierung beschreiben. Der erste Schritt ist dabei, die einzelnen Operationen zu identifizieren, zwischen denen im nächsten Schritt die Abhängigkeiten erfasst werden. Da die Traversierung in Peano ereignisbasiert ist, werde ich die Ereignisaufufe als atomare Tasks betrachten. Die Abarbeitung der Zellen geschieht mithilfe von Traversierungsereignissen, die in der Tabelle 2.1 aufgelistet sind.

Das Peano-Framework arbeitet sowohl auf den Knoten (Vertices), als auch auf den Zellen (Cell) des kartesischen Gitters. Die erste Art der Datenabhängigkeiten ist in diesem Fall die Abhängigkeit zwischen einer Zelle und deren Kindzellen. Die zweite Art der Abhängigkeiten besteht zwischen Nachbarzellen auf einem Level.

Die erste Abhängigkeitsart entsteht, weil die Kindzellen erst bearbeitet werden können, wenn auf der Vaterzelle `startStepDown()` aufgerufen wird. Umgekehrt darf auf der oberen Zelle das Ereignis `startStepsUp()` erst aufgerufen werden, wenn die unteren Zellen bearbeitet wurden. Weiterhin muss die Reihenfolge der Eventaufrufe für eine Zelle erhalten bleiben. Abhängigkeiten, die das Fertigstellen eines Prozesses für die Ausführung eines anderen erfordern, werden als *kausale Abhängigkeiten* bezeichnet. Die *kausalen Abhängigkeiten* definieren eine transitive Relation $t_1 \rightarrow t_2$ und werden durch gerichtete Kanten in einem Graphen dargestellt. Im Peano-Framework sind es die Abhängigkeiten zwischen den Ereignissen einer Zelle. Auf einem verfeinerten Knoten sieht die korrekte Abfolge der Events folgendermaßen aus :

```
enterElement() → loadSubElement() → startStepsDown() →  
→ startStepsUp() → storeSubElement() → leaveElement()
```

Zwischen den Ereignissen eines Vaterknotens und seiner Kinder bestehen Abhängigkeiten für den Übergang von einem Level auf den anderen. Für alle v_i und v_j mit $v_i \sqsubseteq_{child} v_j$ gilt:

$$I.startStepsDown(v_j) \rightarrow enterElement(v_i)$$

$$II.leaveElement(v_i) \rightarrow startStepsUp(v_j)$$

Bei der Bearbeitung der Zellen eines Gitters müssen die Datenabhängigkeiten zwischen den adjazenten Zellen berücksichtigt werden, die die zweite Art der Abhängigkeiten darstellen. Bei der Berechnung einer Zelle werden die Knoten geladen und nach der Bearbeitung wieder gespeichert. Falls adjazente Zellen parallel bearbeitet würden, könnte es Lost Updates und die unberechenbaren Werten in den Knoten als Folge haben. Diese Art der Abhängigkeiten basiert auf einem gemeinsamen Datenzugriff und nennt sich *Datenzugriffsabhängigkeit*. In unserem Fall hat diese Abhängigkeit keine Richtung, weil die Elementenbearbeitung in beliebiger Reihenfolge geschehen kann. Bezeichnen wir die Relation zweier Elemente, zwischen denen eine ungerichtete Datenabhängigkeit besteht, als $t_1 \leftrightarrow t_2$

Beim Feststellen der Abhängigkeiten ist es wichtig, sich klar zu machen, welche Teile des Problems als einzelne Aufgaben betrachtet werden und welche zu einer größeren

Aufgabe zusammengefasst werden. Die Aufteilung des Problems in kleinere Untermengen erhöht die Granularität und bietet mehr Raum für die Parallelisierung der Lösung. Allerdings steigt mit der Anzahl der einzelnen Aufgaben der Kontrollaufwand und die Kosten für die Abhängigkeitsanalyse. In folgenden Abschnitten werde ich detailliert auf einzelne Abhängigkeiten eingehen und den konkreten Aufbau eines Abhängigkeitsgraphen für reguläre kartesische Mehrgitter diskutieren.

4.1 Abhängigkeiten bei der Gittertraversierung

Die Reihenfolge der Zellenbearbeitung bei der Traversierung eines Gitterlevels ist nicht relevant für Ergebnisse. Diese Tatsache ermöglicht es erst, das Gitter parallel zu bearbeiten, wobei die unterschiedlichen nicht adjazenten Zellen gleichzeitig berechnet werden können, und die Reihenfolge sich dabei komplett von der Reihenfolge der sequentiellen Traversierung unterscheiden kann. Anders sieht es aber mit der Reihenfolge der Ereignisaufufe einer Zelle aus. Die Abfolge der Events wird von der hierarchischen rekursiven Struktur des Spacetrees vorgeschrieben. Es ist zum Beispiel nicht möglich, auf einem verfeinerten Knoten nach `startStepsDown()` sofort `startStepsUp()` aufzurufen, weil die Abhängigkeit von dem Ereignis `leaveElement()` seiner Kinder dadurch verletzt würde. Aus diesem Grund darf man nicht alle Ereignisse einer Zelle zu einer Aufgabe zusammenfassen. Die naheliegende Lösung ist in diesem Fall die Ereignisse `enterElement()` bis zum `startStepsDown()` in einer Gruppe und die Ereignisse ab `startStepsUp()` in einer anderen Gruppe zu vereinigen. Durch diese Zerlegung werden zwei disjunkte Mengen von Tasks erstellt: $D = \{d_0, \dots, d_{n-1}\}$ und $U = \{u_0, \dots, u_{n-1}\}$. Die Menge D beinhaltet alle Knoten, die den Ereignissen erster Gruppe entsprechen. Die Menge U enthält die Tasks für die zweite Gruppe der Ereignisse. Da für jede Zelle genau zwei Ereignisgruppen existieren, sind die Mengen U und D gleich mächtig. $|U| = |D| = n$, wobei n die Anzahl der Knoten im Spacetree ist. Seien die Elemente in den Untermengen U bzw. D mit den Indexen von 0 bis $n - 1$ nummeriert. Dann kann die Nummerierung der einzelnen Tasks so gewählt werden, so dass die Tasks u_{n-i} und d_i den Ereignisgruppen der Zelle i entsprechen.

4.2 Aufbau eines Datenabhängigkeitsgraphen

Die Datenabhängigkeiten können auf natürliche Weise mithilfe von gerichteten Graphen dargestellt werden. Sei $G_{dep} = \{V, E_{dep}\}$ ein gerichteter Graph. Die Knotenmenge V entspricht der Menge der Tasks, die aus Ereignisaufrufen gebildet werden, und setzt sich aus zwei disjunkten Mengen D und U zusammen. Die Kantenmenge besteht aus den Tupeln $\{v_1, v_2\}$, wobei v_2 nach der v_1 bearbeitet wird. Die Kanten werden ausgehend von den vorhandenen Abhängigkeiten im Gitter erstellt.

1. Die erste Quelle der Datenabhängigkeiten ist die strenge Reihenfolge bei den Ereignissen eines Elements. Demzufolge muss der Knoten u_{n-i} nach dem Knoten d_i bearbeitet werden, d.h.

$$\forall 0 \leq i < n, \quad E_{dep} = E_{dep} \cup \{d_i, u_{n-i}\}$$

2. Beim Abstieg im Spacetree müssen die Kindzellen nach der Vaterzelle traversiert werden. Daraus folgt, dass die Knoten aus der Menge D , die den Kindzellen entsprechen,

4 Abhängigkeiten

nach dem Knoten für die Vaterzelle bearbeitet werden müssen.

$$\forall c, \quad c \sqsubseteq_{\text{child}} i, \quad E_{\text{dep}} = E_{\text{dep}} \cup \{d_i, d_c\}$$

3. Für den Aufstieg im Spacetree gilt, dass die Vaterzelle nach den Kindzellen traversiert wird, d.h. die Kindknoten aus U werden vor dem Vaterknoten aus U bearbeitet.

$$\forall c, \quad c \sqsubseteq_{\text{child}} i, \quad E_{\text{dep}} = E_{\text{dep}} \cup \{u_{n-c}, u_{n-i}\}$$

4. Für die Traversierung der Zellen eines Levels ist keine Reihenfolge vorgegeben. Um die Datenabhängigkeiten zwischen den adjazenten Zellen i und j darzustellen, werden ungerichtete Kanten $\{d_i, d_j\}$ und $\{u_{n-i}, u_{n-j}\}$ zwischen den entsprechenden Knoten aus den Mengen D bzw. U benötigt. Da G_{dep} ein gerichteter Graph ist, ersetzt man jede ungerichtete Kante durch zwei gerichtete Kanten, d.h.

$$\forall j \in \Gamma(i), \quad E_{\text{dep}} = E_{\text{dep}} \cup \{\{d_i, d_j\}, \{d_j, d_i\}, \{u_{n-i}, u_{n-j}\}, \{u_{n-j}, u_{n-i}\}\}$$

Algorithm 1 Konstruktion eines Abhängigkeitsgraphen für ein Mehrgitter

Require: $V = D \cup U = \{d_0, \dots, d_{n-1}\} \cup \{u_0, \dots, u_{n-1}\}$

Ensure: $G_{\text{dep}} = (V, E_{\text{dep}})$

for $i = 0$ to $n - 1$ **do**

$E_{\text{dep}} = E_{\text{dep}} \cup \{d_i, u_{n-i}\}$

for all $c \sqsubseteq_{\text{child}} i$ **do**

$E_{\text{dep}} = E_{\text{dep}} \cup \{d_i, d_c\}$

$E_{\text{dep}} = E_{\text{dep}} \cup \{u_{n-c}, u_{n-i}\}$

end for

for all $j \in \Gamma(i)$ **do**

$E_{\text{dep}} = E_{\text{dep}} \cup \{\{d_i, d_j\}, \{d_j, d_i\}\}$

$E_{\text{dep}} = E_{\text{dep}} \cup \{\{u_{n-i}, u_{n-j}\}, \{u_{n-j}, u_{n-i}\}\}$

end for

end for

Ein Datenabhängigkeitsgraph ist auf der Abbildung 4.1 veranschaulicht. Die schwarzen Knoten repräsentieren die Ereignisse beim Abstieg, die blauen Knoten sind die Aufstiegsergebnisse.

4.3 Implementierung

4.3.1 Datenstruktur

Für die Darstellung des Abhängigkeitsgraphen wurde die Adjazenzliste ausgewählt. Für die Implementierung standen zwei Datenstrukturen zur Auswahl: ein zweidimensionales Array von Typ `int` oder ein `Vector` von `Sets`. Die erste Variante benötigt weniger Speicherplatz, als die zweite, allerdings erfolgen die Zugriffe auf die Nachfolger eines Knotens im Integer-Array langsamer, weil das ganze Feld im schlimmsten Fall durchlaufen werden muss. Der

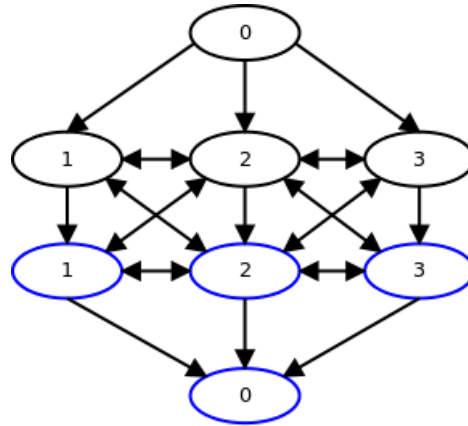


Abbildung 4.1: Abhängigkeitsgraph für einen eindimensionalen Spacetree ersten Levels.

Test, ob ein Element enthalten ist, läuft in einem `stl::vector` in konstanter Zeit $O(1)$ und für den `stl::set` in logarithmischer Zeit $O(\log(n))$, wobei n die Anzahl der Nachfolger eines Elements ist [C++10]. Der maximale Grad eines Knotens im Datenabhängigkeitsgraph ist durch die maximale Anzahl der Kinder und Nachbarzellen beschränkt. Ausgehend von der Kinderanzahl k^d und der Anzahl der adjazenten Zellen $k^d - 1$ berechnet sich der maximale Knotengrad zu $k^d + (k^d - 1) = 2k^d - 1$. Der maximale Aufwand für die Überprüfung, ob eine Kante $\{v_i, v_j\}$ existiert, setzt sich demzufolge aus dem Aufwand $O(1)$ für das Finden des Startknotens v_i und dem Aufwand $O(\log(2k^d - 1))$ für den Test im Set, ob dieser Startknoten einen Nachfolger v_j besitzt, zum Wert $O(\log(k^d))$ zusammen.

4.3.2 Besonderheiten der Implementierung

Um die Abhängigkeiten zwischen einzelnen Tasks zu bestimmen, braucht man die Informationen über die Kinder und Nachbarn der zum Task korrespondierenden Gitterzelle. Da es nicht möglich ist, direkt aus der Nummer einer Zelle die Werte ihrer Kinder und Nachbarn abzulesen, müssen aus dem Knotenwert die kartesischen Koordinaten und der Level berechnet werden, die innerhalb der Mehrgittern die Zelle eindeutig identifizieren. Anhand von Koordinaten werden im nächsten Schritt Kanten hinzugefügt, die Datenabhängigkeiten widerspiegeln.

Definition der Nachbarschaftsrelation durch kartesische Koordinaten:

In einem kartesischen Gitter der Dimension d hat eine Zelle maximal $3^d - 1$ Nachbarn. Als Nachbarzellen werden solche bezeichnet, die einen oder mehr gemeinsame Knoten besitzen. Grund dafür ist die Tatsache, dass die Berechnungen in Peano sowohl auf den Zellen als auch auf den Gitterknoten stattfinden. Die Abbildung 4.2 veranschaulicht die Nachbarschaftsmenge einer Zelle in einem zweidimensionalen Gitter.

Die Nachbarschaftsrelation Γ wird über den Abstand zwischen den Koordinaten definiert. Dabei gilt für die Nachbarschaftsmenge von $y = (y_1, \dots, y_d)$:

$$\Gamma(y) = \{x = (x_1, \dots, x_d) \mid x \neq y \quad \forall i \quad |x_i - y_i| < 2\}$$

Da aus $x \in \Gamma(y)$ unmittelbar $y \in \Gamma(x)$ folgt, werden für jedes Paar der benachbarten Zellen

zwei gerichtete Kanten hinzugefügt.

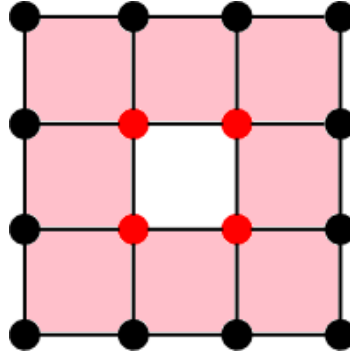


Abbildung 4.2: Die Nachbarzellen der mittleren Zelle sind rosa markiert. Die gemeinsamen Knoten sind rot gekennzeichnet.

Transitivität der kausalen Abhängigkeiten:

Aufgrund der festen Reihenfolge der Ereignissen müssen die Tasks aus der Menge D vor den Tasks aus U auf dem gleichen Element aufgerufen werden, so dass eine kausale Abhängigkeit zwischen den Tasks d_i und u_{n-i} besteht. Die Kanten zwischen den Knoten d_i und u_{n-i} aus D bzw. U werden allerdings nur hinzugefügt, falls das Element i ein Blatt im Spacetree ist. Ansonsten würde es bereits einen gerichteten Pfad von d_i nach u_{n-i} geben, so dass die Kante $\{d_i, u_{n-i}\}$ durch die Transitivität von \rightarrow überflüssig wäre.

4.4 Laufzeit und der Speicherverbrauch

Um den Speicherverbrauch genau zu ermitteln, muss man zuerst die Entscheidung über die benutzte Datenstruktur und die Architektur treffen. Man kann allerdings die Anzahl der Knoten und der Kanten im Graph berechnen. Für einen Datenabhängigkeitsgraph $G = (V, E)$ aus dem Spacetree der Dimension d und der Tiefe $level$ gilt:

$$|V| = 2 \cdot \sum_{i=0}^{level} 3^{d \cdot i}$$

Kantenmenge E setzt sich aus den Kanten zwischen einem Vaterknoten und seinen Kindknoten (E_{child}), Kanten zwischen Ereignissen einer Zelle (E_{event}), Kanten zwischen Ereignissen der Nachbarzellen auf einem Level (E_{adj}) und Kanten zwischen den Ereignissen unterschiedlicher benachbarten Zellen ($E_{adj-event}$) zusammen.

$$|E_{child}| = 2 \cdot 3^d \cdot \sum_{i=0}^{level-1} 3^{d \cdot i}$$

$$|E_{event}| = 3^{d \cdot level}$$

Um die Anzahl der Nachbarn einer Zelle in einem Gitter mit Dimension d zu berechnen, muss zwischen inneren Elementen und *Grenzelementen* unterschieden werden. Jedes Element

4 Abhängigkeiten

<i>level</i>	<i>dim2</i>	<i>dim3</i>	<i>dim4</i>
1	123	765	5043
2	2213	50501	1176773
3	24559	1850197	141717655
4	236409	55182525	13100551017
5	2175395	1539714221	1108742493179
6	19723405	42028788277	91129254151309
7	177947031	1138911324933	7417468219497951

Tabelle 4.1: Die Anzahl der Abhängigkeiten in einem Graphen.

wird durch seine Koordinaten (x_0, \dots, x_{d-1}) identifiziert. Elemente, deren $d - k$ Koordinaten Werte aus $\{0, d - 1\}$ haben, werden *Grenzelemente* der Dimension k genannt. Während die inneren Zellen $3^d - 1$ Nachbarn haben, liegt diese Anzahl für eine Grenzzelle der Dimension k bei $2^{n-k}3^k - 1$. Dann muss man nur noch die Anzahl der Grenzelemente mit Dimensionen $0, \dots, d - 1$ berechnen. In einem 3-Spacetree können Zellenkoordinaten 3^{level} Werte annehmen. Die Anzahl der Grenzelemente mit Dimension k ist:

$$\binom{d}{k} \cdot 2^{n-k} \cdot (3^{level} - 2)^k$$

$|E_{adj}|$ und $|E_{adj-event}|$ können jetzt wie folgt berechnet werden:

$$|E_{adj}| = 2 \cdot \sum_{i=0}^{level} \left(\sum_{k=0}^{d-1} \left(\binom{d}{k} \cdot 2^{n-k} \cdot (3^i - 2)^k \cdot (2^{n-k}3^k - 1) \right) + (3^i - 2)^d \cdot (3^d - 1) \right)$$

$$|E_{adj-event}| = 2 \cdot \left(\sum_{k=0}^{d-1} \left(\binom{d}{k} \cdot 2^{n-k} \cdot (3^{level} - 2)^k \cdot (2^{n-k}3^k - 1) \right) + (3^{level} - 2)^d \cdot (3^d - 1) \right)$$

Insgesamt gibt es in dem Abhängigkeitsgraph $|E_{child}| + |E_{event}| + |E_{adj}| + |E_{adj-event}|$ gerichtete Kanten

In der Tabelle 4.2 sind die Größen der Datenabhängigkeitsgraphen angegeben. Der Speicherplatzbedarf wurde basierend auf der Anzahl der Kanten berechnet, wobei die Länge von **Integer** von 4 Byte angenommen wurde. Obwohl der tatsächliche Speicherverbrauch stark von der Wahl der Datenstruktur abhängt, gibt die Tabelle einen guten Richtwert für das Wachstum der Größe und Komplexität des Problems. Der hier betrachtete Datenabhängigkeitsgraph hat eine viel höhere Anzahl von Kanten, als der Task-Graph, den wir als Endergebnis für die parallele Traversierung des Gitters erhalten möchten. Da dieser Zwischenschritt aber am meisten Speicher benötigt, könnte man auf Kosten der Laufzeit die Informationen über die Abhängigkeiten in den nächsten Schritten direkt aus den Knotendaten berechnen, um die Engstelle im Speicherplatz umzugehen.

4 Abhängigkeiten

<i>level</i>	<i>dim2</i>	<i>dim3</i>	<i>dim4</i>
1	492 B	3,06 KB	20,1 KB
2	8,8 KB	202 KB	4,7 MB
3	98,2 KB	7,4 MB	566,87 MB
4	945,6 KB	220,73 MB	52,4 GB
5	8,7 MB	6,15 GB	4,4 TB
6	78,89 MB	168,1 GB	364,5 TB
7	711,78 MB	4,5 TB	29,6 PB

Tabelle 4.2: Die untere Grenze für den Speicherbedarf eines Datenabhängigkeitsgraphen.

5 Task-Graph

In einem Task-Graph stellen die Knoten einzelne Aufgaben dar, die ausgeführt werden müssen, und gerichtete Kanten zwischen den Knoten geben an, dass der Startknoten vor dem Endknoten abgeschlossen sein muss. Task-Graphen werden oft in den Scheduling Algorithmen verwendet, um die Tasks an vorhandene Prozessoren zu verteilen. Falls die Aufgaben unterschiedliche Zeit benötigen, kann man diese Information zusätzlich in dem Task-Graph speichern und für die Verbesserung der Lastbalancierung nutzen. Task-Graphen sind außerdem bei der Problemanalyse hilfreich, da ein korrekt erstellter Task-Graph die Identifikation von Deadlocks ermöglicht. Dafür müssen beim Aufbau eines Task-Graphen alle Abhängigkeiten zwischen den Aufgaben berücksichtigt werden. Ein Beispiel für einen Task-Graph, der wegen Abhängigkeiten nicht ausgeführt werden kann, ist auf der Abbildung 5.1 dargestellt.

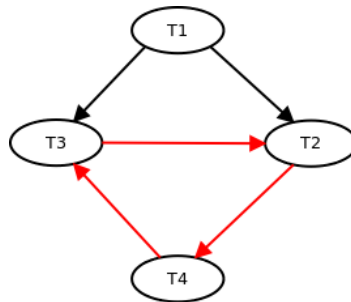


Abbildung 5.1: Task-Graph mit einer Schleife.

Somit muss ein Task-Graph azyklisch sein, weil ein Task im Graph erst ausgeführt wird, wenn alle seine Vorgänger durchlaufen wurden.

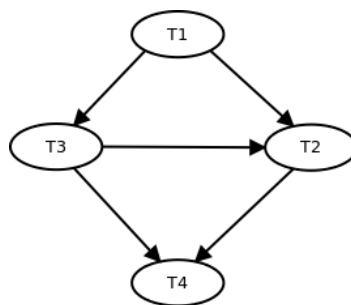


Abbildung 5.2: Task-Graph ohne Schleifen.

5.1 Ziele

Neben der grundlegenden Bedingung der Schleifenfreiheit werden an den Task-Graph weitere Anforderungen gestellt, die der Lastbalancierung dienen. Sei $G = (V, E)$ ein azyklischer Task-Graph, der eine Wurzel $v_0 \in V$ besitzt und in dem alle Knoten aus der Wurzel erreichbar sind. Bezeichnen wir den längsten gerichteten Pfad $P_i = (v_0, \dots, v_i)$ als den maximalen Pfad zum Knoten $v_i \in V$. Da im Task-Graph zuerst alle Vorgänger eines Knotens besucht werden, dauert die gesamte Ausführung mindestens so lange, bis alle Knoten in dem längsten gerichteten Pfad abgearbeitet werden. Aus diesem Grund sollte der längste Pfad unter den maximal Pfaden aller Knoten im Graph minimal sein.

5.2 Algorithmen zur Graphenfärbung

Für die Parallelisierung eines Problems muss in den meisten Fällen zuerst die Aufteilung des gesamten Problembereichs in kleinere Untermengen durchgeführt werden, die jeweils unabhängig voneinander parallel gelöst werden können. Eine effektive Methode zum Partitionieren eines Graphen in unabhängige Mengen stellt die Graphenfärbung dar. Im Fall einer Knotenfärbungen eines ungerichteten Graphen werden die Knoten so partitioniert, dass sich keine zwei Nachbarknoten in der gleichen Partition befinden. Jedem Knoten wird eine Zahl aus \mathbb{N} zugewiesen, so dass die Nachbarknoten auf unterschiedliche Zahlen abgebildet werden. Da ich für den Aufbau eines Task-Graphen die Idee der Greedy-Färbung benutzt habe, möchte ich zuerst das einfache Greedy-Färbungsverfahren vorstellen. Der Greedy-Algorithmus durchläuft die Knotenmenge des Graphen und weist jedem Knoten die minimale zulässige Farbe zu, die noch keinem seiner Nachbarn zugewiesen wurde, oder führt eine neue Farbe ein.

Algorithm 2 Greedy-Färbung [Ste07]

Require: Graph $G = (V, E)$, $V = \{v_0, \dots, v_n\}$

Ensure: Färbung $c[v]$

$c[v_1] \leftarrow 1$

for $i = 2$ to n **do**

$c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c(u) \forall u \in \Gamma(i) \cap \{v_1, \dots, v_{i-1}\}\};$

end for

5.3 Idee

Nach dem Feststellen aller Datenabhängigkeiten haben wir einen Graphen erhalten, in dem zwischen allen Knoten, die auf gemeinsame Daten zugreifen, eine Kante existiert. Allerdings ist der Datenabhängigkeitsgraph nicht schleifenfrei. Die Aufgabe besteht darin, eine partielle Ordnung auf dem Graphen herzustellen. Als Ordnungsrelation kann dabei die `maximalPath` verwendet werden. Die Knoten der Nachbarzellen sind aber miteinander durch ungerichtete Kanten verbunden, die als zwei gerichtete Kanten dargestellt sind, was dazu führt, dass keine Ordnung auf dem Graph hergestellt werden kann. Betrachten wir zwei Teilgraphen von G : $G_1 = (V, E_1)$, $G_2 = (V, E_2) \subseteq G$. Die Kantenmenge E_1 enthält dabei nur gerichtete und die E_2 nur die ungerichteten Kanten. Aus den vorhandenen gerichteten Datenabhängigkeiten

folgt, dass der Graph G_1 azyklisch ist. Die `maximalPath` Relation bildet dabei die gewünschte Ordnung in G_1 . Das Problem der Ordnungsherstellung in G kann auf das Färbungsproblem

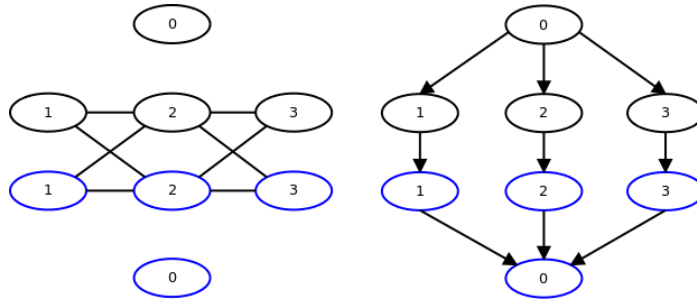


Abbildung 5.3: Teilgraphen von G

auf G_2 zurückgeführt werden, indem wir eine zusätzliche Bedingung aufstellen, dass die zugewiesenen Knotenwerte die Ordnungsrelation in G_1 erhalten. Eine Färbung $c : V \rightarrow \mathbb{N}$ ist dann zulässig, falls gilt:

1. $\forall v, w \in V, \{v, w\} \in E_2 \rightarrow c(v) \neq c(w)$
2. $\forall v, w \in V, \{v, w\} \in E_1 \rightarrow c(v) < c(w)$

5.4 Algorithmus Beschreibung

Als Eingabe für den Algorithmus dienen zwei Teilgraphen G_1 und G_2 , bei denen die Knotenmengen identisch sind. Als Ergebnis sollen wir eine Färbung der Knoten $c' : V \rightarrow \mathbb{N}$ erhalten, die Bedingungen 1 und 2 erfüllt. Der Greedy-Algorithmus wird erweitert, damit beim Vergeben einer Farbe nicht nur die Werte der Nachbarn berücksichtigt werden, sondern auch der aktuelle Wert des Knotens selbst. Wenn der Knotenwert bereits von allen Nachbarn verschieden ist, wird der nächste Knoten bearbeitet. Falls ein der Nachbarn die gleiche Farbe hat, wird der Wert des Nachbarn angepasst und die Bearbeitung wird mit dem Nachbarknoten fortgesetzt. Die Färbung eines Knotens ist im Algorithmus 4 formal beschrieben.

Algorithm 3 $color(v)$: Färbung eines Knoten mit einem Startwert

Require: Graph $G = (V, E)$, $V = \{v_0, \dots, v_n\}$, $v_i \in V$

Ensure: Gültige Färbung $c[v_i]$

if $\exists v_j \in \Gamma(v_i) \mid c[v_i] = c[v_j]$ **then**

$c[v_i] \leftarrow c[v_i] + 1$

$color(v_j)$

end if

Die gesamte Vorgehensweise besteht aus folgenden Schritten:

Schritt 0 Nehme eine Zusammenhangskomponente U_i von G_2 , wenn alle Zusammenhangskomponente bereits bearbeitet wurden, terminiert der Algorithmus.

Schritt 1 Erstelle auf U_i eine gültige Färbung mit dem erweiterten Greedy-Algorithmus.

Schritt 2

a) Überprüfe für alle Knoten aus U_i , die ausgehende gerichtete Kanten besitzen, ob die zweite Bedingung erfüllt wird.

b) Falls nötig, passe den Wert des Endknotens an.

Schritt 3 Falls im Laufe des zweiten Schrittes Knoten aus U_i verändert wurden, setze mit dem Schritt 1 fort. Sonst gehe auf den Schritt 0.

Algorithm 4 Geordnete Greedy-Färbung

Require: Graph $G_1 = (V, E_1)$, $G_2 = (V, E_2)$, $V = \{v_0, \dots, v_n\}$

Require: Zusammenhangskomponenten von $G_2 : \{U_1, \dots, U_m\}$, $U_i = (V_i, E'_i)$

Ensure: Färbung $c[v]$

```

for  $i = 1$  to  $m$  do
   $repeat \leftarrow true$ 
  while  $repeat = true$  do
     $repeat \leftarrow false$ 
    for all  $v \in U_i$  do
       $color(v)$ ;
    end for
    for all  $v \in V_i$  do
      for all  $w \in \Gamma(v), \{v, w\} \in E_1$  do
        if  $c[w] \geq c[v]$  then
           $c[w] \leftarrow c[v] + 1$ 
          if  $w \in V_i$  then
             $repeat \leftarrow true$ 
          end if
        end if
      end for
    end for
  end while
end for

```

Der Algorithmus gibt eine Knotenfärbung zurück, die uns die Angaben über die Ordnung unter den Knoten und die maximalen Pfade für jedes Element liefert. Diese Information kann direkt für die Traversierung mit der Parallel-For-Methode benutzt werden, wie es im Abschnitt 3.3.2 beschrieben wurde. Die Verteilung der Farben auf Tasks ist auf der Abbildung 5.4 zu sehen. Um einen allgemeinen gerichteten Task-Graph zu erstellen, muss man einen Schritt weiter gehen. Den Task-Graph definiert man als $T = (V, E_{task})$, wobei V die ursprüngliche Knotenmenge ist und E_{task} auf folgende Weise aus E konstruiert wird:

$$E_{task} = \{\{v_i, v_j\} \in E \mid c[v_i] < c[v_j]\}$$

Das Ergebnis enthält die komplette Information über die Vorgänger und die Nachfolger der Knoten, die in dem Workqueue Ansatz Gebrauch finden. Ein vollständiger Task-Graph ist auf der Abbildung 5.5 gezeigt.

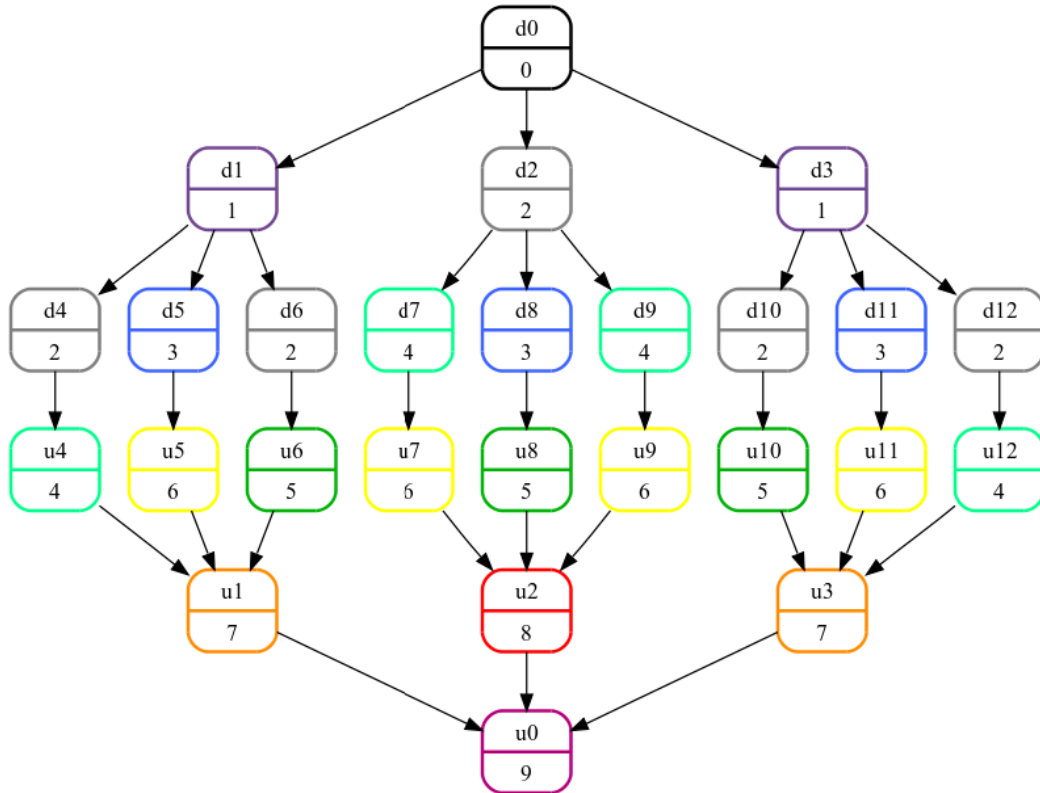


Abbildung 5.4: Färbung eines Task-Graphen mit $d=1$ und der Verfeinerungstiefe 2.
 Der maximale Pfad von der Wurzel (Farbnummer) ist unter dem Tasknamen angegeben. Die Tasks gleicher Farbe können parallel ausgeführt werden.

5.4.1 Implementierung

Der erhaltene Task-Graph erfüllt zwar alle Bedingungen und kann auch ohne weitere Änderungen für die Traversierung eingesetzt werden, er besitzt allerdings sehr viele Kanten, was sowohl den Speicherbedarf erhöht, als auch die Traversierung mit der Queue ineffizient macht. Es ist möglich, die überflüssigen Kanten zu entfernen, indem man die transitive Reduktion anwendet.

Obwohl der Zeitaufwand für die Graphenfärbung, den man der Tabelle 5.1 entnehmen kann, noch akzeptabel ist, kann man sich diese Berechnungen sparen, indem man eine Lookup-Tabelle für unterschiedliche Dimensionen und Levels anlegt. Dabei werden im Voraus Task-Graphen generiert und gespeichert. Für den Parallel-For-Ansatz muss man somit die Liste der Tasks erstellen, die nach deren Pfadlängen im Task-Graph aufsteigend sortiert sind. Die einzelnen Taskgruppen mit den gleichen Pfadlängen werden voneinander getrennt, damit beim Einlesen die Längen festgestellt werden können. Der Speicherverbrauch für solche Pfaddateien ist in der Tabelle 5.2 aufgeführt.

Der Shared Queue-Ansatz benötigt einen kompletten Task-Graph, der die Informationen über alle Kanten im Task-Graph enthält. Da die Kantenanzahl viel größer als die Knotenanzahl ist, belegen sie Dateien mit Task-Graphen mehr Speicherplatz, was die Daten in der Tabelle 5.3 zeigen.

<i>level</i>	<i>dim2</i>	<i>dim3</i>	<i>dim4</i>
1	0.000033	0.000161	0.000889
2	0.000159	0.004447	0.180553
3	0.001028	0.136144	22.410507
4	0.010753	4.712026	-
5	0.100455	-	-
6	1.275048	-	-
7	12.091439	-	-

Tabelle 5.1: Zeitaufwand in Sekunden für verschiedene Dimensionen und Levels.

<i>level</i>	<i>dim2</i>	<i>dim3</i>	<i>dim4</i>
1	124 B	300 B	796 B
2	808 B	6,1 KB	52,2 KB
3	6,5 KB	159,9 KB	4,1 MB
4	57,8 KB	4,2 MB	-
5	519,1 KB	-	-
6	4,6 MB	-	-
7	41,1 MB	-	-

Tabelle 5.2: Gemessener Speicherverbrauch für Pfaddateien.

<i>level</i>	<i>dim2</i>	<i>dim3</i>	<i>dim4</i>
1	592 B	3,2 KB	20,6 KB
2	6,9 KB	138,2 KB	3,2 MB
3	67,4 KB	4,2 MB	330 MB
4	624,9 KB	119,3 MB	-
5	5,5 MB	-	-
6	50,1 MB	-	-
7	451,3 MB	-	-

Tabelle 5.3: Gemessener Speicherverbrauch für vollständige Task-Graphen.

6 Implementierung mit OpenMP

Dieses Kapitel beschäftigt sich mit der realen Umsetzung der Ansätze für die Ausführung eines Task-Graphen, die im vierten Kapitel vorgestellt wurden. Die Parallelität der Traversierung wird mithilfe von OpenMP erreicht. OpenMP ist eine Erweiterung der Sprachen C++ und Fortran, wobei in diesem Fall die C++-Version eingesetzt wird. Die OpenMP API stellt dem Entwickler eine Reihe von Direktiven, Klausel und Umgebungsvariablen zur Verfügung, die bei der Shared-Memory Parallelisierung eines Programms verwendet werden. Über die Umgebungsvariablen kann man z.B. die Anzahl der Threads setzen oder den Scheduling spezifizieren. Mit Direktiven kann man dem Compiler mitteilen, welche Teile des Programms parallel ausgeführt werden sollen und welche sequentiell.

6.1 Shared Workqueue

Die Umsetzung des Shared Queue Ansatzes benötigt eine Queue `workqueue` für die Tasks, die keine Abhängigkeiten von den anderen Tasks haben und schon ausgeführt werden können, und eine Liste mit der Anzahl der verbliebenen Abhängigkeiten für alle Tasks: `parents`. Weiterhin muss man den Abschluss der Traversierung erkennen können, was durch einen Zähler `nodesToDo` der noch nicht abgearbeiteten Knoten realisiert wird.

Die Queue, die Liste der Vorgänger für alle Knoten und die Anzahl der noch nicht bearbeiteten Knoten werden von mehreren Threads parallel gelesen und geschrieben. Aus diesem Grund ist die Synchronisation bei Zugriffen auf diese Variablen notwendig. Der wechselseitige Ausschluss von Threads wird mit dem OpenMP-Konstrukt `critical` realisiert. Die Pragma `#pragma omp critical [(name)]` verhindert Race Conditions, indem sie die gleichzeitige Ausführung ihres Codes von mehreren Threads verbietet. Durch die Angabe des Namen kann man die kritischen Sektionen voneinander unterscheiden, so dass nur die Zugriffe auf gleich benannte Abschnitte synchronisiert werden. Die Parallelisierung erfolgt mit der Direktive `#pragma omp parallel` [Ope08]. Der in der Pragma enthaltene Code wird von allen Threads parallel ausgeführt.

Listing 6.1: Shared Queue Implementierung in OpenMP

```
workQueue.push(0);
#pragma omp parallel shared(workQueue, cellsToDo, parents)
{
    int nextNode, childValue, numberOfChildren;
    while(1){
        nextNode = -1;
        #pragma omp critical (queue)
        {
            if ( workQueue.size() != 0 ) {
                nextNode = workQueue.front();
                workQueue.pop();
            }
        }
    }
}
```

```

    }
  }
  if(nextNode != -1){
    processCell(nextNode);
    #pragma omp critical (cell_done)
    {
      cellsToDo--;
    }
    numberOfChildren = graphArray[nextNode][0];
    for(int child = 0; child < numberOfChildren; child++){
      childValue = graphArray[nextNode][child+1];
      #pragma omp critical (queue)
      {
        parents[childValue]--;
        if(parents[childValue]==0)
          workQueue.push(childValue);
      }
    }
  }
  if(cellsToDo <= 0) break;
}
}

```

6.1.1 Busy Waiting

Die durchgeführten Experimente haben die Ineffizienz des Shared Queue Ansatzes bei kleinen Problemgrößen gezeigt. Dabei verschlechtert sich die Leistung bei größerer Anzahl von Threads, was auf folgende Gründe zurückzuführen ist. Erstens ist die Erstellung der Threads selbst ein Kostenfaktor, der bei mehreren Threads entsprechend steigt. Zweitens werden die vorhandenen Threads nicht ausreichend mit Berechnungen versorgt, so dass sie den überwiegenden Anteil der Zeit warten müssen. Da dieses Warten als Busy Waiting realisiert wird, greifen die wartenden Threads ständig auf die gemeinsame Queue zu, um ihren Status zu überprüfen, was die gesamte Ausführung beeinträchtigt, weil dieser Zugriff in der kritischen Sektion stattfindet. Eine Lösung dieses Problems könnte zum Beispiel durch den Einsatz eines Wait-Mechanismus realisiert werden, wie es in Java mit `wait()/notify()` gemacht wurde. Allerdings enthält der OpenMP Standard momentan kein vergleichbares Konzept [WSL06].

6.2 Parallele For-Schleife

Nachdem die maximalen Pfadlängen aller Knoten bekannt sind, werden die Gruppen zu den vorhandenen Pfadlängen erstellt. Die äußere For-Schleife iteriert über die Pfadlängen von 0 bis N und muss sequentiell ausgeführt werden. Die innere Schleife iteriert über die Knoten gleicher Pfadlänge und wird mithilfe OpenMP Direktive `omp for` parallelisiert. Neben den für die Klausel `#pragma omp parallel` erhältlichen Parameter, kann man dabei den Scheduling spezifizieren. `#pragma omp parallel for schedule(type, [chunk])`. Wenn kein

Scheduling angegeben wird, werden alle Iterationen mit der `static` Methode an die Prozessoren verteilt. Dabei wird am Anfang der For-Schleife die gesamte Arbeitsmenge in gleichen Anteilen den Threads zugewiesen [Ope08].

Listing 6.2: Parallele For-Schleife mit statischer Iterationsverteilung.

```

for(int i = 0; i <= longestPath; i++){
#pragma omp parallel for
    for(int j = 1; j <= paths[i][0]; j++){
        processCell(paths[i][j]);
    }
}

```

6.2.1 Optimierungen

Eine der festgestellten Schwächen dieses Ansatzes ist die Barriere am Ende der inneren For-Schleife. Falls ein Teil der Knoten mit der maximalen Pfadlänge i bearbeitet wurde, könnten eventuell einige Knoten mit der maximalen Pfadlänge $i+1$ traversiert werden, wenn sie keine weitere Abhängigkeiten besitzen. Die Verteilung dieser Knoten auf die weniger belasteten Threads würde die Lastbalancierung verbessern und die gesamte Bearbeitungszeit verkürzen. Allerdings müssen Threads warten, bis die verbliebenen Knoten mit der maximalen Pfadlänge i fertig sind, um die Knoten aus der Gruppe $i+1$ zu bearbeiten. Dieser Nachteil kann zum größten Teil durch den *dynamischen Scheduling* ausgeglichen werden. Während der statische Scheduling im Voraus die Anzahl der Iterationen pro Thread bestimmt, wird beim dynamischen Scheduling jedem Thread eine Gruppe von Iterationen zugewiesen, deren Größe durch den `chunk` Parameter bestimmt wird. Falls ein Thread seine Arbeit erledigt hat, fordert er eine neue Gruppe der Iterationen an. Der dynamische Scheduling hilft, alle Threads möglichst gleich zu belasten.

Eine weitere Optimierungsmöglichkeit bietet das Einführen der minimalen Grenze für die Parallelisierung der For-Schleife. In der Variable `threshold` kann man die Anzahl von Tasks angeben, die für eine parallele Iteration benötigt werden. Kleinere Taskmenge werden sequentiell durchlaufen, was die Kosten für die Threaderstellung spart.

Listing 6.3: Parallele For-Schleife mit dynamischer Iterationsverteilung.

```

int threshold = 2;
for(int i = 0; i <= longestPath; i++){
    if(paths[i][0] > threshold){
#pragma omp parallel for schedule(dynamic)
        for(int j = 1; j <= paths[i][0]; j++){
            processCell(paths[i][j]);
        }
    }
    else{
        for(int j = 1; j <= paths[i][0]; j++){
            processCell(paths[i][j]);
        }
    }
}
}

```

7 Experimente und Vergleich der Ansätze

Um die gewählten OpenMP Ansätze unabhängig vom Peano-Framework zu testen, wurde eine Methode verwendet, die Berechnungen in den echten Ereignisaufrufen simulierte. Um die CPUs zu belasten, wurde dabei ein explizites Euler-Verfahren mit einer gegebenen Anzahl von Iterationen ausgeführt, durch die man die Ausführungszeit steuern kann. Die Messungen beinhalten nur die Ausführung der Task-Graphen, d.h. die Zeit für das Auslesen von generierten Graphen und die Allokation der Datenstrukturen wird nicht berücksichtigt. Die Task-Graphen werden vor der Ausführung erstellt und in Dateien abgespeichert, sodass deren Generierung keinen Einfluss auf die Laufzeit bei der Traversierung hat. Obwohl das Laden der Task Dateien auch etwas Zeit in Anspruch nimmt, wird diese Zeit bei den Messungen nicht mit einbezogen. Da während der gesamten Simulation mehrere gleiche Gitterstrukturen auftreten könnten, das Einlesen und die Initialisierung allerdings nur einmal durchgeführt werden müssten, würden sich diese Kosten über die gesamte Laufzeit verteilen. Die Testläufe wurden auf dem HLRB II Supercomputer im Leibniz Rechenzentrum durchgeführt, der auf der SGI's Altix 4700 Plattform aufbaut. HLRB II verwendet Prozessoren Intel Itanium2 Montecito Dual Core mit Taktfrequenz 1.6 GHz [Lei].

7.1 Vergleich der OpenMP-Ansätze

7.1.1 Konstante Bearbeitungszeiten

In dem ersten Experiment wurde die Leistung bei homogenen Ausführungszeiten aller Tasks verglichen. Diese Situation ist dann der Fall, wenn alle Zellen im Mehrgitter ungefähr gleich lang bearbeitet werden. Die Länge der Tasks ist bedeutend für die Abschätzung des Parallelisierungsaufwandes. Die Testläufe mit sehr kleinen Tasks der Länge von $15\mu s$ haben gezeigt, dass die Laufzeit erst beim zweimal verfeinerten Gitter verbessert wird. Die Ergebnisse sind in der Tabelle 7.1 zusammengefasst. Die Daten in der Tabelle zeigen, dass der Parallel-For Ansatz der Shared Queue in diesem Test deutlich überlegen ist.

Bei längeren Tasks wirkt sich der Overhead nicht so stark aus, was auf der Abbildung 7.1 veranschaulicht ist. Die Leistung beider Ansätze ist hier annähernd gleich. In den weiteren Experimenten mit homogener Lastverteilung wurden Tasks der Länge $68\mu s$ verwendet. In der Tabelle 7.2 sind die Testergebnisse der parallelen Traversierung von 2d und 3d Gittern

	<i>Ansatz</i>	<i>level1</i>	<i>level2</i>	<i>level3</i>	<i>level4</i>	<i>level5</i>
2d	Sequentiell	0.000286818	0.00261092	0.023463	0.211075	1.90005
	Parallel-For	0.000328064	0.000831127	0.00358891	0.0272231	0.238535
	Shared Queue	0.000404119	0.00174212	0.0147569	0.137611	1.25581

Tabelle 7.1: Ausführungszeiten in Sekunden für zwei Dimensionen mit 8 Threads auf Altix. Die Taskdauer beträgt $15\mu s$

7 Experimente und Vergleich der Ansätze

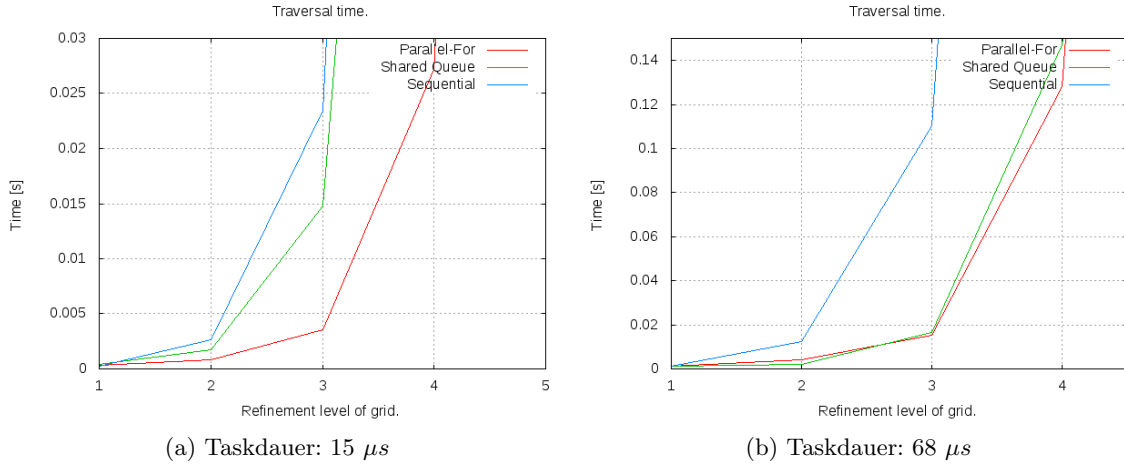


Abbildung 7.1: Ausführungszeiten bei homogener Knotenbelastung, $d=2$.

	<i>Ansatz</i>	<i>level1</i>	<i>level2</i>	<i>level3</i>	<i>level4</i>	<i>level5</i>	<i>level6</i>	<i>level7</i>
2d	Sequentiell	0.00133586	0.0121541	0.109508	0.985697	8.86833	79.8151	718.33
	Parallel-For	0.000843048	0.00254202	0.0150139	0.124729	1.11106	9.99328	89.9316
	Shared Queue	0.00105906	0.002244	0.01653	0.146527	1.32378	12.0895	109.221
3d	Sequentiell	0.00373816	0.10106	2.72852	73.6714	-	-	-
	Parallel-For	0.00152898	0.014874	0.344694	9.22118	-	-	-
	Shared Queue	0.00204301	0.0192361	0.599374	17.526	-	-	-

Tabelle 7.2: Ausführungszeiten in Sekunden für zwei und drei Dimensionen mit 8 Threads auf Altix. Die Taskdauer beträgt 68 μs .

dargestellt. Die Leistungsverbesserung ist bereits in der ersten Spalte für die einmal verfeinerten Gitter zu sehen.

Um die Skalierungseigenschaften der Ansätze zu untersuchen, wurden Testläufe mit unterschiedlicher Anzahl von Threads durchgeführt. Der erreichte Speedup wurde dabei aus der Relation der Zeit für sequentielle Traversierung und der Ausführungszeit der parallelen Methoden berechnet. Die Abbildung 7.2 zeigt die erreichten Speedups. Als Vergleichswert wird zusätzlich der lineare Speedup aufgeführt, der die bestmögliche Skalierung bedeutet.

Die maximale Beschleunigung ist allerdings nur dann möglich, wenn das Problem beliebig weit unterteilt werden kann. Die Parallelisierung mit dem Task-Graph-Pattern ist in seinem Speedup durch die Länge des maximalen gerichteten Pfades beschränkt. Bei kleineren Problemgrößen stehen außerdem nicht genug unabhängige Tasks zu Verfügung, um alle Threads auszulasten. Das erklärt, warum der Leistungsgewinn bei einem zweimal verfeinerten 2d-Gitter nur bis zu einer bestimmter Threadsanzahl erreicht wird, was auf der Abbildung 7.4 beobachtet werden kann.

Beim Shared Queue Ansatz trägt neben dem steigenden Kontrollaufwand auch das Busy Waiting Problem zur Verschlechterung der Leistung bei. Die zusätzlichen Threads, die nicht ausreichend mit Berechnungen ausgelastet werden, befinden sich andauernd in dem kritischen Abschnitt, was die Wartezeit bei Zugriffen auf die gemeinsame Variable `workqueue` erhöht.

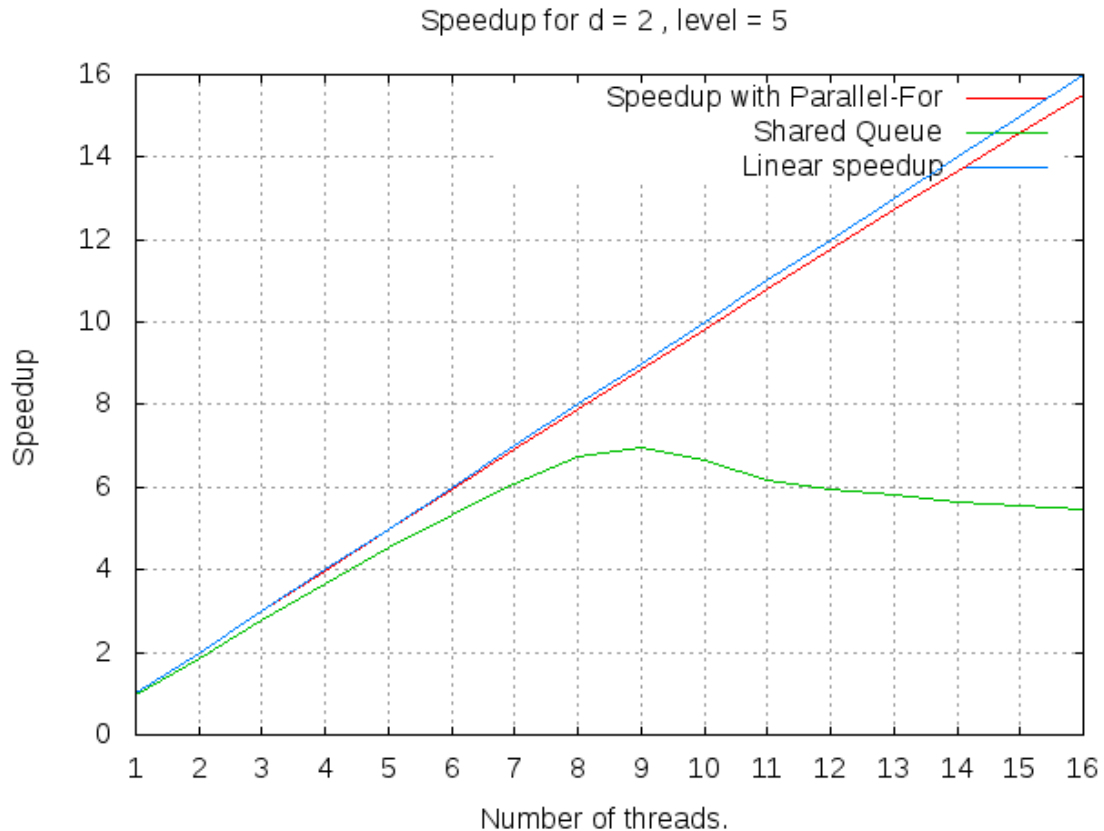


Abbildung 7.2: Speedup auf 16 Cores.

7.1.2 Variable Bearbeitungszeiten

Im zweiten Experiment wird die parallele Traversierung bei inhomogenen Bearbeitungszeiten getestet. Dabei soll überprüft werden, wie erfolgreich die Lastbalancierung einzelner Ansätze funktioniert. Im Peano Framework finden die Berechnungen an den maximal verfeinerten Zellen statt. Die größeren Zellen werden dagegen viel schneller abgearbeitet. Die Verteilung der Ausführungszeiten im zweiten Test berücksichtigt diese Besonderheit, so dass die Tasks, die den Blättern im Spacetree entsprechen, am meisten Zeit benötigen. Die erzielten Ergebnisse werden auf der Abbildung 7.5 veranschaulicht. Dabei wurde in dem ersten Teil des Experiments der statische Scheduling in der For-Schleife verwendet (7.5a), was eine unzureichende Lastbalancierung zur Folge hatte. Der Einsatz des dynamischen Scheduling (7.5b) hat die Laufzeit dahingehend verbessert, dass der For-Schleifen-Ansatz teilweise bessere Ergebnisse als der Shared Queue Ansatz liefert.

7.2 Shared Workqueue

Der Ansatz mit paralleler Queue erfordert mehr Kontrollaufwand, da es erst im Laufe der Traversierung endgültig entschieden wird, in welchem Schritt ein bestimmter Knoten bearbeitet wird. Der Vorteil dabei ist die Adaptivität des Verfahrens, die bei variablen Bearbei-

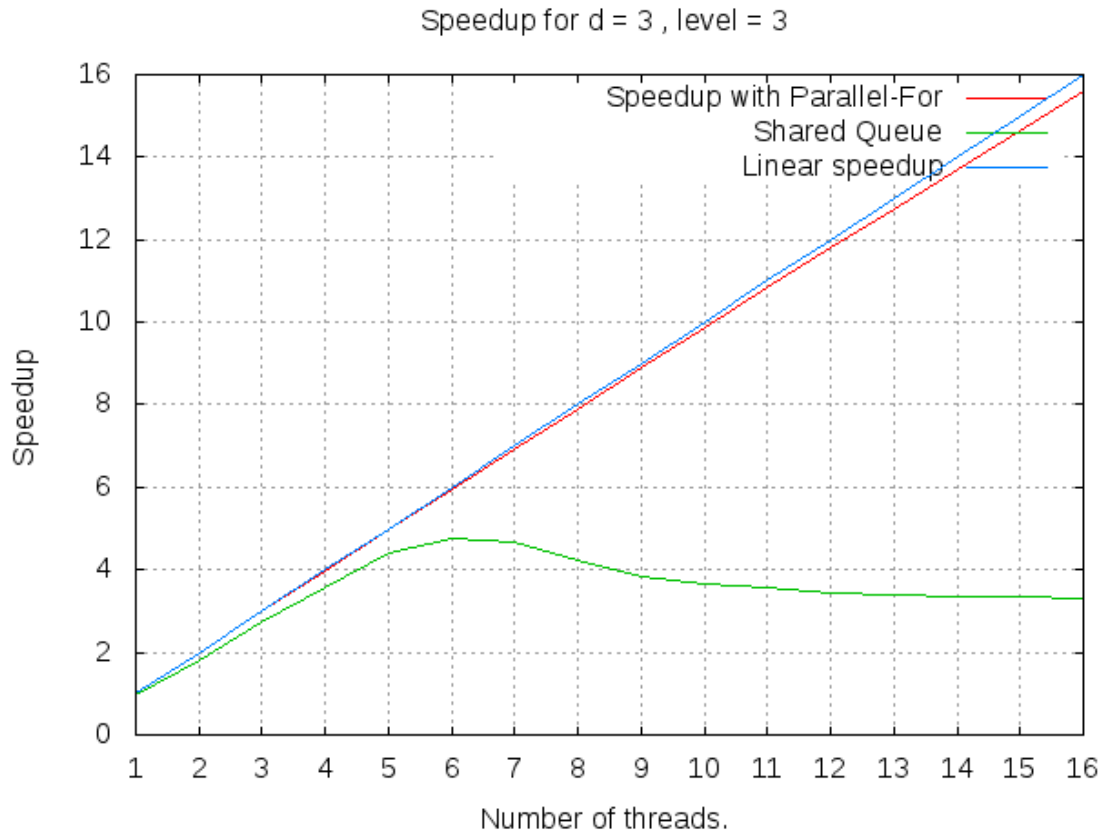


Abbildung 7.3: Speedup auf 16 Cores.

tungszeiten die gesamte Laufzeit verkürzt. Die Anforderung an den minimalen Aufwand ist allerdings nicht erfüllt, was man an der Verschlechterung der Ausführungszeit bei kleinen Task-Graphen sieht. Somit ist die Nutzung dieser Methode auf Probleme beschränkt, die entweder auf einem weit verfeinerten Gitter gelöst werden oder viele rechenlästige Operationen in jedem Task benötigen.

7.3 Parallele For-Schleife

Die Traversierungsmethode mit paralleler For-Schleife zeigt bessere Skalierungseigenschaften, als der erste Ansatz, weil der Aufwand beim Einführen zusätzlicher Threads wegen dem statischen Scheduling geringer ist. Außerdem ermöglicht dieser Ansatz bereits bei kleinen Graphen die Beschleunigung gegenüber der sequentiellen Traversierung, was in der realen Anwendung sehr wichtig ist. Die mangelnde Adaptivität konnte auf Kosten etwas größeren Overheads durch dynamischen Scheduling erfolgreich verbessert werden. Ein weiterer Vorteil ist die kleinere Datenmenge, die für Parallel-For notwendig ist. Im Unterschied zu der Queue-Methode muss nicht der komplette Graph erstellt, gespeichert und geladen werden, sondern nur die Knotenliste, die entsprechend den Pfadlängen sortiert ist.

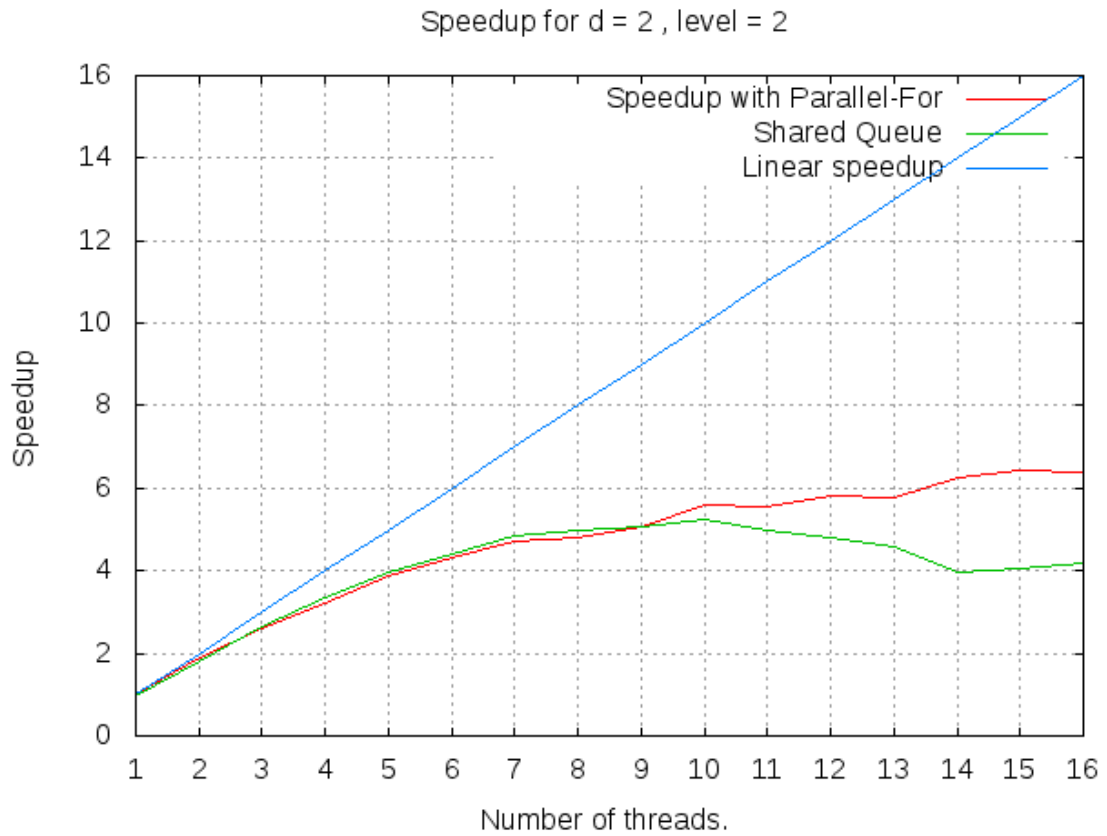


Abbildung 7.4: Speedup auf 16 Cores.

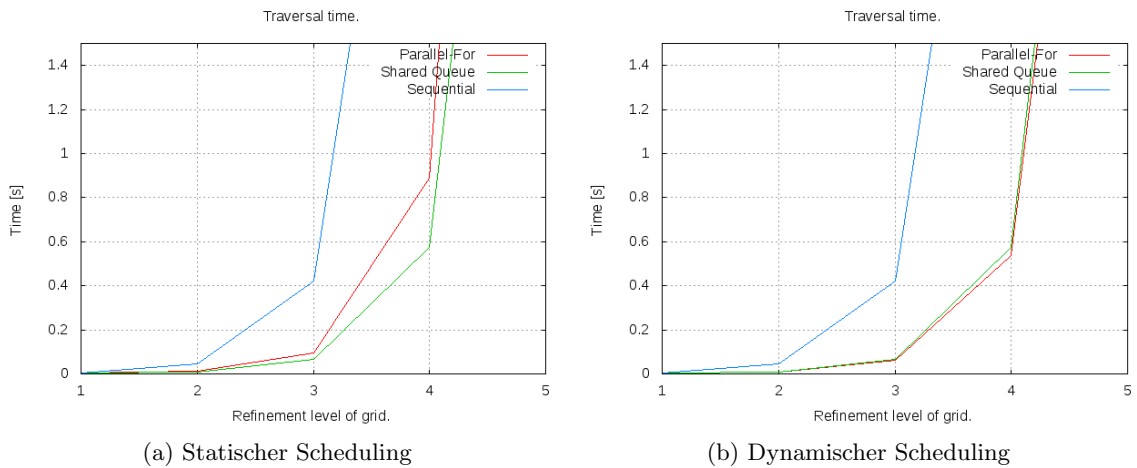


Abbildung 7.5: Ausführungszeiten bei variabler Knotenbelastung.

8 Integration in Peano Framework

In diesem Kapitel werden die wichtigsten Schritte der Integration paralleler Gittertraversierung in das Peano-Framework beschrieben. Ein Teil der Integration wurde bereits durchgeführt, so dass man die ersten Schlüsse aus den Ergebnissen ziehen kann. Im Abschluss werden die verbliebenen Integrationsschritte betrachtet, die eine Richtung für die weitere Arbeit an dem Projekt vorgeben.

8.1 Identifizieren von regulären Gitterstrukturen

Da die parallele Gittertraversierung nur auf den regulär verfeinerten Gittern funktioniert, müssen die passenden Gitterabschnitte als solche erkannt werden. Weiterhin benötigt der Algorithmus die Dimension und die Verfeinerungstiefe des gefundenen regulären Baumes. Während die Dimension bekannt ist und für alle Gitterteile eines Gitter gleich bleibt, muss die Tiefe im Laufe der Ausführung berechnet werden. Für diese Aufgabe konnten einige Ergebnisse der Diplomarbeit von Wolfgang Eckhard benutzt werden [Eck09]. In Rahmen der Arbeit wurde ein Schlüsselwert `patternID` in Peano eingeführt, der für jede Zelle die Tiefe des regulären Baumes angibt, den ihre Kindzellen bilden.

8.2 Traversierung

In meiner Arbeit konnte ich von dem Adapter-Mechanismus von Peano profitieren. Obwohl die Traversierung des Gitters in einer der grundlegenden Komponenten stattfindet, auf der alle anderen Plugins aufbauen, waren dank dem Ereignis-Konzept nur minimale Änderungen direkt im Peano-Code notwendig, um die Taskgraph-basierte Zellenbearbeitung in die sequentielle Traversierung zu integrieren. Im Laufe der sequentiellen Traversierung werden die `patternIDs` der Zellen überprüft, die Information über die Tiefe des Teilbaumes liefern. Nachdem die Dimension und die Tiefe eines Baumes bekannt sind, kann die parallele Traversierung beginnen. Vor dem Start der Traversierung muss die Knotenliste aus der Pfaddatei geladen werden, die mittels Parallel-For-Ansatz durchlaufen wird. Im Unterschied zur Testversion des Traversierungsalgorithmus, müssen dabei allerdings einige zusätzliche Parameter für jede Zelle berechnet werden. In der Abhängigkeit von der Startzelle, an der die Traversierung angefangen hat, werden die Koordinaten und die Maschenweite bestimmt, die später von den PDE-Lösern benötigt werden.

Da das Gitter nicht unbedingt auf dem ganzen Rechengebiet regulär verfeinert wird, ist es nicht immer möglich alle Zellen durch das Ausführen eines Task-Graphen abzuarbeiten. In diesem Fall werden die größtmöglichen regulären Teilbäume parallel traversiert. Der sequentielle Teil wird von der Basiskomponente durchgeführt.

Zusätzlicher Schlüsselwert.

Mittels bereits vorhandener Schlüsselwerten, die in Peano für die Rekursionseliminierung

eingesetzt werden, wurde die Möglichkeit untersucht, die parallele Traversierung in die bestehende sequentielle Gitterabarbeitung einzubauen. Der weitere Schritt wäre es, einen unabhängigen Schlüsselwert einzuführen, der an die Anforderungen der parallelen Traversierung genau angepasst wäre. Dabei müssten im Unterschied zur Definition von `patternID` die Randzellen berücksichtigt werden.

9 Ergebnisse und weitere Arbeitsrichtungen

Die vorliegende Arbeit hat Vor- und Nachteile der parallelen Gittertraversierung mittels Task-Graphen festgestellt. Der untersuchte Parallelisierungsansatz umfasste drei Schritte:

1. Das Aufstellen eines Abhängigkeitsgraphen, der alle Abhängigkeiten zwischen Tasks enthält. Dieser ist auf Peano zugeschnitten, kann aber für allgemeinere Baumtraversierungen verwendet werden.
2. Der Aufbau eines Task-Graphen aus dem Abhängigkeitsgraphen.
3. Zwei verschiedene Traversierungen des Task-Graphen mit den Shared-Memory-Techniken von OpenMP.

Eine Problematik im ersten Schritt ist der Speicherverbrauch. Wie es im Kapitel 4 angesprochen wurde, müssen bei der Traversierung der Mehrgitter sehr viele Abhängigkeiten berücksichtigt werden, deren Anzahl exponentiell mit dem Verfeinerungsgrad des Gitters wächst. Wegen der hohen Speicheranforderungen ist das Erstellen eines kompletten Datenabhängigkeitsgraphen höchstens für ein Gitter der Tiefe 7 bei $d = 2$ und ein Gitter der Tiefe 4 bei $d = 3$ möglich. Eine optimierte Datenstruktur könnte dabei zwar den Speicherverbrauch minimieren, aber durch das exponentielle Wachstum würde der Datenabhängigkeitsgraph trotzdem sehr schnell die Grenze des Arbeitsspeichers erreichen. Da das Aufstellen eines Datenabhängigkeitsgraphen aber nur ein Zwischenschritt in der Parallelisierung ist, der das weitere Vorgehen vereinfacht, könnte man auf das Abspeichern dieses Graphen verzichten und die Abhängigkeiten während der Generierung des Task-Graphen berechnen, denn dank der regulären Struktur der Gitter können die Abhängigkeiten zwischen den Nachbarzellen sehr effizient berechnet werden. Dadurch wächst jedoch der Zeitaufwand für die Generierung des Task-Graphen. Während der Aufbau des Task-Graphen im Vorfeld geschehen kann, erfolgt das Einlesen der Datei direkt vor der Traversierung des Gitters. Somit muss die Task-Datei wegen der Arbeitsspeicherkapazität und der Einlesezeit eine akzeptable Größe besitzen.

Durch die beschriebenen Beschränkungen im Speicher für Abhängigkeitsgraphen und in der Größe der Task Dateien werden Grenzen für Teilgitter gesetzt, die sich im Durchlauf eines Task-Graphen parallel traversieren lassen. Ein reguläres Gitter mit Verfeinerungstiefe l , die den maximalen für die Parallelisierung verfügbaren Level *maximalLevel* übersteigt, könnte jedoch in $k^{d(l-maximalLevel)}$ Teilgitter zerlegt werden, die ein nach dem anderen jeweils parallel abgearbeitet werden könnten.

Im dritten Schritt wurden zwei Methoden untersucht, mit denen man den Task-Graph parallel ausführen kann. Mit dem Parallel-For Ansatz ist es gelungen, bei akzeptablem Overhead eine gute Skalierbarkeit und Lastbalancierung zu erzielen. Die Akzeptanzgrenze für den Overhead richtet sich allerdings nach der Dauer der Tasks, die ausgeführt werden. Bei sehr

kurzen Task-Bearbeitungszeiten bringt dieser Ansatz erst bei der Traversierung von zweimal verfeinerten Gittern einen Vorteil.

Der zweite Ansatz, der auf der Shared Queue basiert, konnte zwar eine gute Lastbalancierung zeigen, war aber wegen dem größeren Overhead im Vergleich zu der Parallel-For Implementierung schlechter für die parallele Gittertraversierung geeignet. Einer der Gründe für den Vorsprung der Parallel-For-Methode war einerseits die bessere Unterstützung der Datenparallelität in OpenMP, die über parallele For-Schleifen realisiert ist, andererseits fehlende Konstrukte, die in OpenMP das Busy-Waiting-Problem in der Shared Queue beheben könnten. Da die Umsetzung des Shared Queue Patterns allerdings nicht an die Verwendung von OpenMP gebunden ist, könnte der Ansatz mit anderen Mitteln implementiert werden, die notwendige Mechanismen enthalten, um das Busy Waiting zu vermeiden. Eine Alternative würde der Einsatz vom neuen Task-Konzept in OpenMP 3.0 darstellen, mit dem man die Tasks während der Laufzeit erstellen kann. Damit könnte man die Anzahl der Tasks verringern, falls weniger Aufgaben gleichzeitig bearbeitet werden können.

Zusammenfassend kann man feststellen, dass die durchgeführten Experimente größtenteils die Erwartungen an die untersuchten Konzepte bestätigt haben. Der Einsatz der Shared-Memory Techniken und insbesondere der parallelen Schleifen bei der parallelen Gittertraversierung erscheint dabei sehr vielversprechend in Hinsicht auf die Leistungssteigerung. Um allerdings die genaue Aussage über das Nutzen dieser Methode in einer realen Anwendung zu treffen, müsste man viele Parameter in Betracht ziehen: die Längen der Tasks, die Anzahl der regulären Teilgitter und den Verfeinerungsgrad dieser Teilgitter. Somit bietet sich als nächster Schritt an, die Integration des Ansatzes in das Peano-Framework durchzuführen, um die Untersuchung mit Realexperimenten zu vervollständigen. Die erfolgreiche Leistungsverbesserung würde den besten Nachweis der Funktionsfähigkeit des Ansatzes liefern.

Das Einsatzgebiet dieses Konzepts ist allerdings bei weitem nicht allein auf das Peano-Framework beschränkt. Auch andere Anwendungen, die auf den Mehrgittern und Spacetrees aufbauen, könnten von seiner Idee profitieren. Somit ergibt sich viel Raum für weitere Verbesserungen und Experimente.

Literaturverzeichnis

- [But09] BUTNARU, DANIEL: *Parallelization of a Multiscale CFD Solver within the Peano Framework*. 2009.
- [C++10] C++ REFERENCE: *STL Containers*, 2010. <http://www.cplusplus.com/reference/stl/>.
- [Eck09] ECKHARDT, WOLFGANG: *Automatisierte Rekursionseliminierung für einen dynamisch adaptiven PDE Löser*. 2009.
- [GMBM08] G. MATTSON, TIMOTHY, A. SANDERS BEVERLY und BERNA L. MASSINGILL: *Patterns for Parallel Programming*. 2008.
- [Lei] LEIBNIZ-RECHENZENTRUM: <http://www.lrz.de/services/compute/hlrb/hardware/>.
- [Ope08] OPENMP APPLICATION PROGRAM INTERFACE: *Version 3.0*, 2008.
- [Ste07] STEGER, ANGELIKA: *Diskrete Strukturen*, Band 1. 2007.
- [Wei09] WEINZIERL, TOBIAS: *A Framework for parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Dr. Hut, 2009.
- [WSL06] WIRZ, ALEXANDER, MICHAEL SÜSS und CLAUDIA LEOPOLD: *A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem*, 2006.