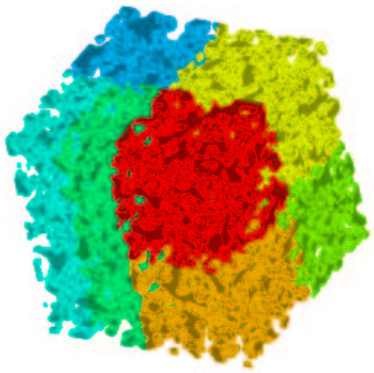
Technische Universität München
Fakultät Bauingenieur- und Vermessungswesen
Lehrstuhl für Bauinformatik
Prof. Dr. rer. nat. Ernst Rank

Diplomarbeit

Entwicklung eines Gebietszerlegers
für parallele Lattice Boltzmann Applikationen
auf der Basis von rekursiv-bipartiven Algorithmen
für equidistante teilgefüllte 3D-Gitter



Verfasser: Alexander Gun
18. November 2002

Betreuer: Dipl. Ing. Manuel Schulz

Verfasser: Alexander Gun
Betreuer: Dipl. Ing. Manuel Schulz

Begonnen am: 18. August 2002
Eingereicht am: 18. November 2002

Aufgabenstellung

Verschiedene Parameter haben einen Einfluß auf die Effizienz und das Verhalten einer Parallelisierung. In erster Linie sind das die Hardware, in Form der Maschinenbauart und Netzwerk, die Umsetzung des Algorithmus, sowie die Zerlegung des Workloads und die daraus folgende Kommunikation zwischen den Prozessen. Da in der Regel die Berechnung der gängigen Lattice Boltzmann-Codes auf uniformen Gittern erfolgt, wird in den meisten parallelen Applikationen auf einen geometrischen Gebietszerleger zurückgegriffen. Das zu zerlegende Simulationsgebiet wird bezüglich des Loadbalancings in gleich große Teile aufgeteilt. Bei den ersten LB-Applikationen wurden selbst unbenutzte Verteilungen auf den Geometrieknoten in die Berechnung der Kollisionroutine und der Propagation einbezogen, was in Applikationen mit einem geringen Geometrieanteil nicht weiter ins Gewicht fiel. Im Gegenteil, Vorteile ergeben sich bezüglich einer einfachen Datenstruktur und der Vektorisierung, was die Effizienz des Codes auf Vektorrechnern entschieden erhöht: Die Gebietszerlegung wird so gewählt, dass bei einem möglichst gleichmäßigen Verhältnis von Knoten pro Prozessor die Anzahl der Randknoten jedes Prozesses minimal gehalten wird. Große Defizite gibt es dagegen bei Simulationsgebieten mit einem hohen Geometrieanteil nicht nur bezüglich des Speicherbedarfs, sondern auch wegen des erhöhten Rechenbedarfs. Resultierend aus diesen Problemen wurde am Lehrstuhl für Bauinformatik eine Lattice Boltzmann Applikation mit einer Datenstruktur entwickelt, bei der nur Speicher für Fluid- bzw. Berandungsknoten der Geometrie allokiert wird. Infolge dessen ist es nur natürlich Verfahren der Gebietszerlegung zu verwenden, die nur die zur Berechnung benötigten Knoten auf die Prozesse verteilen. Für diesen Zweck werden gewöhnlich einfache geometrische Gebietszerleger verwendet, die das Simulationsgebiet in Streifen oder Boxen aufteilen, jedoch den Einfluß der Kommunikation vernachlässigen. Im Zuge der Weiterentwicklung des wissenschaftlichen Rechnen sind sehr viel komplexere Ansätze, wie Graph Partitioning Algorithmen, entwickelt worden, die den Einfluß der Rechen- und Kommunikationssaufwand zwischen Gebieten berücksichtigen können, jedoch in Folge des hohen Speicherbedarfes bei großen Systemen für LB- Methoden nur begrenzt einsetzbar sind.
Erstellen Sie im Rahmen dieser Arbeit einen geometrischen Gebietszerleger, der abhängig vom verwendeten Lattice Boltzmann Modell den Einfluß des Loadba-

lancings und des Kommunikationsvolumens berücksichtigen kann. Da die Applikation vor allem für die Zerlegung großer Systeme geeignet sein soll, ist auf einen minimalen Speicherbedarf zu achten. Vergleichen Sie die Ergebnisse soweit möglich mit dem "state of the art" Gebietszerleger METIS bzw. ParMETIS.

Erklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Ich erkläre mich damit einverstanden, dass die Diplomarbeit auf unbefristete Zeit zu Hochschulzwecken aufbewahrt werden darf.

München, den 18. November 2002

(Alexander Gun)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Viele Zweige der Industrie und Forschung sind damit beschäftigt, die Auswirkung von strömenden Medien auf verschiedene Objekte zu ergründen. In der Vergangenheit konnte man allein durch teuere Versuche im Windkanal (Luft) oder Strömungskanal (Wasser) zuverlässige Ergebnisse ermitteln. Mit zunehmender Rechnerleistung ist es möglich geworden, das Strömungsverhalten fließender Medien am Rechner zu simulieren.
Am Lehrstuhl für Bauinformatik der TU-München verfolgt das Projekt "VirtualFluids" das Ziel, komplexe Strömungssimulationen durchzuführen. Der Rechenkernel des Projekts "VirtualFluids" arbeitet nach der Lattice-Boltzmann-Methode, um die fluidmechanischen Größen zu berechnen [2]. Ein Vorteil der Lattice-Boltzmann-Methode ist, dass sich der Algorithmus effizient parallelisieren lässt. Die Parallelisierung des Rechenkernels wurde in verschiedenen Arbeiten am Lehrstuhl für Bauinformatik ständig verfeinert [1]. Um die Effizenz der parallelen Version zu maximieren, werden die Eingangsdaten für die Simulation unabhängig vom Rechenkernel mittels Preprozessor generiert. Aufgabe des Preprozessors ist es, das Simulationsgebiet so zu generieren, dass die Auslastung der zur Simulation verwendeten Prozessoren möglichst gleichmäßig ist und die Kommunikation zwischen den Prozessoren minimiert wird. Um das zu erreichen, muss der Preprozessor das Strömungsgebiet möglichst geschickt in so viele Teilgebiete unterteilen wie Prozessoren zur Berechnung zur Verfügung stehen.
Ziel dieser Arbeit ist es, verschiedene Methoden der Gebietszerlegung vorzustellen, diese zu analysieren und die Effizienz der Zerlegungen zu vergleichen. Hierfür werden verschiedene Ansätze zur Gebietszerlegung aufgezeigt, ein Beispiel der verfügbaren Gebietszerleger in den Preprozessor integriert und mit einem, im Rahmen dieser Arbeit entstandenen, Gebietszerleger verglichen.
Als Programmiersprache wurde ANSI-C gewählt, weil damit die geringsten Schwierigkeiten bei der Portierung auf diverse Plattformen entstehen und eine hohe Performance erreicht werden kann.

Kapitel 2

VirtualFluids

Das Projekt VirtualFluids besteht im wesentlichen aus drei Teilen. Die erste Komponente ist der Preprozessor, die zweite der eigentliche Berechnungskernel und die dritte Komponente stellen Analysetools für die graphische Aufbereitung der Ergebnisse dar (Postprocessing). Die in dieser Arbeit relevanten Komponenten sind der Preprozessor und der Berechnungskernel.

2.1 Der Preprozessor

Der Preprozessor erstellt die Eingangsdaten für die Simulation. Dabei werden zunächst die geometrischen Daten der um- bzw. durchströmten Objekte und der Strömungsrandbedingungen eingelesen. Die Objekte werden in einem regelmäßigen 3D-Gitter abgebildet. Unter einem regelmäßigen Gitter versteht man ein Gitter, in dem die Gitterpunkte alle den gleichen Abstand von einander besitzen (equidistant). Die eingearbeiteten geometrischen Objekte stellen meistens Hindernisse dar, die von dem Fluid umströmt werden müssen. Die Gitterpunkte die ein Hindernis abbilden werden demnach nicht durchströmt, und können bei der Simulation außer Acht gelassen werden. Durch die dadurch entstehenden Löcher, spricht man von einem equidistanten, teilgefüllten 3D-Gitter. Nachdem die geometrischen und fluidmechanischen Informationen (hier: geometrische Objekte und Strömungsrandbedingungen) in die 3D-Matrix eingearbeitet sind, kann das Berechnungsgebiet in Teilgebiete aufgeteilt werden. Nach Festlegung der Teilgebiete werden die Eingabedateien für den Simulationskern geschrieben. Damit ist die Arbeit des Preprozessors beendet. Im folgenden wird auf die Methoden und Algorithmen bei der Aufteilung des Gesamtgebiets in mehrere Teilgebiete näher eingegangen.

2.2 Der Berechungskernel

Der Rechenkern von VirtualFluids basiert auf dem Lattice-Boltzmann-Verfahren. Zuerst werden die vom Preprozessor erzeugten Eingabedateien eingelesen, danach wird die Simulation gestartet. Am Ende liegen die Ergebnisse in Ausgabedateien vor, die zur Auswertung herangezogen werden. Im Lattice-Boltzmann-Verfahren ist der Zustand an einem Knoten durch einen Satz Wahrscheinlichkeitsverteilungen beschrieben $f_i(t,x), i \in \{0,...,N-1\}$. Die Wahrscheinlichkeiten beschreiben die Eigenschaft eines Partikels zu einem Zeitpunkt t an einem Ort x mit der diskreten Geschwindigkeit e_i zu sein. Die Wahrscheinlichkeiten für einen folgenden Zeitschritt werden nach der diskretisierten Lattice-Boltzmann Gleichung

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x,t) = \Omega_i(x,t) \tag{2.1}$$

berechnet [2]. Der Kollisionsoperator

$$\Omega_i = \frac{1}{\tau}(f_i - f_i^{(eq)}) \tag{2.2}$$

basiert auf der "Single Time Relaxation Approximation" (STRA) und repräsentiert die Änderungsrate während der Kollision. Mit dem Faktor $\frac{1}{\tau}$ kann nach

$$\tau = \frac{6\nu + 1}{2} \tag{2.3}$$

die dynamische Viskosität eingestellt werden. Die makroskopischen Werte p und u berechnen sich zu

$$p(t,x) = \sum f_i(t,x) \tag{2.4}$$

$$u(t,x) = \frac{1}{p}\sum f_i(t,x)e_i. \tag{2.5}$$

Je nach geometrischem Modell[1] bilden entweder 15 oder 19 Wahrscheinlichkeiten einen Satz, der für jeden Knoten bereitstehen muss. Das Lattice-Boltzmann-Verfahren kann deshalb sehr rechen- und speicheraufwendig sein. Durch die lokal begrenzte Abhängigkeit der Knoten ist das Verfahren sehr gut parallelisierbar. Im Rechenkern von VirtualFluids wird zwischen den Knoten, die zur Kollision Daten von Knoten innerhalb des Teilbereiches, und Knoten, die Daten aus anderen Teilbereichen benötigen, unterschieden (siehe Abb. 2.1). Die Kollision ist der Teil der Berechnung, in dem die Ergebnisse von einem Zeitschritt zum nächsten ermittelt werden. Um die Performance zu steigern, wird die Kommunikation der Teilprozesse parallel zur Kollision der "inneren" Knoten durchgeführt (siehe Abb. 2.2) [1].

[1] In dieser Implementierung wird entweder d3q15 oder d3q19 verwendet. Das Lattice-Boltzmann Verfahren ist nicht auf diese beiden geometrischen Modelle festgelegt.

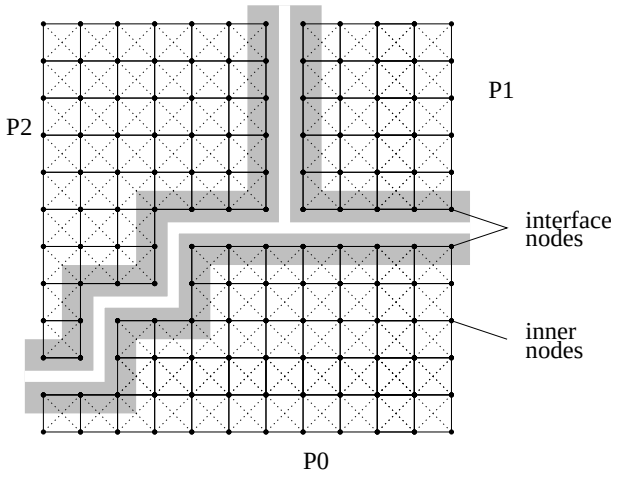


Abbildung 2.1: Unterscheidung der Knotentypen

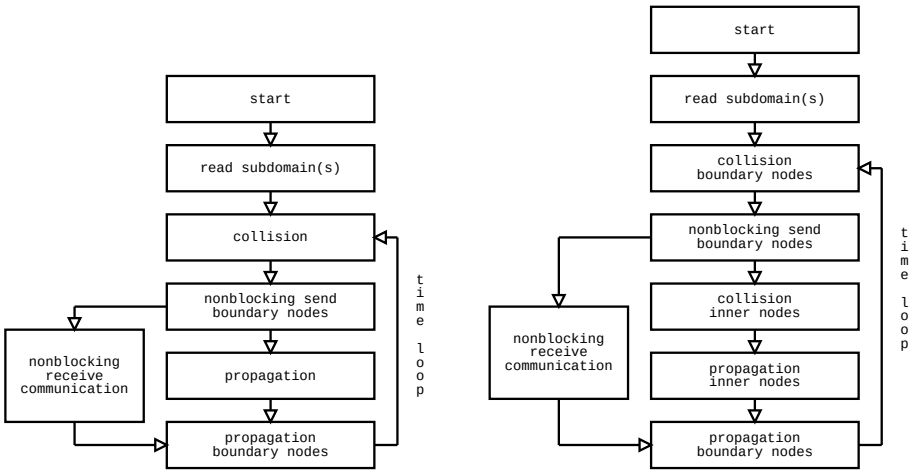


Abbildung 2.2: Flußdiagramm für den parallelen Standart LB-Algorithmus und die optimierte Implementierung (rechts) [1]

2.3 Paralleles Rechnen

Der Rechenkernel des verwendeten Strömungssimulators "VirtualFluids" wird sowohl für parallele Rechner mit "Distributed Memory" als auch für Rechnersysteme mit Kombinationen aus "Distributed Memory" und "Shared Memory" programmiert und optimiert. "Distributed Memory" heißt, dass jeder Prozessor auf seinen Speicherbereich exklusiven Zugriff hat. Werden Daten eines anderen Prozessors benötigt, kann dies nur über explizite Kommunikation zwischen den Prozessoren erfolgen. Im Gegensatz dazu kann in einem "Shared Memory" - System auf den Arbeitsspeicher von allen Prozessoren "gleichzeitig" zugegriffen werden. Da die Speicherzugriffe so erfolgen müssen, dass keine Inkonsistenzen entstehen, steigt die Anzahl der Zugriffskonflikte mit der Zahl der verwendeten Prozessoren. Um eine gute Skalierbarkeit und Flexibilität in der Systemwahl zu erreichen, wurde das "Distributed Memory" System gewählt. Die dazu verwendbare Hardware ist breit gefächert und reicht von Supercomputern (Hitachi SR8000-F1 [22]) über Workstationcluster mit Alpha-Prozessoren bis hin zu Linux-Clustern mit Standard-PCs und Intel-Prozessoren. Die Implementierung des Rechenkernels wurde für die Workstation- und PC-Cluster mit MPI (Message-Passing-Interface [18]) durchgeführt. Für rechenintensive Simulationen wurde der Code zusätzlich speziell an die Hardware des Pseudo-Vektor-Rechners Hitachi SR8000-F1 angepasst [22]. Innerhalb dieses Rechners findet man sowohl Shared- als auch Distributed-Memory Bereiche. Jeweils 8 Nodes bilden einen Shared-Memory-Knoten, welche zusammen mit weiteren Knoten, wie ein Distributed-Memory Rechner agieren. Um diese Hardware optimal zu nutzen wird die Bibliothek COMPAS [23], welche eine Mischung aus OpenMP [24] und MPI [18] darstellt, verwendet. Des weiteren wurde der Kernel darauf optimiert, den Einfluss der Kommunikation während der Simulation so gering wie möglich zu halten. Im Rahmen der Arbeit wird darauf nicht weiter eingegangen. Die Algorithmen im Rechenkernel sind in [1] ausführlich beschrieben.
Der Einsatz von Parallelrechnern bringt neue Probleme mit sich, wie folgende Erläuterungen verdeutlichen.

2.4 AMDAHLsches Gesetz

Um ein Problem mit Hilfe von Parallelrechnern zu bearbeiten, muss ein Algorithmus parallelisierbar sein. Zur Bewertung der Leistung eines parallelen Algorithmus, wird nach [17] die Ausführungszeit t_S des Algorithmus A_S auf einem Prozessor mit der Ausführungszeit t_P des Algorithmus A_P auf P Prozessoren verglichen. Der sog. "Speedup" berechnet sich zu

$$S_P = \frac{t_S}{t_P} \tag{2.6}$$

Idealerweise erhält man einen Speedup von $S_P = P$, d.h. P Prozessoren sind P-mal schneller als ein Einzelner. Ist die Gesamtarbeit eines Problems $N_{ges} = const.$, dann ist die Arbeit eines Prozessors $N_i \sim \frac{1}{P}$, d.h je mehr Prozessoren verwendet werden, desto weniger hat der einzelne Prozessor zu tun. Nach dem AMDAHLschen Gesetz besteht jeder Algorithmus aus Teilen, die sich nicht parallelisieren lassen. Ist s der serielle und p der parallele Anteil des Algorithmus, so wird die Summe $s + p$ so normiert, dass gilt

$$s + p = 1 \tag{2.7}$$

Damit lässt sich die Rechenzeit eines Problems auf einem Prozessor auf $t_1 \rightarrow s+p$ und (bei idealem Verhalten) die Rechenzeit auf P Prozessoren durch $t_P \rightarrow s + \frac{p}{P}$ ersetzen. Eingesetzt in Formel 2.6 ergibt sich eine obere Schranke für den maximal erreichbaren Speedup.

$$S_{P,max} = \frac{s+p}{s+\frac{p}{P}} = \frac{1}{s+\frac{1-s}{P}} \leq \frac{1}{s} \tag{2.8}$$

Daraus resultiert, dass z.B. ein Algorithmus mit einem seriellen Anteil von nur 1% einen maximalen Speedup von $S_{\infty,max} = 100$ hat.
In Tab. 2.1 wurden die maximalen Speedups bei einem seriellen Anteil $s = 1\%$

P	10	100	1000	10000
S_P	9	50	91	99

Tabelle 2.1: max. Speedup bei 1% serieller Anteil

berechnet. Es wird deutlich, dass die Problemlösung mit mehr als 100 Prozessoren unwirtschaftlich ist. Bei zehnfachem Rechnereinsatz ($P = 1000$) steigt die Gesamtperformance nur noch um 82%.
Da das AMDAHLsche Gesetz keine Kommunikationszeiten berücksichtigt, fällt das reale Verhalten des Speedups noch schlechter aus.
Eine effiziente Ausnutzung der Hardware ist also nur dann erreichbar, wenn die Rechenzeit (Problemgröße) entsprechend hoch ist und die Kommunikation zwischen den Prozessoren minimal wird.
Im Rechenkernel von "VirtualFluids" wird die Kommunikation parallel zur Berechung der inneren Knoten ausgeführt (siehe Abb. 2.1). Die Kommunikation erfolgt sozusagen "versteckt". Wenn die Zeit, die der Rechenkern für die Berechnung der inneren Knoten benötigt, größer als die für die Kommunikation erforderliche Zeit ist, dann entstehen keine durch die Kommunikation verursachten Wartezeiten [1].
Der Preprozessor von "VirtualFluids" zerlegt das Berechnungsgebiet in so viele Teilgebiete wie Prozessoren zur Simulation vorhanden sind. Da sich die Teilgebiete im Laufe der Simulation nicht ändern, ist es von Anfang an wichtig diese so aufzuteilen, dass alle an der Simulation beteiligten Prozessoren annähernd die gleiche Auslastung (Loadbalancing) haben. Wäre dies nicht der Fall, so würden

die weniger belasteten Prozessoren auf Ergebnisse der anderen Prozessoren warten müssen, was die Performance des Rechners oder des Rechnersystems herabsetzen würde. Das Erreichen der maximalen theoretischen Leistungsfähigkeit der verwendeten Hardware ist aber in jedem Fall anzustreben. Annähernd gleiche Lastverteilung und Minimierung der Interprozesskommunikation ist, unabhängig von der zur Berechnung verwendeten Hardware, stets das zu erreichende Ziel. Deshalb wird im Folgenden nicht weiter auf die Besonderheiten der Hardware eingegangen, sondern ausgewogenes Loadbalancing und Komunikationsminimierung als generelles Problem bzw. Ziel bei parallelen Berechnungen betrachtet [14].

Kapitel 3

Gebietszerleger auf Basis von Graphenmodellen

Im Grunde gibt es nur zwei wesentliche Ansätze durch Algorithmen eine Gebietsaufteilung zu erhalten. Entweder auf der Basis von Graphentheorien oder auf geometrischem Weg. Das Graphenmodell ist universell einsetzbar, weil das Berechnungsgebiet, welches zerlegt werden soll, nicht notwendigerweise geordnet oder koordinatenabhängig vorliegen muss. Der geometrische Ansatz dagegen setzt voraus, dass das zu zerlegende Gebiet in irgendeiner Weise geordnet im Raum liegt.

3.1 Das Standard-Graphenmodell

Die in der Praxis am meisten eingesetzten Gebietszerleger arbeiten mit dem Graphenmodell. Im Folgenden wird die Assoziation zwischen dem Graphenmodell und dem Berechnungsmodell erklärt. Ein Graph, $G = (V, E)$, besteht aus einer Reihe von Knoten (vertices) $V = v_1, v_2, ..., v_n$, und einer Reihe paarweiser Relationen $E \subset V \times V$, die Kanten (edges) genannt werden. Wenn gilt: $(v_i, v_j) \in E$, dann sind die Knoten v_i und v_j Nachbarn. Für unsere Zwecke dienen die Knoten des Graphen zur Datenhaltung für die Simulation. Sie stellen also die diskreten Punkte der numerischen Berechnung dar. Die Kanten verkörpern den Datenfluss, der nötig ist, um den diskreten Wert an einem Knoten zu berechnen. Als Beispiel dient ein Graph, der aus acht Knoten und elf Kanten besteht (Abb. 3.1).

3.2 Zerlegung auf Basis des Standard-Graphenmodells

Der Algorithmus der Berechnung bestimmt die Struktur des gebildeten Graphen. Je mehr Information für die Berechnung eines Knotenwertes von anderen Knoten

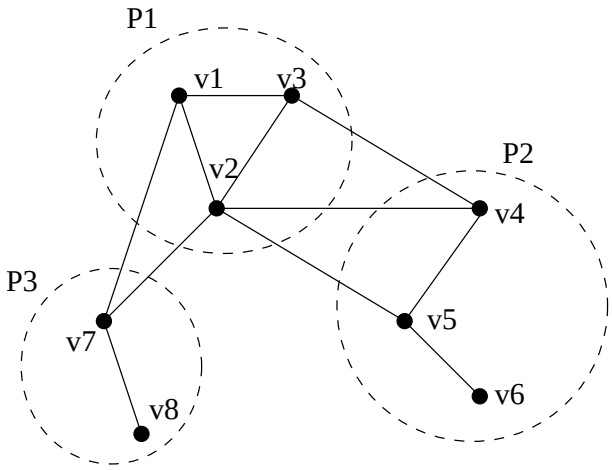


Abbildung 3.1: Das Standard Graphenmodell [4]

erforderlich ist, desto mehr Kanten (Relationen) sind in dem Graphen. Dieser kann dann dazu benutzt werden, eine für einen parallelen Berechnungsalgorithmus effiziente Aufteilung der Gesamtrechenarbeit zu erzeugen. Eine effiziente Aufteilung der Rechenarbeit auf P Prozessoren ist dann erreicht, wenn ein Graph in P gleich große Teile (gleiche Anzahl an Knoten) aufgeteilt wird, wobei die Interprozesskommunikation minimiert wird. Interprozesskommunikation ist immer dann nötig, wenn zur Berechnung eines Knotenwertes Daten von Knoten, die einem anderen Teilgebiet (Partition, Domain) angehören, benötigt werden. Die Kanten, die Knoten aus unterschiedlichen Teilgebieten verbinden, verkörpern im Graphenmodell die Interprozesskommunikation.
Durch diese relativ einfache Assoziation von Berechnungsmodell zum Graphenmodell ist es möglich, sämtliche Arten von Berechnungen, die auf unstrukturierten Gittern basieren, effizient zu verteilen, sofern der Algorithmus im Berechnungskern dies zulässt. Diese umfassen insbesondere "Finite Elemente Methoden", "Finite Differenzen Methoden" und "Finite Volumen Methoden", welche sowohl implizite als auch explizite Berechnungsmodelle verwenden. Auch bei der Simulation neuronaler Netze, Simulationen zu Teilchenbewegungen und der Simulation integrierter Schaltungen wird dieses Verfahren unter Verwendung anderer Berechnungsmethoden erfolgreich eingesetzt [4].
In den letzten Jahren sind im Rahmen von Forschungsprojekten einige Programme und Bibliotheken entstanden, die dieses Verfahren verwenden. Zu ihnen zählt unter anderem Chaco [5] und METIS [10], auf die später noch eingegangen wird.

3.3 Schwächen und Erweiterungen des Standard-Graphenmodells

Es hat sich gezeigt, dass das Modell auch einige Schwächen hat. Im Folgenden werden auf ein paar leicht nachvollziehbare Schwächen und Ansätze zu deren Lösungen hingewiesen.

3.3.1 Der "Edgecut"

Um die Interprozesskommunikation zu minimieren, muss ein Kriterium eingeführt werden, nach dem die Güte der Zerlegung bestimmt werden kann.
Wie in der Abbildung 3.1 zu erkennen ist, sind die Knoten v1, v2 und v3 von Prozessor P1 zu bearbeiten, Knoten v4, v5 und v6 von Prozessor P2 und Knoten v7 und v8 von Prozessor P3. Ein Maß für den Kommunikationsaufwand der Prozessoren untereinander lässt sich an den Kanten abzählen, die die Markierung für die Prozessorzuständigkeit schneiden. Die Anzahl der Überschneidungen nennt man "Edgecut".
Je nachdem welchen Algorithmus man verwendet, kann es vorkommen, dass der Edgecut nicht die tatsächlich notwendige Kommunikation beschreibt. In dem verwendeten Beispiel unterhält der Knoten v7 zwei Beziehungen zu Knoten im Bereich des Prozessors P1. Wenn die beiden Knoten v1 und v2 jeweils die gleichen Daten von v7 erhalten, dann würde es genügen, einmal mit P1 zu kommunizieren, um die erforderlichen Daten auszutauschen. Der "Edgecut" ist also viel höher, als der tatsächliche Kommunikationsaufwand. Berücksichtigt man diesen Umstand und wertet folglich die Beziehung von v7 zu v1 und v2 mit einer einzigen Kommunikation, dann heißt das Ergebnis der Auswertung "Boundarycut". Je nach Algorithmus ist entweder der "Edgecut" oder der "Boundarycut" der realistischere Wert.
Die Zerlegung des Graphen aufgrund der Bewertung des "Edgecut" lässt einige wichtige Kriterien für eine ausgewogene Nutzung des parallelen Rechners außer Acht. Zum einen wird nicht berücksichtigt, dass in einem Cluster nicht alle Prozessoren die gleiche Arbeitsgeschwindigkeit haben müssen und zum anderen wird ebenfalls nicht berücksichtigt, dass die Kommunikationsgeschwindigkeit im Rechnernetzwerk, je nachdem wie die Rechner verbunden sind, unterschiedlich ist. Eine weitere Schwäche bei der Berechnung des "Edgecut" ist, dass entweder auf die Anzahl der erforderlichen Kommunikationseinheiten (Messages) oder auf die Gesamtgröße der erforderlichen Kommunikation optimiert werden kann. Da sich die Geschwindigkeit der Kommunikation sowohl aus Größe der Nachricht als auch aus der Latenzzeit zusammensetzt, wirkt sich eine hohe Anzahl kleiner Nachrichten erheblich auf die Performance aus. Die Latenzzeit ist die Zeit, die nötig ist eine Nachricht "auf den Weg" zu bringen. Je nach verwendeter Architektur ist die Latenzzeit für die Gesamtperformance ausschlaggebender als die

Größe der zu versendenden Nachricht.
Aus den Schwächen der Edgecutmetrik ergibt sich also die Notwendigkeit, den universellen Ansatz der Graphentheorie speziell auf das jeweilige Problem zu optimieren, um eine hohe Performance bei der eigentlichen Simulation oder Berechnung zu erreichen [4].

3.3.2 Gewichtung von Knoten und Kanten

Ist der Rechenaufwand pro Knoten für alle Knoten nicht gleich groß, so können die Knoten verschieden gewichtet werden. Wenn z.B. der Knoten v_i zur Berechnung n Recheneinheiten benötigt und der Knoten v_j m Recheneinheiten, dann kann der Knoten v_i mit $\frac{n}{n+m}$ und der Knoten v_j mit $\frac{m}{n+m}$ gewichtet werden. Um ein ausgewogenes Loadbalancing zu erhalten, sollten die Teilgebiete nicht die gleiche Anzahl an Knoten enthalten. In diesem Fall sollte die Summe der Knotengewichte gleich groß sein.
Ist das Berechnungsgebiet ein unstrukturiertes Gitter, in dem die Abstände der Knoten voneinander nicht konstant sind, dann können die Kanten so gewichtet werden, dass die Differenzen der Knotenabstände mit in das Modell eingearbeitet werden. Die Gewichtung von Kanten wird im Folgenden noch weiter diskutiert.

3.3.3 Gerichtete Graphen

Bisher wird davon ausgegangen, dass die Kommunikation, die im Graphenmodell als Kante abgebildet wird, bidirektional erfolgt, d.h. ein Knoten v_i erhält zur Berechnung einen Datensatz (oder einen Wert) von einem seiner Nachbarn v_j. Im Gegenzug wird ebenfalls ein Wert von v_i an den Nachbar v_j versendet, den dieser zur Berechnung benötigt. Diese Graphen werden "ungerichtete Graphen" genannt. Die Beziehung der Knoten untereinander können in einer Matrix zusammengestellt werden. So eine Matrix wird Adjazenzmatrix genannt. In Tabelle 3.1 werden die Kanten aus dem Graph (Abb. 3.1) zusammengestellt. Es entsteht eine

	1	2	3	4	5	6	7	8
1	x	x	x				x	
2	x	x	x	x	x		x	
3	x	x	x	x				
4		x	x	x	x			
5		x		x	x	x		
6					x	x		
7	x	x					x	x
8							x	x

Tabelle 3.1: Adjazenzmatrix des Graphen Abb.3.1

symetrische Matrix der Dimension 8×8. Die Adjazenzmatrix von ungerichteten Graphen ist immer eine symetrische Matrix.
Es gibt jedoch genügend Algorithmen, die zur Berechnung die Kommunikation in nur eine Richtung notwendig machen. Mit dem Standardgraphenmodell ist es nicht möglich solche Beziehungen zwischen zwei Knoten abzubilden. Man kann demnach das Modell erweitern und "gerichtete Graphen" einführen. In dem Modell der gerichteten Graphen, verkörpert der Knoten unverändert zum Standardmodell die Datenhaltung in der Berechung. Die Kanten stellen jedoch den Fluss der Kommunikation von der Quelle zum Ziel dar. So erweitert, lässt sich das Modell genauer an den Algorithmus anpassen. Zwei Ansätze, die das Modell auf die genannten Bedürfnisse erweitern sind:

- Die Kanten sind per Definition gerichtet z.B. $v_i \rightarrow v_j$. Der zuerst genannte Knoten v_i stellt die Quelle, der Folgeknoten v_j das Ziel der Kommunikation dar. Soll eine Beziehung bidirektional dargestellt werden, dann muss eine weitere Kante erzeugt werden $v_j \rightarrow v_i$, in der der Knoten v_j die Quelle und der Knoten v_i das Ziel ist.

- Oder: Eine gerichtete Kante erhält die Gewichtung 1. Soll eine Beziehung bidirektional erfolgen, erhalten diese Kanten das Gewicht 2.

Im Standardmodell wird ebenfalls nicht berücksichtigt, dass die Größe der Nachrichten für die beiden Kommunikationsrichtungen unterschiedlich sein kann, d.h. das Kommunikationsvolumen ist nicht nur von der Anzahl der Kanten abhängig, sondern auch von der Größe der zu übertragenden Nachricht.
Um einige der Mängel des symetrischen Standardgraphenmodells zu beseitigen, werden die im Folgenden diskutierten Methoden vorgestellt.

3.3.4 Das "bipartite Graph" Modell

Ein "bipartite Graph", $G = (V_1, V_2, E)$, besteht ebenfalls aus Knoten und Kanten, mit dem Unterschied, dass die Knoten in zwei Gruppen V_1 und V_2 unterteilt sind. Eine Kante stellt eine Beziehung zwischen zwei Knoten her, die nicht aus derselben Gruppe sind.
Der Vergleich des Graphen mit dem Berechnungsmodell ist hier jedoch komplizierter als im Standardmodell. Der Einsatz dieses Graphenmodells kommt vorwiegend bei parallelen Algorithmen für Matrix-Vektor Multiplikationen von unsymetrischen Matrizen zum Einsatz. Hier bildet der Graph den Rechenaufwand bei der Multiplikation ab. Die Knoten der Gruppe V_1 repräsentieren dabei die Zeilen der Matrix. Die Knoten der Gruppe V_2 bilden die Spalten der Matrix. Bei einer $m \times n$ Matrix A besteht die Gruppe V_1 aus m Knoten und die Gruppe V_2 aus n Knoten. Eine Kante kommt dann zustande, wenn die Matrix A an der Stelle a_{ij} mit "nicht Null" besetzt ist. Die Kanten bilden somit den Rechenaufwand der Multiplikation ab (siehe Abb. 3.2). Bei einer Minimierung des "Edgecut" solcher Graphen kann die Matrix effizient in Teilmatrizen zerlegt werden [6].

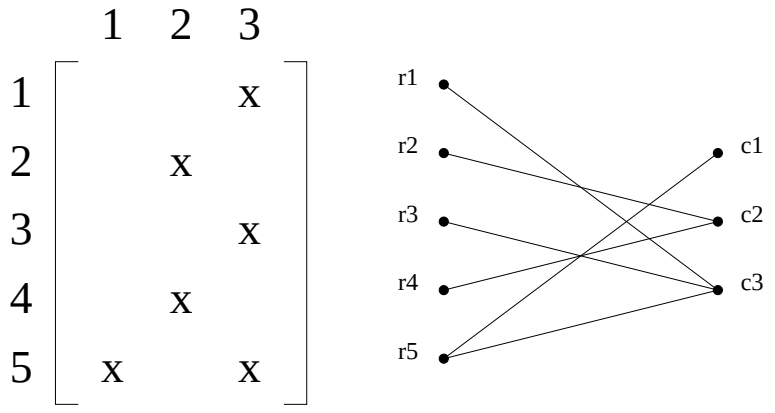


Abbildung 3.2: Unsymetrische $m \times n$ Matrix und zugehöriger "bipartite Graph" [4]

3.3.5 Das "Hypergraph" Modell

Mit dem Modell des Hypergraphen, versucht man die oben beschriebenen Probleme des "Edgecut" bzw. "Boundarycut" auf elegante Weise zu lösen. Ein Hypergraph $G = (V, H)$ besteht aus einer Menge an Knoten V und sog. "Hyperedges" H. Eine Hyperedge ist die Menge an Knoten, die mit einem Knoten aus der Menge V Kanten bilden. Ein Standardgraph ist also eine Untermenge der Hypergraphen, mit der Einschränkung, dass Kanten durch genau zwei Knoten gebildet werden können. Optimales Loadbalancing in parallelen Algorithmen wird auch hier durch

gleich große Gruppen von Knoten erreicht. Die Minimierung der Kommunikation ist dann erreicht, wenn möglichst wenige Hyperedges auf Knoten anderer Prozessorbereiche vorhanden sind. Im Gegensatz zum Standardgraphenmodell ist der "Edgecut" in diesem Modell gleich dem "Boundarycut".

3.3.6 Multi-Constraint und Multi-Objective Partitioning

Die Multi-Constraint Methode findet vor allem dann Anwendung, wenn k verschiedene Rechenoperationen (Phasen) nacheinander ausgeführt werden sollen und diese unterschiedlichen Rechenaufwand benötigen. Die Knoten bilden dabei, wie in den Modellen vorher, die Menge der durchzuführenden Rechenoperationen und sind jetzt mit einem Array verknüpft, welches den Rechenaufwand der k Phasen für jeden Knoten durch gewichtete Werte enthält. Eine effiziente Aufteilung ist dann erreicht, wenn jede der k Phasen gut balanciert ist, und die Kommunikation dabei minimal ist. Mit dieser Methode kann ebenfalls eine Berechnung zur Optimierung des "Boundarycut" erreicht werden. Dazu erhält ein Knoten zwei Phasen, wobei die erste Phase den Rechenaufwand pro Knoten und die zweite Phase die für den "Boundarycut" zuständige Eigenschaft abbildet.
Die Multi-Objective Methode ähnelt der Multi-Constraint Methode. In diesem Modell werden jedoch jeder Kante ein Array für die Gewichtung von verschiedenen Phasen zugeordnet. Die im Array stehenden Werte können jeweils verschiedene Arten an Ressourcenbedarf bedeuten. Es kann also auch eine Gewichtung für den Rechenaufwand der Knoten in den Arrays enthalten sein. Ziel einer erfolgreichen Zerlegung ist hier ein ausgewogenes Loadbalancing durch bilden gleich großer Gruppen mit Knoten und eine Minimierung des "Edgecuts" in jeder Phase.

3.3.7 Skewed partitioning

Diese Methode ist eher eine andere Sichtweise der effizienten Zerlegung als eine Alternative zu den vorher genannten Modellen. Soll ein Graph in p Teile zerlegt werden, so erhält jeder Knoten ein Array der Größe p. Jeder der p Werte im Array stellt dar, wie "wohl" sich ein Knoten in der jeweiligen Gruppe fühlen würde. Die Startwerte werden aufgrund des Kommunikationsaufwandes ermittelt. Sinnvoll ist diese Art der Bewertung, wenn z.B. ein Berechnungsgebiet nach einer erfolgreichen Aufteilung durch einen der bereits vorgestellten Methoden erfolgt ist und im folgenden Rechenschritt, durch eine adaptive Veränderung im Algorithmus, eine Rekalibrierung der Loadbalance notwendig wird. Da sich die Teilgebiete nach der Rekalibrierung ähnlich sind, ist es von Vorteil zu wissen, in welcher Gruppe der Knoten sich vorher befunden hat, und im nächsten Schritt sinnvoll untergebracht wäre. Diese Methode wird in der Praxis vor allem im Bereich integrierter Schaltungen angewendet, um die Leiterbahnen zwischen den Bauteilen möglichst kurz ausführen zu können.

3.4 Die Algorithmen der Zerleger

Egal welches Graphenmodell verwendet wird, ob Standard oder eine der beschriebenen Erweiterungen, müssen Algorithmen für eine effiziente Aufteilung in Teilbereiche bereitstehen. Im Laufe der Zeit hat sich der Ansatz der Multilevel - Algorithmen als der vielversprechendste herauskristallisiert, da diese stabil laufen und für viele Anwendungsbereiche effiziente Ergebnisse liefern. Die Zerleger in METIS [10] und Chaco [5] arbeiten ebenfalls nach dieser Methode. Das grundsätzliche Vorgehen eines Multilevel - Algorithmus besteht darin, dass ein großer Graph auf einen kleineren Graph so abgebildet wird, dass die charakteristischen Eigenschaften des großen Graphen erhalten bleiben. Dieser Vorgang wird mehrfach wiederholt. Der kleinste Graph wird in die gewünschte Anzahl an Teilgebieten zerlegt. Danach wird jedes Teilgebiet Schritt für Schritt zurück in den großen Graphen übergeführt und dabei die Zugehörigkeit der Knoten in die Teilgebiete verfeinert.
Man benötigt für so einen Algorithmus drei im wesentlichen unabhängig voneinander agierende Programme:

- Ein Programm, welches einen großen Graphen sequentiell auf kleinere Graphen abbildet, wobei die Gewichtungen erhalten bleiben bzw. erzeugt werden (Phase 1). Dieser Abschnitt wird "Coarsening Phase" genannt.
- Ein Programm, welches einen Graphen aus gewichteten Knoten und Kanten in beliebig viele Teilgraphen zerlegen kann. Dieser Teil wird mit "Partitioning Phase" (Phase 2) bezeichnet.
- Ein Programm, welches den zerlegten Graphen wieder auf den nächst größeren Graphen abbildet und dabei idealer Weise die Kriterien zur optimalen Zerlegung verfeinert. Die "Uncoarsening Phase" (Phase 3) (siehe Abb.3.3).

3.4.1 Coarsening Phase

Der ursprüngliche Graph $G_0 = (V_0, E_0)$ wird sequentiell in kleinere Graphen $G_1, G_2, ...G_m$ umgewandelt, so dass die Anzahl der Knoten mit jeder Instanz geringer wird $|V_0| > |V_1| > |V_2| > ... > |V_m|$. Es gibt mehrere Ansätze, wie die Verkleinerung erreicht werden kann. Als Beispiel wird lediglich der "Random Matching (RM)" Algorithmus erläutert. Weitere Algorithmen z.B. "Heavy Edge Matching (HEM)", "Light Edge Matching (LEM)" oder "Heavy Clique Matching (HCM)" sind in [7] zu finden.
Der Algorithmus RM läuft wie folgt ab:
Ein Knoten u wird per Zufall ausgewählt. Wurde dieser Knoten noch nicht bearbeitet, dann wird ein ebenfalls noch nicht bearbeiteter Nachbarknoten v per Zufall bestimmt. Falls ein solcher existiert, werden die Beziehung der beiden Knoten notiert und durch einen einzigen Knoten ersetzt. Die Knoten u und v werden

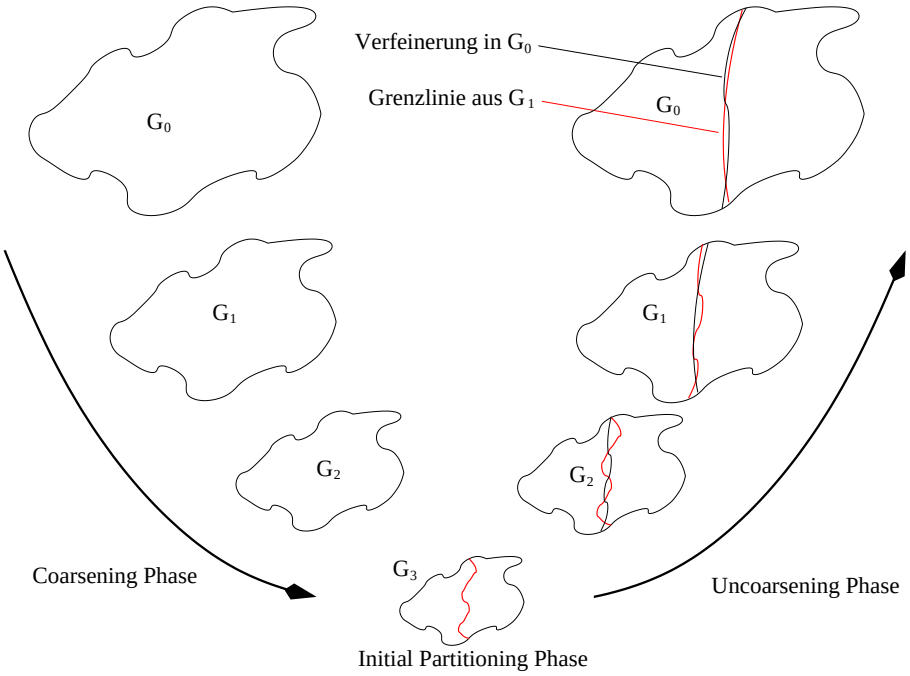


Abbildung 3.3: Schema eines bisectionalen Multilevel Algorithmus [7]

als bearbeitet markiert. Wird kein Nachbarknoten gefunden, der als nichtbearbeitet markiert ist, dann bleibt der Knoten u als nichtbearbeitet markiert. Der Aufwand des Algorithmus ist $O(|E|)$.

3.4.2 Partitioning Phase

In der Phase der Zerlegung werden zahlreiche, sich oft deutlich voneinander unterscheidende Algorithmen verwendet. Die erste Teilung wird Initialteilung genannt. Es kommen sowohl Algorithmen zum Einsatz, die den Graphen in zwei Teile (spectral bisection) zerlegen, als auch Algorithmen, die den Graphen direkt in k Teile (Multi-Level k-way-Partitioning) aufteilen. Ein Graph kann in k Teilgraphen geteilt werden, indem z.B. die Coarsening Phase so lange fortgeführt wird, bis nur noch k Knoten übrig sind. Die Uncoarsening Phase kann dann aber wesentlich aufwendiger werden, weil die initiale Teilung durch die geringe Anzahl an Knoten nicht besonders gut balanciert sein kann. Eine andere Möglichkeit, die z.B. in METIS und ParMETIS Anwendung findet, ist der Algorithmus nach Kernighan-Lin [15]. Hier wird die bestmögliche Teilung eines Graphen iterativ berechnet. Per Zufall werden mehrere Initialteilungen erzeugt und der Edgecut berechnet. Die beste zufällig erzeugte Teilung ist die Basis für die weiteren Iterationen. In jedem Iterationsschritt werden per Zufall Knoten in andere Teilgebiete verschoben und die Änderung des Edgecut ausgewertet. Verbessert sich der

Edgcut, so wird der Iterationsschritt durchgeführt. Führt die Verschiebung des Knotens zu einem schlechteren Ergebnis, so wird diese rückgängig gemacht. Jeder Iterationsschritt hat eine Ordnung (Aufwand) von $O(|E|log|E|)$. Startet der Algorithmus mit einer guten Initialteilung, so sind nur wenige Iterationen nötig. Durch eine Optimierung der Datenhaltung kann eine höhere Performance erreicht werden. Wie in [16] beschrieben ist, kann dadurch der Aufwand auf $O(|E|)$ reduziert werden. Die Funktionsweise weiterer Algorithmen sind in [7] und [8] genauer beschrieben.

3.4.3 Uncoarsening Phase

In der "Uncoarsening Phase" (oder auch "Refinement Phase") werden ausgehend vom Graph G_m die in der "Coarsening Phase" berechneten Vergröberungen des Graphen Schritt für Schritt rückgängig gemacht. Nach jedem Schritt sorgt z.B. der Kerighan-Lin Uncoarsening Algorithmus dafür, dass die Zugehörigkeit der durch die "Uncoarsening Phase" quasi neu entstandenen Knoten optimiert wird. Ähnlich wie in der "Partitioning Phase" werden jetzt Knoten per Zufall zwischen den Teilbereichen ausgetauscht und der Edgecut ermittelt. Falls sich dieser verbessert, werden die Knoten verschoben. Um das Loadbalancing weiterhin zu gewährleisten, müssen in den beiden, zum Knotentausch ausgewählten, Teilbereichen die gleiche Anzahl an Knoten zum Austausch zur Verfügung stehen. Da der Graph in jedem Schritt der "Uncoarsening Phase" optimiert wird, sind immer gute initiale Teilbereiche vorhanden, so dass die Optimierung bereits nach wenigen Iterationsschritten abgebrochen werden kann [7] [8].

3.5 Anpassung an die verwendete Hardware

Nach wie vor lässt sich heterogene Hardware schlecht in den Graphenmodellen darstellen. Die zur Zeit beliebten Linux-Cluster bestehen meistens aus PCs, die jedoch nicht alle die exakt gleiche Hardware besitzen. Einige der im Cluster verwendeten PCs können aus SMP-Maschinen bestehen. SMP-Maschinen sind Rechner mit mehr als einem Prozessor, die auf einen gemeinsamen Hauptspeicher zugreifen (z.B. DUAL-Prozessor Maschinen). Innerhalb der SMP-Maschine würde die Kommunikation zwischen den Prozessoren erheblich schneller ablaufen, als die Kommunikation zwischen den Knoten, die über die Netzwerkinfrastruktur abgewickelt werden muss. Das gleiche Problem besteht bei der Verwendung der Hitachi SR8000-F1, bei der jeweils 8 Prozessoren zu SMP-Knoten mit Shared-Memory zusammengefasst sind [22]. Die Kommunikation innerhalb dieser SMP-Knoten ist schnell im Vergleich zur Kommunikationsgeschwindigkeit von Knoten zu Knoten. Solche Maschinen optimal auszulasten ist Gegenstand aktueller Forschung.

3.6 Software am Beispiel von METIS und ParMETIS

Das Softwarepaket METIS [10] wurde im November 1998 in der noch immer aktuellen Version 4.0.1 herausgegeben. METIS ist eine Sammlung von Algorithmen und Programmen zur Zerlegung unstrukturierter Gitter, Finite-Elemente Netze und zur Umsortierung von Matrizen. Der Quellcode ist frei verfügbar, auf mehreren Plattformen getestet und wird noch immer gepflegt. Das Paket liefert sowohl fertige Programme zur Zerlegung als auch die Möglichkeit, die Zerleger durch Library-Funktionen direkt in eigene Programme einzubauen.
Das Softwarepaket ParMETIS [11] wurde im März 2002 in der Version 3.0.0 freigegeben. ParMETIS ist die parallele Version von METIS und steht als Sammlung von Library-Funktionen auf MPI-Basis zur Verfügung. Der Quellcode und die Nutzung ist für Forschungszwecke oder Evaluierungen frei verfügbar und somit auf eine Reihe verschiedener Plattformen portierbar.

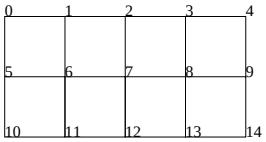
3.6.1 Interfaces von METIS bzw. ParMETIS

Die im Paket METIS enthaltenen Stand-Alone Programme bieten die Möglichkeit, ohne vertieften Einstieg in die Graphentheorie, die Zerleger zu nutzen. Dazu müssen jedoch Eingabe-Dateien erzeugt werden, die einem bestimmten Format entsprechen. Dieses Format ist im mitgelieferten Handbuch [10] ausführlich beschrieben. Die Stand-Alone Programme sind zu Testzwecken oder zum Einstieg äußerst hilfreich, da aber bei der Erzeugung der Eingabe-Dateien zusätzlicher I/O entsteht, ist diese Methode aufgrund mangelnder Performance nicht zu empfehlen.
Wesentlich schneller und flexibler ist es, die Library-Funktionen direkt zu nutzen. Außerdem stehen mit der Library zusätzliche Funktionen zur Verfügung, mit denen z.B. ungleichmäßige Zerlegungen nach dem Muster

- Prozessor 0: 10%,
- Prozessor 1: 30%
- und Prozessor 2: 60% Lastanteil an der Gesamtrechenlast

erzeugt werden können. Um die Library-Funktionen nutzen zu können, muss der Graph ebenfalls in einem bestimmten Format an die Funktionen übergeben werden. METIS benutzt dazu das CSR-Format (compressed storage format). Ein Graph wird durch eine Reihe Arrays abgebildet. In diesem Format ist es auch möglich, Gewichtungen für Knoten und Kanten zu definieren. Da dieses Format ein wesentlicher Bestandteil der Implementierung von METIS oder ParMETIS in eigene Programme ist, wird dieses näher erklärt.

Die Struktur des Graphen mit n Knoten und m Kanten wird in zwei Arrays *xadj[n+1]* und *adjncy[2m]* abgebildet. $2m$ deshalb, weil die Kante zwischen Knoten u und Knoten v sowohl als Kante (u, v) als auch als Kante (v, u) gespeichert wird. Die Adjazenzstruktur des Graphen wird wie folgt gespeichert. Davon ausgehend, dass die Nummerierung der Knoten bei Null beginnt (wie in C üblich), werden die Nummern der Knoten, zu denen ein Knoten i in Beziehung steht, im Array *adjncy*, beginnend mit dem Index *xadj[i]*, und endend, aber nicht eingeschlossen, mit dem Index *xadj[i+1]* gespeichert. Die Knotennummern aus dem Array *adjncy* ab dem Index *xadj[i]* bis einschließlich *xadj[i+1]-1* sind Knoten, die mit dem Knoten i eine Kante bilden (siehe Abb. 3.4). Die Speicherung von



xadj 0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44

adjncy 1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

Abbildung 3.4: Beispielgraph und zugehörige Adjazenz-Arrays [10]

Gewichtungen der Knoten und Kanten wird durch zwei weitere Arrays *vwgt[n]* und *adjwgt[2m]* erreicht. Die Gewichtung des Knotens i wird auf *vwgt[i]* abgespeichert, die Gewichtung einer Kante *adjncy[j]* auf *adjwgt[j]*. Diese Arrays werden dann den METIS-Funktionen als Parameter übergeben.
Da ParMETIS nur Library-Funktionen zur Verfügung stellt, fällt die ohnehin nicht zu empfehlende Weise auf Eingabedateien auszuweichen, weg. Da die Arrays von Anfang an auf alle an der Zerlegung beteiligten Prozessoren verteilt werden, muss für die parallele Version das CSR-Format geringfügig erweitert werden. Es kommt zu den beiden Arrays *xadj* und *adjncy* ein drittes Array *vtxdist[P+1]* hinzu. P ist die Anzahl der Teilgebiete ($\leftrightarrow$ Prozessoren), in die zerlegt werden soll. In *vtxdist* wird gespeichert welcher Prozessor für welchen Knoten die Adjazenzinformationen hält. Der Prozessor i hält die Adjazenzinformationen für die Knoten *vtxdist[i]* bis ausschließlich *vtxdist[i+1]*. Für den Graphen aus Abbildung 3.4 sehen die Arrays wie folgt aus (siehe Abb.3.5): Die Berücksichtigung von gewichteten Knoten und Kanten erfolgt analog METIS. Die beiden zusätzlichen Arrays werden ebenfalls auf die Prozessoren verteilt. Falls keine Gewichtung nötig ist, können die Arrays weggelassen werden. Die Syntax der Arrays ist auch im Handbuch genau erläutert [11].
Die erzeugten Arrays müssen auf die Prozessoren verteilt oder dort erzeugt werden. Sie sind wie in der seriellen Version als Parameter an die Library-Funktionen zu übergeben. Als Ergebnis erhält man jeweils ein Array, welches jedem Knoten

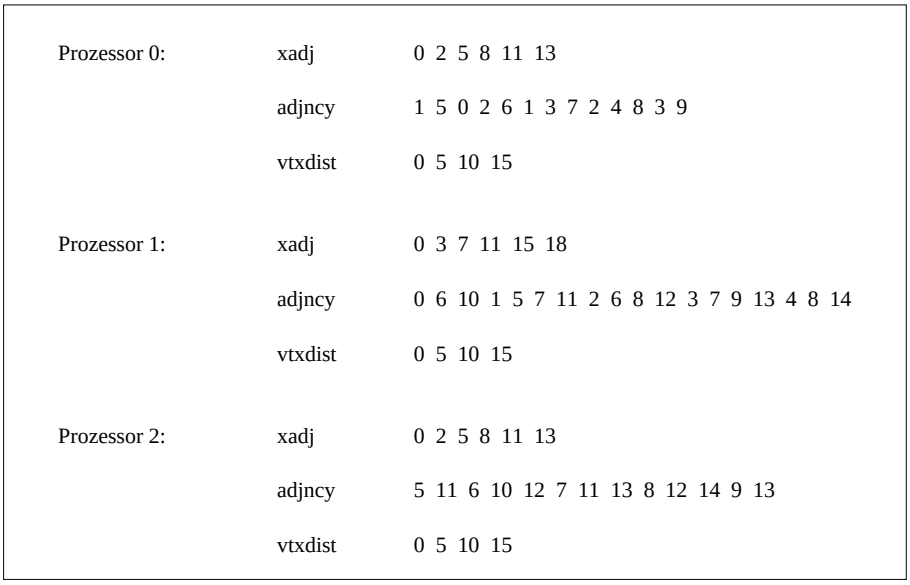


Abbildung 3.5: Verteilte Adjazenz-Arrays [11]

die Nummer seines Teilgraphen zuweist. In der parallelen Version ist das Ergebnisarray auch auf die Prozessoren verteilt und muss zur Auswertung an den Hauptprozessor geschickt werden.

3.6.2 Implementierung von ParMETIS in den Preprozessor der Simulation VirtualFluids

MPI [12] bedeutet "Message Passing Interface" und ist ein offener Standard. In diesem Standard wird unter anderem festgelegt, auf welche Weise Datenpakete von Rechner zu Rechner übertragen werden. Programm-Bibliotheken nach dem MPI-Standard sind in zahlreichen Implementierungen für verschiedene Architekturen vorhanden. Die bekanntesten sind die als Open-Source zur Verfügung stehenden Pakete MPICH [19] und LAM-MPI [20]. Die Algorithmen in ParMETIS sind so programmiert, dass sie sowohl parallel als auch seriell funktionieren. Eine Implementierung mit ParMETIS kann demnach auf einem oder mehreren Prozessoren ausgeführt werden. Im Lattice-Boltzmann Kontext liegt das Berechnungsgebiet, welches der Preprozessor erzeugt, in Form einer 3D-Matrix aus Integer-Werten im Hauptspeicher vor. Die 3D-Matrix bildet ein strukturiertes Gitter. Die Knoten sind durch Koordinaten adressiert. Die Koordinaten entsprechen den Indizes, mit denen auf die Matrix-Struktur zugegriffen werden kann. Die Integer-Werte der Matrix stellen die Eigenschaften des jeweiligen Knotens dar. Es können entweder Fluid-Knoten, Knoten außerhalb des Fluids oder Grenzknoten sein. Im Berechnungskernel des Simulators werden nur Fluidknoten und die

Grenzknoten verwendet. Sämtliche Knoten, die als "Nicht-Fluid" gekennzeichnet sind, werden im Simulationskern nicht benötigt. Diese Knoten bilden die "Löcher" in der 3D-Matrix. Die restlichen Knoten der 3D-Matrix müssen auf die Adjazenz-Arrays, die den Graphen beschreiben, abgebildet werden. Die Kanten, die bei der Simulation den Datenaustausch verkörpern, ergeben sich aus dem Algorithmus des Berechnungskernels. Mit dem Lattice-Boltzmann-Verfahren ergeben sich die Nachbarn eines Knotens wahlweise aus dem "d3q15"(Abb. 3.7) oder "d3q19" (Abb. 3.8) Modell [13]. In dem "d3q15" Modell bilden die Nachbarn nach Tabelle 3.3 Kanten. Bei "d3q19" sind die in Tabelle 3.4 aufgeführten Nachbarknoten, Kanten im Graph, mit denen während der Simulation Daten ausgetauscht werden müssen. Neben den bereits erwähnten geometrischen Modellen "d3q15" und "d3q19" für die 3D-Simulationen wird in 2D-Simulationen das Modell "d2q9" verwendet (Abb. 3.6, Tab. 3.2).

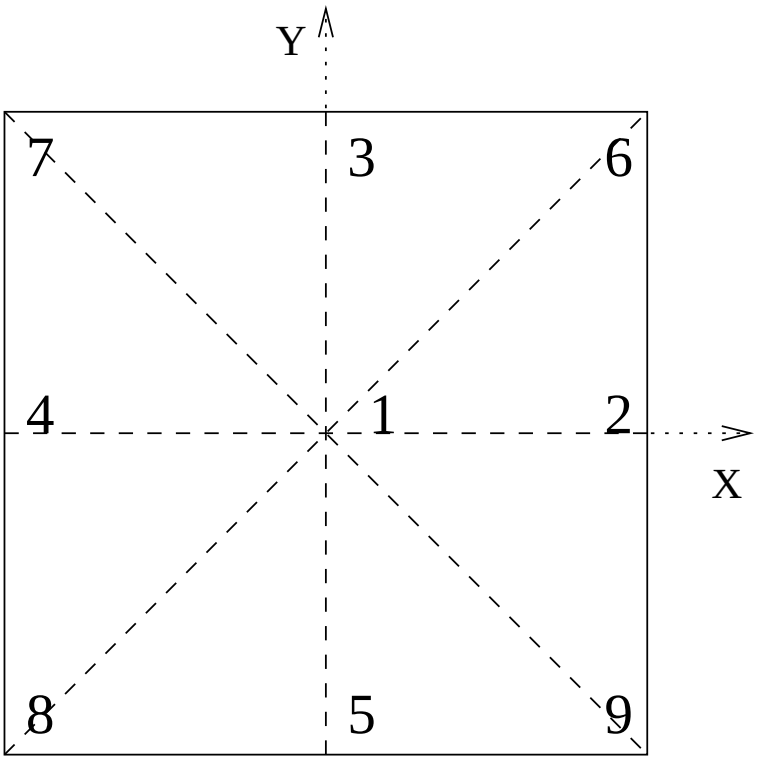


Abbildung 3.6: Modell d2q9

Richtung	1	2	3	4	5	6	7	8	9
x-Richtung	0	1	0	-1	0	1	-1	-1	1
y-Richtung	0	0	1	0	-1	1	1	-1	-1

Tabelle 3.2: Modell d2q9

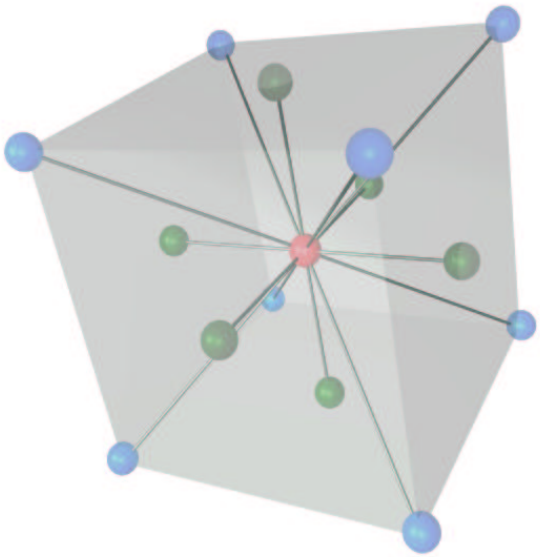


Abbildung 3.7: Modell d3q15

Richtung	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x-Richtung	0	1	-1	0	0	0	0	1	-1	1	-1	1	-1	1	-1
y-Richtung	0	0	0	1	-1	0	0	1	-1	1	-1	-1	1	-1	1
z-Richtung	0	0	0	0	0	1	-1	1	-1	-1	1	1	-1	-1	1

Tabelle 3.3: Modell d3q15

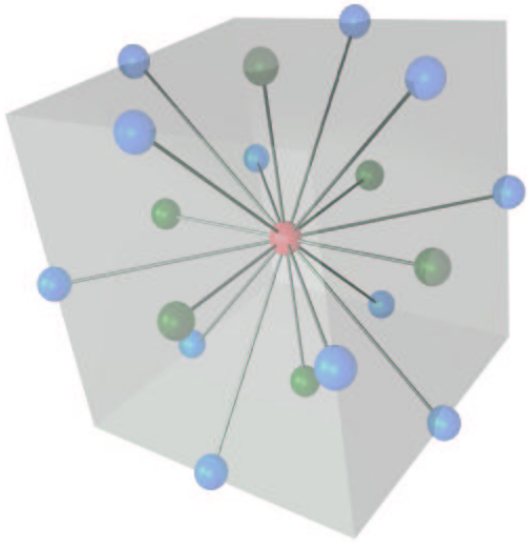


Abbildung 3.8: Modell d3q19

Richtung	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
x-Richtung	0	1	-1	0	0	0	0	1	-1	1	-1	1	-1	1	-1	0	0	0	0
y-Richtung	0	0	0	1	-1	0	0	1	-1	-1	1	0	0	0	0	1	-1	1	-1
z-Richtung	0	0	0	0	0	1	-1	0	0	0	0	1	-1	-1	1	1	-1	-1	1

Tabelle 3.4: Modell d3q19

Die Nachbarbeziehungen im Berechnungskernel von Virtualfluid sind gleichwertig (bidirektional), d.h. der Datenaustausch ist in allen Kanten gleich groß. In jeder Kante wird pro Rechenschritt genau ein Wert von Knoten a zu Knoten b übertragen und umgekehrt. Folglich ist die Adjazenzmatrix des Graphen symetrisch. In der Adjazenzstruktur, wie sie in den Adjazenzarrays des CSR-Formats verwendet wird, sind die Graphen als gerichtete Graphen gespeichert. Da der Graph symetrisch ist, müssen die Kanten sowohl als Kante (a, b) als auch als Kante (b, a) zwischen den Knoten a und b gespeichert werden.
Um die Arrays aufzubauen wird folgendermaßen vorgegangen:

- Alle Knoten, die die Eigenschaft Fluid (also auch Grenzknoten) haben, werden gezählt und numeriert.
- Das Array *vtxdist* wird erzeugt, die Werte für das Array werden berechnet und das Array auf alle beteiligten Prozessoren verteilt.
- Der Reihe nach werden alle Arrays *xadj* und *adjncy* für jeden Prozessor vom Hauptprozessor erzeugt und an die Unterprozessoren verteilt. Dabei werden die Knoten in der 3D-Matrix der Reihe nach durchlaufen und die nach "d3q15" oder "d3q19" in Frage kommenden Nachbarn abgeprüft und gezählt. Im zweiten Durchlauf werden die Kanten im Array *adjncy* festgehalten. Anschließend werden beide Arrays an die Unterprozessoren versendet.

Es muss kein komplettes Array, welches den Gesamtgraphen beschreibt, erzeugt werden. Eine erhebliche Menge Hauptspeicher kann gespart werden, wenn die Teilarrays nach der Erzeugung sofort an die Unterprozessoren versendet und der Speicher sofort wieder freigegeben wird.
Anschließend wird eine Funktion aus der ParMETIS-Bibliothek aufgerufen, die den Graphen zerlegt. Man erhält als Ergebnis ein Array *part*, welches jedem Knoten einen Wert zwischen Null und der Anzahl an Prozessoren P zuweist. Dieses Array muss aus den Teilarrays, die analog *xadj* nach der Verteilung aus *vtxdist* auf die Prozessoren verteilt sind, zusammengebaut werden. Nachdem die Teilarrays zum Hauptprozessor gesendet sind, wird die in *part* geregelte Zugehörigkeit auf die 3D-Matrix abgebildet.
Der Preprozessor erzeugt nun die Eingabedateien für den "VirtualFluids" Simulator.

3.6.3 Resourcenbedarf von METIS bzw. ParMETIS

Um den Resourcenbedarf von METIS zu bestimmen, wurde mit Hilfe von Testzerlegungen der Speicherbedarf gemessen. Bearbeitet wurden würfelförmige Berechnungsgebiete mit den Kantenlängen von 50 bis 140 Knoten. In den folgenden Diagrammen (Abb. 3.9) ist der Speicherbedarf in Bezug auf die Anzahl der Knoten bzw. Kanten aufgetragen. Aus dem Diagramm wird ein linearer Zusammenhang zwischen Speicherverbrauch und der Anzahl der Knoten und Kanten deutlich.

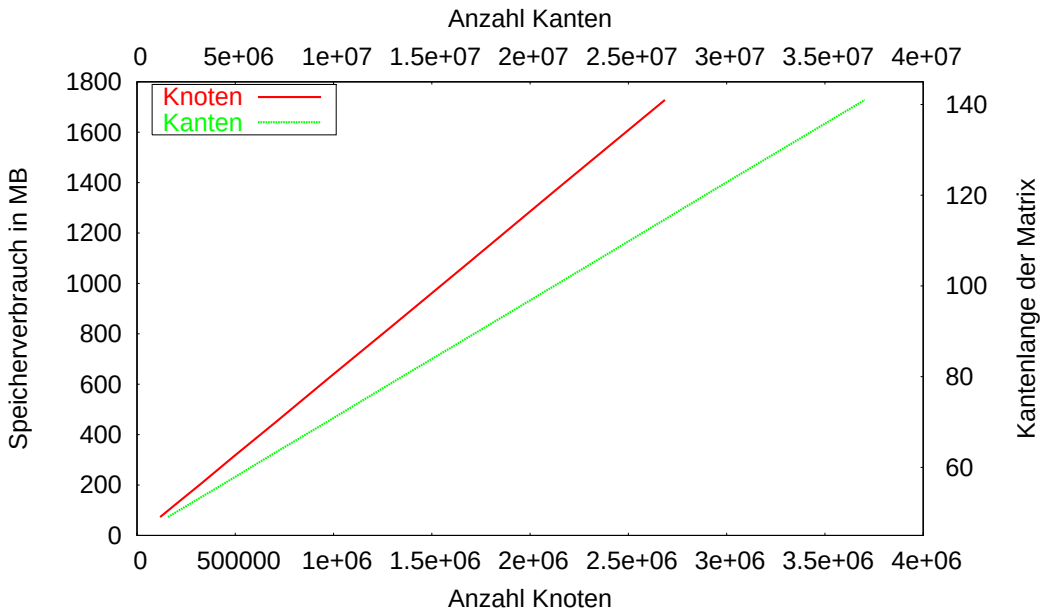


Abbildung 3.9: Speicherbedarf METIS

Kapitel 4

Gebietszerleger auf Basis geometrischer Ansätze

Im Preprozessor wird die Struktur des durchströmten Mediums in Form einer 3D-Matrix gehalten. Jeder Knoten ist durch seine Indizes in der 3D-Matrix festgelegt. Die Indizes bilden die Koordinaten der Knoten. Dieser Umstand macht es möglich, eine Zerlegung mit einem geometrischen Ansatz zu implementieren. Bei dieser Methode wird das 3D-Gitter nicht wie in den graphenorientierten Methoden auf einen Graphen abgebildet. Die Zerlegung findet mit Hilfe von Berechnungen auf Basis der vorliegenden 3D-Matrix statt.
Das Ziel der Zerlegung ist nach wie vor eine in Teilbereichen zerlegte 3D-Matrix. Alle Teilbereiche sollten möglichst die gleiche Anzahl an Knoten besitzen und die Anzahl der bereichsübergreifenden Kanten sollte möglichst gering sein.

4.1 Geometrische Algorithmen

In dem Softwarepaket "Chaco" sind vier geometrisch basierte Zerlegealgorithmen enthalten. Diese sind jedoch relativ einfach, sodass die Qualität der erzeugten Teilgebiete je nach Anwendungsfall sehr unterschiedlich sein kann.

1. Lineares Schema: Hier wird das Gesamtgebiet so zerlegt, dass die Teilgebiete die gleiche Anzahl an Knoten enthalten. Dazu wird der Quotient aus Gesamtanzahl der Knoten v und der Anzahl an zu erzeugenden Teilgebieten p berechnet. Den Knoten wird der Reihe nach bis zum Knoten $\frac{v}{p}$ das erste Teilgebiet zugewiesen, den folgenden Knoten das nächste Teilgebiet. Diese Methode kann trotz ihrer Einfachheit sehr gute Teilungen erzeugen, vor allem dann, wenn die Knoten in einer für die Zerlegung günstigen Reihenfolge für die Verteilung abgearbeitet werden. Da die Knoten oft durch Koordinaten adressiert werden, ist eine günstige Reihenfolge durchaus sehr wahrscheinlich. Die Berechnungsgebiete mit den als X-Slice, Y-Slice und Z-Slice gekennzeichneten Berechnungsergebnissen, sind mit diesem Algorith-

mus zerlegt worden. Dabei wurde z.B. bei der X-Slice-Methode die Knoten von $x = x_{min}$ bis $x = x_{max}$ als außenliegende Schleife und die anderen Koordinaten als innenliegende Schleifen abgearbeitet. In den anderen Algorithmen kennzeichnet die Koordinate ebenfalls die außenliegende Schleife. Je nach Geometrie ist die Qualität der Zerlegung unterschiedlich.

2. Zufallbasiertes Schema: Knoten werden per Zufall auf die Teilgebiete verteilt, so dass am Ende ein ausgewogenes Loadbalancing erreicht wird.

3. Scatterd Methode: Die Knoten werden der Reihe nach dem Teilgebiet zugewiesen, welches die wenigsten Knoten hat.

4. Inertial Methode: Diese Methode teilt das Teilgebiet in Scheiben längs einer Achse der 3D-Matrix auf. Jede der Scheiben erhält dabei annähernd die gleiche Anzahl an Knoten [5].

Eine weitere in "Chaco" und "METIS" nicht implementierte, aber wegen ihrer Einfachheit sehr beliebte Methode eine gut balancierte Zerlegung zu erreichen, ist die Implementierung eines sog. Greedy-Algorithmus. Die Greedy-Algorithmen werden in zahlreichen Problemsituationen verwendet, in denen es gilt, Optimierungen durchzuführen. Diese Algorithmen funktionieren immer nach dem Prinzip, dass das globale Optimum, durch eine Ausweitung eines lokalen Optimums erreicht wird. Für das konkrete Problem der Gebietszerlegung könnte ein Algorithmus wie folgt ablaufen:

- Es werden zufällig soviele Knoten ausgewählt, wie Teilgebiete erstellt werden sollen.

- Jeder dieser Startknoten fügt einen seiner gültigen Nachbarknoten seinem Teilgebiet hinzu.

- Hat der Startknoten keine Nachbarn, die er seinem Teilgebiet hinzufügen kann, wird der als erstes hinzugefügte Knoten der neue Startknoten für das Teilgebiet.

- Der Vorgang wird solange wiederholt, bis alle Knoten einem Teilgebiet zugewiesen sind, oder die Knoten der Teilgebiete keine Knoten mehr erreichen können, die noch keinem Teilgebiet angehören.

Werden die Teilgebiete abwechselnd durchlaufen, so entstehen annähernd gleich große Teilgebiete.
Die genannten Methoden produzieren sehr schnell Teilgebiete, die die Zerlegung mehr oder weniger auf Loadbalancing optimieren. Die Minimierung der Kommunikation wird bisher ausser Acht gelassen. Um dieses Manko zu reduzieren, wurde im Rahmen dieser Arbeit ein geometriebasierter Zerleger entwickelt, der sowohl Loadbalancing als auch Kommunikationsminimierung berücksichtigt. Alternativ

könnte ein durch die Inertial Methode zerlegtes Gebiet durch die Kernighan-Lin Refinement Algorithmen [15] nachträglich auf Kommunikationsminimierung optimiert werden. Da dies aber die Umsetzung der 3D-Matrix in einen Graphen erfordert, wird der Performancegewinn, der durch die geometrische Zerlegung entsteht, relativiert.

4.2 Implementierung eines geometriebasierten Zerlegers: divide

Das im Rahmen dieser Arbeit entstandene Programm "divide" basiert auf einem rekursivem bisectionalen Algorithmus; d.h. das Gesamtgebiet wird zunächst in zwei Teile geteilt (bipartitioniert), die anschließend, jeweils unabhängig voneinander, wieder in zwei Teile geteilt werden usw. Dadurch sind 3D-Gitter in 2^n Teilgebiete zerlegbar.
Die Entscheidung für die Implementierung eines geometrisch basierten Zerlegers, wurde davon beeinflusst, dass sich zum einen die Portierung von METIS auf die Architektur der Hitachi SR-8000 F1 als äusserst schwierig herausgestellt hat, und noch nicht abgeschlossen ist. Zum anderen dadurch, dass die graphenorientierten Methoden mit wachsender Knotenzahl extrem viel Hauptspeicher benötigen. Die geometrischen Methoden sind, was Speicherverbrauch angeht, wesentlich anspruchsloser.
Der entwickelte Algorithmus kann in vier Phasen unterteilt werden:

- Phase 1 (4.2.1): Für jede Koordinatenrichtung wird berechnet, an welcher Stelle eine Teilung erfolgen sollte, um ein optimales Loadbalancing zu erhalten.
- Phase 2 (4.2.2): Für jede Koordinatenrichtung wird berechnet, wie viele Verbindungen nach dem d3q15 oder d3q19 Modell in jedem möglichen Schnitt vorhanden sind.
- Phase 3 (4.2.3): Für jede Koordinatenrichtung werden die Ergebnisse aus Phase 1 und 2 kombiniert.
- Phase 4 (4.2.4): Entscheidung für eine der in Phase 1 - 3 berechneten Koordinatenrichtungen und Teilung des Graphen.

Das Vorgehen des Algorithmus wird im folgenden 2D-Beispiel erklärt.

4.2.1 Phase 1: Loadbalancing in 2D

Das gesamte Berechnungsgebiet besteht in dem verwendeten Beispiel aus einem 6×10 Gitter mit einem Loch in der Mitte, und soll in zwei Teile geteilt werden.

Es sollen nur gerade Schnitte vollzogen werden. Die Teilgebiete werden nach den Schnitten ebenfalls rechteckige 2D-Gitter sein. Eine Teilung kann entweder in x-Richtung oder in y-Richtung erfolgen. Jedes Teilgebiet soll ungefähr die gleiche Anzahl an zu berechnenden Knoten erhalten. Um herauszufinden an welcher Stelle und in welche Richtung geteilt werden muss, um die besten Resultate zu erhalten, wird wie folgt vorgegangen:

1. Erstellen eines Arrays $load\_x[xmax]$ in dem die Anzahl der Punkte gespeichert werden, die in dem Schnitt vorhanden sind; z.B. für $x_0 = 0$ sind in y-Richtung 6 gültige Punkte vorhanden. Somit ist $load\_x[0] = 6$ (siehe Abb. 4.1 und Abb. 4.2).

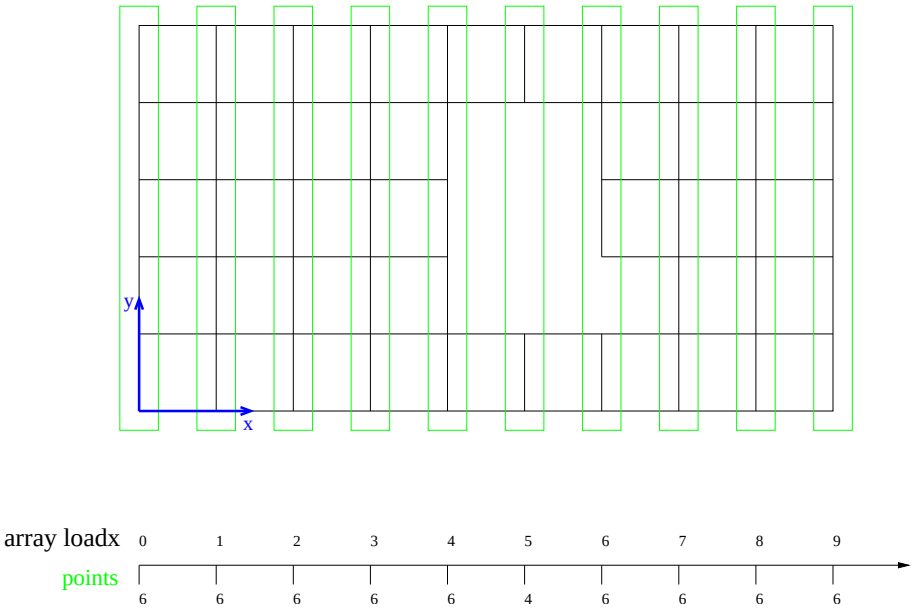


Abbildung 4.1: Addition der gültigen Punkte in y-Richtung bei festem x_0 in Array abspeichern

2. Integration der Arrays $load\_x$ von links nach rechts; (Abb. 4.3) z.B. für $x_i = 3$ beträgt die Summe der links von der Schnittstelle liegenden Punkte (einschließlich Stelle 3) 24. Das neue Array $load\_int\_l[xmax]$ hat an der Stelle 3 den Wert 24; $load\_int\_l[3] = 24$

3. Integration von $load\_x$ von rechts nach links; (Abb. 4.4) z.B. für $x_i = 3$ beträgt die Summe der rechts von der Schnittstelle liegenden Punkte (ohne Stelle 3) 40. $load\_int\_r[3] = 40$.

4. Bilden der Differenz der korrespondierenden Arrays $load\_int\_l$ (links nach rechts) und $load\_int\_r$ (rechts nach links) und erzeugen eines neuen Array

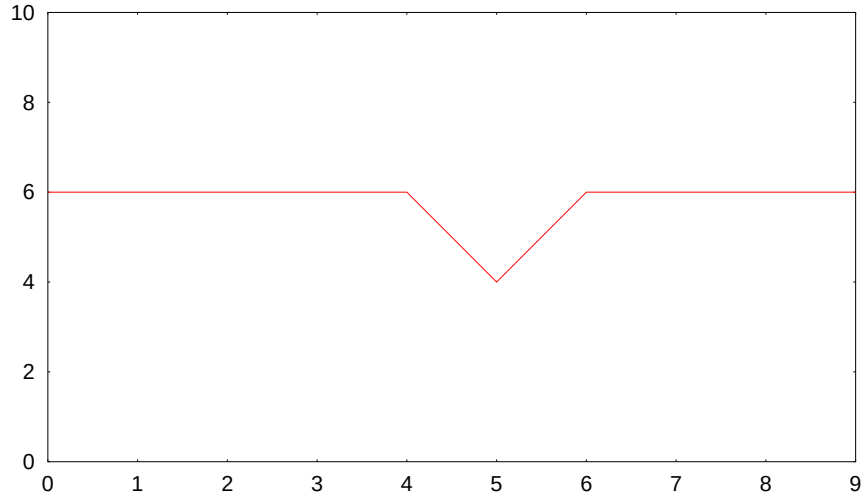


Abbildung 4.2: Array *load_x* in x-Richtung

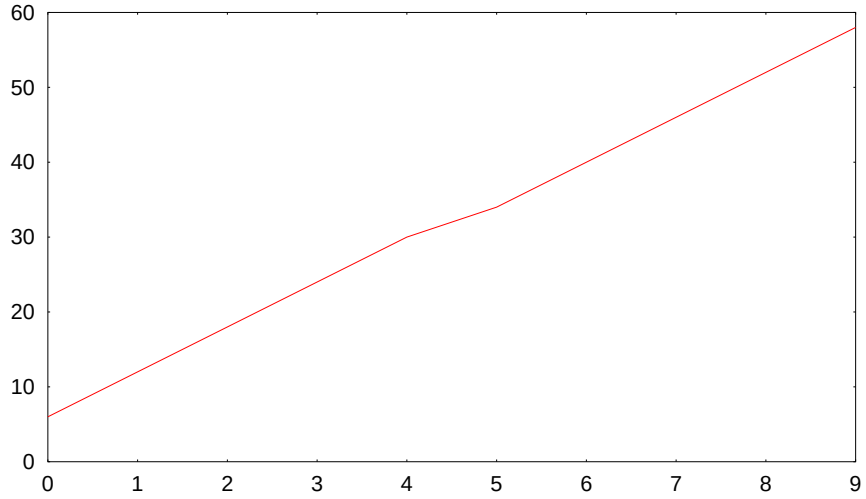


Abbildung 4.3: Integration des Arrays *load_x* (Abb.4.2) an jedem Punkt von links nach rechts

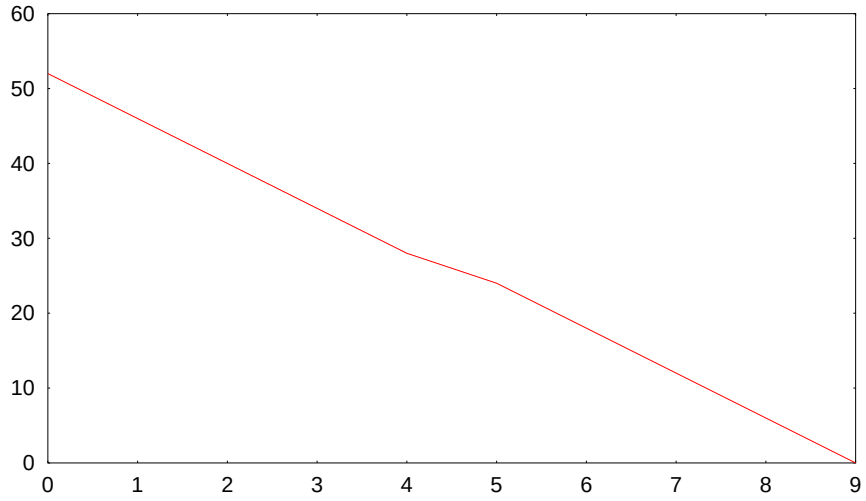


Abbildung 4.4: Integration des Arrays *load_x* (Abb.4.2) an jedem Punkt von rechts nach links

load_diff$[xmax]$ mit dem Absolutwert der Differenzen. (Abb. 4.5) Wird dieses Array graphisch dargestellt, erhält man in der Regel einen V-förmigen Graphenverlauf. Dieser Graph gibt für jede Stelle i die Differenz der Knotenanzahl zwischen den Teilgebieten rechts und links vom Schnitt an der Stelle i an;
z.B. $|load\_int\_l[3] - load\_int\_r[3]| = 16 = load\_diff[3]$
Dieses Array wird im folgenden mit *load_diff* bezeichnet.

5. Das Minima dieser Differenz-Kurve ist an der für den Schnitt günstigsten Stelle, wenn man nur das Loadbalancing betrachtet und "gerade" Schnitte ausführt.

6. Das gleiche Verfahren muss für die andere Koordinatenrichtung (in 2D nur noch in y-Richtung) ausgeführt werden.

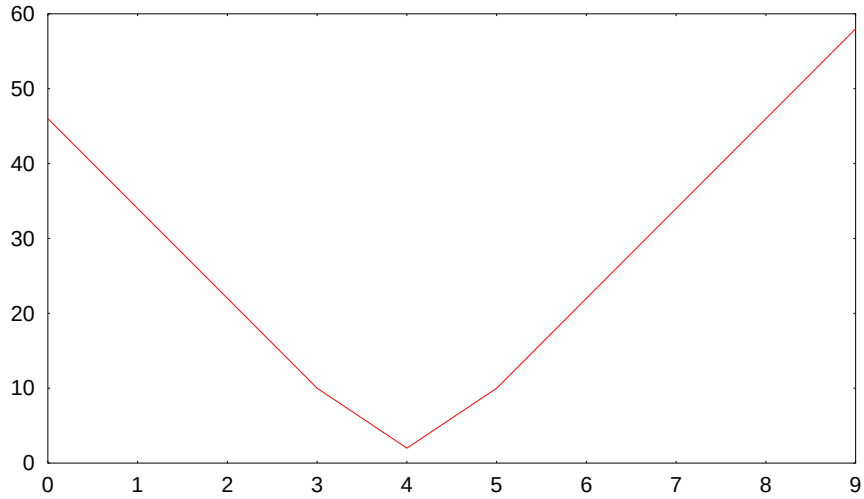


Abbildung 4.5: Differenz der Kurven Abb.4.3 und Abb. 4.4 (Absolut-Werte)

4.2.2 Phase 2: Minimierung der Kommunikation in 2D

1. Es wird ähnlich wie beim Loadbalancing ein Array $comm\_x[xmax]$ erzeugt, welches für jede Stelle x_i die gültigen Kommunikations-Links speichert. Die für gültige Links in Frage kommenden Punkte werden in 2D nach d2q9 (siehe Abb. 3.6) ermittelt (Abb. 4.6). Da die Kommunikation immer bidirektional erforderlich ist, genügt es, nur eine Richtung zu zählen. Deshalb werden nur die rechts von der Schnittstelle liegenden Links auf ihre gültigkeit geprüft. Die tatsächliche Anzahl der Kommunikationslinks ist genau doppelt so groß wie der ermittelte Wert.

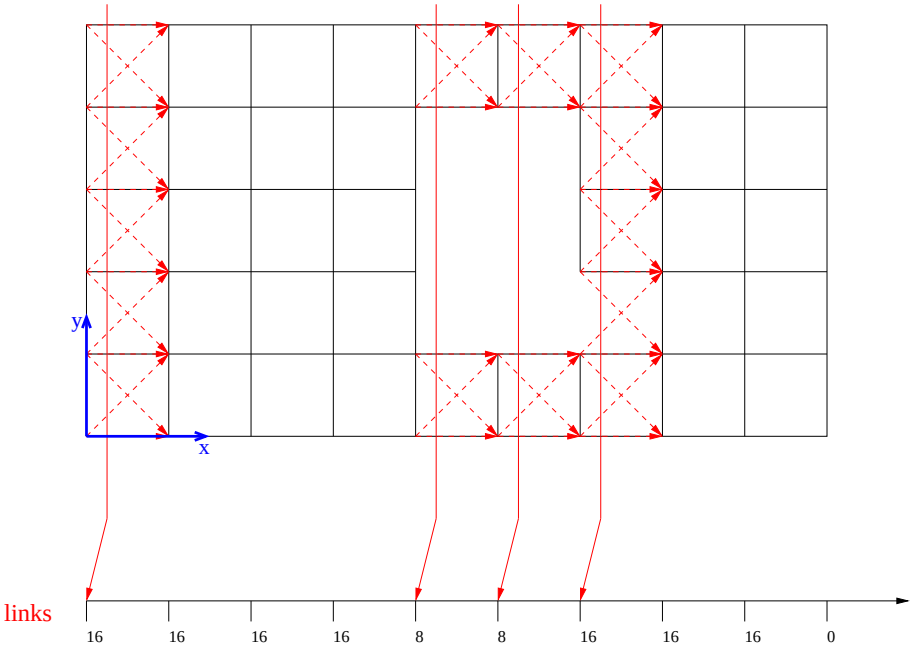


Abbildung 4.6: Zählen der gültigen Links im Schnitt x_0 und in Array abspeichern

2. Die Minima in der Kurve für die Kommunikation stellen die günstigen Stellen dar (Abb. 4.7).

3. Das gleiche Verfahren muss nun für die andere Koordinatenrichtung ausgeführt werden.

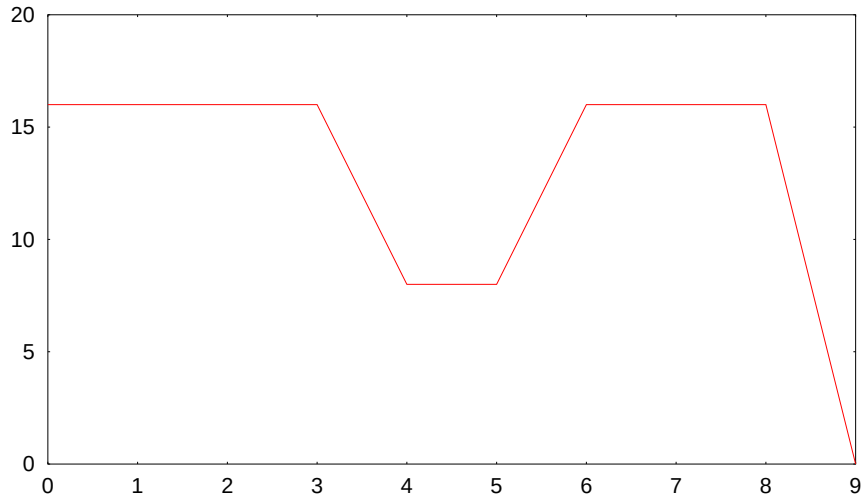


Abbildung 4.7: Array *comm_x* in x-Richtung

4.2.3 Phase 3: Kombination aus Loadbalancing und Minimierung der Kommunikation in 2D

Nachdem die für eine Teilung günstigen Stellen getrennt nach Loadbalancing und Kommunikation ermittelt sind, gibt es verschiedene Ansätze, mit diesen Werten die tatsächlich optimale Stelle zu ermitteln.
Dazu werden die beiden Werte für Loadbalancing und Kommunikation kombiniert. Um die beiden Werte kombinieren und durch Wichtungsfaktoren den Einfluss von Kommunikation und Loadbalancing steuern zu können, werden die beiden Kurven normiert (Abb. 4.8 und Abb. 4.9). Der Wert für die Kommunikation an der Stelle $x_0 = xmax$ beträgt immer Null. Demnach wäre dort immer ein Minimum für die Kommunikation. Da an dieser Stelle eine Zerlegung keinen Sinn machen würde, wird dieser Wert vor der Normierung auf den maximalen Wert gesetzt. Die normierten Kurven werden nun mit den jeweiligen Wichtungsfaktoren multipliziert und addiert. Die hier im Beispiel verwendeten Wichtungsfaktoren sind:

- FAKload = 70 und
- FAKcomm = 30.

Das Minimum dieser Kurve (Abb. 4.10) ist demnach der günstigste Schnitt in x-Richtung, z.B. hier nach $x_i = 4$.

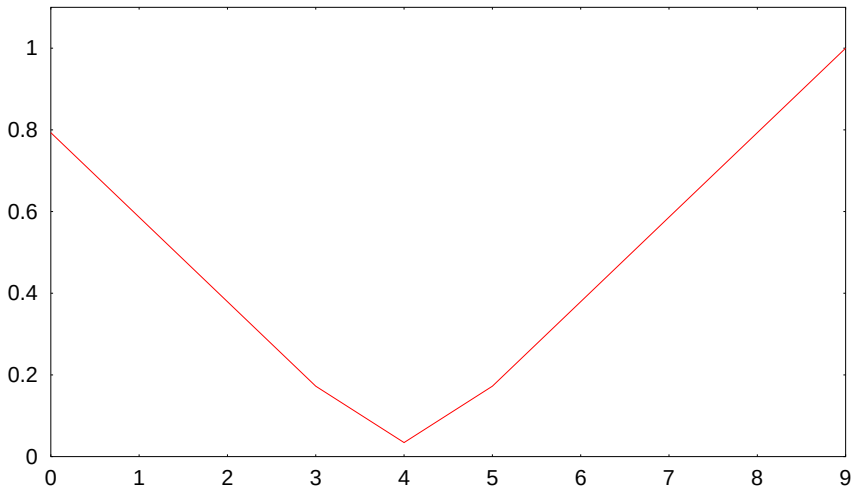


Abbildung 4.8: Normierte Kurve für Load in x-Richtung

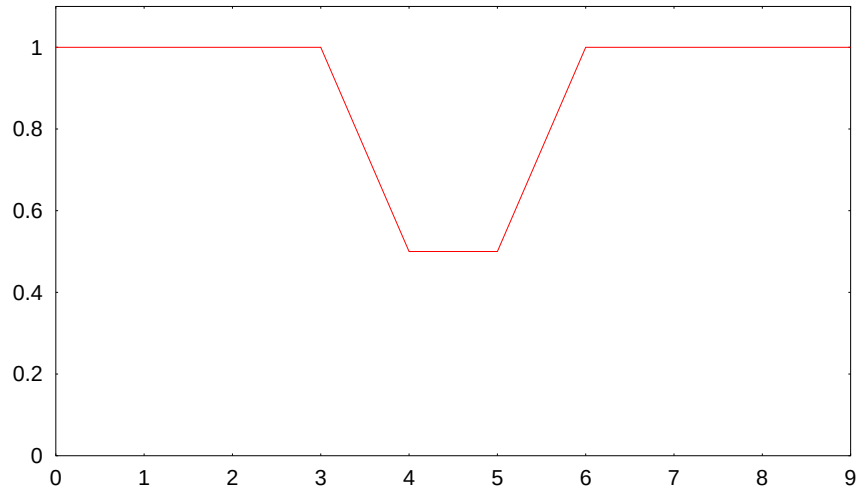


Abbildung 4.9: Normierte Kurve für Kommunikation in x-Richtung

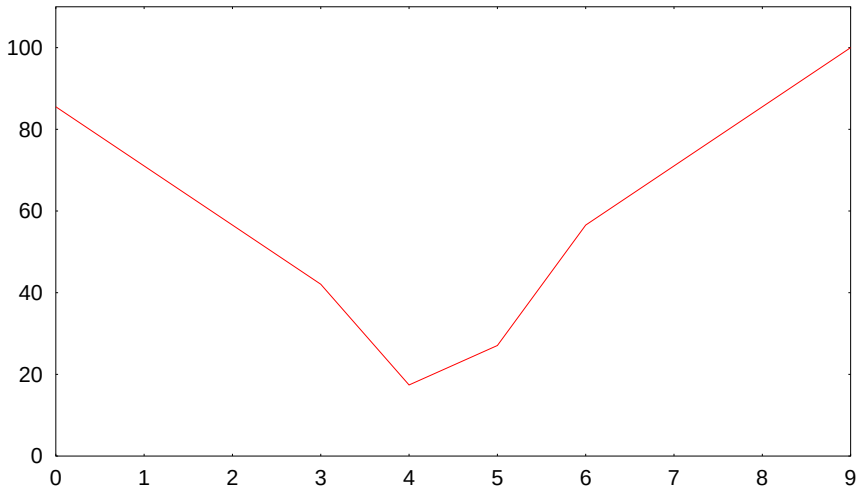


Abbildung 4.10: Gewichtete Summenkurve in x-Richtung

4.2.4 Phase 4: Vergleich der Koordinatenrichtungen

Nachdem die günstigsten Stellen für (in 2D) beide Koordinatenrichtungen gefunden wurden, müssen die Stellen noch miteinander verglichen werden. Das Minimum der beiden Werte ergibt die günstigere Koordinatenrichtung.

4.3 Übertragung des Algorithmus von 2D nach 3D

Der Übergang des Algorithmus von 2D nach 3D ist im wesentlichen durch hinzufügen der dritten Koordinatenrichtung zum Algorithmus erledigt. Beim der Normierung der Arrays muss darauf geachtet werden, die relativen Größen in Bezug auf das Maximum aller Koordinatenrichtungen zu bilden. Die Schnittflächen sind jetzt Scheiben. Es müssen jeweils drei Koordinatenrichtungen untersucht werden. Dabei wird die Kommunikation wahlweise nach dem "d3q15" oder "d3q19" Modell berechnet. Da mit diesem Verfahren nur wenig zusätzlicher Speicher zur Berechnung benötigt wird, kann mit einem PII-350MHz mit 384MB RAM problemlos ein 300^3 Gitter in weniger als zwei Minuten in 8 Teilgebiete zerlegt werden. Mit METIS stößt man bereits mit einem 100^3 Gitter (Faktor 27 !!) an die Grenze dieses Systems.

4.4 Ressourcenbedarf des Algorithmus

Bei der Zerlegung mit diesem Algorithmus wird, zusätzlich zu der sich im Hauptspeicher befindlichen 3D-Matrix, Speicher für die Arrays angelegt (allokiert). Die Größe des Speicherbedarfs ist von den Abmessungen der 3D-Matrix abhängig. Nach der Berechnung einer Teilung wird dieser Speicher sofort wieder freigegeben. Der Speicherverbrauch ist also bei der ersten Teilung am größten. Es werden folgende Arrays angelegt:

- $5 \cdot 3 \cdot max(xmax, ymax, zmax) \cdot sizeof(int)$
  - 5: folgende fünf Arrays: $load\_[x|y|z]$, $load\_int\_l[x|y|z]$, $load\_int\_r[x|y|z]$, $load\_diff[x|y|z]$, $comm\_[x|y|z]$
  - 3: alle drei Koordinatenrichtungen
  - max(xmax,ymax,zmax): Die Größe der Arrays entspricht der längsten Kante des quaderförmigen 3D-Gitters.
  - sizeof(int): Größe des Speicherbedarfs eines Integer-Wertes (bei 32-Bit Systemen in der Regel 4 Bytes)
- $3 \cdot 3 \cdot max(xmax, ymax, zmax) \cdot sizeof(double)$
  - 3: folgende drei Arrays: $norm\_load[x|y|z]$, $norm\_comm[x|y|z]$, $komb\_load\_comm[x|y|z]$
  - 3: alle drei Koordinatenrichtungen
  - max(xmax,ymax,zmax): Die Arrays haben ebenfalls die Länge der größten Quaderkante.
  - sizeof(double): Größe des Speicherbedarfs eines double-Wertes (bei 32-Bit Systemen in der Regel 8 Bytes).
- ca. 10 Variablen um die Minima zu berechnen und die Stellen zu speichern, an denen sich die Minima ergeben. Diese Variablen sind vom Datentyp integer oder double und von der Gebietsgröße unabhängig.

Für eine 300^3 3D-Matrix ist somit maximal
$5 \cdot 3 \cdot 300 \cdot 4Bytes + 3 \cdot 3 \cdot 300 \cdot 8Bytes + 10 \cdot 4Bytes = 39640Bytes$
zusätzlicher Hauptspeicher notwendig (32Bit-System). In den folgenden Diagrammen (Abb. 4.11) ist der Speicherbedarf in Bezug auf die Anzahl der Knoten bzw. Kanten aufgetragen.
Im Gegensatz zu einem graphenbasierten Zerleger wie z.B. METIS oder ParMETIS ist es nicht erforderlich die in der 3D-Matrix abgebildete Geometrie auf einen Graphen abzubilden. Dadurch ist der Resourcenbedarf eines geometriebasierten Algorithmus wesentlich anspruchsloser und somit für große Systeme besser geeignet.

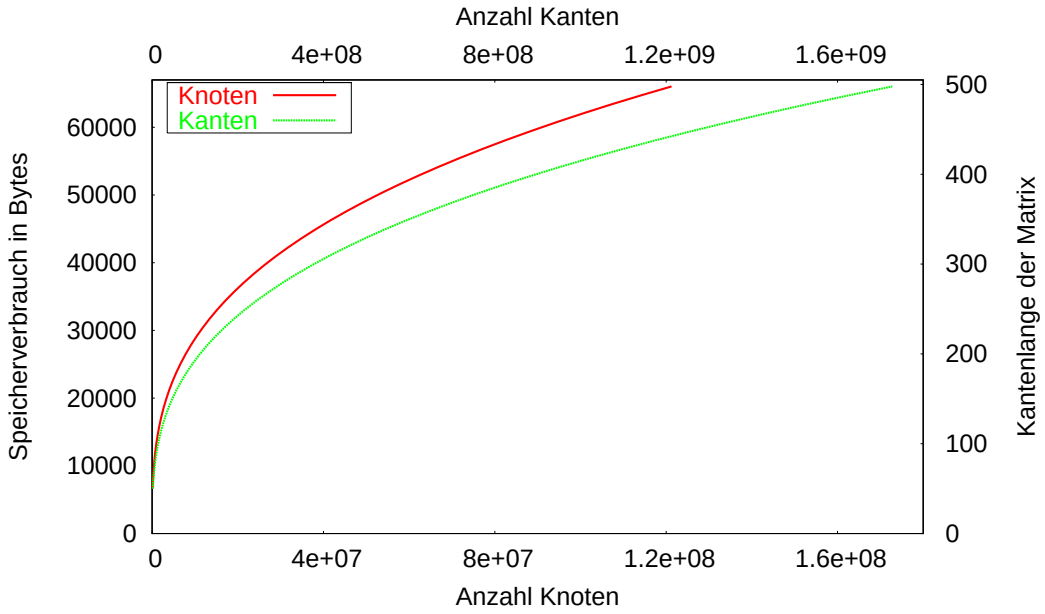


Abbildung 4.11: Speicherbedarf divide

4.5 Parallelisierung von divide

Werden mit Hilfe von Timingfunktionen Performancemessungen durchgeführt, so stellt man fest, dass die meiste Zeit benötigt wird, die Arrays $load\_[x|y|z]$ und $comm\_[x|y|z]$ zu generieren. Da die Arrays vollständig unabhängig von den jeweils anderen Koordinatenrichtungen berechnet werden können, ist keine Interprozesskommunikation während der Berechnung nötig. Erst zur Entscheidung, welche der drei Koordinatenrichtungen die günstigste ist, wird es notwendig, die Ergebnisse der parallelen Array-Berechnung mit den jeweils anderen Prozessoren zu synchronisieren.
Testläufe haben gezeigt, dass die Ergebnisse aus den Berechnungen mit der parallelen und seriellen Version übereinstimmen.
Ablauf der Parallelen Version:

1. Aufstellen der geometrischen Matrix im Hauptprozess
2. Versenden der geometrischen Matrix an die beiden Subprozesse

   Für den Prozess der die z-Richtung bearbeitet, wird die Matrix nicht in der Form init_3dmatrix_int(XMAX,YMAX,ZMAX) initialisiert, sondern in der Form init_3dmatrix_int(ZMAX,XMAX,YMAX). Die Schleifen, die bei der Berechnung der Arrays für load und comm durchlaufen werden, müssen zwangsweise in der Reihenfolge ausgeführt werden, dass die z-Schleife die äußerste ist. Deshalb wird beim Versand der geometrischen Matrix an Prozess 2 (der Prozess, der die Arrays in z-Richtung berechnet) die Matrix so umgebaut, dass mit außen liegender z-Schleife genau so viele Speichersprünge wie in der x- und y-Richtung auftreten. Somit wird verhindert, dass die Prozesse 0 und 1 (x und y) auf den Prozess 2 (z) warten müssen. Dadurch wird ein besseres Loadbalancing (im Zerleger) erreicht.
3. Parallele Berechnung der Arrays für load und comm
4. Parallele Auswertung der Arrays und Berechnung der günstigsten Stellen in den jeweiligen Koordinatenrichtungen
5. Synchronisation der Ergebnisse (jeder Prozess kennt nun alle günstigsten Stellen)
6. Prozess 0 vergleicht die Koordinatenrichtungen miteinander
7. Synchronisation des Ergebnisses: die günstigste Koordinatenrichtung steht fest
8. Jeder Prozess teilt das Gebiet in der berechneten Richtung
9. Die Teilgebiete werden berechnet durch rekursiven Aufruf der Teilungsfunktion

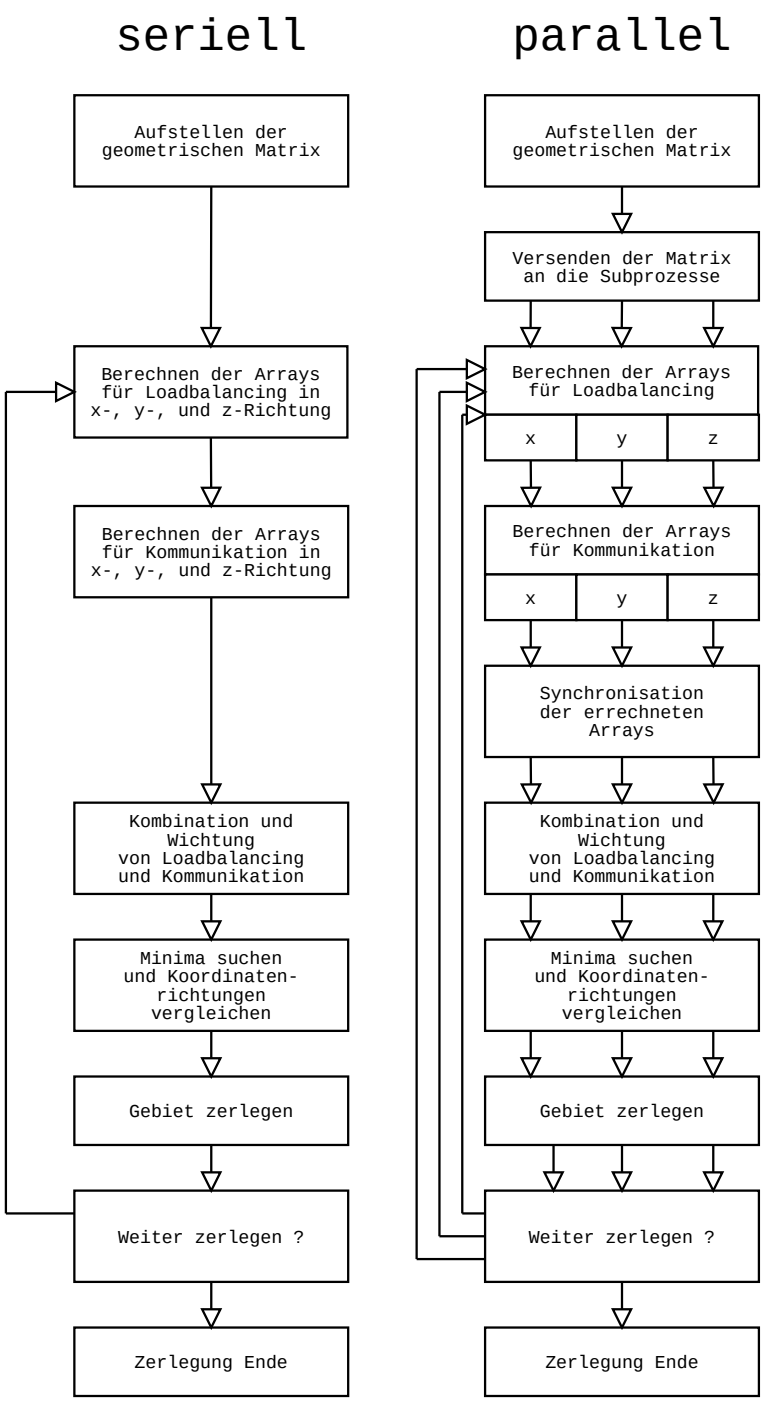


Abbildung 4.12: Flußdiagramm des Standart divide-Algorithmus und einer möglichen Parallelisierung (rechts)

Kapitel 5

Performance und Qualität von METIS und divide

Um METIS bzw. ParMETIS mit divide zu vergleichen, werden die Ergebnisse beider Algorithmen folgendermaßen ausgewertet.

5.1 Loadbalancing

Die Auswertung der beiden Algorithmen für das Loadbalancing ist einfach. Ein Array mit den Werten "Knoten pro Teilgebiet (Domain)" wird erstellt. Es wird berechnet wieviele Knoten in jeder Domain sind. Dazu wird für jeden gültigen Knoten in der Matrix festgestellt in welcher Domain er liegt, und der zugehörige Wert im Array erhöht. Das erhaltene Array ist dann mit mehr oder weniger gleich großen Zahlen belegt. Optimal wäre, wenn alle Werte im Array gleich groß wären.

5.2 Kommunikation

Zur Beurteilung der Qualität der Zerlegung wird eine Kommunikationsmatrix K erstellt. Die Matrix hat die Dimension $[P \times P]$, wobei P die Anzahl der Teilgebiete ist. Der Wert K_{ij} steht für die Anzahl der Datenpakete, die vom Quellprozess i zum Zielprozess j in jedem Zeitschritt verschickt werden müssen.

5.3 Beispielzerlegung

Als Beispiel dient eine 3D-Gitter der Größe 100^3 Knoten, welches in 8 Partitionen aufgeteilt wird und folgende Hohlräume hat (siehe Abb. 5.1 und 5.2).

1. Kugel um (50,50,50) mit Radius 35
2. Kugel um (25,25,25) mit Radius 15

3. Kugel-Serie alle 10 mit Radius 5 jedoch in x-Richtung nur bis zur Hälfte
4. Kugel-Serie alle 10 um 5 versetzt mit Radius 3 jedoch in y-Richtung erst ab der Hälfte

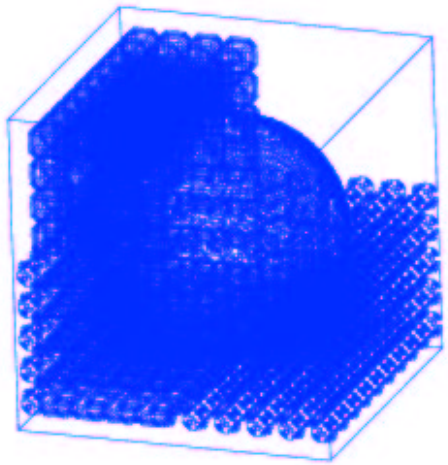


Abbildung 5.1: Geometrie für die Beispielzerlegung

Das Volumen wurde mit folgenden Methoden zerlegt:

1. Zerlegung mit METIS:
2. Zerlegung mit divide:
   Die Zerlegung mit "divide" wurde mit folgenden Parametern aufgerufen:
   (a) Wichtung Load:Kommunikation = 20:80,
   (b) Wichtung Load:Kommunikation = 50:50 und
   (c) Wichtung Load:Kommunikation = 95:5.

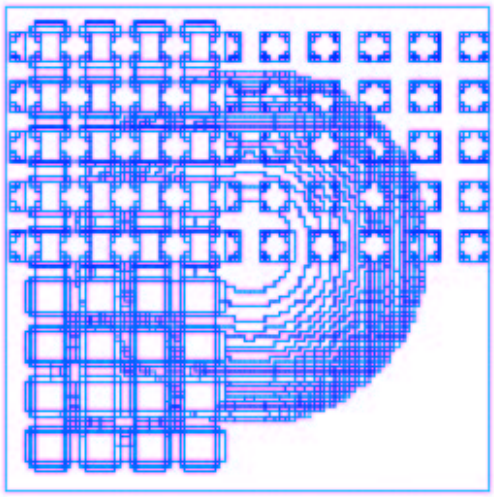


Abbildung 5.2: Geometrie für die Beispielzerlegung im Schnitt

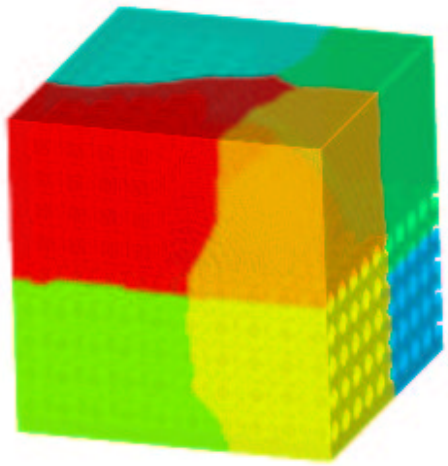


Abbildung 5.3: Ergebnis der Zerlegung mit METIS

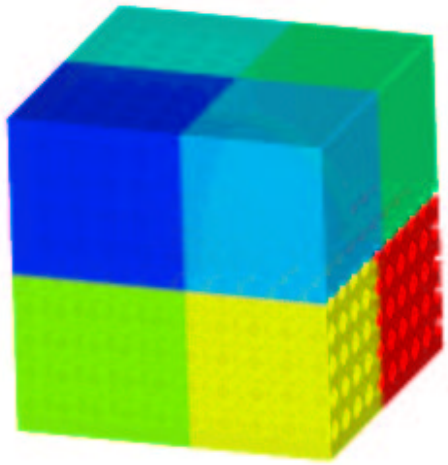


Abbildung 5.4: Ergebnis der Zerlegung mit divide 20:80

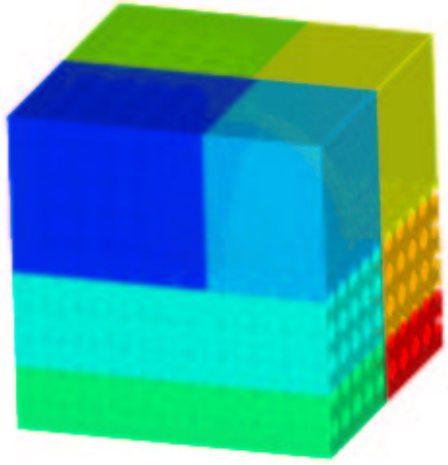


Abbildung 5.5: Ergebnis der Zerlegung mit divide 50:50

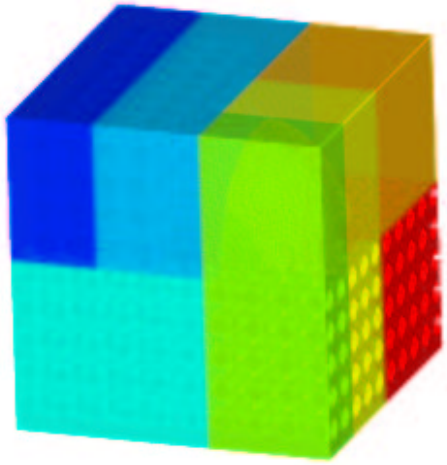


Abbildung 5.6: Ergebnis der Zerlegung mit divide 95:5

5.4 Auswertung der Zerlegungen

Die Auswertung der Load- und Kommunikations-Matrizen erfolgte wie in 5.1 und 5.2 beschrieben. (20:80 heißt: Gewichtung des divide-Algorithmus:
20 % Loadbalancing zu 80 % Kommunikation)
Zerlegung und Bewertung wurde in allen Beispielen nach d3q15 ausgeführt.
Um die Qualität der Zerlegungen vergleichen zu können wurden für jeweils alle vier Beispielzerlegungen folgende Werte ermittelt:

- "Knoten pro Domain"
- "Kommunikationsmatrix"

5.4.1 Loadbalancing

Die Auswertung des Loadbalancing wurde in der Form "Knoten pro Domain" (KpD) dargestellt.

Domain n	divide 20:80	divide 50:50	divide 95:5	METIS
0	67640	77428	73484	72410
1	89171	77151	75393	76808
2	70923	71421	74704	72409
3	85649	71527	74326	76713
4	60039	79152	73968	74604
5	74446	79652	75994	74978
6	65273	71514	74856	75063
7	83490	68786	73906	73646

Tabelle 5.1: Knoten pro Domain

In diesem Beispiel wäre der optimale Wert 74579 Knoten pro Domain. Um die Abweichung vom Loadoptimum (Lopt) in einem einzigen Wert darzustellen, wurden folgende Werte berechnet.

$$\prod_{n=0}^{n} 1 + \frac{abs(KpD_n - Lopt)}{Lopt} \tag{5.1}$$

Optimale Zerlegung	divide 20:80	divide 50:50	divide 95:5	METIS
1	2,3732	1,4819	1,0726	1,1498

Tabelle 5.2: Darstellung der Zerlegungsqualität nach Formel 5.1

Je größer die Abweichung vom Loadoptimum, desto größer die Abweichung von 1. Es wird deutlich, dass sich mit zunehmendem Gewicht auf Loadbalancing die Zerlegung mit dem divide-Algorithmus wie erwartet dem Loadoptimum nähert. Im Fall 95:5 übertrifft die Loadbalanceauswertung sogar die Werte von METIS.

5.4.2 Kommunikation

Die Auswertung der Qualität der Zerlegungen hinsichtlich Minimierung des Kommunikationsvolumens wird in Matrizen dargestellt.
Die Ergebnisse der Auswertung ergeben erwartungsgemäß eine Zunahme der Kommunikation mit der Abnahme der Gewichtung für die Kommunikation im Algorithmus von divide. Im Vergleich zu METIS bzw. ParMETIS ist mit hohem Gewicht auf Kommunikationsminimierung ein Wert für die Gesamtsumme der erforderlichen Kommunikation erreichbar, der unter dem von METIS bzw. ParMETIS liegt. In wieweit das Zusammenspiel zwischen Loadbalancing und Kommunikationsminimierung für die Performance der Simulation ausschlaggebend ist, zeigen die in Kap. 7 erfolgten Beispielrechnungen.

from/to	0	1	2	3	4	5	6	7
0	0	7099	3514	28	1654	10	77	0
1	7099	0	28	6506	10	4162	0	239
2	3514	28	0	6787	0	0	1443	8
3	28	6506	6787	0	0	0	8	4119
4	1654	10	0	0	0	4833	4212	14
5	10	4162	0	0	4833	0	14	4592
6	77	0	1443	8	4212	14	0	5066
7	0	239	8	4119	14	4592	5066	0
$\sum$	12382	18044	11780	17448	10723	13611	10820	14038
$\sum$ gesamt	108846							

Tabelle 5.3: Auswertung für die Zerlegung mit divide 20:80

from/to	0	1	2	3	4	5	6	7
0	0	7534	2089	0	3902	153	8	0
1	7534	0	3941	0	0	6021	26	0
2	2089	3941	0	9548	8	26	3499	0
3	0	0	9548	0	0	0	164	6015
4	3902	0	8	0	0	7211	1665	0
5	153	6021	26	0	7211	0	3967	0
6	8	26	3499	164	1665	3967	0	8415
7	0	0	0	6015	0	0	8415	0
$\sum$	13686	17522	19111	15727	12786	17378	17744	14430
$\sum$ gesamt	128384							

Tabelle 5.4: Auswertung für die Zerlegung mit divide 50:50

from/to	0	1	2	3	4	5	6	7
0	0	6457	4362	4367	0	0	0	0
1	6457	0	2281	2118	4389	2809	7051	0
2	4362	2281	0	3784	4974	2598	0	0
3	4367	2118	3784	0	0	71	73	7378
4	0	4389	4974	0	0	16820	0	0
5	0	2809	2598	71	16820	0	5896	6172
6	0	7051	0	73	0	5896	0	5986
7	0	0	0	7378	0	6172	5986	0
$\sum$	15186	25105	17999	17791	26183	34366	19006	19536
$\sum$ gesamt	175172							

Tabelle 5.5: Auswertung für die Zerlegung mit divide 95:5

from/to	0	1	2	3	4	5	6	7
0	0	2801	4112	0	2663	0	0	871
1	2801	0	931	3960	1164	3783	0	0
2	4112	931	0	7225	0	0	0	4108
3	0	3960	7225	0	0	461	5275	874
4	2663	1164	0	0	0	6136	110	1563
5	0	3783	0	461	6136	0	4589	0
6	0	0	0	5275	110	4589	0	8054
7	871	0	4108	874	1563	0	8054	0
$\sum$	10447	12639	16376	17795	11636	14969	18028	15470
$\sum$ gesamt	117360							

Tabelle 5.6: Auswertung für die Zerlegung mit METIS bzw. ParMETIS

Auch hier wird deutlich, dass der Kommunikationsaufwand bei abnehmender Gewichtung für die Kommunikation kleiner wird. Betrachtet man die Auswertung für das Loadbalancing und die Kommunikation getrennt voneinander, so erkennt man je nach Gewichtung der Zerlegung ein besseres Ergebnis als bei der Zerlegung mit METIS. Nun ist es notwendig durch Parameterstudien die optimalen Wichtungsfaktoren zu ermitteln, um die Beziehungen zwischen Loadbalancing und Kommunikationsvolumen genauer zu untersuchen. Zu diesem Zweck wird ein Beispiel mit einfacher Geometrie zerlegt und kurz angerechnet.

5.5 Einfaches Beispiel der Größe $200 \cdot 100 \cdot 100$

In einem zweiten Beispiel wurde eine einfache Geometrie, die aus einem Kanal mit einer Kugel als Hindernis bestand, erzeugt und mit dem Simulator berechnet. Um die Effektivität der Zerlegung zu beurteilen, genügt es, nur wenige Zeitschritte berechnen zu lassen. Für eine vollständige Simulation müssen mehrere zehntausend Zeitschritte berechnet werden. In diesem Beispiel wurden 50 Zeitschritte berechnet. Die Berechnung wurde auf zwei, vier und acht Prozessoren durchgeführt. Das Gebiet wurde dabei mit dem Preprozessor mit 30 verschiedenen Parametereinstellungen zerlegt, und mit dem Simulator berechnet.
Jeweils für vier und acht Prozessoren mit dem divide Algorithmus bei folgenden Parametern:

- Gewichtung zwischen Load und Kommunikation 0:100,
- Gewichtung zwischen Load und Kommunikation 10:90,
- ...
- Gewichtung zwischen Load und Kommunikation 90:10,
- Gewichtung zwischen Load und Kommunikation 100:0,

Zudem wurde das Gebiet ebenfalls mit METIS, x-Slice, y-Slice und z-Slice zerlegt und angerechnet. Die zur Beurteilung der Performance relevanten Größen wurden den Bildschirmausgaben der Rechendurchgänge entnommen. Diese sind die Updaterate, die Loadunbalance und das Messagevolumen. Die Updaterate beschreibt die Summe der Anzahl der Rechenschritte (Kollision, Propagation, senden und empfangen der Daten) aller an der Rechnung beteiligten Knoten pro Sekunde [$Updates/s$]. Die Loadunbalance ist die Abweichung der Lastverteilung von der optimalen Auslastung und wird nach Formel 5.1 berechnet. Das Messagevolumen ist die Anzahl der pro Zeitschritt versendet und empfangenen Werte [$Anzahl\ der\ Messages/timestep$].
Die Güte der Zerleger wird in den folgenden Abbildungen verglichen.

1. Updaterate bei 4 Prozessoren (Abb. 5.7):

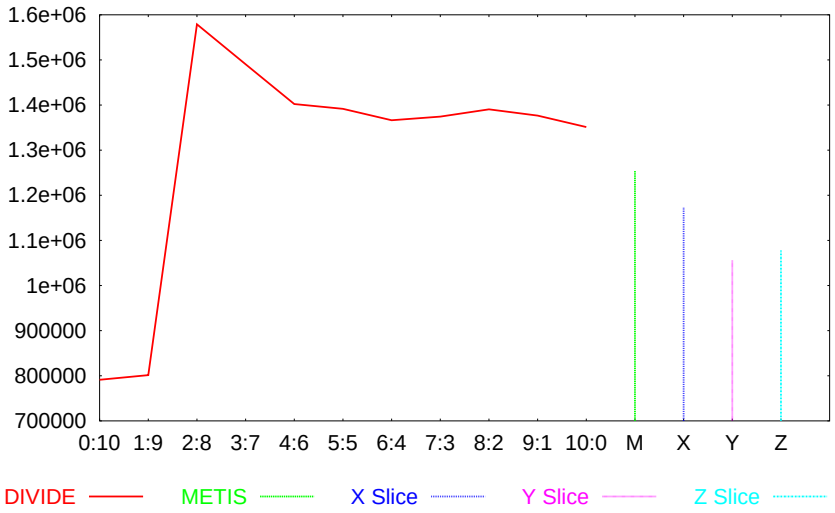


Abbildung 5.7: Updaterate $200 \cdot 100 \cdot 100$ bei 4 Prozessoren

2. Loadunbalancing bei 4 Prozessoren (Abb. 5.8):
3. Kommunikationsvolumen bei 4 Prozessoren (Abb. 5.9):

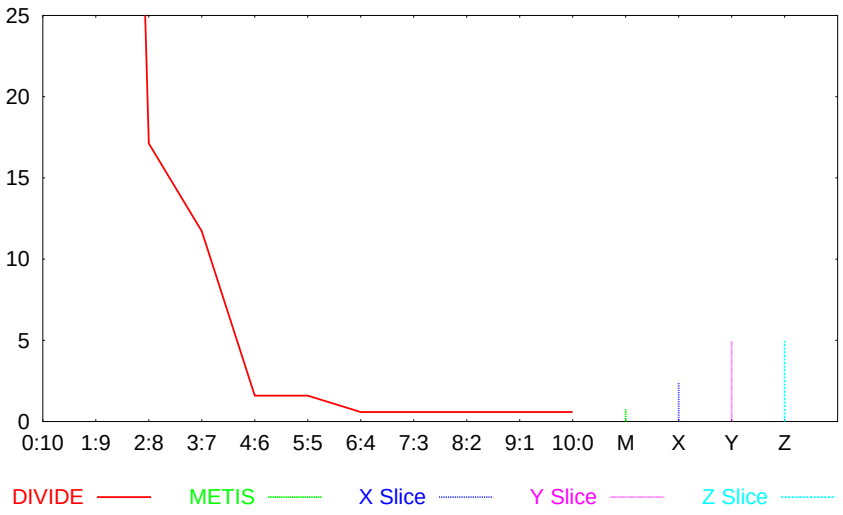


Abbildung 5.8: Loadunbalancing $200 \cdot 100 \cdot 100$ bei 4 Prozessoren

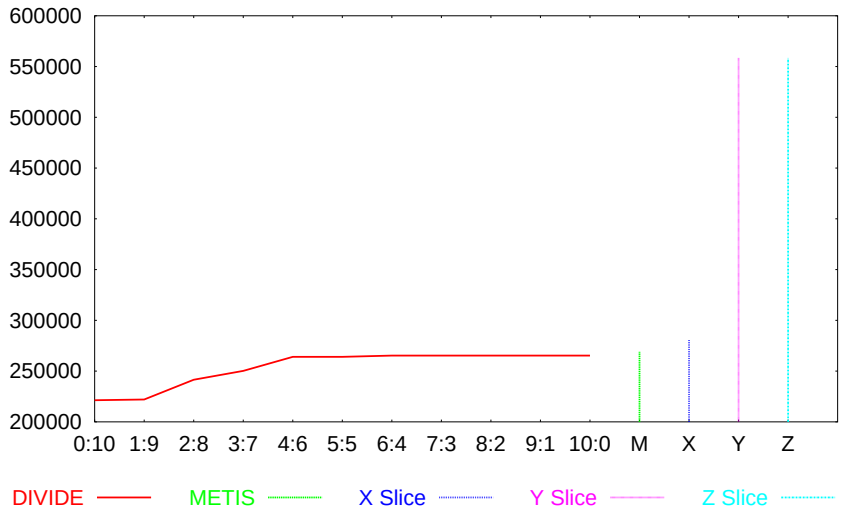


Abbildung 5.9: Kommunikationsvolumen $200 \cdot 100 \cdot 100$ bei 4 Prozessoren

4. Updaterate bei 8 Prozessoren (Abb. 5.10):

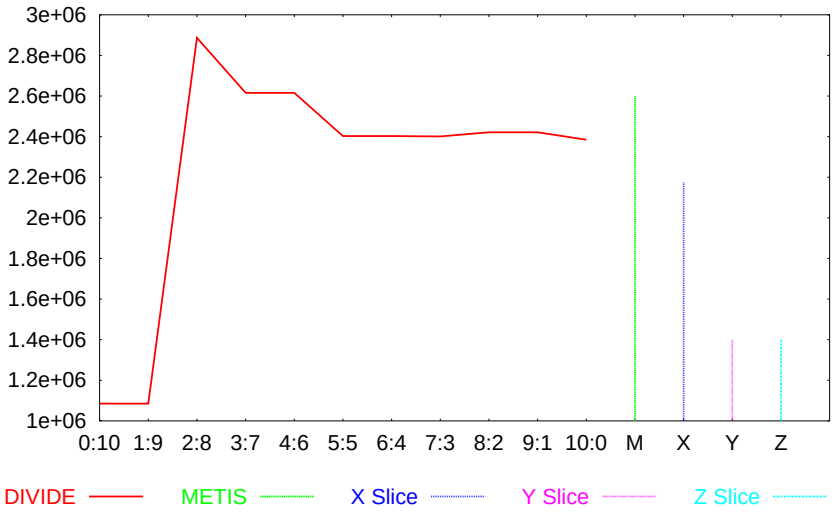


Abbildung 5.10: Updaterate $200 \cdot 100 \cdot 100$ bei 8 Prozessoren

5. Loadunbalancing bei 8 Prozessoren (Abb. 5.11):
6. Kommunikationsvolumen bei 8 Prozessoren (Abb. 5.12):

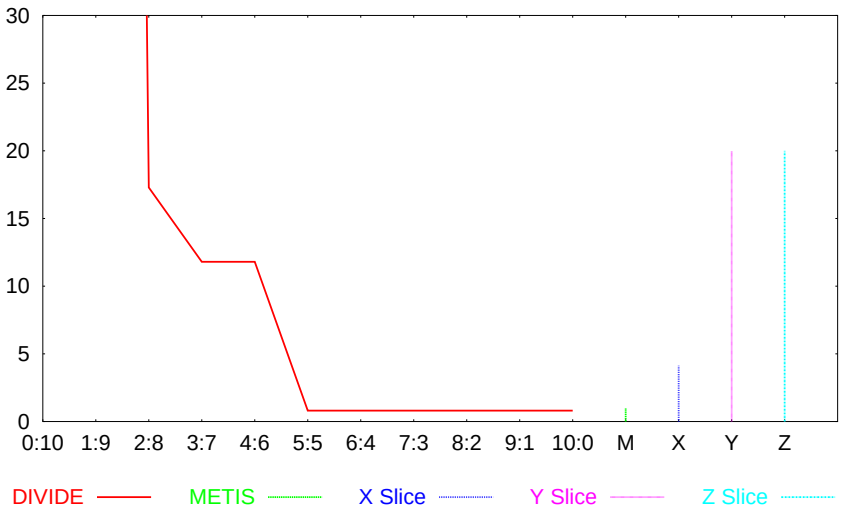


Abbildung 5.11: Loadunbalancing $200 \cdot 100 \cdot 100$ bei 8 Prozessoren

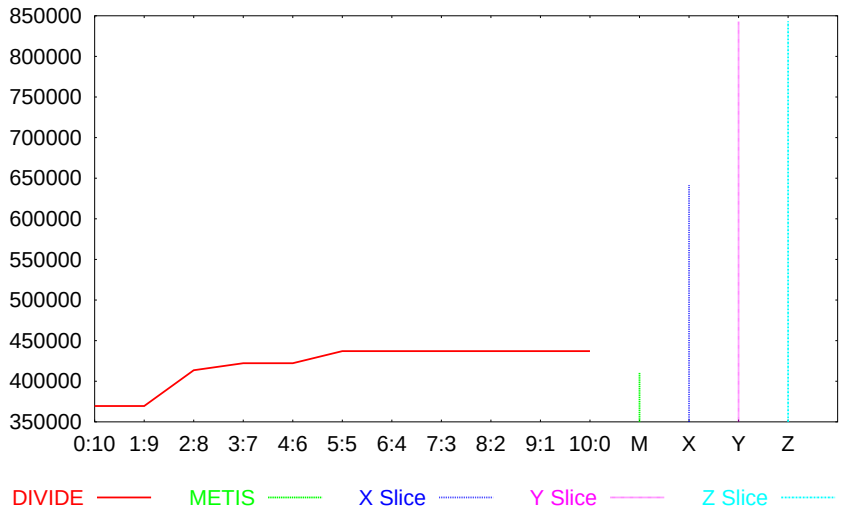


Abbildung 5.12: Kommunikationsvolumen $200 \cdot 100 \cdot 100$ bei 8 Prozessoren

5.6 Tuning des divide Algorithmus

Der Zerleger METIS ist bei für Lattice-Boltzmann Applikationen vergleichsweise kleinen Berechnungsgebieten auf Grund der hohen Resourcenanforderung (Hauptspeicher) nicht mehr einsatzfähig. Der Einsatz von divide ist wegen dem geringen zusätzlichen Speicherverbrauch bei Berechnungsgebieten mit hohen Knotenzahlen zu empfehlen. Durch die Auswertung von mehreren Testrechnungen, vor allem bei komplizierten Geometrien (poröses Medium), wurden die folgenden bei Lattice-Boltzmann charakteristischen Eigenschaften für eine möglichst performante Simulation festgestellt.

- Mit steigender Knotenzahl nimmt die Relevanz eines ausgewogenen Loadbalancings zu.
- Die Berechnungszeit zur Auswertung der geometrischen Matrix steigt mit der Anzahl der Knotenzahl.
- Im Verlauf der Simulation existieren verschiedene Knotentypen, welche unterschiedliche Berechnungszeiten aufweisen.

Um die Performance und die Qualität der Zerlegung im Bereich hoher Knotenzahl zu steigern, wurden die folgenden Optimierungen implementiert.

5.6.1 Definition des Parameters "AREA"

Zum einen steigt der Aufwand zur Berechnung der Arrays für die Kommunikationslinks (siehe: 4.2.2) mit steigender Gittergröße und zum anderen wird ebenfalls bei steigender Gittergröße die Relevanz eines ausgewogenen Loadbalancings immer wichtiger. Um die Performance des Zerlegers für hohe Knotenzahlen zu steigern wurde der Parameter AREA eingeführt. Dieser Parameter begrenzt die Berechnung der Kommunikationslinks auf ein Intervall um die Stelle, bei der die Loadbalance optimal ist. Nach der Berechnung der Arrays zur Auffindung der Stelle, an der das Loadunbalancing minimal ist, erfolgte bisher die Berechnung der Anzahl der Kommunikationslinks in jeder möglichen Schnittfläche für die Koordinatenrichtungen x, y und z. Um Zeit zu sparen erfolgt die Berechnung der Arrays für die Kommunikationsminimierung nicht mehr auf der kompletten geometrischen Matrix, sondern für jede Koordinatenrichtung in einem Intervall um die Stelle, bei der das Loadunbalancing minimal ist. Da sich für hohe Knotenzahlen ein ausgewogenes Loadbalancing als dominierendes Kriterium für eine performante Simulation herauskristallisiert hat, gibt es durch diese Ungenauigkeit in der Regel keine Veränderungen im Ergebnis der Zerlegung. Der Parameter AREA legt dabei die Intervallgrenzen fest und ist wie folgt definiert:
AREA beschreibt eine Obergrenze für die Loadunbalance in %. Die Spanne zwischen dem Maximalwert (LOADmax) und dem Minimalwert (LOADmin) des

Arrays *load_diff* (vgl. Kap. 4.2.1) entspricht dabei $AREA = 100\%$. Folgende Grafik soll dieses Kriterium näher verdeutlichen (Abb. 5.13). Das erzeugte Inter-

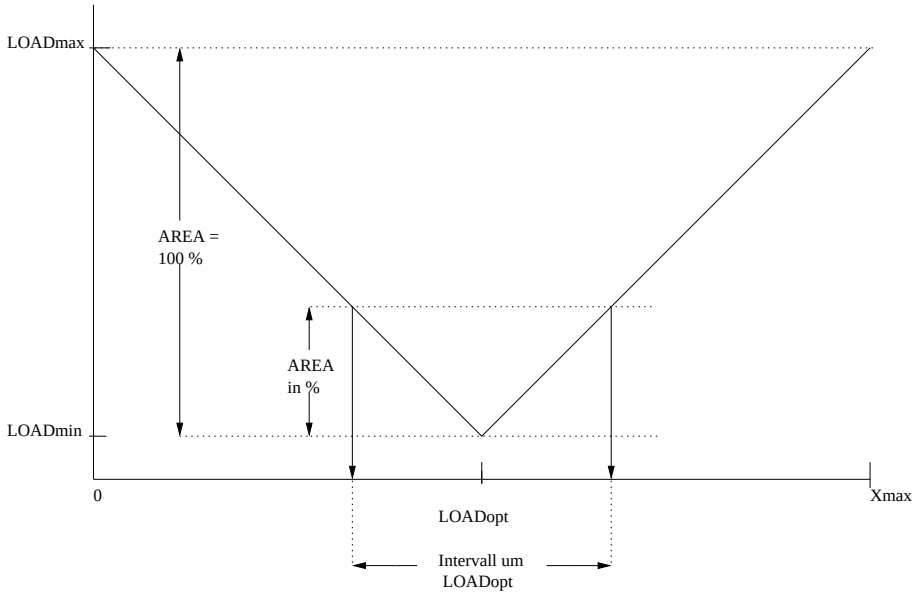


Abbildung 5.13: Definition des Parameters “AREA”

vall ist unabhängig von der Gitterbreite. Ebenfalls wird sichergestellt, dass das Loadbalancing nicht schlechter werden kann, als durch den Parameter festgelegt.

5.6.2 Unterscheidung der Knotenarten

Im Laufe der Simulation gibt es zwei verschiedene für die Berechnung relevante Knotenarten. Dies sind zum einen die Fluidknoten und zum anderen die Randknoten. Randknoten sind die Fluidknoten, deren Nachbarknoten gemäß d3q15 bzw. d3q19 aus mindestens einem Nichtfluidknoten bestehen. Durch Messungen während der Simulation wurde festgestellt, dass die Rechenzeit für die Berechnung der Fluidknoten um den Faktor fünf höher ist, als die benötigte Rechenzeit für die Randknoten. Um dieses schon bei der Zerlegung zu berücksichtigen, wurde in den Algorithmus eine Gewichtung der beiden Knotenarten eingearbeitet.

5.6.3 Berücksichtigung der Baumtiefe

Bedingt durch das bisektionale Vorgehen des Algorithmus von divide wird z.B. bei einer Zerlegung in acht Teilgebiete das gesamte Berechnungsgebiet zunächst in zwei Teilgebiete zerlegt. Diese werden anschließend jeweils unabhängig voneinander und der vorausgegangenen Teilung nochmals in zwei Teilgebiete zerlegt. Die so entstandenen vier Teilgebiete werden nach dem gleichen Schema ein letztes Mal zerlegt, so dass die acht erforderlichen Teilgebiete vorhanden sind. Wird bei der Initialteilung, also der Teilung des Gesamtgebiets, bedingt durch die Wichtungsfaktoren eine Position für den Schnitt festgelegt, welche ein unausgewogenes Loadbalancing zur Folge hat, dann kann dieser Missstand durch die folgenden Teilungen nicht korrigiert werden. Aus diesem Grund ist es möglich, die Wichtung (Load:Comm) zusätzlich durch die Anzahl der zu erzeugenden Teilgebiete zu beeinflussen. Ziel dieser Anpassung ist es, bei Teilgebieten die noch weiter zerlegt werden sollen, die Gewichtung für das Loadbalancing zu erhöhen. Dies wird durch die Faktoren erreicht, der sich nach folgender Formel 5.2 bzw. 5.3 berechnet:

$$FAKTOR\_LOAD = (10 + n - 1)/10.0 \tag{5.2}$$

$$FAKTOR\_COMM = (10 - n + 1)/10.0 \tag{5.3}$$

Dabei ist n jeweils die Anzahl der noch anstehenden rekursiven Aufrufe der Teilungsfunktion. In Tabelle 5.7 ist eine Übersicht der zusätzlichen Faktoren bis zu einer max. Baumtiefe von drei, d.h. für eine Zerlegung in bis zu 16 Teilgebieten dargestellt.

Teilgebiete	max. Baumtiefe	n	Faktor Load	Faktor Comm
2	0	1	1.0	1.0
4	1	2	1.1	0.9
		1	1.0	1.0
8	2	3	1.2	0.8
		2	1.1	0.9
		1	1.0	1.0
16	3	4	1.3	0.7
		3	1.2	0.8
		2	1.1	0.9
		1	1.0	1.0

Tabelle 5.7: Überblick der zusätzlichen Faktoren

Als Beispiel wurde eine Simulation mit und ohne Berücksichtigung der Baumtiefe bei der Zerlegung durchgeführt. Als Testgebiet diente ein Rechendurchgang aus Bsp. 7.4. Das Gebiet wurde zuerst ohne die Zusatzfaktoren mit einer Gewichtung von 60:40 Load:Kommunikation berechnet, anschließend mit

Berücksichtigung der Baumtiefe und im Vergleich dazu mit einer Gewichtung von 100:0 Load:Kommunikation ohne Berücksichtigung der Baumtiefe. Die Ergebnisse sind in folgenden Grafiken dargestellt. Der Parameter AREA war bei allen Durchgängen, mit AREA=20, konstant.

- Updaterate bei 8 Prozessoren (Abb. 5.14):
- Loadunbalancing bei 8 Prozessoren (Abb. 5.15):
- Kommunikationsvolumen bei 8 Prozessoren (Abb. 5.16):
- Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 5.17):
- Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 5.18):

Es wird deutlich, dass durch die Einbeziehung der Baumtiefe in die Gewichtung, die Qualität der Zerlegung erhöht wird. Auch im Vergleich zu einer Berechnung bei der die Gewichtung für das Loadbalancing von Anfang an höher angesetzt wird, erzeugt der Algorithmus eine bessere Zerlegung was die Grafiken Abb.5.14 bis Abb. 5.18 belegen.

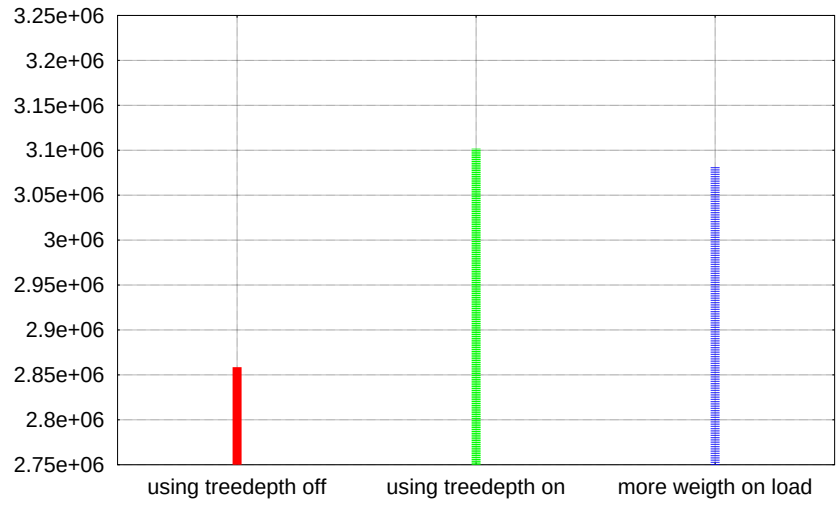


Abbildung 5.14: Updaterate bei 8 Prozessoren

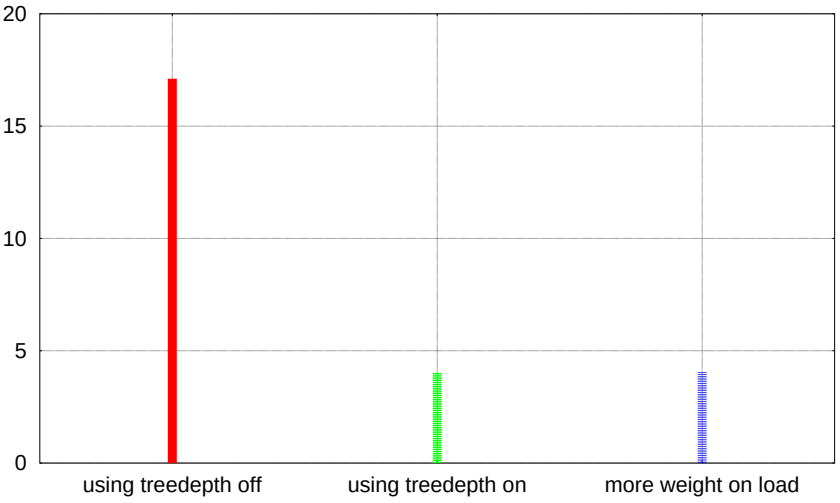


Abbildung 5.15: Loadunbalancing bei 8 Prozessoren

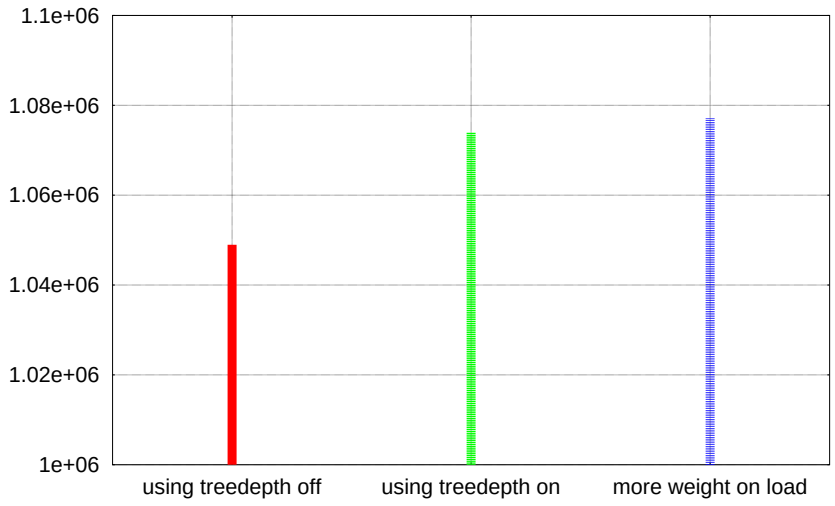


Abbildung 5.16: Kommunikationsvolumen bei 8 Prozessoren

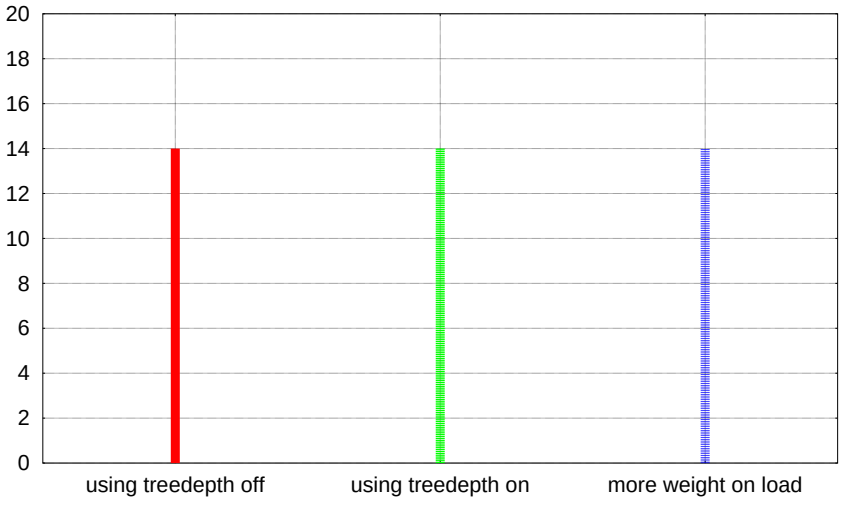


Abbildung 5.17: Anzahl der Messages pro Timestep bei 8 Prozessoren

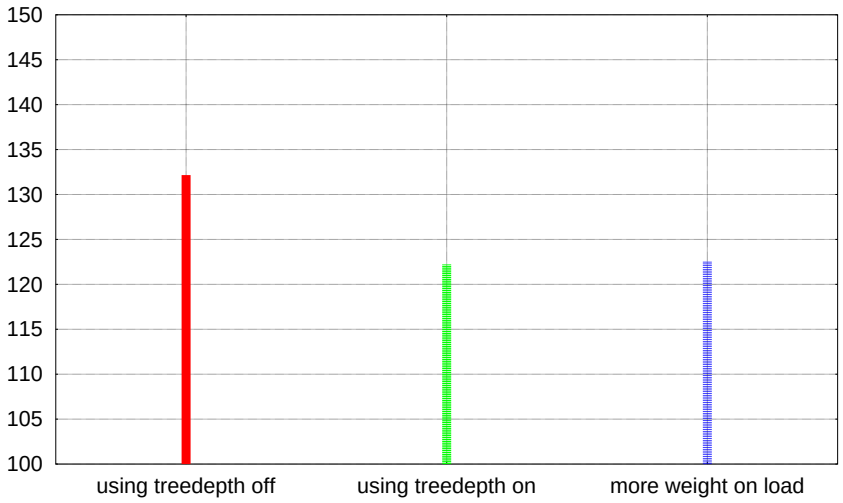


Abbildung 5.18: Zeit für Kollision und Propagation bei 8 Prozessoren

Kapitel 6

Überlegungen und Ausblicke

Die Struktur des Algorithmus von divide berechnet in jedem Zerlegungsschritt, die für den jeweiligen Schritt optimale Stelle. Wenn das Gesamtergebnis als globales Optimum betrachtet wird, so ist die berechnete Teilungsstelle in jedem Schritt ein lokales Optimum. Der Algorithmus errechnet also ein globales Optimum durch rekursive Berechnungen von lokalen Optima. Dieses Vorgehen muß nicht zwangsläufig zu dem qualitativ bestem globalem Optimum führen. Auch wenn die initiale Teilung an der Stelle des lokalen Optimums erfolgte, kann sich diese negativ auf das globale Optimum auswirken. Es stellt sich somit die Frage wie groß der Aufwand ist, eine Prüfung nach dem absoluten globalen Optimum durchzuführen.

6.1 Aufwand einer Zerlegung, die ein globales Optimum durch Berechnung von lokalen Optima erreicht

Die Bipartitionierung des in jedem Berechnungsschritt vorliegenden Gebietes wird durch den Aufruf der Funktion "bipart" erledigt. Als Beispiel soll ein Gebiet in acht Teilgebiete zerlegt werden. Die Vorgänge der rekursiven Zerlegung lassen sich in einer Baumstruktur abbilden. (Abb. 6.1) Um das Gebiet mit dem divide-Algorithmus zu zerlegen sind, abhängig von der Anzahl der zu erzeugenden Teilgebiete (parts), die in Tab. 6.1 aufgeführten Funktionsaufrufe der Teilungsfunktion (bipart) durchzuführen. Die Anzahl der Ebenen n in der Baumstruktur wird als Baumtiefe bezeichnet. In Abbildung 6.1 ist der Ablauf einer Zerlegung in 8 Teilgebiete dargestellt.

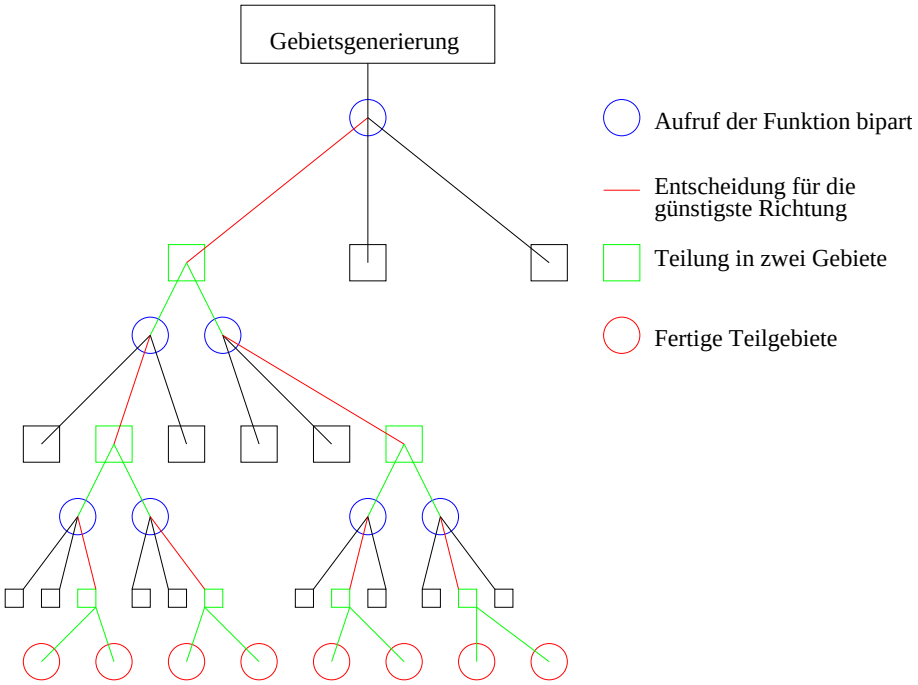


Abbildung 6.1: Zerlege-Algorithmus divide für 8 Teilgebiete

n	parts	bipart
1	2	1
2	4	3
3	8	7
4	16	15
5	32	31
6	64	63
n	2^n	$\sum_0^{n-1} 2^x$

Tabelle 6.1: Aufwand einer Zerlegung ohne Prüfung des globalen Optima

6.2 Aufwand für eine Prüfung des Ergebnisses einer Zerlegung auf ein globales Optimum

Um eine Prüfung des Ergebnisses auf ein globales Optimum durchführen zu können muß man alle möglichen Zerlegungen berechnen. Dazu ist nötig, alle in jedem Schritt möglichen Teilungen weiterzuverfolgen. Diese ergeben sich, wenn man in jeder Ebene das Gebiet in sämtliche Koordinatenrichtungen teilt. Die Struktur eines solchen Vorgehens spiegelt sich in einem vollständigen Baum wieder. So ein Baum hätte die folgenden Strukturmerkmale (Tab. 6.2):

n	parts	biparts	komb.Mögl.	Blätter
1	2	1	3	6
2	4	7	27	36
3	8	43	2187	216
4	16	266	14348907	1296
5	32	1555	$6.1767e14$	7776
6	64	9331	$1.1445e30$	46656
n	2^n	$\sum_0^{n-1} 6^n$	$(3^{2^{n-1}})!$	6^n

Tabelle 6.2: Aufwand einer Zerlegung mit Prüfung des globalen Optima

Dabei ist *komb. Mögl.* die Anzahl der Möglichkeiten, die es gibt, das Gebiet zu zerteilen, wenn alle Koordinatenrichtungen berechnet werden. Die Anzahl der tiefsten Ebene des Baumes werden *Blätter* genannt.
Der Rechenaufwand um die Teilgebiete zu zerlegen steigt mit der Baumtiefe um $\sum_0^{n-1} 6^n$. Nachdem alle durch die Baumstruktur bestimmten Berechnungen ausgeführt wurden, gilt es, aus den $(3^{2^{n-1}})!$ möglichen Kombinationen die zu bestimmen, die ein globales Optimum ergibt. Eine solche Prüfung würde vermutlich länger dauern als die Simulation und ist deshalb nicht zu empfehlen.

6.3 Wäre das Gebiet besser zerlegt worden, wenn für die erste Teilung eine andere Koordinatenrichtung gewählt worden wäre?

Ein Kompromiss wäre möglicherweise die Berechung einer Zerlegung und anschließender Prüfung auf das globale Optima, bei der die initiale Teilung in alle drei Koordinatenrichtungen durchgeführt wird. So eine Berechnung erhöht den Aufwand für eine Zerlegung mindestens um den Faktor drei, da Berechnungen für die Auswertung der drei Ergebnisse ebenfalls erfolgen müssen. Dieses Vorgehen ist von der Teilgebietanzahl unabhängig und somit ein Zusatzaufwand der noch durchführbar ist. Die Rentabilität dieser Erweiterung kann nicht pauschalisiert werden, da diese von der initialen Gebietsstruktur abhängig ist. Sie ist deshalb für jeden Anwendungsfall gesondert zu prüfen.

Kapitel 7

Beispielrechnungen

Um die Funktionalität und die Effektivität zu testen werden einige Beipiele gerechnet. Als Hardware dient ein Linux-Cluster mit 8 Knoten. Jeder Knoten ist mit einen Pentium IV 1.7 GHz, 256kB Cache und 1GB Hauptspeicher ausgestattet. Die Knoten sind via 100MBit Ethernet über einen HP procurve 4000 Fast Ethernet Switch miteinander verbunden. Die verwendete MPI-Implementierung ist lam-mpi in der Version 6.5.4.

7.1 Poröses Medium der Größe 50^3

Als erstes Beispiel wurde der Fluidfluss in einem porösen Medium berechnet. Die Gittergröße betrug 50^3 Knoten, der Grad der Porösität 40,51%. Daraus ergeben sich 77018 Knoten, die für die Simulation relevant sind. Um die Effektivität der Zerlegung zu beurteilen, genügt es, einige hundert Zeitschritte berechnen zu lassen. Für eine vollständige Simulation müssen mehrere zehntausend Zeitschritte berechnet werden. In diesem Beispiel wurden 1000 Zeitschritte berechnet. Die Berechnung wurde auf zwei, vier und acht Prozessoren durchgeführt. Das Gebiet wurde dabei mit dem Preprozessor mit 74 verschiedenen Parametereinstellungen zerlegt, und mit dem Simulator berechnet.
Jeweils für vier und acht Prozessoren mit

- METIS bzw. ParMETIS,
- X-, Y- und Z-Slice,
- divide, mit folgenden Parametern:
  - Gewichtung zwischen Load und Kommunikation 0:100 Area 5,
  - Gewichtung zwischen Load und Kommunikation 0:100 Area 10,
  - Gewichtung zwischen Load und Kommunikation 0:100 Area 20,
  - Gewichtung zwischen Load und Kommunikation 10:90 Area 5,

- Gewichtung zwischen Load und Kommunikation 10:90 Area 10,
- Gewichtung zwischen Load und Kommunikation 10:90 Area 20,
- ...
- Gewichtung zwischen Load und Kommunikation 100:0 Area 5,
- Gewichtung zwischen Load und Kommunikation 100:0 Area 10,
- Gewichtung zwischen Load und Kommunikation 100:0 Area 20,

Die Methoden X-, Y- und Z-Slice sind Algorithmen, mit denen das Gesamtgebiet in die jeweilige Koordinatenrichtung zu gleichen Teilen zerlegt wird. Zum Beispiel soll ein 50^3 Gitter mit der X-Slice Methode (siehe 4.1) in acht Teilgebiete zerlegt werden. Dabei werden acht gleich dicke Scheiben erzeugt, bei denen die x-Achse eine parallele zur Normalen auf die Schnittflächen bildet. Das Verfahren ist einfach zu implementieren und liefert teilweise brauchbare Ergebnisse.
Die zur Beurteilung der Performance relevanten Größen sind die Updaterate $[Updates/s]$, die Loadunbalance (Abweichung von der optimalen Loadbalance), das Kommunikationsvolumen, die Anzahl der Messages pro Zeitschritt und die Zeit, die für Kollision und Propagation benötigt wurden. Die Updaterate beschreibt die Summe der Anzahl der Rechenschritte (Kollision, Propagation, senden und empfangen der Daten) aller an der Rechnung beteiligten Knoten pro Sekunde.
Die Güte der Zerleger wird in den folgenden Abbildungen verglichen.

1. Updaterate bei 4 Prozessoren (Abb. 7.1):
2. Loadunbalancing bei 4 Prozessoren (Abb. 7.2):
3. Kommunikationsvolumen bei 4 Prozessoren (Abb. 7.3):
4. Anzahl der Messages pro Timestep bei 4 Prozessoren (Abb. 7.4):

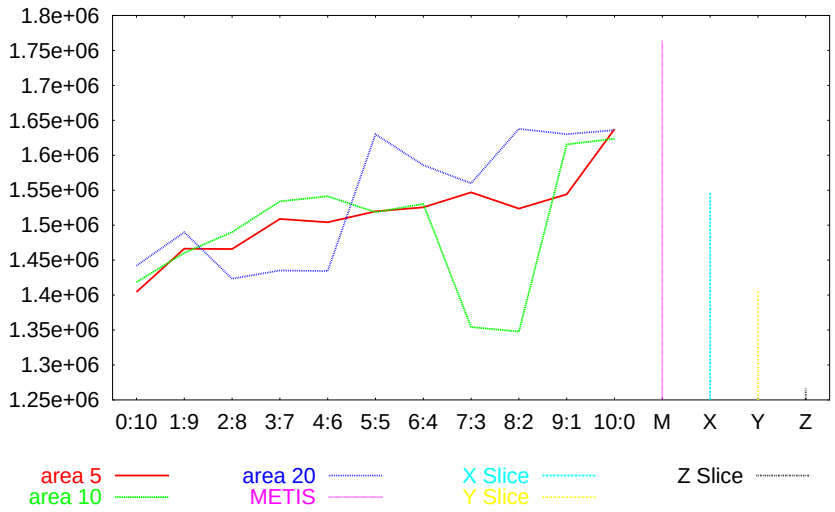


Abbildung 7.1: Updaterate 50^3 bei 4 Prozessoren

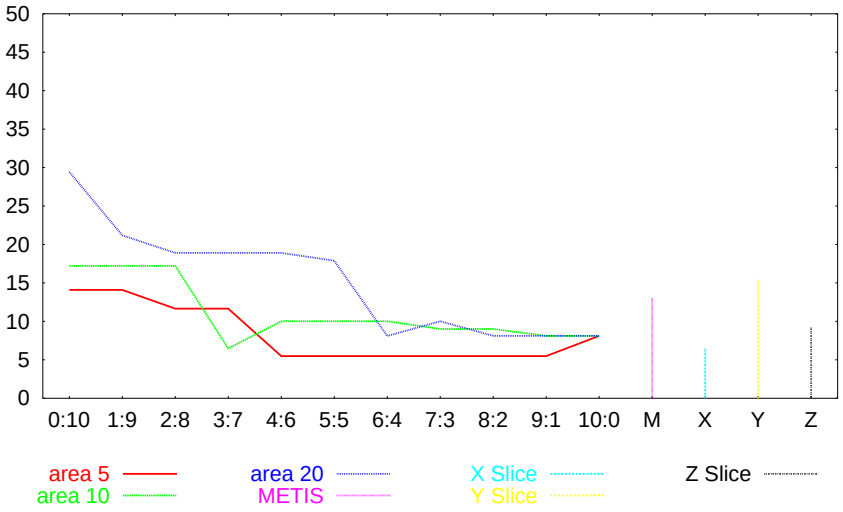


Abbildung 7.2: Loadunbalancing 50^3 bei 4 Prozessoren

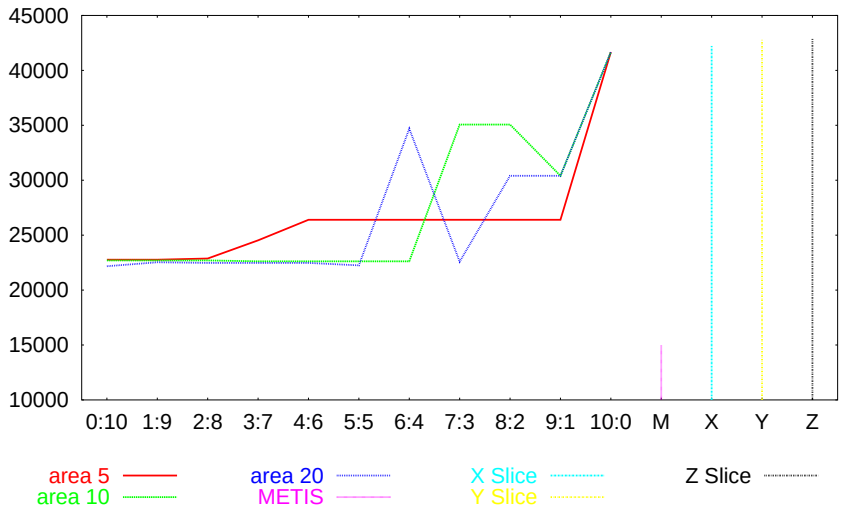


Abbildung 7.3: Kommunikationsvolumen 50^3 bei 4 Prozessoren

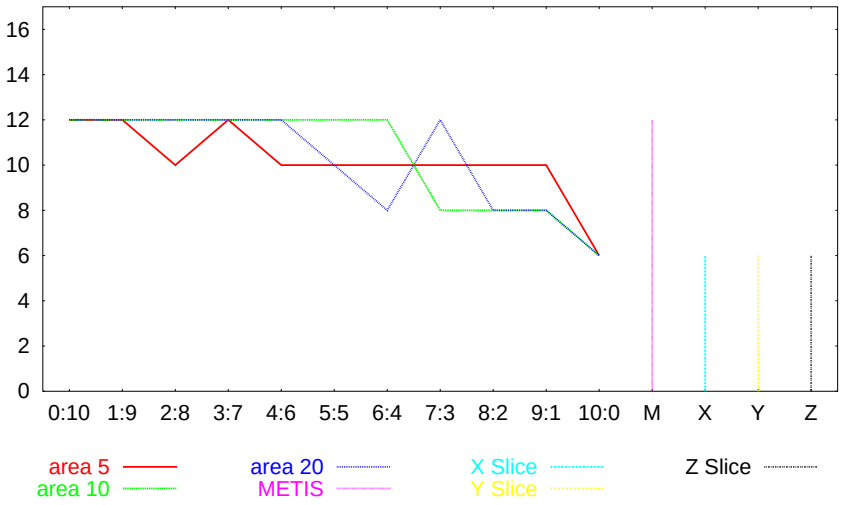


Abbildung 7.4: Anzahl der Messages pro Timestep 50^3 bei 4 Prozessoren

5. Updaterate bei 8 Prozessoren (Abb. 7.5):
6. Loadunbalancing bei 8 Prozessoren (Abb. 7.6):
7. Kommunikationsvolumen bei 8 Prozessoren (Abb. 7.7):
8. Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 7.8):

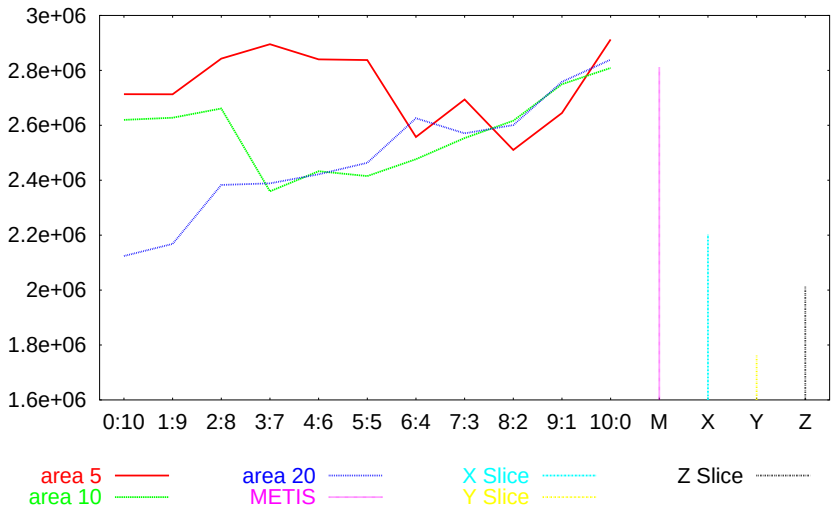


Abbildung 7.5: Updaterate 50^3 bei 8 Prozessoren

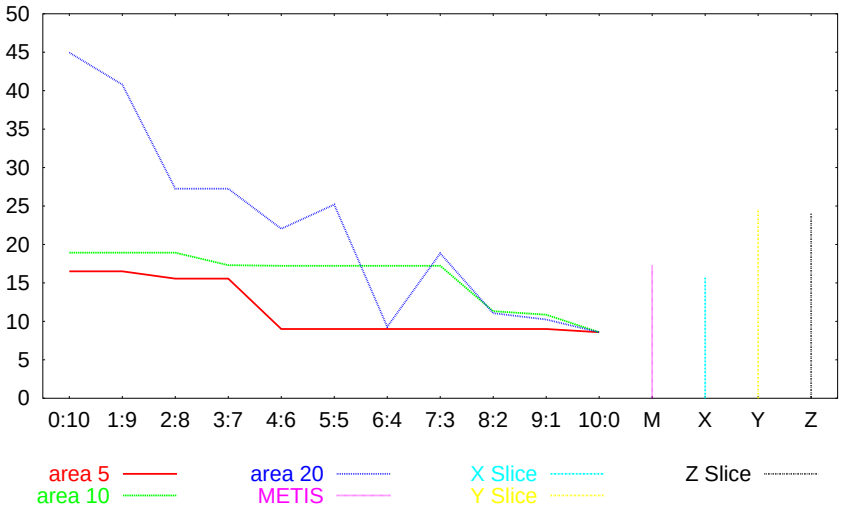


Abbildung 7.6: Loadunbalancing 50^3 bei 8 Prozessoren

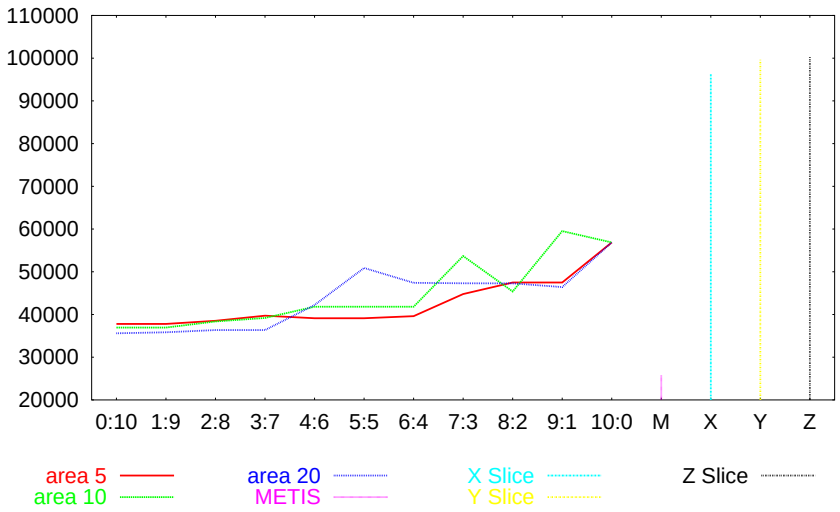


Abbildung 7.7: Kommunikationsvolumen 50^3 bei 8 Prozessoren

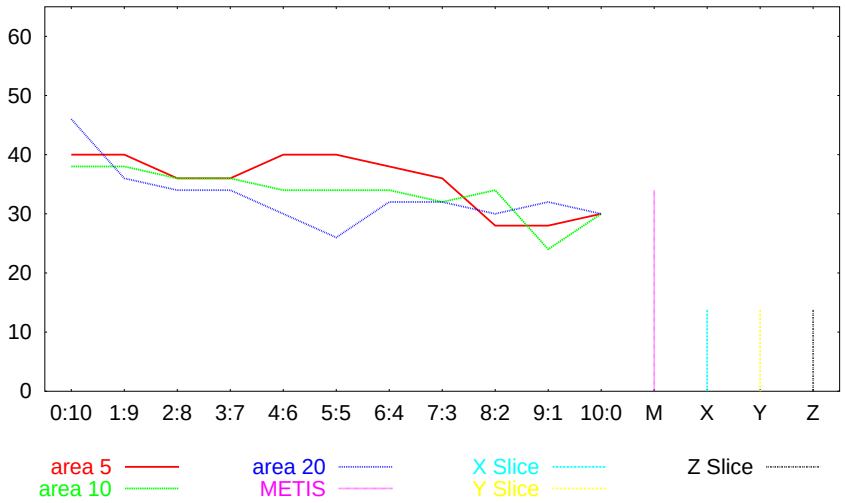


Abbildung 7.8: Anzahl der Messages pro Timestep 50^3 bei 8 Prozessoren

7.2 Poröses Medium der Größe 80^3

Als zweites Beispiel diente ein poröses Medium mit Gittergröße 80^3 Knoten. Der Grad der Porösität lag hier bei 54,86%. Daraus ergeben sich 422220 Knoten, die für die Simulation relevant sind. Das ist bereits über die 5-fache Größe im Vergleich zur Probe aus Beispiel 1. In diesem Beispiel wurden 500 Zeitschritte berechnet. Die Berechnung wurde ebenfalls auf vier und acht Prozessoren durchgeführt. Das Gebiet wurde wie im ersten Beispiel mit sämtlichen Parametereinstellungen zerlegt. Die Darstellung ist analog.

1. Updaterate bei 4 Prozessoren (Abb. 7.9):
2. Loadunbalancing bei 4 Prozessoren (Abb. 7.10):
3. Kommunikationsvolumen bei 4 Prozessoren (Abb. 7.11):
4. Anzahl der Messages pro Timestep bei 4 Prozessoren (Abb. 7.12):

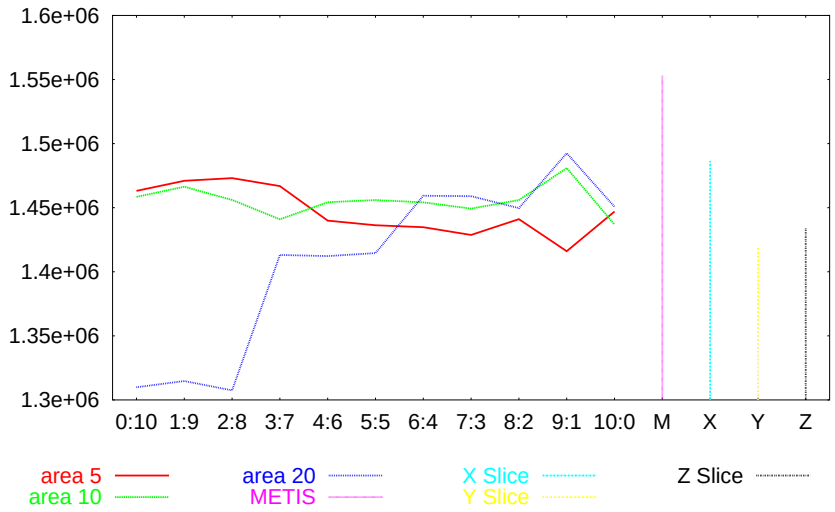


Abbildung 7.9: Updaterate 80^3 bei 4 Prozessoren

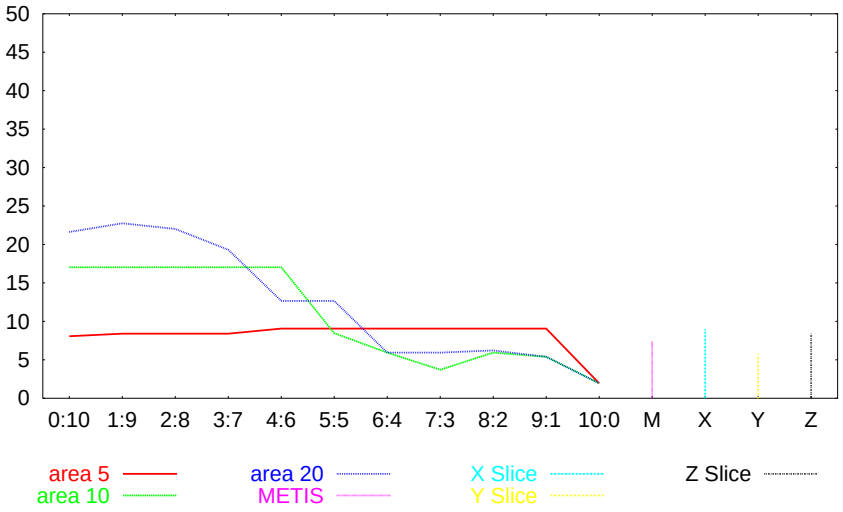


Abbildung 7.10: Loadunbalancing 80^3 bei 4 Prozessoren

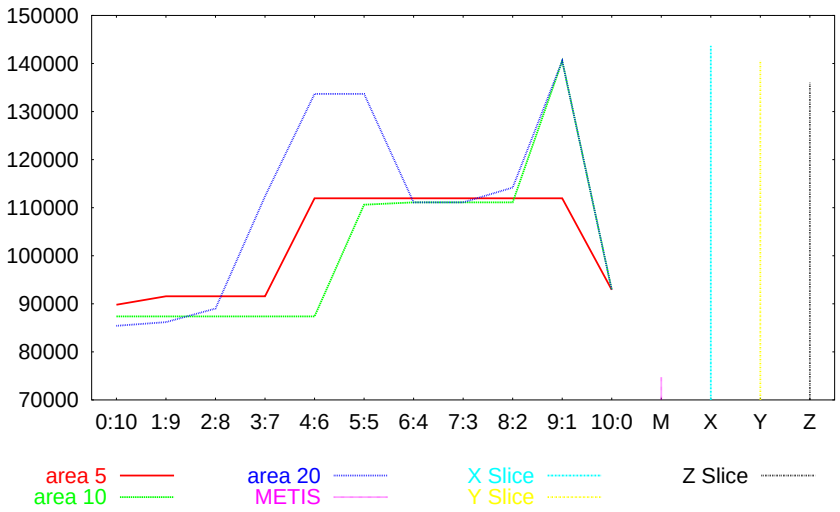


Abbildung 7.11: Kommunikationsvolumen 80^3 bei 4 Prozessoren

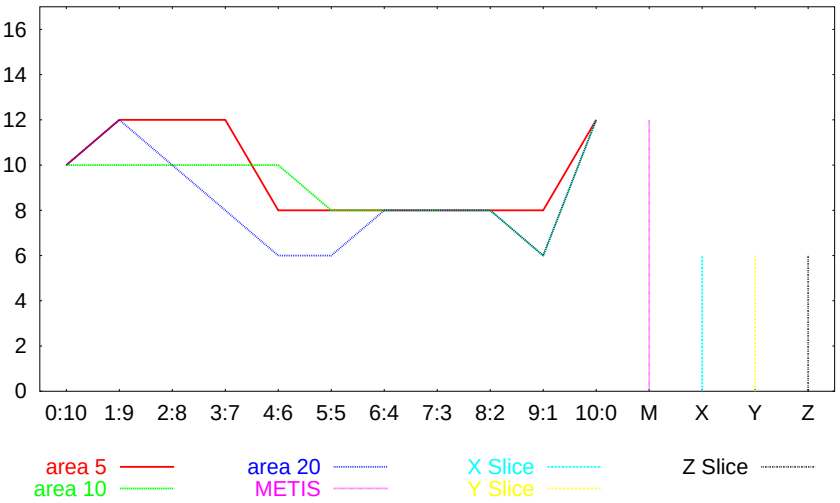


Abbildung 7.12: Anzahl der Messages pro Timestep 80^3 bei 4 Prozessoren

5. Updaterate bei 8 Prozessoren (Abb. 7.13):
6. Loadunbalancing bei 8 Prozessoren (Abb. 7.14):
7. Kommunikationsvolumen bei 8 Prozessoren (Abb. 7.15):
8. Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 7.16):

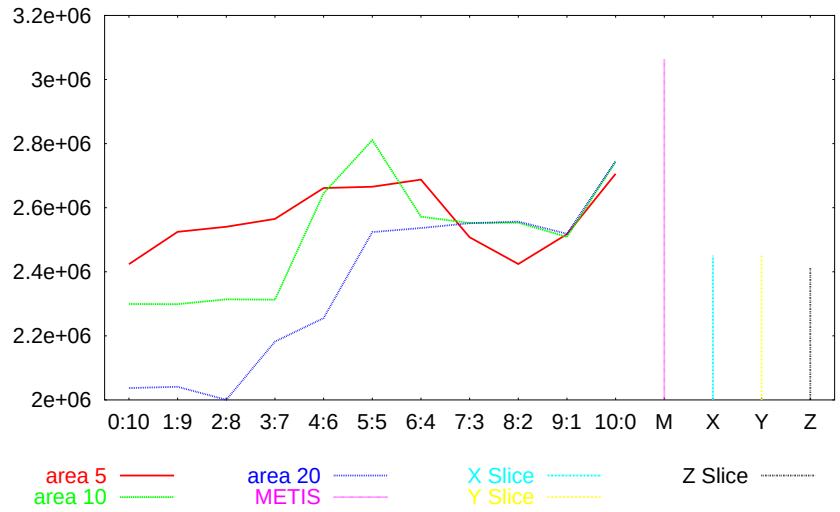


Abbildung 7.13: Updaterate 80^3 bei 8 Prozessoren

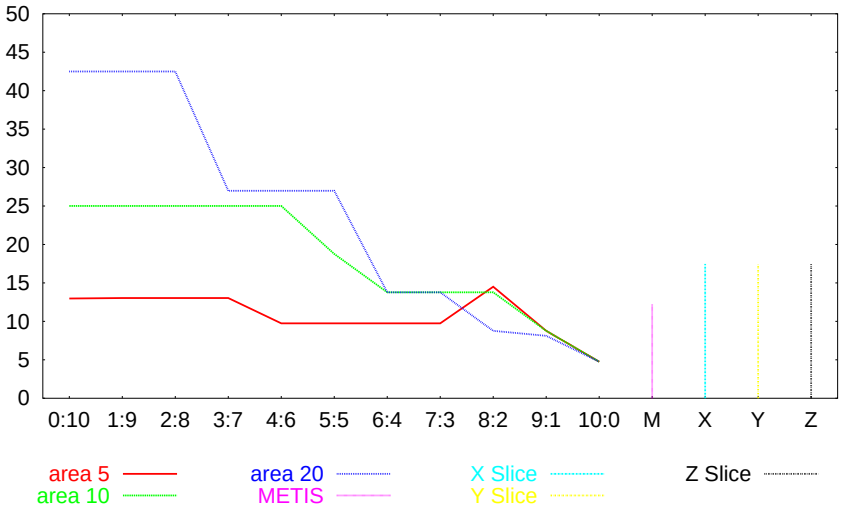


Abbildung 7.14: Loadunbalancing 80^3 bei 8 Prozessoren

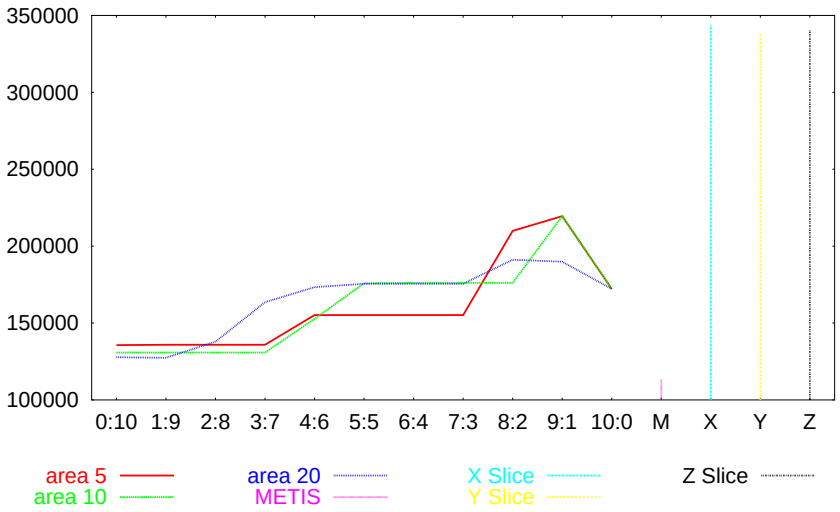


Abbildung 7.15: Kommunikationsvolumen 80^3 bei 8 Prozessoren

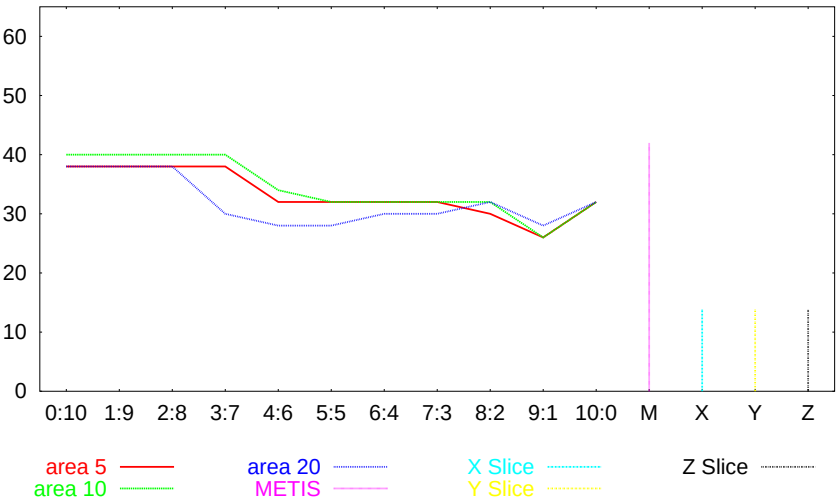


Abbildung 7.16: Anzahl der Messages pro Timestep 80^3 bei 8 Prozessoren

7.3 Poröses Medium der Größe 100^3

Im dritten Beispiel wurde ebenfalls ein poröses Medium mit Gittergröße 100^3 Knoten zerlegt und berechnet. Der Grad der Porösität lag in diesem Beispiel bei 43,10%. Daraus ergeben sich 776561 simulationsrelevante Knoten. Das ist bereits über die 10-fache Größe im Vergleich zur Probe aus Beispiel 1, und fast doppelt so groß wie Beispiel 2. In diesem Beispiel wurden 250 Zeitschritte berechnet. Die Berechnung wurde ebenfalls auf vier und acht Prozessoren durchgeführt. Das Gebiet wurde wie im ersten und zweiten Beispiel mit sämtlichen Parametereinstellungen zerlegt. Die Darstellung ist analog.

1. Updaterate bei 4 Prozessoren (Abb. 7.17):
2. Loadunbalancing bei 4 Prozessoren (Abb. 7.18):
3. Kommunikationsvolumen bei 4 Prozessoren (Abb. 7.19):
4. Anzahl der Messages pro Timestep bei 4 Prozessoren (Abb. 7.20):

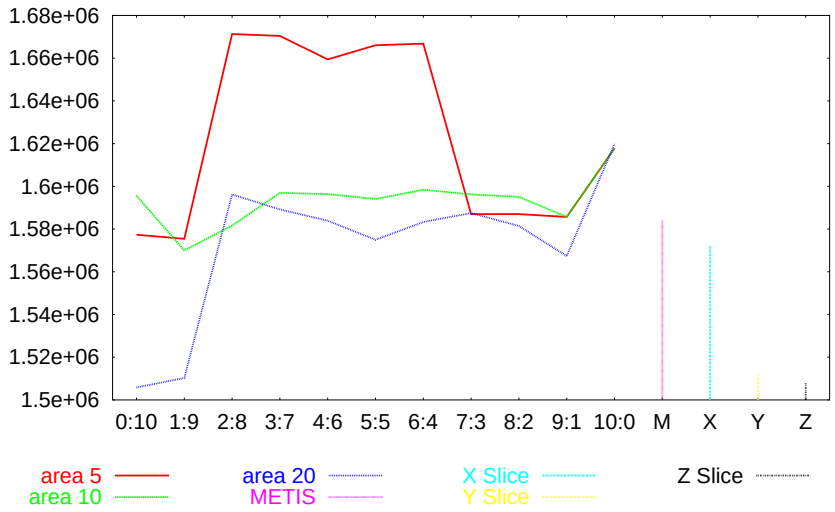


Abbildung 7.17: Updaterate 100^3 bei 4 Prozessoren

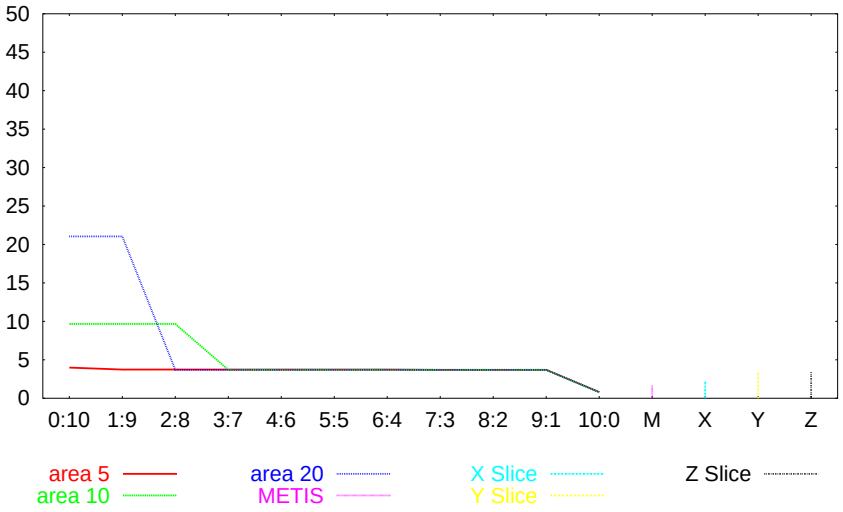


Abbildung 7.18: Loadunbalancing 100^3 bei 4 Prozessoren

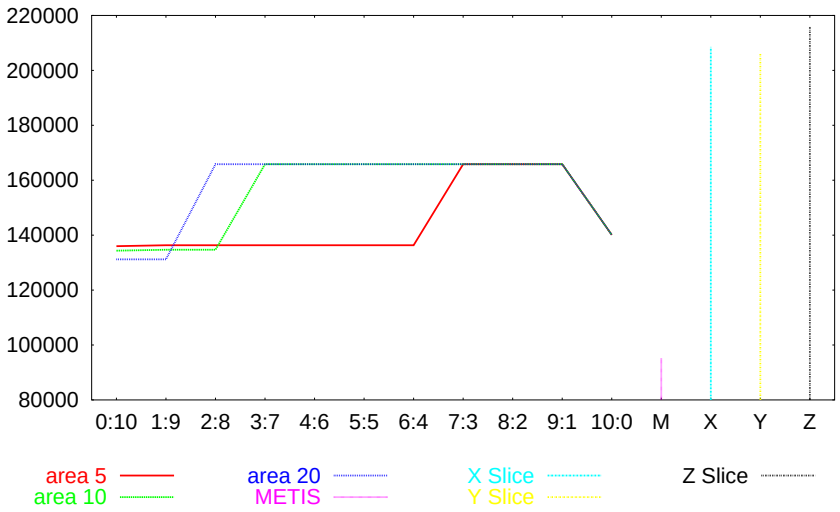


Abbildung 7.19: Kommunikationsvolumen 100^3 bei 4 Prozessoren

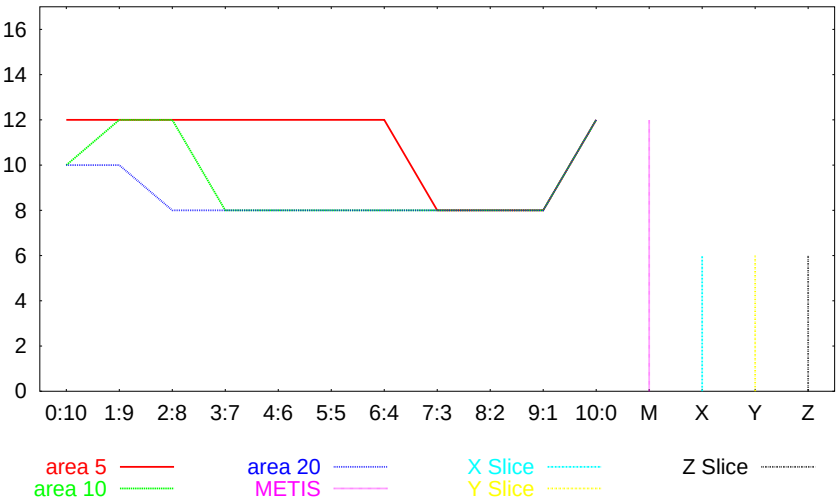


Abbildung 7.20: Anzahl der Messages pro Timestep 100^3 bei 4 Prozessoren

5. Updaterate bei 8 Prozessoren (Abb. 7.21):
6. Loadunbalancing bei 8 Prozessoren (Abb. 7.22):
7. Kommunikationsvolumen bei 8 Prozessoren (Abb. 7.23):
8. Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 7.24):

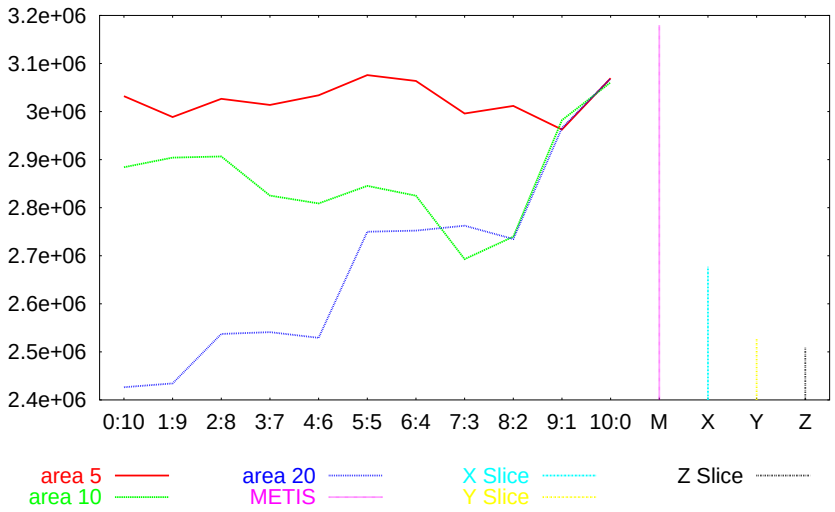


Abbildung 7.21: Updaterate 100^3 bei 8 Prozessoren

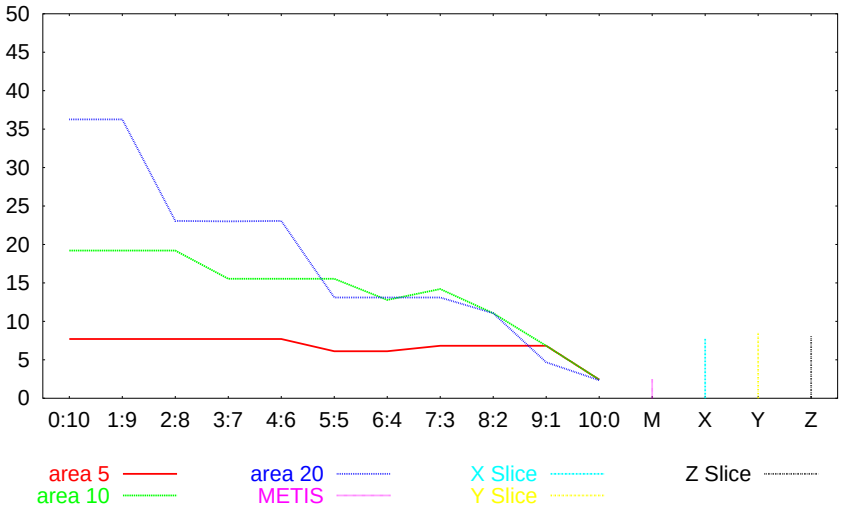


Abbildung 7.22: Loadunbalancing 100^3 bei 8 Prozessoren

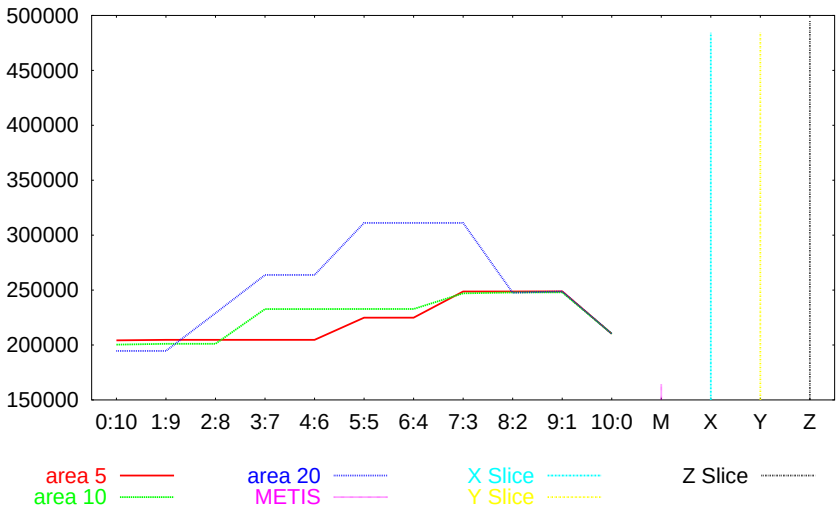


Abbildung 7.23: Kommunikationsvolumen 100^3 bei 8 Prozessoren

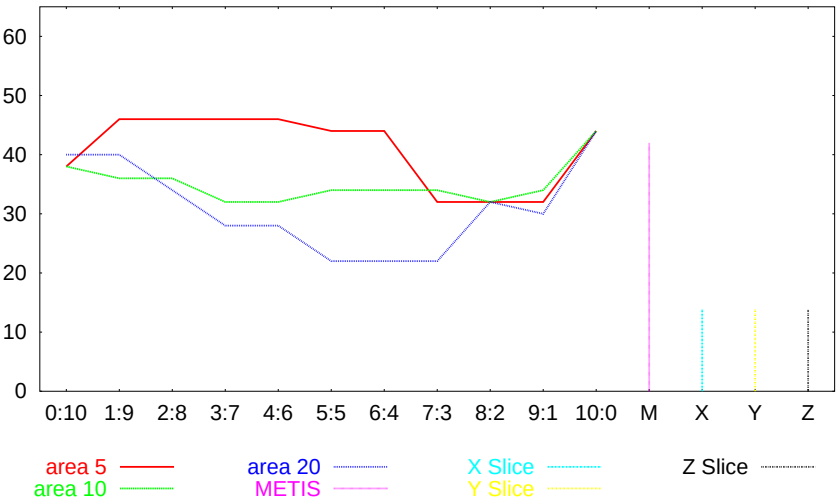


Abbildung 7.24: Anzahl der Messages pro Timestep 100^3 bei 8 Prozessoren

7.4 Poröses Medium der Größe 212^3

Im vierten Beispiel wurde wieder ein poröses Medium mit Gittergröße 212^3 Knoten verwendet. Der Grad der Porösität lag in diesem Beispiel allerdings bei nur 22,08%. Daraus ergeben sich 3734340 simulationsrelevante Knoten. Das ist fast die 50-fache Größe im Vergleich zur Probe aus Beispiel 1, fast 9-mal so groß wie Beispiel 2 und ca. 5 mal größer als Beispiel 3. In diesem Beispiel wurden 100 Zeitschritte berechnet. Die Berechnung wurde auf vier und acht Prozessoren durchgeführt. Das Gebiet wurde wie im ersten und zweiten Beispiel mit sämtlichen Parametereinstellungen zerlegt. Aus Mangel an Hauptspeicher musste die Zerlegung mit METIS einer Zerlegung mit ParMETIS weichen. Die Darstellung ist analog.

1. Updaterate bei 4 Prozessoren (Abb. 7.25):
2. Loadunbalancing bei 4 Prozessoren (Abb. 7.26):
3. Kommunikationsvolumen bei 4 Prozessoren (Abb. 7.27):
4. Anzahl der Messages pro Timestep bei 4 Prozessoren (Abb. 7.28):

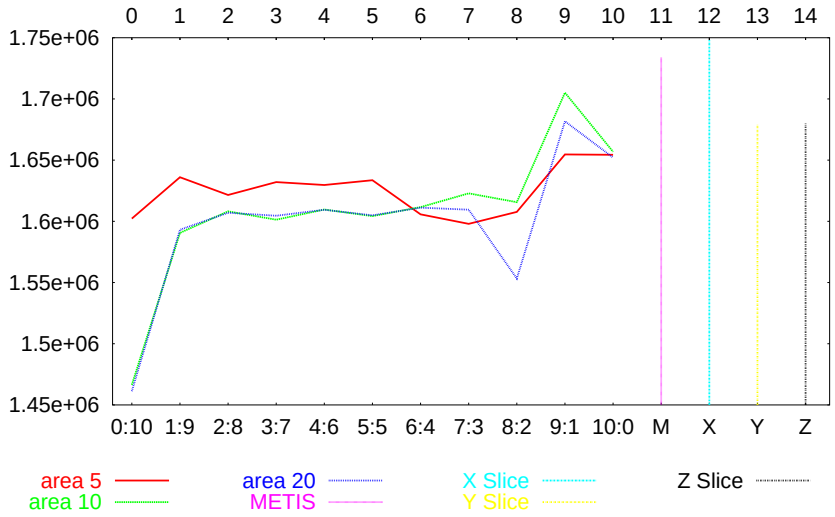


Abbildung 7.25: Updaterate 212^3 bei 4 Prozessoren

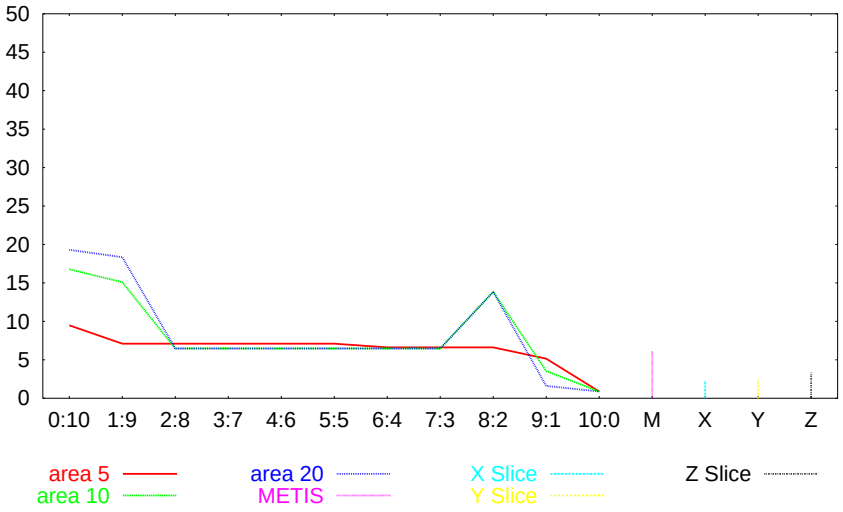


Abbildung 7.26: Loadunbalancing 212^3 bei 4 Prozessoren

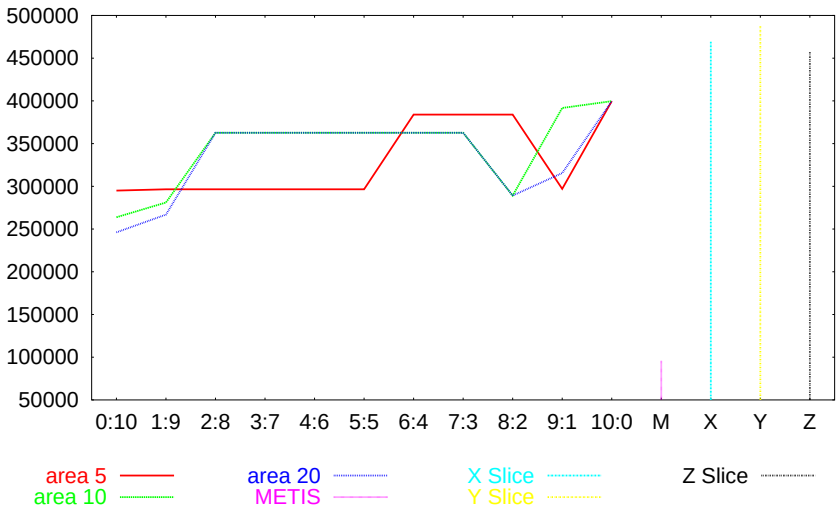


Abbildung 7.27: Kommunikationsvolumen 212^3 bei 4 Prozessoren

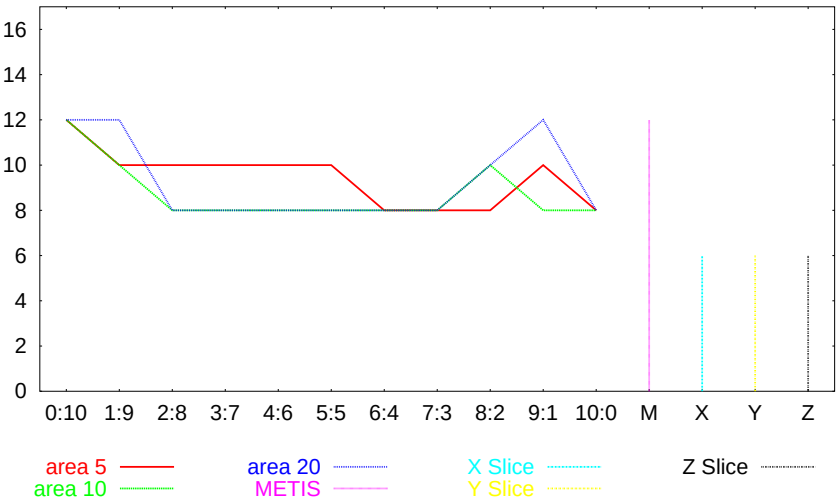


Abbildung 7.28: Anzahl der Messages pro Timestep 212^3 bei 4 Prozessoren

5. Updaterate bei 8 Prozessoren (Abb. 7.29):
6. Loadunbalancing bei 8 Prozessoren (Abb. 7.30):
7. Kommunikationsvolumen bei 8 Prozessoren (Abb. 7.31):
8. Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 7.32):

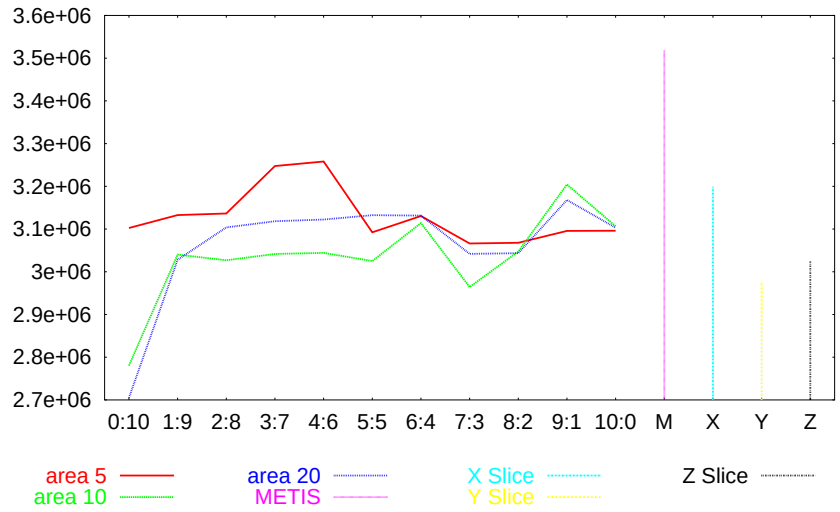


Abbildung 7.29: Updaterate 212^3 bei 8 Prozessoren

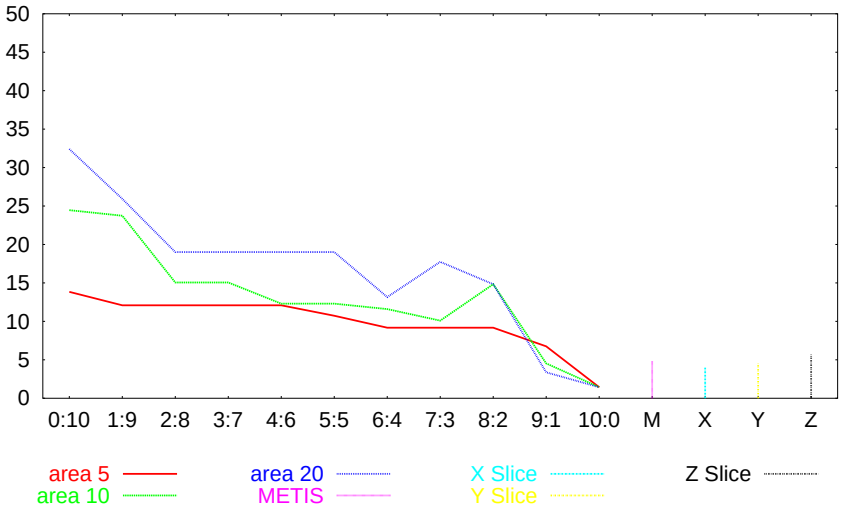


Abbildung 7.30: Loadunbalancing 212^3 bei 8 Prozessoren

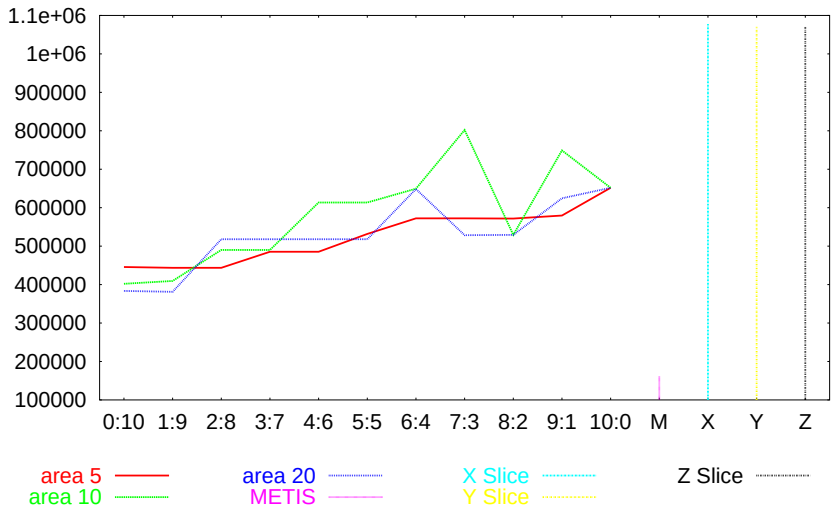


Abbildung 7.31: Kommunikationsvolumen 212^3 bei 8 Prozessoren

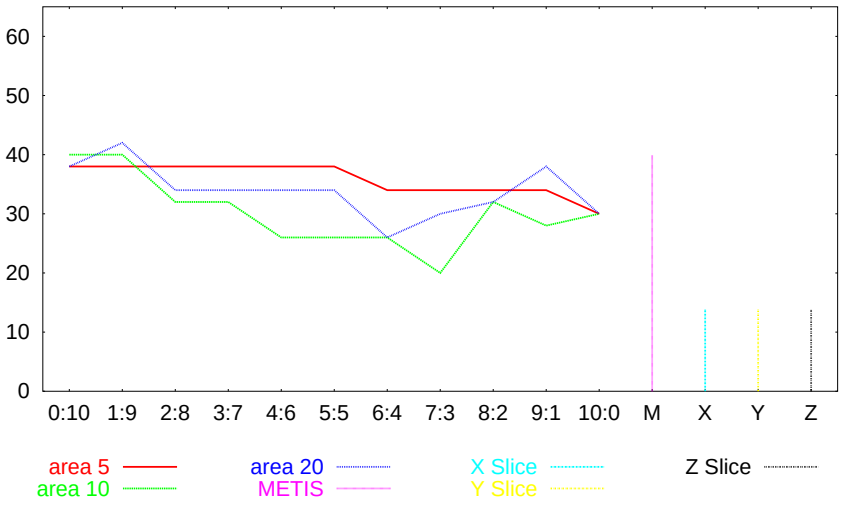


Abbildung 7.32: Anzahl der Messages pro Timestep 212^3 bei 8 Prozessoren

7.5 Poröses Medium der Größe 300^3

Im letzten Beispiel wurde ein poröses Medium mit der Gittergröße 300^3 Knoten gerechnet. Der Grad der Porösität lag in diesem Beispiel wieder höher, nämlich bei 55,37%. Daraus ergeben sich 18282685 simulationsrelevante Knoten. Das ist fast die 240-fache Größe im Vergleich zur Probe aus Beispiel 1, ca. 43 mal so groß wie Beispiel 2, mehr als 23 mal größer als Beispiel 3 und enthält fast 5 mal mehr Knoten als Beispiel 4. In diesem Beispiel wurden 50 Zeitschritte berechnet. Die Berechnung kann jedoch nur noch auf acht Prozessoren durchgeführt werden. Während der Simulation wurde auf jedem Rechenknoten ein Speicherverbrauch von 864MB gemessen. Die Zerlegung mit ParMETIS konnte in dieser Größe nicht mehr erfolgen, da der Preprozessor dazu weit mehr als 3GB Hauptspeicher benötigt hätte. Im Vergleich dazu benötigte die Zerlegung mit divide oder den Slice-Methoden lediglich 1.4GB. Die Zerlegung mit divide erforderte lediglich ca. 60KB zusätzlichen Hauptspeicher. Der hohe Speicherverbrauch des Preprozessors hängt von der Gittergröße und der verwendeten Zerlegungsmethode ab. Wird mit divide oder den Slice-Methoden zerlegt, dann wird kaum mehr Hauptspeicher benötigt, als die von der Gittergröße abhängigen Menge, um die 3D-Matrizen zu speichern. Für METIS müssen zusätzlich große Arrays angelegt werden, um die geometrischen Abhängigkeiten der Knoten zu speichern. Diese sind auch in der parallelen Version von METIS zu groß. Ein 300^3 Gitter liegt schon sehr nahe an der für die hier verwendete Hardwarekonfiguration vorhandene Ausführbarkeitsgrenze. Werden noch größere Gitter zerlegt, muss der Hauptspeicher auf Platte ausgelagert werden. Das hätte inakzeptable Performanceeinbussen zur Folge. Das Gebiet wurde ebenfalls mit sämtlichen Parametereinstellungen zerlegt. Die Ergebnisse mit dem Parameter $AREA = 20$ sind bei niedriger Wichtung für Loadbalancing weit gestreut und wesentlich schlechter als bei Parameter $AREA = 5$ oder $AREA = 10$. Dadurch würde der Darstellungsbereich so breit werden, daß die Kurven für Parameter $AREA = 5$ und $AREA = 10$ nicht mehr deutlich zu unterscheiden wären. Deshalb wurde auf die Darstellung der ohnehin schlechten Ergebnisse mit Parameter $AREA = 20$ in diesem Bereich verzichtet. Die Darstellung ist ansonsten analog zu den vorherigen Beispielen.

1. Updaterate bei 8 Prozessoren (Abb. 7.33):
2. Loadunbalancing bei 8 Prozessoren (Abb. 7.34):
3. Kommunikationsvolumen bei 8 Prozessoren (Abb. 7.35):
4. Anzahl der Messages pro Timestep bei 8 Prozessoren (Abb. 7.36):

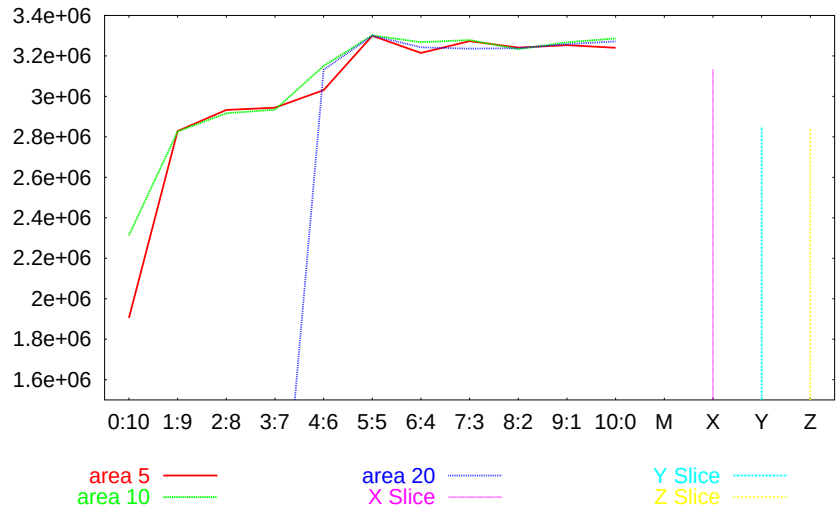


Abbildung 7.33: Updaterate 300^3 bei 8 Prozessoren

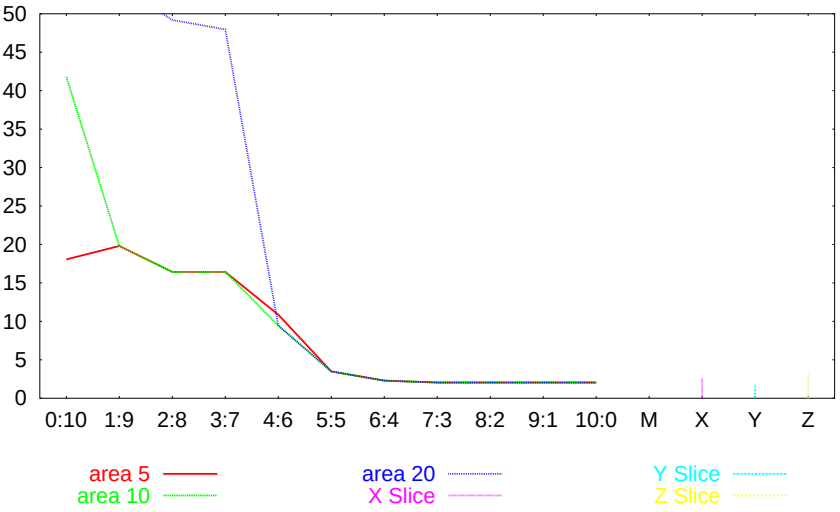


Abbildung 7.34: Loadunbalancing 300^3 bei 8 Prozessoren

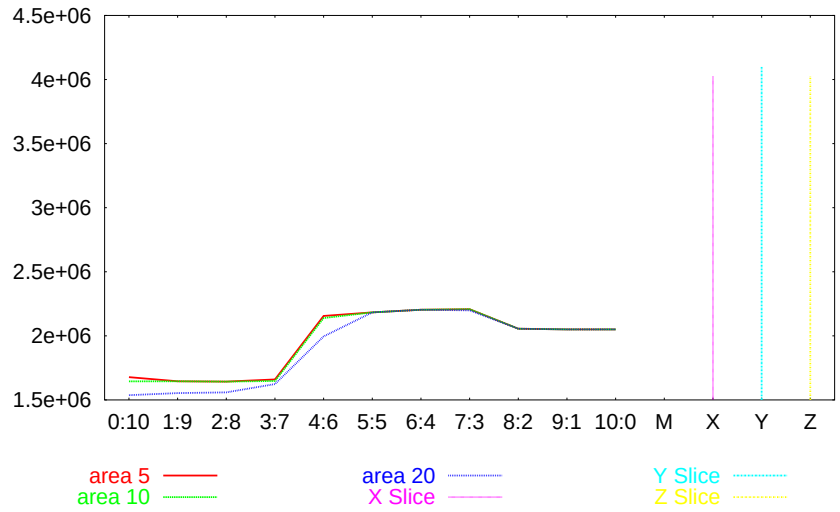


Abbildung 7.35: Kommunikationsvolumen 300^3 bei 8 Prozessoren

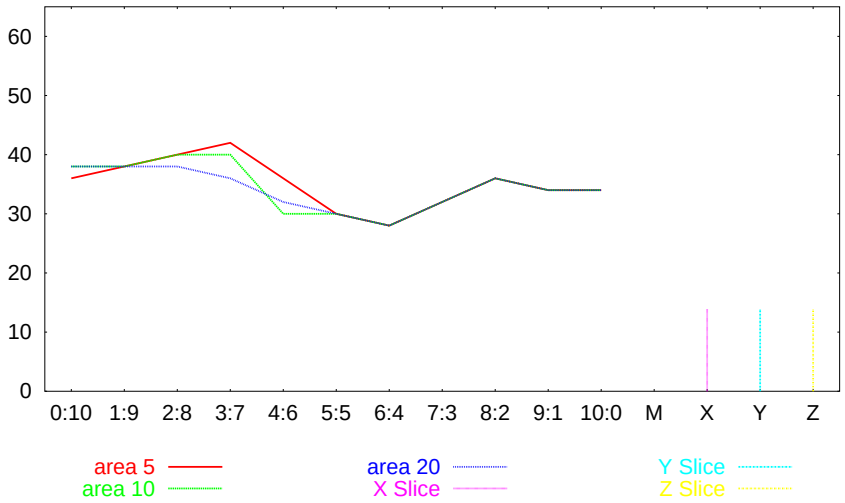


Abbildung 7.36: Anzahl der Messages pro Timestep 300^3 bei 8 Prozessoren

7.6 Auswertung der Ergebnisse

Um die Ergebnisse der Beispielsrechnungen zu bewerten, wurde aus den maximalen Updateraten die Speedups (Tab. 7.1) berechnet. Als Bezug wurde die Updaterate einer seriellen Berechnung verwendet.

DIVIDE	P=1	P=2	P=4	P=8
50^3	1.000	1.831	4.085	6.747
80^3	1.000	1.974	3.397	6.521
100^3	1.000	1.996	3.770	6.921
METIS	P=1	P=2	P=4	P=8
50^3	1.000	2.095	4.081	6.505
80^3	1.000	1.895	3.535	6.974
100^3	1.000	1.944	3.565	7.154
XSLICE	P=1	P=2	P=4	P=8
50^3	1.000	1.914	3.578	5.098
80^3	1.000	1.847	3.383	5.578
100^3	1.000	1.889	3.538	6.025
YSLICE	P=1	P=2	P=4	P=8
50^3	1.000	1.943	3.252	4.076
80^3	1.000	1.842	3.229	5.591
100^3	1.000	1.899	3.401	5.685
ZSLICE	P=1	P=2	P=4	P=8
50^3	1.000	1.771	2.929	4.656
80^3	1.000	1.862	3.264	5.489
100^3	1.000	1.878	3.393	5.644

Tabelle 7.1: Speedups

Vergleicht man die Speedups, so wird deutlich, dass die Slice-Methoden deutlich schlechter als die Methoden divide und METIS bzw. ParMETIS sind. Im Fall 50^3 erreicht METIS auf 2 Prozessoren einen Speedup > 2, divide für 50^3 auf 4 Prozessoren einen Speedup > 4. Dieser extrem hohe Wert unterstreicht die Qualität des parallelen Algorithmus und spiegelt die systembedingten Schwankungen der Performance innerhalb des Clusters, die durch Fremdprozesse entstehen können, wider. Zudem wird die annähernd gleichbleibende Qualität der Zerlegungen mit METIS deutlich.
Desweiteren steigen die Speedups in der Regel mit der Problemgröße. Das läßt darauf schließen, dass bei zu kleinen Problemen (50^3) die Kommunikation länger dauert als die Berechnung der Kollisionen der inneren Knoten. Die Kommunikation kann dadurch nicht erfolgreich im Hintergrund durchgeführt werden. Die Prozessoren werden dabei von der erforderlichen Kommunikation ausgebremst.

Dies wird auch dadurch bestätigt, dass mit zunehmender Problemgröße die Loadbalance mehr Einfluss auf die Performance hat als die Kommunikationsminimierung. Dieser Umstand wird, bis auf einige Ausnahmen, durch die Diagramme für die Updateraten bestätigt.
Da die Speedups für die Berechnung mit divide aus den maximalen Updateraten ermittelt wurden, empfiehlt sich vor einer größeren Berechnungsserie auf ein und derselben Geometrie, eine Parameterstudie der Wichtungsfaktoren für divide, um die für diese Geometrie optimale Kombination zu ermitteln. Grundsätzlich gilt: Je größer die Geometrie, desto wichtiger ein ausgewogenes Loadbalancing. Praktikable Werte für die Gewichtung von Load:Kommunikation sind 80:20 bis 95:5.
Der graphenbasierte Zerleger METIS liefert, solange die Problemgröße eine Anwendung des Algorithmus zulässt, zuverlässig gute Ergebnisse. Die Anwendungsgrenze dieses Zerlegers liegt jedoch bei ca. 200^3 Knoten (je nach Porösität des Mediums), was für Lattice-Boltzmann-Methoden vergleichsweise kleine Geometrien darstellt. Hinzu kommt, dass sich die Qualität der mit divide zerlegten Geometrien mit steigender Problemgröße an die Qualität von METIS annähert. Da bei Lattice-Boltzmann-Verfahren typischer Weise sehr große Geometrien verwendet werden, ist der divide Algorithmus dafür geeigneter als METIS bzw. ParMETIS.

Anhang A

Ergänzungen

A.1 Anmerkungen zum divide Algorithmus

Um sich die Lösungsfindung des Algorithmus von divide besser vorstellen zu können, sind in den folgenden Abbildungen die normierten Werte $load\_diff\_[x|y|z]$, $comm\_[x|y|z]$, sowie deren Kombination $load\_comm\_[x|y|z]$ dargestellt. Die Werte sind der Berechnung der initialen Teilung des Beispiels mit 300^3 Knoten entnommen. Die Parameter der Zerlegung waren:

- FAKload = 50
- FAKcomm = 50
- AREA = 50

Die Load- und Kommunikationswerte an den Minima der drei Kurven werden anschließend noch verglichen. Das Minima dieser Ergebnisse bestimmt die Koordinatenrichtung in die ein Schnitt ausgeführt wird. Die Arrays, welche die Kommunikation beschreiben, sind nicht auf die volle Länge berechnet, sondern nur in dem durch den Parameter "AREA" festgelegten Bereich.

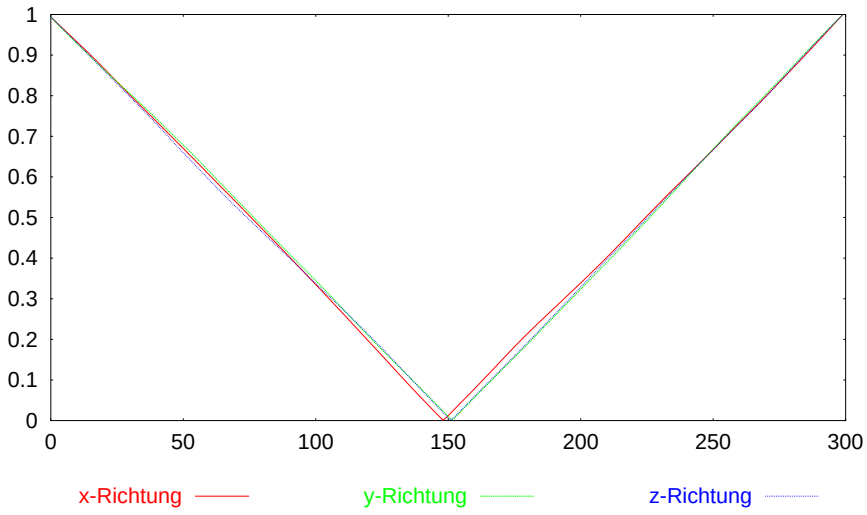


Abbildung A.1: Anzahl der Knotendifferenzen (normiert)

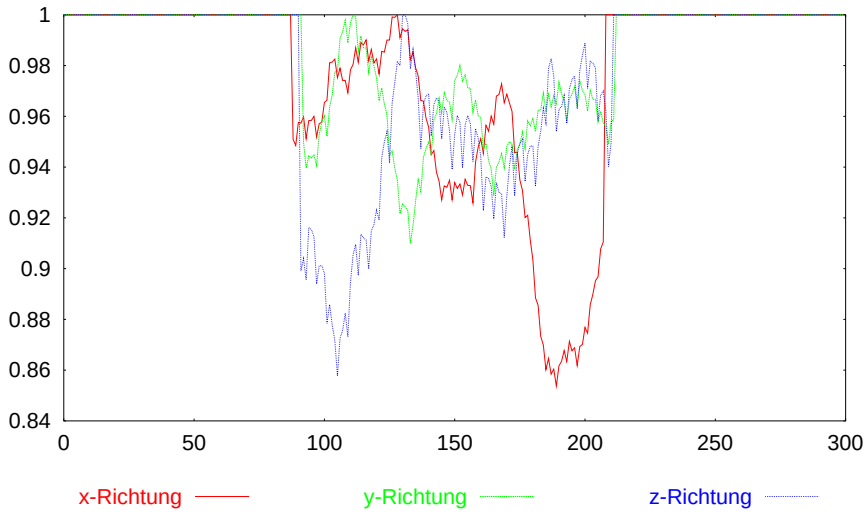


Abbildung A.2: Anzahl der Kanten (Kommunikationsaufwand) (normiert)

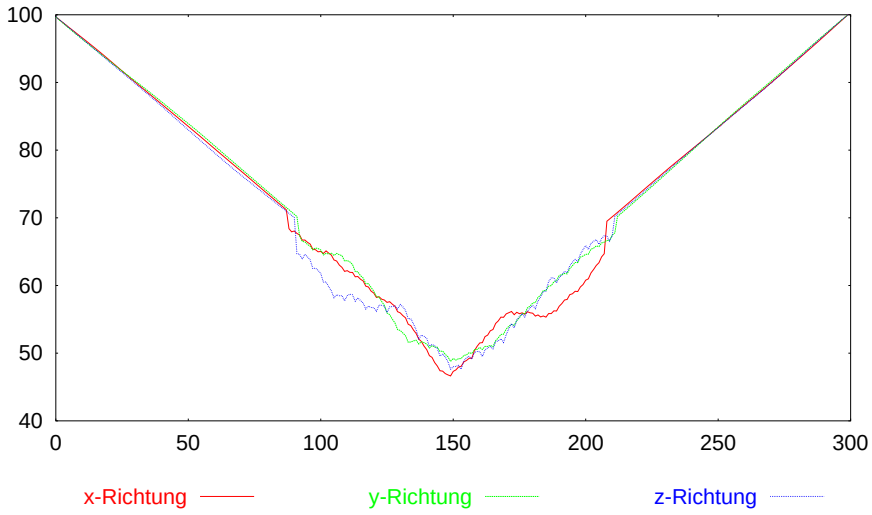


Abbildung A.3: Gewichtet und addierte Kurven A.1 und A.2

A.2 Graphen der Zeit für Kollision und Propagation

A.2.1 Poröses Medium der Größe 50^3

1. Zeit für Kollision und Propagation bei 4 Prozessoren (Abb. 7.1):

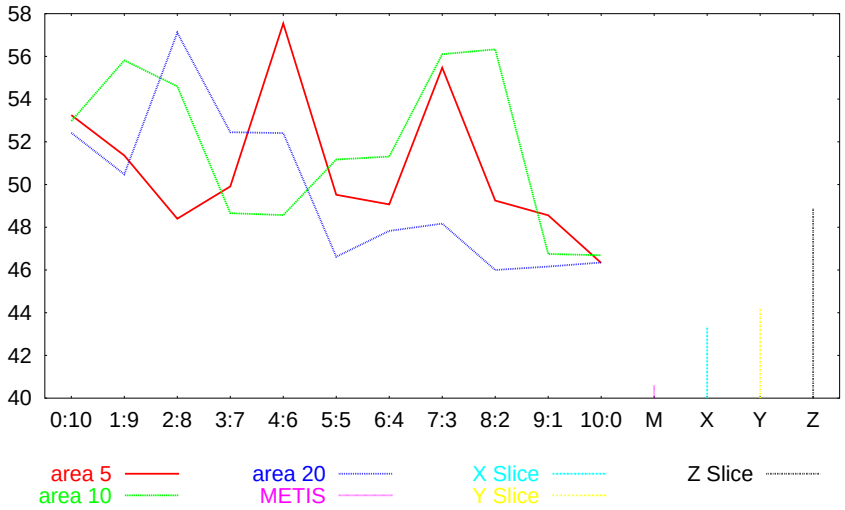


Abbildung A.4: Zeit für Kollision und Propagation 50^3 bei 4 Prozessoren

2. Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 7.5):

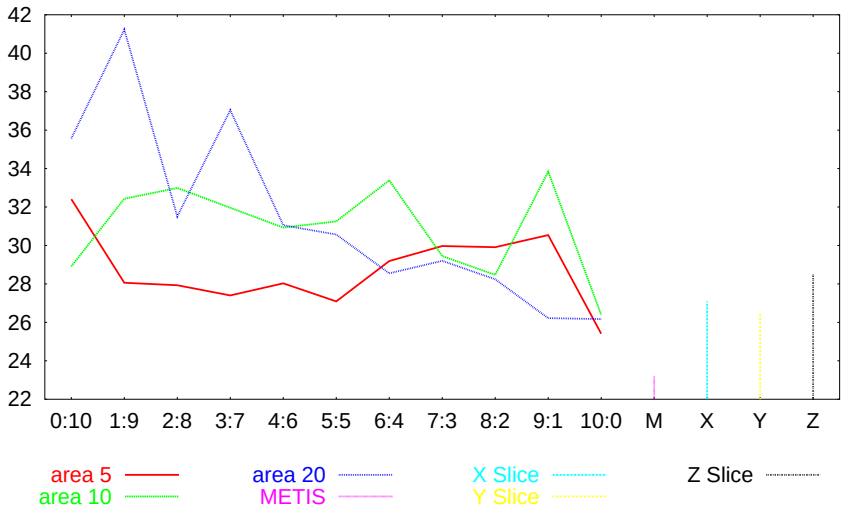


Abbildung A.5: Zeit für Kollision und Propagation 50^3 bei 8 Prozessoren

A.2.2 Poröses Medium der Größe 80^3

1. Zeit für Kollision und Propagation bei 4 Prozessoren (Abb. 7.9):

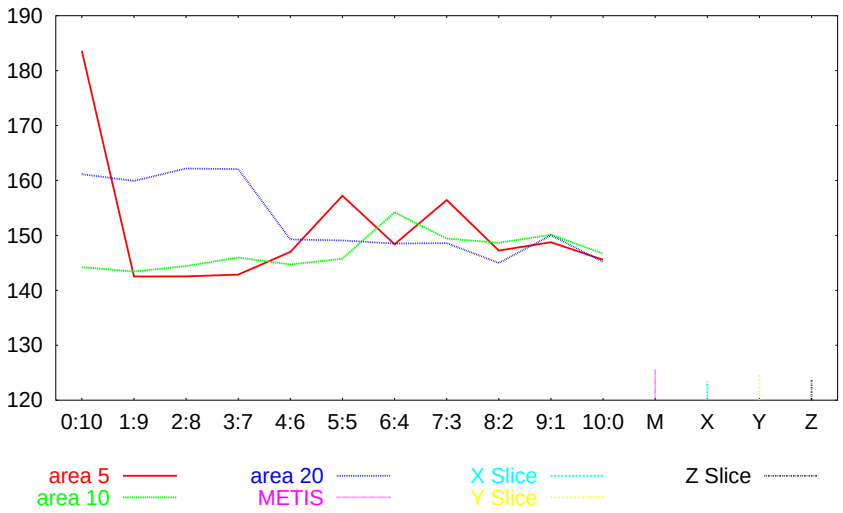


Abbildung A.6: Zeit für Kollision und Propagation 80^3 bei 4 Prozessoren

2. Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 7.13):

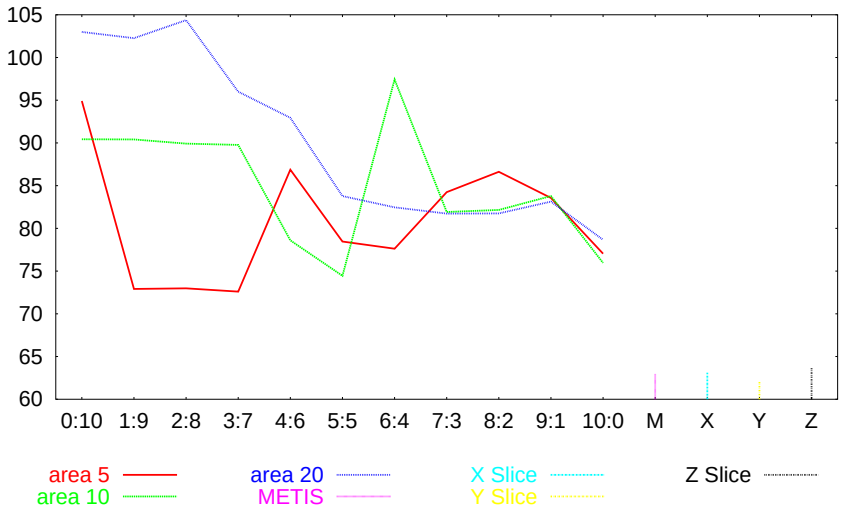


Abbildung A.7: Zeit für Kollision und Propagation 80^3 bei 8 Prozessoren

A.2.3 Poröses Medium der Größe 100^3

1. Zeit für Kollision und Propagation bei 4 Prozessoren (Abb. 7.17):

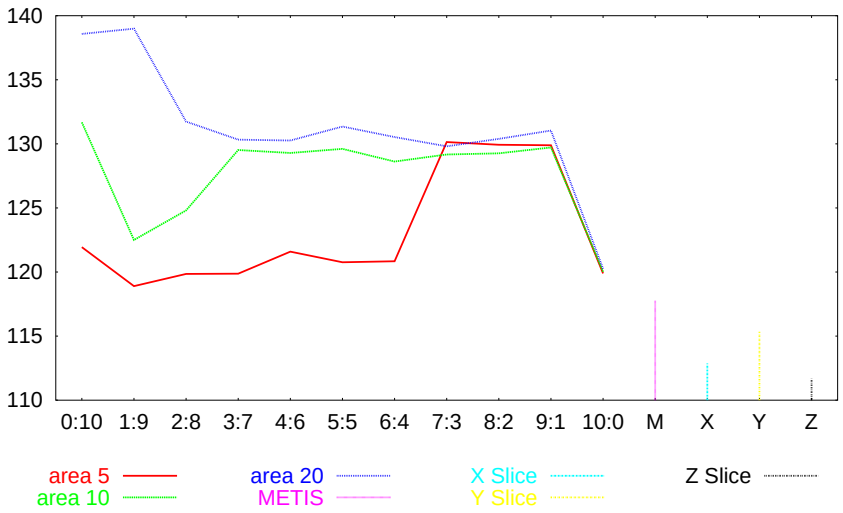


Abbildung A.8: Zeit für Kollision und Propagation 100^3 bei 4 Prozessoren

2. Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 7.21):

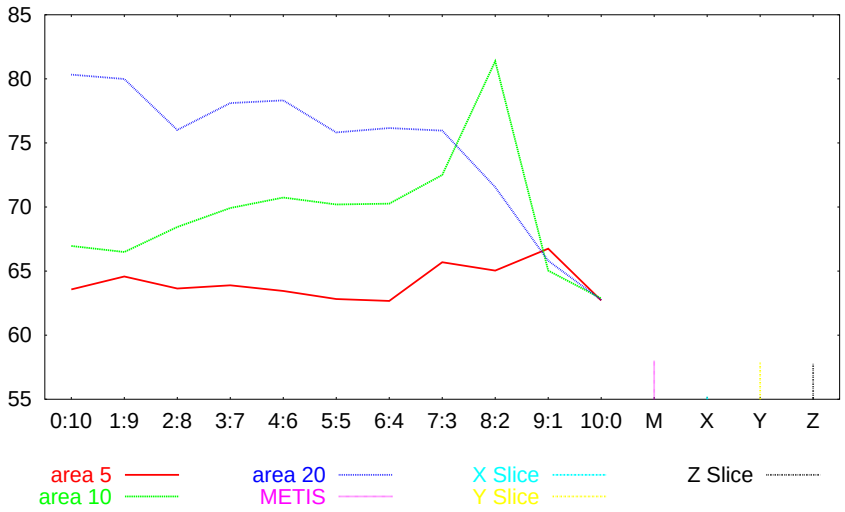


Abbildung A.9: Zeit für Kollision und Propagation 100^3 bei 8 Prozessoren

A.2.4 Poröses Medium der Größe 212^3

1. Zeit für Kollision und Propagation bei 4 Prozessoren (Abb. 7.25):

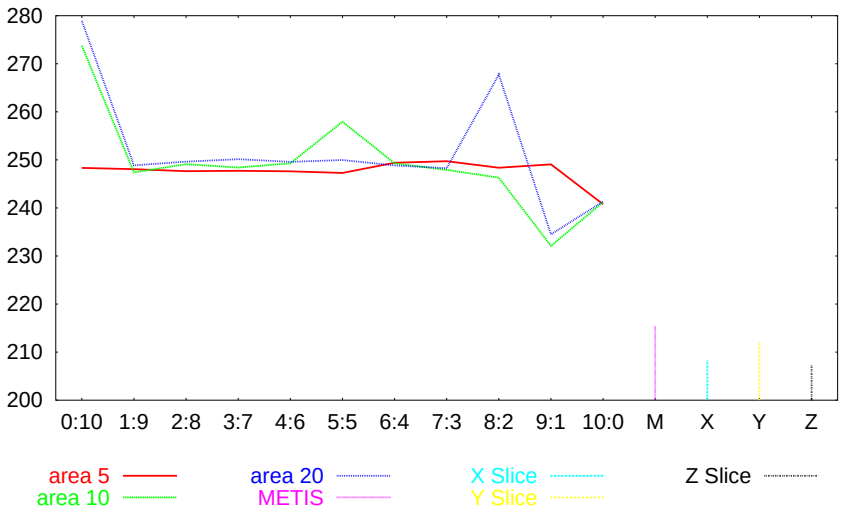


Abbildung A.10: Zeit für Kollision und Propagation 212^3 bei 4 Prozessoren

2. Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 7.29):

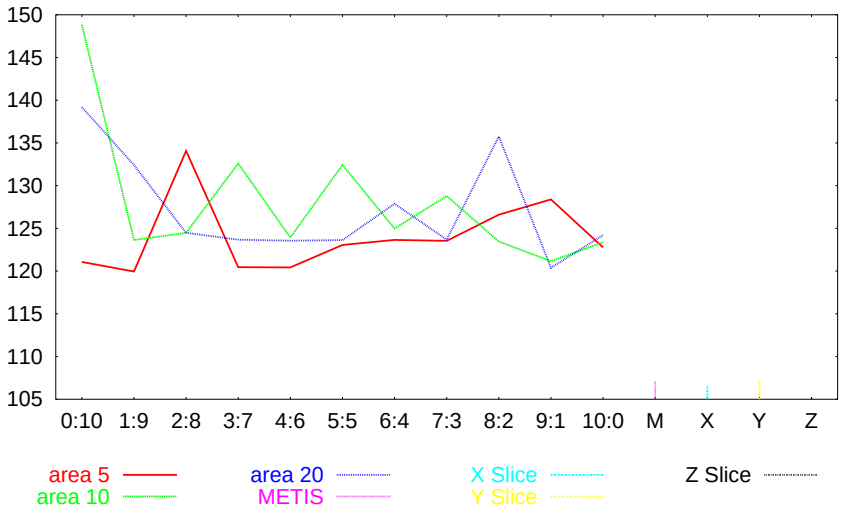


Abbildung A.11: Zeit für Kollision und Propagation 212^3 bei 8 Prozessoren

A.2.5 Poröses Medium der Größe 300^3

1. Zeit für Kollision und Propagation bei 8 Prozessoren (Abb. 7.33):

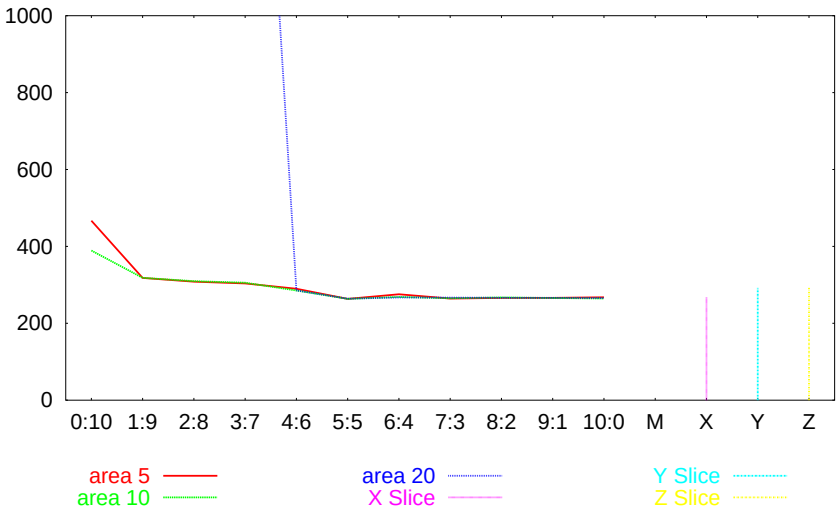


Abbildung A.12: Zeit für Kollision und Propagation 300^3 bei 8 Prozessoren

A.3 Performanceauswertung der Beispielrechnungen in Tabellenform

Tabelle A.1: Datenblatt 50^3 für 4 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	1404378.00	14.09122	53.25	22766.00	12
0:100:10	1418457.00	17.20828	52.97	22700.00	12
0:100:20	1442032.00	29.43651	52.42	22170.00	12
10:90:5	1466341.00	14.09122	51.36	22766.00	12
10:90:10	1460360.00	17.20828	55.81	22700.00	12
10:90:20	1489759.00	21.18016	50.48	22532.00	12
20:80:5	1565848.00	11.65120	48.40	22872.00	10
20:80:10	1490070.00	17.20828	54.60	22700.00	12
20:80:20	1423406.00	18.90310	57.11	22472.00	12
30:70:5	1508863.00	11.65120	49.91	24526.00	12
30:70:10	1534024.00	6.47486	48.66	22618.00	12
30:70:20	1435178.00	18.90310	52.45	22472.00	12
40:60:5	1504138.00	5.48226	57.54	26400.00	10
40:60:10	1541391.00	10.01267	48.57	22618.00	12
40:60:20	1434433.00	18.90310	52.41	22472.00	12
50:50:5	1519726.00	5.48226	49.52	26400.00	10
50:50:10	1518563.00	10.01267	51.17	22618.00	12
50:50:20	1630021.00	17.88828	46.62	22250.00	10
60:40:5	1525497.00	5.48226	49.07	26400.00	10
60:40:10	1530389.00	10.01267	51.31	22618.00	12
60:40:20	1585857.00	8.09710	47.83	34694.00	8
70:30:5	1546950.00	5.48226	55.47	26400.00	10
70:30:10	1354213.00	9.00673	56.10	35066.00	8
70:30:20	1559915.00	10.01267	48.17	22588.00	12
80:20:5	1523694.00	5.48226	49.25	26400.00	10
80:20:10	1347724.00	9.00673	56.33	35066.00	8
80:20:20	1637783.00	8.11487	46.00	30402.00	8
90:10:5	1544114.00	5.48226	48.56	26400.00	10
90:10:10	1615536.00	8.11487	46.75	30402.00	8
90:10:20	1630141.00	8.11487	46.16	30402.00	8
100:0:5	1637432.00	8.11487	46.33	41666.00	6
100:0:10	1623644.00	8.11487	46.69	41666.00	6
100:0:20	1635903.00	8.11487	46.35	41666.00	6
METIS	1764037.00	13.09121	40.63	14976.00	12

X-SLICE	1546931.00	6.48671	43.34	42224.00	6
Y-SLICE	1405686.00	15.48086	44.24	42820.00	6
Z-SLICE	1266121.00	9.25118	48.93	42908.00	6

Tabelle A.2: Datenblatt 50^3 für 8 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	2713272.00	16.50753	32.41	37788.00	40
0:100:10	2619728.00	18.93125	28.91	36936.00	38
0:100:20	2124047.00	44.95663	35.57	35596.00	46
10:90:5	2712870.00	16.50753	28.06	37788.00	40
10:90:10	2627533.00	18.93125	32.43	36936.00	38
10:90:20	2168089.00	40.79512	41.23	35832.00	36
20:80:5	2842771.00	15.55789	27.93	38522.00	36
20:80:10	2661114.00	18.93125	32.99	38368.00	36
20:80:20	2382887.00	27.25131	31.51	36360.00	34
30:70:5	2895303.00	15.55789	27.40	39720.00	36
30:70:10	2359449.00	17.30902	31.96	39190.00	36
30:70:20	2388589.00	27.25131	37.04	36360.00	34
40:60:5	2840107.00	9.00673	28.03	39118.00	40
40:60:10	2432775.00	17.22457	30.93	41820.00	34
40:60:20	2421453.00	22.03942	31.05	42226.00	30
50:50:5	2837568.00	9.00673	27.09	39118.00	40
50:50:10	2415107.00	17.22457	31.25	41816.00	34
50:50:20	2463402.00	25.19056	30.57	50892.00	26
60:40:5	2557461.00	9.00673	29.19	39608.00	38
60:40:10	2476515.00	17.22457	33.39	41816.00	34
60:40:20	2625711.00	9.30007	28.55	47414.00	32
70:30:5	2693958.00	9.00673	29.97	44780.00	36
70:30:10	2553489.00	17.22457	29.45	53684.00	32
70:30:20	2570633.00	18.84829	29.20	47302.00	32
80:20:5	2510224.00	9.00673	29.91	47486.00	28
80:20:10	2616586.00	11.32527	28.47	45386.00	34
80:20:20	2670906.00	11.06453	28.24	47288.00	30
90:10:5	2644771.00	9.00673	30.54	47486.00	28
90:10:10	2749649.00	10.86601	33.84	59520.00	24
90:10:20	2858314.00	10.23934	26.22	46416.00	32
100:0:5	2912621.00	8.58747	25.41	56866.00	30
100:0:10	2808838.00	8.58747	26.39	56866.00	30
100:0:20	2839163.00	8.58747	26.17	56866.00	30
METIS	2812177.00	17.31939	23.22	25840.00	34
X-SLICE	2204038.00	15.73567	27.09	96230.00	14
Y-SLICE	1761900.00	24.72092	26.51	99790.00	14
Z-SLICE	2012620.00	24.09129	28.52	100294.00	14

Tabelle A.3: Datenblatt 80^3 für 4 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	1463114.00	8.06696	183.60	89804.00	10
0:100:10	1458439.00	17.03425	144.22	87378.00	10
0:100:20	1309880.00	21.62580	161.15	85424.00	10
10:90:5	1470974.00	8.39994	142.52	91562.00	12
10:90:10	1466335.00	17.03425	143.42	87378.00	10
10:90:20	1314760.00	22.75281	159.92	86184.00	12
20:80:5	1473066.00	8.39994	142.54	91562.00	12
20:80:10	1456027.00	17.03425	144.43	87378.00	10
20:80:20	1307465.00	22.01976	162.18	89008.00	10
30:70:5	1466855.00	8.39994	142.86	91562.00	12
30:70:10	1440928.00	17.03425	145.97	87378.00	10
30:70:20	1413107.00	19.31292	162.07	112356.00	8
40:60:5	1439924.00	9.06008	147.00	111956.00	8
40:60:10	1454282.00	17.03425	144.71	87378.00	10
40:60:20	1412243.00	12.65957	149.27	133680.00	6
50:50:5	1436300.00	9.06008	157.22	111956.00	8
50:50:10	1455965.00	8.45800	145.75	110622.00	8
50:50:20	1414583.00	12.65957	149.08	133680.00	6
60:40:5	1434787.00	9.06008	148.39	111956.00	8
60:40:10	1454146.00	5.93675	154.22	111112.00	8
60:40:20	1459336.00	5.93675	148.51	111112.00	8
70:30:5	1428722.00	9.06008	156.44	111956.00	8
70:30:10	1449300.00	3.71057	149.43	111112.00	8
70:30:20	1458997.00	5.93675	148.59	111112.00	8
80:20:5	1441062.00	9.06008	147.24	111956.00	8
80:20:10	1455942.00	5.93675	148.66	111112.00	8
80:20:20	1449664.00	6.21830	144.99	114186.00	8
90:10:5	1416052.00	9.06008	148.76	111956.00	8
90:10:10	1480941.00	5.39459	150.12	140538.00	6
90:10:20	1492534.00	5.39459	150.03	140538.00	6
100:0:5	1446984.00	1.94145	145.56	92916.00	12
100:0:10	1436873.00	1.94145	146.68	92916.00	12
100:0:20	1450708.00	1.94145	145.16	92916.00	12
METIS	1552987.00	7.42113	125.64	74774.00	12
X-SLICE	1486513.00	8.99672	123.42	143686.00	6
Y-SLICE	1418803.00	5.73765	124.60	140584.00	6
Z-SLICE	1434267.00	8.45800	123.73	136000.00	6

Tabelle A.4: Datenblatt 80^3 für 8 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	2423683.00	12.97240	94.91	135632.00	38
0:100:10	2299485.00	25.01080	90.43	130800.00	40
0:100:20	2037105.00	42.49734	102.99	127764.00	38
10:90:5	2544640.00	13.03603	72.91	135866.00	38
10:90:10	2298870.00	25.01080	90.41	130800.00	40
10:90:20	2041064.00	42.49734	102.26	127406.00	38
20:80:5	2540494.00	13.03603	72.98	135866.00	38
20:80:10	2313957.00	25.01080	89.92	130800.00	40
20:80:20	2000492.00	42.49734	104.37	137898.00	38
30:70:5	2564962.00	13.03603	72.60	135866.00	38
30:70:10	2312757.00	25.01080	89.77	130800.00	40
30:70:20	2182004.00	26.97850	96.00	163558.00	30
40:60:5	2661280.00	9.74170	86.87	155164.00	32
40:60:10	2644971.00	25.01080	78.60	152760.00	34
40:60:20	2255387.00	26.97850	92.94	173320.00	28
50:50:5	2665383.00	9.74170	78.46	155164.00	32
50:50:10	2810947.00	18.76785	74.45	175840.00	32
50:50:20	2523783.00	26.97850	83.78	175526.00	28
60:40:5	2687640.00	9.74170	77.62	155164.00	32
60:40:10	2572021.00	13.77624	97.43	176148.00	32
60:40:20	2536551.00	13.77624	82.46	175516.00	30
70:30:5	2507607.00	9.74170	84.24	155164.00	32
70:30:10	2552174.00	13.77624	81.92	176148.00	32
70:30:20	2551598.00	13.77624	81.73	175516.00	30
80:20:5	2424213.00	14.50478	86.62	209920.00	30
80:20:10	2553090.00	13.77624	82.16	176148.00	32
80:20:20	2556755.00	8.76872	81.75	191168.00	32
90:10:5	2517252.00	8.76872	83.56	219498.00	26
90:10:10	2508917.00	8.76872	83.78	219498.00	26
90:10:20	2518507.00	8.11813	83.13	189862.00	28
100:0:5	2705876.00	4.75062	77.04	172178.00	32
100:0:10	2743839.00	4.75062	75.96	172178.00	32
100:0:20	2745950.00	4.75062	78.67	172178.00	32
METIS	3064078.00	12.23485	62.96	113346.00	42
X-SLICE	2450797.00	17.47408	63.11	343878.00	14
Y-SLICE	2456470.00	17.47408	62.15	337930.00	14
Z-SLICE	2411555.00	17.47408	63.79	340984.00	14

Tabelle A.5: Datenblatt 100^3 für 4 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	1607343.00	3.99038	121.94	135986.00	12
0:100:10	1595613.00	9.66595	131.68	134344.00	10
0:100:20	1465847.00	21.04791	138.58	131188.00	10
10:90:5	1675416.00	3.73337	118.90	136330.00	12
10:90:10	1570016.00	9.66595	122.50	134688.00	12
10:90:20	1450266.00	21.04791	138.99	131188.00	10
20:80:5	1671289.00	3.73759	119.85	136330.00	12
20:80:10	1581627.00	9.66595	124.81	134688.00	12
20:80:20	1596208.00	3.69236	131.74	165840.00	8
30:70:5	1670423.00	3.73337	119.87	136330.00	12
30:70:10	1596939.00	3.69236	129.52	165840.00	8
30:70:20	1589178.00	3.69236	130.33	165840.00	8
40:60:5	1659418.00	3.73337	121.59	136330.00	12
40:60:10	1596367.00	3.69236	129.29	165840.00	8
40:60:20	1583916.00	3.69236	130.26	165840.00	8
50:50:5	1666048.00	3.73337	120.76	136330.00	12
50:50:10	1594094.00	3.69236	129.61	165840.00	8
50:50:20	1574958.00	3.69236	131.35	165840.00	8
60:40:5	1666806.00	3.73337	120.84	136330.00	12
60:40:10	1598386.00	3.69236	128.62	165840.00	8
60:40:20	1583292.00	3.69236	130.53	165840.00	8
70:30:5	1586986.00	3.69236	130.14	165840.00	8
70:30:10	1596235.00	3.69236	129.17	165840.00	8
70:30:20	1587491.00	3.69236	129.81	165840.00	8
80:20:5	1587005.00	3.69236	129.93	165840.00	8
80:20:10	1595040.00	3.69236	129.26	165840.00	8
80:20:20	1581458.00	3.69236	130.39	165840.00	8
90:10:5	1585674.00	3.69236	129.89	165840.00	8
90:10:10	1585702.00	3.69236	129.72	165840.00	8
90:10:20	1567342.00	3.69236	131.04	165840.00	8
100:0:5	1617807.00	0.80169	119.89	140118.00	12
100:0:10	1617768.00	0.80169	120.02	140118.00	12
100:0:20	1619846.00	0.80169	120.26	140118.00	12
METIS	1584247.00	1.71543	117.78	95358.00	12
X-SLICE	1572220.00	2.27223	112.87	208554.00	6
Y-SLICE	1511525.00	3.46753	115.34	205970.00	6
Z-SLICE	1508026.00	3.46851	111.70	216116.00	6

Tabelle A.6: Datenblatt 100^3 für 8 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	3032094.00	7.70596	63.57	204226.00	38
0:100:10	2884218.00	19.20770	66.96	200336.00	38
0:100:20	2426435.00	36.26256	80.33	194592.00	40
10:90:5	2988625.00	7.70596	64.58	204700.00	46
10:90:10	2904200.00	19.20770	66.49	201078.00	36
10:90:20	2434392.00	36.26256	79.99	194592.00	40
20:80:5	3026509.00	7.70596	63.64	204700.00	46
20:80:10	2906602.00	19.20770	68.44	201078.00	36
20:80:20	2537321.00	23.05969	76.01	229014.00	34
30:70:5	3013751.00	7.70596	63.89	204700.00	46
30:70:10	2825138.00	15.53970	69.92	232718.00	32
30:70:20	2541075.00	23.00490	78.11	263792.00	28
40:60:5	3033826.00	7.70596	63.45	204700.00	46
40:60:10	2808862.00	15.53970	70.74	232718.00	32
40:60:20	2529191.00	23.05969	78.32	263792.00	28
50:50:5	3075871.00	6.11255	62.82	224890.00	44
50:50:10	2845416.00	15.53970	70.20	232782.00	34
50:50:20	2749989.00	13.10411	75.82	311126.00	22
60:40:5	3063680.00	6.11255	62.67	224890.00	44
60:40:10	2824928.00	12.78535	70.26	232782.00	34
60:40:20	2752264.00	13.10411	76.16	311126.00	22
70:30:5	2995970.00	6.82602	65.69	248716.00	32
70:30:10	2692987.00	14.20596	72.50	247024.00	34
70:30:20	2762534.00	13.10411	75.96	311126.00	22
80:20:5	3011755.00	6.82602	65.04	248716.00	32
80:20:10	2739630.00	11.05837	81.38	247810.00	32
80:20:20	2734857.00	11.05837	71.55	247580.00	32
90:10:5	2962511.00	6.82602	66.75	248802.00	32
90:10:10	2982162.00	6.82602	65.04	247960.00	34
90:10:20	2966015.00	4.68375	65.81	248764.00	30
100:0:5	3069357.00	2.42444	62.71	210228.00	44
100:0:10	3060781.00	2.42444	62.87	210228.00	44
100:0:20	3068912.00	2.34177	62.75	210228.00	44
METIS	3179670.00	2.52243	58.03	164432.00	42
X-SLICE	2677652.00	7.81846	55.25	484506.00	14
Y-SLICE	2526430.00	8.57861	57.86	484472.00	14
Z-SLICE	2508572.00	8.04345	57.82	495344.00	14

Tabelle A.7: Datenblatt 212^3 für 4 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	1602247.00	9.48574	248.33	295006.00	12
0:100:10	1466395.00	16.77805	273.76	263800.00	12
0:100:20	1461238.00	19.30841	278.93	246138.00	12
10:90:5	1636019.00	7.09536	248.07	296526.00	10
10:90:10	1590566.00	15.08915	247.39	281064.00	10
10:90:20	1593132.00	18.34380	248.86	266930.00	12
20:80:5	1621515.00	7.09536	247.65	296526.00	10
20:80:10	1608134.00	6.47447	249.11	362678.00	8
20:80:20	1606988.00	6.47447	249.63	362678.00	8
30:70:5	1632056.00	7.09536	247.73	296526.00	10
30:70:10	1601326.00	6.47447	248.41	362678.00	8
30:70:20	1604551.00	6.47447	250.14	362678.00	8
40:60:5	1629668.00	7.09536	247.62	296526.00	10
40:60:10	1609642.00	6.47447	249.28	362678.00	8
40:60:20	1609446.00	6.47447	249.59	362678.00	8
50:50:5	1633596.00	7.09536	247.28	296526.00	10
50:50:10	1604254.00	6.47447	257.94	362678.00	8
50:50:20	1604921.00	6.47447	249.97	362678.00	8
60:40:5	1605705.00	6.61747	249.40	383942.00	8
60:40:10	1611616.00	6.47447	249.27	362678.00	8
60:40:20	1611165.00	6.47447	248.85	362678.00	8
70:30:5	1597946.00	6.61747	249.72	383942.00	8
70:30:10	1622852.00	6.47447	247.92	362678.00	8
70:30:20	1609507.00	6.47447	248.27	362678.00	8
80:20:5	1607719.00	6.61747	248.37	383942.00	8
80:20:10	1615641.00	13.83876	246.30	289140.00	10
80:20:20	1553257.00	13.83876	267.76	289140.00	10
90:10:5	1654652.00	5.14789	249.07	297030.00	10
90:10:10	1705128.00	3.53333	232.09	391700.00	8
90:10:20	1681744.00	1.60197	234.50	315538.00	12
100:0:5	1654339.00	0.87948	240.77	399692.00	8
100:0:10	1656849.00	0.87948	241.22	399692.00	8
100:0:20	1652099.00	0.87948	241.34	399692.00	8
METIS	1734042.00	6.19447	215.53	96156.00	12
X-SLICE	1748060.00	2.35128	208.38	469614.00	6
Y-SLICE	1679108.00	2.56053	212.18	488562.00	6
Z-SLICE	1679809.00	3.32004	207.42	458288.00	6

Tabelle A.8: Datenblatt 212^3 für 8 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	3102465.00	13.83760	121.07	445680.00	38
0:100:10	2780400.00	24.46353	148.82	402100.00	40
0:100:20	2705085.00	32.40854	139.19	383538.00	38
10:90:5	3132733.00	12.08812	119.95	443608.00	38
10:90:10	3040205.00	23.73398	123.64	409448.00	40
10:90:20	3028092.00	25.93046	132.41	381210.00	42
20:80:5	3136417.00	12.08812	134.07	443608.00	38
20:80:10	3026959.00	15.05557	124.47	490042.00	32
20:80:20	3103816.00	19.01207	124.48	517974.00	34
30:70:5	3247508.00	12.08812	120.46	485380.00	38
30:70:10	3041705.00	15.05557	132.59	490042.00	32
30:70:20	3118339.00	19.01207	123.68	517974.00	34
40:60:5	3258121.00	12.08812	120.43	485380.00	38
40:60:10	3044233.00	12.30227	123.93	613500.00	26
40:60:20	3122250.00	19.01207	123.56	517974.00	34
50:50:5	3092334.00	10.73023	123.06	531708.00	38
50:50:10	3025132.00	12.30227	132.43	613500.00	26
50:50:20	3132460.00	19.01207	123.63	517974.00	34
60:40:5	3130540.00	9.17654	123.65	572300.00	34
60:40:10	3114094.00	11.59572	124.98	649480.00	26
60:40:20	3131519.00	13.16014	127.88	648456.00	26
70:30:5	3066247.00	9.17654	123.54	572300.00	34
70:30:10	2964239.00	10.09241	128.79	802090.00	20
70:30:20	3041925.00	17.73763	123.68	528438.00	30
80:20:5	3067842.00	9.17654	126.60	571760.00	34
80:20:10	3047412.00	14.83062	123.46	529298.00	32
80:20:20	3043496.00	14.83062	135.76	529298.00	32
90:10:5	3095606.00	6.74934	128.39	579544.00	34
90:10:10	3204442.00	4.53118	121.16	748950.00	28
90:10:20	3168073.00	3.37014	120.39	624422.00	38
100:0:5	3096095.00	1.43687	122.77	651802.00	30
100:0:10	3106545.00	1.43687	123.37	651802.00	30
100:0:20	3102951.00	1.43687	124.21	651802.00	30
METIS	3518834.00	4.90713	107.12	161936.00	40
X-SLICE	3199095.00	4.04197	106.70	1077846.00	14
Y-SLICE	2976568.00	4.55186	107.31	1070410.00	14
Z-SLICE	3024695.00	5.66465	105.19	1072496.00	14

Tabelle A.9: Datenblatt 300^3 für 8 Prozessoren

Methode	Updaterate	Loadunbal.in %	kol+prop	$\frac{Messagevolume}{Timestep}$	$\frac{Messages}{Timestep}$
0:100:5	1904938.00	18.05678	466.84	1677808.00	36
0:100:10	2313736.00	41.80866	389.26	1645030.00	38
0:100:20	47620.00	87.44689	19599.76	1536990.00	38
10:90:5	2828741.00	19.80198	317.97	1645530.00	38
10:90:10	2826556.00	19.80198	318.24	1645530.00	38
10:90:20	257137.00	53.73196	3142.86	1553416.00	38
20:80:5	2933091.00	16.43397	308.28	1642850.00	40
20:80:10	2916764.00	16.43397	309.55	1642850.00	40
20:80:20	216065.00	49.19664	4157.27	1558996.00	38
30:70:5	2944215.00	16.43397	303.68	1659824.00	42
30:70:10	2934259.00	16.43397	305.51	1648108.00	40
30:70:20	372686.00	47.96156	2448.31	1623528.00	36
40:60:5	3031018.00	10.87275	289.84	2155220.00	36
40:60:10	3150532.00	9.45688	285.27	2138926.00	30
40:60:20	3131392.00	9.45688	285.47	1995446.00	32
50:50:5	3299455.00	3.49085	263.51	2183034.00	30
50:50:10	3301586.00	3.49085	263.58	2183034.00	30
50:50:20	3300111.00	3.49085	263.84	2183034.00	30
60:40:5	3214649.00	2.29631	275.44	2203630.00	28
60:40:10	3267868.00	2.29631	268.49	2203630.00	28
60:40:20	3242580.00	2.29631	267.19	2203630.00	28
70:30:5	3273133.00	2.04566	264.33	2207976.00	32
70:30:10	3278563.00	2.04566	265.07	2207976.00	32
70:30:20	3235713.00	2.04566	266.72	2198856.00	32
80:20:5	3241161.00	2.04566	266.02	2055648.00	36
80:20:10	3233327.00	2.04566	267.11	2055648.00	36
80:20:20	3238306.00	2.04566	266.35	2055648.00	36
90:10:5	3253177.00	2.04566	266.08	2050238.00	34
90:10:10	3267289.00	2.04566	265.74	2050238.00	34
90:10:20	3257770.00	2.04566	266.06	2050238.00	34
100:0:5	3240306.00	2.04566	267.64	2050238.00	34
100:0:10	3286298.00	2.04566	264.53	2050238.00	34
100:0:20	3271863.00	2.04566	265.64	2050238.00	34
X-SLICE	3136308.00	2.70069	267.88	4025744.00	14
Y-SLICE	2849669.00	1.85486	292.12	4101816.00	14
Z-SLICE	2843994.00	3.21525	293.66	4032602.00	14

A.4 Schnittstellenbeschreibung der Bibliothek *libbipart*

Der divide-Algorithmus ist in der Funktion bipart der Bibliothek *libbipart* implementiert.

A.4.1 Datenstruktur der Matrix

Um mit der Funktion *bipart* eine 3D-Matrix in Teilbereiche zu zerlegen, muss diese in einem bestimmten Format erzeugt und an die Funktion übergeben werden. Die Matrix besteht aus einem Integer-Array
der Länge $matrix[max\_z * max\_y * max\_x]$. Die Werte des Arrays werden mit 0 vorbelegt. Die Eigenschaften eines Knotens mit den Koordinaten (x_k, y_k, z_k) können verändert werden, in dem man den Wert an
der Stelle $[(z_k) * max\_x * max\_y + (y_k) * max\_x + (x_k)]$ manipuliert. Die Eigenschaften der Knoten sind auf das Bitfeld des Integer-Wertes abgebildet.
Im Lattice-Boltzmann Kontext gibt es 3 verschiedene Knotentypen.

- Typ 1: Ein Knoten ist ein Geometrieknoten. Diese Knoten stellen die Festkörper in dem Gebiet dar, welches durchströmt werden soll. Sie sind bei der Simulation nicht relevant.
- Typ 2: Ein Knoten ist ein Fluidknoten. Diese Knoten stellen das fliessende Medium dar.
- Typ 3: Ein Knoten ist ein Randknoten. Randknoten sind ebenfalls Fluidknoten, bei denen jedoch mindestens ein Nachbar kein Fluidknoten ist.

Für die Berechnung relevant sind Knoten vom Typ 2 und 3, während Knoten vom Typ 1 nicht relevant sind. Die Kennzeichnung der Knoten erfolgt durch bitweise Addition der Knoteneigenschaften.
Ist der Knoten (x_k, y_k, z_k) ein Knoten vom Typ 1, so erfolgt die Kennzeichnung des Knotens durch:
$matrix[(z_k) * max\_x * max\_y + (y_k) * max\_x + (x_k)] =$
$matrix[(z_k) * max\_x * max\_y + (y_k) * max\_x + (x_k)] + Code_{Typ1}$
Ist der Knoten (x_l, y_l, z_l) ein Knoten vom Typ 2, so erfolgt die Kennzeichnung des Knotens durch:
$matrix[(z_l) * max\_x * max\_y + (y_l) * max\_x + (x_l)] =$
$matrix[(z_l) * max\_x * max\_y + (y_l) * max\_x + (x_l)] + Code_{Typ2}$
Ist der Knoten (x_m, y_m, z_m) ein Knoten vom Typ 3, so erfolgt die Kennzeichnung des Knotens durch:
$matrix[(z_m) * max\_x * max\_y + (y_m) * max\_x + (x_m)] =$
$matrix[(z_m) * max\_x * max\_y + (y_m) * max\_x + (x_m)] + Code_{Typ3}$
In diesem Kontext sind die Codes wie folgt festgelegt:

$Code_{Typ1} = 2$
$Code_{Typ2} = 1$
$Code_{Typ3} = 128$
Im Grunde ist es möglich alle Knotentypen miteinander zu kombinieren. Im Lattice-Boltzmann Kontext macht das jedoch keinen Sinn, da ein Knoten entweder ein Fluid-Knoten oder ein Geometrie-Knoten sein muss. Wird es bei der Zerlegung notwendig, wegen unterschiedlicher Rechenzeit pro Knotentyp, zwei Knotentypen zu unterscheiden, so müssen die Knoten eindeutig einem Knotentyp angehören. Es ist also nicht möglich, dass ein Knoten vom Typ 2 und gleichzeitig ein Knoten vom Typ 3 ist, was auf Grund der Implementierung theoretisch möglich wäre. Ist keine Unterscheidung der Knotenarten nötig, so werden alle Knoten bearbeitet, die nicht vom Typ 1 sind. Die Codes für die einzelnen Knotentypen können in den Parametern unter options[8] - options[10] festgelegt werden.

A.4.2 Die Funktion *bipart*

int bipart (int* matrix, int max_x, int max_y, int max_z,
int*** gpart, int a, int b,
int xmin, int xmax,
int ymin, int ymax,
int zmin, int zmax,
int* options)

Beschreibung

Die Funktion bipart zerlegt eine 3D-Matrix rekursiv bipartiv
mit dem divide Algorithmus.

Parameter

matrix Mit diesem Parameter wird die 3D-Matrix übergeben.
Die 3D-Matrix ist ein int-Array der Größe
$matrix[max\_z * max\_y * max\_x]$.
Die Geometrie des Zerlegungsgebietes muß in diesem Array
eingearbeitet sein.

max_x, max_y, max_z
Diese Parameter beschreiben die Grenzen der 3D-Matrix

gpart In diesem Parameter wird das Ergebnis der Zerlegung gespeichert.
gpart ist eine 3D Matrix der Form $gpart[Anzahl\_der\_Prozesse, 3, 2]$.
z.B.: Das Gebiet für den ersten Prozessor (Teilgebiet 0)
wird durch folgende Koordinaten beschrieben:
$x_1 = gpart[0,0,0]$, $x_2 = gpart[0,0,1]$
$y_1 = gpart[0,1,0]$, $y_2 = gpart[0,1,1]$
$z_1 = gpart[0,2,0]$, $z_2 = gpart[0,2,1]$
usw.

a,b Diese Parameter beschreiben die Start- und Endnummer
der Teilgebiete in die zerlegt werden soll.
Für eine Zerlegung in 8 Teilgebiete sind diese Parameter
in der Regel $a = 0$ und $b = 7$

xmin, xmax, ymin, ymax, zmin, zmax

Mit diesen Parametern wird der Bereich aus der vollständigen 3D-Matrix **matrix** festgelegt, welches in Teilbereiche aufgeteilt wird. In der Regel wird das Gesamtgebiet zerlegt. Die Parameter sind dann:
$xmin = 0$, $xmax = max\_x$
$ymin = 0$, $ymax = max\_y$
$zmin = 0$, $zmax = max\_z$

options Dieser Parameter ist ein int-Array options[12].Mit diesem Array können die Eigenschaften des divide-Algorithmus beeinflußt werden.

options[0] Dieser Parameter legt fest, ob die Standarteinstellungen (default-Werte) oder die in den Parametern options[1] bis options[11] festgelegten Werte für die Zerlegung benutzt werden, d.h.
options[0] = 0 : default-Werte werden benutzt.
options[0] = 1 : die vom Benutzer festgelegten Optionen werden verwendet.

options[1] Dieser Parameter steuert die Ausgabe auf die Konsole.
options[1] = 0 : keine Ausgaben (default).
options[1] = 1 : nur wenige Ausgaben.
options[1] = 2 : viele Ausgaben (Debug-Modus).

options[2] Dieser Parameter ist ebenfalls für Debug-Zwecke geeignet. Mit dieser Option können Dateien mit den bei der Zerlegung verwendeten Arrays erzeugt werden. Dazu muß ein Verzeichnis ${WORKING_DIR}/out existieren.
options[2] = 0 : keine Ausgabe (default).
options[2] = 1 : Ausgabe erfolgt.

options[3] Dieser Wert legt den Parameter AREA im divide-Algorithmus fest.
options[3] = 0 : Es erfolgt keine Optimierung auf Kommunikationsminimierung.
options[3] = 10: Der Parameter AREA beträgt 10 (default).

options[4] Dieser Parameter legt den Faktor für die Gewichtung des Loadbalancings fest.
options[4] = 50: (default)

options[5] Dieser Parameter legt den Faktor für die Gewichtung der Kommunikationsminimierung fest.
options[5] = 50: (default)

options[6] Dieser Parameter bestimmt das geometrische Modell, welches zur Berechnung des Kommunikationsaufwandes benutzt wird.
options[6] = 0 : D3Q15 (default)
options[6] = 1 : D3Q19

options[7] Mit diesem Parameter wird der Gewichtungsfaktor für eine Berechnung mit Unterscheidung der Knotenarten festgelegt. Der Faktor wirkt auf die Anzahl der Knoten, die dem Knotentyp 2 entsprechen. Der Faktor muss in % angegeben werden, z.B.
options[7] = 0 : keine Knotengewichtung (default).
options[7] = 500 : die Rechenzeit für einen Knoten vom vom Typ 2 ist um den Faktor 5 höher als die Rechenzeit für einen Knoten vom Typ 3.

options[8] Code für Knotentyp 1.
Dieser Parameter bezeichnt den Code, für die zur Berechung nicht relevanten Knoten.
options[8] = 2 : (default)

options[9] Code für Knotentyp 2.
Dieser Parameter legt den Code der Knoten fest, die bei Berücksichtung verschiedener Knotentypen, mit dem Gewichtungsfaktor der durch options[8] festgelegt wird, multipliziert wird.
options[9] = 1 : (default)

options[10] Code für Knotentyp 3.
Dieser Parameter legt den Code der Knoten fest, bei dem die Anzahl solcher Knoten, bei Berücksichtigung verschiedener Knotenarten mit dem Faktor 1 multipliziert wird.
options[10] = 128 : (default)

options[11] Mit diesem Parameter wird die Berücksichtigung der Baumtiefe bei der Gewichtung zwischen Loadunbalancing und Kommunikationsminimierung ein- bzw. ausgeschaltet.
options[11] = 0 : ohne Baumtiefe (default)
options[11] = 1 : mit Baumtiefe

Anhang B

Glossar

Boundarycut :

Untermenge des Edgecut, wobei eine 1:n Verknüpfung als "1" gewertet wird.

COMPAS :

"CO-operative Micro-Processors in single Address Space"
Eine auf die Hardware der HITACHI SR8000 abgestimmte Bibliothek, für die automatische Verteilung der Rechenlast in Schleifen auf die acht Prozessoren eines SMP-Knotens durch den Compiler (Autoparallelisierung) und die begleitende Hardware- Unterstützung für die Synchronisation (z.B. Cache- Synchronisation aller acht Prozessoren am Anfang und am Ende eines parallelen Codestücks).

Datenstruktur :

Die Organisation einer Menge von zusammengehörigen Daten, welche in koordinierter und systematischer Art und Weise gespeichert werden.

Diskretisierung (geometrische) :

Ein Objekt (z.B. Strömungsgebiet) wird durch Approximation mit einer endlichen Anzahl von diskreten Elementen oder diskreten Punkten repräsentiert.

Edgecut :

Anzahl der Kanten (edges), die Knoten aus verschiedenen Teilgraphen verbinden.

Interface :

Ein Interface ist eine Schnittstelle, die durch ein definiertes Protokoll den Datenaustausch ermöglicht.

Kanten / edges :

Die Verbindung zwischen zwei Knoten heißt Kante. Sie symbolisiert den Datenfluß bei der Berechnung.

Knoten (cluster) :

Ein Knoten ist ein Rechner im Cluster und führt Berechnungen durch.

Knoten (vertex) :

Ein Knoten ist ein Punkt im Raum, der bestimmte Koordinaten und Eigenschaften hat und Daten für die Berechnung hält.

LAM :

LAM ist eine Abkürzung für Local Area Multicomputer und eine MPI Implementierung.

Loadbalancing :

Loadbalancing beschreibt die Verteilung des Rechenaufwandes so, dass die an der Berechnung beteiligten Knoten (Prozessoren) gleichmäßig ausgelastet sind.

METIS / ParMETIS :

METIS ist eine Bibliothek zur Zerlegung von Graphen, mit dem Ziel möglichst gleich große Teilgraphen zu erzeugen und den Edgecut zu minimieren. ParMETIS ist die parallele Version von METIS und stellt MPI Schnittstellen zur Verfügung.

MPI :

MPI ist eine Abkürzung für Message Passing Interface und ein Standard für parallele Programmierung.

Rekursive Funktion :

Rekursive Funktionen haben die Eigenschaft, sich selbst wieder aufzurufen. Man spricht auch von verschachtelten Funktionsaufrufen.

Skalierbarkeit :

Skalierbarkeit bezeichnet die Fähigkeit der verwendeten Hardware und Software ihre Leistung mit zusätzlichem Hardwareeinsatz (Systemerweiterung) zu steigern. Ein Optimum ist erreicht, wenn bei n-fachem Hardwareeinsatz die n-fache Berechnungsgeschwindigkeit erzielt wird.

SMP-Rechner / SMP-Knoten :

SMP (Symetrisches MultiProcessing) bezeichnet einen Rechner (Knoten) mit mehreren CPUs, die auf einen gemeinsamen Hauptspeicher zugreifen.

VirtualFluids :

Ein Strömungssimulator auf Basis von Lattice-Boltzmann-Verfahren, der am Lehrstuhl für Bauinformatik der TU-München entwickelt wurde.

Literaturverzeichnis

[1] M.Schulz / M. Krafczyk / J.Tölke / E. Rank:

Parallelization Strategies and Efficency of CFD computations in complex geometries using Lattice Boltzmann methods on high-performance Computers

Lehrstuhl für Bauinformatik, TU München (2000)

[2] M. Krafczyk:

Gitter-Boltzmann-Methoden: Von der Theorie zur Anwendung

Lehrstuhl für Bauinformatik, TU München (2001)

[3] Matthias Karl:

Grundlegende Untersuchungen zu Workstation-Clustern und deren Verwendung bei Strömungssimulationen sowie Implementierung einer LAM-MPI Applikation zur Datenausdünnung

Lehrstuhl für Bauinformatik, TU München (2001)

[4] Bruce Hendrickson / Tamara G. Kolda:

Graph Partitioning Models for Parallel Computing

Parallel Computing Sciences Dept., Sandia National Labs, Albuquerque, NM,

Computational Sciences and Mathematics Research Dept., Sandia National labs, Livermore, CA

(08/1999)

[5] Bruce Hendrickson / R. Leland:

The Chaco user's guide, version 2.0

Technical Report SAND95-2344, Sandia National Labs, Albuquerque, NM, 1995

[6] Tamara G Kolda:

Partitioning Sparse Rectangular Matrices for Parallel Processing

Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge

[7] George Karypis / Vipin Kumar:

A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs

University of Minnesota, Dept. of Computer Science, Minneapolis, 95-035

for SIAM Journal of Scientific Computing, 1998

[8] George Karypis / Vipin Kumar:

Multilevel k-way Partitioning Scheme for Irregular Graphs

University of Minnesota, Dept. of Computer Science, Minneapolis, 95-064

for SIAM Journal of Parallel and Distributed Computing, 1998

[9] George Karypis / Vipin Kumar:

Multilevel k-way Hypergraph Partitioning

University of Minnesota, Dept. of Computer Science & Engineering, Minneapolis, 98-036

for the 36th Design Automation Conference

[10] George Karypis / Vipin Kumar:

User Manual METIS Version 4.0: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices

University of Minnesota, Dept. of Computer Science, Minneapolis, 1998

[11] George Karypis / Kirk Schloegel / Vipin Kumar:

User Manual ParMETIS Version 3.0: Parallel Graph Partitioning and Sparse Matrix Ordering Library

University of Minnesota, Dept. of Computer Science, Minneapolis, 2002

[12] Message Passing Interface Forum

MPI-2: Extensions to the Message-Passing Interface

University of Tennessee, 1997

[13] Y. Qian and D. d'Humieres and P. Lallemand:

Lattice BGK for Navier-Stokes equation

Europhysics Letters, 17(6):479–484, 1992

[14] Gundolf Haase:

Parallelisierung und Vektorisierung numerischer Algorithmen

Johannes Kepler Universität Linz, 1998

[15] B.W. Kernighan and S. Lin:

An efficient heuristic procedure for partitioning graphs

The Bell System Technical Journal, 1970

[16] C.M. Fiduccia and R.M. Mattheyses:

A linear time heuristic for improving network partitions

In "In Proceeding 19th IEEE Design Automation Conference", pages 175-181, 1982

[17] Amdahl, G.M.:

Validity of the single-processor approach to achieving large scale computing capabilities.

In AFIPS Conference Proceedings vol.30 (Atlantic City, N.J.,Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

Verzeichnis der URLs

Unter den nachfolgenden Internetadressen (Stand: 18. November 2002) sind vorwiegend Informationen über Paralleles Rechnen und Domain-Partitioning zu finden.

[18] http://www.mpi-forum.org

MPI Message Passing Interface

[19] http://www-unix.mcs.anl.gov/mpi/mpich

MPICH, MPI-Implementation

[20] http://www.lam-mpi.org

LAM-MPI, MPI-Implementation

[21] http://www-users.cs.umn.edu/ ~karypis/

Projekte METIS und ParMETIS

[22] http://www.lrz-muenchen.de/services/compute/hlrb/

Höchstleistungsrechner in Bayern (HLRB): The Hitachi SR8000-F1

[23] http://www.lrz-muenchen.de/services/compute/hlrb/system/

Höchstleistungsrechner in Bayern (HLRB): Systembeschreibung der Hitachi SR8000 F1

[24] http://www.openmp.org

OpenMP, Bibliothek zur Entwicklung von Anwendung für Shared Memory Parallelrechner