Institut für Parallele und Verteilte Systeme
Abteilung Simulation großer Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 16

Parallelisierung des Partition of Unity Codes Crass

Albert Ziegenhagel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Marc Alexander Schweitzer
Betreuer:	Dr. Stefan Zimmer
begonnen am:	18. Juni 2012
beendet am:	18. Dezember 2012
CR-Klassifikation:	D.1.3, I.6.0

Kurzfassung

In dieser Arbeit wird eine Parallelisierung der Partition of Unity Methode vorgestellt. Dabei wird ein Datenparallelisierungsansatz verfolgt, welcher eine schlüsselbasierte Baumdarstellung als Grundlage verwendet. Eine dynamische Lastbalance wird mittels raumfüllender Kurven ermöglicht. Es werden Algorithmen vorgestellt, welche die effiziente, dynamische Berechnung von Nachbarn in der parallelen Umgebung erlauben. Experimente mit bis zu 256 Prozessoren zeigen das mögliche, optimale Skalierungsverhalten des hier vorgestellten Verfahrens.

Abstract

In this thesis we present a parallelisation for the Partition of Unity Method. We follow a data parallelization approach, which uses a key based tree representation as basis. A solution to the dynamic load balancing problem will be offered by space filling curves. Algorithms will be presented, that allow an efficient computation of neighbors in the parallel environment. Experiments with up to 256 processes show the possible optimal scaling behavior of the here presented method.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Die „Partition of Unity Method“ (PUM) ist ein effizientes, gitterfreies Simulationsverfahren zum approximativen Lösen elliptischer partieller Differenzialgleichungen. Problem mit einer hohen Anzahl von Freiheitsgraden stellen jedoch auch moderne Computer vor eine große Herausforderung. In dieser Arbeit stellen wir die Parallelisierung eines existierenden PUM Codes „Crass“ auf Basis der Arbeit [8] und [4] von Schweitzer vor. Wir erarbeiten insbesondere ein Verfahren zu dynamischen Lastbalance welches eine effiziente Durchführung der darauf folgenden Schritte in parallel ermöglicht.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Einführung in die PUM: Es werden die mathematischen Grundlagen der PUM zusammengefasst, welche für das Verständnis der Implementierung von Nöten sind. Außerdem wird eine Übersicht über den bestehenden, sequentiellen Crass Code geliefert um den Ausgangspunkt der Arbeit klar zu definieren.

Kapitel 3 – Parallele PUM Es wird die Umsetzung der parallelen PUM vorgestellt. Nach einer Einführung zur Motivation von Datenparallelisierung im Allgemeinen und speziell bei der PUM wird das Erstellen der Überdeckung, mit all den für die PUM benötigten Eigenschaften, erarbeitet. Anschließend wird auf die Themen der Assemblierung des Gleichungssystems und das Lösen in parallel eingegangen.

Kapitel 4 – Resultate Die im vorherigen Kapitel vorgestellten Algorithmen, werden anhand eines Beispiels verifiziert. Dabei werden die theoretischen Komplexitäten der einzelnen Teilschritte mit Messdaten des durchgeführten Experiments verglichen.

Kapitel 5 – Zusammenfassung und Ausblick Es wird eine abschließende Zusammenfassung über die Resultate dieser Arbeit geliefert. Anschließend wird ein Ausblick auf weitere Themen geliefert, welche in dieser Arbeit noch nicht behandelt wurden.

2 Einführung in die PUM

Dieses Kapitel stellt eine Einführung in die mathematischen Grundlagen der umgesetzten Methoden dar. Dabei werden vor allem diese Teile vorgestellt, welche Einfluss auf die parallele Implementierung haben. Es handelt sich dabei größtenteils um eine Zusammenfassung aus den Arbeiten [6], [2], [7], [8]. Weitere Details können in diesen nachgelesen werden.

Im letzten Teil des Kapitels wird eine kurze Übersicht über den bestehenden Crass Code, auf welchem diese Arbeit aufsetzt, gegeben. Dabei wird der Fokus besonders auf jene Komponenten gelegt, die im Zuge dieser Arbeit geändert, angepasst und erweitert werden mussten.

2.1 Problemstellung

Hauptanwendungsgebiet der PUM ist das Lösen von elliptischen Randwertproblemen, die wie folgt definiert werden

$$\begin{aligned} Lu &= f \quad \text{in} \quad \Omega \subset \mathbb{R}^2 \\ Bu &= g \quad \text{auf} \quad \partial\Omega \end{aligned}$$

wobei L einen symmetrischen partiellen Differenzialoperator zweiter Ordnung und B sinnvolle Randbedingungen darstellt.

Um dieses Problem handgreiflicher zu machen beschränken wir uns im Folgenden auf ein einfaches Poisson Problem mit Dirichlet Randwerten

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega \\ u &= g \quad \text{auf} \quad \partial\Omega \end{aligned} \tag{2.1}$$

Die Aufgabe besteht nun darin, eine möglichst gute Näherung für u zu finden, welche alle Restriktionen erfüllt.

2.2 Konstruktion des PUM Raumes

Um die Gleichung 2.1 näherungsweise zu lösen definieren wir in der PUM eine globale Näherungsfunktion u^{PU} wie folgt

$$u^{PU}(x) := \sum_{i=1}^{N} \varphi_i(x) u_i(x)$$

Dabei sind u_i lokale Teillösungen, welche komplett unabhängig von einander gewählt werden können, und nur auf jenen Teilen zur globalen Lösung beitragen, auf welchen $\varphi_i \neq 0$ ist. Die φ_i stellen dabei die Partition der Eins her. Dabei muss insbesondere $\sum_{i=1}^{N} \varphi_i \equiv 1$ gelten.

Was wollen wir also?

Wir wollen lokale Funktionenräume $V_i^{p_i}$ (wobei p_i den Grad des Raumes angibt), um die lokalen Lösungen u_i zu approximieren und diese lokalen Räume anschließend zu einem global gültigen Raum zusammenfassen, aus welchem wir unsere Approximation für u wählen können. Diesen globalen Raum definieren wir mit unserer Partition der Eins als *PUM Raum* wie folgt:

$$V^{PU} := \sum_{i=1}^{N} \varphi_i V_i^{p_i}$$

Um nun also die Partition herzustellen definieren wir sogenannte *Patches* ω_i mit $supp(\varphi_i) = \omega_i$. Als Summe der Patches definieren wir unsere Überdeckung $C_\Omega := \{\omega_i\}$ welche mindestens das gesamte Gebiet Ω überdeckt.

$$\Omega \subset \bigcup_{i=1}^{N} \omega_i$$

Die Erstellung dieser Überdeckung und ihrer Komponenten ω_i ist von entscheidender Bedeutung für die PUM und ihre Parallelisierung. Sie spiegelt sich direkt im Besetzungsmuster unserer Steifigkeitsmatrix wider (siehe 2.3) und ist wesentlicher Bestandteil der vorliegenden Arbeit. Mit dieser Überdeckung haben wir jedoch noch keine Partition der Eins erhalten. Wir können dies jedoch einfach mit der *Methode von Shepard* bewerkstelligen. Wir definieren unsere φ_i wie folgt

$$\varphi_i := \frac{W_i}{\sum_{\omega_j \in C_i} W_j}$$

wobei $C_i := \{\omega_j \in C_\Omega : \omega_i \cap \omega_j \neq \emptyset\}$ die Menge aller geometrischen Nachbarn ω_j des Patches ω_i darstellt, welche einen nicht leeren Schnitt mit diesem haben. Für die Gewichtsfunktionen W_i wählen wir glatte Funktionen mit ihrem lokalen Support auf ω_i. Damit erfüllen wir nun alle Anforderungen an φ_i um eine Partition der Eins zu erhalten, selbst wenn wir unterschiedliche W_i auf jedem Patch wählen, da durch diese Unterschiede durch die Methode von Shepard heraus normalisiert werden. Beschränken wir uns bei unseren Patches ω_i auf d-dimensionale Rechtecke $[x_i - h_i, x_i + h_i]^d$ mit x_i als Zentrum und h_i als Radius der Rechtecke, so ist auch die Wahl der Gewichtsfunktionen W_i einfach. Wir wählen für W_i das Produkt von eindimensionalen Funktionen $W_i(x) = \prod_{l=1}^{d} W_i^l(x^l)$. Zusätzlich können wir eine Transformation $W_i^l(x^l) = \mathcal{W}(\frac{x^l - x_i^l + h_i^l}{2h_i^l})$ anwenden, welche es uns erlaubt den Support von $\mathcal{W}$ auf $[0, 1]$ zu beschränken und somit für alle Patches dieselbe Funktion $\mathcal{W}$ zu verwenden. Als einfache Wahl für $\mathcal{W}$ ergibt sich nun also z.B. ein B-Spline.

Nachdem wir nun also unsere φ_i definiert haben, welche jedoch nicht ausreichend sind um gute Approximationen u^{PU} für u zu erhalten, definieren wir nun noch die lokalen Funktionenräume $V_i^{p_i} = span\langle\psi_i^t\rangle$ mit t als Zählindex über die einzelnen Funktionen auf dem Patch i. Prinzipiell lassen sich beliebig komplexe Funktionen mit speziellem Problembezug einsetzten. In dieser

Arbeit beschränken wir uns jedoch auf Produkte von eindimensionalen *Legendre Polynomen* $\mathcal{L} : [-1, 1] \mapsto \mathbb{R}$. Wir können erneut eine Transformation $\hat{x}^l := x^l - x_i^l + h_i^l$ anwenden, sodass wir

$$\psi_i^t(x) := \prod_{l=1}^{d} \mathcal{L}^{\hat{t}_l}(\hat{x}^l)$$

mit vom Patch unabhängigen $\mathcal{L}^{\hat{t}_l}$ erhalten. Dabei gilt für den Polynomgrad $\hat{t}_l$, dass $\hat{t} = (\hat{t}_l)_{l=1}^d$ mit $\|\hat{t}\|_1 = \sum_{l=1}^d \hat{t}_l \leq p_i$ der Multiindex dessen ist.

2.3 Diskretisierung

Mit dem PUM Raum aus 2.2 können wir nun die Gleichung 2.1 mittels des Galerkin-Verfahrens diskretisieren und erhalten ein lineares Gleichungssystem der folgenden Form

$$A\tilde{u} = \hat{f} \tag{2.2}$$

wobei A unsere Steifigkeitsmatrix, $\hat{f}$ der Vektor der rechten Seite und $\tilde{u}$ unser Lösungs-Koeffizientenvektor ist. Dabei wird A durch eine Bilinearform $a(\cdot, \cdot)$ und $\hat{f}$ durch eine Linearform $l(\cdot)$ wie folgt besetzt

$$\begin{aligned} A_{(i,n),(j,m)} &= a(\varphi_i\psi_i^n, \varphi_j\psi_j^m) \\ \hat{f} &= l(\varphi_i\psi_i^n) \end{aligned}$$

wobei n und m Indices über unsere Funktionen in $V_i^{p_i}$ bzw. $V_j^{p_j}$ sind. Dies bedeutet also dass unsere Matrix aus Blöcken $A_{i,j} = (A_{(i,n),(j,m)})$ besteht, dessen Größen abhängig von den Funktionenräumen auf den entsprechenden Patches sind. Über die schwache Formulierung von 2.1 erhalten wir

$$a(\varphi_i\psi_i^n, \varphi_j\psi_j^m) = \int_\Omega \nabla(\varphi_i\psi_i^n)\nabla(\varphi_j\psi_j^m) - \int_{\partial\Omega} (\varphi_i\psi_i^n)_v(\varphi_j\psi_j^m)$$

$$l(\varphi_i\psi_i^n) = \int_\Omega f(\varphi_i\psi_i^n)$$

Schreiben wir die erste Formeln mittels des Wissens, dass $supp(\varphi_i) = \omega_i$ zu

$$a(\varphi_i\psi_i^n, \varphi_j\psi_j^m) = \int_{\omega_i\cap\omega_j\cap\Omega} \nabla(\varphi_i\psi_i^n)\nabla(\varphi_j\psi_j^m) - \int_{\omega_i\cap\omega_j\cap\partial\Omega} (\varphi_i\psi_i^n)_v(\varphi_j\psi_j^m)$$

um, so wird sofort ersichtlich, dass unsere Matrix A ein dünnes Besetzungsmuster hat, da $a(\cdot, \cdot)$ nur dort ungleich Null ist, wo $\omega_j \in C_i$. Außerdem wissen wir, dass das Besetzungsmuster einer Reihe i durch die Nachbarschaft C_i gegeben ist, da diese genau $\omega_i \cap \omega_j \neq \emptyset$ impliziert.

2.4 Multilevellöser

Um 2.2 effizient zu lösen verwenden wir einen Multilevellöser. Im folgenden geben wir kurze Zusammenfassung über diesen. Details und die genaue Funktionsweise können in [3] nachgelesen werden, wo dieser genau erarbeitet wurde.

Der Multilelelöser nutzt aus, dass wir mehrere Level von Covern C_Ω^k mit $k \in 1, 2, ...J$ und J als höchstem Level erzeugen. Jedes dieser Cover wird dabei verwendet um einen eigenen PUM Raum V_k^{PU} zu erzeugen. Die Produkte $\varphi_i\psi_i$ sind jedoch nicht interpolatorisch und die Funktionenräume über die Level hinweg nicht geschachtelt, also gilt $V_{k-1}^{PU} \not\subset V_k^{PU}$. Das Aufstellen einer Prolongation P_{k-1}^k und einer Restriktion R_k^{k-1} gestaltet sich deshalb als nicht sehr einfach, einige Lösungen werden jedoch in [3] präsentiert. Mit diesen Transferoperationen ist es uns dann möglich ein iteratives Multilevelverfahren zu erstellen um das Gleichungssystem auf dem jeweils feinsten Level zu lösen.

2.5 Bisheriger sequentieller Code

Das Studienprojekt „CrackBench“ ergab eine PUM-Software „Crass“, welche aus zwei Teilen besteht: einem Front-End für die grafische Eingabe der Simulationsparameter und einem Back-End zur Durchführung der konkreten mathematischen Simulation. Das Back-End dieser Software diente als Grundlage für die in dieser Arbeitet implementierte parallele Version der PUM.

Bei der Entwicklung des Back-Ends wurde eine spätere Parallelisierung mittels MPI bereits in Betracht gezogen und vorbereitet. Dies bedeut, dass keine großen Änderungen an dem bestehenden Softwareentwurf durchgeführt werden mussten, sondern lediglich die entscheidenden Komponenten um die zusätzliche Funktionalität erweitert wurden.

Abbildung 2.1 gibt eine Übersicht über alle Komponenten des ursprünglichen Crass Back-Ends. Im folgenden gehen wir insbesondere auf jene Komponenten ein, welche im Zuge dieser Arbeit von Bedeutung waren. Alle anderen Komponenten funktionieren in der parallelen Version genau wie in der sequentiellen Version weiter und werden hier deshalb nicht näher behandelt.

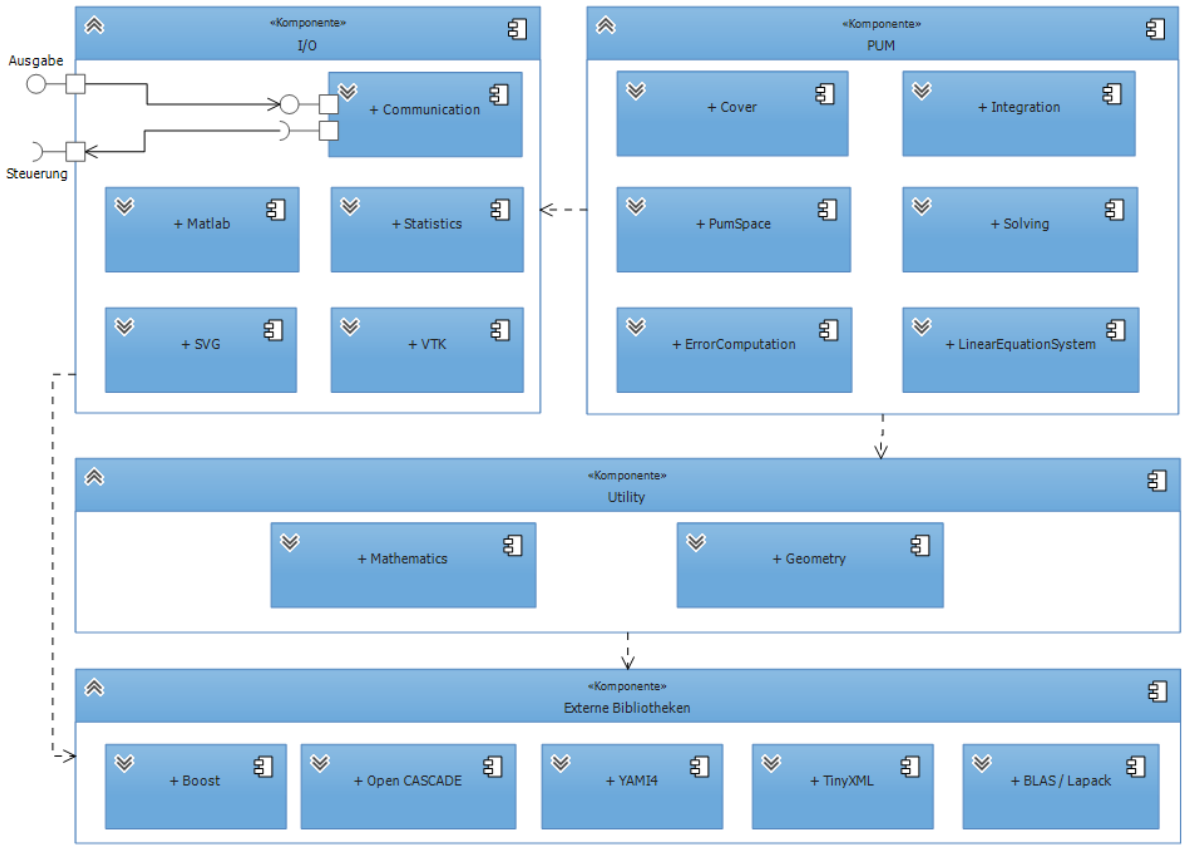

Abbildung 2.1: Übersicht aller Komponenten des bestehenden Crass Back-End Codes.

Cover

Die „Cover“ Komponente ist für die Erstellung der Überdeckungen C_Ω^k auf jedem Level k und dem Errechnen der Nachbarschaften C_i eines Patches ω_i zuständig. Um die Überdeckungen auf den verschiedenen Leveln zu erstellen und zu speichern wird ein Quadtree (in 3D ein Octree) verwendet. Die Tiefe des Baumes steht dabei für den Level unserer Überdeckung (wobei wir für unsere Level bei Null anfangen zu zählen). Durch die Blätter des Baumes lässt sich dann eine Überdeckung erstellen. Abbildung 2.2 zeigt beispielhaft einen solchen Baum und eine von ihm erstellte Überdeckung. Diese Überdeckung entspricht jedoch nicht der tatsächlichen Überdeckung C_Ω^k sondern einer anderen Überdeckung

$$\hat{C}_\Omega^k := \bigcup_i \mathcal{C}_i \subset C_\Omega^k$$

da die lokalen Baumzellen $\mathcal{C}_i := \bigotimes_{l=1}^d [l_i^l, u_i^l] \subset \omega_i$ keinen Überlapp aufweisen. Wir können jedoch sehr einfach aus unseren Baumzellen $\mathcal{C}_i$ Patches ω_i konstruieren, indem wir diese Baumzellen um einem Stretchfaktor α wie folgt vergrößern $\omega_i = \bigotimes_{l=1}^d [c_i^l - \alpha h_i^l, c_i^l + \alpha h_i^l]$ mit $h_i^l = u_i^l - l_i^l$ und $c_i^l = u_i^l - \frac{\hat{h}_i^l}{2}$.

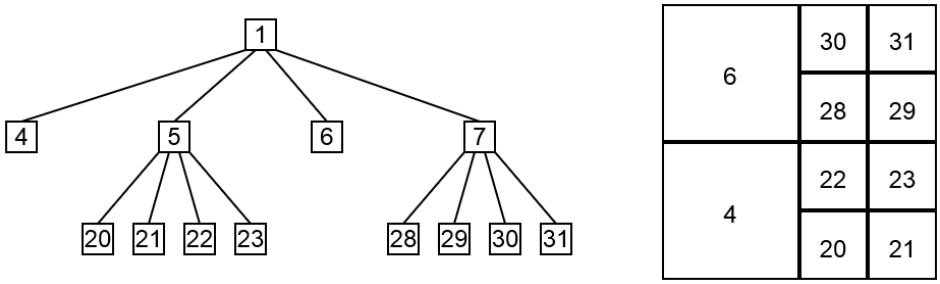

Abbildung 2.2: Links: Quadtree mit individuellen Pfadschlüsseln in den Knoten, welche eine horizontale Ordnung implizieren. Rechts: Überdeckung $\hat{C}^k_\Omega$ welche von den Blättern des Quadtrees erzeugt wird.

Nun stellt sich die Frage welches die beste Datenstruktur ist um diesen Baum zu speichern. Im Crass Back-End wurde hierfür eine Darstellung durch sogenannte Pfadschlüssel gewählt wie sie in **??** vorgestellt wurde. Dabei erhält jeder Knoten im Baum einen eindeutigen Pfadschlüssel k_L, welcher durch die Position innerhalb des Baumes gegeben ist. Die Wurzel des Baumes hat dabei den Schlüssel 1. Die Schlüssel der Kinder $k_{C,i}$ eines Knotens k_L können durch

$$k_{C,i} = 2^d k_L + i \quad \text{mit} \quad i \in \{0, ..., 2^d - 1\}$$

errechnet werden. Das Gebiet der Kindzelle lässt sich durch

$$\mathcal{C}_{C,i} = \bigotimes_{l=1}^{d} [l^l_L + s_l(i), l^l_L + s_l(i) + \tfrac{h^l_L}{2}] \quad \text{mit} \quad s_l(i) = \begin{cases} 0 & \text{für} \lfloor \frac{i}{2^{l-1}} \rfloor \mod 2 > 0 \\ \frac{h^l_L}{2} & \text{sonst} \end{cases}$$

aus der Vaterzelle $\mathcal{C}_L$ an der richtigen Stelle berechnen. Dabei bestimmt die Hilfsfunktion $s_l(i)$ ob das i-te Kind in der Dimension l verschoben wird, also ob es links oder rechts, unten oder oben (,vorne oder hinten, ...) in der Vaterzelle platziert wird. Der Term $\lfloor \frac{i}{2^{l-1}} \rfloor \mod 2 > 0$ kann auch so interpretiert werden, dass er prüft ob die l-te Stelle der Binärdarstellung von i 1 oder 0 ist. Bei 1 findet eine Verschiebung in dieser Dimension l statt, bei 0 nicht.

Diese Pfadschlüssel orientierten Baumdarstellung und die dazugehörigen Überdeckung lässt sich nun mit einem assoziativen Container umsetzen. Dabei werden die Schlüssel der Knoten auf die entsprechenden Baumzellen abgebildet. Wählt man hierfür einen hash-orientierten Container, so lassen sich wahlfreie Zugriffe auf jeden Knoten innerhalb des Baumes im Mittel mit konstantem Zeitaufwand durchführen.

Um nun solch einen Baum zu erzeugen wird in Crass zuerst die Wurzelzelle $\mathcal{R} \supset \tilde{\Omega} \supset \Omega$ als Boundingbox von Ω mit dem Schlüssel 1 erzeugt. Mit Algorithmus 2.1 können wir unseren Baum nun verfeinern und damit diesen um ein weiteres Level erweitern, wenn wir ihn auf alle Blätter unseres Baumes anwenden. Dabei wird vor jedem Aufruf der Verfeinerungsfunktion 2.1 eine sogenannte Verfeinerungsstrategie befragt ob diese Zelle nach der H-Methode, der P-Methode, der HP-Methode oder gar nicht verfeinert werden soll. Eine Verfeinerung nach der H-Methode zieht einen Aufruf von 2.1 nach sich, eine Verfeinerung nach der P-Methode

Algorithmus 2.1 Verfeinern eines Patches nach der H-Methode

procedure RefinePatchSize(**in** k_L)
  for all child $k_{C,i}$ of k_L **do**
    $\mathcal{C}_{C,i} \leftarrow \bigotimes_{l=1}^{d}[l_L^l + s_l(i), l_L^l + s_l(i) + \frac{h_L^l}{2}]$
    $c_{C,i} \leftarrow l_{C,i} + \frac{1}{2}(u_{C,i} - l_{C,i})$
    $h_{C,i} \leftarrow \frac{\alpha}{2}(u_{C,i} - l_{C,i})$
    $\omega_{C,i} \leftarrow \bigotimes_{i=l}^{d}[c_{C,i}^l - h_{C,i}^l, c_{C,i}^l + h_{C,i}^l]$
    $l_{C,i}^{\min} \leftarrow l_L^{\max} + 1$
    $l_{C,i}^{\max} \leftarrow l_L^{\max} + 1$
    make patch $k_{C,i}$ an LEAF node
    add patch $k_{C,i}$ to cover map
  end for
  make patch k_L an INNER node
end procedure

bedeutet, dass der Grad p_i des lokalen Funtionenraumes $V_i^{p_i}$ erhöht wird. Die HP-Methode ist eine Kombination aus den beiden vorhergehenden und bedeutet also sowohl einen Aufruf der Verfeinerungsfunktion, als auch eine Erhöhung des Polynomgrades auf allen Kinderzellen. Findet keine Verfeinerung nach der P-Methode statt, so wird der Polynomgrad der Vaterzelle auf die Kinder übernommen. Findet keine Verfeinerung nach der H-Methode statt, so bedeutet dies, dass die betroffene Blattzelle über mehrere Level hinweg in unseren Überdeckungen mitwirkt. Es wird dann auf der Zelle $l_L^{\max} \leftarrow l_L^{\max} + 1$ gesetzt, wobei $l_L^{\max}$ das höchste Level ist, auf welchem diese Zelle in der Überdeckung mitwirkt ($l_L^{\min}$ ist analog das niedrigste Level).

In 2.3 haben wir gesehen, welche Bedeutung die Nachbarschaften für die Assemblierung der Steifigkeitsmatrix haben. Algorithmus 2.2 zeigt den Algorithmus, welcher in Crass verwendet wird um diese zu berechnen. Es handelt sich dabei um einen einfachen Baumabstieg, welcher genau die Pfade in unserem Baum folgt, welche zu den Nachbarn des gewünschten Patches $k_{\hat{L}}$ führen. Dabei werden nacheinander die Patches k_L besucht. Über das Level l lässt sich steuern für welches Level wir die Nachbarschaften erhalten möchten. Der Algorithmus wird also für alle $k_{\hat{L}}$ aufgerufen welche die Blätter unseres Baumes sind und die Überdeckung erzeugen. Als Initialwert für k_L wird immer die Wurzelzelle mit dem Schlüssel 1 gewählt, so das Algorithmus die Möglichkeit hat alle Knoten zu besuchen und gewährleistet werden kann, das alle Nachbarn gefunden werden.

Integration

Die Integration zum Assemblieren der Matrix und der rechten Seite wird in Crass so gehandhabt, dass für jede Baumzelle $\mathcal{C}_i$ Integrationszellen erzeugt werden. Dabei wird das Gebiet $\mathcal{C}_i$ entsprechend der Überlappe der Nachbarn C_i und der darauf verwendeten φ_i in kleinere Rechtecke $\mathcal{I}_{i,n}^D$, in Crass die sogenannten „IntegrationDomains“ zerlegt. Jedes dieser Rechtecke speichert dabei die Information welche Patches auf diesem leben also eine Liste $P_{i,n} := \{k_j :$

Algorithmus 2.2 Baumabstieg zum Finden von Nachbarn

```
procedure ComputeNeighbors(in k_L̂, in k_L, in l, in out C_L̂)
    if patch k_L is not in cover map then
        return
    end if
    if ω_L ∩ ω_L̂ = ∅ then
        return
    end if
    if ω_L ∩ Ω = ∅ then
        return
    end if
    if l_L^min > l then
        return
    end if
    if l_L^max ≥ l then
        if k_L ≠ k_L̂ then
            C_L̂ ∪ {k_L}
        end if
    else
        for all children k_C of k_L do
            ComputeNeighbors(k_L̂, k_C, l, C_i)
        end for
    end if
end procedure
```

$\omega_j \cap \mathcal{I}^D_{i,n} \neq \emptyset\}$. Anschließend werden diese Integrationsgebiete noch mit Ω geschnitten, wobei die tatsächlichen Integrationszellen $\mathcal{I}^C_{i,m}$ herauskommen. Nun kann über all diese Integrationszellen iteriert werden und auf dessen Gebiet das Integral für alle Patches $k_j \in P_{i,n}$ ausgewertet werden. Dies erlaubt es uns auch, das Integral direkt an die richtige Stelle in unserer Steifigkeitsmatrix oder den Vektor der rechten Seite zu schreiben. Als Ausgangspunkt für die Zerlegung in die Integrationsgebiete haben wir die lokalen Baumzellen $\mathcal{C}_i$ und nicht die tatsächlichen Patches ω_i verwendet, da sich diese nicht Überlappen. Dies hat den Vorteil, dass wir die Zerlegung sehr einfach für all unsere lokalen Blätter durchführen können, ohne dabei doppelte Zellen zu erhalten, wie es der Fall wäre, wenn wir ω_i zerlegt hätten.

Solving

Als Löser wird in Crass das Verfahren der konjugierten Gradienten mit einem symmetrischen Multilevel Vorkonditionierer eingsetzt. Dieser Vorkonditionierer verwendet dabei ein Block-Gauß-Seidel Iterationsverfahren als Glätter. Für genauere Informationen zu den verschiedenen Lösern und ihrem Verhalten siehe [3].

Die Transferoperatoren werden gemäß des „Local-To-Local“ Ansatzes, wie in [3] vorgestellt, erstellt. Zum assemblieren der Prolongation P_{l-1}^{l} wird dabei ein ähnliches Integrationsverfahren wie bei der Assemblierung der Steifigkeitsmatrix verwendet. Ausgangspunkt für das Erstellen der Integrationsgebiete sind hier jedoch nicht die lokalen Zellen $\mathcal{C}_i$, sondern die größeren Patches ω_i. Hierbei benötigen wir jedoch nicht die geometrischen Nachbarschaften C_i unserer Blätter k_i, sondern die Zellen ihrer Väterknoten. Bei den Väterknoten spricht man in diesem Zusammenhang auch von der allgemeineren „hirarchischen Nachbarschaft“. Durch transponieren der Prolongation erhalten wir dann die Restriktion $R_l^{l-1} = (P_{l-1}^{l})^T$.

Mathematics

Die Mathematik-Komponente stellt grundlegende mathematische Operationen und Datentypen zur Verfügung. Insbesondere von Bedeutung ist hier die Klasse, welche zum speichern der dünnbesetzten Blockmatrizen wie der Steifigkeitsmatrix oder den Transferoperatoren verwendet wird. Es handelt sich hierbei um eine Matrix in „Compressed-Row-Storage“. Es werden also für jede Reihe der Matrix die nicht-Null Spalteneinträge in beliebiger Reihenfolge abgelegt. Dies bedeutet insbesondere, dass der Speicher innerhalb unserer Matrix Reihenweise gegliedert ist. Außerdem führt es dazu, dass ein Zugriff auf die Reihen wahlfreie geschehen kann, ein Zugriff auf eine Spalte aber, immer eine Suche in der entsprechenden Reihe mit sich zieht. Zu beachten ist hierbei, dass diese Art der Speicherung ein Matrix-Vektor-Produkt, welches die häufigste Operationen in unserem Löser ist, effizient unterstützt.

3 Parallele PUM

In diesem Kapitel wird behandelt, welche Schritte bei der Parallelisierung des Crass Codes bedacht und umgesetzt wurden. Dabei wird zu Beginn auf Grundlegende Konzepte eingegangen, die während des Vorgehens verfolgt wurden. Anschließend wird gezeigt wie die Datenparallelisierung, mit allen an sie gesetzten Randbedingungen durch die PUM, hergestellt wird. Im letzten Teil wird dann darauf eingegangen, welche Folgen diese Datenparallelisierung auf die weiteren Schritte, wie die Assemblierung der Matrizen und das Lösen des Gleichungssystems, hat.

3.1 Motivation und generelles Vorgehen

Für die Berechnung von großen Problemen mit vielen Freiheitsgraden, sind selbst bei Algorithmen, die linear in der Anzahl der Freiheitsgrade skalieren, moderne Computer nicht groß genug. Dabei stellt sich oft nicht nur die Laufzeit, sondern gerade der Speicher als begrenzender Faktor heraus. Datenparallele Algorithmen können hierbei also Abhilfe schaffen. Ziel ist es hierbei, die Daten zu Beginn eines Rechenschritts auf viele einzelne Computer zu verteilen, um so den benötigten Speicherbedarf pro Computer zu reduzieren. Der Gesamtspeicherbedarf wird hierbei idealerweise auf die einzelnen Computer aufgeteilt. Anschließend soll jeder Computer möglichst die gesamte Arbeit nur auf seinen eigenen lokalen Daten durchführen, sodass sich auch die Rechenzeit auf die einzelnen Computer aufteilt. Um nur die Rechenzeit zu verringern, lassen sich natürlich auch mehrere Datenparallele Prozesse auf einem einzelnen Computer mit gemeinsamen Speicher starten, um z.B. Mehrkernarchitekturen heutiger Prozessoren ausnutzen zu können.

In der Realität tritt dabei natürlich ein gewisser Overhead auf. Die Berechnungen lassen sich oft nicht komplett nur auf den lokalen Daten durchführen, sondern es werden noch einige Daten von anderen Prozessen benötigt. In der PUM handelt es sich bei diesen Daten z.B. um die Nachbarn eines Patches, der an der Grenze von zwei Prozessen angesiedelt ist. Nun ist es nötig auf den einzelnen Prozessen Kopien dieser Daten anzulegen um die lokale Berechenbarkeit aufrecht zu erhalten, was sich negativ auf den Speicherverbrauch auswirkt. Alternativ wäre es möglich, die benötigten Daten auch jedes Mal, wenn sie benötigt werden von einem anderen Prozess anzufordern. Dies würde jedoch Kommunikation erfordern, und sich somit in der Laufzeit niederschlagen. Es ist also von entscheidender Bedeutung für eine gute Parallelisierung abzuwägen, wann welcher der beiden Ansätze besser geeignet ist.

3.2 Datenverteilung

Wir möchten nun also unsere Patches auf unterschiedliche Prozessoren verteilen. Dies setzten wir so um, wie es in [4] vorgestellt wurde. In 2.5 haben wir eine Struktur kennengelernt, in der wir unsere Baumzellen durch Pfadschlüssel k_L identifizieren. Für eine parallele Implementierung zeigt sich als weiterer Vorteil dieser Struktur, dass Zellen über Prozessoren hinweg identifiziert werden können. So kann Prozessor q genau sagen, wo im Baum sich eine Zelle k_L befindet, obwohl er selbst diese Zelle gar nicht besitzt, sondern evtl. ein Prozessor $\tilde{q}$. Zur Verteilung der Patches wählen wir nun Grenzen

$$0 = r_0 \leq r_1 \leq r_2 \leq ... \leq r_p = k_{\max} \tag{3.1}$$

sodass wir mit $[r_q, r_{q+1})$ jedem Prozessor q genau ein Intervall an Patches zuordnen können. Wenn wir diese Intervallgrenzen an alle Prozessoren verteilen, ist es damit jedem Prozessor möglich für einen Patch k_L genau zu bestimmen, welcher Prozess der Besitzer dieses Patches ist.

Betrachten wir nun noch einmal Abbildung 2.2 so bemerken wir aber, das eine Verteilung mit unseren Pfadschlüsseln k_L zu einer horizontalen Verteilung auf die Prozessoren führen würde. Die Prozessoren würde also einzelne „Lagen“ unseres Baumes erhalten. Möchte man nun aber einen Traversierungsalgorithmus wie z.B. Algorithmus 2.2 zum Suchen der Nachbarn einsetzten, so ist diese Wahl der Verteilung unvorteilhaft. In [4] wird weiterhin eine Methode vorgestellt, wie man diese Pfadschlüssel k_L in sogenannte Gebietsschlüssel k_L^D umwandeln kann, welche eine vertikale Ordnung erzeugen und somit einzelne Teilbäume auf die Prozessoren verteilt werden. Zu beachten ist hierbei, dass unsere Schlüssel k_L^D nicht mehr zwangsweise eindeutig sind. Wir können diese also nicht mehr verwenden um Zellen eindeutig in unserem Baum zu identifizieren.

3.3 Lastbalance

Mit den Gebietsschlüsseln k_L^D lässt sich unser Baum nun also so auf die einzelnen Prozessoren verteilen, dass ganze Teilbäume an die Prozessoren zugewiesen werden. Jetzt stellt sich aber die Frage welche Werte für unsere Intervallgrenzen r_q am besten geeignet sind um die Last pro Prozessor ausgeglichen zu halten um das Maximum aus unserer verfügbaren Rechenleistung herauszuholen.

3.3.1 Erstellen einer Ordnung

Zur Berechnung der Grenzen versuchen wir nun also Pakete in unserem Gebiet zu finden, die möglichst kompakt zusammenhängen. Es wird damit versucht ein möglichst gutes Verhältnis zwischen dem Volumen der Teilgebiete und der Oberfläche zu angrenzenden Prozessoren finden, da dies mit dem Verhältnis an lokal berechenbaren Zellen (Volumen) zu Zellen, welche Kommunikation bedürfen (Oberfläche), zusammenhängt. Mit der Konvertierung der Pfadschlüssel

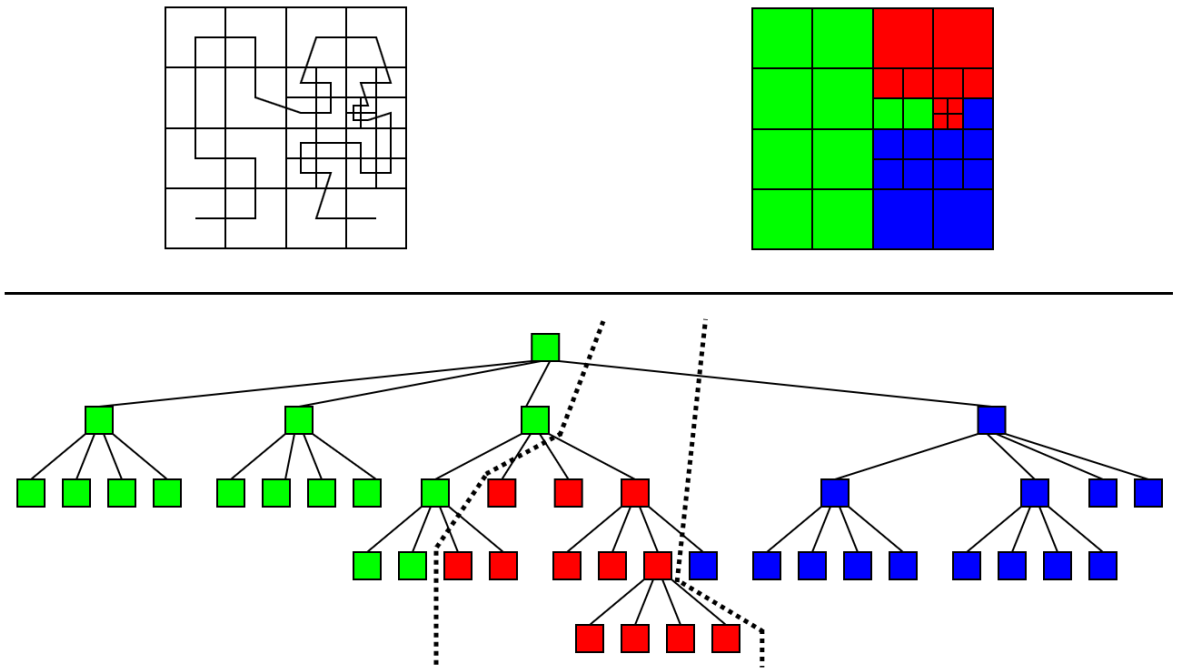

Abbildung 3.1: Oben links: Hilbert-Kurve auf einer adaptiv verfeinerten Überdeckung. Oben rechts: Verteilung des Gebietes auf 3 Prozessoren (rot, grün, blau) nach der Hilbert-Kurve mit starker Lokalität. Unten: Baum entsprechend der Überdeckung mit selbiger Verteilung.

in Gebietsschlüssel, wie in 3.2, angesprochen ergibt sich eine Ordnung unserer Blätter, welche einer raumfüllende Kurve entspricht. Bei der raumfüllenden Kurve handelt es sich dabei um die Lebesgue-Kurve, welche unseren Ansprüchen jedoch noch nicht ausreichend genügt. In [1] werden verschiedene raumfüllende Kurven mit Ihren besonderen Eigenschaften diskutiert. Aus [4] geht hervor, dass die Hilbert-Kurve besonders für unsere Zwecke geeignet ist.

Mit diesen neuen Hilbert-Gebietsschlüsseln k_L^D können wir nun also eine Ordnung unserer Patches erstellen, doch diese können immer noch nicht genutzt werden um Zellen in unserem Baum eindeutig zu identifizieren. Algorithmus 3.1 zeigt hier eine Möglichkeit unsere originalen Pfadschlüssel k_L mit den Gebietsschlüsseln k_L^D zu sortieren. Wir stellen hierfür einen sortierten assoziativen Container *orderedMap* auf, welcher von k_L^D auf k_L Abbildet und nach k_L^D sortiert ist. Bei dem Algorithmus handelt es sich wieder um einen Baumabstiegsalgorithmus, welcher alle eigenen Zellen im Baum besucht. Zellen von anderen Prozessoren und jene, welche nicht auf dem gesuchten Level sind, werden ausgeschlossen. Die Funktion GetOwner findet dabei den Besitzerprozess $\tilde{q}$, so dass $k_L^D \in [r_{\tilde{q}}, r_{\tilde{q}+1})$. Wird die Liste der r_q sortiert gespeichert, so hat dies eine Komplexität von $O(\log p)$. Der gesamte Algorithmus hat somit eine Komplexität von $O(N_q \log p)$ wobei N_q die Anzahl der Zellen auf dem eigenen Prozessor darstellt.

Algorithmus 3.1 Ordnen der Patches auf einem Level

```
procedure ComputeOrder(in out orderedMap, in l, in k_L)
    q_L ← GetOwner(k_L)
    if q_L ≠ q̂ then
        return
    end if
    if l_L^min > l then
        return
    end if
    if l_L^max ≥ l then                          // Patch ist auf gesuchtem Level
        if Patch k_L is not OUTSIDE then
            k_L^D ← ComputeHilbertCurveKey(k_L)
            Add (k_L^D, k_L) to orderedMap
        end if
    else                                         // Patch hat Kinder
        for all child k_C of k_L do
            ComputeOrder(orderedMap,l,k_C)
        end for
    end if
end procedure
```

3.3.2 Lastabschätzung und Neuverteilung

Im folgenden betrachten wir nun, wie wir eine gute Wahl für die Intervallgrenzen r_q finden. Eine gute Wahl bedeutet hier, dass die Blöcke an Last, welche den einzelnen Prozessoren zugewiesen werden sollen, möglichst gleich groß sind. Wie wir noch sehen werden, ist die Definition von Last, die wir hierfür verwenden, von großer Bedeutung. In [4] wird ein Lastbalancierungsverfahren vorgestellt, welches hier in Algorithmus 3.2 dargestellt wird. Wir sehen, dass wir für diesen Algorithmus die Ordnung der Blätter des Baumes benötigen, welche mit Algorithmus 3.1 berechnet werden kann. Die Funktion EstimateLoad ist hier von entscheidender Bedeutung. Durch sie kann gesteuert werden, welche Art von Last balanciert werden soll. Eine einfachste Variante wäre eine Funktion zu verwenden, welche immer 1 zurück gibt, also die Anzahl der Blattzellen balanciert. Diese Funktion würde jedoch den Grad der lokalen Funktionenräume $V_i^{p_i}$ nicht mit in Betracht ziehen. Somit wäre ein Baum, in welchem nach der P-Methode verfeinert wurde, nicht hinreichend balanciert worden. Eine Funktion, welche auf die Freiheitsgrade jeder Zelle abbildet wäre hier besser geeignet. Ein weitere Möglichkeit die Lastabschätzung zu verbessern, wäre die Nachbarn C_i einer Zelle mit in die Berechnungen einfließen zu lassen, da wir gesehen haben, dass diese einen Einfluss auf das Besetzungsmuster unserer Steifigkeitsmatrix haben. Wir sehen also, dass es wichtig ist, diese Funktion EstimateLoad im Code leicht anpassbar zu gestalten, sodass je nach Problemstellung eine geeignete Lastabschätzung entwickelt und verwendet werden kann.

Nachdem wir nun also festgelegt haben, welche Zellen von welchem Prozessor zu berechnen sind, müssen wir diese neue Verteilung auch anwenden. Dazu prüft jeder Prozessor ob die Zellen k_L,

Algorithmus 3.2 Berechnen der balancierten Verteilungsgrenzen

procedure DISTRIBUTEBALANCED(**in** *orderedMap*)
  $\omega^{\hat{q}} \leftarrow 0$
  for all *entry* in *orderedMap* **do** // Berechne lokale Last
    $\omega^{\hat{q}} \leftarrow \omega^{\hat{q}}+$ ESTIMATELOAD(*entry.pathKey*)
  end for
  ALLGATHER($\omega^{\hat{q}}$,ω^{q})
  $\omega_g \leftarrow \sum_{q=0}^{p-1} \omega^q$ // Berechne globale Last
  $\omega_g^{\hat{q}} \leftarrow \sum_{q=0}^{\hat{q}-1} \omega^q$ // Berechne Summe der momentanen Lasten vorhergehender Prozesse
  for all q in $\{0, ..., p\}$ **do** // Berechne gewünschte Lastverteilungsgrenzen
    $\omega_b^q \leftarrow \frac{q\omega_g}{p}$
  end for
  $\omega_r \leftarrow 0$
  for all *entry* in *orderedMap* **do** // Finde lokale Intervallgrenzen
    Find smallest $\tilde{q}$ in $\{0, ..., p\}$ such that $\omega_b^{\tilde{q}} \geq \omega_r$
    $\tilde{r}^{\tilde{q}} \leftarrow entry.domainKey$
    $\omega_r \leftarrow \omega_r+$ ESTIMATELOAD(*entry.pathKey*)
  end for
  ALLREDUCE($\tilde{r}^q$, r^q, *mpi_maximum*)
  $r^0 \leftarrow 0$ // Stelle sicher, dass alle Patches enthalten sind
  $r^p \leftarrow MAX\_HASH\_KEY$
end procedure

welche er im Moment noch lokal gespeichert hat nach der neuen Verteilung r_q auf einen anderen Prozessor gehört. Wenn dies der Fall ist, wird die Zelle für einen Austausch vorgemerkt, welcher dann gemeinsam mit allen anderen Prozessen in einem Kommunikationsschritt durchgeführt wird. Das erstellen der Liste, der zu versendenden Zellen hat eine Komplexität von $O(N_q \log p)$, mit N_q als der Anzahl der Zellen auf dem Prozess vor dem Austausch. Die Kommunikation erfordert p Schritte in denen jeweils jeder Prozess mit einem anderen kommuniziert.

Zu beachten ist, dass wir diese Verteilung jeder Zeit auf einem adaptiv verfeinerten Baum anwenden können und wir die Wahl unsere Grenzen r_q immer auf Zellen unseres tiefsten Levels, also den Blättern des Baumes durchführen. Dies führt dazu, dass, wie in Abbildung 3.1 bereits zu sehen war, die Wurzeln der lokalen Teilbäume nach unten rutschen können. Wir haben dabei anders als in [4] keinen gemeinsamen, globalen Teilbaum dessen Blätter die Wurzeln unserer lokalen Bäume sind. Diesen herzustellen, würde entweder bedeuten, dass wir den oberen Teil unseres Baumes auf alle Prozessoren kopieren müssten (was eine kontrollierte Speicherverteilung immens erschweren würde) oder, dass wir unsere Lastbalance nur noch mit Einschränkungen durchführen könnten. Ohne diesen gemeinsamen Baum stoßen wir jedoch auf das Problem, dass Algorithmen wie 3.1, welche den Baum traversieren und nur über die eigenen Zellen laufen, nicht mehr immer auf der Wurzelzelle mit dem Schlüssel 1 gestartet werden können. Wir müssen also die lokalen Wurzelzellen $R_{\hat{q}}$ (Abbildung 3.2) ermitteln. Diese

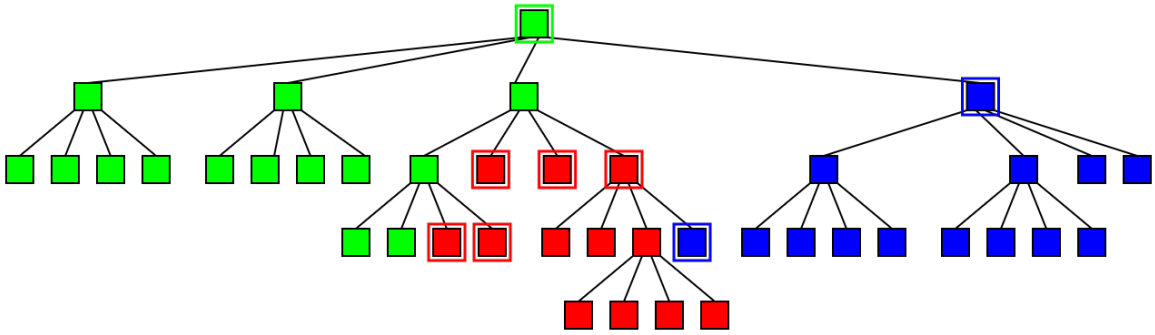

Abbildung 3.2: Die lokalen Baumwurzeln $R_{\hat{q}}$ der einzelnen Prozessoren (rot, grün, blau) sind hervorgehoben.

Algorithmus 3.3 Finden der lokalen Teilbaum-Wurzeln

```
procedure ComputeLocalTreeRoots(in out R_q̂)
    for all k_L in cover map do
        q_L ← GetOwner(k_L)
        k_P ← Key of parent of k_L
        q_P ← GetOwner(k_P)
        if q_L = q̂ and (k_L = 1 or q_P ≠ q̂) then
            R_q̂ ← R_q̂ ∪ {k_L}
        end if
    end for
end procedure
```

sind leicht daran zu erkennen, dass ihr Vater im Besitz eines anderen Prozesses ist. Algorithmus 3.3 zeigt dabei wie die lokalen Baumwurzeln in $O(N_q \log p)$ zu erreichen ist.

3.4 Verfeinern

In den vorherigen Abschnitten sind wir bisher immer von einem bereits existenten Baum mit einer gewissen Tiefe ausgegangen. Nun wollen wir betrachten wie dieser Baum erstellt wird. In Kapitel 2.5 hatten wir bereits gesehen wie in der sequentiellen Version eine Verfeinerung durchgeführt wird. Diese sequentielle Version, für einen Prozess wollen wir uns nun zu Nutzen machen.

Wir beginnen damit, dass alle Prozesse die Wurzelzelle mit dem Schlüssel 1 erstellen. Anschließend rufen sie lokal den Algorithmus 2.1 zum Verfeinern einer Zelle auf diesen auf. Dies wird so lange auf den neu entstandenen Blätter des Baumes fortgeführt, bis die Anzahl der Blätter, die Anzahl der Prozessoren übersteigt. Auf diesem Level wird nun eine Lastverteilung errechnet und angewendet. Die für die Lastverteilung benötigte Order zu erhalten ist hier kein Problem, da

der Algorithmus 3.1 hier noch auf die Wurzelzelle 1 angewendet werden kann, da der komplette Baum auf allen Prozessen zur Verfügung steht. Erst nach diesem Verteilungsschritt müssen wir die lokalen Baumwurzeln berechnen um weiterhin Traversierungsalgorithmen anwenden zu können. Für eine sehr große Anzahl von Prozessoren wäre es möglich eine Optimierung an diesem ersten gemeinsamen Verfeinern durchzuführen. Dazu könnte man die Prozesse nach jedem Verfeinerungsschritt in so viele Gruppen aufteilen, wie Blätter am Baum existieren. Jede dieser Gruppen würde dann das Verfeinern eines Blattes durchführen anschließend würde wieder in Gruppen verteilt werden. Abwandlungen, die nicht in jedem Schritt neue Gruppen erstellen, welche dann nur eine Zelle verfeinern, sind hier natürlich analog möglich um den Overhead unter Kontrolle zu halten.

Nach dem Aufstellen des Initialen Baumes würde jeder weitere Verfeinerungsschritt, welchen wir durchführen, um ein neues Level für unser Multilevelverfahren zu erhalten, auf jedem Prozess, lokal auf seinen eigenen Blattzellen durchgeführt werden, bevor anschließend der gesamte Baum neu balanciert wird. Wie viele Zellen bei dieser Neuverteilung tatsächlich den Besitzer wechseln, und damit einen Kommunikationsaufwand herbeiführen, hängt stark von der gewählten Verfeinerungsstrategie ab.

3.4.1 Rippleeffekt

Bei dem Rippleeffekt handelt es sich um ein Verhalten, das auftreten kann, wenn man die maximale Leveldifferenz zwischen zwei Benachbarten Zellen begrenzen möchte. Dies bedeutet, dass das Verfeinern einer Zelle, die Verfeinerung seiner Nachbarn nach sich ziehen kann. Wenn dies wiederum wieder zum Verfeinern der Nachbarn des Nachbarn usw. führt spricht man vom Rippleeffekt. In [5] wird diskutiert, welche Kosten ein solches Aufrechterhalten der maximalen Leveldifferenz haben kann. In der PUM ist das Aufrechterhalten einer maximalen Leveldifferenz in sofern von Bedeutung, dass damit verhindert werden kann, dass ein Patch auf einem sehr groben Level, Nachbarpatches auf sehr feinen Leveln, komplett einschließt.

Für die Parallelisierung ist dieser Effekt besonders interessant, wenn das Verfeinern einer lokalen Zelle, das Verfeinern einer Zelle auf einem anderen Prozess zur Folge hat. Der Rippleeffekt kann sich also über Prozessorgrenzen hinweg fortsetzten.

Um das Aufrechterhalten der maximalen Leveldifferenz in unserem Code umzusetzen, müssen wir den Algorithmus 2.1 zum Verfeinern von Patches nach der H-Methode erweitern. Dazu fügen wir an das Ende des Algorithmus einen Schritt hinzu, der für alle Nachbarzellen k_C aus C_i^l prüft ob die Differenz der Level dieser Zellen größer als die gewünschte maximale Leveldifferenz ist. Die Level der Zellen lassen sich dabei in $O(J)$ aus ihren Schlüsseln berechnen. Ist die Differenz tatsächlich zu groß, so wird zuerst geprüft wer der Besitzer dieser Nachbarzelle ist. Handelt es sich um den eigenen Prozess, so wird der modifizierte Algorithmus 2.1 rekursiv auf diesen Nachbarn aufgerufen. Ist der Besitzer des Nachbarn jedoch ein anderer Prozess, so wird der Schlüssel k_C einer Liste K_q für den Besitzerprozess q hinzugefügt. Nach Abschluss der lokalen Verfeinerungen hat jeder Prozess also $p-1$ Listen mit den Zellen, welche aufgrund seiner lokalen Verfeinerungen auf den anderen Prozessen zu verfeinern sind.

Algorithmus 3.4 Patches Rekursiv auf anderen Prozessoren verfeinern

procedure PERFORMREMOTEREFINEMENTS
  $anyPatchesToRefine \leftarrow$ ALLREDUCE($\bigwedge_{q\in\{0,...,p-1\}\setminus\{\hat{q}\}} |K_q| > 0$, *mpi_logical_or*)
  if not $anyPatchesToRefine$ **then**
    return
  end if
  EXCHANGE($K_q : q \in \{0, ..., p-1\} \setminus \{\hat{q}\}$, $K_{\hat{q}}$)
  $K_q \leftarrow \emptyset \forall q \in \{0, ..., p-1\} \setminus \{\hat{q}\}$
  for all $k_L \in K_{\hat{q}}$ **do**
    REFINEPATCH(k_L,„H“)
  end for
  PERFORMREMOTEREFINEMENTS()
end procedure

Nun können wir mit Algorithmus 3.4 die Verfeinerungen auf den anderen Prozessen durchführen. Algorithmus 3.4 ist dabei rekursiv, da nach der Verfeinerung jener Zellen, welche ein Prozess $\hat{q}$ von den anderen Prozessen q aufgetragen bekommen hat, erneut der Rippleeffekt einsetzten kann. Deshalb wird zu Beginn des Algorithmus die Funktion ALLREDUCE darüber durchgeführt ob ein Prozess überhaupt eine Zelle für einen anderen Prozess hat. Ohne diese Operation würde der Algorithmus nicht terminieren. Nach dem Austauschen der zu verfeinernden Zellen mit der Funktion EXCHANGE, welche die empfangen Daten in $K_{\hat{q}}$ speichert, werden die übrigen Listen K_q gelöscht. Anschließend werden die Verfeinerungen durchgeführt, was die Listen K_q evtl. erneut füllt und deshalb den rekursiven Aufruf von PERFORMREMOTEREFINEMENTS nach sich zieht.

3.5 Nachbarsuche

Wir haben nun einen Baum erstellt und diesen balanciert auf unsere Prozesse verteilt. Zum assemblieren der Matrix benötigen wir aber, wie schon in 2.5 gesehen, die Nachbarschaften C_i eines Patches ω_i. In parallel tritt hierbei der Fall auf, dass die Nachbarn eines Patches nicht auf demselben Prozess liegen. Ein Prozess $\hat{q}$ kann für seine eigenen Patches nicht bestimmen, welche Patches eines anderen Prozesses in der Nachbarschaft des eigenen Patches liegen. Dazu müsste er bereits wissen, wie in den Nachbarregionen verfeinert wurde. Dies kann jedoch nicht gewährleistet werden ohne, den gesamten Baum der Nachbarprozessoren zu kopieren, was natürlich ausgeschlossen ist. Der hier verfolgte Ansatz ist also, dass jeder Prozess prüft, welche seiner eigenen Patches potentiell von einem anderen Prozess in den Nachbarschaften benötigt werden.

Um dies zu bewerkstelligen berechnet jeder Prozess die „minimale Überdeckung“ seiner eigenen Blattzellen. Die ideale minimale Überdeckung wären die tatsächlichen Blattzellen selbst. Aus Effizienzgründen wollen wir diese aber nun durch eine bessere Darstellung approximieren. Ziel ist es eine Überdeckung zu finden, die aus möglichst wenigen Rechtecken zusammengesetzt werden kann, aber dennoch nicht deutlich größer ist als die ideale minimale Überdeckung.

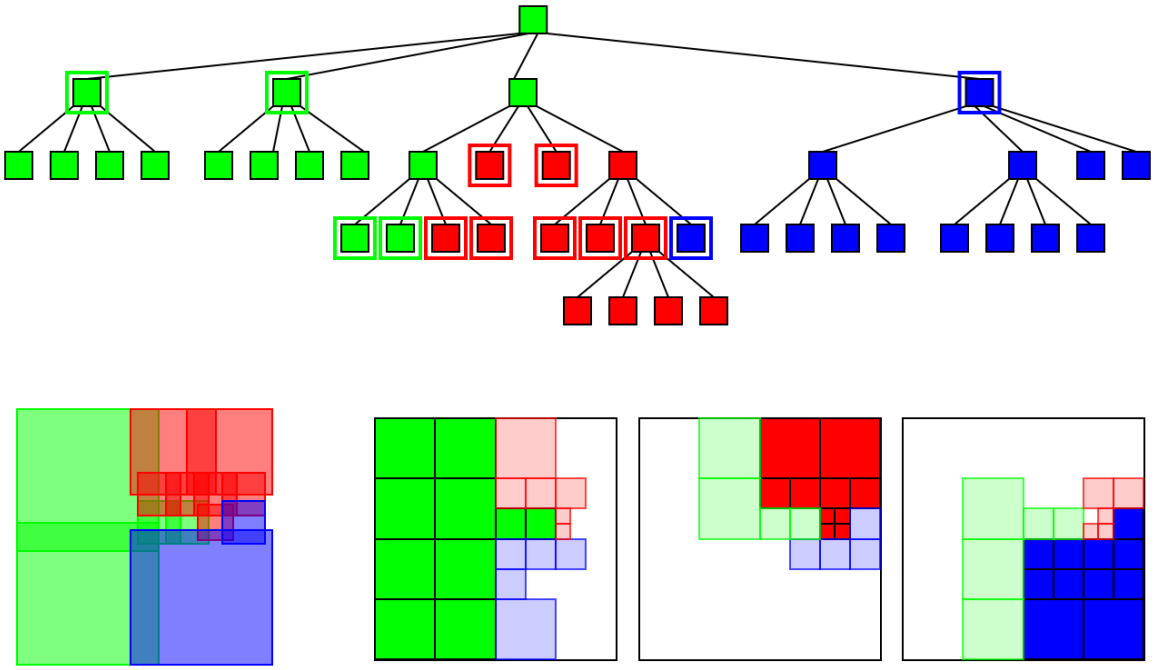

Abbildung 3.3: *Oben:* Die Zellen, welche eine lokale Überdeckung der einzelnen Prozesse (rot, grün, blau) bilden, sind hervorgehoben. *Unten links:* Die Minimalen Überdeckungen der einzelnen Prozesse. *Unten rechts (3 Bilder):* Die lokalen Zellen der einzelnen Prozessoren von links nach rechts. Jeweils mit den Nachbarn der anderen Prozesse, die benötigt werden.

Als Lösung für dieses Problem wählen wir jene Zellen aus, die Wurzeln vollständig lokaler Teilbäume sind. Es müssen also alle Kinder (und deren Kinder usw.) auf demselben Prozessor leben, wie die besagte Zelle. Es muss aber mindestens eine Geschwisterzellen auf einem anderen Prozessor leben. Abbildung 3.3 zeigt die gesuchten Zellen in unserem gewohnten Beispielbaum. Für die Überdeckung wollen wir nun aber nicht die Patches ω_i nehmen, welche zu diesen Wurzelzellen $\mathcal{C}_i$ gehören, da ω_i evtl. viel größer ist als die Summe seiner Kinder. Wir wählen also ein kleineres Rechteck $\tilde{\omega}_i$ mit der Eigenschaft

$$\omega_i \supseteq \tilde{\omega}_i \supseteq \bigcup_s \omega_s$$

wobei ω_s die Patches der Blätter des vollständig lokalen Teilbaums zu $\mathcal{C}_i$ sind.

Algorithmus 3.5 zeigt, wie diese Berechnung implementiert wurde. Es handelt sich dabei um eine Art Tiefensuche. Der Algorithmus wird initial mit den lokalen Teilbaum-Wurzel aus 3.3.2 als k_L aufgerufen um die minimale Überdeckung für das Level l zu finden. Zellen, welche außerhalb des Gebiets liegen oder erst auf einem höheren Level gültig sind, werden übersprungen. Dabei wird der Rückgabeparameter *skipped* auf true gesetzt, sodass diese Information nach oben propagiert wird. Ist die Zelle k_L auf dem gesuchten Level gültig (also

bei $l = J$ eine der Blattzellen), so wird das minimal überdeckte Gebiet dieser Zelle $\tilde{\omega}_i$ auf die exakte Patchgröße ω_i gesetzt. Eine solche Blattzelle wird so behandelt wie eine Zelle dessen gesamten Kinder auf demselben Prozess leben, weshalb der Rückgabeparameter *local* als true zurück gegeben wird. Ist die Zelle nicht bis zu dem gesuchten Level gültig, so wird der Algorithmus rekursiv auf die Kinder, welche auf dem eigen Prozess leben aufgerufen. Wurde eine Kindzelle k_S nicht übersprungen und sind haben sich all ihre Kinder als ebenfalls auf demselben Prozess herausgestellt, so wird das von ihr überdeckte Gebiet zuerst in einer temporären Liste C_{pot} gespeichert, um sie als potentiellen Kandidaten für die minimale Überdeckung $C^{\hat{q}}_{\text{min}}$ vorzumerken. Die eigene minimale Überdeckung $\tilde{\omega}_L$ wird anschließend so erweitert, dass sie die minimale Überdeckung der Kindzelle $\tilde{\omega}_S$ umschließt. Wenn sich nicht alle Kinder k_L als auf demselben Prozess herausgestellt haben, werden die potentiellen Zellen C_{pot} nach $C^{\hat{q}}_{\text{min}}$ übertragen. Der Algorithmus besucht jeden lokalen Knoten im Baum genau ein mal. Dabei wird für jedes Kind einmal der Besitzer bestimmt. Dies führt also zu einer Komplexität von $O(N_q \log p)$

Mit diesen minimalen Überdeckungen ist es uns jetzt möglich effizient die potentiellen Nachbarn für einen anderen Prozess zu bestimmen. Dazu werden zuerst alle lokal berechneten minimalen Überdeckungen zwischen den Prozessen ausgetauscht. Danach kann jeder Prozess im ersten Schritt seine eigene minimale Überdeckung gegen die der anderen Prozesse schneiden um jene Prozesse auszusortieren, die überhaupt nicht mit diesem Prozess benachbart sind. Anschließend führt er wieder einen rekursiven Suchalgorithmus, gestartet auf den lokalen Teilbaum-Wurzeln, aus. Dieser steigt nur jene Zellen hinab, die einen Schnitt mit einer der Boxen aus den minimalen Überdeckungen der benachbarten Prozesse haben. Die Zellen, welche auf dem gewünschte Level gültig sind, werden in die Liste der potentiellen Nachbarn aufgenommen. Um den Algorithmus zu optimieren, können bei den rekursiven Aufrufen nur jene Teile der minimalen Überdeckungen mitgegeben werden, welche die momentan besuchte Zelle geschnitten haben. Der Algorithmus besucht also nur die Zellen am Rand des lokalen Gebiets. Die Kosten der Schnitte können jedoch stark von der Anzahl der Rechtecke, die für die minimalen Überdeckungen benötigt wurden abhängen. In Abbildung 3.3 haben wir gesehen wie stark diese auseinander gehen können. Während der blaue Prozess beispielsweise nur zwei Rechtecke benötigt sind es beim roten Prozess sieben. Weitere Untersuchungen könnten hier getätigt werden um zu untersuchen ob ein leicht freieres balancieren (indem man z.b. die eine blaue Zelle auf den roten Prozess hinüberwechselt) einen Vorteil für die Gesamtrechenzeit haben kann.

Hat jeder Prozess nun die potentiellen Nachbarn für die anderen Prozessoren berechnet können diese ausgetauscht werden. Wir können hier jedoch noch nicht unseren Nachbarschafts-Suchalgorithmus 2.2 anwenden. Würden wir diesen auf unsere lokalen Teilbaum-Wurzeln anwenden, so würde er möglicherweise nicht alle Nachbarn finden, die von anderen Prozessen kommen. Diese müssen nicht zwangsweise alle in einem der Bäume unter den lokalen Teilbaum-Wurzeln liegen, sondern können auch in einem Teilbaum daneben liegen. Um alle Nachbarn finden zu können, müssen wir also an einem höheren Knoten ansetzten. Dazu verwenden wir Algorithmus 3.6. Dieser Algorithmus erstellt Füllzellen von einem Knoten k_L bis hoch zur Wurzelzelle 1. Eigene Zellen werden dabei übersprungen. Bei bereits existenten Zellen wird der Patch ω_P so vergrößert, dass die Patch ω_L, für welche der Pfad bis zur Wurzel erstellt wird, komplett darin enthalten ist. Dies wird benötigt, da der Algorithmus 2.2 für seinen Abstieg prüft ob die Zellen auf dem Abstiegspfad die Zelle, für die die Nachbarn gesucht werden,

Algorithmus 3.5 Berechnen der minimalen Überdeckung

procedure ComputeMinimalCover(**in out** $C^{\hat{q}}_{\min}$, **in** l, **in** k_L, **out** *skipped*, **out** *local*, **out** $\tilde{\omega}_L := \bigotimes_{i=1}^{d}[\tilde{l}^i_L, \tilde{u}^i_L]$)
  if $l^{min}_L > l$ or Patch k_L is OUTSIDE **then**
    skipped ← true
    return
  end if
  skipped ← false
  local ← true
  if $l^{max}_L \geq l$ **then**
    $\tilde{\omega}_L \leftarrow \omega_L$
  else
    $\tilde{\omega}_L \leftarrow \mathcal{C}_L$
    $C_{\text{pot}} \leftarrow \emptyset$
    for all children k_S of k_L **do**
      $q_S \leftarrow$ GetOwner(k_S)
      if $q_S = \hat{q}$ **then**
        ComputeMinimalCover($C^{\hat{q}}_{min}$, l, k_S, *skipped*, *childLocal*, $\tilde{\omega}_S := \bigotimes_{i=1}^{d}[\tilde{l}^i_S, \tilde{u}^i_S]$)
        if *skipped* **then**
          next
        end if
        if *childLocal* **then**
          $C_{\text{pot}} \leftarrow C_{\text{pot}} \cup \{\tilde{\omega}_S\}$
        else
          local ← false
        end if
        $\tilde{\omega}_L \leftarrow \bigotimes_{i=1}^{d}[min(\tilde{l}^i_L, \tilde{l}^i_S), max(\tilde{u}^i_L, \tilde{u}^i_S)]$
      else
        local ← false
      end if
    end for
    if not *local* **then**
      $C^{\hat{q}}_{\min} \leftarrow C^{\hat{q}}_{\min} \cup C_{pot}$
    end if
  end if
end procedure

Algorithmus 3.6 Erstellen aller Patches auf dem Pfad eines Knoten zum Wurzelknoten

procedure CreateParentPath(**in** k_L)
  $k_P \leftarrow$ Key of parent of k_L
  while $k_P > 0$ **do**
    $q_P \leftarrow$ GetOwner(k_P)
    if $q_P = \hat{q}$ **then**
      next
    end if
    if There is a patch for k_P in the cover map **then**
      $\omega_P \leftarrow \bigotimes_{l=1}^{d}[\min(x_L^l - h_L^l, x_P^l - h_P^l), \max(x_L^l + h_L^l, x_P^l + h_P^l)])$
    else
      $\omega_P \leftarrow \omega_L$
    end if
    $k_P \leftarrow$ Key of parent of k_P
  end while
end procedure

schneidet. Wir wenden 3.6 auf alle empfangenen potentiellen Nachbarn und auch unsere lokalen Teilbaum-Wurzeln an. Nun können wir mit einem Aufruf von 2.2 auf den Wurzelknoten 1 alle Nachbarn für jeden unserer Patches finden.

Mit den geometrischen Nachbarn haben wir nun alle Daten um den eigenen Teil der Steifigkeitsmatrix lokal zu assemblieren. Für den Transferoperator P_{l-1}^l benötigen wir jedoch noch zusätzlich die hierarchischen Nachbarn, also die Väter unserer lokalen Blätter. Diese können nun in der parallelen Implementierung erneut im Besitz eines anderen Prozesses sein, wir müssen also auch diese kommunizieren. Hierbei bestimmt jedoch jeder Prozess selbst, welche Patches genau er von den anderen Prozessen benötigt. Dazu prüft der Prozess für alle seinen lokalen Blätter den Besitzer des Vaterknotens. Ist dieser ein anderer als der Prozess selbst, so wird der Schlüssel dieses Knotens in eine Liste für diesen Prozess angefügt. Das erstellen der Listen hat eine Komplexität $O(\hat{N}_q \log p)$ mit $\hat{N}_q$ als der Anzahl der lokalen Blätter. Die erstellten Listen werden dann mit allen Prozessen ausgetauscht. Wird dieser Schritte zwischen dem bestimmen der potentiellen geometrischen Nachbarn und dem versenden dieser eingefügt, so lassen sich die hierarchischen Nachbarn einfach zusammen mit den geometrischen Nachbarn in einem Schritt austauschen.

3.6 Assemblierung

Wie bereits erwähnt kann die Steifigkeitsmatrix und der Vektor der rechten Seite nun komplett lokal assembliert werden. Dabei berechnet jeder Prozess aber auch nur jenen Teil der Matrix (des Vektors), für welchen er die Daten besitzt. Unsere Intervalle der sortierten Blätter des Baumes lassen sich dabei ideal auf die Reihen innerhalb der Steifigkeitsmatrix abbilden. Der Startindex, also der Index der ersten Reihe auf diesem Prozess, lässt sich mit $b_{\hat{q}} =$

$\left|\{k_L^D : k_L^D \in [0, r_{\hat{q}})\}\right|$ berechnen. Dazu berechnet jeder Prozess zuerst die Menge der lokalen Zellen $\tilde{N}_{\hat{q}} = \left|\{k_L^D : k_L^D \in [r_{\hat{q}}, r_{\hat{q}+1})\}\right|$ und sendet diese mittels AllGather an alle Anderen Prozesse. Anschließend kann die Summe $b_{\hat{q}} = \sum_{q=0}^{\hat{q}-1} \tilde{N}_q$ für den eigenen, aber auch für alle anderen Prozesse berechnet werden. Den Endindex wählen wir einfach als den Startindex des nächsten Prozesses $e_{\hat{q}} = b_{\hat{q}+1}$. Der lokale Index i_l^L eines Patches mit dem Schlüssel k_L^D lässt sich Mittels $i_l^L = \left|\{k_S^D : k_S^D \in [r_{\hat{q}}, k_L^D)\}\right|$ berechnen und durch $i_g^L = i_l^L + b_{\hat{q}}$ in einen globalen Index umrechnen, wobei für alle globalen Indices $b_{\hat{q}} \leq i_g^L \leq e_{\hat{q}}$ gilt. In unserer Matrix werden nun nur die Reihen $b_{\hat{q}}$ bis $e_{\hat{q}}$ gespeichert. Insgesamt erhalten wir eine Verteilung der Reihen

$$0 = b_0 \leq b_1 \leq b_2 \leq ... \leq b_p$$

welche ähnlich der Verteilung, für die Patches ist (3.1). Unsere Matrix im Compressed-Row-Storage Format erweist sich hier als sehr vorteilhaft, denn diese lässt sich aufgrund ihrer reihenorientierten Speicherauslegung sehr einfach reihenweise auf die Prozesse partitionieren. Die Partitionsgrenzen b_q werden dabei mit jeder Matrix gespeichert, sodass wir die Verteilung der Matrix, von der unseres Baumes, nach dem einmaligen Erstellen komplett abgetrennt haben. Für den Vektor der rechten Seite gilt all dies analog.

Eine weitere Änderungen zur Assemblierung der Matrix müssen wir bei der Integration vornehmen. In der sequentiellen Version aus 2.5 haben wir als Ausgangspunkt für die Zerlegung in unsere Integrationsgebiete $\mathcal{I}_{i,n}^D$ die lokalen Baumzellen $\mathcal{C}_i$ verwendet um so das Erstellen von doppelten Gebieten zu vermeiden. Dies war kein Problem, da auf jeden Fall auch die Nachbarzellen $\mathcal{C}_j$ aus C_i zerlegt wurden und dabei den Teil für $\mathcal{C}_i$ in ihre Integrationsgebiete mit aufgenommen hatten, um diesen anschließend an der richtigen Stelle in der Matrix zu integrieren. In parallel ist dies nicht überall möglich, da die Nachbarzellen $\mathcal{C}_j$ evtl. auf einem anderen Prozess $\tilde{q}$ angesiedelt sind. Dieser kann den Teil für k_i jedoch nicht in die Matrix assemblieren, da dies in einer Reihe geschehen müsste, welche auf nicht vom Prozess $\tilde{q}$ verwaltet wird. Als erstes ändern wir unsere Integrationsroutine also so ab, dass die Integrale der Zellen $\mathcal{I}_{i,n}^C$ nur für jene Patches ausgewertet werden, welche auf dem eigenen Prozess liegen, um zu garantieren, dass nur in den lokalen Teil der Matrix geschrieben wird. Weiterhin müssen wir nun für alle Zellen k_i welche Patches $k_j \in P_{i,n}$ besitzen, für dessen Besitzer $q \neq \hat{q}$ gilt, auch jene Gebiete $\mathcal{I}_{i,n}^D$ genieren, die in $\mathcal{C}_j$ liegen. Dazu verwenden wir bei diesen Zellen nicht $\mathcal{C}_i$ als Ausgangspunkt für die Zerlegung in die $\mathcal{I}_{i,n}^D$, sondern das größere Gebiet ω_i. Von den Zellen $\mathcal{I}_{i,n}^D$ wählen wir nun aber nur jene aus, welche die folgenden Kriterien erfüllen:

1. $\mathcal{I}_{i,n}^D \cap \mathcal{C}_i \neq \emptyset$ **oder**
   a) $\exists k_j \in P_{i,n} : q \neq \hat{q}$, mit q als Besitzer von k_j **und**
   b) $k_i = \min P_{i,n}^D$

Durch das Erfüllen dieser Kriterien haben wir wieder sicher gestellt, dass keine der Zellen doppelt vorkommt. Trotzdem generieren wir aber alle benötigten Zellen, um den lokalen Teil unseres Gleichungssystems vollständig aufzustellen.

3.7 Multilevel Lösung

Außer dem Aufstellen der Transferoperatoren P_{l-1}^{l} und R_{l}^{l-1}, benötigen wir für unser Lösungsverfahrens größten Teils lineare Algebra. Die wichtigste Operation ist dabei das Matrix-Vektor-Produkt. Wir wollen nun also Betrachten wie sich dieses in unserer Implementierung verhält. Im Allgemeinen muss für das Matrix-Vektor-Produkt der gesamte Vektor auf allen Prozessen vorhanden sein, um diesen auf die einzelnen Reihen der Matrix anzuwenden. Wir müssten also alles Kommunizieren und den gesamten Vektor auf allen Prozessen kopieren. Da unsere Steifigkeitsmatrix jedoch dünn besetzt ist, brauchen wir nur deutlich weniger Elemente des Vektors kommunizieren. Anschaulich brauchen wir nur dort zu kommunizieren, wo ein Patch ω_i einen Nachbarn $\omega_j \in C_i$ hat, welcher auf einem anderen Prozess liegt. Dies bedeutet also wir kommunizieren im Verhältnis zu den lokal berechenbaren Elementen (Volumen der lokalen Teilgebiete) nur wenige andere Zellen (Oberfläche der lokalen Teilgebiete). Um dies effizient nutzen können implementieren wir die Berechnung eines Kommunikationsmusters, welches uns Verrät, welche Elemente ein Prozess $\hat{q}$ mit welchen anderen Prozessen q austauschen muss. Das berechnen dieses Kommunikationsmusters erfolgt mit einer Komplexität von $O(N_{\text{nz}}^{\hat{q}} \log p + p)$, mit $N_{\text{nz}}^{\hat{q}}$ als der Anzahlen der lokalen nicht-Null Elemente der Matrix. Dieses Besetzungsmuster müssen wir jedoch nicht für jedes einzelne Produkt berechnen. Da unsere Vektoren eines Levels alle gleich verteilt sind, reicht es, dieses Besetzungsmuster nur einmal zu berechnen und anschließend, für alle Produkte mit einem Vektor gleicher Art, wieder zu verwenden.

In 3.5 hatten wir bereits sichergestellt, dass auch die hierarchischen Nachbarn kommuniziert wurden. Somit lässt sich hier auch die Prolongation P_{l-1}^{l} komplett lokal aufstellen. Die Aufteilung der Matrix geschieht hier, genau nach demselben Muster, wie bei der Steifigkeitsmatrix.

Das Transponieren der Prolongation, um die Restriktion zu erstellen, stellt sich jedoch als nicht ganz so gutmütig heraus. Im Allgemeinen ist das Transponieren einer nach Reihen verteilten, parallelen Matrix eine sehr teure Operation. Unsere Local-To-Local Prolongation hat jedoch eine besondere Eigenschaft. Ihre Gestalt ähnelt der einer Diagonalmatrix. Durch dieses Besetzungsmuster erhalten wir, dass eine Kommunikation nur dann von Nöten ist, wenn wenn der Vater eines Blattes auf einem anderen Prozess lebt. Dieser besondere Spezialfall kann jedoch nur am Rand unserer Verteilungsintervallgrenzen r_q eintreten. Damit erhalten wir also einen in der Komplexität konstanten Kommunikationsaufwand. Wir nehmen jedoch noch eine Änderungen vor. Mit dem Wissen, dass die Restriktion in unserem Lösungsverfahren mit dem Lösungsvektor multipliziert wird, können wir die Restriktion direkt so verteilen, wie es bei diesem Lösungsvektor der Fall ist. Dadurch ersparen wir uns den Kommunikationsaufwand bei jedem der Produkte von Restriktion und Lösungsvektor. Anschaulich, müssen wir also für das Transponieren, nicht nur kommunizieren, wenn der Vater eines Blattes auf einem anderen Prozess liegt, sondern wenn der Vater gemäß der letzten Verteilung, vor dem Verfeinern, auf einem anderen Prozess gelegen hat. In der Regel finden diese Verschiebungen jedoch auch nur am Rand der Intervalle statt und der Kommunikationsaufwand ist somit überschaubar. Doch auch im Allgemeinen, also auch bei starken Verteilungsänderungen, ist das Kommunizieren beim Transponieren empfehlenswert, da die erhöhte Kommunikation sonst auch bei jedem Restriktion-Lösungsvektor-Produkt auftritt.

4 Resultate

In diesem Kapitel wollen wir nun das Laufzeitverhalten unserer Implementierung mittels Messungen verifizieren. Dazu lösen wir folgendes Poisson Problem

$$\begin{aligned} -\Delta u &= 1 \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{auf} \quad \partial\Omega \end{aligned}$$

mit $\Omega = [0,1]^2$ auf verschiedenen Leveln, mit sich immer verdoppelnden Anzahlen von Prozessen. Wir verfeinern dabei immer Global, das gesamte Gebiete nach der H-Methode. Eine Verfeinerung nach der P-Methode findet nicht statt. Für unsere Lastbalance wählen wir eine Lastabschätzungsstrategie, welche nur die Anzahl der Patches betrachtet. Da der Polynomgrad überall gleich bleibt und wir global Verfeinern, erhalten wir damit, bis auf leichte Unregelmäßigkeiten am Rand des Gebiets, wo die Zellen weniger Nachbarn haben, eine ideale Verteilung. Unsere Patches ω_i wurden mit dem Faktor $\alpha = 1,5$ aus den Baumzellen $\mathcal{C}_i$ erstellt.

Als Rechnerinfrastruktur wurde dabei das Cluster „Hestia“ verwendet. Dieses besteht aus 64 Knoten mit jeweils 8 Prozessoren und 24GB Arbeitsspeicher. Verbunden sind diese Knoten mittels Infiniband. Aufgrund eines Fehlers in einigen der Knoten während des Messzeitraums, konnten jedoch nur 32 der gesamten Knoten verwendet werden. Somit ergibt sich ein Maximum von 256 Prozessoren und 768GB Arbeitsspeicher. Aufgrund der Speicherbegrenzung können Probleme mit hohen Freiheitsgraden erst bei höheren Anzahlen von involvierten Knoten (und damit Prozessen) durchgeführt werden. Trotzdem werden kleinere Problemgrößen auch mit hohen Prozessanzahlen berechnet. Es kann in den folgenden Tabellen also sowohl schwache Skalierung, als auch starke Skalierung abgelesen werden.

Wir teilen die Messergebnisse dabei in die verschiedene Schritte (Überdeckungserstellung, Operatorassemblierung, Lösen) während der Berechnung auf. Dadurch kann die Komplexität der einzelnen Teile getrennt von einander verifiziert werden. Alle Zeitangaben in diesem Kapitel werden in Sekunden gemacht.

Konstruktion der Überdeckung

Der erste Schritt in unserem Simulationsverfahren ist das Erstellen der Überdeckung. Dieser beinhaltet das Verfeinern des Baumes, Balancieren der Last, Neuverteilen der Last und das Berechnen und Austauschen aller Nachbarn. Insgesamt erwarten wir hier also eine Komplexität von $O(\frac{N}{p}\log p + \frac{\hat{N}}{p}J + p)$ mit N als Gesamtanzahl der Knoten im Baum und $\hat{N}$ als Anzahl der Blätter des Baumes. Tabelle 4.1 zeigt hier die Resultate der Messungen, welche diese Annahme

Problemgröße		Prozessanzahl								
Level	N	1	2	4	8	16	32	64	128	256
5	1,024	1.64_{-2}	2.49_{-2}	8.29_{-3}	6.99_{-3}	7.07_{-3}	1.94_{-2}	2.1_{-2}	2.99_{-2}	0.13
6	4,096	4.77_{-2}	2.88_{-2}	2.39_{-2}	1.2_{-2}	1.07_{-2}	1.53_{-2}	1.69_{-2}	3.83_{-2}	6.78_{-2}
7	16,384	0.16	0.29	0.25	0.24	1.85_{-2}	0.22	5.19_{-2}	4.87_{-2}	9.37_{-2}
8	65,536	0.61	0.34	0.38	0.3	0.25	0.24	4.05_{-2}	3.88_{-2}	7.07_{-2}
9	262,144	2.44	1.3	0.65	0.34	0.39	0.3	7.3_{-2}	7.05_{-2}	0.1
10	1,048,576	10.15	5.37	2.72	1.46	0.68	0.37	0.2	0.13	0.14
11	4,794,304	42.25	22.78	11.21	6	2.97	1.51	0.79	0.41	0.27
12	16,777,216						6.6	3.31	1.6	0.92
13	67,108,864								7.24	3.55

Tabelle 4.1: Zeiten für die Überdeckungserstellung auf den verschiedenen Leveln. Das Level und die Anzahl N der darauf für die Überdeckung erstellten Patches sind nach dem durchführen der Verfeinerung angegeben.

bestätigten. Betrachten wir hier die starke Skalierung auf tiefen Leveln, so sehen wir, dass mit wenigen Prozessen noch eine Verringerung der Messzeiten stattfindet, auf hohen Leveln jedoch wieder ein Anstieg bemerkbar ist. Dies kommt daher, dass bei den geringen Anzahlen von Zellen im Baum der prozessabhängige Anteil der Komplexität sehr schnell überwiegt. Bei höheren Zellenanzahlen tritt dieser Effekt nur noch sehr verringert auf.

Assemblieren der Operatoren

Beim Assemblieren betrachten wir zum einen das Aufstellen des Gleichungssystems, aus der Steifigkeitsmatrix A und dem Vektor der rechten Seite $\hat{f}$ und zum anderen das Assemblieren der Prolongation P_{l-1}^{l}. Das Transponieren der Prolongation, um die Restriktion zu erhalten, wird hier nicht betrachtet. Dieses ist im Falle der globalen Verfeinerung trivial, da keinerlei Kommunikation anfällt. Die Zeiten für dieses Transponieren sind hierbei um ein vielfaches kleiner, als jene zum Aufstellen der Prolongation.

Für das Aufstellen des Gleichungssystems und der Prolongation erwarten wir beides mal eine Komplexität von $O(\frac{\hat{N}}{p})$. Dies bedeutet, dass diese Schritte also ideal Skalieren. Die Tabelle 4.2 bestätigt diese Annahmen. Wir sehen für beide Schritte sowohl die erwartete schwache, als auch die starke Skalierung bestätigt.

Problemgröße	Prozessanzahl								
dof	1	2	4	8	16	32	64	128	256
Gleichungssystem									
3,072	0.21	0.1	5.41_{-2}	2.85_{-2}	1.53_{-2}	8.98_{-3}	4.25_{-3}	2.33_{-3}	1.3_{-3}
12,288	0.85	0.43	0.22	0.11	5.84_{-2}	3.04_{-2}	1.54_{-2}	8.18_{-3}	4.28_{-3}
49,152	3.5	1.77	0.89	0.45	0.23	0.12	5.91_{-2}	4.04_{-2}	2.07_{-2}
196,608	14.22	7.16	3.59	1.82	0.91	0.46	0.23	0.12	5.96_{-2}
786,432	58.99	29.78	14.91	7.46	3.74	1.91	0.99	0.47	0.27
3,145,728	237.42	118.94	60.05	30.14	15.05	7.57	3.78	1.93	0.99
12,582,912	955.26	478.38	240.47	120.16	60.34	30.37	15.11	7.54	3.82
50,331,648						122.93	61	30.42	15.19
201,326,592								122.78	61.45
Prolongation									
3,072	7.28_{-2}	3.62_{-2}	1.87_{-2}	9.52_{-3}	5.01_{-3}	4.26_{-3}	1.6_{-3}	1.68_{-3}	2.48_{-3}
12,288	0.3	0.15	7.52_{-2}	3.71_{-2}	1.9_{-2}	9.63_{-3}	5.09_{-3}	3.56_{-3}	3.32_{-3}
49,152	1.18	0.6	0.3	0.15	7.54_{-2}	3.75_{-2}	1.94_{-2}	1.08_{-2}	6.98_{-3}
196,608	4.73	2.4	1.19	0.6	0.31	0.16	8.28_{-2}	3.85_{-2}	2.12_{-2}
786,432	18.86	9.63	4.81	2.41	1.2	0.61	0.31	0.15	8.77_{-2}
3,145,728	76.05	38.29	19.26	9.66	4.85	2.43	1.2	0.63	0.33
12,582,912	301.54	152.85	77.32	38.4	19.18	9.69	4.82	2.43	1.21
50,331,648						38.81	19.32	9.66	4.85
201,326,592								38.89	19.39

Tabelle 4.2: *Oben:* Zeiten für die Assemblierung der Steifigkeitsmatrix und des Vektors der rechten Seite. *Unten:* Zeiten für das Assemblieren der Prolongation

Multilevellöser

Kommen wir nun zum Skalierungsverhalten unserer Multilevellösers. Wir erwarten hier die Komplexität $O(\frac{\hat{N}}{p} + (\frac{\hat{N}}{p})^{\frac{d-1}{d}} + J + \log p)$ wie es aus [4] bekannt ist. Betrachten wir jedoch die Messergebnisse für den Lösungschritt (Tabelle 4.3 oben), so bemerken wir, dass sich diese Annahme nicht bestätigt. Für wenige Prozesse skaliert der Löser (stark) noch sehr gut. Bei einer höheren Anzahl von Prozessen bricht die Skalierungsrate jedoch zusammen und die Lösungszeit steigt sogar wieder an. Wir begründen dies mit der Tatsache, dass in der Implementierung das Austausch von Daten zwischen den Prozessoren noch nicht ideal umgesetzt ist. Hier kommt ein Algorithmus der Komplexität $O(p)$ zum Einsatz, bei welchem jeder Prozess $\hat{q}$ zu allen anderen Prozessen q eine Verbindung aufbaut und kurz kommuniziert, ohne das wirklich Daten zwischen diesen beiden Prozessen ausgetauscht werden müssen. Dass diese Annahme richtig ist bestätigen die Messungen aus Tabelle 4.3 (Mitte und unten). Dabei wurden Zeiten beim Lösen für das Berechnen und für die Kommunikation getrennt von einander gemessen. Wir sehen hier, dass die Berechnungszeiten wie gewollt skalieren. Nur die Kommunikationszeit weist den Umstand auf, dass diese mit der Anzahl der Prozesse nach oben skaliert.

Problemgröße	Prozessanzahl								
dof	1	2	4	8	16	32	64	128	256
				Gesamt					
3,072	3.94_{-3}	2.13_{-3}	1.25_{-3}	1.09_{-3}	1.83_{-3}	1.68_{-2}	5.5_{-2}	0.14	0.35
12,288	1.27_{-2}	6.71_{-3}	3.6_{-3}	2.34_{-3}	2.79_{-3}	2.15_{-2}	7.08_{-2}	0.18	0.46
49,152	5.7_{-2}	3_{-2}	1.62_{-2}	8.16_{-3}	5.78_{-3}	2.75_{-2}	8.76_{-2}	0.22	0.58
196,608	0.24	0.13	6.87_{-2}	3.43_{-2}	1.91_{-2}	3.79_{-2}	0.11	0.25	0.67
786,432	0.96	0.51	0.29	0.15	7.35_{-2}	6.93_{-2}	0.14	0.3	0.78
3,145,728	3.88	2.01	1.13	0.58	0.29	0.19	0.21	0.39	0.93
12,582,912	16.81	8.75	4.82	2.31	1.17	0.64	0.45	0.63	1.25
50,331,648						2.4	1.38	1.05	1.48
201,326,592								3	2.48
				Berechnungen					
3,072	3.33_{-3}	1.69_{-3}	8.75_{-4}	4.71_{-4}	2.6_{-4}	1.72_{-4}	1.06_{-4}	8.15_{-5}	6.25_{-5}
12,288	1.14_{-2}	5.86_{-3}	2.98_{-3}	1.54_{-3}	8.13_{-4}	4.63_{-4}	2.65_{-4}	1.82_{-4}	1.32_{-4}
49,152	5.27_{-2}	2.74_{-2}	1.43_{-2}	6.65_{-3}	3.16_{-3}	1.6_{-3}	8.5_{-4}	5.1_{-4}	3.36_{-4}
196,608	0.22	0.11	5.97_{-2}	2.98_{-2}	1.46_{-2}	6.96_{-3}	3.32_{-3}	1.74_{-3}	1.2_{-3}
786,432	0.88	0.46	0.24	0.12	6.09_{-2}	3.07_{-2}	1.56_{-2}	7.53_{-3}	3.81_{-3}
3,145,728	3.55	1.81	0.96	0.49	0.25	0.12	6.39_{-2}	3.38_{-2}	1.85_{-2}
12,582,912	15.45	7.86	4.1	1.94	0.98	0.5	0.25	0.14	7.27_{-2}
50,331,648						1.98	1.01	0.52	0.28
201,326,592								2.06	1.07
				Kommunikation					
3,072	2.43_{-6}	9.76_{-5}	1.54_{-4}	4.02_{-4}	1.23_{-3}	1.59_{-2}	5.41_{-2}	0.14	0.35
12,288	3.86_{-6}	1.61_{-4}	2.48_{-4}	5.31_{-4}	1.62_{-3}	2.08_{-2}	7_{-2}	0.18	0.46
49,152	7.58_{-6}	5.52_{-4}	5.02_{-4}	9.05_{-4}	2.27_{-3}	2.55_{-2}	8.64_{-2}	0.22	0.58
196,608	1.35_{-5}	1.15_{-3}	1.66_{-3}	1.76_{-3}	3.16_{-3}	3.03_{-2}	0.11	0.25	0.67
786,432	1.58_{-5}	5.96_{-3}	3.72_{-3}	4.94_{-3}	5.71_{-3}	3.63_{-2}	0.12	0.29	0.78
3,145,728	1.87_{-5}	9.42_{-3}	1.36_{-2}	1.6_{-2}	1.2_{-2}	4.63_{-2}	0.14	0.36	0.92
12,582,912	2.49_{-5}	0.21	0.11	5.9_{-2}	3.7_{-2}	7.31_{-2}	0.17	0.5	1.19
50,331,648						0.14	0.24	0.48	1.2
201,326,592								0.74	1.34

Tabelle 4.3: Zeiten für das Lösen des Gleichungssystems, heruntergerechnet auf die Zeiten pro Iterationsschritt: *Oben:* Gesamtzeiten. *Mitte:* Berechnungen. *Unten:* Kommunikation

Für eine genauere Untersuchung des Sachverhalts messen wir, unabhängig von der restlichen Simulation, die Zeiten für einzelne Matrix-Vektor Produkte. Als Referenzoperationen dient dabei das Matrix-Vektor-Produkt der Steifigkeitsmatrix auf Level 11 (12582912 Freiheitsgrade) mit dem Vektor der rechten Seite auf dem selben Level. Als zweite Operationen nehmen wir das Produkt der Prolongation auf Level 11 mit dem Lösungsvektor des Levels darunter. Hierbei erhalten wir das selbe Verhalten wie für die Zeiten des gesamten Multilevellösers. Der Berechnungsteil skaliert wie erwartet, doch der Kommunikationsteil bricht Abhängig von der Anzahl der Prozesse zusammen. Das Produkt aus Prolongation und Lösungsvektor belegt dabei, dass das negative Verhalten nicht an einem schlecht skalierenden Kommunikationsvolumen liegt, da hier überhaupt keine Daten kommuniziert werden. Trotzdem erhalten wir dabei eine schlechte Skalierung bei der Kommunikation.

Das hier auftretende Problem, kann jedoch mit einer Anpassung der Austauschroutine behoben werden. Eine Kommunikation von jedem Prozess mit jedem anderen ist gar nicht nötig. Die Prozesse müssen nur mit denjenigen Prozessen kommunizieren, mit welchen sie Daten auszutauschen haben. Dabei handelt es sich anschaulich um die Prozesse, welche zu ihren lokalen Überdeckungen benachbart sind. Diese Anzahl dieser ist dabei bei schwacher Skalierung in $O(1)$. Die Berechnung der Prozesse, mit welchen überhaupt kommuniziert werden muss, kann direkt beim Erstellen des Kommunikationsmusters mitberechnet werden.

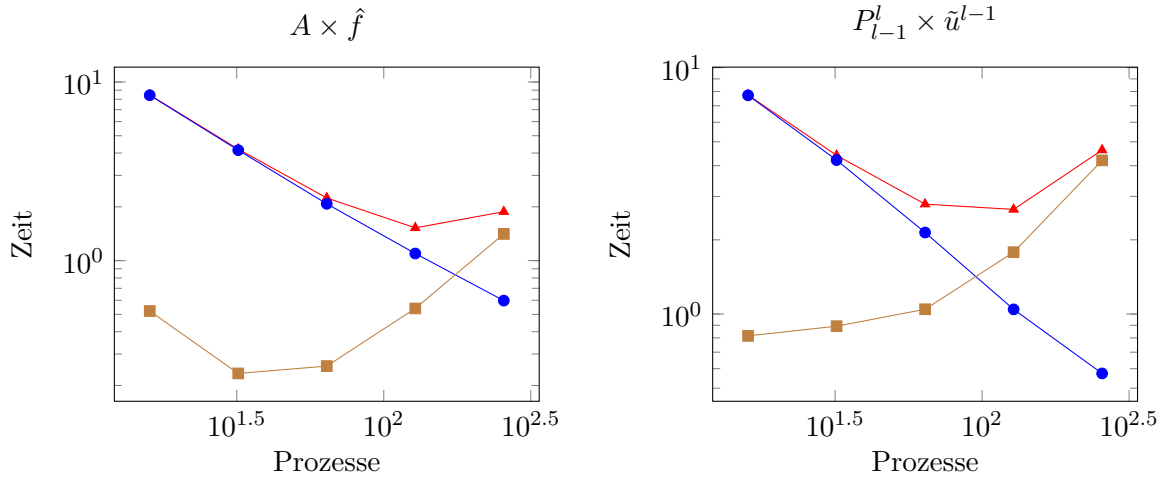

Abbildung 4.1: Skalierungsverhalten des Matrix-Vektor Produkts aufgeteilt in Gesamtzeit (rot), Berechnunganteil (blau) und Kommunikationsanteil (braun). *Links:* die Operation $A \times \hat{f}$. *Rechts:* die Operation $P_{l-1}^{l} \times \tilde{u}^{l-1}$

5 Zusammenfassung und Ausblick

Wir haben in dieser Arbeit die parallele Implementierung der PUM Vorgestellt und untersucht. Wir haben dazu eine Datenparallelität in der Überdeckungskonstruktion wie in [3] vorgestellt umgesetzt. Zusätzlich geht die hier vorgestellte Implementierung jedoch von einer Schrittweisen Verfeinerung des Baumes aus. Dabei wurde ein Verfahren Vorgestellt, welches zwischen jedem Schritt eine dynamische Lastbalance durchführen, welche über beliebige Lastabschätzungen und das gewünschte Problem angepasst werden kann. So ist es möglich sowohl adaptiv H als auch adaptiv P verfeinerte Probleme zu Balancieren.

Wir haben anschließend gezeigt, dass die Implementierung für ein einfaches, globales Problem sowohl bei der Erstellung der Überdeckung als auch dem Assemblieren der Operatoren wie erwartet skaliert. Bei der Verifikation des Skalierungsverhaltens des Lösers sahen wir ein Problem mit dem Kommunikationsaufwand, für welchen jedoch eine Lösung vorgestellt wurde.

Ausblick

Das in dieser Arbeit zur Verifikation genutzte Szenario ist ein sehr spezieller Fall, bei welchem der global verfeinerte Baum relativ leicht zu balancieren ist. Um das Skalierungsverhalten der Algorithmen auch bei stark adaptiv verfeinerten Bäumen zu präsentieren und zu verifizieren, müssten weitere Berechnungen mit anderen Szenarien durchgeführt werden. Kleine Messungen zeigten auch bei adaptiver H Verfeinerung ein auf den ersten Blick ideales Skalierungsverhalten, diese Annahmen müssten aber in größeren Experimenten bestätigt werden. Ebenso verhält es sich mit Problemen die sowohl H und P Verfeinerung an unterschiedlichen Stellen beinhalten. Hier wäre eine Lastabschätzung nur mit der Anzahl der Zellen nicht ausreichend. Der ebenfalls implementierte Lastabschätzer nach Freiheitsgraden schafft hier Abhilfe müsste jedoch ebenfalls in großen Experimenten verifiziert werden. Ein weiterer interessanter Fall, welcher hier nicht behandelt wurde, ist das Rechnen in drei Raumdimensionen. Auch dies wird vom Code unterstützt. Es fehlt jedoch noch an der Verifikation durch formale Experimente.

Literaturverzeichnis

[1] M. Bader, *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, vol. 9 of Texts in Computational Science and Engineering, Springer-Verlag, 2013. (Zitiert auf Seite 21)

[2] M. Griebel and M. A. Schweitzer, *A Particle-Partition of Unity Method for the solution of Elliptic, Parabolic and Hyperbolic PDE*, SIAM J. Sci. Comp., 22 (2000), pp. 853–890. also as SFB Preprint 600, SFB 256, Institut für Angewandte Mathematik, Universität Bonn. (Zitiert auf Seite 9)

[3] M. Griebel and M. A. Schweitzer, *A Particle-Partition of Unity Method—Part III: A Multilevel Solver*, SIAM J. Sci. Comp., 24 (2002), pp. 377–409. (Zitiert auf den Seiten 12, 16, 17 und 39)

[4] M. Griebel and M. A. Schweitzer, *A Particle-Partition of Unity Method—Part IV: Parallelization*, in Meshfree Methods for Partial Differential Equations, M. Griebel and M. A. Schweitzer, eds., vol. 26 of Lecture Notes in Computational Science and Engineering, Springer, 2002, pp. 161–192. (Zitiert auf den Seiten 7, 20, 21, 22, 23 und 35)

[5] D. Moore, *The cost of balancing generalized quadtrees*, in In Proceedings of the 3rd Symposium on Solid Modeling and Applications, 1995, pp. 305–312. (Zitiert auf Seite 25)

[6] M. A. Schweitzer, *Ein Partikel-Galerkin-Verfahren mit Ansatzfunktionen der Partition of Unity Method*, diplomarbeit, Institut für Angewandte Mathematik, Universität Bonn, 1997. (Zitiert auf Seite 9)

[7] M. A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*, Dissertation, Institut für Angewandte Mathematik, Universität Bonn, 2002. (Zitiert auf Seite 9)

[8] M. A. Schweitzer, *Meshfree and Generalized Finite Element Methods*, habilitation, Institute for Numerical Simulation, University of Bonn, 2008. (Zitiert auf den Seiten 7 und 9)

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Ort, Datum, Unterschrift)