Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 123

Effiziente verteilte Hierarchisierung und Dehierarchisierung auf vollen Gittern

Philipp Butz

Studiengang:	Informatik
Prüfer:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Betreuer:	M.Sc. Mario Heene
Beginn am:	10. April 2014
Beendet am:	10. Oktober 2014
CR-Nummer:	G.1.0

Kurzfassung

In vielen wissenschaftlichen Bereichen werden numerische Verfahren für komplexe Simulationen eingesetzt. Diese basieren auf beliebig dimensionalen numerischen Gleichungen, welche beispielsweise für die Simulation von Aktienmärkten mehrere 100 Dimensionen umfassen können. Auf Grund der Größe der Gitter liegen die Daten meist verteilt vor. Die vollen Gitter für die Lösung dieser höher dimensionalen Probleme unterliegen dem Fluch der Dimensionalität. Ein Ansatz, um dem entgegen zu wirken, sind dünne Gitter, welche auf hierarchischen Basen basieren. Diese Arbeit widmet sich insbesondere der Problematik der verteilten Gitter, wobei die verteilte Transformation der nodalen in die hierarchische Basis thematisiert wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

In vielen wissenschaftlichen Bereichen werden numerische Verfahren für komplexe Simulationen eingesetzt. Beispielsweise für die Untersuchung von physikalischen Phänomenen, welche schwer oder sogar unmöglich durch Experimente nachvollziehbar sind [1]. Die Simulationen von komplexen Vorgängen basieren auf beliebig dimensionalen numerischen Gleichungen, welche beispielsweise für die Simulation von Aktienmärkten mehrere 100 Dimensionen umfassen können.

Die vollen Gitter für die Lösung dieser höher dimensionalen Probleme unterliegen dem Fluch der Dimensionalität, da die Anzahl der Gitterpunkte $N = n^d$, bei n Gitterpunkten pro Dimension, exponentiell mit der Anzahl der Dimensionen wächst. Bei steigender Genauigkeit wachsen somit sowohl Gitter als auch die Zeit für die Berechnung exponentiell.

Ein Ansatz, um der Größe entgegen zu wirken, sind dünne Gitter, welche auf hierarchischen Basen basieren. Für beliebig dimensionale Probleme wird ein Tensorprodukt dieser Basen gebildet. Die hierarchische Struktur führt dazu, dass der Beitrag zur Lösung mit steigendem Level kleiner wird. Für die dünnen Gitter werden nur Basisfunktionen verwendet, welche in jede Richtung nur ein bestimmtes Level aufweisen. Die Gitterstruktur ist daraufhin nicht mehr regulär, wodurch die direkte Anwendung von Algorithmen, zur Berechnung einer Lösung, erschwert wird.

Die Kombinationstechnik setzt an diesem Problem an [2]. Hierbei werden dünne Gitter durch die Kombination von regulären Teilräumen dargestellt. Die Lösung wird auf jedem der Teilräume separat berechnet und kombiniert.

1.1 Problemstellung

Der Gegenstand dieser Arbeit ist die Transformation der nodalen Basis in die hierarchische Basis. Für diese Hierarchisierung und dessen inverse, die Dehierarchisierung, existieren Algorithmen bei denen die Daten komplett in einem Speicherbereich liegen. Bei großen beliebig dimensionalen Problemen ist es jedoch oft der Fall, dass die Daten auf mehrere Knoten aufgeteilt vorliegen.

In dieser Arbeit werden Verfahren für die direkte Hierarchisierung und Dehierarchisierung auf verteilten Daten entwickelt. Diese werden in das Dünngitter-Framework SG++ eingebunden und mit Messungen bewertet [3, 4].

1.2 Gliederung

In Kapitel 2 werden die mathematischen Grundlagen für die Problemstellung dieser Arbeit erläutert. Dazu zählen insbesondere hierarchische Basisfunktionen und das Konzept der dünnen Gitter, welches auf diesen Basisfunktionen aufbaut.

In Kapitel 3 wird beginnend ein Algorithmus für die serielle Lösung der Hierarchisierung beschrieben. Diese wird schrittweise erweitert zu einer verteilten Hierarchisierung und Dehierarchisierung, wobei zwei Varianten für die verteilte Dehierarchisierung vorgestellt werden.

In Kapitel 4 werden die Lösungsansätze durch Messungen bewertet und verglichen. Insbesondere werden die Laufzeiten und Skalierung der zwei Varianten der verteilten Dehierarchisierung betrachtet.

Im letzten Kapitel werden die Erkenntnisse dieser Arbeit zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

2 Grundlagen

Im folgenden Kapitel werden die für die Hierarchisierung wichtigen mathematischen Grundlagen erläutert, sowie die Grundzüge der Möglichkeiten durch die Dünn-Gitter-Methode als Motivation. Die Definitionen und Notationen sind aus der Doktorarbeit „Spartially Adaptive Sparse Grids for High-Dimensional Problems“ von Dirk Pflüger entnommen [3].

Für das wissenschaftliche Rechnen werden Probleme bzw. Funktionen beliebiger Dimensionalität d diskretisiert. Dafür werden die Funktionen an bestimmten Stellen ausgewertet und anhand dieser Werte mit dazugehörigen Ansatzfunktionen approximiert. Diese Arbeit verwendet reguläre volle Gittern in dem Bereich $[0,1]^d$.

Die Anzahl der Punkte in einer Dimension wird durch den level Vektor $\overrightarrow{l}$ vorgegeben, wobei dieser für jede Dimension das Disktretisierungslevel

$$\overrightarrow{l} = (l_1, ..., l_d) \qquad l_i \in \mathbb{N} \tag{2.1}$$

definiert. Aus dem Level einer Dimension i resultiert der Abstand der Stützstellen und die Anzahl der Punkte. Der Abstand h zwischen zwei Stützstellen in einer Dimension i ist durch

$$h_i = 2^{-l_i} \tag{2.2}$$

definiert. Die Anzahl der Punkte in Dimension i resultiert daraus zu

$$N_i = \begin{cases} 2^{l_i} + 1, & \text{falls Randwerte in Dimension } i \text{ vorhanden sind} \\ 2^{l_i} - 1, & \text{sonst} \end{cases} \tag{2.3}$$

Die Stützstellen bilden ein volles Gitter, wie in Abbildung 2.1 für $d = 2$ und $\overrightarrow{l} = (3, 3)$ dargestellt.

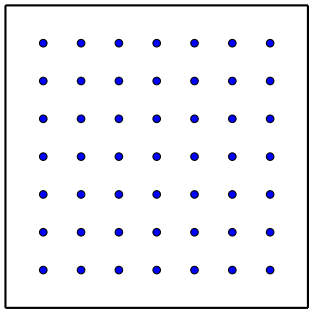

Abbildung 2.1: Volles Gitter mit $l = (3, 3)$ ohne Randwerte (49 Punkte)

2.1 Interpolation mit der Knotenbasis

Die Funktionswerte an den Stützstellen werden stückweise linear interpoliert, wie in Abbildung 2.3 dargestellt. Hierbei werden die Funktionswerte α_j an den Stützstellen x_j jeweils mit einer Hutfunktion, wie in Abbildung 2.2 grafisch und in Definition 2.4 formal dargestellt, gewichtet.

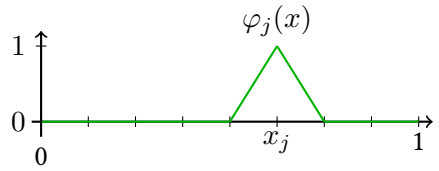

Abbildung 2.2: Hutfunktion aus Definition 2.4 für eine Stützstelle x_j

(2.4) $\varphi_j(x) := \max(1 - |x - x_j|, 0)$

Eine Funktion $f(x)$ lässt sich approximieren durch die Summe der Funktionswerte an den gewählten Stützstellen multipliziert mit deren Gewichtung durch je eine Ansatzfunktion. Die formale Definition

(2.5) $f(x) \approx u(x) := \sum_{j=1}^{N_i} \alpha_j \cdot \varphi_j(x)$

ist grafisch in Abbildung 2.3 dargestellt. Hierbei zeigt die rechte Grafik die gewichteten Basisfunktionen und die Summe $u(x)$ aller Ansatzfunktionen. In der linken Grafik ist eine Funktion $f(x)$ und ihre Approximation $u(x)$, aus dem rechten Teil der Abbildung, dargestellt.

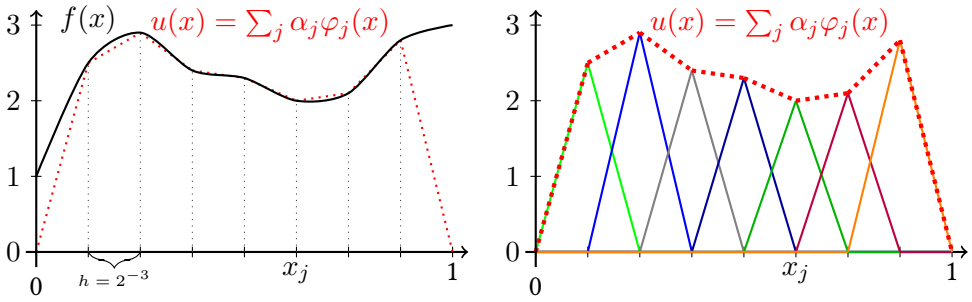

Abbildung 2.3: Links: Eine Funktion $f(x)$ und deren lineare Approximation $u(x)$, Rechts: Die Funktion $u(x)$ als Summe der gewichteten Basisfunktionen, basiert auf [3]

2.2 Hierarchische Basisfunktionen

Im Gegensatz zu den Ansatzfunktionen aus Definition 2.4 sind hierarchische Basen definiert durch

(2.6) $\varphi_{l,i}(x) := \varphi(2^l x - i)$

mit Level l und Index $i \quad 0 < i < 2^l$. Der Unterschied zwischen den hierarchischen Hutfunktionen und den Standart-Hutfunktionen ist in Abbildung 2.4 dargestellt.

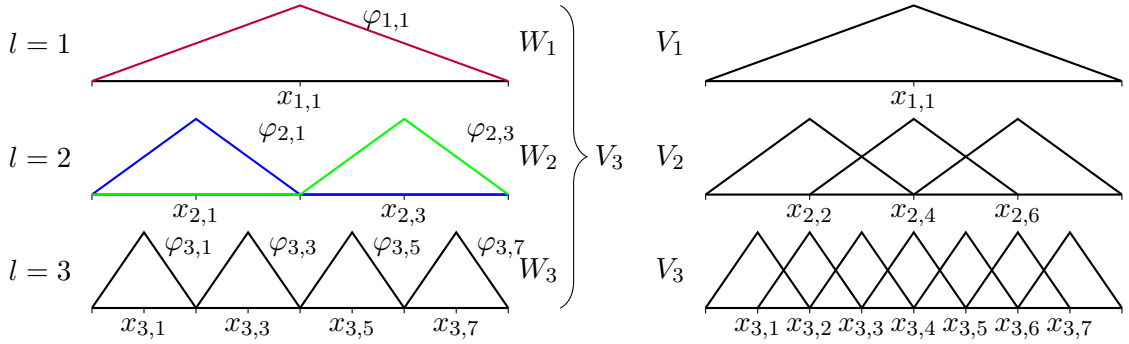

Abbildung 2.4: Links: Hierarchische Hutfunktionen, Rechts: Standart-Hutfunktionen, basiert auf [3]

In der linken Grafik der Abbildung 2.4 ist die hierarchisierte Basis aus Hutfunktionen für die Level 1 bis 3 dargestellt. Es ist ersichtlich, dass die Hutfunktionen auf dem gleichen Level jeweils einen gleich großen Bereich abdecken und sich summiert auf den gesamten Wertebereich erstrecken. Im Gegensatz dazu, überlappen sich die Basisfunktionen der Knotenbasis. Die formale Definition der Teilräume V und W sind in Definition 2.10 und 2.9 einzusehen und werden im Abschnitt 2.3 erläutert.

Die Basisfunktionen für höher dimensionale Probleme sind das Tensor-Produkt der Basisfunktionen in jede Dimension an jeder Stützstelle. Diese sind somit definiert als:

(2.7) $$\varphi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j)$$

Die Definition der Approximation $u(\vec{x})$ lässt sich damit definieren durch die Formel in Definition 2.13 im folgenden Abschnitt.

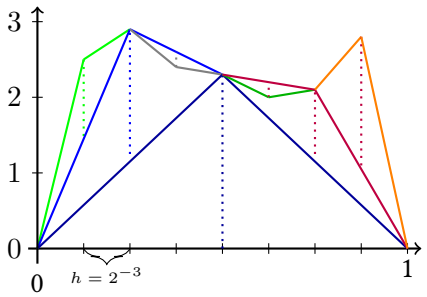

Abbildung 2.5: Die Funktion $u(x)$ und deren Ansatzfunktionen $\varphi_{l,j}$, basiert auf [3]

2.3 Dünne Gitter und Kombinationstechnik

Dünne Gitter bestehen aus wesentlich weniger Punkten als volle Gitter und werden durch eine Kombination von Teilräumen aus Basisfunktionen, welche abhängig von der Wahl einer entsprechenden Norm ist, gebildet. Für 1D wurden in Abbildung 2.4 Teilräume W_1, W_2 und W_3 dargestellt.

Die Teilräume sind definiert durch eine Menge an Indizes

$$I_{\vec{l}} := \{\vec{i} : 1 \leq i_j \leq 2^{l_j} - 1,\ i_j \text{ ungerade},\ 1 \leq j \leq d\} \tag{2.8}$$

welche die enthaltenen Basisfunktionen definiert.

Aus diesen Mengen an Basisfunktionen lassen sich, wie in Abbildung 2.9 auf Seite 17, die Teilräume $W_{\vec{l}}$ definieren durch

$$W_{\vec{l}} := span\{\varphi_{\vec{l},\vec{i}}(\vec{x}) \ :\ \vec{i} \in I_{\vec{l}}\} \tag{2.9}$$

Die Dünn-Gitter-Methode besteht darin die Teilräume, welche am meisten zur Lösung beitragen, zu kombinieren, um die Genauigkeit der Approximation, trotz weniger Stützstellen, nicht wesentlich zu verschlechtern. In Abbildung 2.9 werden die, in blau dargestellten, Stützstellen zu einem dünnen Gitter kombiniert, welches auf der rechten Seite der Abbildung dargestellt ist.

Der Dünngitterraum V_n ist definiert durch

$$V_n := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} \tag{2.10}$$

wobei die Summennorm definiert ist durch

$$|\vec{l}|_1 := \sum_{j=1}^{d} l_j \tag{2.11}$$

Damit kann die, in Abbildung 2.9 auf der rechten Seite dargestellte, Kombination von Teilräumen als V_3 eindeutig bestimmt werden. Am Beispiel der Teilräume in Abbildung 2.9 lässt sich leicht einsehen, dass das dünne Gitter gerade einmal 17 Gitterpunkte enthält im Gegensatz zu den 49 eines vollen Gitters.

Durch das Ändern der Norm der direkten Summe der Teilräume lassen sich dünne Gitter mit verschiedenen Eigenschaften erstellen. Beispielsweise lässt sich das volle Gitter durch die Maximumsnorm herstellen, da die Maximumsnorm definiert ist durch

$$|\vec{l}|_\infty := \max_{1 \leq j \leq d} |l_j| \tag{2.12}$$

Damit lässt sich die Approximation $u(x)$ auf dem Dünngitterraum V_n definieren als

$$u(x) := \sum_{\substack{|\vec{l}| \leq n+d-1 \\ \vec{i} \in I_{\vec{l}}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}) \tag{2.13}$$

Für genügen glatte Funktion $f \in H_2^{\text{mix}}$ ergibt sich bezüglich der Approximation $u(x) \in V_n$ eine Approximationsordnung von

(2.14) $||f(\overrightarrow{x} - u(\overrightarrow{x}))||_{L_2} \in O(h_n^2(log(h_n^{-1}))^{d-1})$

2.4 Randwerte

In der bisherigen Betrachtung wurden für die Randwerte keine Werte verschieden der Null angenommen. Für Funktionen wie in Abbildung 2.3 verbessern Randwerte die Approximation $u(x)$ deutlich. Diese können auf einem neuen Level, Level 0, definiert werden, jedoch entstehen dadurch neue Teilräume. Bei der Kombination dieser führt dies dazu, dass auf dem Rand doppelt so viele Punkte wie auf der Hauptachse innerhalb des Gitters sind. Aus diesem Grund werden die Randwerte auf Level 1 hinzugefügt und bei der Hierarchisierung und Interpolation speziell behandelt.

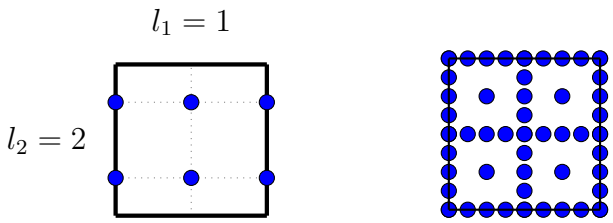

Abbildung 2.6: Links: Teilraum W_l mit $l = \{1, 2\}$. Rechts: V_3 aus Abbildung 2.9 ergänzt um Randwerten

Zwei zusätzliche Basisfunktionen $\varphi_{0,1}$ und $\varphi_{0,2}$ erweitern die Approximation aus Abbildung 2.3 zu einer Approximation mit Randwerten, welche in Abbildung 2.8 dargestellt ist.

Die Teilräume lassen sich damit wie in der linken Grafik in Abbildung 2.6 darstellen. Die Teilraumkombination V_3 aus Abbildung 2.9, ebenfalls mit Randwerten, ist auf der rechten Seite zu sehen.

Die Abhängigkeiten zwischen den Stützstellen lassen sich damit abermals durch einen Baum darstellen. Abbildung 2.7 ist eine Erweiterung des Baumes aus Abbildung 2.10 und beinhaltet die neuen Abhängigkeiten zwischen den Stützstellen und den Randwerten.

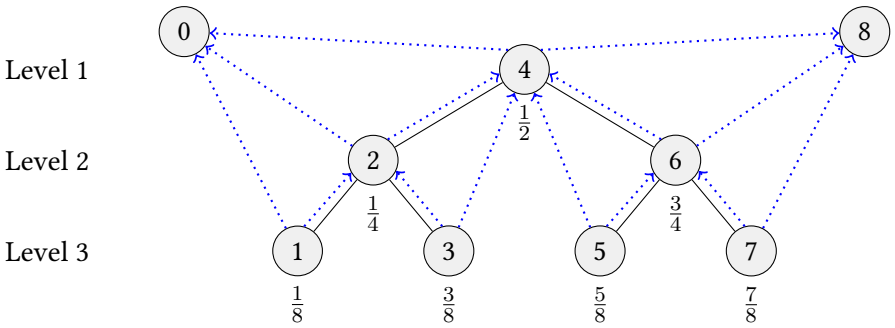

Abbildung 2.7: Erweiterung der Baumdarstellung aus Abbildung 2.10 um Randwerte

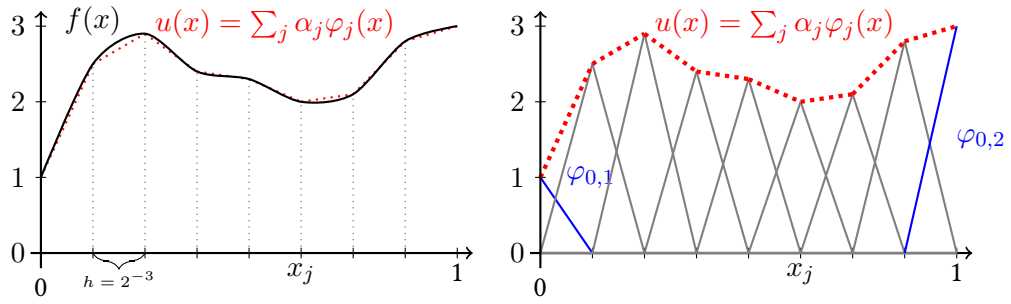

Abbildung 2.8: Links: Eine Funktion $f(x)$ und deren lineare Approximation $u(x)$ mit Randwerten, Rechts: Die Funktion $u(x)$ als Summe der Stützstellen x_j und deren Ansatzfunktionen φ_j mit Randwerten durch die Basisfunktionen $\varphi_{0,1}$ und $\varphi_{0,2}$, basiert auf [3]

2.5 Hierarchisierung und Dehierarchisierung

Die Hierarchisierung beschreibt die Transformation der nodalen in die hierarchische Basis. Die Dehierarchisierung beschreibt entsprechend die inverse Transformation. Bei diesen Transformationen werden die Werte an den Stützstellen neu bestimmt.

Die Funktionswerte auf größeren Leveln tragen die Differenz zwischen dem Funktionswert an dessen Stützstelle und der schon vorhandenen Approximation bei. Eine Stützstelle x_j wird in Abhängigkeit der Stützstellen auf niedrigeren Level neu berechnet zu

$$\begin{aligned} x_j &= x_j - 0.5 \cdot (x_{left} + x_{right}) \\ &\text{mit } left \;\; = j - 2^{l_i - l_{x_j}} \\ &\text{mit } right = j + 2^{l_i - l_{x_j}} \\ &\text{mit } \; l_{x_j} := \; (\text{Level der Stützstelle } \; x_j) - 1 \end{aligned} \tag{2.15}$$

wobei der $x_{j-2^{l_i - l_{x_j}}}$ der linke und $x_{j+2^{l_i - l_{x_j}}}$ der rechte hierarchische Vorgänger der Stützstelle x_j ist.

Dabei werden die Werte beginnend bei dem Level l_i hierarchisiert, wodurch die Abarbeitung bezüglich des Binärbaums einer „reverse level-order“ Traversierung entspricht. Die Dehierarchisierung entspricht der Inversen der Hierarchisierung damit der „level-order“ Traversierung des Binärbaums.

Weitere Formeln für die Berechnung von Nachbarschaftsbeziehungen zwischen den Funktionswerten abhängig von dem jeweiligen Level der Stützstelle x_j werden im Kapitel 3 vorgestellt.

Die Abhängigkeiten lassen sich übersichtlicher durch eine Baumstruktur, wie in Abbildung 2.10, beschreiben. In dieser Grafik ist ein Baum für die eindimensionalen Hierarchisierung mit $l = 3$ dargestellt. Die gestrichelten blauen Linien in dem Binärbaum symbolisieren die Abhängigkeiten. Für die Stützstellen x_4, x_6 und x_7 sind die Verbindungen zu deren Ansatzfunktionen in dem rechten Teil der Abbildung verdeutlicht.

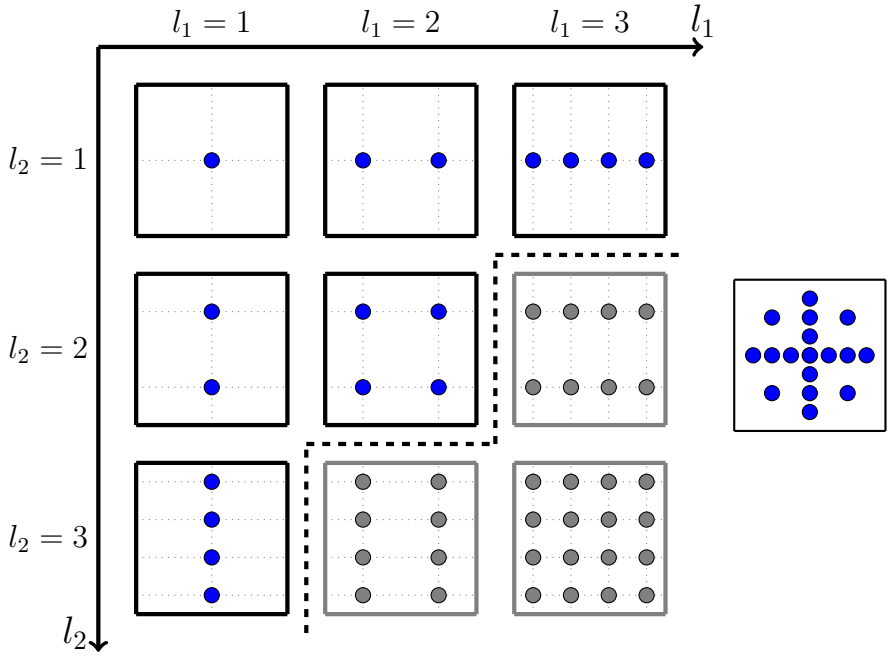

Abbildung 2.9: Links: Alle Teilräume für $l = 3$ in beiden Dimensionen, Rechts: Dünnes Gitter V_3 bestehend aus allen blau dargestellten Stützstellen im linken Teil der Grafik [3]

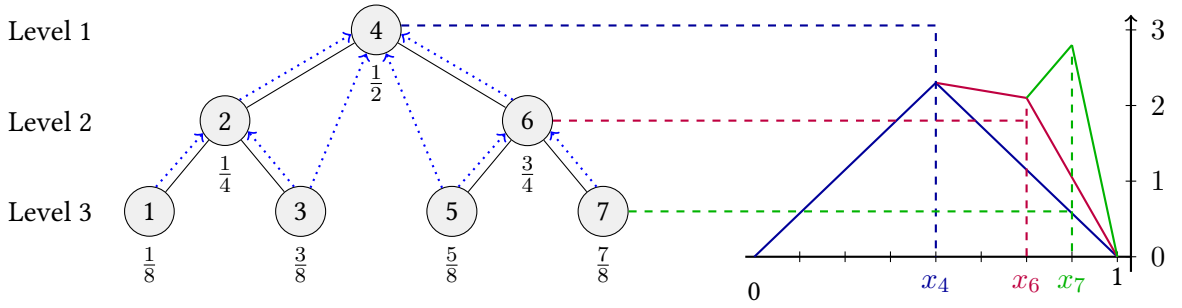

Abbildung 2.10: Baumstruktur der hierarchisierten Punkte. Die gestrichelten blauen Linien beschreibt die Vorgängerbeziehung zwischen den Punkten für $l = 3$, basiert auf [3]

3 Implementierung

Basierend auf den mathematischen Grundlagen aus Kapitel 2 werden in diesem Kapitel Algorithmen für das verteilte Umrechnen zwischen nodalen Basen und hierarchisierten Basen erarbeitet. Hierfür werden die Algorithmen für die verteilte Hierarchisierung und Dehierarchisierung aus dem seriellen Algorithmus hergeleitet.

3.1 Ausgangspunkt

Abstrakt lässt sich die mathematische Definition der Hierarchisierung aus Kapitel 2 zum Beispiel durch den Algorithmus 3.1 basiernd auf dem Algorithmus aus dem Paper „Performance of Unidirectional Hierarchization for Component Grids Virtually Maximized“ von Philipp Hupp darstellen [5].

Hierbei wird in der äußeren Schleife, in Zeile 2, über alle Dimensionen iteriert, so dass die Hierarchisierung nacheinander für jede Dimension ausgeführt wird. In Zeile 3 wird über alle „Pole“ in der entsprechenden Dimension dd iteriert.

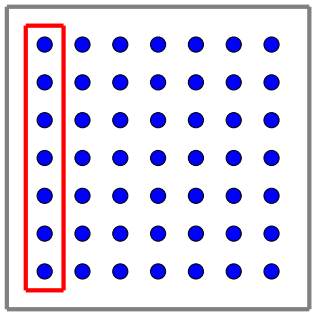

Abbildung 3.1: Ein Pol bezüglich der Dimension 2 eines vollen Gitters mit $l_2 = 3$

Algorithmus 3.1 Basis Hierarchisierungs-Algorithmus, basiert auf [5]

```
1:  function HIERARCHISIERUNG
2:      for dd ← 1 to d do
3:          for all 1-dim Pole P in Richtung dd do
4:              for l ← l_dd to 2 do
5:                  for all x_i auf Level l do
6:                      grid[i] = grid[i] − 0.5 · linkerVorgänger(i, dd, l)
7:                      grid[i] = grid[i] − 0.5 · rechterVorgänger(i, dd, l)
8:                  end for
9:              end for
10:         end for
11:     end for
12: end function
```

Ein Pol bezüglich einer Dimension i ist eine Menge an Stützstellen, welche sich nur in der Koordinate bezüglich der Dimension i unterscheiden. Ein Beispiel für ein Pol ist in Abbildung 3.1 dargestellt. Jeder dieser Pole ist bezüglich der Dimension i als unabhängiges eindimensionales Gitter zu betrachten. Die Anzahl der Pole einer Dimension i errechnet sich somit durch

$$Pole_i := \prod_{\substack{j=0 \\ i \neq j}}^{d} 2^{l_j} - 1 \tag{3.1}$$

Der Algorithmus 3.1 behandelt keine Randwerte, wodurch in Zeile 3 über alle Level, beginnend mit dem höchsten, bis zum Level 2 iteriert wird. Der Funktionswert an dem Stützpunkt auf Level 1 muss nur für Randwerte ungleich 0 betrachtet werden. In Zeile 5 wird über alle Funktionswerte in dem jeweiligen eindimensionalen Pol und dem entsprechenden Level iteriert. Daraufhin werden die neuen Werte der Funktionswerte, wie in Definition 2.15, durch die Zeilen 6-7 berechnet, wobei die Funktionen jeweils den Wert des linken und rechten hierarchischen Vorgängers ausgibt. Der Ablauf des Algorithmus für einen Pol (Zeile 4ff) ist in Abbildung 3.2 grafisch dargestellt.

Die Level werde entsprechend dem Algorithmus 3.1 von großem nach kleinen Level durchlaufen, wobei der Ablauf in Abbildung 3.2 für jeden Pol wiederholt wird.

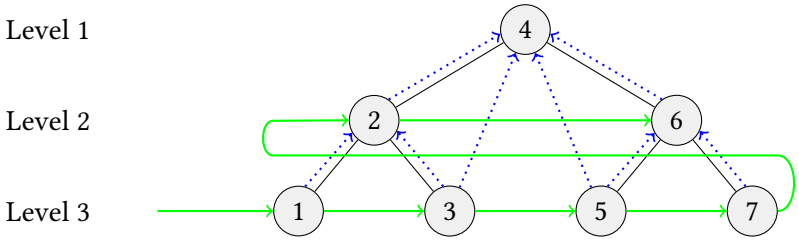

Abbildung 3.2: Blau: hierarchische Vorgänger, Grün: Ablauf der Hierarchisierung nach Algorithmus 3.1 für einen Pol.

3.1.1 Datenstruktur

An dieser Stelle gilt es die Datenstruktur für die Speicherung der Werte an den Stützstellen zu beschreiben. Die Anzahl der Dimensionen kann variieren, dabei soll die Datenstruktur möglichst generisch sein. Aus diesem Grund werden die Funktionswerte in einem eindimensionaler Array (grid[]) des entsprechenden Datentyps der Funktionswerte (z.B. „double“) gespeichert. Die Reihenfolge der Speicherung ist einfach gehalten, so dass sich die Pole bezüglich Dimension 1 aneinanderreihen. Somit liegen die Stützstellen in der ersten Dimension aus Sicht des Speichers in direkter Nachbarschaft, wodurch sich die „Schrittweite“ im Array zwischen den einzelnen Werten eines Pols dieser Dimension als 1 ergibt. Für alle weiteren Dimensionen herrschen von 1 verschiedene Schrittweiten zwischen den

Werten innerhalb eines Pols. Diese Schrittweite im Array zwischen den Werten eines Pols bezüglich einer Dimension i ergibt sich zu

$$Schrittweite_i := \prod_{j=1}^{i-1} N_j \qquad N_j \quad \text{aus Definition 2.3} \tag{3.2}$$

3.2 Randwerte

Randwerte sollen so generisch wie möglich sein, so dass der Basisalgorithmus um variable Randwerte in jeder Dimension erweitert wird. Hierfür wird zu Beginn für jede Dimension festgelegt, ob diese Randwerte enthält.

In Kapitel 2 wurden die formalen Änderungen für diese Erweiterung vorgestellt. Aus algorithmischer Sicht ändern sich im wesentlichen zwei Faktoren, wobei sowohl die Struktur der Speicherung als auch der generelle Ablauf des Algorithmus erhalten bleibt.

Zum einen hinsichtlich Zeile 4 in Algorithmus 3.1. Hierbei müssen die Funktionswerte auf Level 1 nun zusätzlich bezüglich den Randwerten hierarchisiert werden, sofern welche in der aktuell betrachteten Dimension dd vorhanden sind.

Zum anderen hinsichtlich der Funktionen in Zeile 6 und 7. Hierbei muss überprüft werden, ob es einen linken bzw. rechten hierarchischen Vorgänger gibt. Falls es Randwerte gibt, gibt es auch einen linken hierarischen Vorgänger der Punkte $1, 2$ und 4 in Abbildung 3.2.

Da die Vorgängerberechnung bisher nicht ausführlich betrachtet wurde, im weiteren Verlauf allerdings von Bedeutung ist, stellt der Algorithmus 3.2 die Funktion „linkerVorgänger()“ dar. Hierbei ist der Übergabeparameter i die Position von x_i im Array aus Algorithmus 3.1 Zeile 5. dd ist die Dimension, in welcher hierarchisiert wird (Algorithmus 3.1 Zeile 2) und l ist das Level von x_i bezüglich Dimension dd (Algorithmus 3.1 Zeile 5).

Für den linken Nachfolger gibt es drei Möglichkeiten, welche durch die „if“-Bedingungen in Zeile 3 und 5 in Algorithmus 3.2 abgefragt werden.

1. Der Vorgänger ist kein Randwert (Zeile 3).
2. Der Vorgänger ist ein Randwert und es gibt in Dimension dd Randwerte (Zeile 5).
3. Der Vorgänger ist ein Randwert und es gibt in Dimension dd keine Randwerte (Zeile 7).

Die Berechnung des rechten hierarischen Vorgängers ist ähnlich, im Algorithmus 3.2 wird in Zeile 2 der Offset bezüglich des Vorgängers nicht subtrahiert sondern addiert.

Für die ersten zwei Bedingungen ist die berechnete Position $linkerVorgänger$ ein gültiger Index bezüglich des Arrays *grid*[] und eine gültige Position bezüglich der mathematischen Beziehungen zwischen den Punkten x_i und $x_{linkerVorgänger}$.

Algorithmus 3.2 Berechnung des Wertes des linken hierarchischen Vorgänger zum Punkt i

```
1:  function leftPredecessor(i, dd, l)
2:      linkerVorgänger := i − Schrittweite · (2^(l_dd − l))        // Definition 2.15
3:      if linkerVorgänger ist kein Randwert then
4:          return grid[linkerVorgänger]                  // grid[] Array mit Funktionswerten
5:      else if linkerVorgänger ist Randwert und Dimension dd hat Randwerte) then
6:          return grid[linkerVorgänger]
7:      else
8:          return 0
9:      end if
10: end function
```

3.3 Dehierarchisierung

Die Dehierarchisierung ist das Umwandeln der hierarchisierten Basis in die ursprüngliche nodale Basis. Die Dehierarchisierung ist somit das rückwärts Anwenden des Algorithmus 3.1.

Explizit bedeutet dies, dass die „for“-Schleife in Zeile 4 von hinten beginnt. Die Dehierarchisierung beginnt damit bei dem kleinsten Level. Dieses ist Level 1, falls Randwerte vorhanden sind, und Level 2 falls keine vorhanden sind.

Die Reihenfolge der Abarbeitung der Pole sowie der einzelnen x_i muss dabei nicht verändert werden. Im Fall der Pole ist dies leicht ersichtlich, da die Pole einer Dimension i bezüglich dieser unabhängig voneinander sind. Die Unabhängigkeit der einzelnen x_i mit gleichem Level lässt sich durch die Baumstruktur in Abbildung 3.2 einsehen.

Da dieser Aspekte für die spätere verteilte Dehierarchisierung eine Rolle spielt, wird nochmals verdeutlicht, dass die Werte beginnend bei dem kleinsten Level dehierarchisiert werden. Die Dehierarchisierung der x_i auf höheren Level setzt somit die Dehierarchisierung der x_i der kleineren Level voraus.

3.4 Verteilte Hierarchisierung und Dehierarchisierung

Für die Verteilung der Daten und das anschließende Hierarchisieren oder Dehierarchisieren gibt es mehrere Möglichkeiten.

Ein naiver Ansatz ist, die Daten auf einem Prozess zu sammeln, sofern die Kapazität des Hauptspeicher dies zulässt, diese zu hierarchisieren oder dehierarchisieren und daraufhin auf die Prozesse zu verteilen. Dabei bildet der Kommunikationsschritt den Flaschenhals der Hierarchisierung beziehungsweise Dehierarchisierung, da alle Prozesse ihre Daten an einen Prozess senden und dieser, nach Beenden der Bearbeitung, die veränderten Daten zurück verteilen muss.

Alternativ kann die Hierarchisierung und Dehierarchisierung verteilt auf allen Prozessen ablaufen. Diese Variante beinhaltet mehr Komplexität, bezüglich des Datenaustausches und der eigentlichen

Verarbeitung der Daten, verzichtet jedoch auf das Sammeln und Verteilen aller Daten auf einem Prozess. Diese Alternative wird im Folgende erläutert und in Kapitel 4 mit Laufzeitmessungen bewertet.

3.4.1 Aufteilung der Daten

Zu Beginn gilt es die Anforderungen an die Aufteilung der Daten zu definieren.

Die Anzahl der Level ist für jede Dimension beliebig wählbar, so dass es sinnvoll erscheint auch eine unterschiedliche Aufteilung für jede Dimension zuzulassen. Abbildung 3.3 stellt zwei Gitter mit einer, in rot dargestellten, Unterteilung in 4 (links) beziehungsweise 9 Prozesse (rechts) dar.

Dabei wurde die Unterteilung in Abbildung 3.3 anhand der Koordinaten der Stützstellen gewählt. Für eine Unterteilung in 2 Prozesse, werden die Stützstellen aufgeteilt in die zwei Bereiche $[0, \frac{1}{2}]$ und $]\frac{1}{2}, 1]$. Die Punkte auf der oberen Grenze eines Prozesses sind damit inklusive und die auf der unteren Grenze exklusive.

In der linken Grafik der Abbildung 3.3 ist ein Gitter mit Level $l = (3, 3)$ und einer Aufteilung in je 2 Prozesse pro Dimension dargestellt. Auf der rechten Seite ist ein Gitter mit $l = (4, 2)$ und keinen Randwerten in Dimension 2 zu sehen, wobei die Daten auf 3 Prozesse bezüglich Dimension 1 aufgeteilt wurden. Eine variable Aufteilung, in Abhängigkeit der Dimensionen, macht es somit möglich die Prozesse gleichmäßig auszulasten.

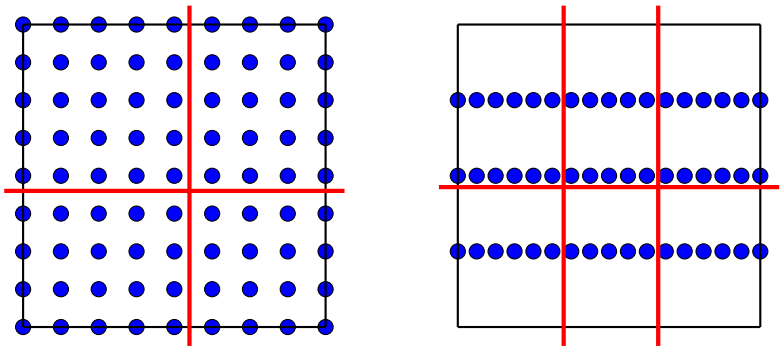

Abbildung 3.3: Links: Volles Gitter mit $l = (3, 3)$ mit Randwerten und einer Unterteilung in $(2, 2)$ Prozesse; Rechts: Volles Gitter mit $l = (4, 2)$ ohne Randwerten in Dimension 2 und einer Unterteilung in $(3, 2)$ Prozesse.

Im Allgemeinen wird eine variable Aufteilung der Punkte in einen Prozess in Abhängigkeit einer Dimension unterstützt, so dass für jede Dimension die unter und ober Grenzen der einzelnen Prozesse festgelegt werden kann.

Die Datenstruktur verändert sich insofern, dass der Array nicht nach einer bestimmten Anzahl an Feldern abgeschnitten wird, sondern dass die Daten in Blöcke bezüglich der Koordinaten der Stützstellen aufgeteilt werden. Daraus folgt, dass für jede Dimension, bei der eine Aufteilung auf mehr als 1 Prozess gewählt wurde, die Daten für jeden Pol in dieser Dimension verteilt vorliegen und abhängig von der Aufteilung Kommunikation zwischen den Prozessen notwendig ist.

3.4.2 Kommunikationsmedium

Für die Kommunikation der verteilten Prozesse wird das „Message Passing Interface“ (MPI) verwendet [6]. Dieses bietet Möglichkeit für den effizienten Austausch von Daten auf verschiedenen Rechnerarchitekturen. Dabei erhält jeder Prozess beim Start eine eindeutige ID (Rang), durch welche er von den anderen Prozessen über das Interface angesprochen werden kann. MPI enthält viele Funktionalitäten bezüglich des Verteilens und Sammelns von Daten auf verschiedenen Topologien von Prozessen.

Hierbei sind für diese Arbeit zwei Funktionalitäten hinsichtlich der zuvor besprochenen Datenstruktur und den Datenaustausch relevant.

Zum einen bietet MPI nicht blockierendes Senden und Empfangen von Daten zwischen Prozessen an. Dies ermöglicht es parallel Daten mit allen Nachbarn auszutauschen. Insbesondere bei einer Aufteilung der Daten in mehr als 2 Prozesse in eine Dimension, ist es von Bedeutung Daten gleichzeitig auszutauschen, um den Datenaustausch nicht durch gegenseitiges Warten zu verlangsamen.

Zum anderen bietet MPI verschiedene Prozess-Topologien an. Diese ermöglichen es den Zugriff auf Prozesse zu erleichtern. Insbesondere ist hier einer kartesische Ordnung der Prozesse, welche in Abbildung 3.4 dargestellt ist, von Bedeutung. Das Gitter lässt sich mit beliebiger Anzahl an Dimensionen und Prozessen in jede Dimension erstellen.

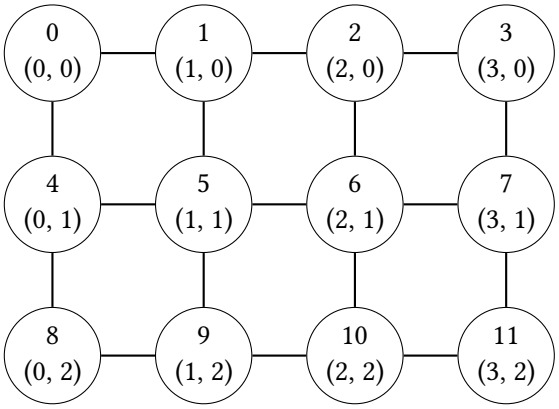

Abbildung 3.4: Kartesische Topologie von Prozessen durch MPI. Rang (oben) und Koordinaten der Prozesse (unten) in den Knoten, basiert auf [**?**]

Durch diese Topologie ermöglicht MPI weitere Funktionalitäten, wie Koordinaten auf den Rang abzubilden und umgekehrt. Des weiteren lassen sich die Nachbarschaftsbeziehungen zwischen den Prozessen abhängig von der Dimension abfragen. Diese sind in Abbildung 3.4 durch schwarze Verbindungen zwischen den Prozessen dargestellt. Zum Beispiel gibt MPI für eine Anfrage nach den Nachbarn in Dimension 1 des Prozesses mit dem Rang 5 die Prozessränge 1 und 9 zurück.

Es sei noch angemerkt, dass Supercomputer-Systeme, wie zum Beispiel die Testarchitektur Hermit, welche in Kapitel 4 beschrieben wird, für die Kommunikation mit MPI optimiert sind.

3.4.3 Hierarchisierung

Nachdem die Aufteilung der Daten im Unterabschnitt 3.4.1 erläutert wurde, befasst sich der folgende Abschnitt mit den Veränderungen bezüglich der verteilten Hierarchisierung.

In Abbildung 3.5 ist die Baumstruktur für ein eindimensionales Gitter mit Level $l = 3$ dargestellt. Hierbei wurden die Punkte auf die Prozesse 0 und 1 aufgeteilt, wodurch hierarchische Abhängigkeiten zwischen Punkten über Prozessgrenzen hinausgehen. Beispielsweise lässt sich der Punkt 5 in Prozess 1 nicht ohne weiteres hierarchisieren, da sein linker hierarchischer Vorgänger sich auf Prozess 0 befindet.

Im folgenden gilt es demnach vorerst für jeden Prozess diejenigen Punkte zu ermitteln, welche der jeweilige Prozess benötigt und welche die Nachbarprozesse benötigen. Daraufhin werden die ermittelten Punkte ausgetauscht.

Weiterhin wird die Hierarchisierung auf die geänderte Datenstruktur angepasst, da, im Vergleich zum Algorithmus 3.1, nur über lokal vorhandene Punkte iteriert wird.

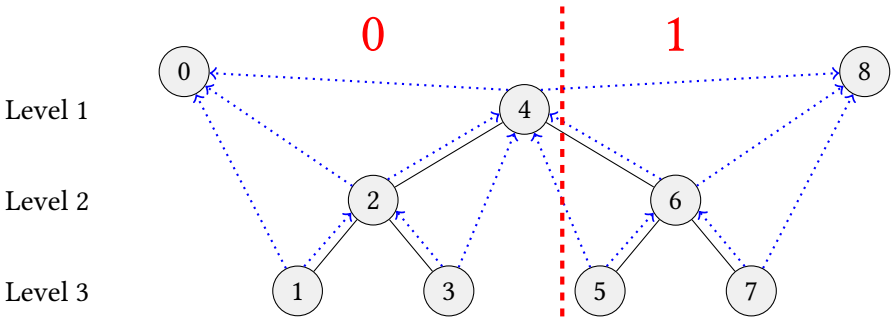

Abbildung 3.5: Baumdarstellung eines eindimensionalen Gitters mit $l = 3$ und Randwerten, wobei das Gitter auf 2 Prozesse aufgeteilt ist.

Datenaustausch

Der Datenaustausch besteht im wesentlichen aus drei Schritten.

1. Berechnung der zu sendenden und der benötigten Punkte.
2. Sammeln der zusendenden Werte für alle Nachbarprozesse bezüglich einer Dimension.
3. Austausch der Daten mittels MPI.

Die Berechnung der Punkte setzt voraus, dass jeder Prozess weiß, über welchen lokalen Bereich des globalen Gitters die einzelnen Prozesse verfügen. Dabei hat jede Dimension eine festen Unterteilung in Prozessen, wodurch jeder Pol bezüglich einer Dimension die gleiche eindimensionale Struktur aufweist. Die Berechnung der Punkte lässt sich für jede Dimension zu einem eindimensionalen Modell, wie in Abbildung 3.5, vereinfachen.

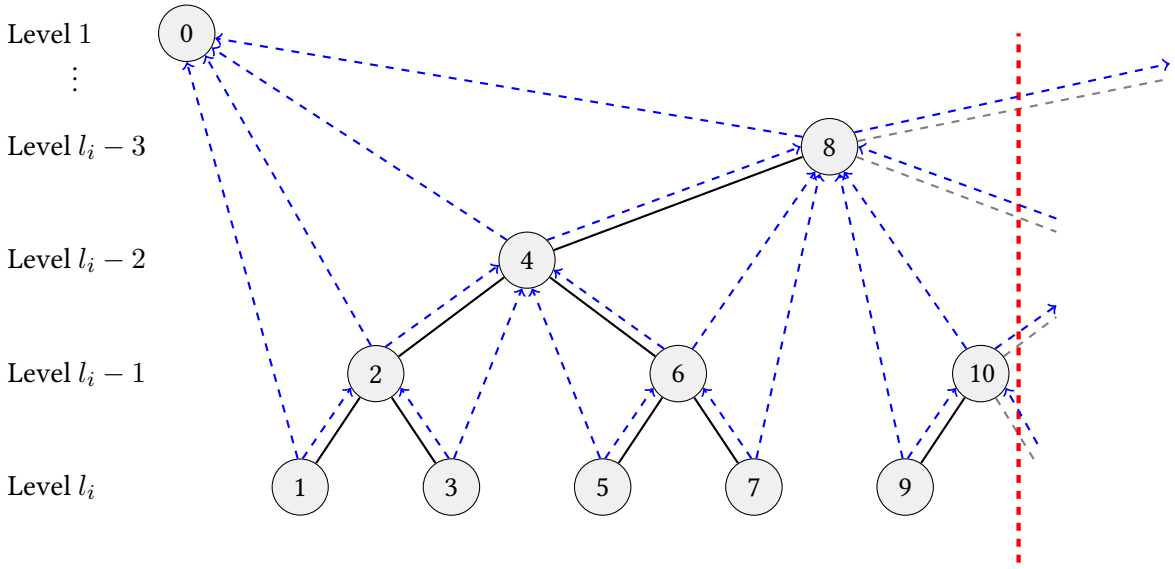

Abbildung 3.6: Linker Teilbaum eines Gitters mit Randwerten und $l_i \geq 5$ und den hierarchischen Abhängigkeiten (blau) der Punkte.

Dieser Feststellung lässt sich durch die Abbildung 3.3 veranschaulichen. Betrachtet man hier die horizontale Unterteilung des Gitters, so ist ersichtlich, dass die oberen Prozesse jeweils 5 Punkte (4 Gitterpunkte + 1 Randpunkt) in der senkrechten Achse beinhalte. Die unteren Prozesse beinhaltet beide 4 Punkte (3 Gitterpunkte + 1 Randpunkt) in der senkrechten Achse.

Daraus lässt sich die Erkenntnis gewinnen, dass die Berechnung über die eindimensionale Struktur jeder Dimension erfolgt und diese anschließend auf die lokale Arraystrukur umgerechnet wird.

Für die Prozesse ist daher die Berechnung des ersten und letzten lokalen Punktes bezüglich der eindimensionalen Struktur wichtig. Diese lässt sich für Dimension i berechnen durch

$$(3.3)\quad (\text{Erster Punkt})_i := \left\lceil \frac{(2^{l_i} - 1) \cdot (\text{Koordinaten des Prozesses})_i}{(\text{Anzahl der Prozesse})_i} \right\rceil + 1$$

wobei die Koordinate bezüglich einer Dimension i eines Prozesse anhand der Topologie, welche in Unterabschnitt 3.4.2 dargestellt wurde, von MPI ausgelesen werden kann. Dementsprechend lässt sich der letzte Punkt durch

$$(3.4)\quad (\text{Letzter Punkt})_i := \left\lceil \frac{(2^{l_i} - 1) \cdot ((\text{Koordinaten des Prozesses})_i + 1)}{(\text{Anzahl der Prozesse})_i} \right\rceil$$

berechnen.

Im folgenden wird aus Gründen der Übersichtlichkeit vorerst die Berechnung der benötigten Punkte jedes Prozesses dargestellt. Dieser wird daraufhin erweitert, um auch die Punkte, welche an die Nachbarn geschickt werden müssen, zu berechnen.

Algorithmus 3.3 Berechnung der auszutauschenden Punkte anhand der eindimensionalen Struktur, wie in Abbildung 3.6

```
1:  function berechnePunkte(i)
2:      punkt := (Erster Punkt)_i
3:      while punkt <= (Letzter Punkt)_i do
4:          if linker Vorgänger von punkt außerhalb then
5:              merke linken Vorgänger
6:          end if
7:          if rechter Vorgänger von punkt außerhalb then
8:              merke rechten Vorgänger
9:              punkt := nächster lokaler Punkt mit Level = min(l_i, Level(punkt) + 1)
10:         else
11:             punkt := rechter Vorgänger von punkt
12:         end if
13:     end while
14: end function
```

Die eigentliche Berechnung der Punkte lässt sich naiv auf einen einfachen Algorithmus übertragen. Dabei wird jeder Punkt betrachtet und jeweils überprüft, ob dessen hierarchische Vorgänger in einem Nachbarprozess oder lokal vorhanden sind. Dieser lässt sich jedoch noch verkürzen.

Die Abbildung 3.6 zeigt den linken Teilbaum eines Gitters mit beliebigem Level $l \geq 5$. Hierbei wurden die hierarchischen Abhängigkeiten in blau dargestellt. Betrachtet man nun die Abhängigkeiten des Punktes 3, so lässt sich feststellen, dass dieser bei Betrachtung der Punkte 2 und 4 nicht notwendigerweise beachtet werden muss. Dies lässt sich ebenso für die Punkte 5, 6, 7 und 9 feststellen. Für den Punkt 10 fehlt der rechte Vorgänger, so dass dieser nicht übersprungen werden kann.

Der optimierte Algorithmus zur Berechnung der nötigen Punkte ist durch den Pseudocode in Algorithmus 3.3 dargestellt.

Dieser Algorithmus beschreibt einen Pfad über eine Teilmenge der Punkte eines Prozesses, wobei an jedem Punkt, beginnend beim kleinsten lokal vorhandenen Punkt, die hierarchischen Vorgänger auf lokale Existenz überprüft werden. Der nächste Punkt ist entweder der rechter Vorgänger, falls dieser lokal vorhanden ist oder der nächste lokale Punkt mit Level $l_{next} > l_{x_i}$ und l_{next} kleinstmöglich. Dabei wird beginnend beim Level des Punktes bis zum Level l_i überprüft, ob der nächste Punkt dieses Levels innerhalb des lokalen Bereichs des Prozesses liegt, wobei der nächste Punkt auf Level l_i gewählt wird, wenn dieser innerhalb oder schon außerhalb des lokalen Bereichs liegt. Durch diese Wahl terminiert die „while“-Schleife in Zeile 3.

Der Algorithmus 3.3 durchläuft die Punkte des Teilbaums in Abbildung 3.6 beginnend bei 1 über 2, 4, 8 zu 10. Hierbei werden für jeden Punkt die entsprechenden hierarchischen Vorgänger betrachtet und gegebenenfalls in einer Liste gespeichert. Beispielsweise für den Punkt 8, ist der rechte Vorgänger nicht vorhanden, so dass der Punkt 24 hinzugefügt wird. Hierbei basiert die Berechnung des Vorgängers auf der Definition 2.15 aus Kapitel 2.

Algorithmus 3.4 Überprüfung aller hierarchischen Nachfolger eines Punktes

```
1:  function checkNachfolger(punkt)
2:      for level := Level(punkt) − 1 to l_i do
3:          if punkt + 2^(l_i − level) außerhalb then                    // rechte Nachfolger
4:              speichere punkt für entsprechenden Nachbarprozess
5:          end if
6:          if punkt − 2^(l_i − level) außerhalb then                    // linke Nachfolger
7:              speichere punkt für entsprechenden Nachbarprozess
8:          end if
9:      end for
10: end function
```

Die Berechnung der Punkte, welche an die Nachbarprozesse geschickt werden, lassen sich in den Algorithmus 3.3 einbinden. Hierbei werden für jeden Punkt nicht nur dessen Vorgänger, sondern auch dessen hierarchischen Nachfolger betrachtet. Die Iteration über die Nachfolger ist in Algorithmus 3.4 dargestellt. Dabei werden beginnend beim Nachfolger mit dem kleinsten Level alle weiteren Nachfolger bis Level l_i betrachtet und gegebenenfalls gespeichert.

Mit Hilfe dieser Algorithmen werden die auszutauschenden Punkte in jedem Prozess berechnet. Diese eindimensionalen Punkte lassen sich auf Positionen bezüglich des lokalen Arrays umrechnen, welche durch MPI ausgetauscht werden.

Hierarchisierung

Zur Visualisierung der Datenstruktur nach dem Austausch der Punkte, stellt Abbildung 3.7 ein Gitter mit $l = 3$, Randwerten und einer Aufteilung in 2 Prozesse dar. Hierbei sind die, mit Hilfe der Algorithmen des vorherigen Abschnitt,s ausgetauschten Punkte durch rote Kreise dargestellt. Jeder Prozess verfügt somit über die benötigten Punkte und kann die lokale Hierarchisierung ausführen.

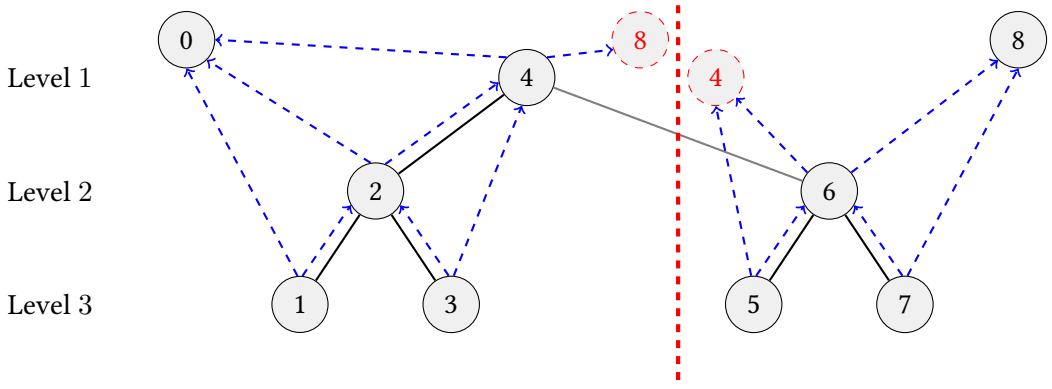

Abbildung 3.7: Aufteilung eines Gitters mit $l = 3$ in 2 Prozesse. Ausgetauschten Punkte 4 und 8 in rot dargestellt.

Dabei lässt sich der Algorithmus der Hierarchisierung zu Beginn dieses Kapitels durch leichte Änderungen anpassen. Zum einen werden für jede Dimension die entsprechenden Punkte ausgetauscht und zum anderen wird nur über die lokal vorhandenen x_i iteriert. Der geänderte Ablauf der Hierarchisierung ist zusammenfassend in Algorithmus 3.5 dargestellt.

Algorithmus 3.5 Erweiterung des Basis Hierarchisierungs-Algorithmus 3.1

```
1:  function VERTEILTEHIERARCHISIERUNG
2:      for dd ← 1 to d do
3:          berechnePunkte(dd)                                   // Datenaustausch
4:          for all lokale 1-dim Pole P in Richtung dd do
5:              for l ← l_dd to { 2, (keine Randwerte)_dd ; 1, (Randwerte)_dd } do
6:                  for all (Erster Punkt)_dd ≤ x_i ≤ (Letzter Punkt)_dd auf Level l do
7:                      grid[i] = grid[i] − 0.5 · leftPredecessor(i, dd, l)
8:                      grid[i] = grid[i] − 0.5 · rightPredecessor(i, dd, l)
9:                  end for
10:             end for
11:         end for
12:     end for
13: end function
```

3.4.4 Dehierarchisierung

Der Algorithmus in Abschnitt 3.3 lässt sich nicht ohne weiteres durch Hinzufügen eines Datenaustausches, ähnlich zu dem der Hierarchisierung, erweitern.

Die Dehierarchisierung ist das Umkehren der Hierarchisierung, der Ablauf ist somit genau entgegengesetzt. Der Knackpunkt bezüglich des Datenaustausches ergibt sich aus dem Durchlauf der Level, beginnend bei Level 1 bzw. 2 bis l_i. Hierbei setzen die x_i auf höheren Level die schon dehierarchisierten Werte auf den niedrigeren Level voraus.

Abbildung 3.8 zeigt die eindimensionale Struktur eines Gitters mit $l = 3$ und Randwerten. Beim Dehierarchisieren wird hier mit dem Punkt 4 begonnen. Dieser benötigt die Punkte 0 und 8 des Nachbarprozesses 0 beziehungsweise 2. Bis hier ist ein Austausch äquivalent zum Austausch der Hierarchisierung möglich, jedoch benötigen die Punkte 2 und 6 jeweils den dehierarchisierten Wert des Punktes 4. Ebenso ist für die Punkte 3 und 5 keine Dehierarchisierung ohne die dehierarchisierten Werte der Punkte 2 und 6 möglich. Ein Austausch äquivalent zu dem der Hierarchisierung ist demnach nicht möglich.

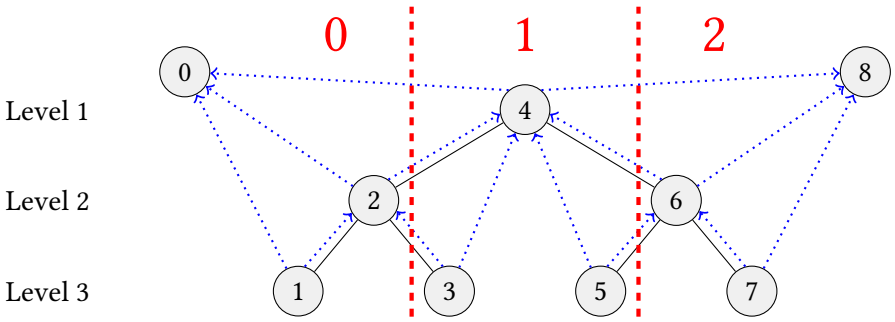

Abbildung 3.8: Baumdarstellung eines eindimensionalen Gitters mit $l = 3$ und Randwerten, wobei das Gitter in 3 Prozesse aufgeteilt ist.

Naiv

Die naive Lösung besteht darin, nach der Dehierarchisierung jedes Levels nur die geänderten Funktionswerte bezüglich diesem auszutauschen. Hierbei werden jeweils nur die, für die Dehierarchisierung des folgenden Levels, relevanten Punkte ausgetauscht.

Verglichen mit dem Austausch der Hierarchisierung für jede Dimension, ist hier ein Austausch für jede Dimension und jedes Level notwendig, wodurch sich die Anzahl der Nachrichten erhöht bei gleichzeitiger Verringerung der Größe der Nachrichten. Dies hat eine häufige Synchronisierung der Prozesse zur Folge.

Der Ablauf des naiven Algorithmus auf das, in Abbildung 3.8 dargestellte, Gitter ergibt sich zu:

1. Austausch der Randwerte, falls diese in Dimension i vorhanden.
2. Lokale Dehierarchisierung der x_i auf Level 1 (Punkt 4)
3. Austausch der x_i auf Level 1 (Punkt 4 an die Prozesse 0 und 2)
4. Lokale Dehierarchisierung der x_i auf Level 2 (Punkt 2 und 6)
5. Austausch der x_i auf Level 2 (Punkt 2 und 6 an Prozess 1)
6. Lokale Dehierarchisierung der x_i auf Level 3 (Punkt 1, 3, 5 und 7)

Das Austauschen, von nur wenigen Punkten bei jedem Level, kann, durch gegenseitiges Warten, zum Leerlauf von einigen Prozessen führen, wodurch Rechenzeit verschenkt wird und eine unausgeglichene Auslastung entsteht.

Optimiert

Das Ziel des optimierten Algorithmus ist, das Austauschen äquivalent zu dem der Hierarchisierung, einmal pro Dimension, durchzuführen. Dabei wird das häufigere Synchronisieren der Prozesse durch zusätzliche Rechenarbeit ausgeglichen, wobei sich die Abhängigkeiten zwischen den x_i nicht beeinflussen lassen.

In Abbildung 3.8 bestehen für den Punkt 3 nicht lokal vorhandene hierarchische Abhängigkeiten. Bei der Betrachtung der Abhängigkeit des linken Vorgängers (Punkt 2) lässt sich erkennen, dass dieser wiederum den linken Randwert (Punkt 0) und den Punkt 4 benötigt. Hierbei fällt auf, dass der linke Randwert schon für das Dehierarchisieren des Punktes 4 benötigt wird und somit lokal auf Prozess 1 vorhanden ist. Ein erneutes Austauschen nach Level 1 lässt sich an diesem Beispiel vermeiden, durch das Senden der Punkte 0 und 2 zu Beginn. Der Punkt 2 lässt sich auf Level 2 lokal in Prozess 1 Dehierarchisieren, wodurch der Punkt 3 ohne erneuten Austausch dehierarchisiert werden kann.

Nach diesem Schema erhält Prozess 0 sowohl den Punkt 4 von Prozess 1 als auch den Punkt 8 von Prozess 2, so dass dieser den Punkt 4 selbständig dehierarchisieren kann. Äquivalent dazu läuft die Dehierarchisierung gespiegelt in Prozess 2 ab.

Die Komplexität des Austausches wird durch das Austauschen der gesamten Abhängigkeitsstruktur erhöht, wobei im Gegenzug Leerlaufzeiten reduziert oder komplett vermieden werden. Abbildung 3.9 stellt die Struktur der Daten nach dem Austausch für das Beispiel eines Gitters mit $l = 3$ und einer Aufteilung in 3 Prozesse dar.

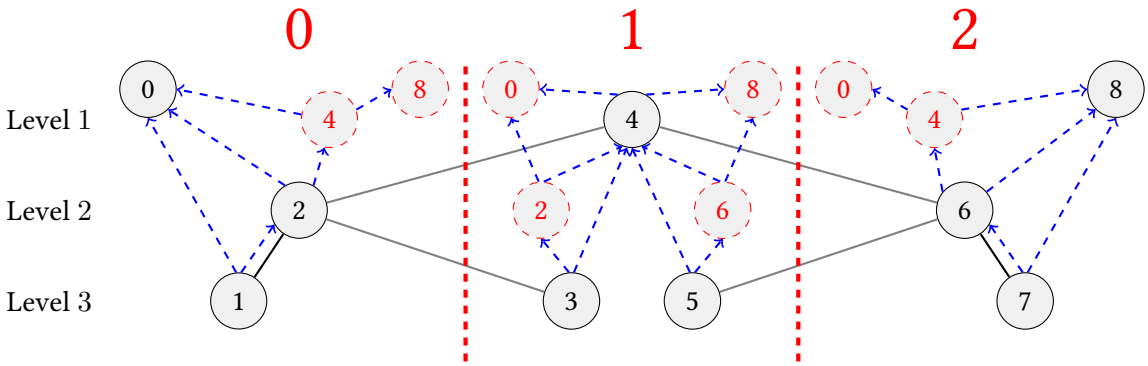

Abbildung 3.9: Baumdarstellung eines eindimensionalen Gitters, nach dem Datenaustausch, mit $l = 3$ und Randwerten, wobei das Gitter in 3 Prozesse aufgeteilt ist. Dabei sind die ausgetauschten Punkte, analog zu Abbildung 3.7 in rot dargestellt.

Der Algorithmus bezüglich des Berechnens der auszutauschenden Punkte der Hierarchisierung wird für die Dehierarchisierung erweitert. Hierbei wird die Reihenfolge der Betrachtung der Punkte nicht verändert, wohingegen die betrachtenden Abhängigkeiten erweitert werden.

Benötigten Punkte: Bei der Berechnung der benötigten Punkte werden, wie in Abbildung 3.9 dargestellt, zusätzlich alle hierarchischen Vorgänger betrachtet. Für einen lokalen Punkt x_i mit einem nicht lokalen hierarchischen Vorgänger $x_{Vorgänger}$ ergeben sich bezüglich der hierarchischen Vorgängern von $x_{Vorgänger}$ mehrere Möglichkeiten:

1. Sowie $x_{Vorgänger}$ als auch ein hierarchischer Vorgänger dessen befinden sich im gleichen Prozess. Der Prozess sendet somit beide bzw. alle weiteren. Ein Beispiel für diese Beziehung sind die Punkte 2 und 0 bezüglich des Punktes $x_i = 3$ in Abbildung 3.9.
2. Ein Vorgänger von $x_{Vorgänger}$ befinden sich weder in dem gleichen Prozess von x_i noch von $x_{Vorgänger}$.
3. Ein Vorgänger von $x_{Vorgänger}$ befindet sich im gleichen Prozess wie x_i. Hierbei entstehen bezüglich dieses Punktes keine weiteren zu sendenden Daten. Ein Beispiel für diese Beziehungen sind die Punkte 2 und 4 bezüglich des Punktes $x_i = 3$. Dabei ist der Vorgänger von Punkt 2, Punkt 4, wiederum in Prozess 1 und somit im gleichen Prozess wie x_i.

Aus diesen Möglichkeiten folgt, dass die Berechnung der Vorgänger rekursiv berechnet werden, da diese in verschiedenen Prozessen sein können.

Zusendende Punkte: Bei der Betrachtung der ausgetauschten Punkte in Abbildung 3.9 ist ersichtlich, dass die Punkte auf Level 1 an alle Nachbarn verteilt wurden. Für alle weiteren Punkte werden alle hierarchischen Nachfolger betrachtet. Diese Abhängigkeit beschreibt den Teilbaum mit x_i als Wurzel und Level l_{x_i} bis Level l_j in Dimension j. Formal lässt sich der Teilbaum darstellen durch

$$\begin{aligned} (\text{kleinster Nachfolger})_i &:= i - (2^{l_j - l_{x_i}} - 1) \\ (\text{größter Nachfolger})_i &:= i + (2^{l_j - l_{x_i}} - 1) \end{aligned} \tag{3.5}$$

Der Punkt x_i wird daraufhin an alle Prozesse verteilt, welche einen Teil der Punkte zwischen $(\text{kleinster Nachfolger})_i$ und $(\text{größter Nachfolger})_i$ beinhalten.

Dehierarchisierung

Das erweiterte Austauschen von Abhängigkeiten bis auf Level 1 erzeugt einen komplexeren Ablauf der Dehierarchisierung. Hierbei werden für eine Dimension i die Daten ausgetauscht und daraufhin sowohl die lokalen als auch die ausgetauschten Werte dehierarchisiert.

Im Vergleich zur Hierarchisierung werden dabei gegebenenfalls auch ein Teil der ausgetauschten Werte dehierarchisiert. Das Austauschen nach jedem Level des naiven Algorithmus wird somit ersetzt durch das Dehierarchisieren der ausgetauschten Werte auf dem entsprechenden Level. Analog zur verteilten Hierarchisierung lässt sich der Ablauf wie in Algorithmus 3.6 darstellen.

Algorithmus 3.6 Verteilte Dehierarchisierung mit optimiertem Ablauf

```
1:  function VERTEILTEDEHIERARCHISIERUNG
2:      for dd ← d to 1 do
3:          berechnePunkteDehierarchisierung(dd)                    // Datenaustausch
4:          for all lokale 1-dim Pole P in Richtung dd do
5:              for l ← { 2, (keine Randwerte)_dd ; 1, (Randwerte)_dd } to l_dd do
6:                  for all (Erster Punkt)_dd ≤ x_i ≤ (Letzter Punkt)_dd auf Level l do
7:                      grid[i] = grid[i] + 0.5 · leftPredecessor(i, dd, l)
8:                      grid[i] = grid[i] + 0.5 · rightPredecessor(i, dd, l)
9:                  end for
10:                 for all ausgetauschte Punkte auf Level l do
11:                 // Dehierarchisierung der ausgetauschten Punkte gespeichert im Array extern[]
12:                     extern[i] = extern[i] + 0.5 · leftPredecessor(i, dd, l)
13:                     extern[i] = extern[i] + 0.5 · rightPredecessor(i, dd, l)
14:                 end for
15:             end for
16:         end for
17:     end for
18: end function
```

3.5 Cache-Effizienz

Der Ablauf der Hierarchisierung und Dehierarchisierung erfolgt Polweise. In Abbildung 3.10 ist die Arraystruktur für ein Gitter mit $l = 3$ dargestellt. Dabei beginnt das eindimensionale Gitter links oben und verläuft zeilenweise bis zum letzten Eintrag rechts unten. Hierbei wurde die Nummerierung anhand der Baumdarstellung der Pole bezüglich Dimension 2 gewählt. Für einen blau umrahmten Pol bezüglich Dimension 2 ist ersichtlich, dass die Werte der Punkte im Array lokal beieinander liegen.

Das Bearbeiten eines Punktes führt in der Hardware dazu, dass die Werte in Cache-Lines aus dem Hauptspeicher in den Cache geladen werden [7, S. 334ff]. Beim Hierarchisieren von Punkt 1 des ersten Pols wird demnach nicht nur genau dessen Wert geladen, sondern die gesamte Cache-Line. Daraus resultiert, dass für das Hierarchisieren von Punkt 1 der rot umrahmte Bereich in Abbildung 3.10 in den Cache geladen wird. Dabei ist der Punkt 0 der linke Vorgänger und Punkt 2 der rechte Vorgänger.

Bei dem, in Abschnitt 3.4.3 vorgestellten, Algorithmus findet eine polweise Abarbeitung statt. Für Dimensionen mit hohem Level und daraus resultierenden großen Polen werden die schon in die Cache geladenen Werte der Nachbarpole zu Beginn des Pols durch Werte am Ende des Pols ersetzt, bevor dieses benutzt wurden. Daraus folgt, dass die Werte mehrere Male geladen werden, ohne dass sie verwendet werden.

Für bessere Ausnutzung der Lokalität werden die Punkte nicht mehr nach Polen abgearbeitet, sondern in Blöcken, welche jeweils aus mehreren Polen bestehen. Anhand der Abbildung 3.10 werden 4 Pole gleichzeitig hierarchisiert bzw. dehierarchisiert.

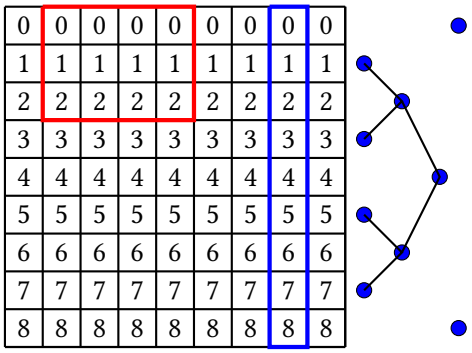

Abbildung 3.10: Array grid[] eines Gitters von links oben nach rechts unten mit $l = 3$, wobei sich die Nummerierung an den Punktindizes der eindimensionalen Baumstruktur (rechts) der Pole bezüglich Dimension 2 orientiert. In blau ist ein Pol bezüglich der Dimension 2 hervorgehoben und in rot die in die Cache geladenen Punkte für das Hierarchisieren von Punkt 1.

3.6 SG++

SG++ ist eine Toolbox, welche Gegenstand der Dissertation „Spatially Adaptive Sparse Grids for Higher-Dimensional Problems“ von Dirk Pflüger war [3, 4]. Diese Toolbox ermöglicht das Bearbeiten verschiedenster höherdimensionalen Probleme mit adaptiven dünnen Gittern und umfasst Interpolation mit verschiedenen Basen, Approximationen und das Lösen von partiellen Differentialgleichungen. Die Algorithmen in SG++ sind zum Großteil in C++ programmiert, wodurch die vorgestellten Algorithmen ebenfalls in C++ umgesetzt wurden und in SG++ eingebunden wurden.

4 Evaluation

In diesem Kapitel werden Messungen, bezüglich der in Kapitel 3 vorgestellten Algorithmen für die Hierarchisierung und Dehierarchisierung, vorgestellt und bewertet. Dabei werden jeweils die Hierarchisierung und die Varianten der Dehierarchisierung dargestellt.

Die angegebenen Zeiten in diesem Kapitel sind jeweils die durchschnittlichen Zeiten für die Hierarchisierung und Dehierarchisierung. Dabei wurde weiter unterschieden zwischen durchschnittlicher Berechnungszeit und Zeit für den Datenaustausch.

4.1 Architektur der Testsysteme

Die Messungen in diesem Kapitel wurden auf den folgenden zwei Systemen ausgeführt.

Hermit

Hermit ist ein Cray XE6 Supercomputer [8]. Dieser verfügt über 3552 Knoten mit je 2 Prozessoren, welche aus 16 Kernen mit einer Taktrate von je 2.3 GHz bestehen. Diese verfügen über einen L1 Cache mit 16 KB, einen L2 Cache mit 2 MB und einen L3 Cache mit 6 MB. Zusätzlich verfügt jeder Knoten über 32 GB bzw. 64 GB Hauptspeicher.

Die Knoten von Hermit sind in einem 3D-Torus-Netzwerk aus Cray Germini Netzwerkcontrollern verbunden [9].

Kepler

Kepler verfügt über ein Prozessor von Intel, welcher aus 2 CPUs mit je 8 Kernen besteht. Dabei verfügen diese auf gemeinsame 20 MB L3 Cache und pro Kern jeweils über 256 KB L2 und 32 KB L1 Cache. Die Messungen auf diesem System beinhalten nur „single core“ Messungen.

4.2 Messungen

4.2.1 Blockgröße

In den Messungen in Abbildung 4.1 wurde die benötigte Zeit in Abhängigkeit der Blockgröße dargestellt. Dabei wurden die Zeit für das Hierarchisieren und Dehierarchisieren auf einem Prozess, bei einem Gitter mit Dimension $d = 5$ und einer Gittergröße von ca. 484 Mio. Punkten, gemessen. Die Messungen wurden auf Kepler ausgeführt.

Hierbei spielte das Abarbeiten von mehreren Polen gleichzeitig nicht nur bezüglich der Cacheausnutzung eine Rolle. Durch das parallele Abarbeiten werden Berechnungen, wie das Berechnen der Vorgänger, des nächsten Punktes des gleichen Levels und if-Abfragen bezüglich vorhandener oder nicht vorhandener Randwerten, eingespart.

In Abbildung 4.1 ist deutlich zu erkennen, dass sich das Abarbeiten von mehreren Polen positiv auf die Laufzeit auswirkt.

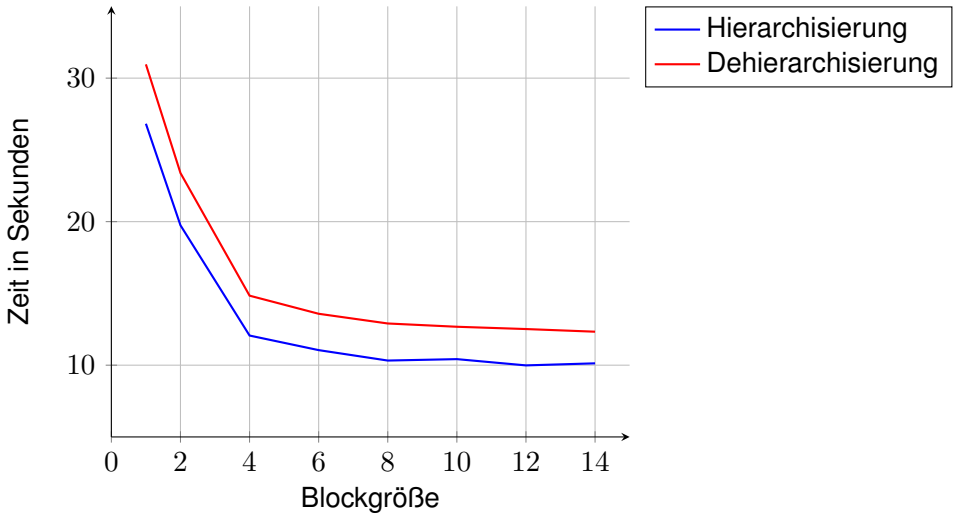

Abbildung 4.1: Unterschiedliche Blockgrößen bei der Bearbeitung eines Gitters mit insgesamt ca. 484 Mio. Punkten in 5 Dimensionen auf einem Prozess.

4.2.2 Starke Skalierung

Für das Messen der starken Skalierbarkeit wurde die Anzahl der verwendeten Prozessoren von Messung zu Messung verdoppelt, wobei die Gittergröße unverändert blieb. Hierbei wurde ein Gitter mit 5 Dimensionen und insgesamt ca. 2 Mrd. Punkten gewählt. Die Messung der Zeiten wurden auf Hermit durchgeführt.

Die obere Grafik der Abbildung 4.2 zeigt die benötigte Zeit in logarithmischer Skala für die Hierarchisierung. Dabei wird im folgenden die Gesamtzeit in Blau, die Berechnungszeit in Grün und die Zeit für den Datenaustausch in Rot dargestellt. Beim Vergleich der Varianten der Dehierarchisierung werden die Messungen für den Austausch pro Dimension gestrichelt, jedoch mit identischer Farbcodierung, veranschaulicht.

In den Grafiken in Abbildung 4.2 ist zu erkennen, dass der Datenaustausch der Dehierarchisierung deutlich mehr Zeit beansprucht als der der Hierarchisierung. Dies ist vermeintlich auf die häufige Synchronisierung der Prozesse auf Grund des Datenaustausches bei jedem Level (Naiv) beziehungsweise auf die höhere Menge an Daten, im Fall des Austausches pro Dimension (Optimiert), zurückzuführen. Die Berechnungszeit halbiert sich beinahe bei jeder Verdopplung der Prozesse.

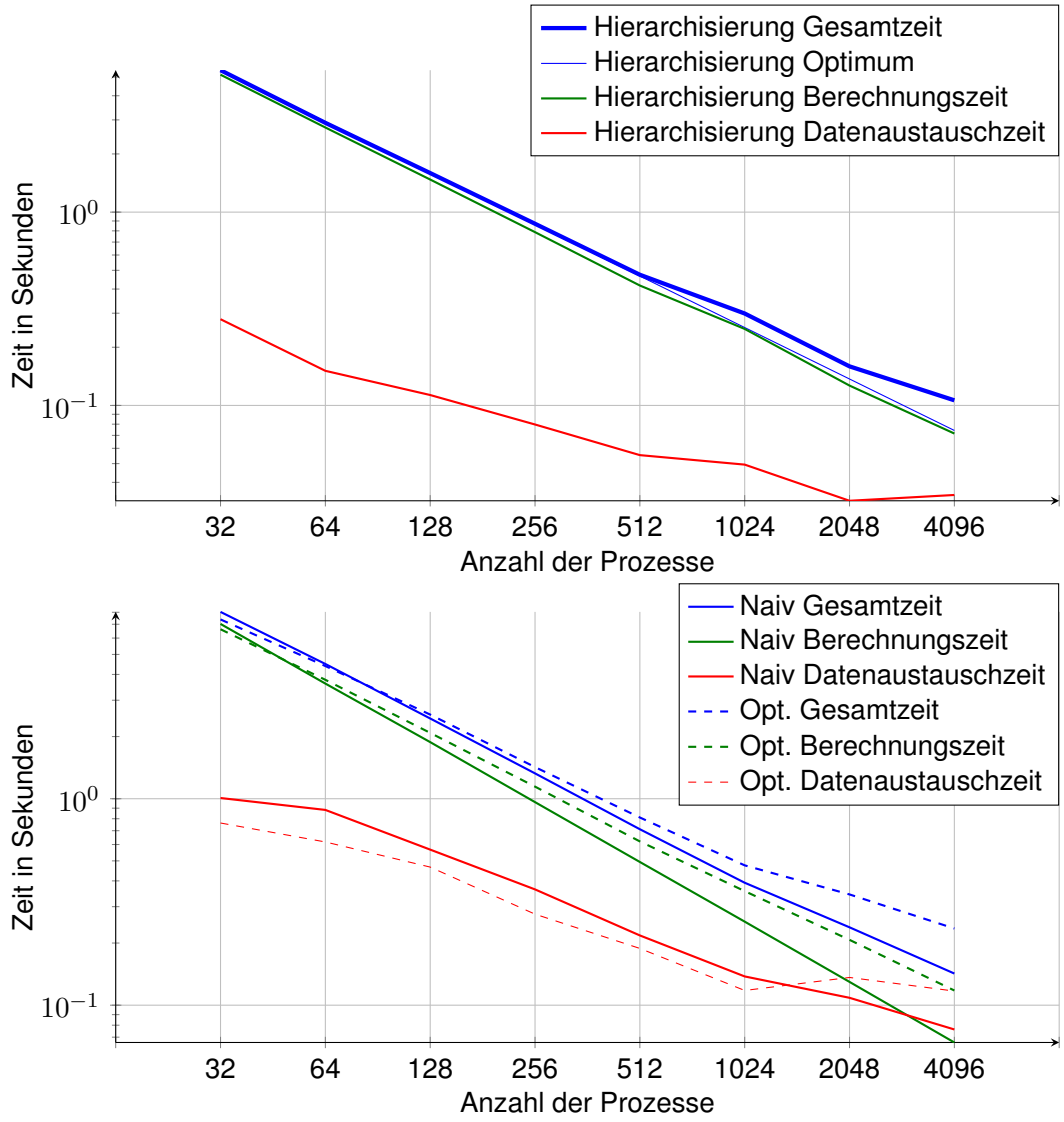

Abbildung 4.2: Oben: Hierarchisierung; Unten: naive (Austausch pro Dimension und Level) und optimierte (Austausch pro Dimension) Dehierarchisierung. Jeweils mit Blockgröße 8 bei konstanter Gittergröße und 5 Dimensionen.

Die untere Grafik in Abbildung 4.2 vergleicht die Varianten der Dehierarchisierung. Zu Beginn ist die Zeit des Datenaustausches im optimierten Fall um ca. 20% geringer als beim Austauschen pro Dimension und Level. Die beinahe konstante Zeit ab 1024 Prozessen, entsteht vermeintlich durch das Verschicken aller Vorgänger.

Für insgesamt 1024 Prozesse bedeutet dies eine Aufteilung der eindimensionalen Baumstruktur mit jeweils 63 Punkten (127 in Dimension 5) in 4 Teile (in jeder Dimension). Die einzelnen Prozesse, bei einer Aufteilung von 63 Punkten auf 4 Prozesse, beinhalten komplette Teilbäume ab Level 2. Bei weiteren Unterteilungen, werden die Baumstrukturen in kleinere Teilbäume unterteilt, so dass die

auszutauschenden Vorgänger auf höheren Level sind und die Anzahl der hierarchischen Vorgänger steigt. Auf Grund dessen wird die Zeit für den Datenaustausch nicht im gleichen Maße wie die Berechnungszeit kleiner.

4.2.3 Schwache Skalierung

Für das Messen der schwachen Skalierbarkeit wurde sowohl die Anzahl an Prozessen als auch auch die Gittergröße von Messung zu Messung verdoppelt. Dabei wurde, beginnend bei einem Gitter mit 5 Dimensionen und insgesamt ca. 1 Mrd. Punkten auf 32 Prozessoren, die Zeiten für die Hierarchisierung und beide Varianten der Dehierarchisierung gemessen, sowie jeweils die Zeiten für die Berechnung und den Datenaustausch.

Dabei ist die Hierarchisierung in der oberen Grafik in Abbildung 4.3 und die Varianten der Dehierarchisierung in der unteren Grafik. Sowohl für die Hierarchisierung als auch für die Varianten der Dehierarchisierung wurden in dünnem (gestrichelten) Blau die optimale Zeit eingetragen, welche konstant bei gleichzeitiger Verdopplung der Prozesse und der Gitterpunkte ist.

Die Hierarchisierung weicht geringfügig vom Optimum ab, wohingegen bei beiden Fällen der Dehierarchisierung eine höhere Abweichung zu erkennen ist. Des Weiteren zeigt die Grafik, dass sich bei mehr Prozessen die Variante mit dem Austausch pro Dimension negativ auf die Berechnungszeit und bei einer vielen Prozessen auch auf die Zeit des Datenaustausches auswirkt.

Dieser Aspekt wurde zuvor bei den Messungen für die starke Skalierbarkeit angesprochen und näher erläutert. Die Wahl einer Variante der Dehierarchisierung ist somit stark abhängig von der Aufteilung der Daten auf die Prozesse.

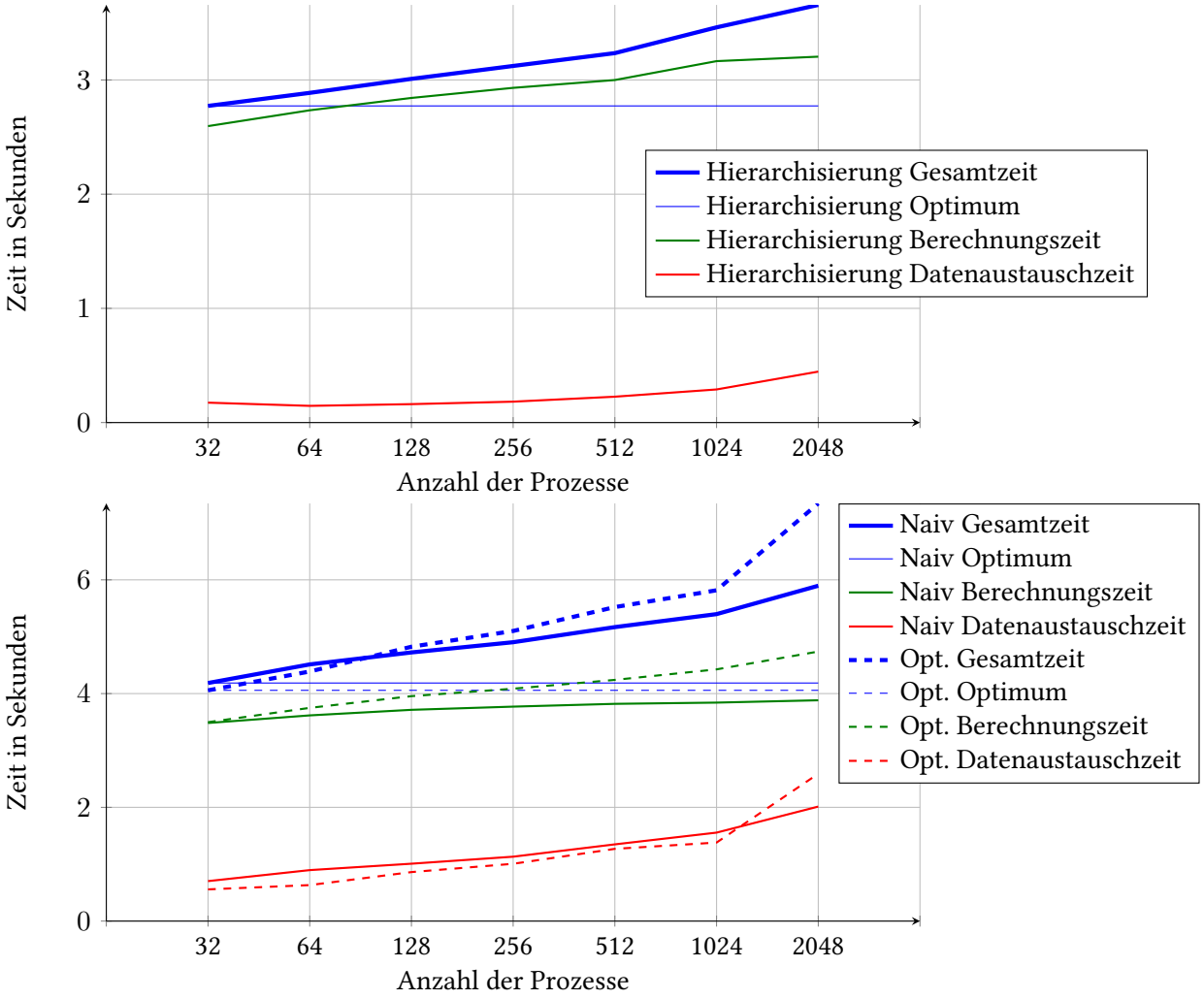

Abbildung 4.3: Oben: Hierarchisierung; Unten: naive (Austausch pro Dimension und Level) und optimierte (Austausch pro Dimension) Dehierarchisierung. Jeweils Verdopplung der Prozesse und Verdopplung der Gitterpunkte. Anzahl der Gitterpunkte ausgehend von ca. 1,160 Mrd. mit Blockgröße 8.

4.2.4 Optimierbarkeit

Im folgenden wird die Effizienz der Algorithmen verglichen und in Bezug zu der theoretisch Obergrenze basierend auf der Speicherbandbreite gesetzt.

Die theoretische Obergrenze für die Hierarchisierung lässt sich durch die Formel

$$\frac{2 \cdot d \cdot N_i \cdot (\textit{Größe des Datentyps})}{\textit{Bandbreite des Speichers}} \tag{4.1}$$

beschreiben [5]. Hierbei wird jeder Punkt des Gitters für jede Dimension einmal betrachtet und dabei geladen, verändert und zurück geschrieben.

Dabei hängt die Bandbreite wesentlich von N_i ab, da diese in Abhängigkeit von den Kapazitäten der L1, L2 und L3 Caches unterschiedlich schnell geladen werden können. Für sehr große N_i und einer Blockgröße von 1 werden die Daten der Cache-Line nicht komplett verwendet, da sie zuvor überschrieben werden.

Für die Messung der oberen Grenze wurde der „Stream Benchmark“ von John D. McCalpin an die Hierarchisierung angepasst [10]. Hierbei wurde für verschiedene Gittergrößen die Anzahl der ausgeführten Berechnungen der Definition 4.2 gemessen.

$$\begin{aligned} a[i] &:= a[i] + \cdot a[i-1] \\ a[i] &:= a[i] + \cdot a[i+1] \end{aligned} \tag{4.2}$$

Die Ergebnisse dieses Benchmarks sind in Abbildung 4.4 dargestellt, wobei die Anzahl der bearbeiteten Gitterpunkte pro Sekunde aufgezeigt ist. Die Messungen wurden auf Kepler durchgeführt, wobei jeweils nur ein Prozess für die Berechnung verwendet wurde.

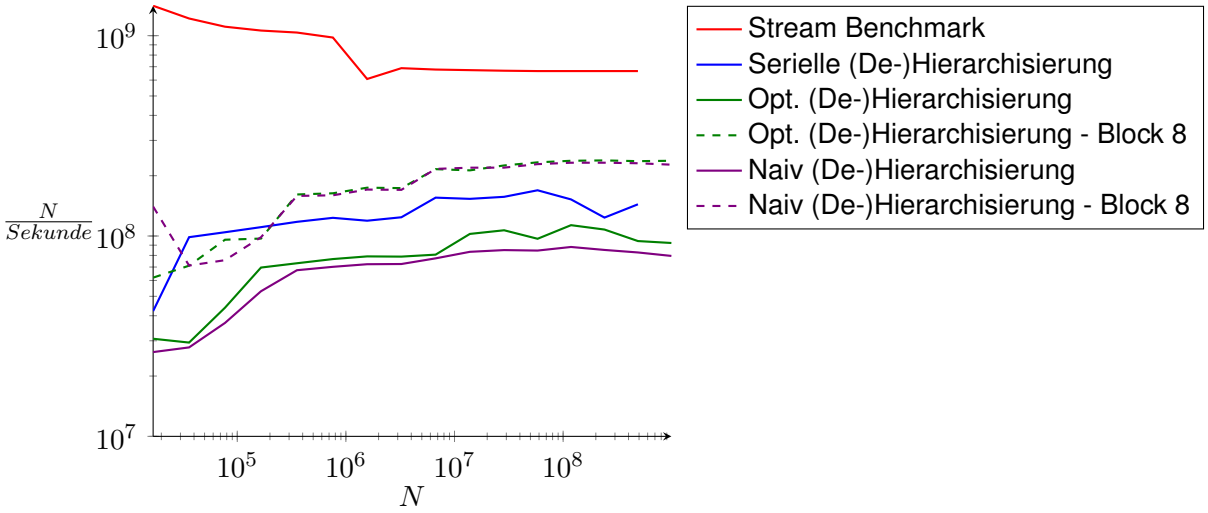

Abbildung 4.4: Effizienz der Algorithmen im Vergleich und im Bezug zur theoretischen Obergrenze.

Die Ergebnisse in Abbildung 4.4 zeigen, dass die Algorithmen nicht optimal sind.

Der Algorithmus für die serielle Hierarchisierung und Dehierarchisierung wird durch die Flexibilität bezüglich Randwerten verlangsamt, da hier viele if-Abfragen notwendig sind und die Startwerte der Pole nicht direkte berechnet werden können.

5 Schluss

5.1 Zusammenfassung

Zu Beginn dieser Arbeit wurden die mathematischen Grundlagen für die Hierarchisierung und Dehierarchisierung beschrieben. Daraufhin folgte in Kapitel 3 der serielle Algorithmus für die Hierarchisierung, welcher um variable Randwerte erweitert wurde. Um auf der veränderten Datenstruktur für verteilte Gitter zu arbeiten, wurden Algorithmen bezüglich des Datenaustausches erläutert. Dabei wurden für die Dehierarchisierung zwei Varianten mit unterschiedlichen Austauschverfahren vorgestellt. Die entwickelten Algorithmen wurden im Kapitel 4 mit Messungen bewertet und verglichen.

5.2 Ausblick

Die Implementierung der Hierarchisierung und Dehierarchisierung verwendet in der seriellen Variante jeweils zwei Methoden. Dabei bestimmt eine die Startwerte der eindimensionalen Pole und die andere arbeitet die eindimensionalen Pole ab. Durch den variablen Rand und das zusätzliche Aufteilen der Pole auf verschiedene Prozesse entsteht Komplexität, welche durch das Verwenden vieler if-Abfragen und mehrerer Hilfsmethoden gelöst wurde. Auch wurden die Werte der Nachbarn in einem eigenen Array gespeichert, wodurch weitere if-Abfragen entstanden sind, um die Position der benötigten Werte zu ermitteln.

Eine mögliche Erweiterung wäre das Verwenden von nur einem Array, so dass die übertragenen Werte an das Ende des Arrays, mit den lokalen Punkten, angehängt werden. Beim Durchlauf des ersten Pols könnten die Offsets der benötigten Vorgänger gespeichert werden, so dass das Abarbeiten der übrigen Pole ohne if-Abfragen auskommt.

Dies führt zu einer weiteren möglichen Erweiterung. Da für jede Dimension klar definiert ist, ob diese Randwerte beinhaltet, könnte es sinnvoll sein die Methoden für das Abarbeiten der Pole aufzuteilen, so dass es für die Bearbeitung der Dimension mit Randwerten andere Methoden als für die ohne Randwerte gibt.

Auch kann es möglich sein, dass für eine Dimension keine Aufteilung in mehrere Prozesse benötigt wurde, so dass die Verwendung der seriellen Bearbeitung für diesen Fall die vorzuziehende Methode wäre.

Literaturverzeichnis

[1] Prof. Dr. rer. nat. habil. Miriam Mehl. Skript zur Vorlesung Grundlagen des wissenschaftlichen Rechnens. 2014. (Zitiert auf Seite 9)

[2] Michael Griebel, Michael Schneider, and Christoph Zenger. A combination technique for the solution of sparse grid problems, 1992. (Zitiert auf Seite 9)

[3] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, August 2010. (Zitiert auf den Seiten 6, 9, 11, 12, 13, 16, 17 und 34)

[4] Sg++. http://www5.in.tum.de/SGpp/releases/index.html. (Zitiert auf den Seiten 9 und 34)

[5] Philipp Hupp. Performance of unidirectional hierarchization for component grids virtually maximized. *Procedia Computer Science*, 29(0):2272 – 2283, 2014. 2014 International Conference on Computational Science. (Zitiert auf den Seiten 7, 19 und 40)

[6] Mpi. http://www.open-mpi.org/. (Zitiert auf Seite 24)

[7] Prof. Dr. Hans-Joachim Wunderlich. Skript zur Vorlesung Rechnerorganisation. 2008. (Zitiert auf Seite 33)

[8] Hermit. http://www.prace-ri.eu/best-practice-guide-cray-xe-xc-html/?lang=en#sec-3-1. (Zitiert auf Seite 35)

[9] Germini. http://www.training.prace-ri.eu/uploads/tx_pracetmo/BestPracticeHermit.pdf. (Zitiert auf Seite 35)

[10] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995. (Zitiert auf Seite 40)

Alle URLs wurden zuletzt am 7. Oktober 2014 geprüft.

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