Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Lastbalancierung durch dynamische Aufgaben-Umverteilung mit der Dünngitter-Kombinationstechnik

Marvin Dostal

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dirk Pflüger
Betreuer/in:	M. Sc. Theresa Pollinger
Beginn am:	9. Juli 2019
Beendet am:	9. Januar 2020

Kurzfassung

In dieser Arbeit wird das DisCoTec-Framework um eine dynamische Aufgaben-Umverteilung ergänzt. Das DisCoTec-Framework bietet ein Grundgerüst für das Arbeiten mit der Dünngitter-Kombinationstechnik. Mithilfe des DisCoTec-Frameworks ist es möglich Aufgaben, wie das Lösen von partiellen Differenzialgleichungen, effizient und parallelisiert durchzuführen. Es ist dabei insbesondere für hochdimensionale Probleme entworfen. Das DisCoTec-Framework behandelt die zu bearbeitenden Aufgaben mit dem Blackbox Prinzip. Bisher bietet das DisCoTec-Framework nur für Aufgaben mit konstanter Laufzeit eine gute Lastverteilung. Das Ziel ist das Ermöglichen des effizienten Berechnen von Aufgaben mit variablem Laufzeitverhalten. Dazu wird eine dynamische Aufgaben-Umverteilung in das DisCoTec-Framework eingeführt, analysiert und bewertet. Die dynamische Aufgaben-Umverteilung ist in der Lage die Effizienz der Berechnung von Aufgaben mit variabler Laufzeit zu erhöhen. Die Laufzeitkosten der dynamischen Aufgaben-Umverteilung werden dabei relativ zu der Berechnungsdauer der Aufgaben für vernachlässigbar empfunden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

DisCoTec-Framework Distributed-Combigrid-Technique-Framework. 15

DUNE Distributed and Unified Numerics Environment. 35

MPI Message Passing Interface. 17

1. Einleitung

Die Computertechnologie ist fest mit dem Bereich der Simulation verbunden. Schon die ersten Computer, wie der ENIAC, wurden für das Berechnen von (ballistischen) Simulationen konzipiert [EMGB45]. Aufgrund des rasanten Fortschritt der Computertechnologie, ist die Computersimulation zu einem populären Werkzeug der wissenschaftlichen Forschung in verschiedensten Bereichen aufgestiegen [GNW10]. Das für diese Arbeit relevante Problem ist das Lösen von hochdimensionalen Problemen, insbesondere hochdimensionale partielle Differenzialgleichungen [Hee18]. Um effiziente mathematische Verfahren benutzen zu können, werden die hochdimensionalen Problemen auf diskreten Räumen durchgeführt. Daraus folgt, dass Computersimulation nur Annäherungen liefern. Das Ziel ist nun mithilfe einer intelligenten Diskretisierung den Fehler zu verringern. Falls man für die Diskretisierung stur feinere Auflösungen verwendet, muss aufgrund des Ziels des Lösens von hochdimensionalen Problemen beachtet werden, dass eine immer feinere Diskretisierung mit dem Fluch der Dimensionalität einherkommt. Der Fluch der Dimensionalität [Bel15] beschreibt, dass der Aufwand von Berechnungen stark mit der Anzahl an Dimensionen steigt. Betrachtet man z. B. ein isotropes Vollgitter mit n Punkte in alle d-Dimensionen; es enthält insgesamt n^d Punkte – exponentiell viele in der Anzahl der Dimensionen. Während ein solches isotropes Vollgitter mit einer kleinen Anzahl von Dimensionen noch akzeptable Speicher- und Recheneigenschaften besitzt, wird in hohen Dimensionen der Speicher- und Rechenaufwand zu groß um praktisch nützlich zu sein.

Das Ausführen von Computersimulationen ist eine Aufgabe die als Hochleistungsrechnen eingeordnet wird. Zum Berechnen sogenannter Hochleistungsrechenaufgaben werden die leistungsstärksten Computer, wie sie in der TOP500 Liste [TOP] gelistet werden, verwendet. Die regelmäßig aktualisierte TOP500 Liste listet Hochleistungsrechner mit Informationen über ihre Hardware und Rechenleistung. Betrachtet man die 10 leistungsstärksten Hochleistungsrechner aus der November 2019 Liste, so liegt die Anzahl an Kernen zwischen 288.288 und 10.649.600. Die minimale Anzahl an Kernen eines Hochleistungsrechner aus der TOP500 Liste bis Rang 500 liegt bei 12.800. Es ist erkennbar, dass Simulationssoftware für eine effiziente Nutzung der Hochleistungsrechner zwingend parallele Programmiertechniken verwenden muss.

Das Distributed-Combigrid-Technique-Framework (DisCoTec-Framework) bietet nun sowohl für den Fluch der Dimensionalität als auch der parallelen Programmierung entsprechende Lösungen an. Um die negativen Auswirkungen auf Rechen- und Speicheraufwand – aufgrund des Fluchs der Dimensionalität – mit einem möglichst kleinen Annäherungsfehler zu minimieren, werden Dünngitter verwendet. Es wird dabei die Dünngitter-Kombinationstechnik – statt den klassischen hierarchischen Dünngittern – verwendet, da diese gute Eigenschaften für die Parallelisierung bietet. Die mathematischen und algorithmischen Grundlagen des DisCoTec-Framework wurden in der Dissertation „A massively parallel combination technique for the solution of high-dimensional PDEs“ von Mario Heene [Hee18] beschrieben.

Das Ziel dieser Arbeit ist das Einführen einer dynamische Aufgaben-Umverteilung in das DisCoTec-Framework. Die einzige Möglichkeit der Lastbalancierung innerhalb des DisCoTec-Framework vor dieser Arbeit war statisch: Die parallel abzuarbeitenden Aufgaben werden einmalig zu Beginn der Simulation basierend auf Schätzungen verteilt. Die statische Lastbalancierung liefert bei einer hohen Anzahl von parallel abzuarbeitenden Aufgaben, deren Laufzeit über den Verlauf der Simulation gleich bleibt, eine gute Lastverteilung [Hee18]. Da das Ziel des DisCoTec-Framework das Anbieten eines möglichst generischen Frameworks ist, sollt ein Maximum an Methoden zur Berechnung sinnvoll zu verwenden sein. Deshalb ist es sinnvoll für Aufgaben mit über den Verlauf der Simulation sich verändernden Laufzeiten eine dynamische Umverteilung einzuführen, sodass die Lastinbalance durchgehend minimal bleibt. Eine Lastinbalance würde dazu führen, dass die parallel laufenden Aufgabengruppen aufeinander warten müssen. Somit werden die benutzbaren Computerresourcen nicht mehr effizient verwendet und die Dauer der Berechnung länger als nötig.

Die Arbeit beginnt mit den Grundlagen in Kapitel 2, welches an die im DisCoTec-Framework benutzte Dünngitter-Kombinationstechnik heranführt. Zum Ende des Grundlagenkapitel wird beschrieben wie parallele Programmierung im DisCoTec-Framework umgesetzt ist. Im darauffolgenden Kapitel 3 werden die Komponenten und der grundlegende Ablauf des DisCoTec-Framework vorgestellt. Anschließend wird in Kapitel 4 das Problem der Lastbalancierung von Aufgaben mit variabler Laufzeit vorgestellt und Lösungen angeboten. Das Kapitel 5 stellt die Ergebnisse der im vorherigen Kapitel angebotenen Lösungen anhand von Beispieldaten dar. Eine Diskussion, Fazit und Ausblick findet sich im letzten Kapitel 6.

Im Appendix A wird zuerst das Testsystem auf dem die Berechnungen liefen beschrieben. Die exakten Parameter zur Erstellung der Plots sind in Appendix B dokumentiert.

2. Grundlagen

Das DisCoTec-Framework bietet eine generische, parallelisierte Implementierung der Dünngitter-Kombinationstechnik. Für die Implementierung der Parallelisierung wird das Message Passing Interface (MPI) verwendet.

Für die mathematischen Grundlagen der Gitter, Dünngitter und Dünngitter-Kombinationstechnik wurden die Quellen [Hee18], [Val19], [Gar12] und [Gri92] verwendet und kombiniert. Die Gleichungen werden immer für d-Dimensionen formuliert. Falls eine Gleichung im Ein- oder Zweidimensionalen besser zu Veranschaulichung ist, so wird diese zusätzlich behandelt. Des Weiteren wird der Definitionsbereich zur Vereinfachung auf das Einheitsintervall $\Omega_{\text{Domäne}} = [0, 1]^d$ beschränkt. Es ist dabei möglich beliebige rechteckige Domänen über entsprechende Skalierungen darzustellen [Gar12]. Die Gitterstrukturen werden verwendet um passende Interpolanten für Funktionen $f : \Omega_{\text{Domäne}} \to \mathbb{R}$ zu finden.

MPI ist ein Standard für parallele bzw. verteilte Ausführung, welche von dem MPI Forum standardisiert wurde. Der MPI Standard spezifiziert ein Ausführungsmodell mit zugehörigen Funktionen und Datenstrukturen. Offizielle Schnittstellen sind dabei ausschließlich für die Programmiersprachen C/C++ und Fortran angegeben. MPI ist dabei insbesondere im Feld des Hochleistungsrechnen populär. Die aktuelle und hier verwendete Version des MPI Standards ist Version 3.1 aus dem Jahr 2015.

Die Grundlagen der Dünngitter-Kombinationstechnik werden in vier aufeinander aufbauenden Sektionen behandelt: Die grundlegenden regulären Gitter werden zuerst vorgestellt (Abschnitt 2.1), danach werden hierarchische Gitter (Abschnitt 2.2) und die darauf aufbauenden Dünngitter (Abschnitt 2.3) hergeleitet. Anschließend findet sich eine Beschreibung der Dünngitter-Kombinationstechnik (Abschnitt 2.4). Eine Einführung der relevanten Aspekte von MPI im Bezug auf das DisCoTec-Framework findet sich in der letzten Sektion (Abschnitt 2.5) dieses Kapitels.

2.1. Gitter

In dieser Arbeit werden Gitter $\Omega_{\text{Gitter}} \subsetneq \Omega_{\text{Domäne}}$ mit einer endlichen Anzahl an Punkten $|\Omega_{\text{Gitter}}| < \infty$ betrachtet. In dieser Arbeit wird ausschließlich mit einer endlichen Anzahl an Gitterpunkten gearbeitet, da das Ziel die Verarbeitung mithilfe eines Computers ist; beliebige Gitter mit unendlich vielen Punkten sind nur schwer mithilfe eines Computers zu verarbeiten. Da Gitter in dieser Arbeit nur endlich viele Punkte enthalten und auf den Bereich von $\Omega_{\text{Domäne}}$ beschränkt ist, sind sie echte Teilmengen von $\Omega_{\text{Domäne}}$.

Eine reguläres Gitter besitzt in jeder Dimension einen konstantem Gitterpunktabstand. Die Definition eines regulären Gitters im Zweidimensionalen lautet wie folgt:

(2.1) $\Omega_{\binom{l_x}{l_y}}$ mit Punkteabstand $h_x = 2^{-l_x}$ in x-Richtung und $h_y = 2^{-l_y}$ in y-Richtung.

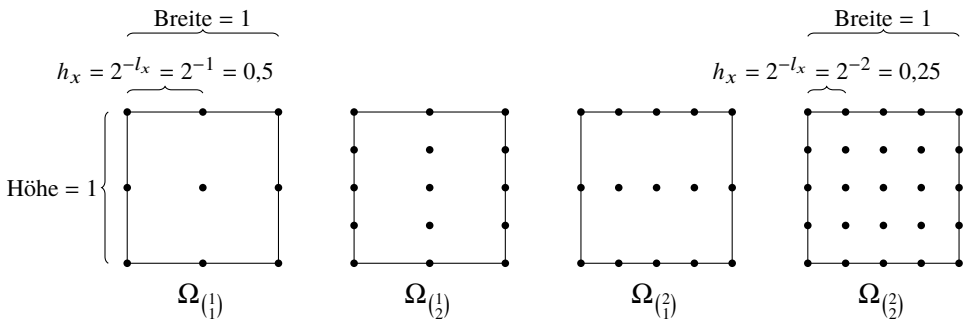

Abbildung 2.1.: Regulären isotrope ($\Omega_{\binom{1}{1}}, \Omega_{\binom{2}{2}}$) und anisotrope ($\Omega_{\binom{1}{2}}, \Omega_{\binom{2}{1}}$) Gitter

Die Gitter sind ausschließlich abhängig von l_x und l_y definiert. Die Variablen l_x und l_y sind Teil des (hier: zweidimensionalen) Levelvektor $\vec{l} = (l_x, l_y)^{\mathrm{T}}$. Ein Levelvektor definiert immer den Punkteabstand bzw. die Auflösung in den entsprechenden Dimensionen.

Beispiele von regulären Gittern finden sich in Abbildung 2.1: Man beachte, dass die Breite und Höhe bei allen vier Gittern identisch und gleich 1 ist. Einzig die Anzahl und Positionierung der Punkte unterscheidet sich. Es ist zu erkennen, dass der Wert des Levelvektors die Anzahl – und somit auch den Abstand – der Punkte in der entsprechenden Dimension verändert. Z. B. ist bei $\Omega_{(1,\,2)^{\mathrm{T}}}$ in der ersten (x-)Dimension der Abstand genau doppelt so groß wie in der zweiten (y-)Dimension, da für die Gleichung des Abstands $h = 2^{-l}$ gilt; l_y ist eins größer als l_x.

Die Definition des zweidimensionalen regulären Gitters $\Omega_{(l_x,l_y)^{\mathrm{T}}}$ lässt sich in das d-Dimensionale erweitern:

$$\text{(2.2)}\quad \Omega_{\vec{l}} \text{ mit } \vec{l} = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_d \end{pmatrix} \text{ und } \forall i \in [1, d] : \text{Punkteabstand } h_i = 2^{-l_i} \text{ in } i\text{-Richtung.}$$

Hier wurden die Komponenten des Vektors $\vec{l}$ explizit definiert. Im Folgenden wird implizit auf die i-te Komponente $i \in [1, d]$ eines d-dimensionalen Vektor $\vec{v}$ mit der Schreibweise v_i oder $(\vec{v})_i$ verwiesen.

In Abbildung 2.2 ist ein dreidimensionales reguläres Gitter mit Levelvektor $\vec{l} = (1, 2, 3)^{\mathrm{T}}$ zu sehen. Die erste Komponente $l_1 = 1$ gibt die Auflösung des Gitters in der Breite, die zweite Komponente $l_2 = 2$ in der Höhe und die dritte Komponente $l_3 = 3$ in der Tiefe an.

Gilt für den Levelvektor $\vec{l}$; $l_1 = l_2 = \cdots = l_d$, so enthält das Gitter $\Omega_{\vec{l}}$ ausschließlich äquidistante Gitterpunkte – die Gitterpunkte sind unabhängig der Richtung identisch angeordnet – und ist somit isotrop. Ist ein Gitter nicht isotrop, und somit die Position und Anzahl der Gitterpunkte abhängig von der Richtung, wird es anisotrop genannt. Beispiele für isotrope und anisotrope Gitter sind in Abbildung 2.1 zu sehen.

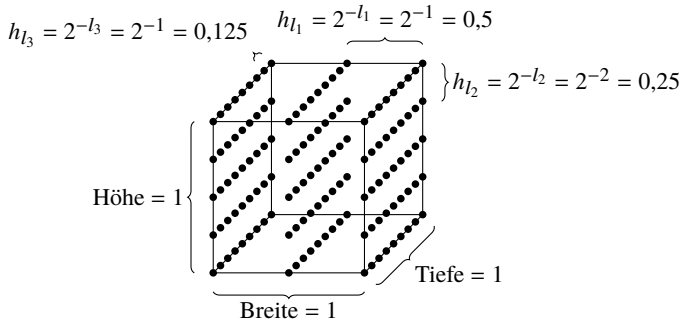

Abbildung 2.2.: Reguläres dreidimensionale anisotrope Gitter $\Omega_{\binom{1}{2}{3}}$

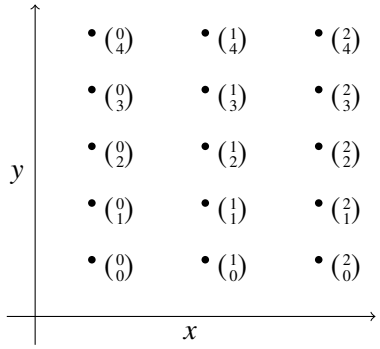

Abbildung 2.3.: Indizes der Punkte auf dem Gitter $\Omega_{\binom{1}{2}}$

Die Punktepositionen $\vec{p}_{\vec{l},\vec{i}}$ auf dem regulären Gitter $\Omega_{\vec{l}}$ sind eindeutig mit dem dazugehörigen Index $\vec{i}$ referenziert:

$$\vec{p}_{\vec{l},\vec{i}} = \begin{pmatrix} 2^{-l_1} i_1 \\ 2^{-l_2} i_2 \\ \vdots \\ 2^{-l_d} i_d \end{pmatrix} \quad \text{mit } \forall k \in [1,d] : i_k \in \{0,1,\ldots,2^{l_k}\} \,. \tag{2.3}$$

Der Levelvektor $\vec{l}$ wird benutzt um den Abstand der Gitterpunkte in der entsprechenden Dimension zu erhalten. Der Gitterpunkteabstand wird dann mit dem Index multipliziert. Das Ergebnis der Berechnung ist die Position des Punktes. Der Index $\vec{i}$ beschreibt exakt einen Punkt des regulären Gitters. In der Dimension k gibt es in einem Gitter $\Omega_{\vec{l}}$ genau $2^{l_k} + 1$ Punkte, die abhängig von ihrer Position einen Index von 0 bis 2^{l_k} zugewiesen bekommen. Siehe z. B. Abbildung 2.3; die Indizes beginnen unten links mit $\binom{0}{0}$. Für den nächsten Punkt in x-Richtung (visuell der nächste Punkt rechts) wird die erste Komponente des Index inkrementiert. Für den nächsten Punkt in y-Richtung (visuell der nächste Punkt oben) wird die zweite Komponente des Index inkrementiert.

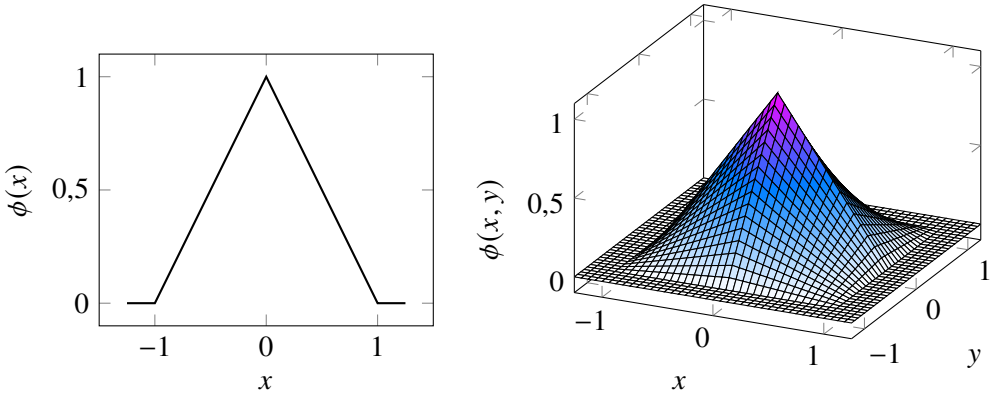

Abbildung 2.4.: Links: $\phi(x) = \max(1 - |x|, 0)$; eine eindimensionale Hütchenfunktion. Rechts: $\phi(x,y) = \max(1 - |x|, 0) \cdot \max(1 - |y|, 0)$; eine zweidimensionale Hütchenfunktion.

Um nun eine Funktion $f : \Omega_{\text{Domäne}} \to \mathbb{R}$ mithilfe der Struktur des Gitters $\Omega_{\vec{l}}$ zu interpolieren, wird eine zu dem Gitter gehörige Knotenbasis $V_{\vec{l}}$, bestehend aus d-dimensionalen Basisfunktionen $\phi_{\vec{l},\vec{i}}$, definiert:

$$V_{\vec{l}} = \text{span}\{\phi_{\vec{l},\vec{i}} \,|\vec{i} \in I_{\vec{l}}\} \quad \text{mit } I_{l_k} = \{0, 1, \ldots, 2^{l_k}\} \,. \tag{2.4}$$

Aus dem Raum $V_{\vec{l}}$ möchte man nun eine Funktion – den Interpolanten – finden, welche f möglichst gut annähert. Dazu muss zuerst noch eine Basisfunktion $\phi_{\vec{l},\vec{i}}$ gewählt werden. Für die Wahl der Basisfunktion $\phi_{\vec{l},\vec{i}} : \Omega \to \mathbb{R}$ gibt es viele Optionen. Eine simple und beliebte Wahl ist die für diese Arbeit gewählte d-dimensionale Hütchenfunktion:

$$\phi_{\vec{l},\vec{i}}(\vec{x}) = \prod_{k=1}^{d} \max\left(1 - \frac{\left|x_k - (\vec{p}_{\vec{l},\vec{i}})_k\right|}{2^{-l_k}}, 0\right) \,. \tag{2.5}$$

Die Spitze bzw. Mitte der Hütchenfunktion liegt auf den Gitterpunkten. Beispiele der Hütchenfunktion im Ein- und Zweidimensionalen sind in Abbildung 2.4 zu sehen.

Eine Hütchenfunktion hat einen rechteckigen Definitionsbereich auf dem sie ungleich 0 ist. Auf diesem Definitionsbereich nimmt sie im eindimensionalen Werte in Hütchenform an, während es im Zweidimensionalen einem Pagoden ähnlichen Gebilde entspricht. Die Form des Hütchen ist im Vieldimensionalen trotzdem noch zu erkennen: Siehe in Abbildung 2.4 im zweidimensionalen Fall mit $x = 0$ und $y \in [-1, 1]$.

Die zu der Knotenbasis $V_{(2)}$ gehörenden Hütchenfunktionen sind in Abbildung 2.5 zu sehen.

Es wird der folgende Interpolant aus der Knotenbasis $V_{\vec{l}}$ gewählt:

$$u_{\vec{l}}(\vec{x}) = \sum_{\vec{i} \in I_{\vec{l}}}^{d} \alpha_{\vec{i}} \phi_{\vec{l},\vec{i}} \quad \text{mit } \alpha_{\vec{i}} = f(\vec{p}_{\vec{l},\vec{i}}) \,. \tag{2.6}$$

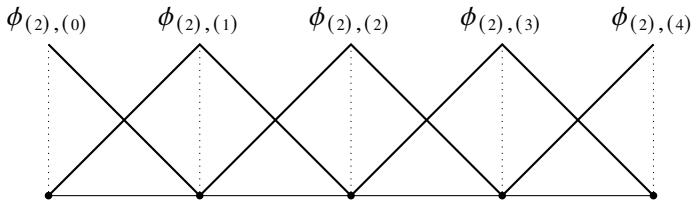

Abbildung 2.5.: Die Hütchenfunktionen von $V_{(2)}$ auf den Punkten des Gitter $\Omega_{(2)}$

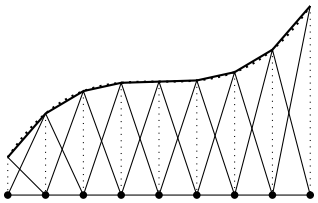

Abbildung 2.6.: Der Interpolant (durchgezogen) der Funktion $f(x) = x^3$ (gepunktet) aus der Knotenbasis $V_{(3)}$

Dank der Wahl der Hütchenfunktion als Basisfunktion kann durch das direkte Setzen der α_is direkt ein passender Interpolant aus der Knotenbasis gefunden werden. Eine Beispielhafte Interpolation ist in Abbildung 2.6 zu sehen: Es ist zu erkennen, dass die Spitze der Hütchenfunktionen immer genau auf dem Funktionswert liegt.

Komplexität Die Anzahl der zu speichernden Werte für einen Interpolanten aus der Knotenbasis $V_{\vec{l}}$ ist identisch zu der Anzahl der Gitterpunkte des unterliegenden regulären Gitters $\Omega_{\vec{l}}$. Ein reguläres isotropes Gitter $\Omega_{\vec{l}}$ mit $\forall i \in [1, d] : l_i = n$ enthält 2^{nd} Gitterpunkten.

Fehler Der Fehler des Interpolieren einer Funktion f mit einem Interpolanten $u_{\vec{l}}$ aus $V_{\vec{l}}$, wobei $\forall i \in [1, d] : l_i = n$ gilt, ist nach der L_2-Norm: $\|f - u_{\vec{l}}\|_{L_2} \in O(2^{-2n})$.

2.2. Hierarchische Gitter

Als Zwischenschritt zu den (hierarchischen) Dünngitter wird zuerst noch das hierarchische Gitter betrachtet. Während in der Knotenbasis die Hütchenfunktionen nebeneinander liegen, so werden bei der hierarchischen Basis die Hütchenfunktionen übereinander gestapelt: Statt dass wie bei der Knotenbasis Hütchenfunktionen der selben Form zu verwenden, wird bei der hierarchischen Basis mit deutlich größeren/breiteren Hütchen angefangen auf die anschließend kleinere Hütchen obendrauf gesetzt werden.

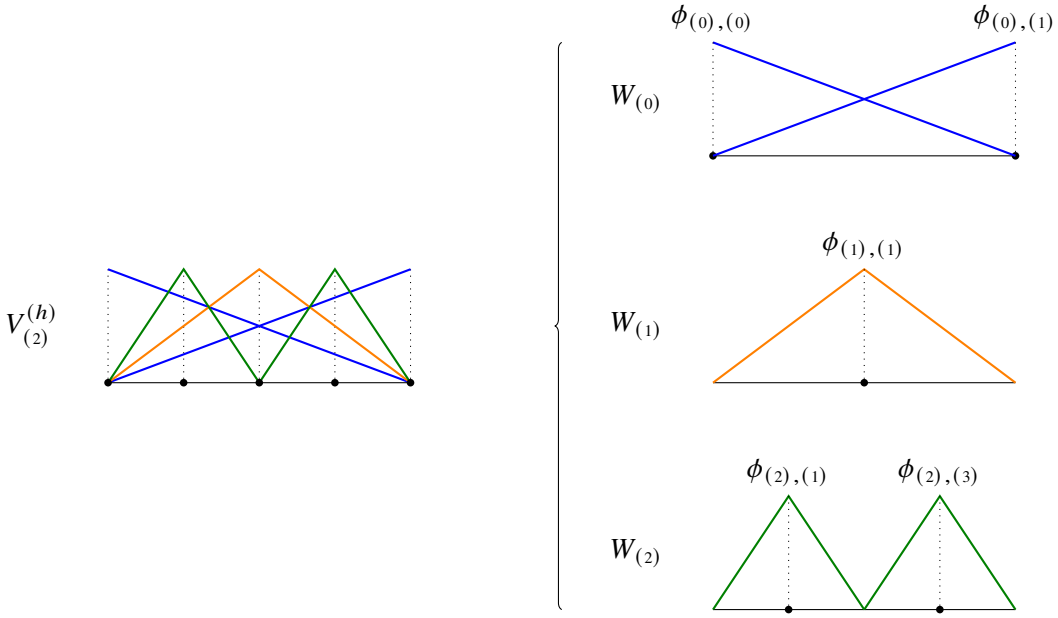

Abbildung 2.7.: Die hierarchische Basis $V^{(h)}_{(2)}$

Jede hierarchische Ebene ist ein Raum $W_{\vec{l}}$ abhängig von einem Levelvektor $\vec{l}$. Die Hütchenfunktionen sitzen im Vergleich zu $V_{\vec{l}}$ nur noch auf jedem zweiten Punkt des Gitters $\Omega_{\vec{l}}$. Die Definition der Indizes, welche die Punkte auf denen die Hütchenfunktion sitzen referenzieren, wird deshalb entsprechend angepasst:

$$I^{(h)}_{\vec{l}} = I^{(h)}_{l_1} \times I^{(h)}_{l_2} \times \cdots \times I^{(h)}_{l_d} \quad \text{mit } I^{(h)}_{l_k} = \begin{cases} \{1, 3, \ldots, 2^{l_k} - 1\}, & k > 0 \\ \{0, 1\}, & k = 0 \end{cases} . \tag{2.7}$$

Mithilfe der definierten Indexmenge $I^{(h)}_{\vec{l}}$ wird nun der Raum

$$W_{\vec{l}} = \text{span}\{\phi_{\vec{l},\vec{i}} | \vec{i} \in I^{(h)}_{\vec{l}}\} \text{ definiert.} \tag{2.8}$$

Für die hierarchische Basis $V^{(h)}_{\vec{l}}$ werden nun alle entsprechenden $W_{\vec{l}}$ mithilfe der direkten Summe kombiniert:

$$V^{(h)}_{\vec{l}} = \bigoplus_{\vec{l} \leq \vec{m}} W_{\vec{m}} \quad \text{mit } \vec{l} \leq \vec{m} \Leftrightarrow \forall i \in [1, d] : l_i \leq m_i . \tag{2.9}$$

Die zur hierarchischen Basis $V^{(h)}_{(2)}$ gehörenden Basisfunktionen sind in Abbildung 2.7 zu sehen: Die hierarchische Basis mit Levelvektor (2) hat drei Unterräume mit entsprechenden Hütchenfunktionen. Die direkte Summe „addiert“ diese Unterräume auf, sodass die hierarchische Basis entsteht, welche alle Hütchenfunktionen der Unterräume beinhaltet.

Betrachtet man die Punkte auf denen die Hütchenfunktionen liegen, so deckt $V^{(h)}_{\vec{l}}$ alle Punkte des regulären Gitters $\Omega_{\vec{l}}$ ab (siehe Abbildung 2.8).

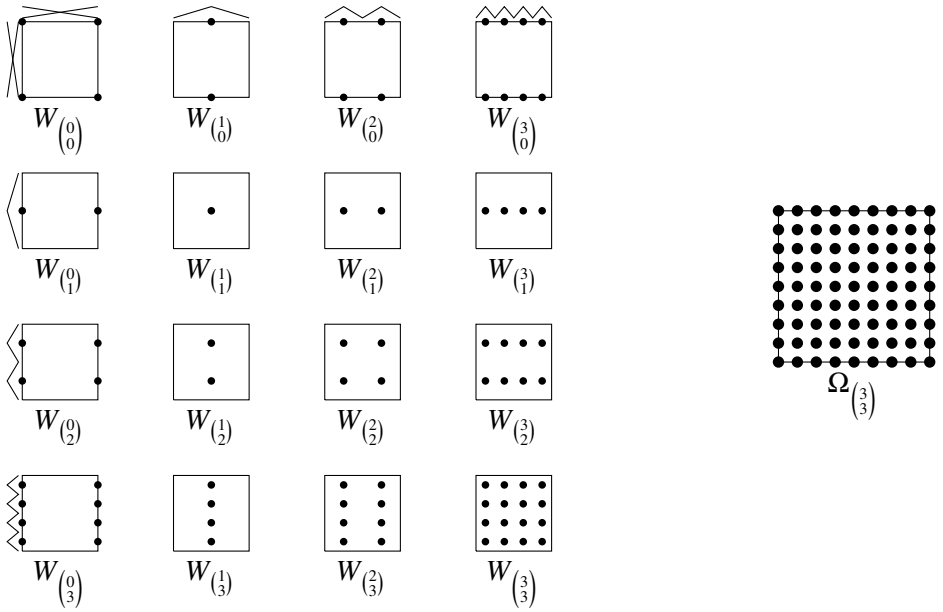

Abbildung 2.8.: Die unterliegenden Gitter der Räume $W_{\vec{l}}$ und das resultierende Vollgitter $\Omega_{(3)}$

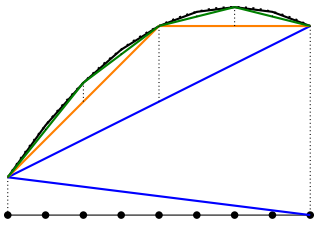

Abbildung 2.9.: Interpolation einer Funktion (gepunktet) auf der hierarchischen Basis $V^{(h)}_{(3)}$. Die Hütchenfunktionen sind aufaddiert dargestellt. Blau: $W_{(0)}$; Orange: $W_{(1)}$; Grün: $W_{(2)}$; Schwarz: $W_{(3)}$.

Die Wahl des Interpolanten ist in der hierarchischen Basis nicht trivial. Für die Erstellung (Hierarchisierung) und Auswertung (Dehierarchisierung) von hierarchischen Interpolanten müssen Algorithmen benutzt werden, welche in dieser Arbeit nicht behandelt werden. Genaueres zur (De-) Hierarchisierung wird in [Hee18] behandelt. Eine Beispielinterpolation kann in Abbildung 2.9 betrachtet werden. Beachte dass, im Gegensatz eines Interpolanten aus einer Knotenbasis, es möglich ist Hütchenfunktionen mit nur einem geringfügigen Fehler zu entfernen. Würde man eine Hütchenfunktion aus dem Interpolanten einer Knotenbasis entfernen, so hätte der Interpolant an dieser Stelle immer den Funktionswert 0.

Komplexität und Fehler Komplexität, d. h. die Anzahl an zu speichernden Werten, und Fehler ist identisch zur Interpolation auf Knotenbasis.

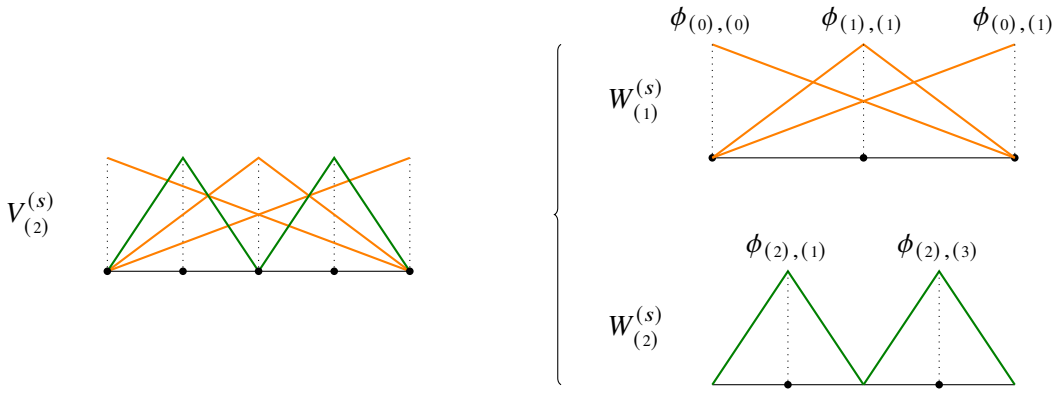

Abbildung 2.10.: Die hierarchische Basis $V^{(s)}_{(2)}$

2.3. Dünngitter

Im Vieldimensionalen $d > 1$ kann nun festgestellt werden, dass die Räume $W_{\vec{l}}$ mit einem hohen Levelvektor typischerweise nur einen kleinen Einfluss auf den Fehler besitzen, dabei allerdings die höchste Anzahl an zu speichernden Werte besitzen und somit stark zum Fluch der Dimensionalität beitragen. Die Struktur des Dünngitters ergibt sich wenn Räume mit hohem Levelvektor weggelassen werden; der Fehler steigt nicht zu stark und eine große Zahl an Punkten des Gitters kann weggelassen werden. Es wurde darauf geachtet, dass der Fehler bezüglich der L_2-Norm mit einer entsprechend glatten Funktion f nur minimal steigt:

$$V^{(s)}_{\vec{l}} = \bigoplus_{|\vec{m}|_1 \leq n+d-1} W^{(s)}_{\vec{m}} \quad \text{mit } |\vec{m}|_1 = \sum_{k=1}^{d} m_k \text{ und } \forall i \in [1,d] : l_i = n \,. \tag{2.10}$$

Um ein mit dem in der Dünngitter-Kombinationstechnik übereinstimmendes resultierendes Gittermuster zu erhalten wurde zusätzlich noch $W_{\vec{l}}$ leicht modifiziert: In $W^{(s)}_{\vec{l}}$ fällt $l_k = 0$ weg, stattdessen werden die zwei Hütchen in das Level $l_k = 1$ hinzugefügt. Z. B. wird im eindimensionalen $W_{(1)}$ und $W_{(0)}$ zusammen in $W^{(s)}_1$ gepackt. Vergleiche Abbildung 2.7 mit Abbildung 2.10 und Abbildung 2.8 mit Abbildung 2.11.

Komplexität Die Anzahl der zu speichernden Werte eines Interpolanten aus $V^{(s)}_{\vec{l}}$ ist identisch zu der Anzahl der Punkte des unterliegenden Gitters $\Omega^{(s)}_{\vec{l}}$. Die Anzahl der Gitterpunkte des Gitters $\Omega^{(s)}_{\vec{l}}$ liegt mit $\forall i \in [1,d] : l_i = n$ in $O(2^n \log(2^n)^{d-1})$, somit ist die Anzahl der benötigten Punkte deutlich geringer als mit einem Vollgitter auf Knoten- oder Hierarchiebasis.

Fehler Der Fehler des Interpolieren einer entsprechend glatten Funktion f mit dem Interpolanten $u^{(s)}_{\vec{l}}$ aus $V^{(s)}_{\vec{l}}$, wobei $\forall i \in [1,d] : l_i = n$ gilt, ist nach der L_2-Norm: $\|f - u^{(s)}_{\vec{l}}\|_{L_2} \in O(2^{-2n} \log(2^n)^{d-1})$.

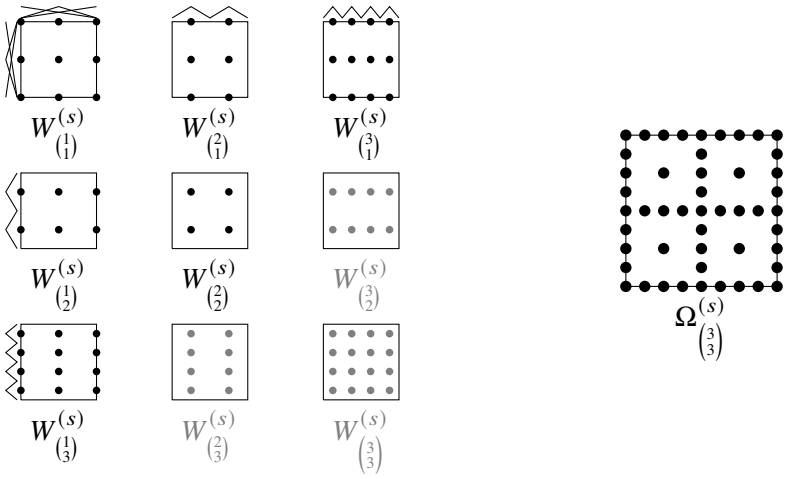

Abbildung 2.11.: Die unterliegenden Gitter der Räume $W^{(s)}_{\vec{l}}$ und das resultierende Dünngitter $\Omega^{(s)}_{(3)}$ (die ausgegrauten Gitter fallen weg)

2.4. Dünngitter-Kombinationstechnik

Mit Dünngittern auf hierarchischer Basis wurde nun der Fluch der Dimensionalität signifikant reduziert, allerdings wurde das Problem der Parallelisierung noch nicht angegangen. Aufgrund der hierarchischen Eigenschaft ist ein parallelisiertes Benutzen eines Dünngitters nur schwer umzusetzen: Die Unterräume können z. B. nicht sinnvoll unabhängig voneinander betrachtet und manipuliert werden.

Die Dünngitter-Kombinationstechnik verwendet mehrere (unabhängige) Unterräume in Form von regulären Gitter. Jeder Gitterpunkt bekommt einen Wert $u_{\vec{l}}(\vec{x}) : \Omega_l \to \mathbb{R}$ zugewiesen. Die unterliegenden Gitter $\Omega_{\vec{l}}$ sind dabei entsprechend gewählt, dass die Kombination davon einem Dünngitter $\Omega^{(s)}_{\vec{l}} = \Omega^{(c)}_{\vec{l}}$ entspricht. Zur Anschaulichkeit wird die namensgebende Kombination zuerst im Zweidimensionalen betrachtet:

$$u^{(s)}_{\binom{n}{n}}(\vec{x}) \approx u^{(c)}_{\binom{n}{n}}(\vec{x}) = \sum_{i+j=n+1} u_{\binom{i}{j}}(\vec{x}) - \sum_{i+j=n} u_{\binom{i}{j}}(\vec{x}) \; . \tag{2.11}$$

In Abbildung 2.12 ist eine beispielhafte Kombination auf das Zielgitter mit Gitterpunktwerten $u^{(c)}_{(3,\,3)^{\mathrm{T}}}(\vec{x})$ mit den Gitterpunktpositionen $\vec{x} \in \Omega^{(c)}_{(3,\,3)^{\mathrm{T}}}$ abgebildet.

Die Dünngitter-Kombinationstechnik ist folgendermaßen in d-Dimensionen formuliert:

$$u^{(s)}_{\vec{l}}(\vec{x}) \approx u^{(c)}_{\vec{l}}(\vec{x}) = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{|\vec{v}|_1 = n+d-1-q} u_{\vec{v}}(\vec{x}) \quad \text{mit } \forall i \in [1,d] : l_i = n \; . \tag{2.12}$$

Die Unabhängigkeit der zu kombinierenden Gitter kann dazu benutzt werden z. B. partielle Differenzialgleichungen parallel auf jedem der Gitter zu lösen. Klassische Parallelisierungsverfahren können zusätzlich angewandt werden. Es ist z. B. möglich mithilfe von Partitionierung die Berechnung auf mehreren Teilgebieten eines Gitters parallel auf mehreren Prozessen durchzuführen.

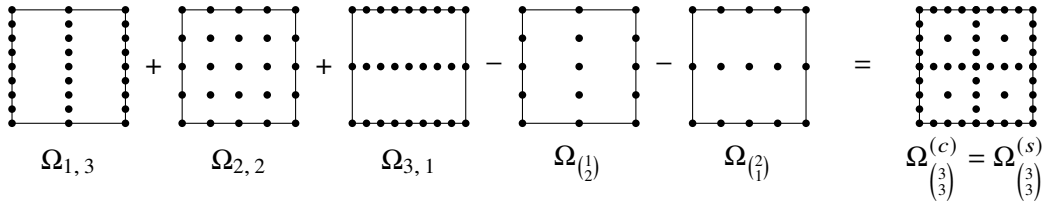

Abbildung 2.12.: Kombination im Zweidimensionalen mit Zielgitter $\Omega^{(c)}_{\binom{3}{3}}$

Komplexität Die Anzahl an verwendeten Gittern für eine Kombination auf $\Omega^{(c)}_{\vec{l}}$ mit $\forall i \in [1,d]$: $l_i = n$ liegt in $O(d \log{(2^n)^{d-1}})$. Jedes Gitter besitzt eine Anzahl an Punkten in $O(2^n)$. Die Anzahl an Punkten ist somit zwar größer als für ein Dünngitter, aber immer noch asymptotisch weniger als ein Vollgitter.

Fehler Für das Interpolieren einer entsprechend glatten Funktion f ist der Fehler nach der L_2-Norm identisch zu einem Dünngitter. Werden jedoch andere Probleme mithilfe der Dünngitter-Kombinationstechnik gelöst, wie z. B. das Lösen von partiellen Differenzialgleichungen, so fällt der Fehler im Vergleich zum Dünngitter meist größer aus.

2.5. Parallele Programmierung mit MPI

Das MPI definiert Schnittstellen zum Programmieren im Parallelen mit den Programmiersprachen Fortran, C und C++. Der Aktuelle Standard ist MPI 3.1 [Mes15], welcher auch als primäre Quelle für dieses Kapitel verwendet wurde. Implementierungen der Schnittstellen werden dabei unter anderem von zwei Open-Source Projekten angeboten: MPICH [MPI] (unterstützt von Intel, IBM, Cray und andere) und Open MPI [Ope] (unterstützt von AMD, ARM, Intel, IBM, Amazon, Facebook, Nvidia und andere). Für Hochleistungsrechner bieten außerdem die jeweiligen Hersteller angepasste Implementierungen – meist basierend auf MPICH oder Open MPI – an, welche Unterstützung für die entsprechende Hardware bietet.

Während MPI ursprünglich parallele Programmierung ausschließlich mithilfe von nachrichtenbasierter Kommunikation anbot, wurde in nachfolgenden Versionen weitere Möglichkeiten der Kommunikation hinzugefügt: Z. B. Schnittstellen für das Benutzen geteilten Speichers. Da das DisCoTec-Framework zum Zeitpunkt dieser Arbeit ausschließlich Nachrichten für die Kommunikation verwendet, wird einzig die nachrichtenbasierte Kommunikation vorgestellt.

Eine Implementierung von MPI stellt zum einen einen angepassten Compiler und zum anderen ein Programm zum Ausführen des erstellten MPI Programms bereit. Das Ausführen von MPI Programmen ist zwingend über die bei der Implementierung mitgelieferten Startprogramme, meist `mpirun` und/oder `mpiexec` genannt, notwendig. Der Grund weshalb ein MPI Programm nicht einfach gestartet werden kann liegt daran, dass unter anderem angegeben werden muss mit wie vielen Prozessen das Programm gestartet werden soll. Beim Start muss außerdem die Kommunikation der Prozesse initialisiert werden. Der Code für MPI muss daher spezielle Funktionen nach dem Start und vor dem Ende aufrufen (siehe: Listing 2.1).

Listing 2.1 MPI Initialisierung und Finalisierung (Beenden)

```
1 #include <mpi.h>
2
3 int main(int argc, char **argv)
4 {
5   MPI_Init(&argc, &argv);
6
7   // ...
8
9   MPI_Finalize();
10 }
```

Ein MPI-Prozess ist jeweils eine Ausführung des Programms; Prozesse werden parallel zueinander ausgeführt. Es ist möglich einen Prozess auf einen Thread, CPU-Kern, Sockel oder Computer zu legen. Von MPI werden auch Hochleistungsrechner – oft ein Zusammenschluss aus vielen einzelnen Computern verbunden mit einem schnellen Interconnect – unterstützt.

2.5.1. Nachrichtenaustausch

In MPI können Prozesse sich untereinander Nachrichten schicken. Eine Nachricht enthält die Information welcher (Daten-)Typ und wie viele Elemente verschickt bzw. empfangen werden. Beim Senden und Empfangen wird außerdem die Anfangsadresse eines Buffers mitgegeben, in dem die Daten zum Versenden stehen oder die empfangenen Daten hineingeschrieben werden. Das MPI spezifiziert, dass Nachrichten eine „exactly-once“ Garantie besitzen. Das bedeutet, eine Nachricht wird exakt einmal verschickt und empfangen. Dadurch sinkt der Implementierungsaufwand für nachrichtenbasierte Kommunikation deutlich.

Um anzugeben an wen die Nachricht geschickt wird, muss angegeben werden welcher Kommunikator verwendet wird und an welchen Prozess in dem Kommunikator die Nachricht gehen soll. Jede Prozess in einem Kommunikator hat in diesem einen eindeutigen Rang. Am Anfang existiert ein einzelner Kommunikator: Das `MPI_COMM_WORLD` Kommunikator-Handle, in welchem jeder MPI-Prozess einen eindeutigen Rang besitzt. Im Laufe des Programms können zusätzliche Kommunikatoren mit einer Untermenge von MPI-Prozessen erstellt werden.

Senden und Empfangen von Nachrichten ist mithilfe der in MPI angebotenen Funktionen `MPI_Send` und `MPI_Receive` möglich. Mithilfe von `MPI_Send` können Nachrichten zu einem anderen Prozess geschickt werden, welcher diese Nachricht mit `MPI_Receive` empfangen muss. In Listing 2.2 ist ein Beispiel-Programm beschrieben, welches mit mindestens 2 MPI-Prozessen gestartet werden muss. In dem Beispiel schickt der Prozess mit dem Rang 0 im `MPI_COMM_WORLD`-Kommunikator dem Prozess mit Rang 1 einen Integer.

Nachrichten können synchron oder gebuffert gesendet werden. Bei einer synchronen Nachricht werden die Daten von dem Sender direkt in den Empfänger gespeichert, dabei muss der Sender warten bis der Empfänger bereit ist; erst dann kann mit der Datenübertragung angefangen werden. Im Gegensatz dazu kann bei einer gebufferten, asynchronen Übertragung der Sender die Nachricht sofort in den einen temporären Buffer legen: Der Sender muss nicht auf den Empfänger zu warten. Der Empfänger liest dann die Nachricht aus dem Buffer. Die beiden Methoden des Nachrichtenversenden haben dabei gegensätzliche Vor- und Nachteile. Während man in MPI angeben kann welches Senden

Listing 2.2 Beispiel des Senden eines Integers von einem Prozess zu *einem* anderen.

```
int my_rank;
int result = 0;

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) {
  int to_send = 42;
  MPI_Send(
      &to_send,       // IN: Sende die Daten ab der Adresse der to_send Variable
      1,              // IN: Sende nur 1 Element
      MPI_INT,        // IN: Typ des zu versendenden Elements
      1,              // IN: Rang des Ziels im benutzten Kommunikator
      99,             // IN: Nachrichten Tag
      MPI_COMM_WORLD  // IN: Der für die Nachricht zu benutzende Kommunikator
  );
} else if (my_rank == 1) {
  MPI_Recv(
      &result,          // OUT: Zieladresse der Empfangenen Daten
      1,                // IN : Empfange nur 1 Element
      MPI_INT,          // IN : Typ des zu empfangenden Elements
      0,                // IN : Rang der Quelle im benutzten Kommunikator
      99,               // IN : Nachrichten Tag - muss identisch zu dem Tag des Senden sein
      MPI_COMM_WORLD,   // IN : Der für die Nachricht benutzte Kommunikator
      MPI_STATUS_IGNORE  // OUT: Rückgabe des Status wird hier ignoriert
  );
}

// if my_rank == 1:
//   prints: 42
// else:
//   prints: 0
std::cout << result << "\n";
```

und Empfangen genutzt werden soll (synchron oder gebuffert), so wird bei dem empfohlenen `MPI_Send` je nach Nachrichtengröße entschieden ob synchron oder gebuffert bessere Performance liefert. Große Nachrichten werden synchron gesendet, sodass das Speichern und Laden in bzw. aus dem temporären Buffer übersprungen wird. Kleine Nachrichten werden gebuffert gesendet.

3. Das DisCoTec-Framework

Wie in der Einleitung erwähnt, sind die Ziele des DisCoTec-Framework das Minimieren des Fluchs der Dimensionalität und Parallelisierung um effiziente Berechnungen auf Hochleistungsrechner durchzuführen. Für das Erreichen beider Ziele wird die in Abschnitt 2.4 vorgestellte Dünngitter-Kombinationstechnik benutzt. Die Dissertation von Heene [Hee18] beschreibt die grundlegende Architektur und Algorithmen des DisCoTec-Framework. Implementiert wurde allerdings nur eine Untermenge der in der Dissertation vorgestellten Methoden.

Ein aktueller Stand des DisCoTec-Framework ist in dem Github-Repository des SG++ Projektes [SG] zu finden. Dieses Kapitel zur Beschreibung des DisCoTec-Framework nimmt als Basis den Stand des Git-Repository zum Commit `1980262`.

Ein Überblick über die Komponenten wird in Abschnitt 3.1 geliefert. Anschließend wird der Ablauf in Abschnitt 3.2 beschrieben.

3.1. Komponenten

Für das DisCoTec-Framework hat man sich für eine Aufteilung in Manager und Worker entschieden. Der Manager ist zuständig für die Orchestrierung der Schritte der Dünngitter-Kombinationstechnik, wie „Berechnung“ und „Kombination“. Während auf den Worker die Berechnungen auf den unabhängigen regulären Gittern stattfinden, welche kombiniert das Ergebnis repräsentieren.

Das DisCoTec-Framework besteht aus mehreren Komponenten, welche in entsprechenden MPI-Prozessen ausgeführt werden. Der Manager ist eine Instanz des `ProcessManager`. Dieser enthält Instanzen von `ProcessGroupManager`, die Kommunikation mit jeweils einer Prozessgruppe bereit stellen. Um die Parallelität zu erhöhen kann eine Prozessgruppe mehr als einen Worker besitzen. Der `ProcessGroupManager` kommuniziert immer nur mit einem Worker der Prozessgruppe, dem sogenannte `ProcessGroupWorker`-Master. Der `ProcessGroupWorker`-Master ist dafür verantwortlich die ihm übertragenen Aufgaben zusammen mit den anderen `ProcessGroupWorker` der Prozessgruppe auszuführen. Eine Visualisierung der Komponenten und ihre Kommunikation wird in Abbildung 3.1 dargestellt.

3.1.1. `ProcessManager`

Der `ProcessManager` ist das zentrale Objekt, welches alle auszuführenden Aktionen orchestriert. Es sorgt dafür, dass die Aufgaben zu Beginn auf die Prozessgruppen verteilt werden, die nächste Berechnung wird über ihn angestoßen, der Kombinationsschritt wird über ihn angestoßen und es ist möglich über ihn das aktuelle Ergebnis der Berechnung abzurufen.

3.1.2. `ProcessGroupManager`

Eine Instanz des `ProcessGroupManager` repräsentiert jeweils genau eine Prozessgruppe. Sie ist in der Lage dieser Prozessgruppe Aufgaben zuzuweisen und Nachrichten wie „starte die nächste Berechnung“ oder „führe die Kombinationsroutine aus“ zu senden. Jeder `ProcessGroupManager` speichert außerdem welche Aufgaben der dazugehörigen Prozessgruppe zugewiesen sind.

3.1.3. `ProcessGroupWorker`

Jeder `ProcessGroupWorker` wird in einem eigenen MPI-Prozess gestartet. Die Kommunikation findet daher ausschließlich über die von MPI angebotenen Konzepte, wie MPI-Nachrichten, statt. Die Kommunikation mit einem `ProcessGroupWorker` basiert auf Signale: Soll z. B. der nächste Berechnungsschritt ausgeführt werden, so muss dem Worker das `RUN` Signal gesendet werden. Für alle Aktionen gibt es ein Signal, welches sobald es Empfangen wird die entsprechende Aktion ausführt. Ist das Ausführen der empfangenen Aktion abgeschlossen, wird auf das nächste Signal gewartet. Eine Prozessgruppe besteht aus mindestens einem `ProcessGroupWorker`. Die Anzahl der zu einer Prozessgruppe gehörenden `ProcessGroupWorker` ist konstant.

Die `ProcessGroupWorker` einer Prozessgruppe arbeiten die der Prozessgruppe zugewiesenen Aufgaben im Zusammenspiel ab. Dazu bekommt jeder `ProcessGroupWorker` eine Partition des unterliegenden Gitters zugeteilt.

3.1.4. `ProcessGroupWorker`-Master

Jede Prozessgruppe hat genau einen `ProcessGroupWorker`-Master. Der `ProcessGroupWorker`-Master ist ein `ProcessGroupWorker`, mit der zusätzlichen Aufgabe des Weiterleiten von Nachrichten an die anderen `ProcessGroupWorker`. Es ist somit möglich die gesamte Kommunikation des `ProcessGroupManager` für die Prozessgruppe über den `ProcessGroupWorker`-Master abzuwickeln.

3.1.5. `Task`

Der Begriff des „Tasks“ bzw. eingedeutscht „Aufgabe“ beschreibt das Durchführen der Berechnung auf einem spezifischen regulären Gittern der Dünngitter-Kombinationstechnik. Jeder Task hat eine eins zu eins Beziehung mit einem regulären Gitter und dessen Levelvektor.

Ein `Task` ist eine abstrakte Klasse bzw. Interface, von welchem der Anwender die Implementierung vervollständigen muss. Der `Task` erhält für die Initialisierung einen MPI-Kommunikator der Prozessgruppe, der er zugeteilt wurde. Teil des `Task` ist dabei das reguläre Gitter auf dem die Berechnungen durchgeführt werden müssen. Das DisCoTec-Framework bietet eine verteilte Implementierung eines regulären Gitter an. Die `run` Funktion ist dazu da einen Berechnungsschritt auszuführen. Der Berechnungsschritt muss dabei das dem Task gehörende verteilte Gitter aktualisieren.

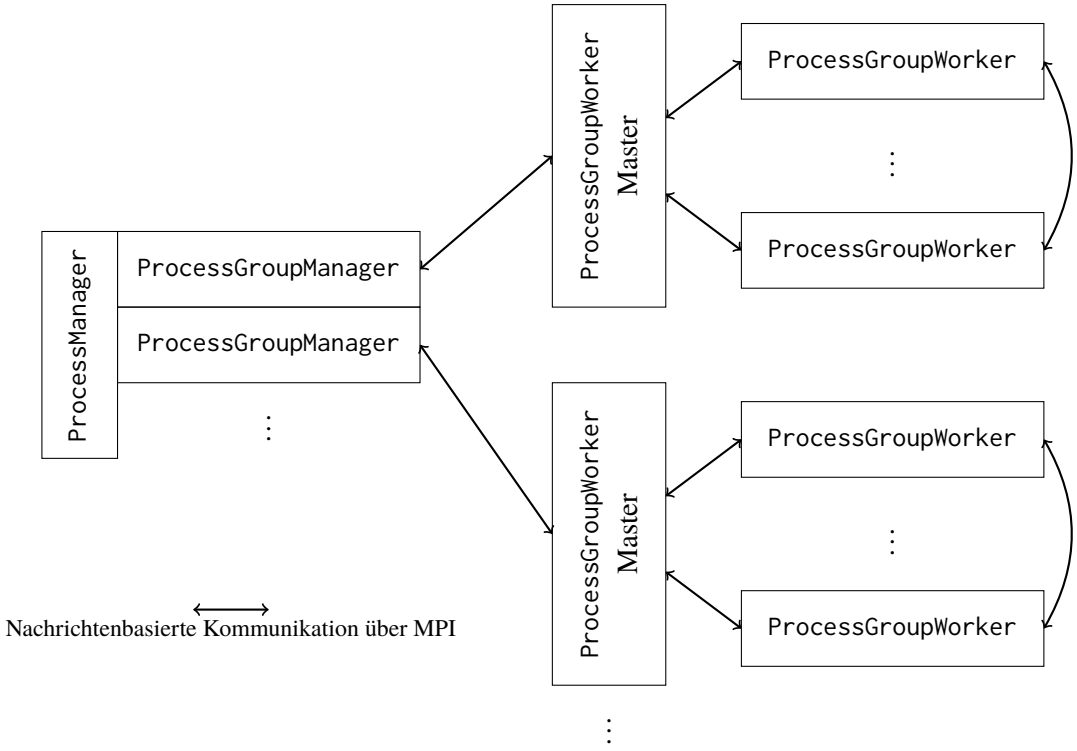

Abbildung 3.1.: Anordnung und Kommunikation der Komponenten des DisCoTec-Frameworks

3.2. Ablauf

Die Komponenten müssen nun auf entsprechenden MPI-Prozessen gestartet werden. Die Komponente `ProcessManager` steht an oberster Stelle: Der `ProcessManager` enthält für jede Prozessgruppe einen `ProcessGroupManager`. Die Kommunikation zwischen `ProcessManager` und `ProcessGroupManager` findet mithilfe von klassischen C++ Funktionen innerhalb eines MPI-Prozesses statt. Jeder `ProcessGroupManager` kennt einen `ProcessGroupWorker`, welcher in einem anderen MPI-Prozess ausgeführt wird. Der `ProcessGroupWorker` mit dem die Kommunikation mit dem `ProcessGroupManager` stattfindet ist der `ProcessGroupWorker`-Master. Jeder Prozessgruppe hat genau einen `ProcessGroupWorker`-Master und einen `ProcessGroupManager`. Zu einer Prozessgruppe können noch weitere `ProcessGroupWorker` gehören, welche über den `ProcessGroupWorker`-Master verwaltet werden. Jede Prozessgruppe besitzt identisch viele `ProcessGroupWorker`.

Die Abbildung 3.1 visualisiert die Kommunikation der vorgestellten Komponenten. Beachte, dass der Pfeil bedeutet, dass die entsprechenden Komponenten das MPI zum Nachrichten schicken und empfangen benutzen. Komponenten die sich in Abbildung 3.1 berühren werden auf demselben MPI-Prozess ausgeführt: Die Manager werden zusammen auf einem MPI-Prozess ausgeführt, während jeder `ProcessGroupWorker` jeweils auf einem MPI-Prozess ausgeführt wird. Die Kommunikation der Komponenten, die im selben MPI-Prozess liegen, findet regulär über die von der C++ Programmiersprache angebotenen Funktionsaufrufe statt. Der Manager-Prozess kommuniziert nur mit

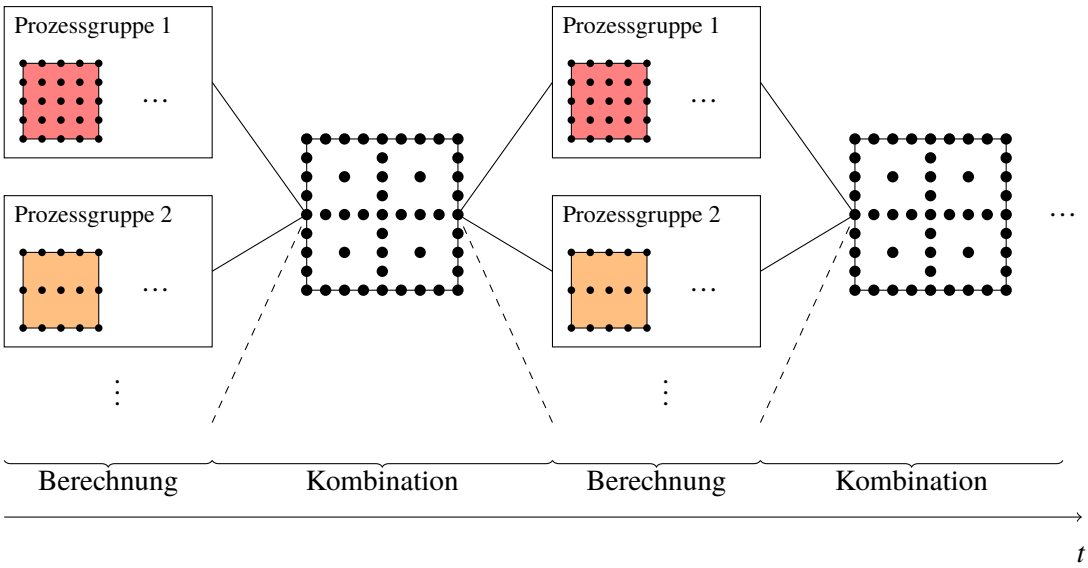

Abbildung 3.2.: Schritte der Ausführung des DisCoTec-Frameworks

den `ProcessGroupWorker`-Master. Die `ProcessGroupWorker` einer Prozessgruppe kommunizieren untereinander mit allen, der Grund dafür ist, dass das Gitter der Tasks in einer Prozessgruppe untereinander ausgetauscht wird.

Zum Initialisieren werden die Tasks nacheinander auf die Prozessgruppen verteilt. Danach folgt der in Abbildung 4.3 visualisierte Berechne, Kombiniere Zyklus. Der erste Schritt ist das Ausführen der Tasks. Für jede Prozessgruppe ist zuvor schon definiert welche Tasks sie berechnen muss. Ein Task ist in Abbildung 4.3 als das zugehörige Gitter dargestellt. Der nächste Schritt ist das Kombinieren, bei dem die berechneten Gitter der Tasks zu einem Dünngitter mithilfe der Dünngitter-Kombinationstechnik kombiniert werden. Das kombinierte Dünngitter wird wiederum auf die Prozessgruppen verteilt, welche die Gitter ihrer Tasks aktualisieren. Der Berechne, Kombiniere Zyklus wiederholt sich solange, bis die Abbruchbedingung des zu lösenden Problems erreicht ist.

4. Lastbalancierung

In der Dissertation von Heene [Hee18] wurden schon die Aspekte für die Lastbalancierung von Tasks mit konstanter Laufzeit genau beschrieben. Aufgrund der konstanten Laufzeit der Tasks wurde in der Dissertation davon ausgegangen, dass sobald eine gute Lastverteilung gefunden wurde diese für den Rest der Simulation gleich bleiben kann. Abschnitt 4.1 beschreibt die Lastverteilung mit Tasks konstanter Laufzeit genauer.

Wenn nun allerdings die Laufzeit der Tasks sich über die Zeit verändert, so werden sich dadurch auch die Laufzeiten der Prozessgruppen verändern. Es ist daher sehr wahrscheinlich, dass Prozessgruppen sich über den Verlauf der Simulation in ihrer Laufzeit unterscheiden werden. Das Resultat ist, dass schnelle Prozessgruppen auf langsame Prozessgruppen warten müssen. Dadurch sinkt die Effizienz der Berechnung und die Laufzeit der gesamten Simulation steigt. Das Ziel ist nun die Annäherung der Laufzeit der Prozessgruppen. Um die Laufzeit der Prozessgruppe zu verändern werden die Tasks zwischen den Prozessgruppen verschoben. Eine Detailliertere Begründung findet sich in Abschnitt 4.2.

Danach wird die in dieser Arbeit vorgestellte Lösung der dynamischen Aufgaben-Umverteilung in Abschnitt 4.3 behandelt. Am Ende werden noch die erwarteten Ziele werden in Abschnitt 4.4 beschrieben.

4.1. Lastbalancierung von Tasks mit konstanter Laufzeit

Für die Lastbalancierung von Tasks mit konstanter Laufzeit wurden in der Dissertation von Heene [Hee18] entsprechende Verfahren vorgestellt. Diese Verfahren fanden sich zu der Zeit dieser Arbeit allerdings nur eingeschränkt in dem DisCoTec-Framework.

Die Lastbalancierung findet gleichzeitig mit dem Verteilen der Tasks statt: Zuerst werden die Tasks nach ihrer vermuteten Laufzeit sortiert. Es werden dann die Tasks mit längster erwarteter Laufzeit auf die Prozessgruppen verteilt. Jede Prozessgruppe initialisiert und berechnet den ersten Berechnungsschritt der empfangenen Tasks. Nach dem Abschließen des Berechnungsschrittes des Tasks wartet die Prozessgruppe auf einen nächsten Task. Die erwarteten nächst schnelleren Tasks werden auf eine aktuell wartende Prozessgruppe verteilt. Sind alle Tasks verteilt, wird gewartet bis die Prozessgruppen fertig mit der Initialisierung und dem Berechnen des ersten Schritts sind. Ab jetzt wird der reguläre Ablauf (vgl. Abbildung 4.3) einer Kombination, Berechnung, Kombination usw. ausgeführt.

Wenn die Tasks über die Simulation konstante Laufzeiten des Berechnungsschritt besitzen und die Anzahl an Tasks pro Prozessgruppe groß ist, folgt aus dieser Art der Lastverteilung eine gute Effizienz. Zusätzlich gilt: Sollte die benötigte Zeit der Initialisierung im Vergleich zu der Laufzeit des Berechnungsschrittes relevant sein, so wird die Verteilung der Tasks negativ beeinträchtigt. Der

Algorithmus 4.1 Lastbalancierungs-Algorithmus mit Korrektur aus [Hee18]

Input: g_i Laufzeit der Prozessgruppe i
n Anzahl Prozessgruppen
ϵ Grenzwert der Unausgewogenheit
$k_{\max}$ Maximale Anzahl an Iterationen

1: $k \leftarrow 0$
2: $g_{\text{avg}} \leftarrow \sum_i^n g_i \div n$
3: **while** $k < k_{\max}$ **do**
4: $g_{\text{slow}} \leftarrow \max_i g_i$
5: **if** $g_{\text{slow}} \div g_{\text{avg}} < \epsilon$ **then**
6: exit
7: **end if**
8: $g_{\text{fast}} \leftarrow \min_i g_i$
9: $t \leftarrow$ GetTaskDurationsOfProcessGroup(slow)
10: $j \leftarrow \arg\min_i \left|g_{\text{fast}} + t_i - g_{\text{avg}}\right|$
11: **if** $g_{\text{fast}} + t_j < g_{\text{slow}}$ **then**
12: move task j to g_{fast}
13: $k \leftarrow k + 1$
14: **else**
15: exit
16: **end if**
17: **end while**

Grund dafür ist, dass die Tasks im Hinblick auf die Dauer von Initialisierung *und* Berechnung verteilt wurden, nun allerdings nur noch die Berechnung ausgeführt wird. Um dieses Problem zu lösen wird in [Hee18] vorgeschlagen die Zeit der Berechnung zu messen und danach mithilfe des Ausführens von Algorithmus 4.1 eine Lastumverteilung durchzuführen. Die Lastumverteilung benutzt nun die Messdaten der Berechnungsdauer um die Tasks besser zu verteilen. Der Algorithmus 4.1 wurde bis auf eine kleine Korrektur und Umbenennung von Variablen aus [Hee18] übernommen. Die vorgenommene Korrektur ist, dass in Zeile 12 „move task j to *slowest* group“ zu „move task j to *fastest* group“ geändert wurde, da die schnellste Prozessgruppe mehr Tasks zum Abarbeiten bekommen und die langsamste Prozessgruppe schneller werden sollte. Die Grundidee des Algorithmus 4.1 ist ein möglichst gutes Annähern der Prozessgruppenlaufzeit an den Durchschnittswert. Zu sehen ist das in Zeile 12: Es wird ein Task aus der Langsamen Prozessgruppe gesucht, welche die zu schnelle Prozessgruppe möglichst dich an die Durchschnittslaufzeit annähert. Ist die Prozessgruppenlaufzeit aller Prozessgruppen identisch mit dem Durchschnittswert, so ist die Lastverteilung perfekt, da jede Prozessgruppe dieselbe Laufzeit besitzt.

Der Stand der Lastbalancierung des DisCoTec-Framework vor dieser Arbeit war der Folgende: Die Tasks wurden einmalig während dem Start der Simulation anhand ihrer geschätzten Laufzeit, wie oben beschrieben, nacheinander Verteilt. Eine nachträgliche Umverteilung der Tasks existierte nicht.

4.2. Beispiel für die Notwendigkeit von dynamischer Aufgaben-Umverteilung

Das DisCoTec-Framework betrachtet den Berechnungsschritt der Tasks als Blackbox. Das Ziel ist nun Mittel anzubieten um Tasks mit einer über die Simulation verändernden Berechnungslaufzeit sinnvoll mithilfe des DisCoTec-Framework zu behandeln. Dass die Laufzeit des Berechnungsschritt der Tasks sich über die Zeit verändert, kann viele Gründe haben. Z. B. könnten die übergebenen Tasks ihre Berechnung mithilfe von internen adaptiven Zeitschritten berechnen. Je nachdem wie viele interne Zeitschritte benutzt werden wird die Laufzeit der Berechnung unterschiedlich ausfallen. Oder wenn die Aufgaben einen impliziten Solver, wie z. B. das Rückwärts Eulerverfahren, benutzen, dann kann auch nicht mehr von einer konstanten Laufzeit ausgegangen werden. Der Grund dafür liegt an der Herangehensweise der impliziten Verfahren: Betrachtet man z. B. die zu lösende Gleichung des Rückwärts Eulerverfahren [BZBP13]

$$y_{k+1} = y_k + \delta t f(t_{k+1}, y_{k+1}) \ ,$$

so erkennt man, dass y_{k+1} auf beiden Seiten der Formel auftaucht. Um eine solche Gleichung zu lösen muss ein (eventuell nichtlineares) Gleichungssystem aufgestellt und gelöst werden. Die Lösung des Gleichungssystems findet meist über iterative Verfahren statt. Diese iterative Verfahren sind in ihrer Laufzeit nicht konstant.

Das zu lösende Beispielproblem, welches in dieser Arbeit verwendet wird, ist eine Advektionsgleichung. Dazu wird eine Masse von einer Ecke Diagonal in die gegenüberliegende bewegt. Das Ziel ist dabei unterschiedliche Veränderungen auf den verschiedenen Gittern der Tasks zu beobachten, sodass im Zusammenspiel mit einem impliziten Verfahren sich das Laufzeitverhalten über den Lauf der Simulation ändert. Als Algorithmus für das Lösen des Problems wird das Rückwärts Eulerverfahren aus dem PDELab des Distributed and Unified Numerics Environment (DUNE) [BBD+16] benutzt.

Ein Beispielhaftes Laufzeitverhalten des Tasks lässt sich in Abbildung 4.1 betrachten. Es ist zu sehen, dass der Task 5,8 in der Laufzeit des Berechnungsschritt variiert.

Aus den stark variierenden Dauern der Tasks folgt, dass die Prozessgruppen definitiv aufeinander warten müssen. In Abbildung 4.2 ist die Berechnungsdauer der Prozessgruppen abgebildet; die Laufzeit ergibt sich aus der Summe der Berechnungszeiten der jeweils zugewiesenen Tasks. Aus dem Plot der Prozessgruppenlaufzeiten ist erkennbar, dass während den Berechnungsschritten 20 bis 40 keine gute Lastverteilung vorhanden ist. So muss die Prozessgruppe 0 160.000 Mikrosekunden auf Prozessgruppe 2 warten. Die Prozessgruppe 0 verbringt während den Berechnungsschritten also mehr Zeit mit Warten, als Berechnungen. Vergleicht man Abbildung 4.2 mit Abbildung 4.1 so sieht man, dass die Tasks 8,5 und 5,8 zwischen Berechnungsschritt 20 und 40 die selbe Berechnungsdauer haben. Eine bessere Verteilung für diese Berechnungsschritte wäre: Für Task 8,5 und 5,8 eine eigene Prozessgruppe zu benutzen und alle übrigen (schnellen) Tasks in die dritte Prozessgruppe zu packen.

Eine gute Verteilung der Prozessgruppen ist dann gegeben, wenn die Laufzeiten nahe dem Durchschnitt liegen. Liegen alle Prozessgruppenlaufzeiten nahe an dem Durchschnitt, sind sich die Laufzeiten untereinander sehr ähnlich, somit ist auch notwendiges aufeinander Warten minimiert.

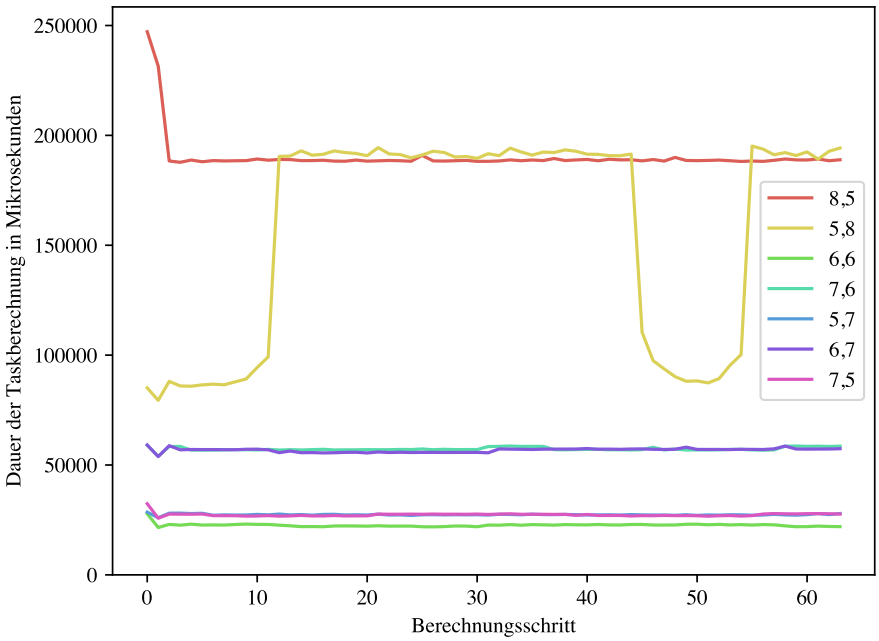

Abbildung 4.1.: Dauer der Taskberechnung

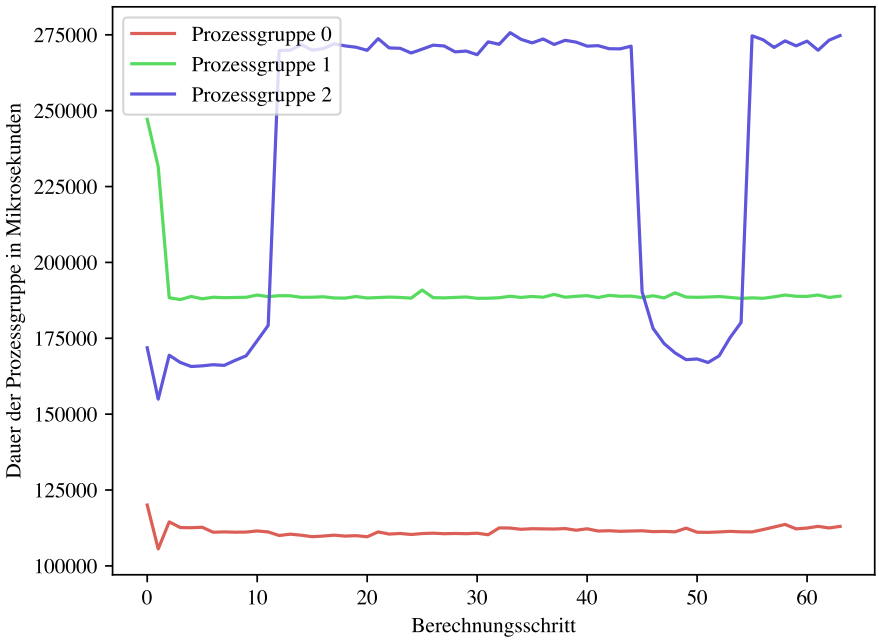

Abbildung 4.2.: Dauer der Prozessgruppen

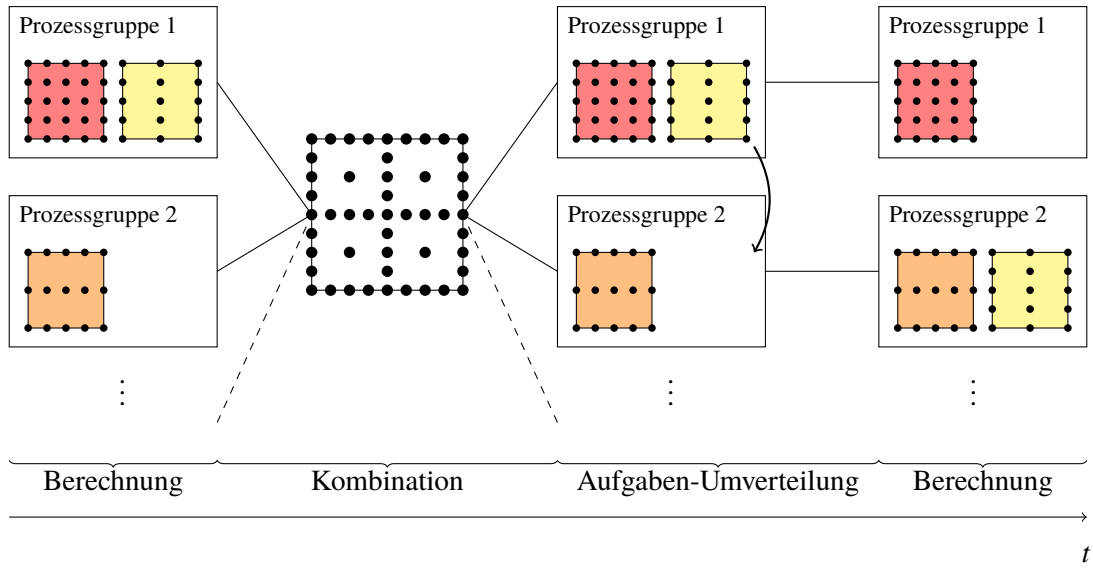

Abbildung 4.3.: Schritte der Ausführung des DisCoTec-Frameworks mit zusätzlicher Aufgaben-Umverteilung

4.3. Dynamische Aufgaben-Umverteilung

Die dynamische Aufgaben-Umverteilung wird nach dem Kombinationsschritt angesetzt (siehe Abbildung 4.3). Der Grund dafür ist, dass im DisCoTec-Framework nach der Kombination jede Prozessgruppe das aktuelle kombinierte Dünngitter besitzt. Es kann nun eine Aufgabe umgezogen werden indem nur dazugehörige Levelvektor übermittelt werden muss. Den aktuellen Stand des Gitters kann die Prozessgruppe dann mithilfe ihrem Dünngitter aus dem Kombinationsschritt wiederherstellen.

Um zu entscheiden welche Aufgaben umgezogen werden, wird der von [Hee18] vorgeschlagene Algorithmus Algorithmus 4.1 benutzt. Dazu passen wir den Algorithmus folgendermaßen an, dass eine Prognosefunktion statt der aktuellen Aufgabenberechnungsdauer benutzt wird. In dem Algorithmus 4.1 gibt es zwei Stellen an denen die prognostizierte zukünftige Laufzeit benutzt werden könne.

1. Ersetze die gegebenen Laufzeiten der Prozessgruppen g_i durch Schätzungen der Prognosefunktion.
2. Ersetze die in Zeile 9 gegebenen Aufgabenlaufzeiten durch die Werte der Prognosefunktion.

Für diese Arbeit wurde ausschließlich der zweiten Punkt umgesetzt. Der Nachteil ist allerdings, dass wenn für den nächsten Schritt andere Laufzeiten Prognostiziert werden, in diesem Schritt allerdings noch eine Perfekte Lastbalancierung vorhanden ist, noch keine Aufgabenumverteilung stattfinden kann. Dafür ist allerdings vorausgesetzt, dass ein entsprechend ausgefeilten Algorithmus benutzen wird um eine sinnvolle Umsortierung zu erzielen. Warum der erste Punkt trotzdem nicht implementiert wurde ist, dass die Implementierung dadurch einfacher ausfällt und für die in dieser Arbeit benutzten (simplen) Prognosefunktionen es der bessere Ansatz ist.

Für die Prognose zukünftiger Laufzeiten wird in dieser Arbeit der Durchschnitt der letzten n Laufzeiten, wobei $n \in \mathbb{N}^+$ beliebig gewählt werden kann, benutzt.

Die dynamische Aufgaben-Umverteilung wurde auf dem Stand des DisCoTec-Framework Git-Repositorys zu Git-Commit `1980262` implementiert. Die Implementierung der dynamischen Aufgaben-Umverteilung ist ein zusätzliches, optional angebotenes Feature. Es wurde versucht so wenig Folgeänderungen wie möglich nach sich zu ziehen und die Seiteneffekte so klein wie möglich zu halten. Ein Rescheduling wird manuell über den `ProcessManager` gestartet; identisch dazu, wie ein Berechnungsschritt oder eine Kombination gestartet wird. Das Rescheduling ruft den angepassten Algorithmus 4.1 mit der entsprechenden Prognosefunktion auf. Bei der Prognosefunktion muss beachtet werden, dass die Ausgabe die richtige Zeiteinheit besitzt, sodass die Wahl auf einen korrekten Task fallen kann, der nach der Umverteilung dazu beiträgt, dass die Prozessgruppen näher am Durchschnitt liegen. Die `ProcessManager`-Klasse benutzt neu hinzugefügte Funktionen des `ProcessGroupManagers`, welche das Entfernen und Hinzufügen von Tasks mit der rescheduling Semantik erlauben. Den `ProcessGroupWorker` werden entsprechende Signale hinzugefügt entsprechende Tasks hinzuzufügen und zu entfernen. Dabei müssen jeweils nur die Information des Levelvektors übergeben werden, der eindeutig einen Task referenziert. Der `ProcessGroupWorker` kann basierend auf den empfangenen Levelvektor den Task entweder entfernen oder eine Task mit dem vom Levelvektor beschriebenen Gitter kreieren. Das Interface des `Task` musste erweitert werden: Bei der Initialisierung wird die Anzahl der vergangenen Kombinationsschritte mitgegeben. Damit wird sichergestellt, dass der Task z. B. die Randbedingungen auf den korrekten aktuellen Zeitpunkt setzen kann.

4.4. Ziele und Auswertung

Das Grundlegende Ziel ist die Reduzierung der Simulationsdauer. Dazu muss die Simulation beschleunigt werden. Das Ziel einer guten Taskverteilung ist die Maximierung der Effizienz während des Berechnungsschrittes. Eine hohe Effizienz bedeutet, dass Prozessgruppen möglichst wenig aufeinander warten müssen.

Eine der wichtigsten Metriken ist die Gesamtdauer der Berechnungsschritte. Diese ergibt sich aus der Summe der Prozessgrupppenlaufzeit des jeweils langsamsten Prozessgruppe des Zeitschritts. Da die Laufzeiten der Tasks nicht abhängig der zugewiesenen Prozessgruppe ist, kann eine Gesamtdauer der Berechnungsschritte mit dem statischen Taskverteilung angenähert werden. Zusätzlich werden die Laufzeiten der Schritte „Berechne“, „Kombiniere“ und „Umverteilung der Tasks“ aufgespalten. In Abschnitten mit Laufzeitmessungen wurde sichergestellt, dass jegliche Output-Befehle entfernt wurde.

Bei der dynamische Aufgaben-Umverteilung gibt es die folgenden Parameter:

1. Lastinbalance ϵ
2. Maximale Iterationen $k_{\max}$
3. Die Wahl der Prognose

5. Ergebnisse

Bei den hier vorgestellten Beispielen werden Simulationen mit Tasks, mit variabler Laufzeit über die Zeit, präsentiert.

5.1. Beispiel 1

In dem Abschnitt 4.2 wurde ein Beispiel mit kleiner Anzahl an Tasks vorgestellt. Dabei wurden die Probleme der Verwendung von statische Lastverteilung demonstriert. Für dieses Beispiel wird die dynamische Aufgaben-Umverteilung aktiviert.

5.1.1. Parameter

Die für die Lastumverteilung gewählten Parameter sind:

- $k_{\max} = 10$
- $\epsilon = 0{,}05$

Die Prognosefunktion gibt die Dauer des letzten Durchlaufs zurück.

5.1.2. Messdaten

Für die hier erhobenen Daten wurde die selbe Simulation, wie in dem Abschnitt 4.2, durchgeführt. Vergleicht man die erhobenen Task Laufzeiten Abbildung 5.1 mit Abbildung 4.1 erkennt man, dass der der Unterschied der Daten sehr klein ist und der Allgemeine Trend identisch ist. Es wurde nun die implementierte dynamische Aufgaben-Umverteilung benutzt um die Effizienz der Ausführung zu erhöhen. Anhand der Prozessgruppenlaufzeit kann die Effizienz der Berechnung bewertet werden: Um so geringer der Abstand der Prozessgruppenlaufzeiten um so näher ist jede der Laufzeiten dem optimalen Durchschnitt. Vergleicht man die Prozessgruppenlaufzeiten Abbildung 5.2 und Abbildung 4.2 ist eine deutlich effizientere Ausführung von ungefähr Zeitschritt 15 bis 45 erkennbar. In den Bereichen direkt davor und danach ist keine bessere Verteilung möglich, da die Berechnungsdauer abhängig des langsamsten Task 5,8 ist. Da der Task 5,8 der einzige Task der Prozessgruppe ist kann die Laufzeit der Prozessgruppe nicht durch Aufgaben-Umverteilung verbessert werden. Die Peak der Laufzeit von Prozessgruppe 2 in Berechnungsschritt 12 ist aufgrund der in dieser Arbeit gewählten Implementierung und des Prognosemodells nicht vermeidbar.

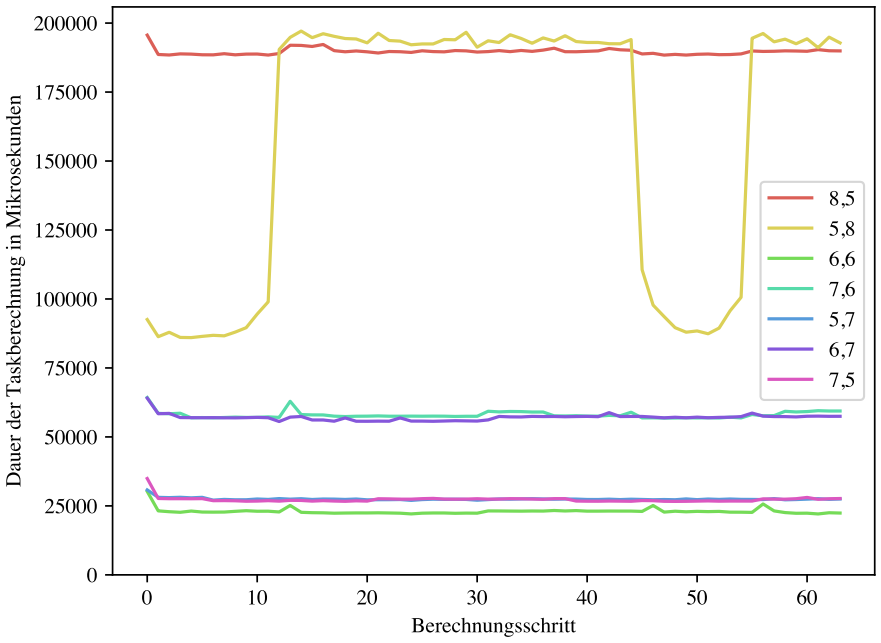

Abbildung 5.1.: Messdaten der Aufgabenlaufzeit des Beispiels 1

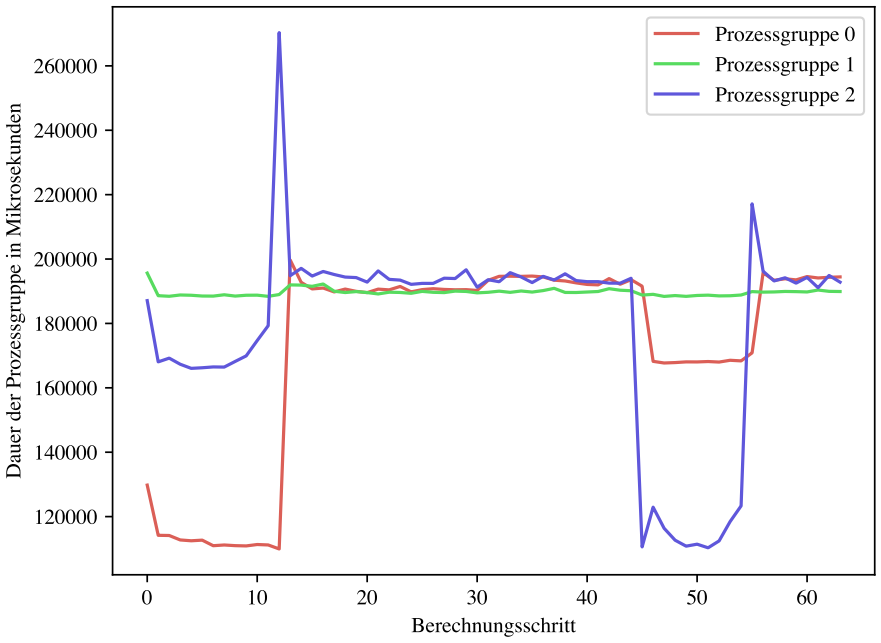

Abbildung 5.2.: Messdaten der Prozessgruppenlaufzeit des Beispiels 1

Die Dauer der aufsummierten Berechnungsschritten mit dynamischer Aufgaben-Umverteilung liegt bei 12.420.214 Mikrosekunden. Falls dieselben Task Laufzeiten mit der ursprünglichen Verteilung durchgelaufen wären, so läge die Zeit die Zeit für die Berechnung bei 15.704.883 Mikrosekunden. Daraus folgt, dass mithilfe von dynamischer Aufgaben-Umverteilung die Zeit für den Berechnungsschritt um $\approx 20\%$ beschleunigt werden konnte.

Betrachtet man die Dauer der der einzelnen Schritte, so überwiegt der Berechnungsschritt deutlich die anderen Schritte. Die Berechnung nimmt ca. 180.000 bis 200.000 Mikrosekunden pro Berechnungsschritt ein. Der Kombinationsschritt benötigt dagegen nur 1.100 bis 1.500 Mikrosekunden Die Zeit für das Ausführen des Umverteilungsalgorithmus ist vernachlässigbar; er liegt maximal im niedrigen zweistelligen Bereich in Mikrosekunden. Die Migration benötigt linear für jeden umziehenden Task ca. 4.000 Mikrosekunden im Verlauf der Simulation wurde allerdings nur 4 mal ein Task migriert.

5.2. Beispiel 2

Es werden die Daten einer Größeren Simulation dargestellt. Dazu wurden im Vergleich zu Abschnitt 5.1 deutlich mehr Tasks mit höheren Auflösungen und feineren Berechnungsschritten gewählt.

5.2.1. Parameter

Die für die Lastumverteilung gewählten Parameter sind:

- $k_{\max} = 10$
- $\epsilon = 0{,}05$

Die Prognosefunktion gibt die Dauer des letzten Durchlaufs zurück.

5.2.2. Messdaten

Die Laufzeit der Tasks Abbildung 5.3 variieren über die Zeit deutlich. Es ist auch deutlich zu sehen, dass in der Mitte und am Ende die Laufzeit ein Maximum annimmt. Die schnell zu berechnenden Tasks sind im Vergleich zu den Langsamen so gut wie konstant. Die Dauer der Prozessgruppen Abbildung 5.4 deutlich der Entwicklung der langsamen Tasks abhängig. Die Unterschiede in Prozessgruppenlaufzeit liegt bei maximal ca. 50.000 Mikrosekunden, identisch zu den schnellsten Tasks.

Die Dauer der aufsummierten Berechnungsschritten mit dynamischer Aufgaben-Umverteilung liegt bei 167.366.746 Mikrosekunden. Falls dieselben Task Laufzeiten mit der ursprünglichen Verteilung durchgelaufen wären, so läge die Zeit für die Berechnung bei 212.305.355 Mikrosekunden. Daraus folgt, dass mithilfe von dynamischer Aufgaben-Umverteilung die Zeit für den Berechnungsschritt um $\approx 20\%$ beschleunigt werden konnte.

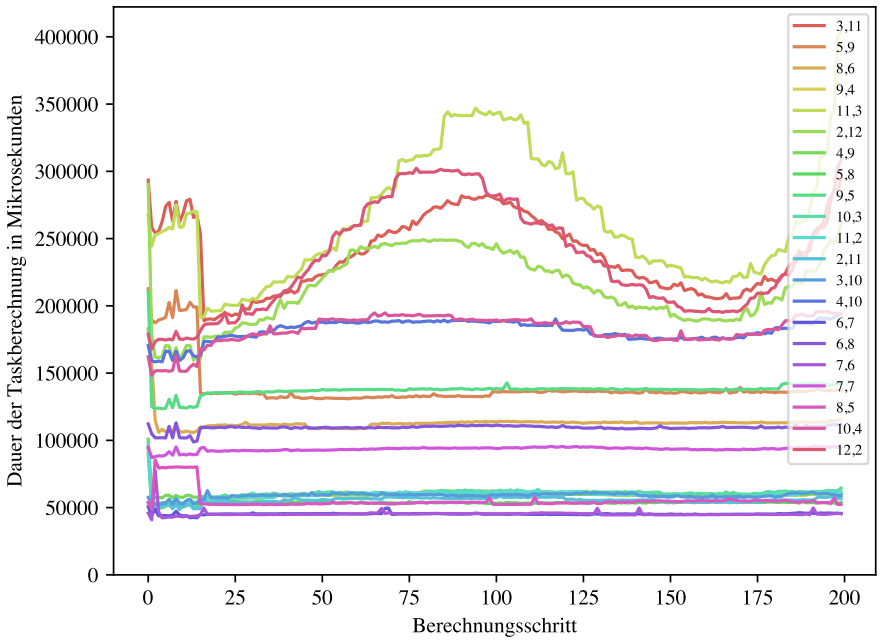

Abbildung 5.3.: Messdaten der Aufgabenlaufzeit des Beispiels 2

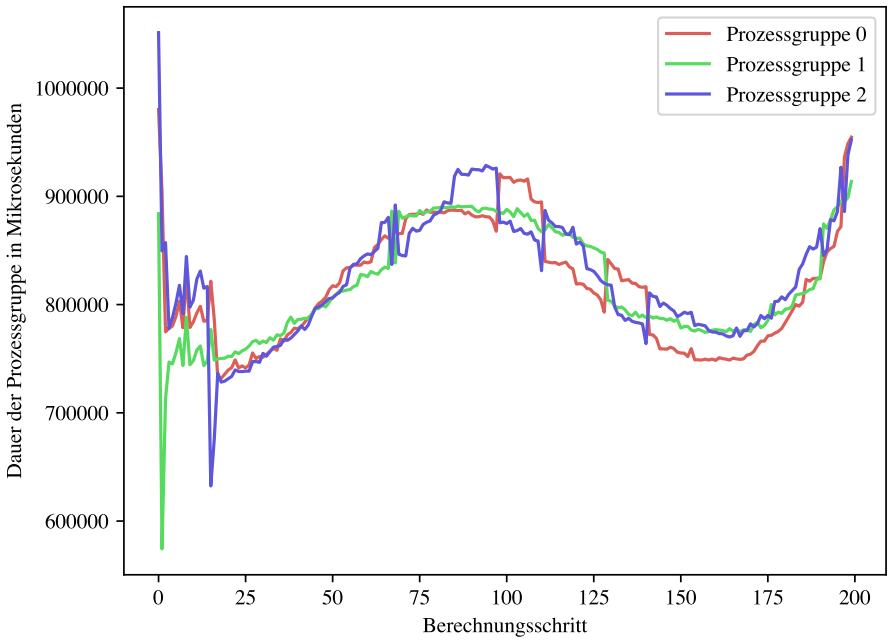

Abbildung 5.4.: Messdaten der Prozessgruppenlaufzeit des Beispiels 2

Die Laufzeit der Berechnung überwiegt die anderen Schritten. Die Berechnung benötigte ca. 770.000 bis 900.000 Mikrosekunden. Die Kombinationen lagen bei ca. 10.000 Mikrosekunden und die Berechnung für die Aufgaben-Umverteilung bei ca. 20 bis 40 Mikrosekunden. Die Task Migration skalierte Linear und benötigte ca. 6.000 Mikrosekunden pro Task. Es wurden am Anfang der Simulation ein Maximum von 4 Tasks in einer Umverteilung migriert. Insgesamt wurden in 14 Berechnungsschritten mindestens ein Task umverteilt und migriert.

5.3. Beispiel 3

In diesem Beispiel wird die Advektion in einem dreidimensionalem Raum gelöst. Die Laufzeiten liegen im Sekundenbereich und sind somit deutlich größer als bei den vorherigen Beispielen.

5.3.1. Parameter

Die für die Lastumverteilung gewählten Parameter sind:

- $k_{\max} = 10$
- $\epsilon = 0{,}05$

Die Prognosefunktion gibt die Dauer des letzten Durchlaufs zurück.

5.3.2. Messdaten

Die Laufzeiten der Tasks sind in Abbildung 5.5 sichtbar. Die langsamen Tasks haben die größte Veränderung der Laufzeit über die Simulation. Die Laufzeit Veränderung der langsamen Tasks dauert über mehrere Zeitschritte an. Bei den schnelleren Tasks ist eine große Anzahl an kurzfristigen Peaks zu erkennen. Im Vergleich zu den langsamen Tasks sind die schnellen so gut wie konstant. Abbildung 5.6 zeigt die Laufzeit der Prozessgruppen.

Die Dauer der aufsummierten Berechnungsschritten mit dynamischer Aufgaben-Umverteilung liegt bei 3.405.083.003 Mikrosekunden. Falls dieselben Task Laufzeiten mit der ursprünglichen Verteilung durchgelaufen wären, so läge die Zeit für die Berechnung bei 4.306.977.147 Mikrosekunden. Daraus folgt, dass mithilfe von dynamischer Aufgaben-Umverteilung die Zeit für den Berechnungsschritt um $\approx 20\%$ beschleunigt werden konnte.

Die Laufzeit der Berechnung überwiegt die anderen Schritten. Die Berechnung benötigte ca. 30.000.000 bis 37.000.000 Mikrosekunden. Die Kombinationen lagen bei ca. 50.000 bis 60.000 Mikrosekunden und die Berechnung für die Aufgaben-Umverteilung bei ca. 20 bis 50 Mikrosekunden. Die Task Migration skalierte Linear und benötigte ca. 80.000 Mikrosekunden pro Task. Es wurden öfter mehrere Tasks auf einmal umverteilt. Das Maximum an Umverteilungen war 4 Tasks. Es wurden auch mehrmals 3 Tasks gleichzeitig umverteilt. Insgesamt wurden 53 Tasks über die gesamte Zeit umverteilt. Insgesamt wurden in 34 Berechnungsschritten mindestens ein Task umverteilt und migriert.

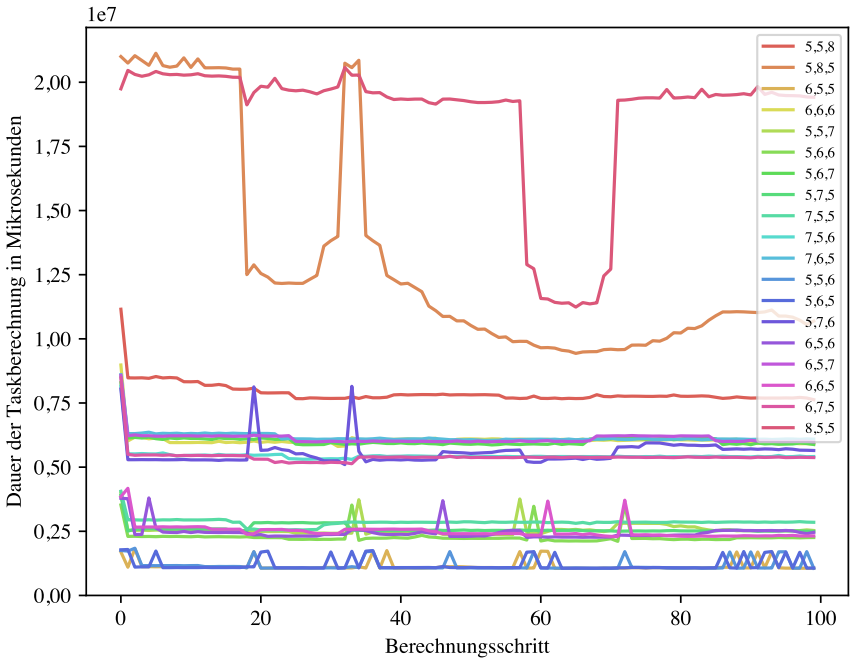

Abbildung 5.5.: Messdaten der Aufgabenlaufzeit des Beispiels 3

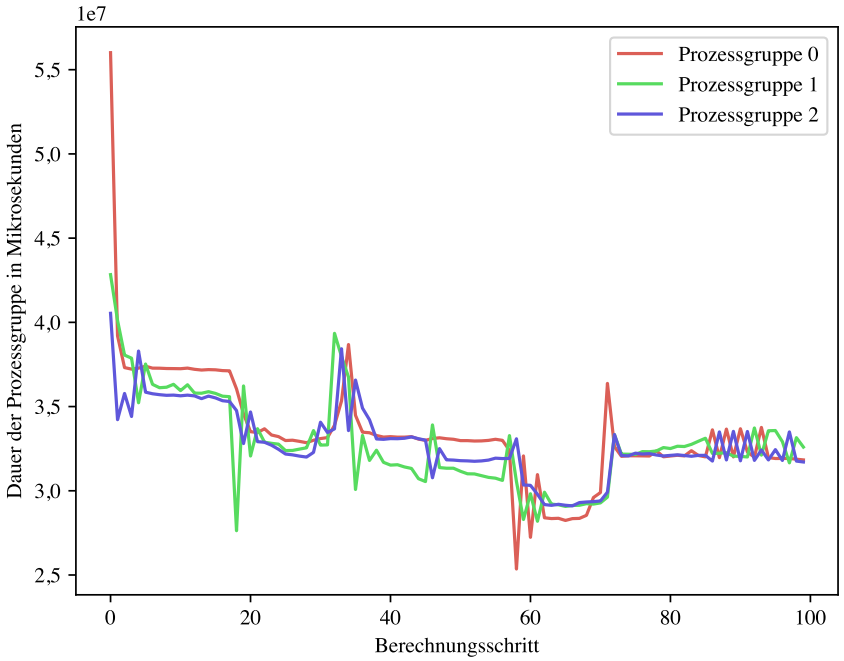

Abbildung 5.6.: Messdaten der Prozessgruppenlaufzeit des Beispiels 3

6. Diskussion und Fazit

Die Wahl der Parameter der dynamischen Aufgaben-Umverteilung sollte nach Erfahrung mit der entsprechenden Simulation getroffen werden. So kann z. B. durch das Betrachten der Laufzeiten der Tasks bereits genügend Erkenntnisse gesammelt werden, die für die Wahl sinnvoller Parameter nötig sind.

Ist das ϵ zu klein gewählt, so wird immer Versucht eine bessere Verteilung zu erzielen. Dabei wird eine bessere Verteilung nur dann erzielt werden können, wenn es entsprechenden Tasks zum Umverteilen gibt. Insbesondere wenn viele sehr schnelle Tasks vorhanden sind sollte das ϵ nicht zu klein gewählt werden, da immer Tasks für die Lastverteilung gefunden werden und die Kosten der Migration zu groß im Vergleich zu der verbesserten Lastverteilung sein wird.

Die Anzahl an Iteration $k_{\max}$ ist die obere Grenze für die Anzahl der in einem Umverteilungsschritt möglichen Umverteilungen. In den hier gezeigten Beispielen gab es allerdings nie eine große Anzahl an Tasks, die Umverteilt werden sollten. In größeren Simulationen mit mehr Zeitschritten können die Umverteilungen außerdem ohne allzu große Probleme über mehrere Berechnungsschritte verteilt werden: Über zwei Berechnungsschritte hat mit einem $k_{\max}$ ist es schon möglich 20 Tasks Umzuverteilen Ein kleines $k_{\max}$ hat allerdings den Vorteil, dass die maximale Laufzeit des Algorithmus beschränkt wird.

Für die hier gewählte Prognosefunktion, die einen Durchschnitt aus den letzten n Zeitschritte berechnet, muss dieses n entsprechend gesetzt werden. Ein kleines n, wie $n = 1$, kann gewählt werden, sodass der aktuellste Wert der Laufzeit verwendet wird. Falls die Laufzeit der Aufgaben stark schwankt kann ein größeres n zu einem besseren Ergebnis führen.

Nachteile Für Aufgaben mit langen Initialisierungszeiten ist eine Umverteilung nur unter Umständen sinnvoll. Um so mehr `Tasks` mit langen Initialisierungszeiten umverteilt werden um so schlechter. Man kann nun allerdings stattdessen nicht nach jedem Kombinieren, sondern nach jedem 10ten Kombinieren erst immer die Lastumverteilung laufen lassen. Wenn sich die Laufzeiten sehr rapide ändern und dabei kein Muster ergibt, welches sich durch die Berechnung des Durchschnitts betrachten lässt, so ist eine sinnvolle Lastumverteilung nur schwer möglich.

Vorteile Wie in den Beispielhaften Messdaten zu sehen ist eine Steigerung der Effizienz unter benutzen von Aufgaben mit veränderbaren Laufzeiten im Vergleich zu dem Statischen deutlich größer. Vorteile einer dynamischen Aufgeben-Umverteilung ist allgemein, dass eine schlechte Lastverteilung im nächsten Schritt wieder korrigiert werden kann und so die Varianz der Laufzeit deutlich kleiner ist.

6.1. Ausblick

Die dynamische Aufgaben-Umverteilung half in den Beispielen dieser Arbeit die Effizienz des Berechnungsschrittes von Simulationen zu erhöhen. Es gibt allerdings noch mögliche Stellen der Verbesserung. Zum einen ist das Prognosemodell in dieser Arbeit sehr simpel gehalten. Der benutzte Scheduling-Algorithmus ist zwar in den hier gezeigten Beispielen sehr schnell, er liefert allerdings oftmals keine optimal Verteilung. Eventuell wäre es nützlich bei der Taskmigration die alten Werte der Gitterpunkte mitzumigrieren, z. B. wenn während der Kombination nicht alle Punkte in die Kombination miteinbezogen werden. Für weitergehende Resultate wäre es außerdem interessant die hier vorgeschlagene dynamische Aufgaben-Umverteilung auf Hochleistungsrechner mit einem entsprechend dimensioniertem Problem zu testen.

Es ist allgemein festzuhalten, dass die dynamische Aufgaben-Umverteilung im Zusammenhang mit über die Zeit ändernder Aufgabenlaufzeiten eine gesteigerte Effizienz im Berechnungsschritt liefert, zusätzlich haben sich keine bemerkbaren Nachteile, wie hohe Laufzeitkosten, gezeigt.

Literaturverzeichnis

[BBD+16] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, O. Sander. „The Distributed and Unified Numerics Environment, Version 2.4“. eng. In: *Archive of Numerical Software* Vol 4 (2016). DOI: 10.11588/ANS.2016.100.26526 (zitiert auf S. 35).

[Bel15] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, 2015. ISBN: 0691625859 (zitiert auf S. 15).

[BZBP13] H.-J. Bungartz, S. Zimmer, M. Buchholz, D. Pflüger. *Modellbildung und Simulation*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-37656-6 (zitiert auf S. 35).

[EMGB45] J. J. P. Eckert, J. W. Mauchly, H. H. Goldstine, J. G. Brainerd. *Description of the ENIAC and Comments on Electronic Digital Computing Machines*. MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA, 30. Nov. 1945. URL: https://apps.dtic.mil/docs/citations/AD0498867 (zitiert auf S. 15).

[Gar12] J. Garcke. „Sparse Grids in a Nutshell“. In: *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2012, S. 57–80. DOI: 10.1007/978-3-642-31703-3_3 (zitiert auf S. 17).

[GNW10] D. Goldsman, R. E. Nance, J. R. Wilson. „A Brief History of Simulation Revisited“. In: *Proceedings of the Winter Simulation Conference*. WSC '10. Baltimore, Maryland: Winter Simulation Conference, 2010, S. 567–574. ISBN: 978-1-4244-9864-2. URL: http://dl.acm.org/citation.cfm?id=2433508.2433574 (zitiert auf S. 15).

[Gri92] M. Griebel. „THE COMBINATION TECHNIQUE FOR THE SPARSE GRID SOLUTION OF PDE'S ON MULTIPROCESSOR MACHINES“. In: *Parallel Processing Letters* 02.01 (01 März 1992), S. 61–70. DOI: 10.1142/s0129626492000180 (zitiert auf S. 17).

[Hee18] M. Heene. „A massively parallel combination technique for the solution of high-dimensional PDEs“. en. Diss. Institut für Parallele und Verteilte Systeme der Universität Stuttgart, 2018. DOI: 10.18419/opus-9893 (zitiert auf S. 15–17, 23, 29, 33, 34, 37).

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. 4. Juni 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (zitiert auf S. 26).

[MPI] MPICH. *MPICH*. URL: https://www.mpich.org/ (zitiert auf S. 26).

[Ope] Open MPI. *Open MPI*. URL: https://www.open-mpi.org/ (zitiert auf S. 26).

[SG] SG++. *DisCoTec*. URL: https://github.com/SGpp/DisCoTec (zitiert auf S. 29).

[TOP] TOP500. *TOP500*. URL: https://www.top500.org/ (zitiert auf S. 15).

[Val19] J. Valentin. „B-Splines for Sparse Grids: Algorithms and Application to Higher-Dimensional Optimization“. en. Diss. Universität Stuttgart, 2019. DOI: 10.18419/OPUS-10504 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 08. 01. 2020 geprüft.

A. Benutztes Testsystem

A.1. Hardware

Die hier vorgestellten Daten wurden mithilfe des Fachbereichs Simulation großer Systeme (SGS) des IPVS internen Berechnungs-Cluster erhoben. Der Clusters besteht aus 16 Rechner, welche jeweils eine vier-Kern Intel Xeon E3-1585v5 CPU und 32GB Arbeitsspeicher besitzen Die Rechner sind mithilfe von 10Gbit Ethernet zusammengeschlossen.

A.2. Software

Das auf dem Cluster installierte Betriebssystem war Ubuntu 16.04.6 LTS. Es wurde der Slurm Workload Manager zum Starten der Simulationssoftware verwendet.

B. Genaue Parameter der Plots

Runscript Das benutzte Runscript, welches mithilfe von `sbatch` zum Starten der Simulation benutzt wurde:

```
#!/bin/bash
#SBATCH --job-name="combi"
#SBATCH --ntasks=4
#SBATCH --threads-per-core=1

srun --mpi=pmi2 ./advection-example
```

`settings.ini` der Plots

- Abbildung 4.1; Abbildung 4.2; Abbildung 5.1; Abbildung 5.2:

```
[manager]
ngroup = 3
nprocs = 1

[simulation]
time_step = 0.03125
time_start = 0.0
time_end = 2.000

[ct]
lmin = 5 5
lmax = 8 8
leval = 4 4
procs = 1 1
ncombi = 64
```

- Abbildung 5.3; Abbildung 5.4:

```
[manager]
ngroup = 3
nprocs = 1

[simulation]
time_step = 0.01
time_start = 0.0
time_end = 2.000

[ct]
lmin = 2 2
lmax = 12 12
leval = 4 4
procs = 1 1
ncombi = 200
```

- Abbildung 5.5; Abbildung 5.6:

```
[manager]
ngroup = 3
nprocs = 1

[simulation]
time_step = 0.02
time_start = 0.0
time_end = 2.000

[ct]
lmin = 5 5 5
lmax = 8 8 8
leval = 4 4 4
procs = 1 1 1
ncombi = 100
```

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