

Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit

Optimierung in und von Partitionierungsproblemen in der Molekulardynamik

Dennis Kreittner

Studiengang: Informatik B.Sc.
Prüfer/in: Prof. Dr. Dirk Pflüger
Betreuer/in: Steffen Hirschmann, M.Sc.

Beginn am: 5. August 2019
Beendet am: 5. Februar 2020

Kurzfassung

Um reale Anwendungen in der Molekulardynamik betrachten zu können muss das Simulationsgebiet in den meisten Fällen auf mehrere Prozessoren aufgeteilt und damit parallelisiert werden. Dieses Partitionierungsproblem wird in der Literatur primär durch Heuristiken gelöst. In dieser Arbeit betrachten wir die Problemstellung durch einen theoretischen Zugang als Optimierungsproblem. Dazu wird aus dem Problem PARTITION und den in der Literatur verwendeten Zielfunktionen und Maßen ein Modell erstellt. Dieses minimiert die Abweichung von der mittleren Partikelzahl pro Prozessor und wird in zwei Prototypen implementiert. Die Programme lösen mit den stochastischen Ansätzen Simulated Annealing und genetischen Algorithmen das Optimierungsproblem. Mit Datensätzen aus zwei Rußpartikelsimulationen und verschiedenen Einstellungen werden Testläufe für die Prototypen vorgenommen. Dabei variieren wir die Anzahl der Prozessoren, Partikelverteilung und andere Parameter. Die entstehenden Partitionierungen werden bezüglich der Abweichung von der mittleren Partikelzahl und ihrer Imbalance miteinander verglichen. Außerdem betrachten wir am Schluss die Form der entstehenden Partitionen und ob diese zusammenhängend sind.

Inhaltsverzeichnis

1	Einleitung	7
1.1	Related work	7
2	Grundlagen	9
2.1	Parallelisierung und Partitionierung	9
2.2	PARTITION	10
2.3	Heuristiken	11
2.4	Stochastische Verfahren zur Lösung von Optimierungsproblemen	14
3	Literaturrecherche und Modell	17
3.1	Modellbildung	19
3.2	Löser für Optimierungsprobleme	21
4	Prototypen	23
4.1	Prototyp für Simulated Annealing	23
4.2	Prototyp für genetische Algorithmen	27
5	Ergebnisse	35
5.1	Ergebnisse mit Simulated Annealing	35
5.2	Ergebnisse mit genetischen Algorithmen	41
5.3	Visualisierung der Partitionen	46
6	Zusammenfassung und Ausblick	49
	Literatur	51

1 Einleitung

Die Molekulardynamik ist ein aktuelles Themengebiet in der Forschung. Viele atomare und molekulare Problemstellungen können damit simuliert werden. Dabei betrachtet man die Teilchen eines Simulationsgebiets einzeln und modelliert ihre Interaktion mit anderen aufgrund von Kräften. Der Rechenaufwand realer Anwendungen ist durch hohe Teilchenzahlen entsprechend hoch. Eine naheliegende Lösung ist den Rechenaufwand zu verteilen, also das Problem zu parallelisieren. Aufgrund der Größenordnungen im atomaren Bereich werden in vielen realen Anwendungen dennoch eine große Anzahl an Prozessoren notwendig sein. Die sinnvolle und vor allem kostengünstige Aufteilung des Rechenaufwands ist dabei das Ziel. In der Molekulardynamik sind Berechnungskosten direkt verknüpft mit dem Simulationsgebiet und den darin enthaltenen Teilchen. Daher wird bei einer Parallelisierung das Gebiet in räumliche Subgebiete unterteilt. Dabei kommen viele Faktoren zusätzlich zum reinen Rechenaufwand der Molekülinteraktion ins Spiel, die eine Partitionierung beeinflussen. Kommunikationskosten zwischen Prozessoren, die Form der Gebiete, das Oberflächen-Volumen-Verhältnis von Partitionen oder die Imbalance der Partitionierung sind einige Konzepte. Das Problem, das eine Aufteilung der Gebiete beschreibt nennt man Partitionierungsproblem.

In der Literatur zum Partitionierungsproblem der Molekulardynamik werden oft Heuristiken zur Lösung eingesetzt. In dieser Arbeit führen wir allerdings eine theoretische Betrachtung als Optimierungproblem durch. Zunächst wird mit einer Literaturrecherche zu verwendeten Modellen, Zielfunktionen und Maßen bei Partitionierungen begonnen. Mit den Erkenntnissen davon und einer Verbindung zum NP-vollständigen Problem `PARTITION` wird ein Modell erstellt, das die Anzahl der Partikel als Maß verwendet. Ziel davon ist es die Abweichung der Prozessoren von der mittleren Partikelzahl zu minimieren.

Anschließend wird das Modell in zwei Prototypen implementiert. Diese verwenden die stochastischen Ansätze Simulated Annealing und genetische Algorithmen und sollen später zur Bewertung der Qualität anderer Partitionierer dienen. Für Simulated Annealing implementieren wir eine Abbruchbedingung basierend auf der Imbalance der Partitionierung. Des Weiteren werden für die genetischen Algorithmen speziell auf die Molekulardynamik angepasste Operatoren definiert. Mit Datensätzen aus zwei Rußpartikelsimulationen führen wir mehrere Testläufe der Prototypen durch. Dabei variieren wir die Prozessorzahl, Partikelverteilung, Operatoren und andere Parameter. Die sich daraus ergebenden Partitionen werden bezüglich ihrer Abweichung von der mittleren Partikelzahl und Imbalance graphisch miteinander verglichen. Zum Schluss werden die Partitionen visualisiert um den Zusammenhang der Gebiete zu veranschaulichen.

1.1 Related work

In diesem Abschnitt werden Arbeiten erwähnt, die sich hauptsächlich mit Lastbalancierung und Partitionierung in der Molekulardynamik beschäftigt haben. In [WGW06] wird ein Algorithmus zur Lösung von Problemen des elektromagnetischen Felds vorgestellt. Dabei verwenden die Autoren die

Bisektion zur Lastbalancierung. Boillat et al. stellen in [BB91] einen Algorithmus zur Lastbalancierung mit Diffusion für die Simulation spinodaler Zerlegungen vor. Dabei werden die verwendeten Modelle angelehnt an [CG89]. Catalyurek et al. verwenden in [CB09] Knoten-basierte Graphpartitionierung um Lastbalancierung durch Repartitionierung von Hypergraphen durchzuführen. Begau et al. betrachten in [BS15] mit der Gitter-basierten Partitionierung vor allem unregelmäßige Gebiete. Darüber hinaus benutzen Xiaolin et al. in [XZ04] Hilbert-Kurven und Bisektion um eine skalierbare Methode zur Partitionierung der Gebiete in der Molekulardynamik zu erhalten. Buchholz beschreibt in [BU10] ein Framework, das hochparallele Molekulardynamik ermöglicht. Dazu werden mehrere Heuristiken als Strategien definiert und diese auf heterogenen Partikelverteilungen verwendet. Dieses Vorgehen ist sehr eng mit dem dieser Arbeit verwandt, da ebenso Verfahren auf heterogenen Partikeln verglichen werden. Wir nutzen lediglich einen theoretischen Zugang zum Partitionierungsproblem und keine Heuristiken.

2 Grundlagen

In der Molekulardynamik, nachzulesen zum Beispiel in [BZ13] Kapitel 13, werden Moleküle bzw. Atome, deren Wechselwirkungen und Bewegungen simuliert. Dabei bestimmen vier grundlegende physikalische Kräfte (Gravitationskraft, elektromagnetische Kraft, starke und schwache Kernkraft) die Interaktionen und Trajektorie der Teilchen. Um diese Kräfte zu beschreiben benutzt man in den meisten Fällen Paar-Potentiale, die die Kräfte zwischen zwei Teilchen i und j in Abhängigkeit vom Abstand r_{ij} modellieren. Oftmals werden zur Vereinfachung nicht alle oben genannten Kräfte in einem Potential berücksichtigt. Dies ist auch beim häufig in der Literatur verwendeten Lennard-Jones-Potential U_{LJ} der Fall, welches sich aus van-der-Waals-Kraft und Pauli-Repulsion zusammensetzt. Die Faktoren ϵ und σ repräsentieren die Energie und Größe von Teilchen.

$$U_{LJ}(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^6 \right) \quad (2.1)$$

Das in Gleichung (2.1) dargestellte Lennard-Jones-Potential ist mit Faktoren r_{ij}^{-6} und r_{ij}^{-12} vom Abstand der beiden betrachteten Teilchen abhängig. Deshalb verliert das Potential mit großer Distanz schnell an Wirkung. Das heißt es kann ein Abschneideradius r_c definiert werden, nach dessen Überschreitung das Potential vernachlässigbar ist. Dies hat wiederum zur Folge, dass sich die Anzahl der Teilchen, die für die Berechnung von Kräften relevant sind drastisch reduziert. Vor dieser Erkenntnis mussten noch alle Teilchen der Simulation in Betracht gezogen werden, $O(n)$ bei n Teilchen. Inzwischen sind es nur noch die Partikel in einem Abstand von r_c , also $O(1)$ Teilchen. Dies verbessert vor allem die Laufzeit der Berechnungen. Außerdem erlaubt dieser Radius ein Gitter, auch Linked-Cells genannt, über das betrachtete Gebiet zu legen. Idealerweise werden die Gitterabstände entsprechend dem Abschneideradius gewählt wodurch sich die Suche nach Partikeln, die bei der Wechselwirkung relevant sind, auf die benachbarten Zellen im Gitter beschränkt. Alle Partikel außerhalb dieser Nachbarregion sind nicht von Bedeutung, da das mit ihnen bestehende abgeschnittene Paar-Potential Null ist.

2.1 Parallelisierung und Partitionierung

Es wird ein Gebiet betrachtet, das in Zellen mit der Größe r_c aufgeteilt ist. Da sich bereits innerhalb kleiner Volumeneinheiten sehr viele Teilchen befinden können (vgl. 1 Mol $\approx 6 \cdot 10^{23}$ Teilchen) ist das maximal simulierbare Gebiet durch die verfügbare Rechenleistung stark eingeschränkt. Eine naheliegende Lösung für dieses Problem ist die Parallelisierung des Rechengangs. Das heißt für unsere Problemstellung, dass das Simulationsgebiet in mehrere Subgebiete aufgeteilt wird und diese dann auf verschiedene Prozessoren verteilt werden. Dieses Partitionierungsproblem orientiert sich bei der Aufteilung der Gebiete an einem Maß, das in einer Funktion beschrieben wird. Maße können dabei Rechenzeit, Anzahl an Partikeln oder andere aus der Molekulardynamik abgeleitete Einheiten

sein. Zusätzlich zieht man oftmals die Form, Verteilung und den Zusammenhang der Gebiete in dieser Kostenfunktion in Betracht. Wenn Zellen verschiedenen Prozessoren zugewiesen sind, müssen diese zur Berechnung von Kräften miteinander kommunizieren. Dies verursacht zusätzliche, nicht vernachlässigbare Kosten, die bei der Zerlegung berücksichtigt werden. Die Lösung des Partitionierungsproblems ist eine Lastbalancierung, da der Berechnungsaufwand für die einzelnen Gebiete auf mehrere Prozessoren verteilt wird.

Allerdings kann die Lastbalancierung auch als eigenständiges, der Partitionierung übergeordnetes Problem betrachtet werden. Diese beschäftigt sich dann mit der Verteilung von Last auf Prozessoren, welche in generischen Einheiten wie „work-load“ angegeben wird. Die Verfahren zur Lastbalancierung lassen sich in statisch und dynamisch aufteilen. Jeder dieser Begriffe besitzt dabei eine doppelte Bedeutung. Statische Verfahren bestimmen vor der Laufzeit oder vor jedem Iterationsschritt eine Verteilung, dynamische Verfahren dagegen während der Laufzeit oder während jedem Iterationsschritt. In dieser Arbeit wird ausschließlich statisch partitioniert. In der späteren Literaturrecherche werden auch Paper zur reinen Lastbalancierung besprochen, da diese häufig mit ähnlichen Heuristiken arbeiten und die verwendeten Modelle für unsere Modellbildung von Nutzen sein können.

Die Imbalance einer Partitionierung beschreibt das Verhältnis zwischen dem am stärksten ausgelasteten Prozessor und der durchschnittlichen Last über alle Prozessoren P_1 bis P_k . Mit einer Funktion $\text{Last}(P_i)$, die die Last des Prozessors i beschreibt und der durchschnittlichen Last pro Prozessor mean ergibt sich für die Imbalance folgende Formel.

$$\text{imbalance} = \frac{\max_j \{\text{Last}(P_j)\}}{\text{mean}} \quad (2.2)$$

2.2 PARTITION

Die Ursprünge jedes Lastbalancierungs- oder Partitionierungsproblems liegen in PARTITION. Ein NP-vollständiges Problem aus der theoretischen Informatik. Die grundlegende Form (engl. „two-way number partitioning“) beschreibt die Aufteilung von ganzen Zahlen aus einer Menge N auf zwei Mengen M_1 und M_2 , so dass die Summen der Zahlen in den Mengen möglichst gleich sind. Es soll also

$$\left| \sum_{i \in M_1} i - \sum_{i \in M_2} i \right|$$

minimiert werden. Die komplexere Form davon ist das „ k -way number partitioning“, das wiederum eine Verteilung der Zahlen auf k viele Mengen beschreibt. Dies weist eine starke Ähnlichkeit mit der Verteilung von Last auf k Prozessoren oder auch der Partition eines Gebiets in k Subgebiete nach einem einfachen Maß auf. Das werden wir später in die Modellbildung einfließen lassen.

2.3 Heuristiken

Dieser Abschnitt umfasst die fünf hauptsächlich in der Literatur verwendeten Partitionierungsverfahren der Molekulardynamik. Allgemein können diese in globale und lokale Varianten unterteilt werden. Eine globale Methode teilt das komplette Simulationsgebiet in Partitionen auf, wohingegen in lokalen Verfahren zwischen benachbarten Gebieten Last ausgetauscht wird. In den meisten Fällen setzen lokale Methoden eine bestehende Partitionierung voraus.

Die Bisektion ist ein globales Partitionierungsverfahren bei dem das Gebiet rekursiv entlang der Achsen geteilt wird. Dabei wählt man in der Literatur vermehrt die momentan längste Achse des Gebiets zur Teilung. Die Zerlegung wird an einer Koordinate durchgeführt an der das zuvor definierte Maß beider Subgebiete möglichst gleich ist.

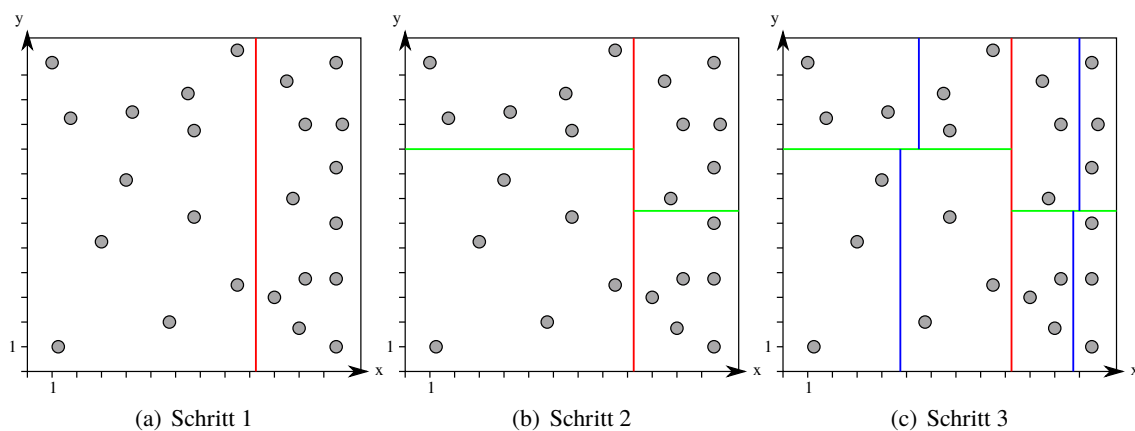


Abbildung 2.1: Beispiel für eine Bisektion

Abbildung 2.1 zeigt eine beispielhafte Bisektion mit der Anzahl an Partikeln als Maß. Zunächst wird in Schritt 1 die x-Achse durch eine rote Linie in zwei Subgebiete mit je 12 Partikeln geteilt. In Schritt zwei teilen die beiden grünen Linien die y-Achse so, dass 4 Subgebiete mit je 6 Partikeln entstehen. Schließlich erzeugen die blauen Linien 8 Subgebiete mit je 3 Partikeln. Wenn man abwechselnd entlang der Achsen die Gebiete teilt können die entstehenden Subgebiete mit Hilfe eines kD-Baumes gespeichert werden. Dabei entspricht der Schnitt durch ein Gebiet gerade einem Knoten im kD-Baum. Abbildung 2.2 zeigt den durch die Beispiel-Bisektion entstandenen kD-Baum. Die Wurzel repräsentiert die erste Teilung durch die rote Linie und die anderen Knoten entsprechend die übrigen Zerlegungen. Links von einem Knoten ist die Koordinate kleiner, rechts größer als der abgebildete Wert. Genauere Informationen zur Bisektion mit kD-Bäumen sind in [BU10] von Buchholz enthalten.

Ein anderes, lokales Verfahren ist die Diffusion. Dabei werden an Grenzen zwischen Gebieten, die unterschiedlichen Prozessoren zugeordnet sind, Zellen ausgetauscht. In [WL93] sind verschiedene Formen der Diffusion vorgestellt, bei denen der Austausch entweder vom Sender oder Empfänger initialisiert wird. Dabei senden Prozessoren mit Unterlast eine Anfrage um Last zu erhalten, überlastete Prozessoren schicken dagegen ihre Last selbstständig weiter. Abbildung 2.3 zeigt schematisch die verschobenen Grenzen. Dabei erhält das rote Gebiet von Schritt 1 nach 2 einige Zellen aus der Grenzregion mit grün und blau.

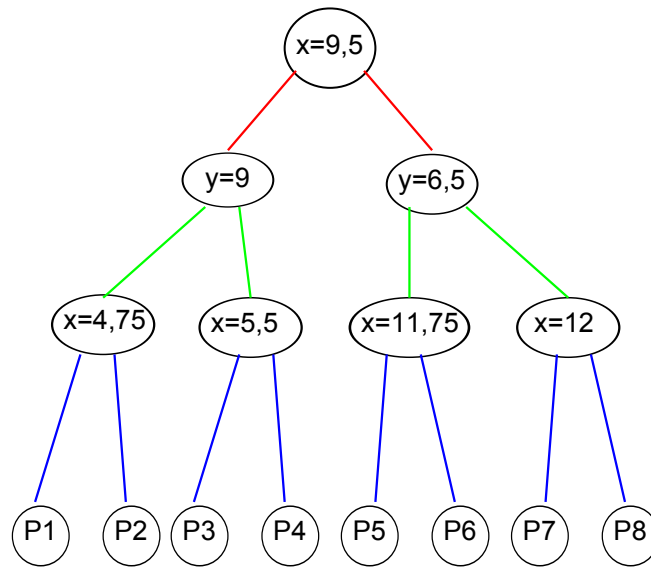


Abbildung 2.2: kD-Baum erzeugt aus einer Bisektion

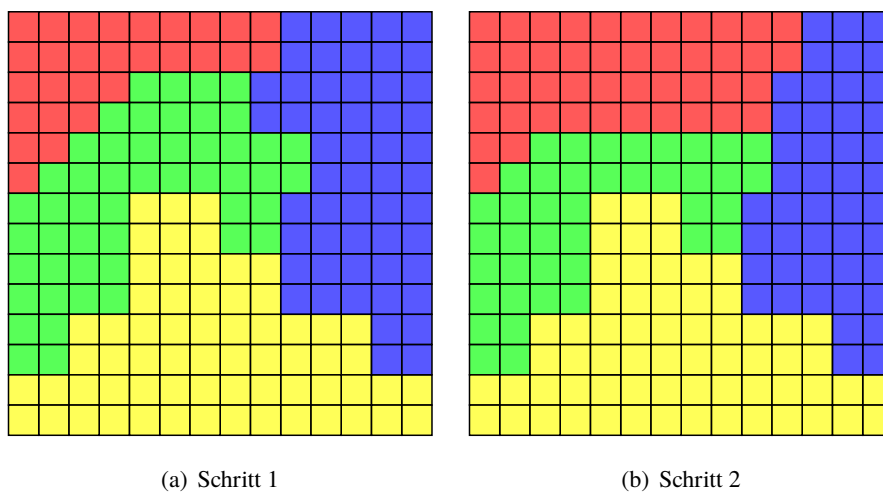


Abbildung 2.3: Beispiel Partitionierung mit Diffusion

Eine weitere Methode ist die Abbildung des Partitionierungsproblems auf ein Graphpartitionierungsproblem. Dabei werden die Gitterzellen jeweils auf einen Knoten des Graphen abgebildet und Nachbarschaftsbeziehungen von Zellen durch Kanten modelliert. Anschließend wird auf den dadurch entstandenen Graphen ein Graphpartitionierer angewandt. Dies ist beispielhaft in Abbildung 2.4 zu sehen. Graphpartitionierer sind Heuristiken, die das NP-harte Graphpartitionierungsproblem lösen. Dabei versucht man bei der Zerlegung eines Graphen in k disjunkte Subgraphen die Anzahl geschnittener Kanten zu minimieren. Um reale Anwendungen besser repräsentieren zu können werden bei der Modellierung oft noch weitere Einschränkungen aufgestellt. In [KL70] wird das

Problem für elektronische Schaltkreise modelliert. Andere Varianten partitionieren Graphen über die Gewichte der Knoten. Eine einfache molekulardynamische Modellierung dazu ist die Zuweisung der Partikelzahl einer Zelle als Gewichte der Knoten.

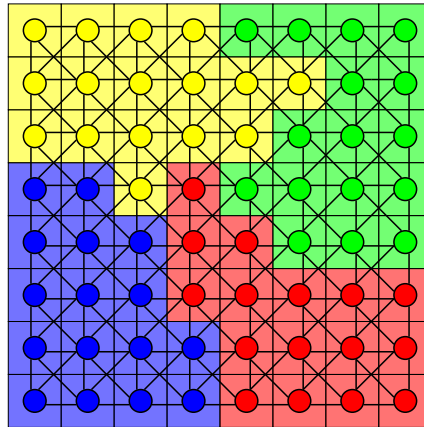


Abbildung 2.4: Beispiel einer Graph Partitionierung

Die Gitter-basierte Partitionierung, beschrieben von Begau et al. in [BS15], setzt zu Beginn eine kartesische Aufteilung des Simulationsgebiets voraus. Dabei entstehen quaderförmige Partitionen für die Prozessoren. Diese Quader werden durch 8 Eckpunkte beschrieben. Die Idee dieses dynamischen Verfahrens ist es, die Eckpunkte zu bewegen um eine Rebalancierung der Last und damit eine neue Partitionierung zu erzeugen. Dazu berechnet man für jede Partition ein Lastzentrum, ähnlich einem Massezentrum, das eine virtuelle Kraft hervorruft, die auf die Eckpunkte wirkt. Diese werden dann so bewegt, dass sich die Last des zugehörigen Gebiets dem Mittelwert annähert. Die Bewegung der Punkte ist bei Verwendung von Linked-Cells auf die Gitterzellen eingeschränkt.

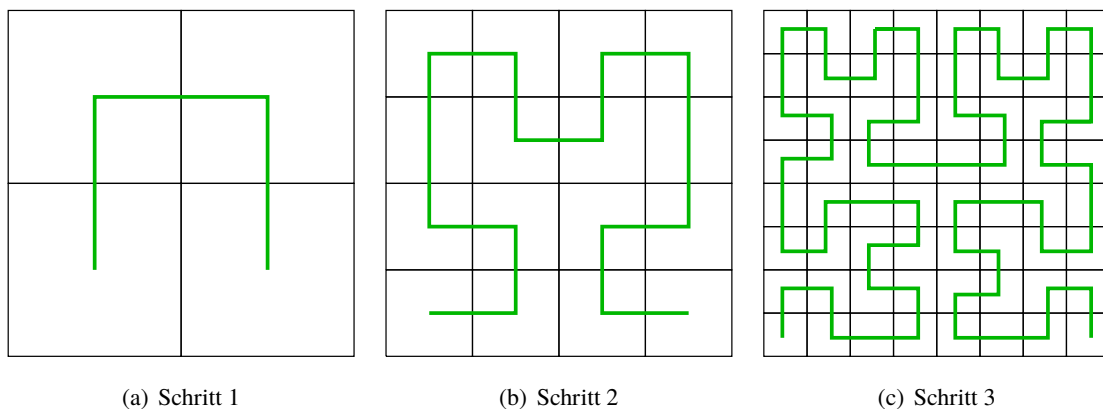


Abbildung 2.5: Beispiel für die Iterationsschritte bei einer Hilbert-Kurve

Die raumfüllenden Kurven (nachzulesen in [BU10]) sind surjektive Abbildung von \mathbb{R} nach \mathbb{R}^n , in unserem Fall $n = 3$. Das heißt jeder Raumpunkt wird mindestens einmal von der Abbildung getroffen. Man konstruiert die Kurven rekursiv und unterteilt dabei die Gebiete in jedem Iterationsschritt erneut. Dies ist für $n = 2$ in Abbildung 2.5 für eine Hilbert-Kurve schrittweise dargestellt. Ein Quadrant wird im zweidimensionalen Fall in vier neue Gebiete unterteilt, bei $n = 3$ wären es 8.

Sobald die Größe der Gebiete, die bei den Iterationsschritten entstanden sind den Zellen entspricht kann durch Anwenden der Abbildung partitioniert werden. Es entsteht eine eindimensionale Kette der nacheinander besuchten Zellen. Dieses eindimensionale Partitionierungsproblem kann zum Beispiel durch Bisektion gelöst werden. Eine beispielhafte Partitionierung durch eine Hilbert-Kurve ist ein Abbildung 2.6 dargestellt.

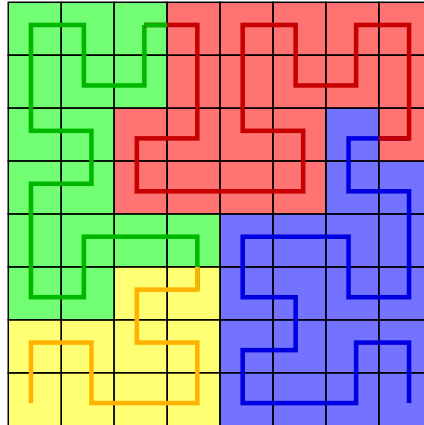


Abbildung 2.6: Beispiel Partitionierung durch raumfüllende Kurven

2.4 Stochastische Verfahren zur Lösung von Optimierungsproblemen

Das erste Verfahren ist Simulated Annealing. Diese Methode optimiert eine Zielfunktion f ähnlich der Suche nach einem Systemzustand mit günstiger Energie in der Physik, welcher durch eine Boltzmann-Verteilung beschrieben wird. Daraus leitet man eine Wahrscheinlichkeit P ab, die man bei der Auswahl neuer Lösungen verwendet. Während des Verfahrens vergleicht man den aktuell betrachtete Zustand mit benachbarten Lösungen. Diese entstehen durch minimale Veränderungen der momentanen Lösung $f(x)$ für eine Zielfunktion f . Ist eine benachbarte Lösung $f(y)$ besser, wird sie übernommen. Ist diese schlechter wird der Wert mit Wahrscheinlichkeit P dennoch behalten. P hängt von $f(x)$, $f(y)$ und T folgendermaßen ab:

$$P(f(x), f(y), T) \propto \exp\left(-\frac{f(y) - f(x)}{T}\right)$$

T ist die Temperatur und eine monoton gegen Null fallende Folge. Diese kann entweder vor Beginn festgelegt werden oder durch den Quotient aus der maximalen Anzahl an Iterationsschritten mit der aktuellen Anzahl bestimmt werden. Mit kleinem T wird auch P kleiner und damit auch die Chance, dass man eine schlechtere benachbarte Lösung wählt. Weitere Informationen zu Simulated Annealing sind zum Beispiel in [KG83] zu finden.

Das zweite Verfahren sind genetische Algorithmen. Diese aus der Biologie abgeleitete Methode orientiert sich an den Prinzipien der natürlichen Selektion um Lösungen für Optimierungsprobleme zu finden. Dabei wird eine Population aus Individuen betrachtet, die jeweils eine Lösung des Problems repräsentieren. Diese Lösungen sind meist binär codiert und werden als Chromosome

bezeichnet. Eine Fitnessfunktion bewertet die Güte der Individuen und damit die der Lösungen. Zusätzlich gibt es drei zentrale Operatoren bei der Lösungsfindung: Selektion, Kreuzung (engl. Crossover) und Mutation. Durch die Selektion werden Individuen mit guter Fitness ausgewählt um an der Kreuzung teilzunehmen, welche wiederum beschreibt wie die Gene von Vater und Mutter zu Kindern kombiniert werden. Wir erzeugen immer zwei Kinder bei einer Kreuzung. Die Mutation kann zum Beispiel in Form von einem oder mehreren Bitflips Vielfalt in die Population bringen und vorzeitigen Konvergenz des Verfahrens verhindern. Es ist auch möglich, dass die Fitness- und Zielfunktion identisch sind. Weitere Ausführung zu genetischen Algorithmen sind zum Beispiel in [GO89] enthalten.

Typische Selektion-Operatoren, die wir im späteren Verlauf ebenfalls verwenden werden sind:

- **Tournament-Selection:** An einem Turnier der Größe k können ebenso viele Individuen teilnehmen. Die beiden Teilnehmer mit der besten Fitness werden zur Kreuzung zugelassen.
- **Roulett-Wheel-Selection:** Die Teilnehmer haben entsprechend ihrer Fitness eine hohe oder niedrige Wahrscheinlichkeit ausgewählt zu werden. Eine gute Fitness entspricht dabei einer hohen Chance.
- **Linear-Ranking-Selection:** Zuerst werden die Individuen nach ihrer Fitness sortiert. Basierend auf den Positionen in dieser Liste und dem Selektionsdruck wird durch einen linearen Term eine Rangliste erstellt. Die Individuen zur Kreuzung werden anschließend anhand der Rangliste ausgewählt. Der Selektionsdruck ist die Wahrscheinlichkeit das Individuum mit der besten Fitness zu wählen verglichen mit der durchschnittlichen Wahrscheinlichkeit.
- **Exponential-Ranking-Selection:** Analoges Vorgehen zum linearen Fall. Allerdings wird die Rangliste durch einen exponentiellen Term erzeugt.

3 Literaturrecherche und Modell

Zu Beginn dieser Arbeit wird eine Literaturrecherche zu Partitionierungsverfahren in der Molekulardynamik durchgeführt, die in den Grundlagen vorgestellt wurden. Die Paper analysieren wir insbesondere auf verwendete Modelle, Maße und Zielfunktionen, die zur Partitionierung genutzt werden. Mit Hilfe dieser Erkenntnisse erzeugen wir später unser Modell zur Optimierung.

Zur Bisektion werden mehrere Paper betrachtet. Poirrier [PO09] beschreibt einen pruning Algorithmus, thematisiert aber kein Modell, Maße oder eine Zielfunktion. Fleissner und Eberhard [FE08] nutzen das Oberflächen-Volumen-Verhältnis zum Minimieren und damit als Maß. Nyland et al. verwenden als Maß die Anzahl der Atome, geben aber ansonsten nur eine textuelle Beschreibung ihrer Bisektionsstrategie. Wolfheimer et al. assoziieren in [WGW06] Gewichte $w_{i,j,k}$ einer Zelle (i, j, k) mit Feldern und Partikelzahl. Bei einer Partitionierung auf P Prozessoren wird das ideale Gewicht einer Partition berechnet als:

$$w_{\text{ideal}} = \frac{1}{P} \sum w_{i,j,k}$$

Die Bisektion wird dann an eine Bedingung geknüpft, die Prozessorzahlen und Gewichte der Subgebiete in Verbindung bringt. Für die Prozessorzahlen P_1, P_2 und Gewichte $w_{\text{sub}_1}, w_{\text{sub}_2}$ der Subgebiete muss gelten:

$$\frac{P_1}{P_2} = \frac{w_{\text{sub}_1}}{w_{\text{sub}_2}}$$

Wobei zusätzlich gilt $P_1 = \lceil \frac{P}{2} \rceil$ und $P_2 = \lfloor \frac{P}{2} \rfloor$.

Cybenko [CG89] und Hu et. al [HB98] betrachten die Diffusion im Kontext der Lastbalancierung. Dabei repräsentiert $w_i^{(t)}$ die „workload“ beziehungsweise Aufgaben, die Prozessor i zu Zeitpunkt t zu erledigen hat. Das Modell hat die Form

$$w_i^{(t+1)} = w_i^{(t)} + \sum_j \alpha_{ij} (w_j^{(t)} - w_i^{(t)}) + \eta_i^{(t+1)} - c \quad (3.1)$$

Dabei beschreibt α_{ij} die Verbindung zwischen Prozessoren i und j . Sollte diese nicht vorliegen ist $\alpha_{ij} = 0$. Die Summanden der mittleren Summe stehen für die zwischen Prozessor i und j transferierte Last. $\eta_i^{(t+1)}$ ist die neu generierte Arbeit während Zeitschritt t . c ist die Arbeit, die von t bis $t + 1$ von einem Prozessor verrichtet werden kann und wird daher abgezogen. In beiden Papern wird daraus dann ein Gleichungssystem erstellt $w^{(t+1)} = M w^{(t)}$. M enthält dabei die teilweise abgewandelten Faktoren α_{ij} . Eine Lösung davon stellt eine valide Partitionierung durch Diffusion dar. Willebeek-Lemair et al. verwenden in [WL93] das gleiche Maß wie Cybenko. Im Gegensatz dazu betrachten Boillat et al. in [BB91] die Diffusion im Kontext der Molekulardynamik. Das verwendete Maß $l(i, t)$ sind die Partikel von Prozessor i zu Zeit t . Es werden einige Gleichungen aufgestellt, die letztlich wie Cybenko und andere zu einem Gleichungssystem führen.

Die Graphpartitionierung wird von Kernighan et al. in [KL70] zunächst nur für zwei Partitionen A, B betrachtet. Dabei ist es das Ziel, für die Kosten $c_{i,j}$ der Kante zwischen Knoten i und j , den Term

$$\sum_{A \times B} c_{ab} ; a \in A; b \in B$$

zu minimieren. Die Heuristik tauscht lokal Knoten zwischen den Partitionen aus, wenn sich daraus bessere Kosten ergeben. Das Maß ist nicht genauer definiert. Catalyurek et al. beschäftigen sich in [CB09] mit der Zerlegung eines Hypergraphen in k Partitionen im Kontext der Lastbalancierung. Hierbei muss gelten $\forall p = 1, \dots, k : W_p \leq W_{avg}(1 + \epsilon)$, wobei

$$W_p = \sum_{v_i \in G_p} w_i$$

die Summe der Gewichte w_i der Knoten v_i des Gebiets G_p ist. Außerdem ist

$$W_{avg} = \frac{\sum_{v_i \in G} w_i}{k}$$

die Summe der Gewichte aller Knoten im gesamten Gebiet G geteilt durch die Prozessorzahl k . ϵ beschreibt hierbei die maximal tolerierbare Imbalance.

Die Gitter-basierte Partitionierung wird von Begau et al. in [BS15] auf ein Lastbalancierungsproblem angewandt. Initial ist das Gebiet kartesisch in quaderförmige Partitionen aufgeteilt, welche durch ihre acht Eckpunkte $\vec{c}_i(p)$ beschrieben werden können. Dabei ist $i \in [0, 7]$ und $p \in [0, P - 1]$ für P Gebiete, die durch P Prozessoren verwaltet werden. Als Maß dient erneut die generische „workload“ λ_p eines Gebiets p . Durch Bewegen der Eckpunkte wird versucht in jedem Gebiet die durchschnittliche „workload“ zu erreichen. Das Modell zur Bewegung der Eckpunkte hat die Form

$$\vec{c}_p^{(n+1)} = \vec{c}_p^{(n)} + \mu \vec{f}_p^{(n)} \quad (3.2)$$

Dabei ist $\vec{c}_p^{(n)}$ die Position der p -ten Ecke zum Zeitpunkt n , μ ein Faktor, der die Wachstumsrate der Gebiete bestimmt und $\vec{f}_p^{(n)}$ ist die virtuelle Kraft, die durch das Lastzentrum hervorgerufen wird. Begau et al. erwähnen allerdings auch, dass die CPU Berechnungszeit für Partikelinteraktionen oder die Partikelzahl als mögliche Maße geeignet sind.

Bei den raumfüllenden Kurven beschäftigen sich Pilkington et al. in [PB96] im Kontext der Lastbalancierung mit generischen Gewichten. In den meisten Fällen wird die Hilbert-Kurve in Kombination mit Bisektion verwendet, so auch von Xiaolin und Zeyao, die in [XZ04] ein molekulardynamisches Szenario betrachten. Dabei approximieren sie die work load in ihrem Modell durch die Partikelzahl in jeder Zelle.

Die Modelle der verschiedenen Verfahren sind natürlich sehr unterschiedlich. Allerdings lassen sich bei der Wahl der Maße durchaus Gemeinsamkeiten feststellen. Nachfolgend sind die Erkenntnisse der Literaturrecherche in Tabelle 3.1 noch einmal zusammengefasst. Darin tritt die Partikelzahl vermehrt als gewähltes oder vorgeschlagenes Maß auf. Dies versuchen wir im nächsten Kapitel mit dem Problem PARTITION zu einem Modell für die Optimierung zu verbinden.

Verfahren	Modell/Zielfunktion	Maß
Bisektion	$w_i \approx w_{\text{average}} \forall \text{Partitionen } i$	Oberflächen-Volumen-Verhältnis, Felder, Partikelzahl
Diffusion	Gleichung (3.1)	„workload“, Last/Partikel von Prozessor i zu Zeit t
Graphpart.	$\min \{ \sum_{A \times B} c_{ab} \}$	Knoten-/Kantengewichte
Gitter-basiert	Gleichung (3.2)	„workload“, Berechnungszeit für Partikelinteraktionen, Partikelzahl
SFC	Hilbert-Kurve + Bisektion	„workload“, Partikelzahl

Tabelle 3.1: Zusammenfassung der Literaturrecherche

3.1 Modellbildung

Um das Partitionierungsproblem der Molekulardynamik als Optimierungsproblem betrachten zu können, muss dafür zunächst ein Modell formuliert werden. Dieses soll dann mit Frameworks zur Lösung von Optimierungsproblemen untersucht werden. Das zuvor erwähnte *k-way number partitioning* ähnelt stark der Verteilung von gewichteter Last auf k Prozessoren. Diese Last kann durch die Anzahl der Partikel, die in der Literatur häufig als Maß verwendet wird, modelliert werden.

Die Idee ist nun die Anzahl der Partikel, die auf die Prozessoren verteilt werden, genauer zu betrachten. Dabei soll der am stärksten ausgelastete Prozessor, also der mit den meisten zugeordneten Partikeln, als Kern der zu minimierenden Zielfunktion dienen. Unter allen vorhanden k Prozessoren wird zunächst jener ermittelt, der die größte Last besitzt. Diese gilt es dann zu minimieren. Für Prozessoren P_1 bis P_k und einer Funktion $f_{\text{Kosten}}(P_j)$, die die Anzahl der Partikel angibt, die Prozessor P_j zugewiesen sind, lässt sich ein Ausdruck für den am stärksten ausgelasteten Prozessor angeben mit:

$$\max_j \{ f_{\text{Kosten}}(P_j) \} \quad (3.3)$$

Um festlegen zu können nach welcher Variablen der Term (3.3) minimiert werden soll definieren wir nun wie eine valide Partitionierung auf k Prozessoren dargestellt wird. Dazu nutzen wir für ein gegebenes zu partitionierendes Gebiet mit n Zellen einen Vektor \vec{x} mit ebenfalls n Einträgen. Jeder Eintrag dieses Vektors repräsentiert eine Zelle im Gebiet. Dabei wird das Gebiet mit einer zuvor festgelegten Traversierungsmethode durchschritten um eine konsistente Nummerierung der Zellen zu erhalten. Für ein dreidimensionales Gebiet bietet es sich zum Beispiel an von vorne unten links nach hinten oben rechts Ebene für Ebene abzuarbeiten. Dies ist schematisch in Abbildung dargestellt.

Die Einträge des Partitionierungsvektors \vec{x} können dann Werte von 1 bis k annehmen um dem Index des Prozessors zu entsprechen dem diese Zelle zugewiesen wurde.

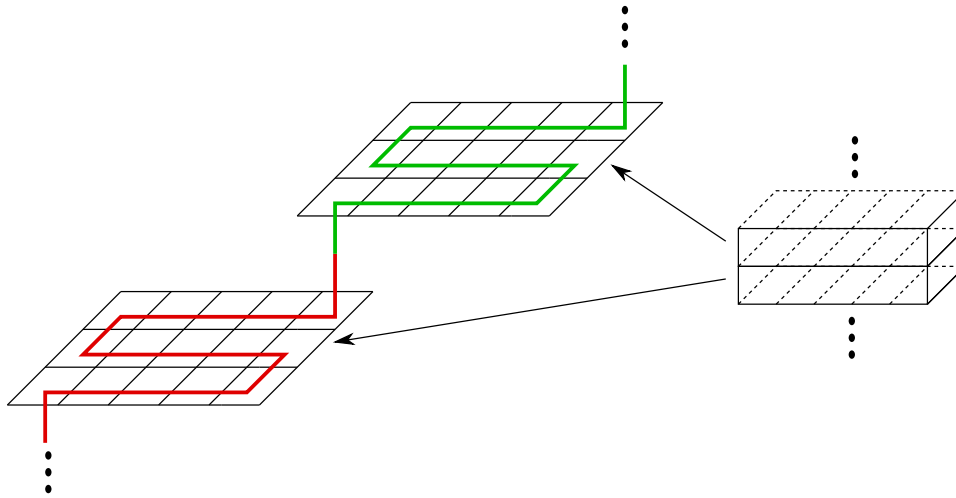


Abbildung 3.1: Beispiel für eine Traversierungsmethode eines dreidimensionalen Simulationsgebiets

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, x_i \in \{1, \dots, k\}, i \in \{1, \dots, n\}$$

Um das vollständige Modell zu formulieren benötigen wir noch die Definition von f_{Kosten} . Da nach \vec{x} minimiert wird, wird diese Funktion erweitert zu $f_{Kosten}(\vec{x}, j)$. Dazu wird noch eine Hilfsfunktion $f_{Partikel}(i)$ benötigt, die für eine Zelle i des Simulationsgebiets die Anzahl der darin enthalten Partikel zurückgibt.

$$f_{Kosten}(\vec{x}, j) = \sum_{i=1}^n \begin{cases} f_{Partikel}(i) & , \text{ wenn } x_i = j \\ 0 & \text{sonst} \end{cases} \quad (3.4)$$

Mit Gleichung (3.3) und der Definition von f_{Kosten} in (3.4) kann das Modell für das Optimierungsproblem vervollständigt werden.

$$\min_{\vec{x}} \left\{ \max_j \{ f_{Kosten}(\vec{x}, j) \} \right\} \quad (3.5)$$

Die Funktion $f_{Partikel}(i)$ erlaubt Anpassungen des Modells. Sollten Kommunikationskosten oder andere Faktoren für ein Anwendungsszenario relevant sein, können diese durch Veränderung dieser Funktion integriert werden. In dieser Arbeit schränken wir uns allerdings auf die Partikelzahl ein und modellieren keine zusätzlichen Kosten. Insbesondere achtet das Modell nicht auf zusammenhängende Gebiete bei der Partitionierung.

3.2 Löser für Optimierungsprobleme

Um Optimierungsprobleme zu lösen gibt es Löser Bibliotheken. Zuerst wird `scipy.optimize.minimize` [SCMI] aus der `scipy` Bibliothek für python genutzt. Diese Methode bietet verschiedene Lösungsverfahren zu Auswahl an. Teilweise erlauben die Verfahren die Definition von Grenzen (engl. bounds) und Bedingungen (engl. constraints). Zur Implementierung des Modells aus Gleichung (3.5) benötigen wir Grenzen, da jeder Eintrag des Partitionierungsvektors \vec{x} nur erlaubte Prozessornummern annehmen kann. Dadurch können Verfahren ohne Grenzen ausgeschlossen werden. Ein Löser, den die Bibliothek als Standardeinstellung festgelegt hat ist L-BFGS-B. Genaueres zu diesen Lösungsverfahren ist in [BL95] und [ZB97] nachzulesen. Erste Testläufe zeigen allerdings, dass eine Formulierung des Modells als Minimum eines Maximum für den Löser nicht geeignet ist. Daher wird der Ausdruck mit dem Maximum ersetzt durch eine Summe über die Abweichung aller Prozessoren vom Mittel. Sei die durchschnittliche Partikelzahl c_{mean} pro Prozessor bei k Prozessoren gegeben durch:

$$c_{mean} = \frac{\sum_{i=1}^n f_{Partikel}(i)}{k} \quad (3.6)$$

Die Gleichung 3.5 ändert sich durch die Summe der Abweichungen zu:

$$\min_{\vec{x}} \left\{ \sum_{j=1}^k |f_{Kosten}(\vec{x}, j) - c_{mean}| \right\} \quad (3.7)$$

In Gleichung (3.7) ist der Betrag dadurch hinzugekommen, dass Prozessoren existieren, die eine Partikelzahl kleiner dem Durchschnitt zugewiesen haben. Deren Abweichung vom Mittel würde als negativer Summand die Summe verfälschen. Diese Veränderung des Modells ermöglicht den erfolgreichen Ablauf der Tests, allerdings zeigen die ersten Ergebnisse, dass der initial eingegebene Partitionierungsvektor nicht verändert wurde. Dies lässt auf einen weiterhin fehlerhaften Durchlauf schließen.

Das betrachtete Optimierungsproblem nutzt für die Prozessornummern ganzzahlige Einträge, die von einem Löser während des Lösungsvorgangs verändert werden. Daher bietet es sich an Bibliotheken zu verwenden, die explizit für Integer (dt. Ganzzahl) Probleme geeignet sind. Ein Framework, das Integer Optimierungsprobleme mit und ohne constraints lösen kann ist die *SCIP (Solving Constraint Integer Programs) Optimization Suite* [SCIP]. Bei der Implementierung von (3.7) für diesen Löser werden die Einträge des Partitionierungsvektors auf Integer-Variablen eingeschränkt. Die Ergebnisse der Testläufe weisen allerdings auf keine durchgeführte Optimierung seitens des Frameworks hin. Daher wurde eine andere Formulierung der Gleichung (3.7) implementiert, die die von der Bibliothek ebenfalls angebotenen Boolean-Variablen nutzt. Der Partitionierungsvektor änderte sich dadurch in eine Partitionierungsmatrix X für k Prozessoren und n Zellen:

$$X = \begin{pmatrix} x_{1,1} & \dots & x_{1,k} \\ & \vdots & \\ x_{n,1} & \dots & x_{n,k} \end{pmatrix} \forall x_{i,j} \in \{\text{True}, \text{False}\}, i \in \{1, \dots, n\}, j \in \{1, \dots, k\}$$

Eine Zeile der Matrix repräsentiert eine Zelle des Simulationsgebiets. Eine Zelle i ist Prozessor j zugewiesen wenn $x_{i,j} = \text{True}$. Zusätzlich werden noch constraints benötigt, da in einer Zeile nur genau ein Eintrag True sein kann. Testläufe mit dieser Formulierung erzeugten eine Fehlermeldung

beim Start. Ausdrücke die Summen von Summen enthalten (beachte f_{Kosten} enthält eine Summe) sind nicht für diesen Optimierer geeignet. Da es keine Möglichkeit gibt das Modell noch weiter zu vereinfachen werden keine weiteren Versuche mit Lösern für Optimierungsprobleme durchgeführt. Stattdessen nutzen wir nun stochastische Verfahren und implementieren diese in Prototypen. Die Formulierung des Problems aus Gleichung 3.7 wird für die im nächsten Kapitel vorgestellten Prototypen beibehalten.

4 Prototypen

Im folgenden werden nun Prototypen beschrieben für die stochastischen Methoden *Simulated Annealing* und *genetische Algorithmen*. Den Prototypen werden Datensätze zweier Rußpartikelsimulation übergeben, die $5 \cdot 10^4$ beziehungsweise $3,2 \cdot 10^6$ Partikel enthalten. Das Ziel ist es dann den Raum, der die Partikel enthält zu partitionieren. Die Prototypen sollen später zur Bewertung der Qualität der Lösungen anderer Partitionierer eingesetzt werden. Der Fokus beider Prototypen ist dabei die Partitionierung der Last, die Laufzeit wurde nicht optimiert.

4.1 Prototyp für Simulated Annealing

In der bereits zuvor verwendeten Bibliothek *scipy* sind auch stochastische Verfahren enthalten, darunter zum Beispiel *scipy.optimize.dual_annealing* [SCDA]. Diese Funktion arbeitet nach dem Prinzip des *generalized simulated annealing* aus [XSG97], welches eine Verallgemeinerung von *classical simulated annealing* und *fast simulated annealing* ist. Die Effizienz dieser Methode ist höher als die der anderen beiden Varianten insbesondere deshalb, da sie sich in Tests mit heterogenen Nickel-Teilchen bei steigender Anzahl an Variablen durch eine stetig wachsende Effizienz ausgezeichnet hat (siehe [XG00]). Das bedeutet, sie ist für diese Problemstellung besonders gut geeignet, da die Anzahl der Zellen und damit der Variablen meist äußerst groß ist (Maximum der Datensätze bei ca. $2 \cdot 10^6$ Variablen).

Wie bereits aus der Wahl der Bibliothek hervorgeht ist dieser Prototyp in *python* implementiert. Dabei besteht die grundlegende Funktion darin eine durch Eingaben definierte Kostenfunktion der *dual_annealing* Methode zu übergeben und deren Ausgaben sinnvoll zu verarbeiten. Die Eingabeparameter bestehen aus:

- Eingabe-Datei: Die Datensätze, die als 3-dimensionales Array die Partikelzahl pro Zelle beschreiben.
- Prozessorzahl: Die gewünschte Anzahl an Prozessoren, die je eine Partition der ermittelten Partitionierung erhalten sollen.
- Ausgabe-Datei: Der Name, der für mehrere Ausgabe Dateien genutzt wird, die unter anderem die Partitionierung enthalten.
- Durchläufe Simulated Annealing: Dies legt die Anzahl der Iterationsschritte beim *Simulated Annealing* fest.
- Imbalance Threshold: Eine Abbruchbedingung, die mit der Imbalance der aktuellen Partitionierung korrespondiert.
- Drei Faktoren: Diese beschreiben eine Zerlegung der Prozessorzahl in drei Faktoren.

- Dateiname um eine Datei zu laden: Bietet die Option bereits durchgeführte Testläufe erneut zu laden um eventuelle Verbesserungen zu erzielen oder abgebrochene Testläufe wieder aufzunehmen.

Zu Beginn werden im Prototyp Eingabeparameter und Dateien geladen. Dann wird die durchschnittliche Partikelzahl *mean* analog zu Gleichung (3.6) berechnet. In Abbildung 4.1 ist eine Implementierung von Gleichung (3.7) in Pseudocode dargestellt.

```

1: function CLEANCOST( $\vec{x}$ ):
2:   TotalCost = 0
3:   CostPerProcessor= [0]·NumberOfProcessors
4:   for ( $x_1, x_2, x_3$ ) in Particles do
5:     CostPerProcessor[x[IndexOf( $x_1, x_2, x_3$ )]-1] += Particles[x1][x2][x3]
6:   end for
7:   for ( $x_1, x_2, x_3$ ) in Particles do
8:     TotalCost += abs(CostPerProcessor[i]-mean)
9:   end for
10:  return TotalCost
11: end function

```

Abbildung 4.1: Pseudocode der zu minimierenden Kostenfunktion

Diese Kostenfunktion erhält den Partitionierungsvektor \vec{x} als Eingabe. Das dreidimensionale Array *Particles* ist der Datensatz der aktuellen Simulation. Dieser enthält die Anzahl der Partikel pro Zelle. Zuerst werden in den Zeilen 2 und 3 die Gesamtkosten *TotalCost* und die Kosten pro Prozessor *CostPerProcessor* jeweils mit 0 initialisiert. In Zeile 4 bis 6 werden in einer for-Schleife über die Partikel im Datensatz die Kosten für die einzelnen Prozessoren bestimmt. Der Ausdruck in Zeile 5 besteht aus:

- *IndexOf(x_1, x_2, x_3)*: Diese Funktion gibt für die Koordinaten (x_1, x_2, x_3) die Nummer dieser Zelle zurück, die zuvor durch eine Traversierungsmethode wie in Abbildung 3.1 bestimmt wurde.
- $x[\text{IndexOf}(x_1, x_2, x_3)]$: Ruft die Zelle des Partitionierungsvektor \vec{x} auf, die zur zuvor ermittelten Nummer korrespondiert.
- -1 : Die Nummerierung der Prozessoren im Vektor läuft von 1 bis zur Anzahl an Prozessoren. Das *CostPerProcessor* Array beginnt mit Index 0.
- *CostPerProcessor*[$x[\text{IndexOf}(x_1, x_2, x_3)]-1$]: Ermöglicht das Ergänzen der Kosten des Prozessors zu dem die Zelle (x_1, x_2, x_3) in \vec{x} zugewiesen wurde.
- $+= \text{Particles}[x_1][x_2][x_3]$: Addiert die Partikelzahl der Zelle (x_1, x_2, x_3) als Kosten hinzu.

In den Zeilen 7 bis 9 werden durch eine zweite for-Schleife über den Datensatz die Abweichungen der zuvor ermittelten Kosten pro Prozessor *CostPerProcessor* zu den Gesamtkosten *TotalCost* addiert. Dabei wird analog zu Gleichung (3.7) die durchschnittliche Partikelzahl pro Prozessor subtrahiert und davon der Betrag gebildet. Schließlich werden die ermittelten Gesamtkosten in dem gewählten Maß Partikelzahl in Zeile 10 zurückgegeben. Die Namensgebung „cleanCost“ stammt

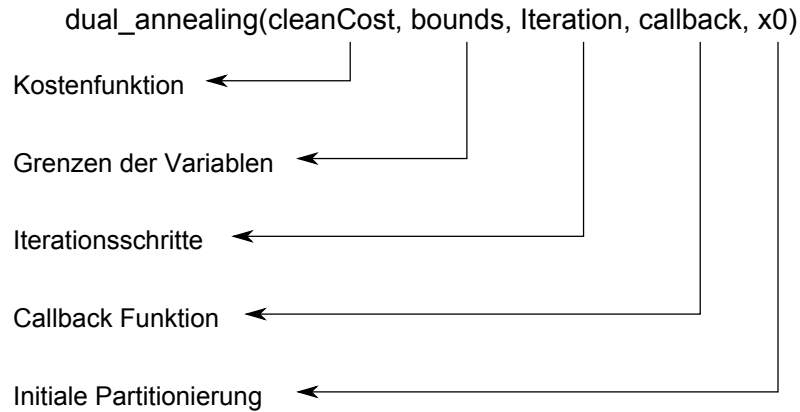


Abbildung 4.2: Methodensignatur *dual_annealing*

daher, dass bereinigte Kosten berechnet werden. Bei ersten Testläufen wurde festgestellt, dass die Datensätze einige leere Zellen enthalten. Das Entfernen dieser Zellen hat die Laufzeit der Partitionierung mit beiden Prototypen auf ein annehmbares Maß reduziert, da in beiden Fällen viele Funktionsevaluationen der Kostenfunktion durchgeführt werden.

Die Abbildung 4.2 zeigt die Methodensignatur von *dual_annealing*. Als erstes wird die zuvor definierte Kostenfunktion `CLEANCOST` übergeben. Für die Grenzen (engl. *bounds*) der Variablen im Partitionierungsvektor wird 1 bis Anzahl Prozessoren angegeben. Durch die Iterationsschritte wird normalerweise die Dauer beim klassischen *simulated annealing* festgelegt. In diesem Prototyp wird eine Abbruchbedingung mittels der Imbalance der Partitionierung formuliert, wodurch es notwendig ist die Anzahl der Iterationsschritte auf einen hohen Wert zu setzen. Für alle untersuchten Datensätze waren 10000 Iterationsschritte vollkommen ausreichend. Die *callback*-Funktion wird von der Methode automatisch aufgerufen, wenn ein neues Minimum gefunden wurde. Der genaue Aufbau dieser Funktion wird im späteren Verlauf dieses Kapitels erläutert werden. Schließlich wird noch eine initiale Partitionierung, als Partitionierungsvektor `x0`, der Funktion übergeben. Dieser wird nicht zufällig gewählt sondern kartesisch bestimmt. Eine kartesische Partitionierung ist leicht zu bestimmen und besitzt zu Beginn bereits eine kleine Imbalance. Wodurch sie gegenüber zufälligen Startwerten vorzuziehen ist. Außerdem erhalten wir vergleichbare Ergebnisse, die sich lediglich durch stochastische Entscheidungen während des Optimierungsprozesses unterscheiden, da alle Durchläufe eines Szenarios mit den gleichen Startvoraussetzungen beginnen. Ein Szenario wird durch die Wahl von Datensatz und Prozessorzahl bestimmt.

Die in den Eingabeparametern erwähnten drei Faktoren werden, wie in Abbildung 4.3 zu sehen, genutzt um eine kartesische Aufteilung des betrachteten Gebiets zu erzielen. Dabei wird je eine Achse in gerade so viele Abschnitte aufgeteilt wie der Betrag des Faktors. Dies erzeugt eine Partitionierung mit quaderförmigen Segmenten. Die Realisierung in der Implementierung ist nachvollziehbar durch den Pseudocode in Abbildung 4.4.

Zunächst werden in Zeile 2 *xWidth*, *yWidth* und *zWidth* als Quotient der Gittergröße und eines Faktors bestimmt. Diese *Width*-Faktoren repräsentieren die Breite der Teilgebiete entlang der jeweiligen Achsen. Anschließend werden daraus in den Zeilen 4 bis 12 die Grenzen des Gebiets eines Prozessors

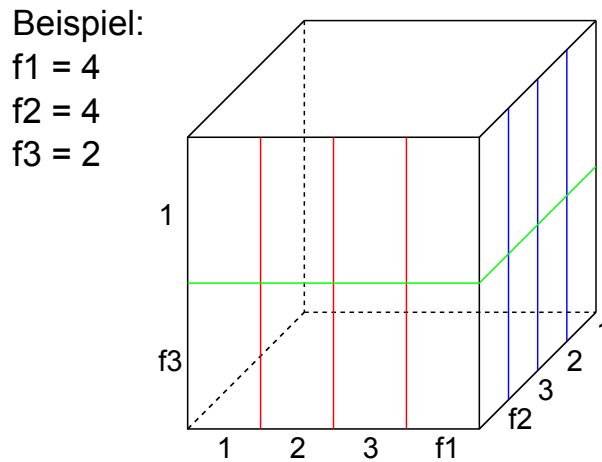


Abbildung 4.3: Beispielhafte initiale kartesische Partitionierung

in allen Dimensionen festgelegt. Damit wird anschließend in den Zeilen 13 bis 20 geprüft ob eine Zelle (x_1, x_2, x_3) im Gebiet eines Prozessors liegt (Zeile 15). Wenn das der Fall ist wird dieser Zelle die Nummer dieses Prozessors zugewiesen (Zeile 16). Sobald alle Zellen bearbeitet wurden ist eine kartesische Partitionierung des Gebiets entstanden, die in x_0 zurückgegeben wird. Die Funktion *IndexOf* ist analog zum Pseudocode in Abbildung 4.1 definiert.

Jetzt muss nur noch die Funktionalität der callback Funktion geklärt werden um einen vollständigen Aufruf von *dual_annealing* durchführen zu können. Grundsätzlich wird callback immer nach dem Fund eines neuen Minimums automatisch vom Framework aufgerufen. Dabei werden insbesondere die aktuelle Partitionierung und der momentane Funktionswert der Kostenfunktion zurückgegeben. Daher nutzen wir callback vor allem zur Verarbeitung der mit jedem neuen Fund generierten Daten und zum potentiellen Abbruch des Durchlaufs. Die Imbalance aus Gleichung (2.2) dient diesem Prototyp in leicht veränderter Form als Abbruchbedingung.

$$\text{imbalance}(\vec{x}) = \frac{\max_j \{f_{\text{Kosten}}(\vec{x}, j)\}}{\text{mean}} \quad (4.1)$$

In den Eingabeparametern des Prototypen kann eine Grenze (imbalance threshold) festgelegt werden. Bei jedem Aufruf der callback-Funktion wird geprüft ob die Grenze erreicht ist und der Durchlauf beendet werden muss. In callback werden Funktionswerte der Kostenfunktion, Imbalance Werte, Zeitdauer und Anzahl an Funktionsevaluationen bei jedem Aufruf gespeichert. Dies ist insbesondere dann hilfreich wenn Durchläufe manuell abgebrochen werden müssen, da dann trotzdem verwertbare Ergebnisse vorhanden sind.

Schließlich gibt es noch die Option alte Durchläufe durch Laden fortzuführen. Dies findet dann Anwendung, wenn Testläufe vorzeitig abgebrochen wurden und dient damit als Sicherheit, damit keine Ergebnisse verloren gehen beziehungsweise unbrauchbar werden.

```

1: function INITCARTGUESS(x1, x2, x3):
2:   xWidth =  $\frac{\text{Gittergröße}}{x1}$ ; yWidth =  $\frac{\text{Gittergröße}}{x2}$ ; zWidth =  $\frac{\text{Gittergröße}}{x3}$ 
3:   processorCount = 1
4:   for z in range(x3) do
5:     for y in range(x2) do
6:       for x in range(x1) do
7:         split[processorCount] = [[x*xWidth,(x+1)*xWidth],
8:           [y*yWidth,(y+1)*yWidth],[z*zWidth,(z+1)*zWidth]]
9:         processorCount += 1
10:      end for
11:    end for
12:  end for
13:  for (x1, x2, x3) in Particles do
14:    for i in split do
15:      if (x,y,z) in split[i] then
16:        x0[IndexOf(x,y,z)] = i
17:        break
18:      end if
19:    end for
20:  end for
21:  return x0
22: end function

```

Abbildung 4.4: Pseudocode zu initialer kartesischer Partitionierung

In den frühen Phasen der Entwicklung dieses Prototyps wurde parallel noch eine Version ohne Bereinigung erstellt. Dabei werden Nullzellen nicht mehr gefiltert sondern auf 1 Partikel gesetzt. Alle anderen Zellen werden mit einer Konstanten multipliziert beziehungsweise es wird eine Konstante addiert. Dies gewährleistet, dass der Einfluss der ehemaligen Nullzellen nicht zu groß ist. Es hat sich allerdings in den ersten Testläufen bereits abgezeichnet, dass die Laufzeit dieses Prototyps um einiges schlechter ist, als die bereinigte Variante. Außerdem können Datensätze, die keine Nullzellen enthalten ebenso vom bereinigten Prototypen bearbeitet werden.

4.2 Prototyp für genetische Algorithmen

Um einen Prototyp mit genetischen Algorithmen zu realisieren wurde das Framework GAFT („A Genetic Algorithm Framwork in pyThon“, siehe [GAFT]) genutzt. Dieses bietet sowohl bereits vorhandene Operatoren zu Selektion, Crossover und Mutation, als auch die Möglichkeit eigene zu kreieren. Das ist im Kontext der Molekulardynamik wichtig, da zusätzlich zu den Operatoren, die auf den codierten Chromosomen arbeiten, eigene erstellt werden können, die direkt die Partitionierung beeinflussen. Dies erlaubt zum Beispiel die Implementierung von Operatoren, die die Prozessorzuweisung verändern.

Den Kern eines genetischen Algorithmus, der mit diesem Framework erstellt wurde, bildet eine „Engine“. Dieser werden alle notwendigen Parameter wie Population, Selektion-, Crossover- und Mutation-Operator übergeben. Anschließend wird die gewünschte Fitnessfunktion bei der Engine registriert. Beim Start wird dann nur noch die Anzahl an Generationen festgelegt und damit nach wie vielen Schritten der Algorithmus beendet wird. In unserem Fall wird die Fitnessfunktion zur Bewertung und als Zielfunktion der Minimierung genutzt.

Ein Individuum repräsentiert in diesem Prototyp eine mögliche Partitionierung. Bei der Initialisierung eines Individuums wird festgelegt wie eine Lösung aussieht. Dazu wird die Anzahl der Zellen als Größe der Lösung definiert. Die Einträge werden durch die gleichen Grenzen wie beim Prototyp zu Simulated Annealing beschränkt. Die Fitnessfunktion wird ganz analog zur Implementierung der Kostenfunktion in Abbildung 4.1 eingefügt. Der Spitzenreiter einer Generation ist dann das Individuum mit der besten Fitness beziehungsweise den niedrigsten Kosten seiner Partitionierung. Dieser wird von der „Engine“ als das aktuell beste Ergebnis geführt.

Im Prototyp sind ebenfalls Möglichkeiten zur Analyse und Speicherung von Ergebnissen implementiert. Beide Funktionalitäten können neu definiert und bei der „Engine“ registriert werden. Erstere Funktion wird nach jedem Iterationsschritt, also jeder Generation, ausgeführt und die andere wird am Ende nach Ablauf aller Schritte zur Speicherung verwendet. Dabei werden wieder Kostenfunktion, Imbalance, Zeit und Anzahl an Funktionsevaluationen gespeichert.

Die Eingabeparameter dieses Prototyps bestehen zum Teil aus Parametern des Simulated-Annealing-Prototyps. Diese werden in der folgenden Auflistung lediglich benannt.

- Eingabe-Datei
- Prozessorzahl
- Ausgabe-Datei
- Pfad Ausgabe-Datei
- Generationen: Die Anzahl an Generationen, die zeitgleich auch der Anzahl an Iterationsschritten des genetischen Algorithmus entspricht.
- Größe Population: Die feste Anzahl an Individuen, die der Algorithmus in einer Generation betrachten soll.
- Selektion Switch: Auswahl, welche Selektion benutzt werden soll.
- Crossover Switch: Auswahl, welches Crossover benutzt werden soll.
- Mutation Switch: Auswahl, welche Mutation benutzt werden soll.

Das Erhöhen der Populationsgröße bietet zwar mehr Optionen an Partitionierungsvektoren und eventuell bessere Fitness in der aktuellen Generation, wirkt sich allerdings stark negativ auf die Laufzeit des Algorithmus aus. Zielführender ist es, eine kleine bis moderat große Population zu wählen und diese mit einem guten Crossover-Operator zu besseren Fitness-Ergebnissen zu führen. Bei den Selektion-Operatoren besteht ausschließlich die Wahl zwischen bereits im Framework vorhandenen Varianten. Dazu gehören Tournament-, Roulette-Wheel-, Linear-Ranking- und Exponential-Ranking-Selection, die in Abschnitt 2.4 der Grundlagen kurz erwähnt werden. Die möglichen Crossover-Operatoren so wie Mutation-Operatoren sind im nächsten Abschnitt genauer erläutert.

4.2.1 Operatoren

Die bereits im Framework vorhandenen Operatoren arbeiten ausschließlich auf dem codierten Partitionierungsvektor, dem Chromosom. Da wir auf die Problemstellung der Molekulardynamik angepasste Operatoren verwenden wollen, werden diese in verschiedenen Varianten neu definiert. Wir implementieren drei neue Crossover- und ein neuer Mutation-Operator.

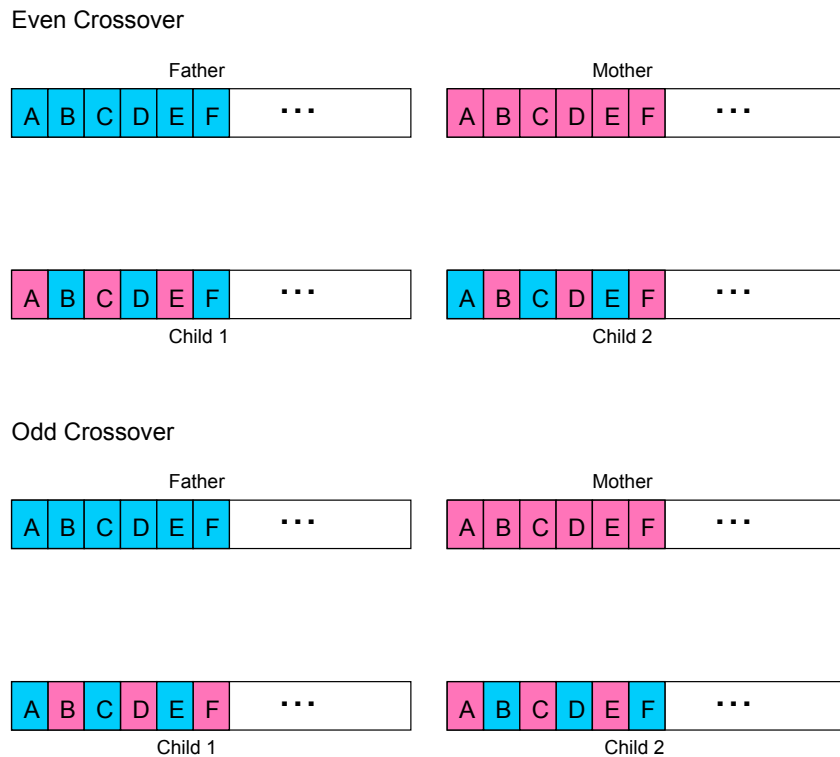


Abbildung 4.5: Even Odd Crossover-Operator

Aus dem Crossover von je einem Vater und Mutter Individuum entstehen zwei Kinder 1 und 2. Beim Even Odd Crossover-Operator beginnt in der Variante Even Kind 1 beziehungsweise 2 zunächst als eine Kopie von Vater respektive Mutter. Danach werden alle Einträge mit geradem Index ersetzt durch die des anderen Elternteils. Bei der Odd Variante werden die Einträge mit ungeraden Indices ersetzt. Dies führt in der Gesamtheit zu den gleichen Kombinationen allerdings sind die Rollen von Kind 1 und 2 vertauscht. In Abbildung 4.5 ist der Even Odd Crossover-Operator schematisch zu sehen. Über den Crossover Switch kann zwischen den Varianten Odd und Even gewählt werden.

Abbildung 4.6 zeigt ein Beispiel für den Alternating Crossover-Operator. Dieser erzeugt in Anzahl Prozessor vielen Schritten zwei Kinder durch abwechselndes Füllen des Partitionierungsvektors mit Einträgen von Vater oder Mutter. Jeweils in Schritt i werden Zuweisungen zu Prozessor i betrachtet. Zunächst erhält Kind 1 alle Einträge des Vaters, die eine 1 enthalten. Analog für Kind 2 und seiner Mutter. In Schritt 2 erhält Kind 1 die Einträge der Mutter, die 2 enthalten. Dabei können eventuell 1 Einträge überschrieben werden. Umgekehrt erhält Kind 2 nun seine Informationen vom Vater. Diese

Alternating Crossover Beispiel

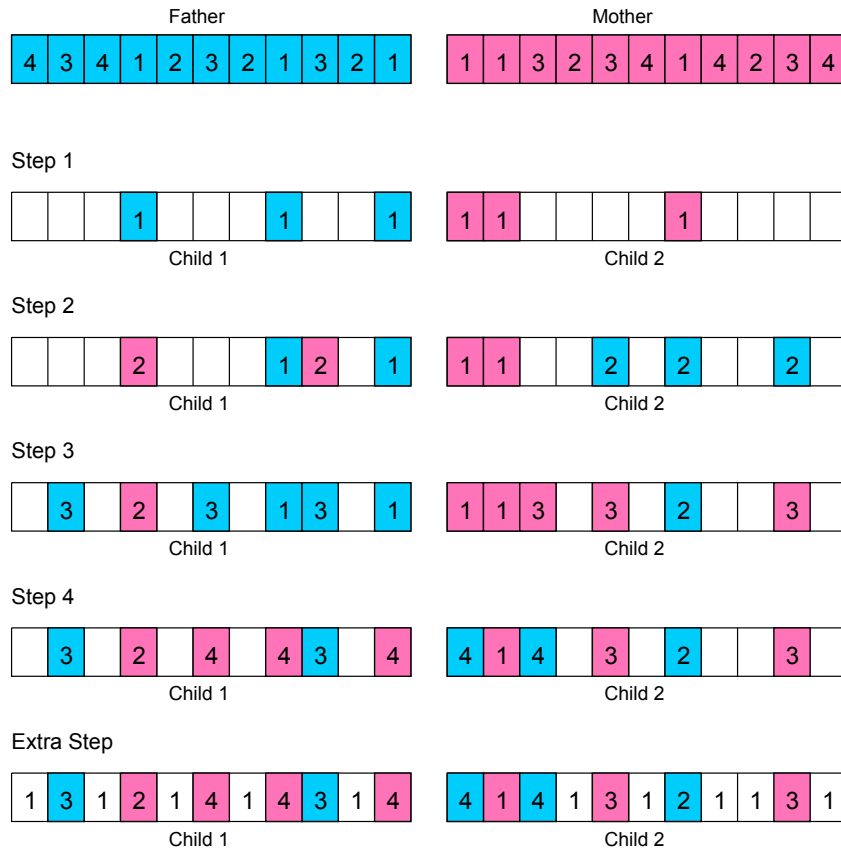


Abbildung 4.6: Alternating Crossover-Operator

abwechselnde Zuweisung wird bis zur im aktuellen Szenario maximalen Prozessorzahl durchgeführt. Etwaige noch nicht belegte Einträge werden anschließend auf den Wert 1 gesetzt. Am Ende bleiben keine leeren Einträge zurück.

Ein Beispiel für den letzten implementierten Crossover-Operator ist in Abbildung 4.7 zu sehen. Der Modulo Crossover-Operator berechnet jeden Eintrag aus Einträgen von Vater und Mutter. Sei dazu k die Anzahl der Prozessoren, $x_{\text{Kind } j, i}$ der i -te Eintrag des Partitionierungsvektors von Kind j , $x_{\text{Vater}, i}$ der i -te Eintrag des Vaters und $x_{\text{Mutter}, i}$ der i -te Eintrag der Mutter. Dann gilt für die Zuweisung:

$$x_{\text{Kind } 1, i} = (x_{\text{Vater}, i} + x_{\text{Mutter}, i}) \% k \quad \forall i \in \{1, \dots, n\}$$

$$x_{\text{Kind } 2, i} = k - x_{\text{Kind } 1, i} \quad \forall i \in \{1, \dots, n\}$$

Es wird also für den Eintrag an der Stelle i bei Kind 1, die jeweiligen Einträge von Vater und Mutter addiert und modulo der Anzahl an Prozessoren gerechnet. Für Kind 2 wird das Ergebnis von Kind 1 von der Prozessorzahl subtrahiert. Nach diesen Berechnungen werden alle Einträge bei Kind 1, die Null ergaben auf die maximal Prozessornummer gesetzt, da kein Prozessor 0 vorhanden ist.

Modulo Crossover Beispiel

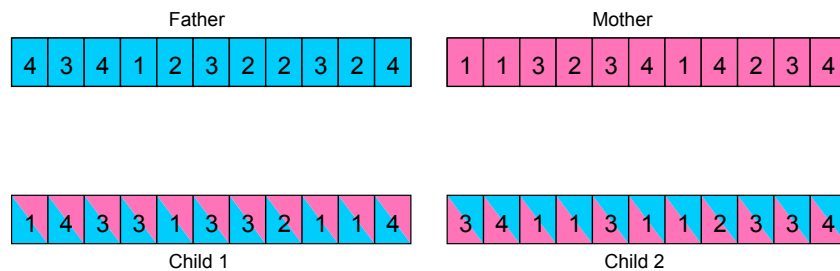


Abbildung 4.7: Modulo Crossover-Operator

Außerdem wird im Prototyp ein Mutation-Operator definiert, der wie in Abbildung 4.8 zu sehen die Einträge eines Partitionierungsvektors um 5 permutiert. Der Standard Mutation-Operator führt dagegen auf den binär codierten Chromosomen lediglich einen Bitflip durch.

Vor Mutation

1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

Nach Mutation

11	10	9	8	7	1	2	3	4	5	6
----	----	---	---	---	---	---	---	---	---	---

Abbildung 4.8: Permutation Mutation-Operator

4.2.2 Startpopulation

Der Prototyp mit genetischen Algorithmen benötigt eine Startpopulation. Diese besteht aus initialen, kartesischen Partitionierungsvektoren. Da die Population mehr als ein Individuum umfasst benötigen wir verschiedene Partitionierungen.

Der Algorithmus in Abbildung 4.9 zeigt das Vorgehen bei der Erzeugung der Startpopulation. In Zeile 2 werden ein Array initialisiert in dem letztlich die verschiedenen Kombinationen der Zerlegungsfaktoren enthalten sind. Der implementierte Algorithmus beschränkt sich auf Zweierpotenzen als Prozessorzahlen, da die Zerlegung beliebiger Werte deutlich komplexer zu implementieren ist. Zunächst werden in den Zeilen 3 bis 8 einfache Kombination aus Zweierpotenzen erzeugt. Diese werden anschließend in den Zeilen 9 bis 17 an eine abgewandelte Form des Algorithmus in Abbildung 4.4 übergeben. Bei Angabe von *ShiftX1*, *ShiftX2* und *ShiftX3* werden die Grenzen der Gebiete in den Zeilen 7 und 8 von Abbildung 4.4 verändert. Entsprechend dem Wert der Shift-Variablen werden die Grenzen für einen Prozessor mit dieser Nummer um 1 verschoben. Die Koordinate ergibt sich dabei aus der Shift-Variablen. Zum Beispiel bedeutet $\text{ShiftX1} = 9$, dass für Prozessor

```
1: function CARTESIANPOPULATION(NumberOfProcessors,PopulationSize):
2:   Factors={ (NumberOfProcessors,1,1)}
3:   for i in range(log2(NumberOfProcessors)) do
4:     for (x,y,z) in Factors do
5:       Factors.add(x/2,y*2,z)
6:       Factors.add(x/2,y,z*2)
7:     end for
8:   end for
9:   for (x,y,z) in Factors do
10:    for ShiftX1 in range(NumberOfProcessors) do
11:      for ShiftX2 in range(NumberOfProcessors) do
12:        for ShiftX3 in range(NumberOfProcessors) do
13:          Population.append(INITCARTGUESS(x,y,z,ShiftX1,ShiftX2,ShiftX3))
14:        end for
15:      end for
16:    end for
17:   end for
18:   return Population
19: end function
```

Abbildung 4.9: Pseudocode für Startpopulation mit kartesischen Partitionierungen

9 die Grenze in x_1 Richtung verschoben wird. Eine Shift-Operation ist noch einmal graphisch in Abbildung 4.10 dargestellt. Das durch `INITCARTGUESS` erzeugte Individuum wird in Zeile 13 der Population hinzugefügt.

Ein paar Sonderfälle müssen allerdings beachtet werden. Am Rand kann man keine Shift-Operation durchführen. Außerdem wenn $xWidth$, $yWidth$ oder $zWidth$ 1 sind, kann in diese Richtung kein Shift ausgeführt werden, da sonst Gebiete verschwinden würden.

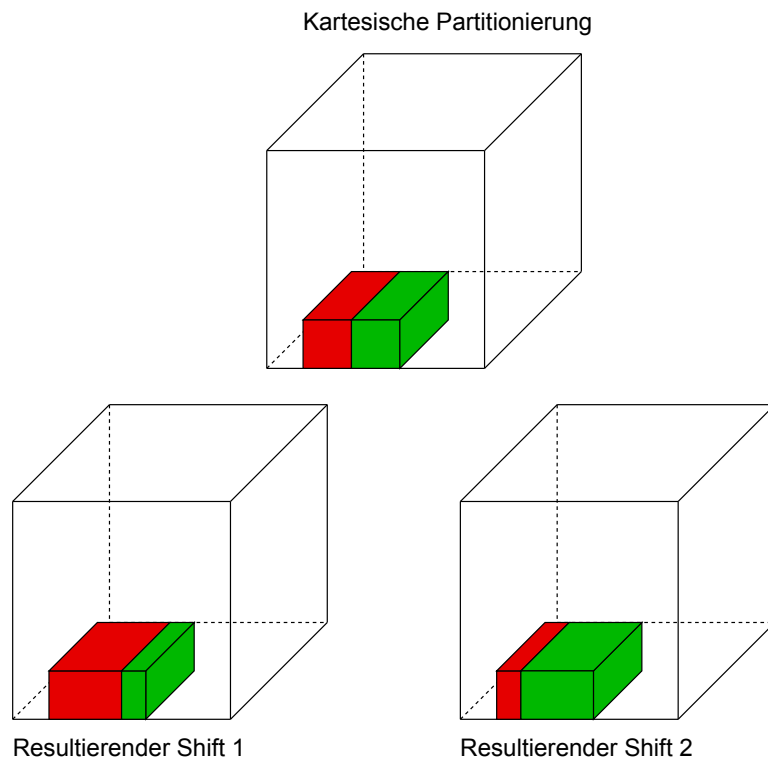


Abbildung 4.10: Shift-Operation bei kartesischer Partitionierung

5 Ergebnisse

Die beiden im letzten Kapitel vorgestellten Prototypen werden genutzt um stochastisch Lösungen für das in Gleichung (3.7) dargestellte Optimierungsproblem zu finden. Dazu werden Datensätze zweier Rußpartikelsimulation genutzt mit 50000 respektive 3,2 Millionen Partikeln. Daraus werden mittels eines Skripts die Eingabedaten für beide Prototypen generiert. Dabei handelt es sich um ein dreidimensionales Array, das nach Definition der zu betrachtenden Box und dem anzulegenden Gitter, die Anzahl der Partikel pro Zelle enthält. Für den 50k-Datensatz werden hauptsächlich Gitter der Größe $32 \times 32 \times 32$ und $64 \times 64 \times 64$ und eine Box der Größe $200 \times 200 \times 200$ verwendet. Für den 3,2-Mio.-Datensatz wird ein Gitter der Größe $128 \times 128 \times 128$ und eine Box der Größe $800 \times 800 \times 800$ benutzt. Mit den beiden Prototypen wird das Partitionierungsproblem für die Datensätze mit verschiedenen Einstellungen gelöst. Variiert werden die Prozessorzahl, die Partikelverteilung und für genetische Algorithmen die Crossover-Operatoren. Während des gesamten Lösungsverfahrens zeichnen wir vorläufige und finale Ergebnisse zur Partitionierung, der Kostenfunktion `CLEANCOST` und der Imbalance auf. Die Zeit wird dabei jeweils vom Start des Lösungsverfahrens bis zu dessen automatischem oder manuellen Abbruch gemessen. Die daraus resultierenden Ergebnisplots werden in den nachfolgenden Abschnitten verglichen und besprochen.

5.1 Ergebnisse mit Simulated Annealing

In Abbildung 5.1 ist die durchschnittliche Abweichung pro Prozessor gegen die Gesamtzeit aufgetragen. Dabei handelt es sich um die in Gleichung (3.7) dargestellte Abweichung geteilt durch die Anzahl an Prozessoren. Deshalb geben diese Kurven einen Eindruck um wie viele Partikel jeder Prozessor vom Mittelwert abweicht. Jeder Punkt auf den Kurven ist dabei ein Aufruf der callback-Funktion, beschreibt also den Fund eines neuen Minimums. Bei Erreichen der Abbruchbedingung, hier durch das Ende der Kurve markiert, ist die Abweichung nahezu Null. Die korrespondierende absolute Abweichung der Kurven liegt zwischen 11 bis 13 Partikeln über alle Prozessoren. Eine weitere Beobachtung ist, dass in den letzten 30-40 Sekunden äußerst viele neue Minima gefunden werden.

Die zu diesen Durchläufen gehörende Imbalance ist in Abbildung 5.2 zu sehen. Die Abbruchbedingung für jede dieser Kurven betrug 1.001 beziehungsweise 0,1%. Diese wird von den Kurven mit Werten von 0,07-0,08% unterschritten. In der Abbildung ist zur Verdeutlichung 1 von der Imbalance subtrahiert. Die Verläufe der Kurven verhalten sich ähnlich zu den Plots der Abweichung pro Prozessor. Allerdings schwanken die Kurven während der ersten gefundenen Minima. Für einen festen Funktionswert zum Zeitpunkt t sind verschiedene Imbalance Werte möglich. Die Gleichung (4.1) zur Imbalance hängt nur vom am stärksten ausgelasteten Prozessor ab. Die maximal mögliche Imbalance wird erreicht wenn alle Partikel einem Prozessor zugeordnet werden. Die beste Imbalance

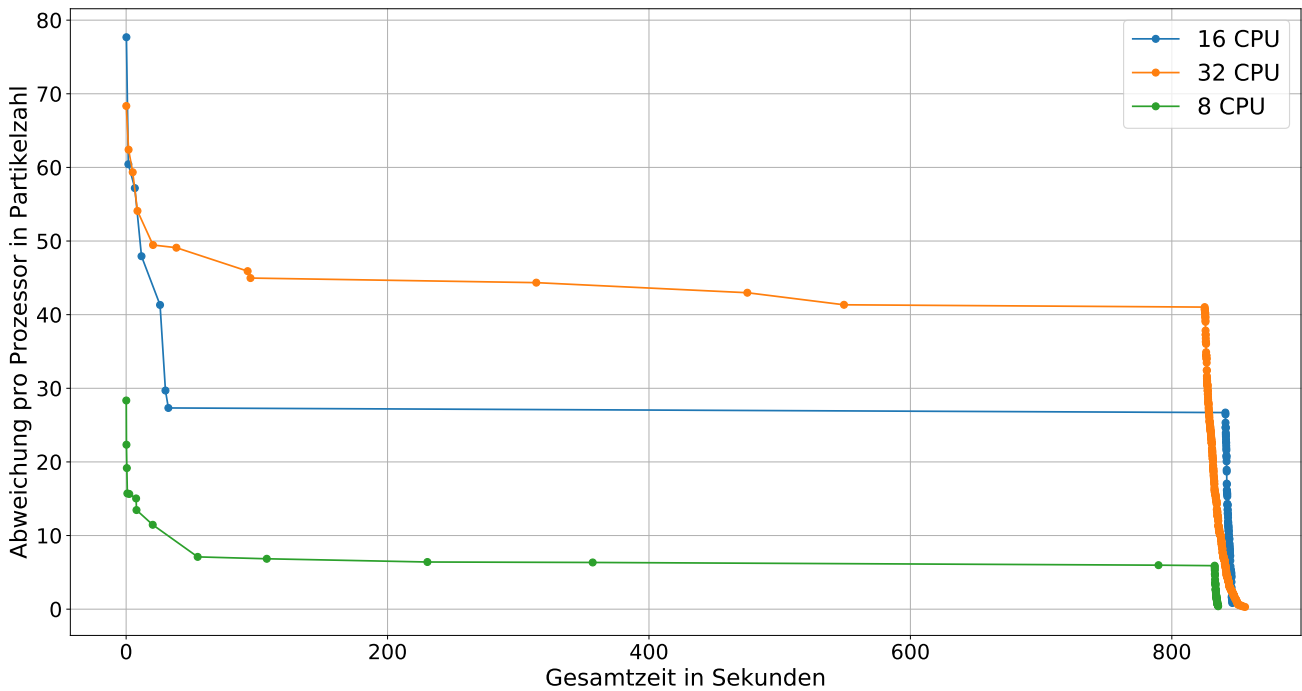


Abbildung 5.1: Durchschnittliche Abweichung von der durchschnittlichen Partikelzahl pro Prozessor für ein Gitter der Größe $32 \times 32 \times 32$, bei dem 50k-Datensatz und bei unterschiedlicher Anzahl an Prozessoren

ergibt sich wenn alle Prozessoren die gleiche Anzahl an Partikeln haben. Daher kann es vor allem am Anfang einer Kurve zu Schwankungen kommen, da die Funktionswerte dies aufgrund ihrer Größe hier noch erlauben.

Zur Verdeutlichung der unterschiedlichen Zeitdauer wird je eine Kurve zur Abweichung und Imbalance bei einem Gitter der Größe $32 \times 32 \times 32$ verglichen mit dem Gegenstück bei Gittergröße $64 \times 64 \times 64$. Dabei ist der 50k-Datensatz und 32 Prozessoren verwendet worden.

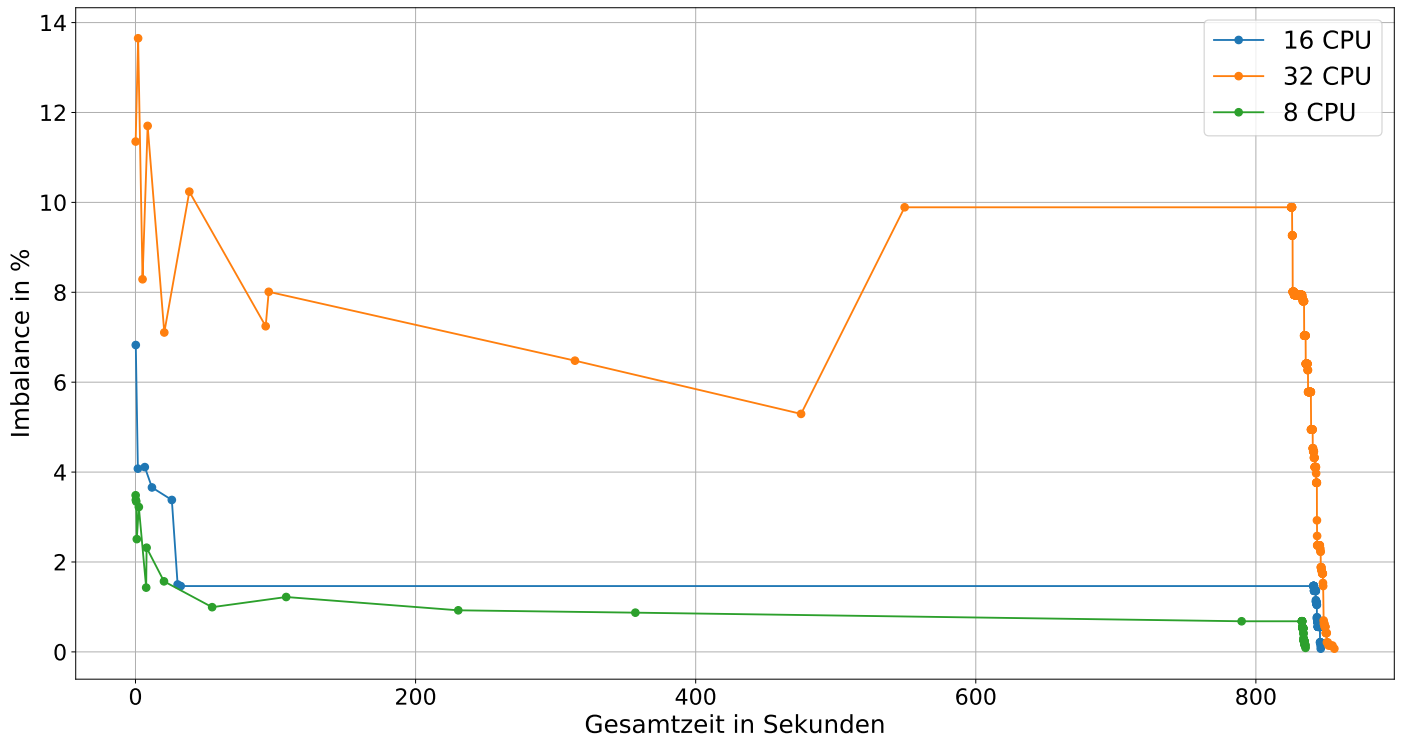


Abbildung 5.2: Imbalance für ein Gitter der Größe $32 \times 32 \times 32$ und den 50k-Datensatz bei unterschiedlicher Anzahl an Prozessoren

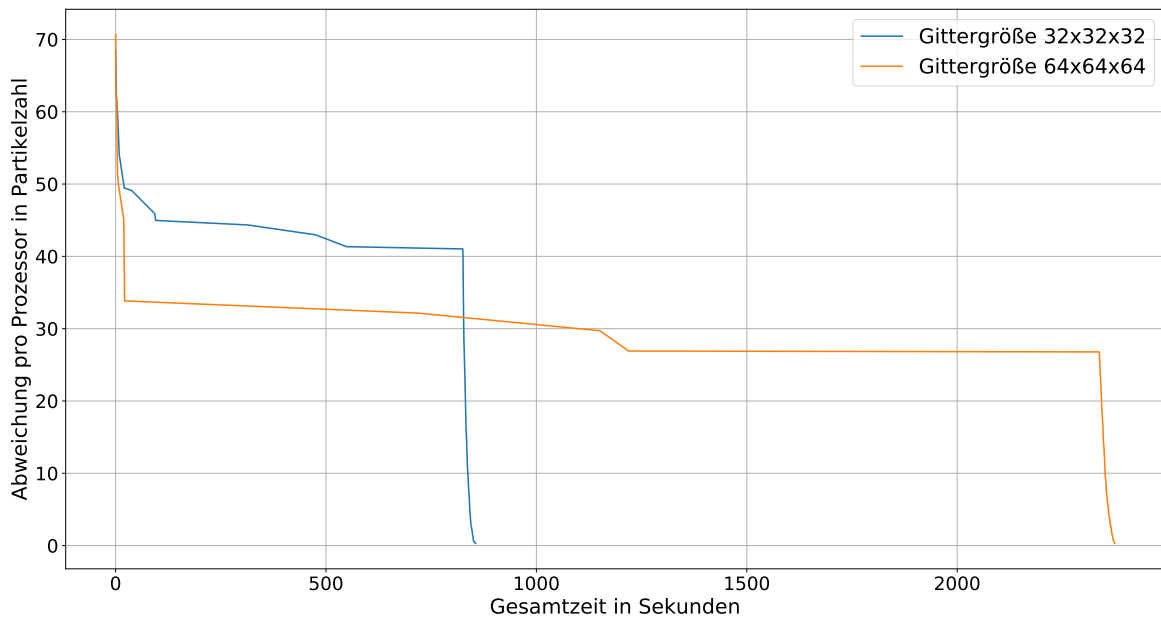


Abbildung 5.3: Vergleich der durchschnittlichen Abweichung pro Prozessor für Gittergrößen $32 \times 32 \times 32$ und $64 \times 64 \times 64$ beim 50k-Datensatz und 32 Prozessoren

Die in Abbildungen 5.3 und 5.4 dargestellten Kurven benötigen bei einem Gitter der Größe $64 \times 64 \times 64$ und einem imbalance threshold von 0,1% ungefähr 39 Minuten zur Fertigstellung. Um Vergleiche mit diesem Szenario ziehen zu können wurde eine Imbalance von 0,8% für raumfüllende Kurven (SFC) als Richtwert gemessen. Diesen Wert kann der Simulated-Annealing-Prototyp mit Imbalance Ergebnissen von 0,07% um eine Größenordnung unterbieten. Die Laufzeit mit raumfüllenden Kurven ist um einiges besser, allerdings lag in dieser Arbeit der Fokus auf der Partitionierung und Lastreduktion und nicht der Laufzeit.

Ebenfalls ein interessantes Szenario mit dem Gitter der Größe $64 \times 64 \times 64$ ist der Vergleich des heterogenen Rußpartikel Datensatzes mit einer homogenen Partikelverteilung. Abbildung 5.5 zeigt, dass eine homogene Verteilung nur ein Viertel der Zeit einer heterogenen benötigt und die Hälfte der Zeit als das Szenario bei einem Gitter der Größe $32 \times 32 \times 32$.

Schließlich bleibt noch der 3,2-Mio.-Datensatz zu untersuchen. Dabei ist in Abbildung 5.6 der vordere Abschnitt der Kostenfunktion zu sehen. In den nachfolgenden 90 Stunden dieses Durchlaufs sind keine großen Sprünge mehr vorhanden. Die Prozessorzahlen 320 und 3200 wurden gewählt um eine ähnlich mittlere Partikelzahl pro Prozessor zu erreichen wie ein vergleichbares Szenario mit dem 50k-Datensatz. Wie bereits hier und auch in Abbildung 5.7 zu sehen ist, ist der Prototyp für diesen Datensatz mit den entsprechenden Einstellung äußerst überfordert. Imbalance Werte unter 15% für 320 Prozessoren sind in annehmbarer Zeit nicht zu erreichen. Eher wird die maximale Anzahl an Iterationsschritten des Simulated Annealing erreicht.

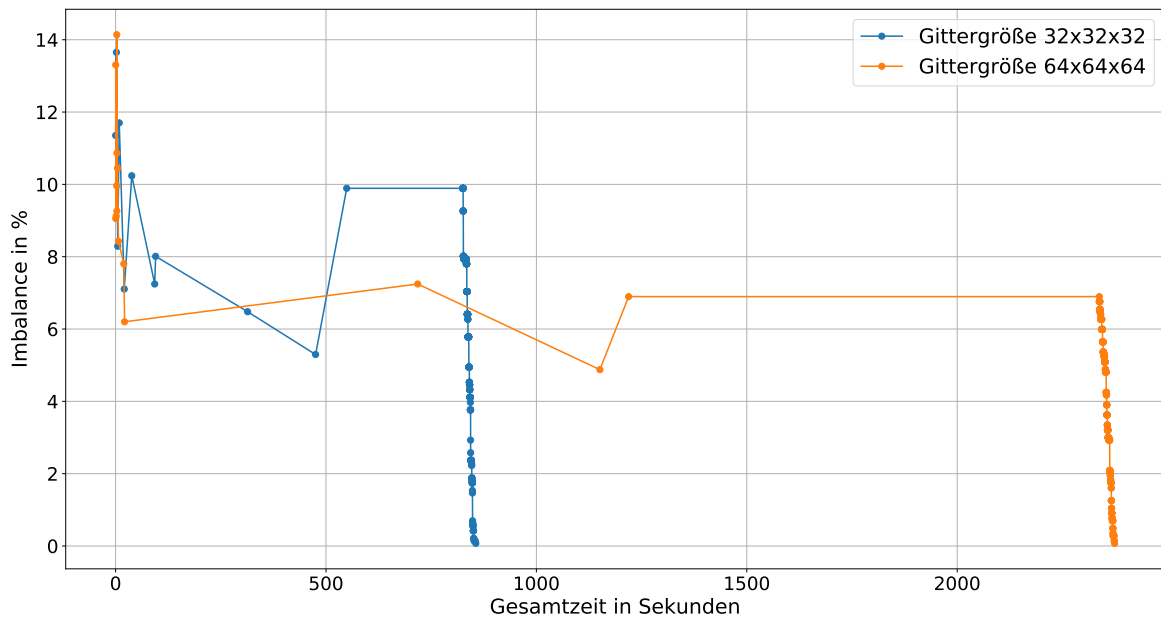


Abbildung 5.4: Vergleich der Imbalance für die Gittergrößen $32 \times 32 \times 32$ und $64 \times 64 \times 64$ beim 50k-Datensatz und 32 Prozessoren

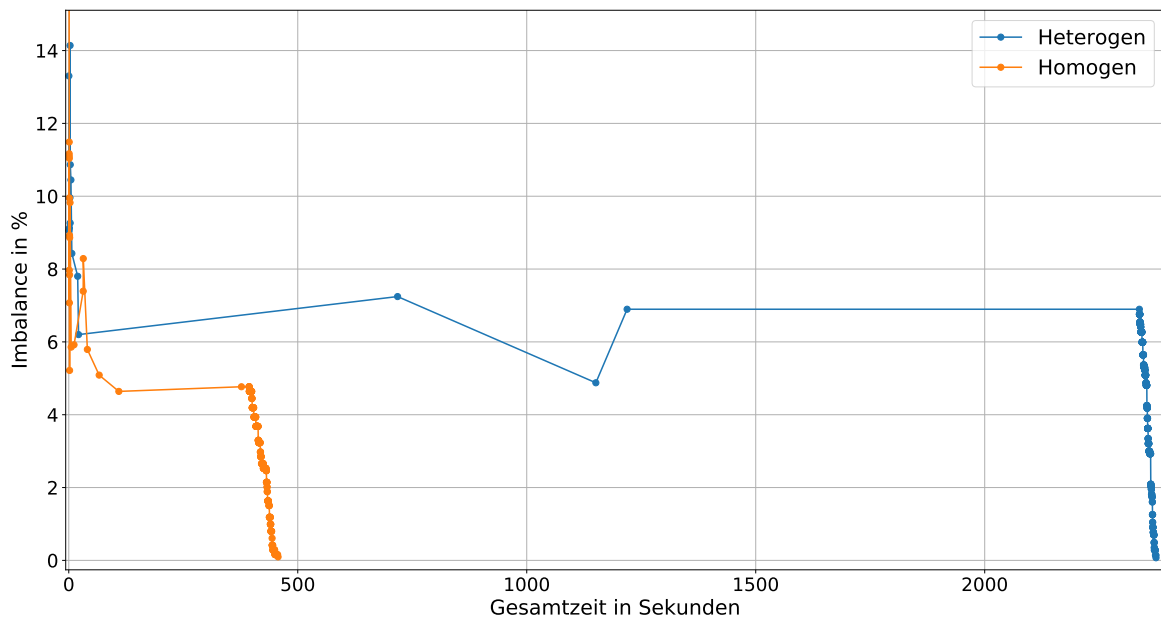


Abbildung 5.5: Vergleich der Imbalance von homogenen und heterogenen Partikelverteilungen bei Gittergröße $64 \times 64 \times 64$ und 50k-Datensatz

5 Ergebnisse

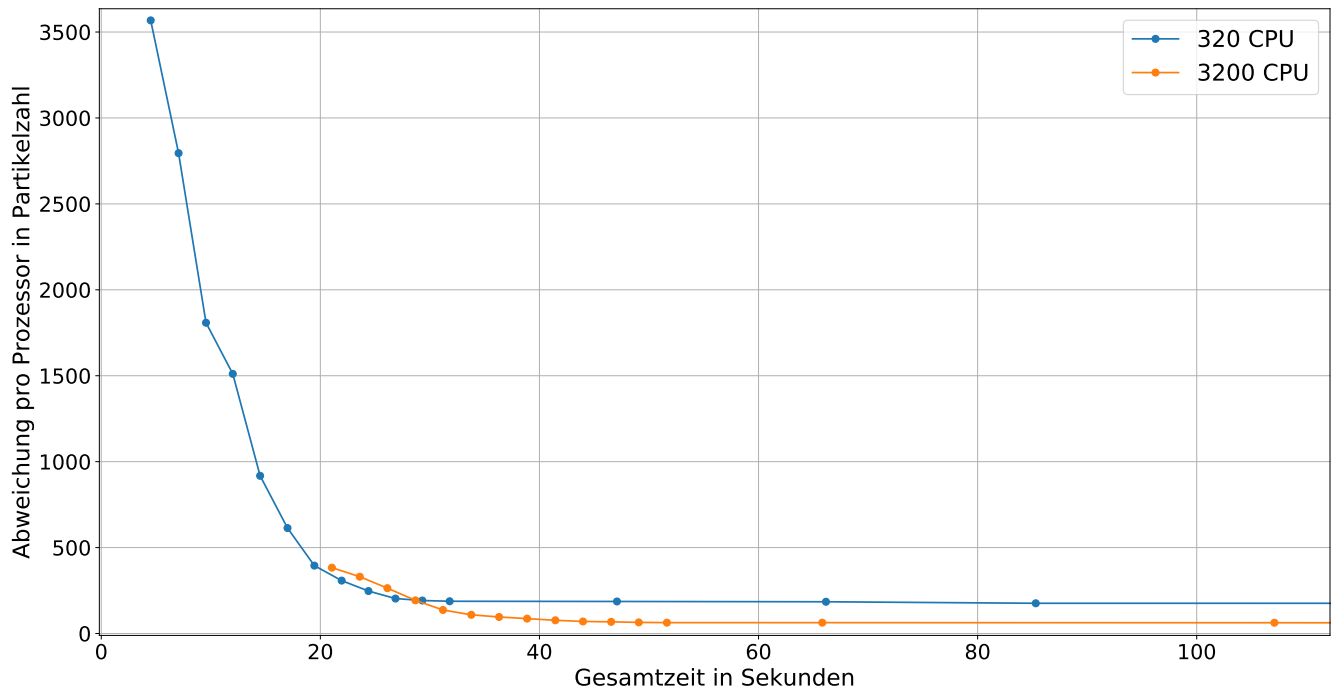


Abbildung 5.6: Vergleich der Abweichung pro Prozessor für ein Gitter der Größe $128 \times 128 \times 128$ und dem 3,2-Mio.-Datensatz bei verschiedenen Prozessorzahlen

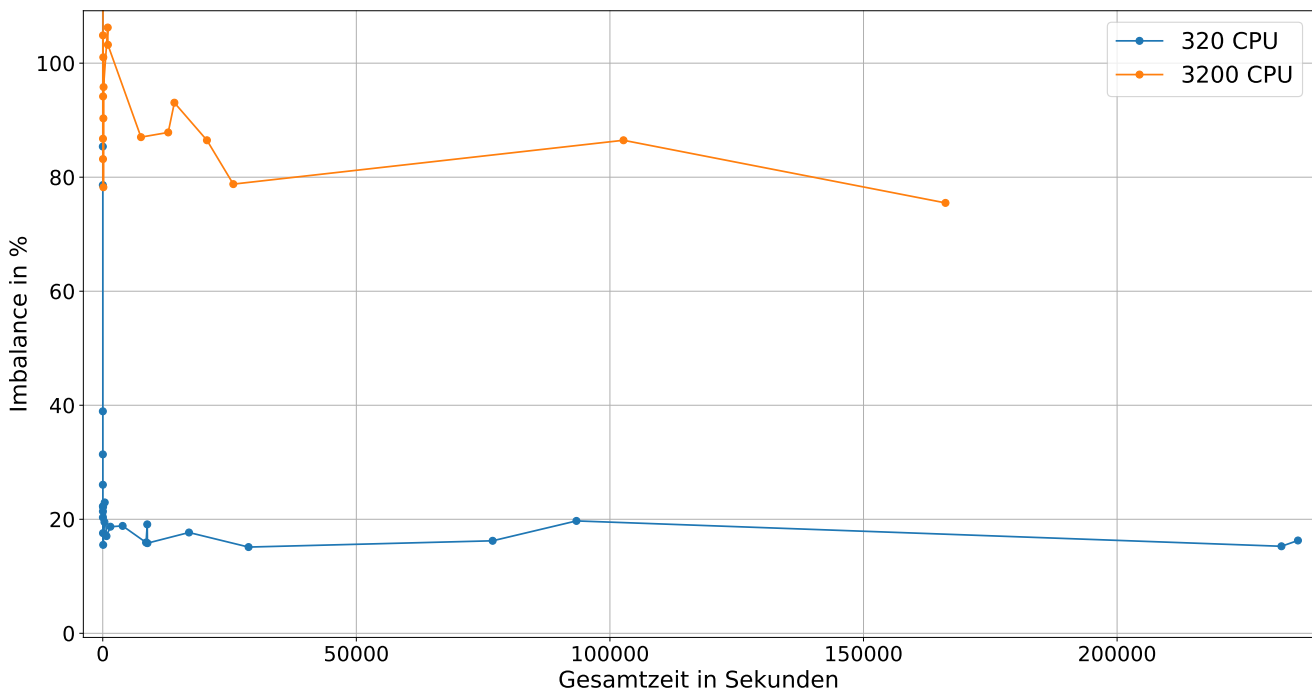


Abbildung 5.7: Vergleich Imbalance für ein Gitter der Größe $128 \times 128 \times 128$ und dem 3,2-Mio.-Datensatz bei verschiedenen Prozessorzahlen

5.2 Ergebnisse mit genetischen Algorithmen

Der Prototyp mit genetischen Algorithmen arbeitet pro Zeitschritt langsamer als jener mit Simulated Annealing. Das ist nicht weiter verwunderlich, da in jeder Iteration eine ganze Population, also mehrere Partitionierungsvektoren betrachtet und mit diesen gerechnet werden muss. Abbildung 5.8 zeigt, dass die Kurve deutlich weniger glatt ist als noch beim simulated annealing und dennoch ähnlich lange braucht. Das heißt es werden weniger Iterationsschritte in der gleichen Zeit geschafft. Dies ist auch gut in Abbildung 5.9 zu erkennen. Im Vergleich zu Simulated Annealing und dem Richtwert von SFC 0,8% schneiden genetische Algorithmen in diesem Szenario schlechter ab.

5 Ergebnisse

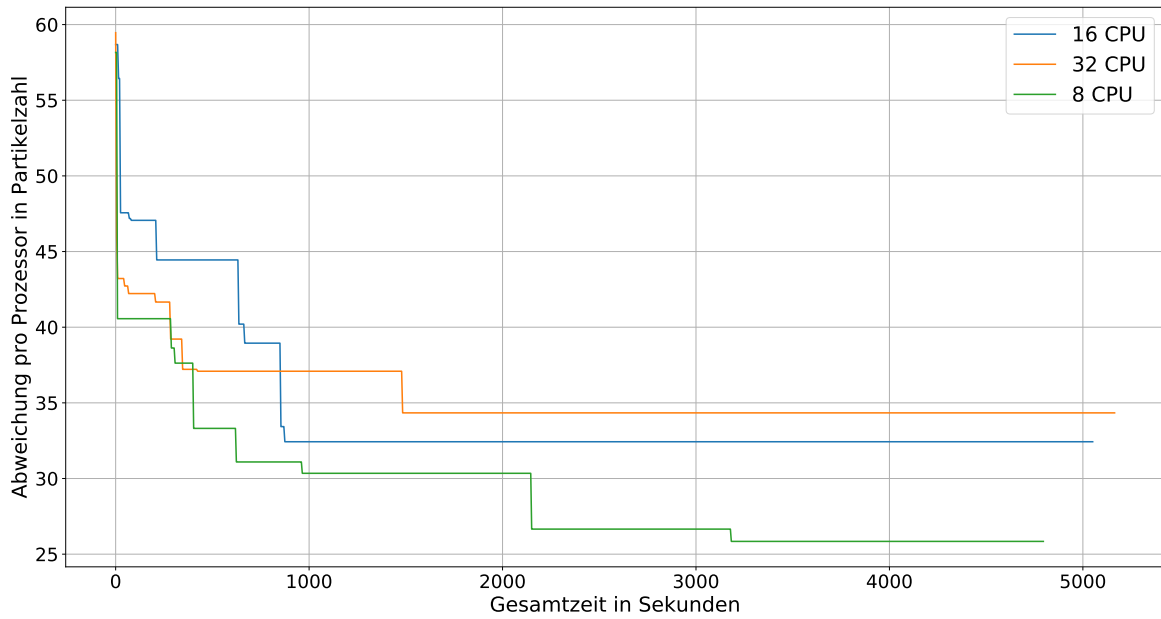


Abbildung 5.8: Vergleich der Abweichung pro Prozessor für den 50k-Datensatz und einem Gitter der Größe $64 \times 64 \times 64$ bei verschiedenen Prozessorzahlen

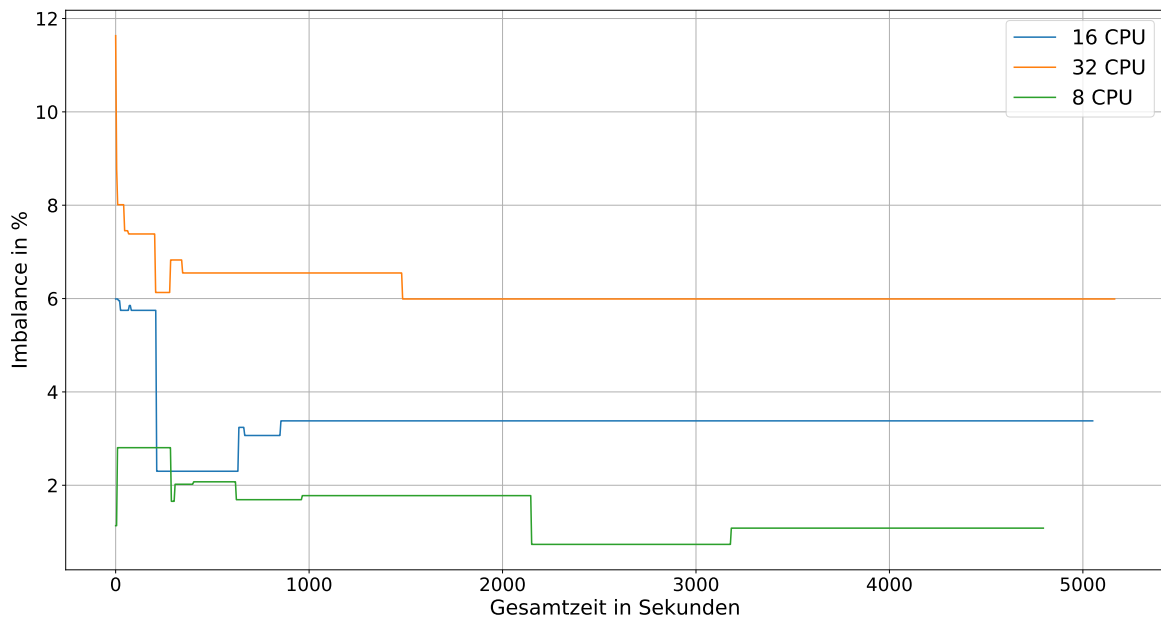


Abbildung 5.9: Vergleich der Imbalance für den 50k-Datensatz und einem Gitter der Größe $64 \times 64 \times 64$ bei verschiedenen Prozessorzahlen

Nachdem einige Crossover-Operatoren im Prototyp definiert wurden ist deren Vergleich interessant zu sehen. Für das Szenario mit einem Gitter der Größe $32 \times 32 \times 32$ und dem 50k Datensatz mit 16 Prozessoren wurde ein Vergleich der Crossover-Operatoren in den Abbildungen 5.10 und 5.11 erstellt. In der ersten Abbildung ist wieder die Abweichung vom Mittel pro Prozessor aufgetragen bei der sich die Operatoren nicht großartig unterscheiden. Bei der Imbalance in der zweiten Abbildung ist zwar eine fallende Tendenz zu erkennen allerdings schwanken die Werte noch sehr stark. Da genetische Algorithmen ein stochastische Verfahren sind müssten viele Testläufe desselben Szenarios gemacht werden um einen klaren Gewinner bestimmen zu können. Wenn man die Performance über alle durchgeführten Testläufe verschiedener Szenarien in Betracht zieht ist der Modulo Crossover-Operator zu bevorzugen.

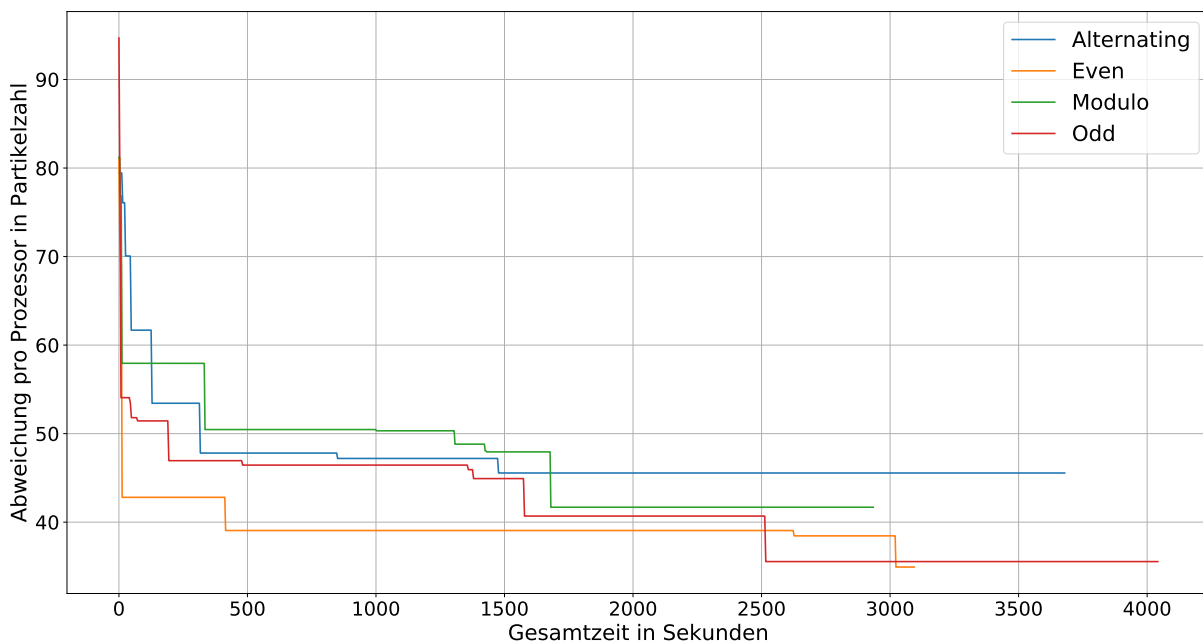


Abbildung 5.10: Vergleich der Abweichung pro Prozessor für verschiedene Crossover-Operatoren bei einer Gittergröße $32 \times 32 \times 32$, dem 50k-Datensatz und 16 Prozessoren

Eine mögliche Einstellung, die bisher noch nicht betrachtet wurde ist die Größe der Box. Diese kann bei der Konvertierung der Datensätze in die Eingabedaten verändert werden. Das hat zur Folge, dass sich um das gesamte Gebiet eine Schicht aus Zellen legt, die fast alle keine Partikel enthalten. Es ist zu erwarten, dass davon die Imbalance erhöht wird, was in Abbildung 5.12 auch zu sehen ist. Dabei ist die Größe $250 \times 250 \times 250$ neu hinzugekommen und die Größe $200 \times 200 \times 200$ wurde bisher verwendet. Obwohl für diesen Vergleich nicht so viele Datenpunkte vorliegen ist dennoch die erhöhte Imbalance bei vergrößerter Box zu erkennen. Die Größe der Gitter unterscheiden sich ebenfalls bei verschiedenen Boxgrößen um die gleiche Zellgröße beizubehalten und werden daher nicht im Plot erwähnt.

5 Ergebnisse

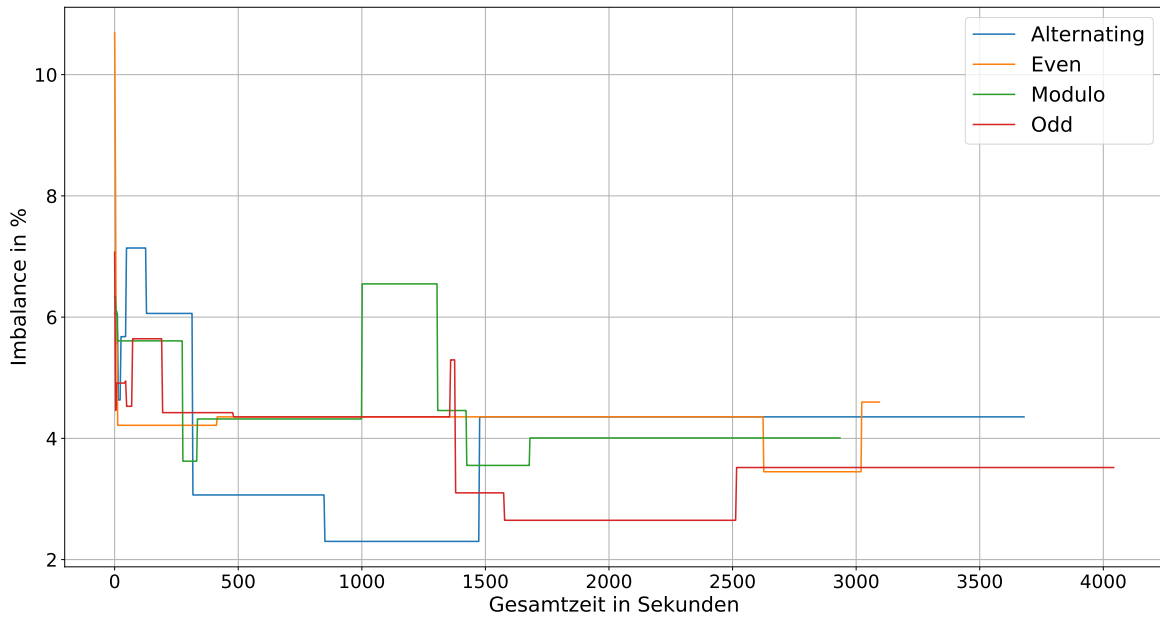


Abbildung 5.11: Vergleich der Imbalance für verschiedene Crossover-Operatoren bei einer Gittergröße $32 \times 32 \times 32$, dem 50k-Datensatz und 16 Prozessoren

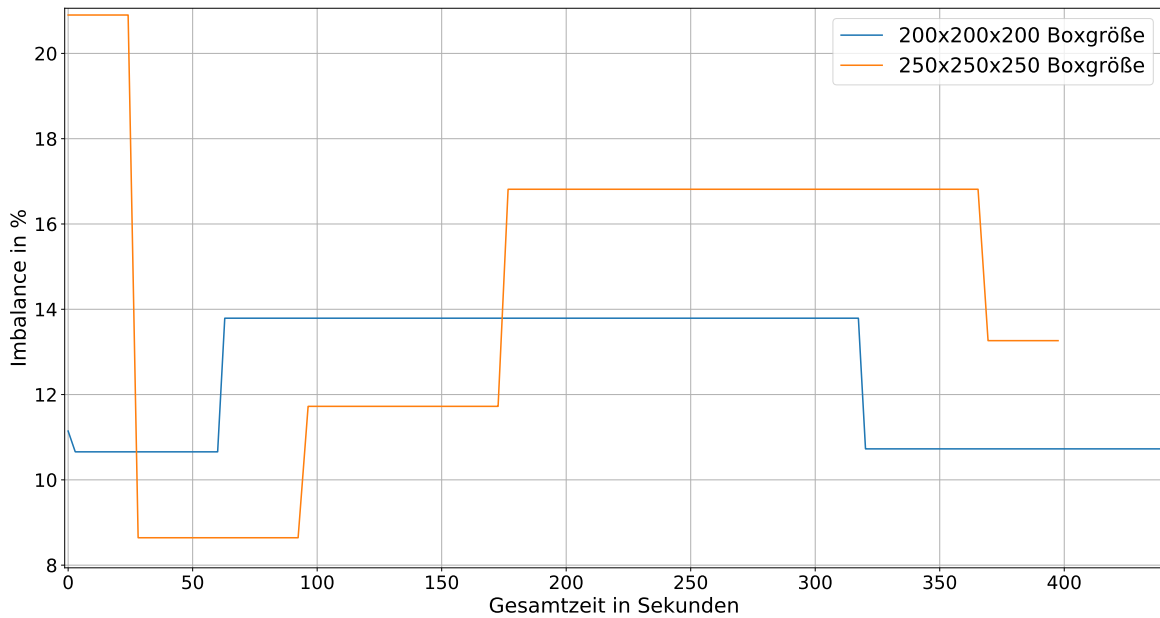


Abbildung 5.12: Vergleich der Imbalance für verschiedene Boxgrößen beim 50k-Datensatz und 32 Prozessoren

Beim letzte Plot zu genetischen Algorithmen handelt es sich um ein Szenario mit dem 3,2-Mio.-Datensatz, zu sehen in Abbildung 5.13. Darin wird eine Imbalance ähnlich der von Simulated Annealing erreicht in einem Bruchteil der benötigten Zeit. Es scheint wengleich der Prototyp mit genetischen Algorithmen bei kleineren Szenarien nicht so gut wie Simulated Annealing abschneidet, erzielt dieser beim großen Datensatz durchaus vergleichbare Ergebnisse. Vermutlich ist gerade das was diesen Prototypen in kleineren Szenarien ausbremst, seine Population mit vielen Partitionierungsvektoren, bei größeren Datenmengen hilfreicher, da schneller Ergebnisse erreicht werden.

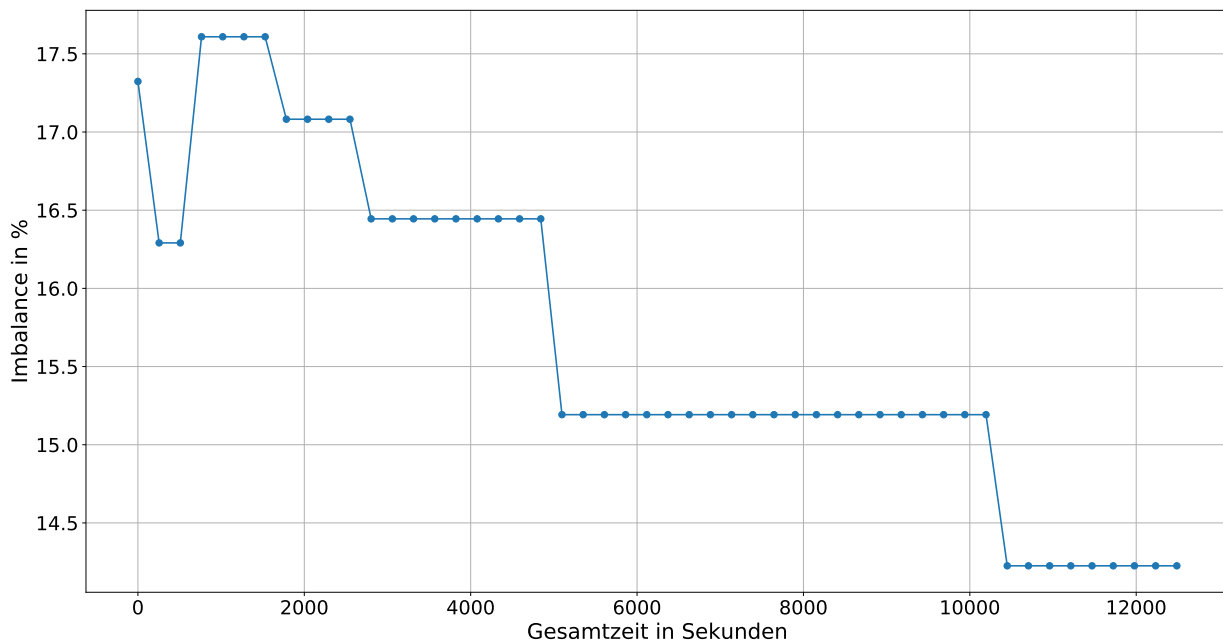


Abbildung 5.13: Imbalance für den 3,2-Mio.-Datensatz bei 256 Prozessoren und Alternating Crossover-Operator

5.3 Visualisierung der Partitionen

Schließlich ist es noch interessant zu sehen, wie die Gebiete aussehen, die von den Prototypen erzeugt werden. Es ist zu erwarten, dass die Gebiete nicht zusammenhängend sind, da bei der Erstellung des Modells kein Fokus daraufgelegt wurde. Zum Vergleich, die in den letzten Abschnitten erwähnten Richtwerte von raumfüllende Kurven erzeugen zusammenhängende Gebiete.

Bei der initialen, kartesischen Partitionierung, zu sehen in Abbildung 5.14 sind die Gebiete natürlich noch zusammenhängend. Allerdings entsteht am Ende eine nicht mehr zusammenhängende Partitionierung. Dies ist in den Abbildungen 5.15 für Gittergröße $8 \times 8 \times 8$ und 5.16 für Gittergröße $32 \times 32 \times 32$ zu sehen.

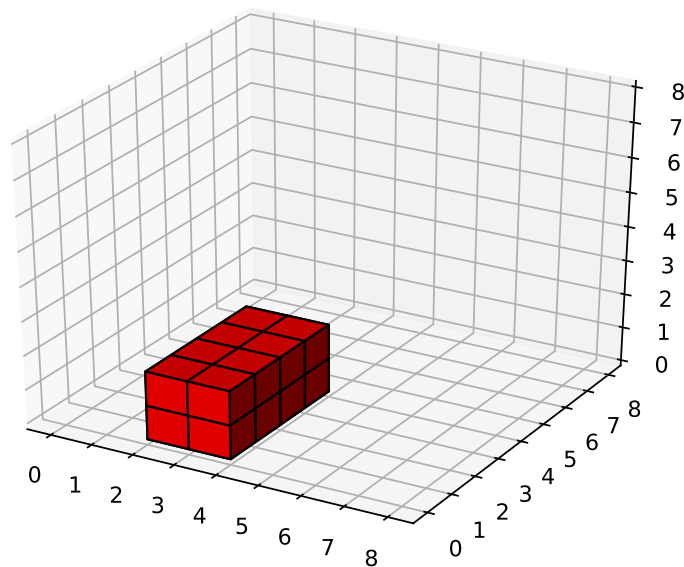


Abbildung 5.14: Partition des Prozessors Nr. 3 bei initialer, kartesischer Partitionierung für ein Gitter der Größe $8 \times 8 \times 8$

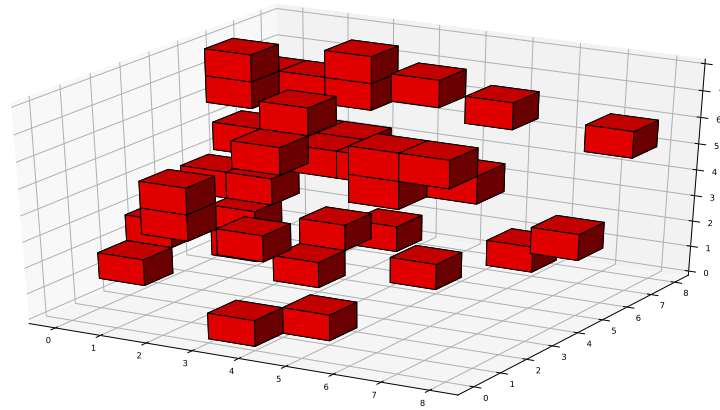


Abbildung 5.15: Partition des Prozessors Nr. 3 in der finalen Partitionierung für ein Gitter der Größe $8 \times 8 \times 8$

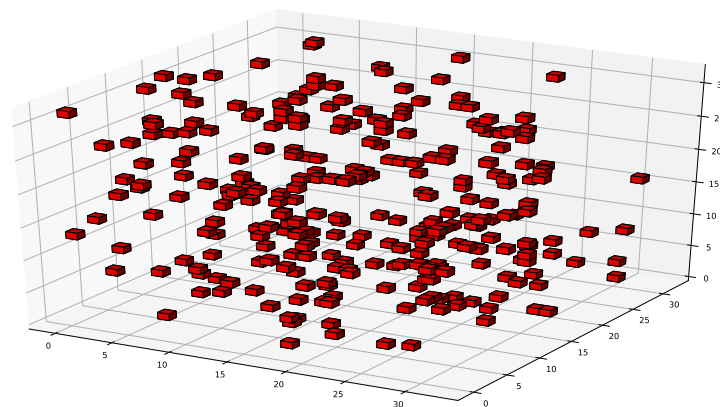


Abbildung 5.16: Partition des Prozessors Nr.25 in finaler Partitionierung für ein Gitter der Größe $32 \times 32 \times 32$

6 Zusammenfassung und Ausblick

In dieser Arbeit wird der Leser zunächst an das Themengebiet Molekulardynamik herangeführt und Grundlagen zu Partitionierung und deren Verbindung zum Problem der theoretischen Informatik `PARTITION` hergestellt. Danach erfolgt ein Überblick und eine kurze Beschreibung von in der Literatur typischen Partitionierungsverfahren. Anschließend stellen wir noch die Grundzüge von Simulated Annealing und genetischen Algorithmen vor, die später Kernkonzepte der beiden Prototypen sind.

In einer Literaturstudie zu Lastbalancierung und Partitionierungsverfahren in der Molekulardynamik werden Paper vor allem auf verwendete Modelle, Zielfunktionen und genutzte Maße hin analysiert. Daraus und in Verbindung zu `PARTITION` erstellen wir dann ein Modell für ein zu lösendes Optimierungsproblem. Dieses konnte nach ersten Testläufen mit Bibliotheken zur Lösung von Optimierungsproblemen weiter verfeinert werden.

Im Anschluss stellen wir zwei Prototypen vor, die Optimierungsprobleme stochastisch lösen. Dabei werden Simulated Annealing und genetische Algorithmen verwendet. Anschließend besprechen wir Eingabeparameter und die Implementierung des Modells als Kostenfunktion für beide Prototypen. Außerdem wird erläutert, wie wir eine Startpartitionierung /-population bilden. Für den Prototyp mit Simulated Annealing wird die Imbalance einer Partitionierung als Abbruchkriterium eingeführt. Später definieren wir für den zweiten Prototyp drei Crossover-Operatoren und ein Mutation-Operator, die besser an die gegebene Problemstellung in der Molekulardynamik angepasst sind. Die entstandenen Prototypen sollen später dafür genutzt werden die Qualität von Lösungen anderer Partitionierer zu überprüfen.

Schließlich werden die Ergebnisse einiger Testläufe verschiedenster Szenarien vorgestellt. Dabei variieren wir die Prozessorzahl, die Gittergröße, die Größe der Box, die Datensätze und die Verteilung der Partikel. Mit dem Simulated-Annealing-Prototyp kann für den 50k-Datensatz, ein Gitter der Größe $64 \times 64 \times 64$ und 32 Prozessoren eine bessere Imbalance als bei raumfüllenden Kurven erzielt werden. Der zweite Prototyp schneidet in vergleichbaren Szenarien schlechter ab, kann allerdings mit dem großen 3,2-Mio.-Datensatz in einem Bruchteil der Zeit ähnliche Ergebnisse wie Simulated Annealing liefern. Der darauf folgende Abschnitt mit Visualisierungen zeigt, dass die Implementierung der Kostenfunktion im Gegensatz zu raumfüllenden Kurven keine zusammenhängenden Partitionen liefert.

Ausblick

Im Zuge dieser Arbeit können nicht alle möglichen Aspekte der Problemstellung betrachtet werden. Für zukünftige Arbeiten wäre vor allem ein Einstieg beim Modell beziehungsweise der Kostenfunktion sinnvoll. Bisher noch völlig unbeachtet sind andere Faktoren, die man in die Kostenfunktion mit einfließen lassen kann. So könnte insbesondere auf zusammenhängende Gebiete bei der Partitionierungen und die Modellierung von Kommunikationskosten zwischen Prozessoren Wert gelegt

werden. Außerdem haben wir die Laufzeit der verwendete Algorithmen vollkommen außer Acht gelassen. So können vielleicht Verbesserungen in den Ergebnissen erzielt werden, wenn zuvor die Laufzeit der Prototypen optimiert wurde.

Literatur

- [BZ13] BUNGARTZ, H. ; ZIMMER, S. ; BUCHHOLZ, M. ; PFLÜGER, D.: *Modellbildung und Simulation*, 2013, 2. Auflage, <https://doi.org/10.1007/978-3-642-37656-6>, Springer-Verlag Berlin Heidelberg 2013
- [KG83] KIRKPATRICK, S. ; GELATT JR., C.D. ; VECCHI, M.P.: *Optimization by Simulated Annealing*, Science 1983, Volume 220 Issue 4598, Page 671-680, <https://doi.org/10.1126/science.220.4598.671>, American Association for the Advancement of Science
- [GO89] GOLDBERG, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, 1989, 1. Auflage, ISBN 0201157675, Addison-Wesley Longman Publishing Co., Inc.
- [PO09] POIRRIER, L.: *An efficient space partitioning technique based on linear kd-trees for simulation of short-range interactions in particle methods*, 2009
- [FE08] FLEISSNER, F. ; EBERHARD, P.: *Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection*, International Journal for Numerical Method in Engineering 2008, Volume 74 Issue 4, Page 531-553, <https://doi.org/10.1002/nme.2184>, John Wiley & Sons, Ltd.
- [BU10] BUCHHOLZ, M.: *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*, Verlag Dr. Hut, 2010
- [WGW06] WOLFHEIMER, F. ; GJONAJ, E. ; WEILAND, T.: *A parallel 3D particle-in-cell code with dynamic load balancing*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 2006, Volume 558 Issue 1, Page 202-204, <https://doi.org/10.1016/j.nima.2005.11.003>, Elsevier B.V.
- [NP97] NYLAND, L. ; PRINS, J. ; RU HUAI, Y.. ; HERMANS, J.. ; HYE-CHUNG, K.. ; WANG, L.: *Achieving Scalable Parallel Molecular Dynamics Using Dynamic Spatial Domain Decomposition Techniques*, Journal of Parallel and Distributed Computing 1997, Volume 47 Issue 2, Page 125-138, <https://doi.org/10.1006/jpdc.1997.1408>, Academic Press.
- [CG89] CYBENKO, G.: *Dynamic load balancing for distributed memory multiprocessors*, Journal of Parallel and Distributed Computing 1989, Volume 7 Issue 2, Page 279-301, [https://doi.org/10.1016/0743-7315\(89\)90021-X](https://doi.org/10.1016/0743-7315(89)90021-X), Elsevier Inc.
- [HB98] HU, Y. ; BLAKE, R.J. ; EMERSON, D.: *An optimal migration algorithm for dynamic load balancing*, Concurrency Practice and Experience 1998, Volume 10 Issue 6, [https://doi.org/10.1002/\(SICI\)1096-9128\(199805\)10:6<467::AID-CPE325>3.0.CO;2-A](https://doi.org/10.1002/(SICI)1096-9128(199805)10:6<467::AID-CPE325>3.0.CO;2-A)
- [WL93] WILLEBEEK-LEMAIR, M.H. ; REEVES, A.P.: *Strategies for dynamic load balancing on highly parallel computers*, IEEE Transactions on Parallel and Distributed Systems 1993, Volume 4 Issue 9, Page 979-993, <https://doi.org/10.1109/71.243526>, IEEE

- [BB91] BOILLAT, J.E. ; BRUGÉ, F. ; KROPF, P.G.: *A dynamic load-balancing algorithm for molecular dynamics simulation on multi-processor systems*, Journal of Computational Physics 1991, Volume 96 Issue 1, Page 1-14, [https://doi.org/10.1016/0021-9991\(91\)90263-K](https://doi.org/10.1016/0021-9991(91)90263-K), Elsevier Inc.
- [KL70] KERNIGHAN, B.W. ; LIN, S.: *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical Journal 1970, Volume 49 Issue 2, Page 291-307, <https://doi.org/10.1002/j.1538-7305.1970.tb01770.x>, Nokia Bell Labs
- [CB09] CATALYUREK, U.V. ; BOMAN, E. ; DEVINE, K.D. ; BOZDAG, D.: *A Repartitioning Hypergraph Model for Dynamic Load Balancing*, Journal of Parallel and Distributed Computing 2009, Volume 69 Issue 8, Page 711-724, <https://doi.org/10.1016/j.jpdc.2009.04.011>, Elsevier B.V.
- [BS15] BEGAU, C. ; SUTMANN, G.: *Adaptive dynamic load-balancing with irregular domain decomposition for particle simulations*, Computer Physics Communications 2015, Volume 190, Page 51-61, <https://doi.org/10.1016/j.cpc.2015.01.009>, Elsevier B.V.
- [PB96] PILKINGTON, J.R. ; BADEN, S.B.: *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*, IEEE Transactions on Parallel and Distributed Systems 1996, Volume 7 Issue 3, Page 288-300, <https://doi.org/10.1109/71.491582>, IEEE
- [XZ04] XIAOLIN, C. ; ZEYAO, M.: *A new scalable parallel method for molecular dynamics based on cell-block data structure*, ISPA'04: Proceedings of the Second international conference on Parallel and Distributed Processing and Applications 2004, Page 757-764, https://doi.org/10.1007/978-3-540-30566-8_88, Springer
- [SCMI] *Bibliothek: scipy, Methode: minimize*, <https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html>
- [BL95] BYRD, R.H. ; LU, P. ; NOCEDAL, J.: *A Limited Memory Algorithm for Bound Constrained Optimization*, SIAM Journal on Scientific and Statistical Computing 1995, Volume 16 Issue 5, Page 1190-1208, <https://doi.org/10.1137/09160692>, Society for Industrial and Applied Mathematics
- [ZB97] ZHU, C. ; BYRD, R.H. ; NOCEDAL, J.: *Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization*, ACM Transactions on Mathematical Software 1997, Volume 23 Issue 4, Page 550-560, <https://doi.org/10.1145/279232.279236>, Industrial Engineering and Management Sciences
- [SCIP] GLEIXNER, A. ; BASTUBBE, M. ; et al.: *The SCIP Optimization Suite 6.0*, ZIB-Report, Zuse Institut Berlin, 2018, Nummer 18-26 <http://nbn-resolving.de/urn:nbn:de:0297-zib-69361>, <https://scip.zib.de/#scipoptsuite>
- [XSFG97] XIANG, Y. ; SUN, D.Y. ; FAN, W. ; GONG, X.G.: *Generalized simulated annealing algorithm and its application to the Thomson model*, Physics Letters A 1997, Volume 233 Issue 3, Page 216-220, [https://doi.org/10.1016/S0375-9601\(97\)00474-X](https://doi.org/10.1016/S0375-9601(97)00474-X), Elsevier B.V.
- [XG00] XIANG, Y. ; GONG, X.G.: *Efficiency of generalized simulated annealing*, Physical Review E 2000, Volume 62 Issue 3, Page 4473-4476, <https://doi.org/10.1103/PhysRevE.62.4473>, American Physical Society

[SCDA] *Bibliothek: scipy, Methode: dual_annealing*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html#scipy.optimize.dual_annealing

[GAFT] *GAFT: A Genetic Algorithm Framwork in pyThon*, Github: <https://github.com/PytLab/gaft> ; Dokumentation: <https://gaft.readthedocs.io/en/latest/?badge=latest>

Alle URLs wurden zuletzt am 26.01.2020 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