Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 235

Verteiltes Dünngitter Clustering mit großen Datensätzen

Gregor Daiß

Studiengang:	Informatik
Prüfer/in:	Jun.-Prof. Dr. Pflüger
Betreuer/in:	Dipl.-Inf David Pfander
Beginn am:	25. Mai 2015
Beendet am:	25. November 2015
CR-Nummer:	I.5.3

Kurzfassung

Clustering ist ein Verfahren, das in vielen unterschiedlichen Disziplinen eingesetzt wird, um Muster in Daten zu erkennen. Wachsende Datenvolumen erfordern hierzu effiziente Algorithmen, welche auch große Datenmengen in akzeptabler Zeit clustern können. In dieser Arbeit wird zu diesem Zweck ein Clustering Verfahren genutzt, das auf einer Dichteschätzung mit dünnen Gittern und einem k-nearest-neighbors Verfahren basiert. Dieser Algorithmus ist gut geeignet um große, höherdimensionale Datensätze zu clustern und in verrauschten Datensätzen Cluster beliebiger Form zu suchen. Um die heutigen Parallelrechner ausnutzen zu können, wird zur Implementierung OpenCL verwendet. Zur weiteren Aufteilung des Problemes wird das Message Passing Interface genutzt, um das Clustering auf mehrere Rechner verteilen zu können. Mit dieser Implementierung wurde auf einem Rechencluster mit 64 Xeon Phi Coprozessoren ein zehndimensionaler, stark verrauschter Datensatz mit einer Million Datenpunkte in 1434 Sekunden geclustert, wobei sämtliche Cluster erkannt und 88.99% der Datenpunkte dem richtigen Cluster zugeordnet wurden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

1. Einleitung

Clustering bezeichnet die Klassifikation einer Menge von Datenpunkten in einzelne Gruppen, so dass die Datenpunkte aus gleichen Gruppen sich einander ähnlicher sind als die Datenpunkte aus unterschiedlichen Gruppen [JMF99]. Da jeder Datenpunkt ein d-dimensionaler Vektor ist, kann man diese Ähnlichkeit anhand der einzelnen Komponenten zweier dieser Vektoren feststellen. Falls die Datenpunkte beispielsweise räumliche Koordinaten darstellen, besteht ein Cluster aus den Datenpunkten, die eng zusammen liegen. Wir nutzen in dieser Arbeit die euklidische Distanz zwischen den Vektoren, um die Ähnlichkeit von Datenpunkten zu definieren.

Clustering wird in vielen Bereichen eingesetzt. Im Bereich des Data-Minings wird es verwendet, um Datenbanken nach Mustern zu durchsuchen. Anwendungsfälle hierfür sind beispielsweise die frühe Erkennung und Verhinderung von Krebs [RGB14] oder die Wettervorhersage [CND14]. Auch für die Einteilung von Kunden in bestimmte Gruppen zur Marktanalyse wird Clustering verwendet [Raj11]. Die Anwendungsgebiete von Clustering sind vielfältig, da man aus bereits vorhandenen Daten Wissen erzeugt. Da die Größe dieser Datenbanken durch den Einsatz von wissenschaftlichen Applikationen, sozialen Netzwerken und anderen Anwendungen stetig anwächst [CSD11], wird die Bedeutung von effizienten Clustering Algorithmen immer wichtiger. Allerdings müssen die gefundenen Cluster häufig von einem menschlichen Experten bewertet werden, da der Algorithmus selbst keine Interpretation liefert. Zur Durchführung des Clusterings in Datensätzen wurden viele Arten von Algorithmen entwickelt. Manche Algorithmen verwenden hierarchische Ansätze, bei denen ausgenutzt wird, dass Datenpunkte innerhalb einer Gruppe eine geringe Distanz zueinander haben. Andere wiederum nutzen Dichteschätzungen, oder verteilungsbasierte Verfahren.

Wir implementieren in dieser Arbeit den Clustering Algorithmus nach Peherstorfer [PPB12]. Dieser Algorithmus eignet sich gut für große Datensätze mit moderat vielen Dimensionen und kommt auch mit verrauschten Datensätze zurecht. In diesem Algorithmus verwendet ein hierarchischen Ansatz den k-nearest-Neighbors Algorithmus, verbessert diesen jedoch durch die Nutzung einer Dichteschätzung. Für die Erstellung der Dichteschätzung wird ein dünnes Gitter verwendet, um auch mit höhereren Dimensionen eine gute Laufzeit zu erzielen.

Zur Analyse der immer größer werdenden Datensätze ist dies alleine jedoch nicht ausreichend, um das Clustering in akzeptabler Zeit auszuführen. Um große Datensätze nach Clustern zu analysieren, müssen wir die vorhanden Ressourcen zum Rechnen ausnutzen können. Da die verwendeten Rechnerarchitekturen zunehmend parallel werden, ist es wichtig diese

Parallelrechner – wie die Intel Xeon Phi – auszunutzen zu können, als auch das Clustering Problem auf mehrere dieser Rechner aufteilen zu können. Zur Nutzung der Parallelrechner benutzen wir das OpenCL Framework [TS12]. Dieses erlaubt es uns, den Code auf mehreren unterschiedlichen Arten von Geräten auszuführen, sofern diese OpenCL unterstützen. Wir nutzen das Message Passing Interface (MPI), zur Aufteilung des Clustering Problems auf mehrere Rechner [JT97]. Wenn wir den Clustering Algorithmus mit diesen beiden Frameworks implementieren, ist es uns möglich, das Clustering auf unterschiedlichen Plattformen auszuführen und deren Parallelität auszunutzen, falls diese Plattformen über OpenCL-fähige Geräte verfügen. Da der OpenCL Code plattformunabhängig ist, ist es beispielsweise mit dem gleichen Programm möglich, sowohl einen Rechencluster mit Grafikkarten zu verwenden als auch einen Rechencluster, der Xeon Phi Coprozessoren verwendet.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Dünne Gitter: Dieses Kapitel beschreibt, wie wir mit Gittern Funktionen erzeugen können und was dünne Gitter sind.

Kapitel 3 – Erstellen einer Dichteschätzung beschreibt, wie man mit einem dünnen Gitter eine Schätzung der Dichte für einen Datensatz erstellen kann.

Kapitel 4 – Clustering Algorithmus: Hier wird die Funktionsweise der Clustering Algorithmus beschrieben.

Kapitel 5 – Paralleles Clustering auf einem Rechenknoten: Beschreibt die Implementierung dieses Algorithmus mit OpenCL.

Kapitel 6 – Verteiltes Clustering mit MPI beinhaltet eine Beschreibung der Verteilung des Algorithmus auf mehrere Rechenknoten mit MPI.

Kapitel 7 – Tests: In diesem Kapitel wird die Implentierung des verteilten Algorithmus auf mehreren Plattformen getestet. Untersucht werden die Skalierung bei unterschiedlicher Anzahl Rechenknoten und die Erkennungsrate bei verschiedenen Datensätzen.

Kapitel 8 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2. Dünne Gitter

Wie bereits in der Einleitung erwähnt, benötigen wir für den Clustering Algorithmus eine Dichteschätzung des Datensatzes. Diese Dichteschätzung erstellen wir mit einem Gitter. Auf diese Weise ist die Laufzeit von der Anzahl der Gitterpunkte abhängig anstatt von der Anzahl der Datenpunkte. Um auch auch in der Anzahl der Dimensionen skalieren zu können, werden wir statt regulären Gittern dünne Gitter verwenden [PPB12][BG04].

Im Folgenden gehen wir darauf ein was dünne Gitter sind und wie man Funktionen mit Gittern erstellt.

2.1. Mathematische Vorbemerkungen

Die Trägermenge $supp(f)$ einer Funktion f ist die Nichtnullstellenmenge dieser Funktion, also den Bereich in dem diese Funktion nicht 0 ist.

2.2. Eindimensionale volle Gitter

Die Idee ist es Funktionen durch die Linearkombination von Basisfunktionen zu approximieren. Das bedeutet, wir benötigen zuerst eine geeignete Menge an Basisfunktionen $\varphi(x) \in B$. Wir nutzen als Basisfunktionen stückweise definierte lineare Funktionen, auch Hutfunktionen genannt.

$$\varphi(x) = \begin{cases} 1 - |x| & \text{falls } x \in [-1, 1] \\ 0 & \text{sonst} \end{cases} \tag{2.1}$$

Die Funktion $\varphi(x)$ hat offensichtlich ihr Maximum bei $x = 0$, und die Trägermenge supp$(\varphi) = [0, 1]$. Da wir mehr als eine einzige Basisfunktion benötigen, verteilen wir diese Basisfunktionen mit äquidistant im Raum $\Omega = [0, 1]$. Wir wählen als Anzahl der Basisfunktionen $|B| = 2^l$.

Der Parameter l soll den Level des Gitters bezeichnen und $h_l = 2^{-l}$ den Abstand zwischen den Maximas der benachbarten Basisfunktionen. Die Trägermenge einer Basisfunktion soll

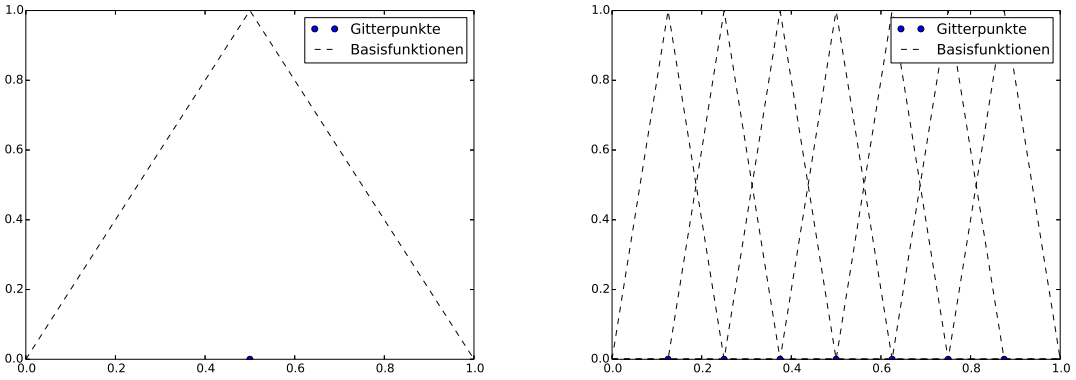


Abbildung 2.1.: Einzelne Hutfunktion (links), mehrere Hutfunktionen (rechts)

sich nur mit den der direkt benachbarten Basisfunktionen überschneiden. Mit diesen Überlegungen können wir nun die Basisfunktion so modifizieren, dass sie unseren Ansprüchen genügt. Durch i wird der Index der Basisfunktion bezeichnet.

$$\varphi_{l,i}(x) := \varphi\left(\frac{x - i \cdot h_l}{h_l}\right) \tag{2.2}$$

Die Maximas der Basisfunktionen eines Gitters des Levels l befinden sich also an den Stellen $x_{l,i}$.

$$x_{l,i} := i \cdot h_l, 1 \leq i \leq 2^l - 1 \tag{2.3}$$

Diese Stellen werden wir von nun an als Gitterpunkte bezeichnen. Jeder Basisfunktion $\varphi(x)_{l,i}$ wird nun ein Gitterpunkt an der Stelle $x_{l,i}$ zugeordnet.

Durch diese Basisfunktionen wird ein Funktionsraum V_l aufgespannt. Wir können nun wie gewohnt durch Linearkombination der Basisfunktionen, Funktionen aus V_l erzeugen.

$$V_l = \text{span}\left\{\varphi_{l,i}(x), 1 \leq i \leq 2^l - 1\right\} \tag{2.4}$$

Wenn wir nun beispielsweise eine Funktion $f : [0, 1] \to \mathbb{R}$ mit einem Element $u(x)$ aus V_l interpolieren wollen, brauchen wir nur die richtigen Koeffizienten für die Linearkombination der Basisfunktionen. Wie man in 2.3 sehen kann, sind an einem Gitterpunkt einer Basisfunktion $\varphi(x)_{l,i}$ alle anderen Basisfunktionen 0. Daher können wir, in diesem Fall, die Koeffizienten α_i direkt durch den Funktionswert von f an dem Gitterpunkt $x_{l,i}$ bestimmen.

$$\alpha_i = f(i \cdot h_l) \tag{2.5}$$

Mit diesen α_i können durch Linearkombination die Interpolante $u(x)$ bestimmen.

$$f(x) \approx u(\vec{x}) = \sum_{i=1}^{N} \alpha_i \varphi_i(\vec{x}) \tag{2.6}$$

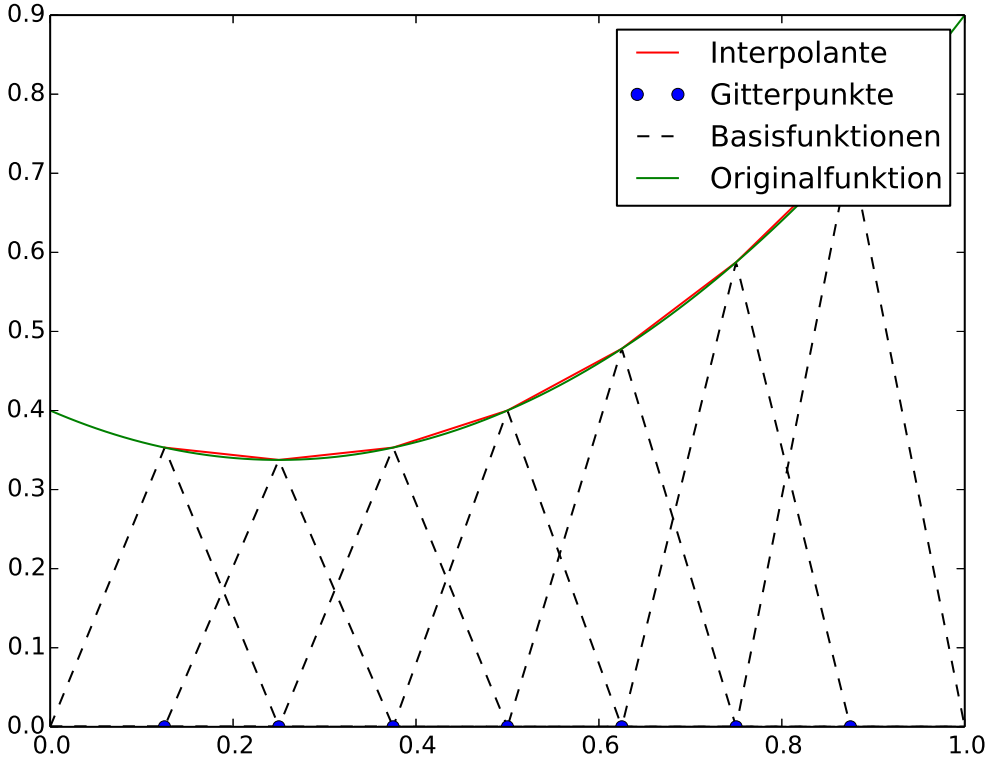


Abbildung 2.2.: Interpolante u für f

2.3. Mehrdimensionale volle Gitter

Von diesem eindimensionalen Fall können wir nun auf den mehrdimensionalen Fall erweitern. Hier ist unser Urbildraum für die Funktionen $\Omega = [0,1]^d$, wobei d die Anzahl der Dimensionen ist.

Wie zuvor wollen wir wieder äquidistant Basisfunktionen auf diesen Raum verteilen.

Die Parameter l und i sind hier jedoch keine Skalare mehr, sondern Vektoren. Wir haben also einen Level $\vec{l} = \begin{pmatrix} l_1 \\ l_2 \\ \vdots \\ l_d \end{pmatrix} \in \mathbb{N}^d$ und einen Index $\vec{i} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_d \end{pmatrix}$. Die neuen Basisfunktionen für

den mehrdimensionalen Raum, ergeben sich so per Tensorprodukt aus den eindimensionalen Basisfunktion, die wir zuvor benutzt haben.

$$\varphi\left(\vec{x}\right) := \prod_{j=1}^{d} \varphi_{l_j,i_j}\left(x_j\right) \tag{2.7}$$

Im eindimensionalen Fall hatten wir 2^l Basisfunktionen, und damit ebenso viele Gitterpunkte. Im mehrdimensionalen Fall haben wir nun für jede Dimension einen Level, das bedeutet die Anzahl unserer Basisfunktionen lässt sich wie folgt berechnen.

$$N = |B| = \prod_{z=1}^{d} 2^{l_z} \tag{2.8}$$

Falls wir in jeder Dimension denselben Level haben, lässt sich die Anzahl der Dimensionen auch einfacher aufschreiben.

$$N = h_l^{-d} \tag{2.9}$$

Wir könnten anstelle der Hutfunktionen natürlich auch andere Basisfunktionen nutzen [PPB10]. Im Zuge dieser Arbeit beschränken wir uns jedoch auf die oben genutzten Hutfunktionen. Ein Nachteil der von uns gewählten Basisfunktionen ist, dass die Funktionen aus V_l am Rand unseres Bereiches Ω nur 0 sein können. Sobald die Koeffizienten α_i für das Gitter bestimmt sind und man sich für eine geeignete Menge an Basisfunktionen φ_i entschieden hat, ist die Funktion des Gitters definiert. Dies könnte wie oben die Interpolante einer Funktion sein, oder aber auch eine Dichteschätzung. Für ein Dichteschätzung müssen wir jedoch $\vec{\alpha}$ anders bestimmen.

Die Nutzung eines vollen Gitters zur Diskretisierung einer Funktion hat allerdings zwei Nachteile. Erstens steigt die Anzahl der Gitterpunkte exponentiell mit der Anzahl der Dimensionen (Fluch der Dimensionalität [BG04]). Das erhöht nicht nur den Speicheraufwand und den Aufwand zur Bestimmung der Koeffizienten, sondern verlangsamt auch die Auswertung der Funktion des Gitters, da jede Basisfunktion ausgewertet werden muss. Zweitens befinden sich die Gitterpunkte nicht immer an den Stellen, an denen sie benötigt werden. So haben wir bei einem Gitter mit äquidistanten Gitterpunkten Gegenden mit vielen überflüssigen Gitterpunkten, dafür in anderen Gegenden zu wenige. Es ist wünschenswert, die Anzahl der Gitterpunkte zu reduzieren und dabei nur die Gitterpunkte zu entfernen, die weniger zum Ergebnis beitragen.

2.4. Hierarchische Basis für eindimensionale Gitter

Hierzu müssen wir Gitterpunkte entfernen. Dazu stellen wir das Gitter zunächst nicht einfach durch eine Menge an Gitterpunkten dar, sondern durch eine hierarchische Verteilung der Gitterpunkte. Die Anzahl der Gitterpunkte selbst und ihre Position im Raum Ω ändert sich

nicht. Allerdings werden die Gitterpunkte anders adressiert. Dies erlaubt es uns später einfacher, geeignete Gitterpunkte zu finden, die wir entfernen können.

Bei der hierarchischen Verteilung werden die Gitterpunkte in verschiedene Level unterteilt. Im Gegensatz zu den vollen Gittern bei denen jede Basisfunktion denselben Level hatte, hat hier jede Basisfunktion ihren eigenen Level. So bilden sich Unterfunktionsräume von Basisfunktionen, welche denselben Level haben.

Jeder Gitterpunkt gehört zu einem bestimmten Level in der Hierarchie wie in 2.5 dargestellt. Man erkennt in dieser Abbildung ebenfalls, dass der Level in dem sich ein Gitterpunkt befindet, Einfluss auf dessen Basisfunktion hat.

Da wir die Anzahl der Basisfunktionen jedoch im Vergleich zu den vollen Gittern nicht erhöhen wollen, müssen wir die Menge der Indices verändern. Jeder Level bekommt nun eine eigene Indexmenge. Da die Gitterpunkte selbst später an den Stellen $x_{l,i} = i \cdot h_l$ liegen kann man sich leicht klar machen, dass keine zwei Gitterpunkte des gesamten Gitters an derselben Stelle liegen, falls i ungerade ist.

$$\mathcal{I}_l := \left\{ i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, \text{i ungerade} \right\} \tag{2.10}$$

Den Funktionsraum, den alle Basisfunktionen eines Levels aufspannen, nennen wir W_l.

$$W_l := \text{span} \left\{ \varphi_{l,i} : i \in \mathcal{I}_l \right\} \tag{2.11}$$

Der Funktionsraum des gesamten Gitters ergibt sich als direkte Summe aus den Funktionsräumen der einzelnen Level. Die Anzahl dieser Funktionsräume, die man nutzen will, ist durch die Variable L begrenzt. Diese drückt hier den Maximallevel aus.

$$V_L = \bigoplus_{l \leq L} W_l \tag{2.12}$$

In der Abbildung 2.4 ist abgebildet, wie die Interpolation der vorherigen Funktion nun in hierarchischer Basis aussieht. Man erkennt, dass die Basisfunktionen anders aussehen und dass man daher einen anderen Koeffizientenvektor $\vec{\alpha}$ benötigt. Die Interpolationsfunktion selbst sieht jedoch exakt gleich aus.

2.5. Hierarchische Basis für mehrdimensionale volle Gitter

Dieses Konzept lässt sich nun auch wieder auf mehrdimensionale Fälle erweitern. Zunächst müssen wir hierfür das Gitter auf den $[0, 1]^d$ erweitern. Dazu nutzen wir wieder die hierarchische Basis, nur dieses Mal wird einem Gitterpunkt nicht nur ein Level zugeordnet, sondern d Level. Für jede Dimension hat ein Gitterpunkt also einen Index und einen Level. Der Level

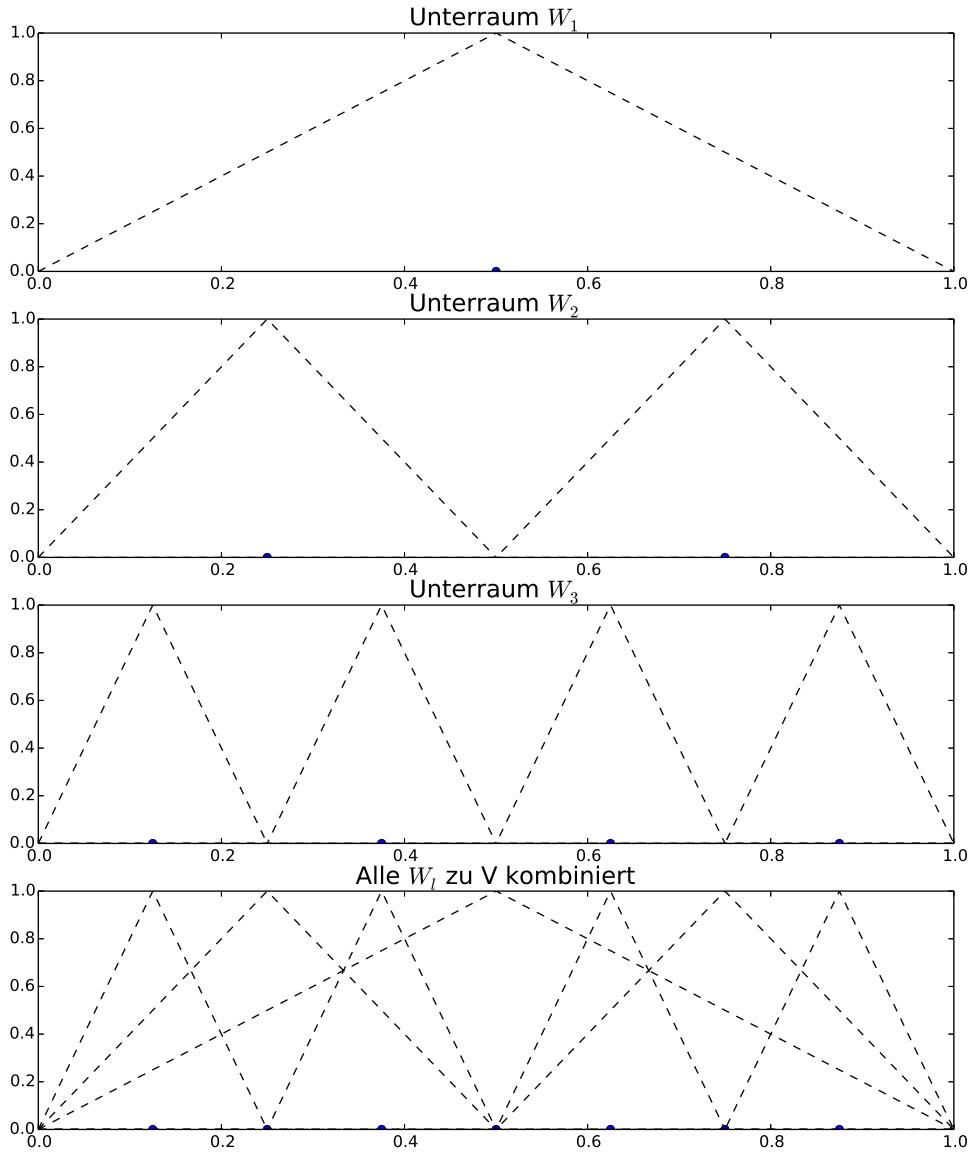


Abbildung 2.3.: Aufteilung der Gitterpunkte (blau) in einer hierarchischen Basis

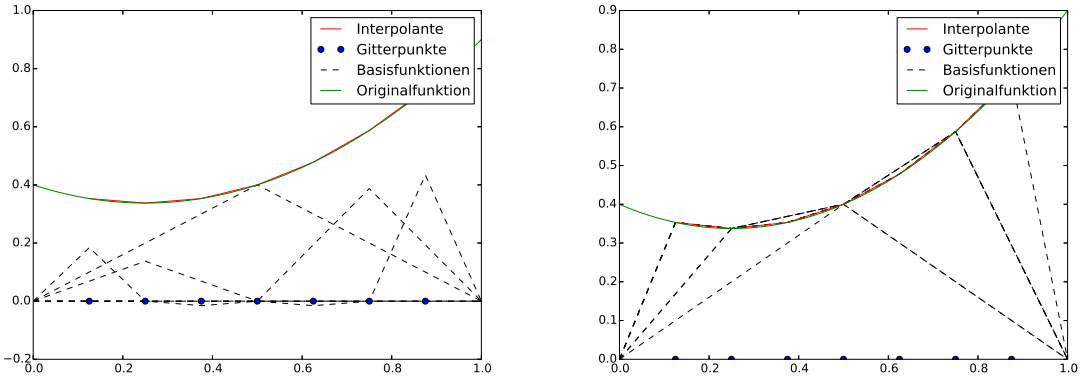


Abbildung 2.4.: Interpolation in hierarchischer Basis dargestellt (links), Interpolation mit bereits addierten Basisfunktionen dargestellt (rechts)

wird mit dem Multi-Index $\vec{l} := (l_1, l_2, ..., l_d)$ ausgedrückt. Jedem dieser Multi-Indices wird nun, wie zuvor, wieder eine Indexmenge $\mathcal{I}_{\vec{l}}$ zugeordnet.

$$\mathcal{I}_{\vec{l}} := \left\{ \vec{i} \in \mathbb{N}^d : 1 \leq i_j \leq 2^{l_j} - 1, i_j \text{ ungerade}, 1 \leq j \leq d \right\} \tag{2.13}$$

Die Basisfunktion eines Gitterpunktes erhalten wir nun als Tensorprodukt der entsprechende Basisfunktionen der einzelnen Dimensionen.

$$\varphi\left(\vec{x}\right)_{\vec{l},\vec{i}} := \prod_{j=1}^{d} \varphi_{l_j,i_j}\left(\vec{x}\right) \tag{2.14}$$

Auch hier ergibt sich wieder für jeden Level $\vec{l}$ ein Funktionsraum $W_{\vec{l}}$

$$W_{\vec{l}} := \text{span}\left\{ \varphi_{\vec{l},\vec{i}}\left(\vec{x}\right) : \vec{i} \in \mathcal{I}_{\vec{l}} \right\} \tag{2.15}$$

Der gesamte Funktionsraum ist wieder die direkte Summe der Unterfunktionsräume W.

$|\vec{l}|_\infty$ Bezeichnet wie gewohnt die Maximumsnorm.

$$V_L = \bigoplus_{|\vec{l}|_\infty \leq L} W_{\vec{l}} \tag{2.16}$$

In 2.5 sehen wir, wie ein volles Gitter in hierarchischer Basis aussehen kann. Bisher haben wir allerdings nur unsere Darstellung des vollen Gitters geändert. Wir haben noch immer die gleiche Anzahl der Gitterpunkte wie zuvor. Nun können wir aber die hierarchische Darstellung des Gitters nutzen, um Unterfunktionsräume auszulassen und so Gitterpunkte einzusparen.

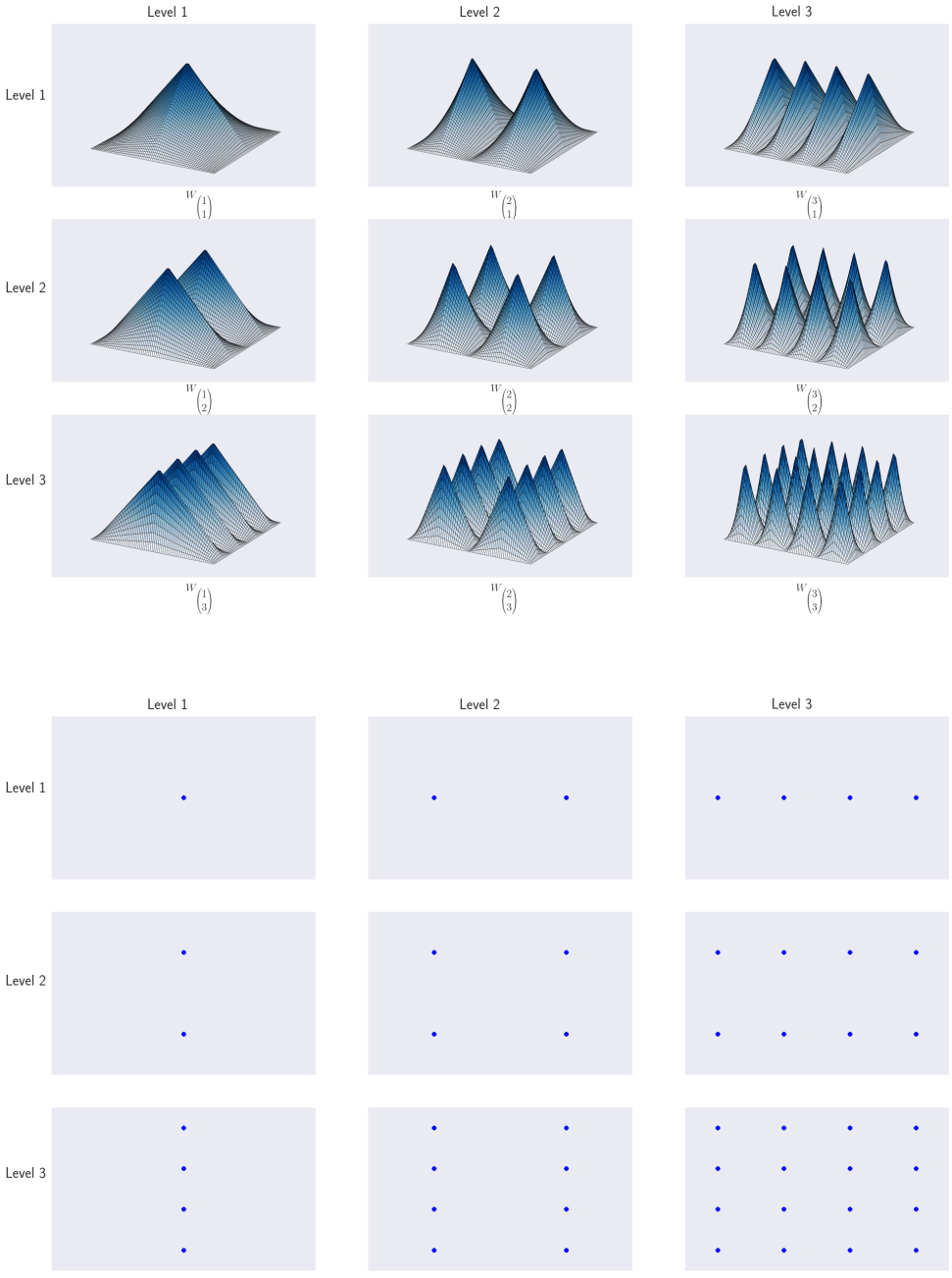


Abbildung 2.5.: Funktionsunterräume und Gitterpunkte eines vollen Gitters in hierarchischer Basis

2.6. Dünne Gitter

Der Funktionsraum des vollen Gitters beinhaltet im d-dimensionalen Fall alle Unterfunktionsräume $W_{\vec{l}}$, für die der größte Level $|\vec{l}|_\infty$ kleiner ist als die festgelegte Levelgrenze L. Auf diese Weise erhalten wir am Ende noch immer h_n^{-d} viele Gitterpunkte. Bisher haben wir durch die hierarchische Darstellung noch keinerlei Gitterpunkte einsparen können. Gerade das exponentielle Wachstum der Anzahl der Gitterpunkte bei steigender Anzahl der Dimensionen ist ein großes Problem. Um die Anzahl der Gitterpunkte auszudünnen, können wir nun aber die hierarchische Darstellung des Gitters ausnutzen. Dazu entfernen wir einige der Unterfunktionsräume $W_{\vec{l}}$. Bei einem vollen Gitter ist das Kriterium für die Auswahl

$$|\vec{l}|_\infty \leq L \tag{2.17}$$

Bei einem dünnen Gitter schränken wir die Auswahl der Unterfunktionsräume weiter ein.

$$\sum_{j=1}^{d} l_j \leq L + d - 1 \tag{2.18}$$

Ein dünnes Gitter erzeugt den Funktionsraum

$$V_L := \bigoplus_{\sum_{j=1}^{d} l_j \leq L+d-1} W_{\vec{l}} \tag{2.19}$$

Wenn wir nun beispielsweise das volle Gitter aus 2.5 ausdünnen, erhalten wir das dünne Gitter 2.6. Sobald der Koeffizienten Vektor α bestimmt ist, kann man also wie folgt die durch das Gitter erzeugte Funktion an einer Stelle $\vec{x}$ auswerten.

$$u(\vec{x}) = \sum_{s \leq L+d-1} \sum_{\vec{i} \in \mathcal{I}_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}) \tag{2.20}$$

$$s = \sum_{j=1}^{d} l_j \tag{2.21}$$

Um die Bezeichnungen im Folgenden übersichtlich zu halten, nutzen wir allerdings eine durchgehende Nummerierung für den α Vektor und die Basisfunktionen. Hierzu ist N die Anzahl der Gitterpunkte und i eine durchgehende Nummerierung.

$$u(\vec{x}) = \sum_{i=1}^{N} \alpha_i \varphi_i(\vec{x}) \tag{2.22}$$

Nachdem wir nun in einem dünnen Gitter weniger Basisfunktionen verwenden, als in dem entsprechenden vollen Gitter, stellt sich die Frage, wie sich die Genauigkeit entwickelt. Dies wurde in [BG04] untersucht. Der Verlust an Genauigkeit ist verhältnismässig gering, obwohl wir gerade in hohen Dimensionen viele Basisfunktionen einsparen. Unser Rechenaufwand und Speicheraufwand für höherdimensionale Gitter nimmt also bei dünnen Gittern, im Vergleich zu vollen Gittern, stark ab und wir haben dennoch eine ähnliche Genauigkeit. Ein genauerer Vergleich ist in Tabelle 2.1 zu sehen.

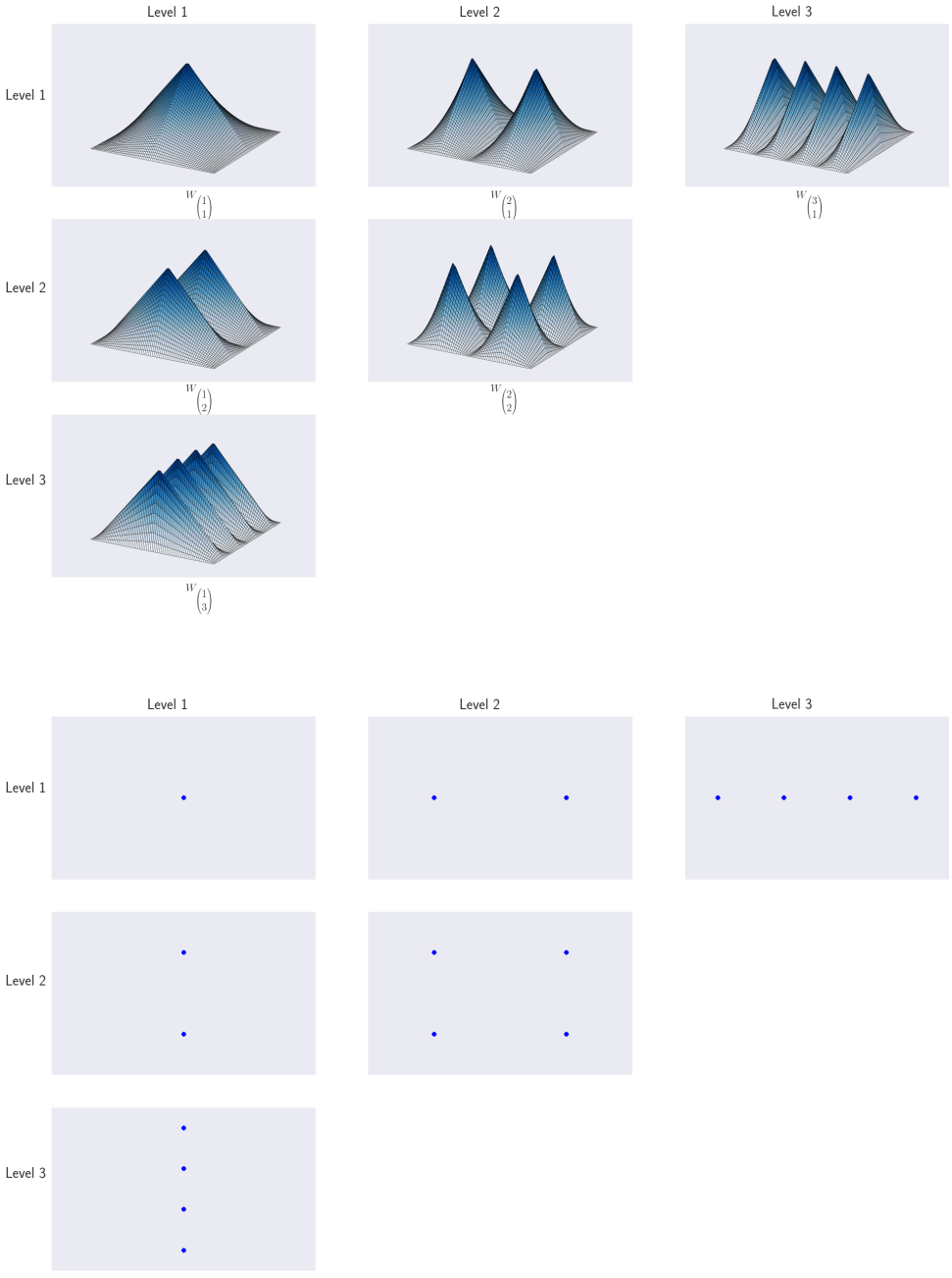


Abbildung 2.6.: Funktionsunterräume und Gitterpunkte eines dünnen Gitters in hierarchischer Basis

	Anzahl der Gitterpunkte	**Asymptotische Genauigkeit**
Volles Gitter	$\mathcal{O}\left(h_n^{-d}\right)$	$\mathcal{O}\left(h_n^2\right)$
Dünnes Gitter	$\mathcal{O}\left(h_n^{-1}\left(logh_n^{-1}\right)^{d-1}\right)$	$\mathcal{O}\left(h_n^2\left(logh_n^{-1}\right)^{d-1}\right)$

Tabelle 2.1.: Asymptotische Analyse aus [PPB10]

3. Erstellen einer Dichteschätzung

In diesem Kapitel stellen wir eine Variationsgleichung vor, mit der wir die Abschätzung der Dichte eines Datensatzes erhalten. Dazu stellen wir zunächst eine Variationsgleichung vor, mit der wir das Problem beschreiben können, und erstellen mit dieser Variationsgleichung eine lineares Gleichungssystem um die Koeffizienten $\vec{\alpha}$ zu erhalten. Mit diesen Koeffizienten können wir dann die Dichteschätzung als Linearkombination der Basisfunktionen eines Gitters erstellen.

3.1. Variationsgleichung

Es sei ein d-dimensionaler Datensatz D mit M Datenpunkten gegeben. Zunächst benötigen wir eine Variationsgleichung, deren Lösung eine Dichteschätzung ergibt. Dafür nutzen wir den Ansatz aus [HHR00]:

$$u(x) = \underset{u \in V}{\operatorname{argmin}} \left(\int_{\Omega} \left(u\left(x\right) - f_{\epsilon} \right)^2 dx + \lambda \int_{\Omega} \left(Lu \right)^2 dx \right) \tag{3.1}$$

Diese Variationsgleichung besagt zu einem, dass die gesuchte Funktion $u\left(\vec{x}\right)$ möglichst nahe an einer initialen Dichteschätzung f_{ϵ} liegen soll. Zum anderen soll die Dichtefunktion $u\left(x\right)$ im Vergleich zur initialen Schätzung geglättet werden. Der Term $\int_{\Omega} \left(u\left(x\right) - f_{\epsilon} \right)^2 dx$ bewirkt, dass die Abweichung möglichst gering von der initialen Dichteschätzung ist.

Der Term $\lambda \int_{\Omega} \left(Lu \right)^2 dx$ wird Regularisierungsterm genannt. Durch diesen wird bestimmt, wie glatt die Ergebnisfunktion der Variationsgleichung wird. Dies wird über den Regularisierungs Parameter λ bestimmt.

Nach einigen Umformungen erhält man die Variationsgleichung 3.2. Diese muss für alle Funktionen $s \in V$ erfüllt sein, wobei V der Testraum ist.

$$\int_{\Omega} s\left(x\right) \left(u\left(x\right) - f_{\epsilon} \right) dx + \lambda \int_{\Omega} Lu\left(x\right) Ls\left(x\right) dx = 0 \;\; \forall s \in V \tag{3.2}$$

Um diese Gleichung zu erfüllen, benötigen wir noch eine initiale Dichteschätzung f_{ϵ}, einen passenden Regularisierungsterm, der die Funktion glättet und einen Testraum V.

3.2. Initiale Dichteschätzung f_ϵ

Für die initiale Dichteschätzung gibt es mehrere Möglichkeiten, unter denen wir wählen können [HHR00]. Wir benutzen, wie in [PPB12], die folgende Funktion als initiale Dichteschätzung.

$$f_\epsilon = \frac{1}{M} \sum_{z=1}^{M} \delta_{x_z} \tag{3.3}$$

Hier ist δ_{x_i} die Dirac Funktion mit x_i als Zentrum. x_z sind die Datenpunkte unseres Datensatzes $x_z \in D$.

Durch das Einsetzen der initialen Dichteschätzung erhalten wir die Gleichung 3.4.

$$\int_\Omega s\left(x\right) \left(u\left(x\right) - \frac{1}{M} \sum_{z=1}^{M} \delta_{x_z} \right) dx + \lambda \int_\Omega Lu\left(x\right) Ls\left(x\right) dx = 0 \;\; \forall s \in V \tag{3.4}$$

Wenn wir nun noch wie in [HHR00] davon ausgehen, dass unsere initiale Schätzung nah an den Daten liegt, können wir folgende Approximation verwenden.

$$\int_\Omega \frac{1}{M} \sum_{j=1}^{M} \delta_{x_j} s(x) f_\epsilon dx \approx \frac{1}{M} \sum_{z=1}^{M} s(x_z) \tag{3.5}$$

Mit dieser Approximation und einer Umstellung der Terme erhalten wir

$$\int_\Omega s\left(x\right) u\left(x\right) dx + \lambda \int_\Omega Lu\left(x\right) Ls\left(x\right) dx = \frac{1}{M} \sum_{z=1}^{M} s(x_z) \;\; \forall s \in V \tag{3.6}$$

3.3. Testraum V

Als nächstes wählen wir den Testraum V. Bisher müssten unsere Koeffizienten noch unendlich vielen Gleichungen erfüllen, da sie für alle Funktionen eines Testraums erfüllt sein müssen. Wir wenden das übliche Ritz-Galerkin Verfahren an und nehmen somit als Testraum V_l, also den gleichen Funktionsraum der durch unser Gitter erzeugt wird. Die Gleichung muss nun nur noch für alle Basisfunktionen des Testraums erfüllt sein.

$$\int_\Omega \varphi_j\left(x\right) u\left(x\right) dx + \lambda \int_\Omega Lu\left(x\right) Ls\left(x\right) dx = \frac{1}{M} \sum_{z=1}^{M} \varphi_j(x_z) \;\; \forall \varphi_j \in B \tag{3.7}$$

Wir haben nun N Gleichungen, eine für jede Basisfunktion des Gitters. Wenn wir ein $u(x)$ finden, das alle diese Gleichungen erfüllt, haben wir die Lösung der Variationsgleichung.

3.4. Regularisierungsterm

Bevor wir diese Gleichungen lösen, benötigen wir allerdings noch einen geeigneten Regularisierungsterm. Als solchen wählen wir die euklidische Norm wie in [PPB12] vorgeschlagen.

$$\int_{\Omega} \left(Lu\left(x\right)\right)^2 dx = \sum_{i=1}^{N} \alpha_i^2 \tag{3.8}$$

Der Vorteil dieses Regularisierungsterms ist, dass man später eine Identitätsmatrix erhält, was die Matrix-Vektor Multiplikation erleichtert.

3.5. Resultierendes LGS

Wenn wir diesen Term und die erzeugende Formel für u nun noch einsetzen, erhalten wir ein LGS, das wir nach $\vec{\alpha}$ auflösen können.

$$\sum_{i=1}^{N} \alpha_i \int_{\Omega} \varphi_j\left(x\right) \varphi_i\left(x\right) dx + \lambda \sum_{i=1}^{N} \alpha_i = \frac{1}{M} \sum_{z=1}^{M} \varphi_j(x_z) \;\; \forall \varphi_j \in B \tag{3.9}$$

Wenn man die Terme umsortiert, sieht man, dass die linke Seite des Gleichungssystems eine Matrix-Vektor Multiplikation ist und die rechte Seite ein Vektor.

Der Regularisierungsterm besteht hier nur noch aus einem λ was das Matrix-Vektor Produkt erleichtert. Der Term $\left(\varphi_i\left(x\right), \varphi_k\left(x\right)\right)_{L2}$ bezeichnet wie üblich das L2 Skalarprodukt.

$$\left(\int_{\Omega} \varphi_j\left(x\right) \varphi_i\left(x\right) dx + \lambda 1\right) \sum_{i=1}^{N} \alpha_i = \frac{1}{M} \sum_{z=1}^{M} \varphi_j(x_z) \;\; \forall \varphi_j \in B \tag{3.10}$$

In Matrixform sieht dieses Gleichungssystem nun wie folgt aus:

$$\left(\mathcal{A} + \lambda I\right) \alpha = \frac{1}{M} \mathcal{B} \cdot \vec{1} \tag{3.11}$$

Die Matrix $\mathcal{A}$ bezeichnet die Matrix der einzelnen L2 Skalarprodukte.

$$\mathcal{A}_{i,k} = \left(\varphi_i, \varphi_k\right)_{L2} \tag{3.12}$$

Das Matrix-Vektor Produkt $\mathcal{B}$ bezeichnet die rechte Seite unseres Gleichungssystems.

$$\mathcal{B}_{i,j} = \varphi_i\left(x_j\right) \tag{3.13}$$

Wir müssen nun nur noch das Gleichungssystem 3.11 lösen um eine Funktion zu erhalten, die unsere Variationsgleichung 3.1 erfüllt.

4. Clustering Algorithmus

In diesem Kapitel gehen wir darauf ein, was Clustering eigentlich ist und welche Art von Algorithmus dafür wir verwenden. Danach werden wir die einzelnen Schritte des gewählten Clustering Algorithmus im Detail erläutern.

4.1. Clustering Ansätze

Das Einordnen von Datenpunkte in verschiedene Gruppen nennt man Klassifizieren. Wir haben Datenpunkte aus $x \in \Omega$ und möchten diese auf eine Menge diskreter Klassen $y \in \{1, ..., C\}$ abbilden. Falls wir nun keine Menge von Klassen, auf die wir die Datenpunkte abbilden können, zur Verfügung haben, sprechen wir von unüberwachter Klassifikation [XW+05]. Bei dieser müssen wir selbst die Klassen entdecken und die Datenpunkte diesen Klassen zuordnen. Diesen Vorgang nennt man Clustering [XW+05]. Das Clustering Problem ist sehr alt und kann bereits auf Aristoteles zurückgeführt werden [HJ97].

Da wir beim Clustering keine vordefinierte Klassen zur Verfügung haben, müssen wir versteckte Strukturen innerhalb des Datensatzes finden und die Datenpunkte diesen zuordnen. Anders formuliert suchen wir nach Datenpunkten, die einander ähnlich sind. Eine Menge solcher, sich ähnlichen, Datenpunkte nennt man Cluster. Eine mögliche Metrik für die Ähnlichkeit zweier Datenpunkte ist die euklidische Distanz. Diese werden wir verwenden, um zwei Datenpunkte miteinander zu vergleichen. Es sind jedoch auch andere Metriken möglich, wie in [XW+05] zu sehen ist. Mit einer solchen Metrik ist es möglich Daten, die sich ähnlich sind und so ein Cluster bilden, zu finden. Sobald man eines dieser Cluster gefunden hat, hat man eine Klasse mehr auf die die Datenpunkte einfach abgebildet werden können.

Da Clustering in vielen unterschiedlichen Disziplinen genutzt wird, gibt es mehrere, unterschiedliche Ansätze zum Clustering [Mad12] [OES07]. Beim hierarchischen Clustering startet man mit einem großen Cluster, das nach und nach aufgeteilt wird, oder vielen kleinen Clustern, die später zu größeren Clustern kombiniert werden. Es gibt außerdem noch das Clustering mit Dichtefunktion, bei dem Cluster anhand Gebieten mit hoher Dichte identifiziert werden können, das Clustering mit Graphentheorie und mehrere andere [XW+05]. Diese Ansätze unterscheiden sich bezüglich Skalierung in Anzahl der Datenpunkte und Dimensionen. Da heutige Datensätze, durch den billiger werdenden Speicher und vielen Sensoren ständig größer werden, ist es wichtig, dass der von uns gewählte Algorithmus mit

großen Datensätzen und moderat vielen Dimensionen zurecht kommt. Zwar steigt die uns zur Verfügung stehende Rechenleistung auch mit der Zeit, allerdings muss ein Algorithmus auch gut parallelisierbar sein, um diese Leistung nutzen zu können.

Wir nutzen in dieser Arbeit den Clustering Algorithmus von [PPB12]. Dieser kommt gut mit großen Datensätzen zurecht und, dank der Verwendung von dünnen Gittern, auch mit moderat vielen Dimensionen. Außerdem hat der Algorithmus den Vorteil, dass er auch bei verrauschten Datensätzen Cluster erkennen kann, wie in Abbildung 4.1 zu sehen ist. Im Gegensatz zu vielen anderen Algorithmen, ist dieser zudem in der Lage, andere Clusterformen als die üblichen Hypersphären und Hyperellipsen, zu entdecken. Dies ist in Abbildung 4.2 zu erkennen: hier wird zum Beispiel ein Ring als Cluster erkannt. Solange die Datenpunkte ausreichend eng aneinander liegen kann, ist es möglich beliebige Formen von Clustern zu erkennen.

Der Algorithmus besteht aus einem k-nearest-neighbors Verfahren, das mit Hilfe einer Dichteschätzung verfeinert wird. Wie diese Dichteschätzung mit Hilfe eines dünnen Gitters erstellt wird, wurde bereits in Kapitel 3 erläutert. Durch die Nutzung von dünnen Gittern anstatt vollen Gittern, können wir auch Datensätzen mit 10 oder mehr Dimensionen in annehmbarer Zeit bearbeiten.

Im Wesentlichen müssen wir vier Arbeitsschritte durchführen, um die Cluster eines Datensatzes D mit diesem Algorithmus zu finden. Wir gehen davon aus, dass sich dieser Datensatz in dem Raum $\Omega = [0, 1]^d$ befindet.

1. Erstellen des k-nearest-neighbors Graphen G
2. Erstellen einer Dichteschätzung $\kappa(\vec{x}) : \Omega \to \mathbb{R}$
3. Entfernen von Knoten und Kanten in Gebieten mit geringer Dichte aus G zur Erstellung eines neuen Graphen G'
4. Suche nach Zusammenhangskomponenten im Graph G'

Wir gehen nun im Detail auf die vier Arbeitsschritte ein, beginnend mit der Erstellung des Graphen G mit dem k-nearest-neighbors Algorithmus.

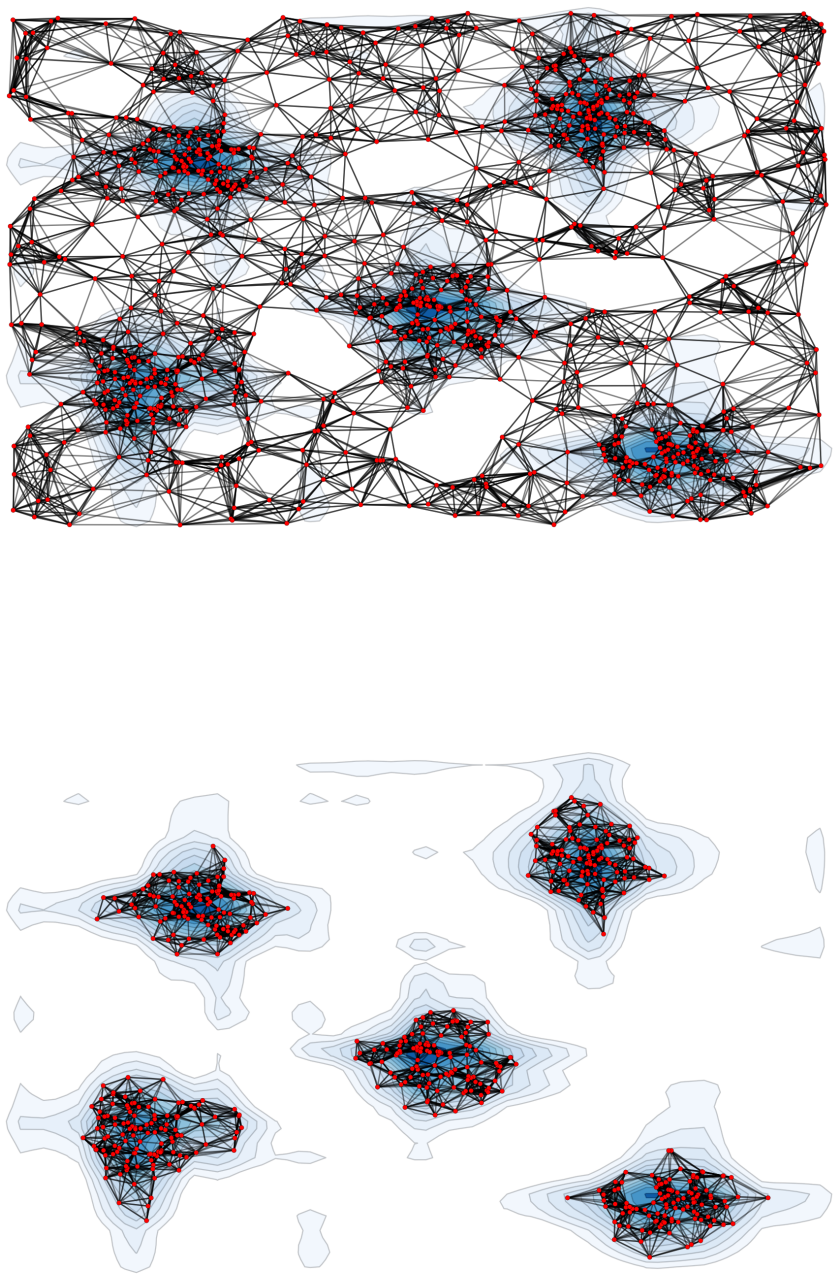


Abbildung 4.1.: Reines k-nearest-neighbors Clustering (oben) und mit Nutzung einer Dichteschätzung (unten) bei einem Datensatz mit Rauschen

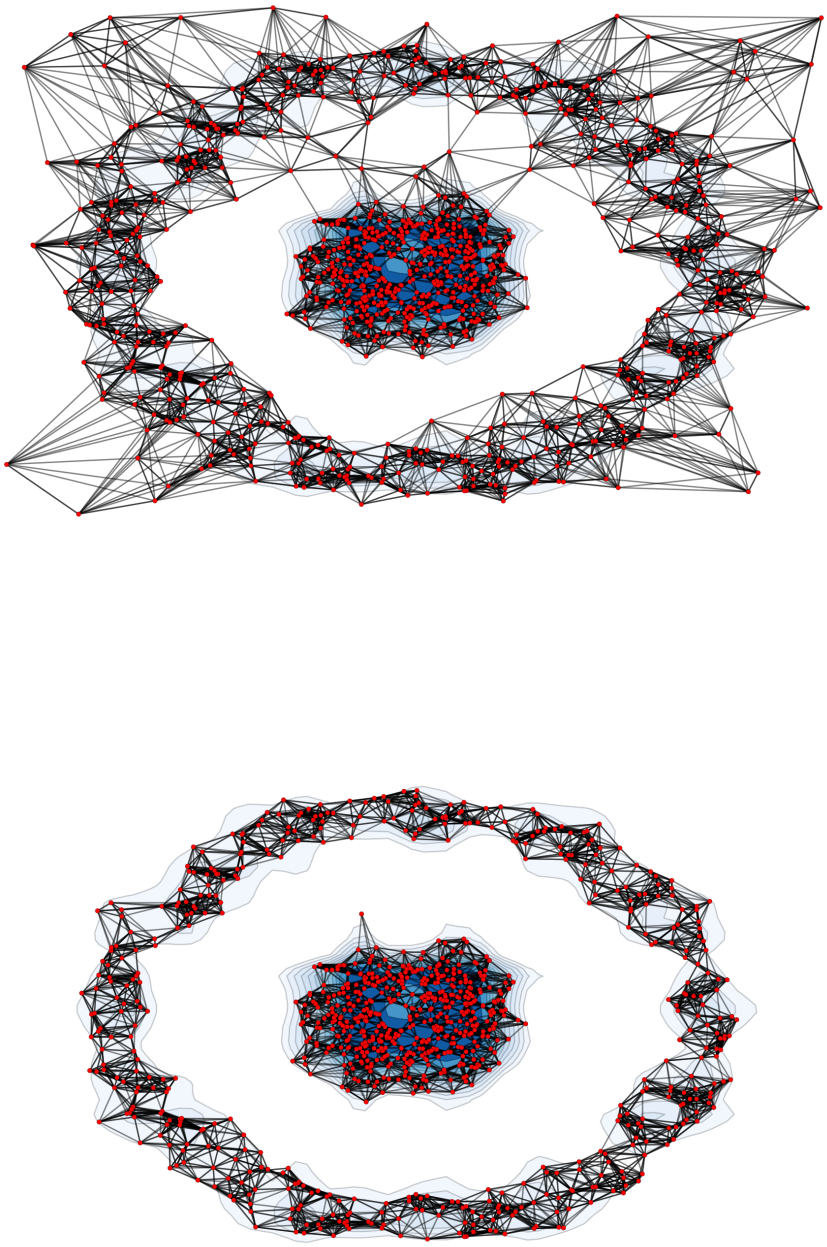


Abbildung 4.2.: Reines k-nearest-neighbors Clustering (oben) und mit Nutzung einer Dichteschätzung (unten) bei einem Datensatz mit unterschiedlichen Clusterformen

4.2. Erstellen des k-nearest-neighbors Graphen G

Der k-nearest-neighbors Algorithmus ist, für sich allein genommen, bereits eine Möglichkeit zum Aufspüren von Cluster in einem Datensatz [DH04]. Bei diesem Algorithmus wird der Datensatz in einen Graphen G umgewandelt. Hierbei werden die Datenpunkte zu den Knoten V des Graphen. Für jeden dieser Knoten suchen wir nun die k nächsten Nachbarn und legen Kanten zu ihnen an. Dieser Vorgang wird für jeden Datenpunkt wiederholt und wir erhalten am Ende einen gerichteten Graphen G mit der Kantenmenge E und der Knotenmenge V.

$$G = (V, E) \tag{4.1}$$

Für einen Datensatz D, der aus einer Menge an d-dimensionalen Vektoren $\vec{v_i} \in D$ besteht, ergibt sich folgendermaßen die Knotenmenge V.

$$V = \left\{ v_i | \vec{v_i} \in D \right\} \tag{4.2}$$

Die Kantenmenge E_{v_i}, die von einem einzigen Knoten v_i ausgeht, beinhaltet die Kanten $e = (v_i, v)$, wobei v einer der k nächsten Nachbarn von v_i ist. Die Distanz zwischen zwei Knoten wird mit der euklidischen Norm berechnet.

$$E_{v_i} = \left\{ e \mid (v_i, v) = e \wedge v \in V \wedge v_i \in V \wedge \left| \left\{ (v, v'') \mid |v - v''|_2 < |v - v_i|_2 \wedge v'' \in V \right\} \right| < k \right\} \tag{4.3}$$

Die gesamte Kantenmenge ergibt sich durch die Vereinigung aller einzelnen Kantenmengen.

$$E = \cup_{i=1}^{|V|} E_{v_i} \tag{4.4}$$

Der Graph selbst wird im Programm als Adjazenzliste abgelegt. Da die Datenpunkte eines Clusters nah beieinander liegen, sollten die Knoten der Datenpunkte in dem Cluster nur mit Knoten verbunden sein, die ebenfalls in dem gleichen Cluster liegen. Wenn k richtig gewählt wird, wird im Idealfall jedes Cluster durch eine Zusammenhangskomponente im Graphen beschrieben werden. Um die Cluster eines Datensatzes zu finden, muss man bei diesem Verfahren zuerst, wie oben beschrieben, den Graphen erstellen. Daraufhin durchsucht man diesen Graphen nach Zusammenhangskomponenten. Jede solche Zusammenhangskomponente stellt ein Cluster dar. Falls man k jedoch zu groß wählt, steigt die Wahrscheinlichkeit, dass die Cluster auch untereinander verbunden werden. Wenn man es zu klein wählt, steigt wiederum die Wahrscheinlichkeit, dass die Cluster aus mehr als nur einer Zusammenhangskomponente bestehen. In der Abbildung 4.4 wurde beispielsweise ein k-nearest-neighbors Graph erzeugt, alle sechs Cluster des Datensatzes sind im Graph jedoch auch untereinander verbunden und werden so als einziges Cluster erkannt. Auch einzelne Datenpunkte, die sich zwischen den einzelnen Cluster befinden, führen zu Problemen, da durch diese Datenpunkte im Graph G eine Verbindung zwischen den Clustern entstehen kann. Dies ist gerade bei Datensätzen mit Rauschen eine Gefahr.

Eine Möglichkeit um überflüssige Knoten und Kanten aus dem Graphen G zu entfernen, wäre also wünschenswert. Hierfür nutzen wir aus, dass Knoten und Kanten, welche nicht zu einem Cluster gehören, in der Regel in Gebieten geringer Dichte liegen [PPB12].

4.3. Erstellen der Dichteschätzung

Um das Clustering mit dem k-nearest-neighbor Algorithmus zu verbessern, brauchen wir eine Abschätzung der Dichtefunktion des Datensatzes. Diese erstellen wir mit der in Kapitel 3 vorgestellten Methode. Dazu wird Dichteschätzung mit einem dünnen Gitter erstellt und ist die Lösung der Variationsgleichung 3.1. Um den Koeffizientenvektor $\vec{\alpha}$ zur Erstellung dieser Funktion zu finden, müssen wir das lineare Gleichungssystem 3.11 lösen.

Um das Gleichungssystem 3.11 nach α aufzulösen, benötigen wir ein iteratives Verfahren, da die Größe der Matrix $\mathcal{A} + \lambda I$, N^2 ist und damit sehr groß werden kann. Bei einem Gitter mit 10^6 Gitterpunkten hätte man zum Beispiel direkt 10^{12} Einträge. Eine direkte Lösung der LGS ist demnach teuer. Als iteratives Verfahren setzen wir das Verfahren der konjugierten Gradienten ein [She94]. Alternativ kann man natürlich auch andere, iterative Lösungsverfahren für Gleichungssysteme nutzen. Wir beschränken uns jedoch in dieser Arbeit auf das CG-Verfahren.

Ein weiterer Aspekt den wir zu berücksichtigen haben, ist, dass man diese Matrix $\mathcal{A}$ aufgrund ihrer Größe nicht direkt abspeichern kann. Um das iterative Verfahren auszuführen, müssen wir also die Matrix-Vektor Multiplikation implementieren, ohne auf die Matrixelemente explizit zuzugreifen. Ein expliziter Zugriff würde erfordern, dass wir die Matrix abspeichern. Stattdessen verwenden wir die Einträge der Matrix bei der Multiplikation nur implizit, das heißt die Einträge müssen in jeder Multiplikation erneut berechnet werden.

Da wir die Einträge $\mathcal{A}_{ij}$ in jeder Matrix-Vektor Multiplikation erneut berechnen müssen, lohnt es sich, uns die Berechnung eines Eintrages näher anzuschauen. Jeder Eintrag $\mathcal{A}_{ij}$ ist das L2 Skalarprodukt zweier Basisfunktionen φ_i und φ_j. Wir müssen also jeweils die zwei Basisfunktionen multiplizieren und dann die Ergebnisfunktion über den gesamten Raum Ω integrieren.

$$\mathcal{A}_{ij} = \left(\varphi_i\left(x\right)\varphi_j\left(x\right)\right)_{L_2} = \int_{\Omega} \varphi_i\left(x\right)\varphi_j\left(x\right)dx \tag{4.5}$$

Wir betrachten für dieses Problem zunächst nur den eindimensionalen Fall und wählen wie gehabt die Hutfunktionen als Basisfunktionen für unser Gitter. Mit diesem eindimensionalen Szenario können wir später leicht das L_2 Skalarprodukt von mehrdimensionalen Basisfunktionen bestimmen.

Für das Integral des Produktes zweier eindimensionaler Basisfunktionen $\varphi_i(x)\,\varphi_j(x)$ müssen wir nun mehrere Fälle unterscheiden. l_i bezeichnet im Folgenden den Level der Basisfunktion φ_i und l_j bezeichnet den Level der Basisfunktion φ_j.

Zunächst untersuchen wir den Fall für den gilt $l_i < l_j$.

Zur Vereinfachung verschieben wir die Funktion φ_j so, dass der von 0 unterschiedliche Teil der Funktion sich auf dem Intervall $[0, 2^{-l_j+1}]$ befindet. Da sich die Trägermenge einer beliebigen Dreiecksfunktion $\varphi_z(x)$ nur auf dem Intervall $(2^{-l_z}(i_z-1), 2^{-l_z}(i_z+1))$ befinden und die Basisfunktion φ_i gleich verschoben wird, ändert sich nichts an dem Integral, das wir berechnen werden.

Bevor die Funktion verschoben wird, bestimmen wir die Funktionswerte der Funktion φ_i am Rand der Trägermenge von φ_j. Diese zwei Funktionswerte werden von nun an mit u_{right} und u_{left} bezeichnet und werden zur Bestimmung des Integrals von $\varphi_i(x)$ benötigt.

$$u_{left} = \varphi_i\left(2^{-l_j}\left(i_j - 1\right)\right) \tag{4.6}$$

$$u_{right} = \varphi_i\left(2^{-l_j}\left(i_j + 1\right)\right) \tag{4.7}$$

Von hier an gehen wir davon aus, dass die beiden Basisfunktionen, wie oben beschrieben, um $2^{-l_j}\left(i_j - 1\right)$ nach links verschoben sind.

Da der Level l_i der Basisfunktion $\varphi_i(x)$ kleiner ist als der Level l_j, verläuft die Funktion $\varphi_i(x)$ linear auf der Trägermenge von $\varphi_j(x)$. Wir können sie also für die folgende Rechnung mit der Geradengleichung

$$f(x) := u_{left} + \frac{(u_{right} - u_{left})}{2^{-l_j+1}}x \tag{4.8}$$

ausdrücken. Diese Gerade ist auf dem Intervall $[0, 2^{-l_j+1}]$ gleichwertig zu φ_i. Wir benötigen zudem noch später die Stammfunktion dieser Gerade

Wir ersetzen nun auch die andere Basisfunktion φ_j. Für die Ersetzung dieser Funktion brauchen wir zwei Geraden, da die Funktion φ_j auf dem Intervall $[0, 2^{-l_j+1}]$ stückweise definiert ist. Wir werden eine Gerade $b_1(x)$ für den Intervall $[0, 2^{-l_j}]$ und eine Gerade b_2 für den Intervall $2^{-l_j}, 2^{-l_j+1}$ benötigen. Damit diese zwei Geraden den Basisfunktion φ_j auf diesem Intervall ersetzen können, müssen sie wie folgt definiert sein:

$$b_1(x) := 2^{l_j} \cdot x \tag{4.9}$$

$$b_2(x) := 1 - \frac{\left(x - 2^{-l_j}\right)}{2^{-l_j}} = 2 - 2^{l_j} \cdot x \tag{4.10}$$

Mit den Hilfsfunktionen b_1, b_2 und f, kann man das L_2 Skalarprodukt für den Matrixeintrag $\mathcal{A}_{ij}$, nun folgendermaßen schreiben:

$$\mathcal{A}_{ij} = \int_\Omega \varphi_i(x)\,\varphi_j(x) = \int_0^{2^{-l_j}} f(x) \cdot b_1(x)dx + \int_{2^{-l_j}}^{2^{-l_j+1}} f(x) \cdot b_2(x)dx \tag{4.11}$$

Um den Matrixeintrag zu bestimmen noch die beiden Integrale aufgelöst werden.

$$\begin{aligned}
\int_0^{2^{-l_j}} f(x)\cdot b_1(x)dx &= \int_0^{2^{-l_j}} 2^{l_j}x\left(u_{left}+\frac{(u_{right}-u_{left})}{2^{-l_j+1}}x\right)dx \\
&= \int_0^{2^{-l_j}} \left(2^{l_j}\cdot x\cdot u_{left}\right)dx + \int_0^{2^{-l_j}} \left(\frac{(u_{right}-u_{left})}{2^{-l_j+1}}\right)\cdot 2^{l_j}\cdot x^2 dx \\
&= \left(\frac{2^{l_j}\cdot x^2\cdot u_{left}}{2}\right)\Bigg|_0^{2^{-l_j}} + \left(\frac{(u_{right}-u_{left})}{3\cdot 2^{-l_j+1}}\right)\cdot 2^{l_j}\cdot x^3\Bigg|_0^{2^{-l_j}} \\
&= 2^{-l_j}\left(\frac{u_{left}}{2}\right)+\left(\frac{(u_{right}-u_{left})}{6}\right)\cdot 2^{-l_j} \\
&= 2^{-l_j}\left(\frac{2u_{left}+u_{right}}{6}\right)
\end{aligned} \tag{4.12}$$

$$\begin{aligned}
\int_{2^{-l_j}}^{2^{-l_j+1}} f(x)\cdot b_2(x)dx &= \left(2-2^{l_j}\cdot x\right)\cdot\left(u_{left}+\frac{(u_{right}-u_{left})}{2^{-l_j+1}}x\right) \\
&= 2\int_{2^{-l_j}}^{2^{-l_j+1}} \left(u_{left}+\frac{(u_{right}-u_{left})}{2^{-l_j+1}}x\right) - \\
&\quad \int_{2^{-l_j}}^{2^{-l_j+1}} 2^{l_j}\cdot x\left(u_{left}+\frac{(u_{right}-u_{left})}{2^{-l_j+1}}x\right) \\
&= 2^{-l_j}\cdot\left(\frac{(3u_{left}+9u_{right})}{6}\right)-2^{-l_j}\cdot\left(\frac{(2u_{left}+7u_{right})}{6}\right) \\
&= 2^{-l_j}\cdot\left(\frac{u_{left}+2u_{right}}{6}\right)
\end{aligned} \tag{4.13}$$

Damit haben wir nun das Gesamtergebnis für den Fall $l_i < l_j$ mit

$$\begin{aligned}
\mathcal{A}_{ij} &= \int_0^{2^{-l_j}} f(x)\cdot b_1(x)dx + \int_{2^{-l_j}}^{2^{-l_j+1}} f(x)\cdot b_2(x)dx \\
&= 2^{-l_j}\left(\frac{2u_{left}+u_{right}}{6}\right)+2^{-l_j}\left(\frac{u_{left}+2u_{right}}{6}\right) \\
&= 2^{-l_j}\left(\frac{u_{left}+u_{right}}{2}\right)
\end{aligned} \tag{4.14}$$

Der Fall $l_i > l_j$ ist analog zu behandeln. Wir müssen lediglich i durch j vertauschen.

$$\mathcal{A}_{ij} = \frac{\left(u_{left}+u_{right}\right)}{2}2^{-l_i} \tag{4.15}$$

Der letzten Fall, den wir betrachten müssen, ist $l_j = l_i$. Für diesen Fall ist das L_2 Skalarprodukt für alle bis auf einen Fall 0. Nur falls $i = j$ gilt, ist das L_2 Skalarprodukt ungleich 0. Hier lässt sich das Integral durch Substitution lösen.

$$\begin{aligned}\mathcal{A}_{ij} = \int_\Omega \varphi_i(x)\,\varphi_j(x) &= \int_0^{2^{-l_i}} b_1(x)\cdot b_1(x)dx + \int_{2^{-l_i}}^{2^{-l_i+1}} b_2(x)\cdot b_2(x)dx \\ &= \int_0^{2^{-l_i}} 4^{l_i}\cdot x^2 dx + \int_{2^{-l_i}}^{2^{-l_i+1}} \left(2^{-l_i}\cdot x - 2\right)^2 dx \\ &= \frac{2^{-l_i}}{3} + \frac{2^{-l_i}}{3} \\ &= \frac{2}{3}\cdot 2^{-l_i}\end{aligned} \tag{4.16}$$

Mit diesen drei Fällen, können wir nun alle Einträge der Matrix $\mathcal{A}$ für den eindimensionalen Fall bestimmen. Diesen eindimensionalen Fall können wir nutzen, um das L2 Skalarprodukt für höhere Dimensionen zu berechnen. Für den mehrdimensionalen Fall gilt ergibt sich das L2 Skalarprodukt aus dem Produkt der einzelnen, eindimensionalen L2 Skalarprodukte [PPB10].

$$(\varphi_i(\vec{x}), \varphi_j\vec{x})_{L2} = \prod_{r=1}^{d} (\varphi_{i_r}(x_r), \varphi_{j_r}(x_r))_{L2} \tag{4.17}$$

Damit haben wir nun eine Formel um die Einträge der Matrix $\mathcal{A}$ exakt zu berechnen. Wir können somit eine Matrix-Vektor Multiplikation für das iterative Verfahren durchführen und so das lineare Gleichungssystems 3.11 lösen. Durch diese Lösung erhalten wir den Koeffizientenvektor $\vec{\alpha}$ und können mit diesem eine Linearkombination der Basisfunktionen des Gitters erstellen. Diese Linearkombination der Basisfunktionen ist die gesuchte Schätzung der Dichte. Im Folgenden bezeichnen wir diese Funktion als $\kappa(\vec{x}) : \Omega \to \mathbb{R}$. Eine solche Abschätzung der Dichte ist in den Abbildungen 4.4 und 4.5 in Blau zu sehen.

Für die Lösung des LGS müssen wir allerdings zuvor noch zwei Parameter angeben. Wir brauchen einen Regularisierungsparameter λ, wie in Kapitel 3 beschrieben. Über diesen Parameter wird die Glätte der Dichtefunktion gesteuert. Als zweiten Parameter benötigen wir noch einen Abbruchsfehler für das CG-Verfahren. Da wir keine exakte Lösung für unsere Dichteschätzung benötigen, können wir so Rechenleistung sparen, indem wir das Verfahren, je nach gewünschter Genauigkeit, früher abbrechen.

4.4. Entfernen von Knoten und Kanten

Da wir den Vektor α zur Berechnung der Dichteschätzung bestimmt haben, können wir diese Schätzung dazu nutzen, die Knoten und Kanten in Gebieten mit geringer Dichte zu entfernen, um den Graphen G' zu erhalten. Die Gebiete in denen die eigentlichen Cluster liegen, haben

aufgrund der Häufung der Datenpunkte, eine hohe Dichte, Gebiete zwischen Clustern haben hingegen nur eine geringe Dichte, da hier nur vereinzelte Datenpunkte auftauchen. Wenn man also die Knoten und Kanten des Graphen G in Gebieten mit geringer Dichte entfernt, kappt man effektiv die Verbindungen zwischen den unterschiedlichen Cluster im Graph G. In 4.5 ist zudem dargestellt, wie Knoten, bei deren Datenpunkt nur eine geringe Dichte vorhanden ist, entfernt wurden.

Man erhält so einen Graphen G' mit neuer Knoten- und Kantenmenge. Wenn alle Verbindungen zwischen den Clustern gekappt werden, zerfällt der Graph G' in eine Zusammenhangskomponente pro Cluster.

$$G' = \left(V', E'\right) \tag{4.18}$$

τ soll ein Schwellwert sein, mit dem bestimmt wird, ob ein Knoten oder eine Kante entfernt wird. Somit erhalten wir die Definitionen für die neue Knoten- und Kantenmenge.

$$V' = \left\{v | v \in V \wedge \kappa\left(v\right) > \tau\right\} \tag{4.19}$$

$$p(e) = v_1 + \frac{v_2 - v_1}{2} \quad \forall e \in E \tag{4.20}$$

$$E' = \left\{e | e \in E \wedge \kappa\left(p(e)\right) > \tau\right\} \tag{4.21}$$

Um dies im Programm zu realisieren, müssen wir alle Knoten durchgehen und für den entsprechenden Datenpunkt den Wert der Dichteschätzung bestimmen. Bei den Kanten wird die Dichte an der Mitte der Kanten bestimmt. Falls die Dichte an einem Knoten unter dem Schwellwert τ liegt, markieren wir die entsprechende Zeile in der Adjazenzliste mit -1. Falls die Dichte auf der Mitte einer Kante unter dem Schwellwert liegt, markieren wir den Eintrag mit -2. So können wir die entfernten Knoten und Kanten während der Suche nach den Zusammenhangskomponenten erkennen und entsprechend berücksichtigen. Sobald dies erledigt ist erhalten wir den Graphen G'.

Die Wahl des Schwellwerts, der bestimmt, ab welcher Dichte man beginnt Knoten und Kanten zu entfernen, ist von entscheidender Bedeutung. Wählt man ihn zu klein, werden nicht genügend Knoten und Kanten entfernt und wir erhalten beinahe dasselbe Ergebnis wie bei einem k-nearest-neighbors Algorithmus. Falls wir ihn zu groß wählen, werden viele Datenpunkte ignoriert, die eigentlich noch zu einem Cluster gehören würden, aber entfernt wurden, da die Dichte den Schwellwert nicht überschritten hat.

4.5. Suche nach Zusammenhangskomponenten

Den Graphen G' müssen wir nun nach Zusammenhangskomponenten durchsuchen. Jede Zusammenhangskomponente ist ein Cluster. Um nach den Zusammenhangskomponenten zu suchen, verwenden wir eine Tiefensuche. Dazu starten wir für jeden Knoten, der noch nicht besucht wurde, einmal die Tiefensuche, markieren den Knoten als besucht und gehen zu den Kindknoten weiter, falls diese noch nicht besucht wurden. Alle besuchten Kindknoten bekommen den gleichen Clusterindex wie der Elternknoten. Falls ein Kindknoten vorhanden ist, der bereits einem anderen Cluster zugeordnet ist, gehen wir durch die Rekursion zurück und setzen den Clusterindex aller Knoten auf diesen Clusterindex.

Wenn diese Prozedur abgeschlossen ist, haben wir eine Liste, in der jedem Datenpunkt des ursprünglichen Datensatzes D ein Index zugeordnet ist. Falls ein Knoten entfernt wurde, hat dieser Index den Wert -1, ansonsten jeweils den Index des Clusters, zu dem der Knoten gehört. Wenn wir in unserem Beispiel nun die Datenpunkte je nach Index ihres Clusters einfärben, erhalten wir die Abbildung 4.6.

Wir haben nun aus dem ursprünglichen Datensatz D, eine Menge an Clustern gewonnen, und können jedem Datenpunkt, der nicht entfernt wurde, eines dieser Cluster zuordnen.

Die beispielhafte Anwendung des Algorithmus auf einen kleinen Datensatz mit 600 Datenpunkten ist in den Abbildungen 4.3, 4.4, 4.5 und 4.6, zu sehen.

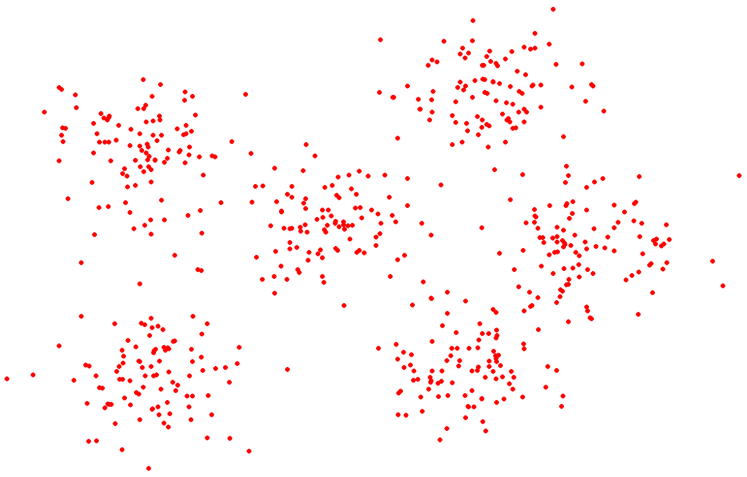


Abbildung 4.3.: Input: Beispieldatensatz mit 600 Datenpunkte

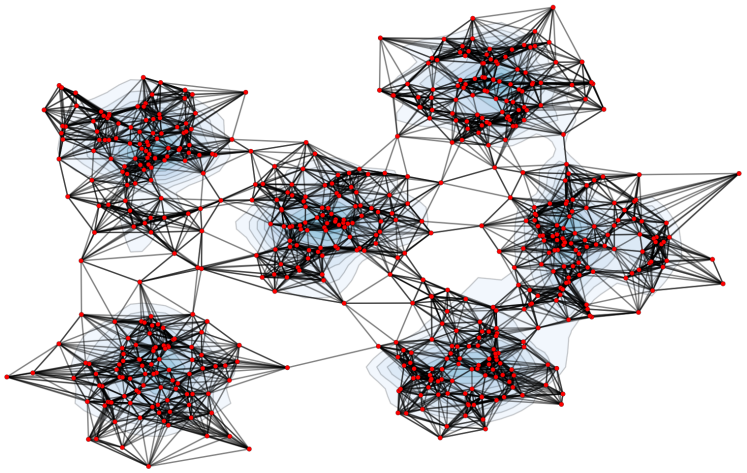


Abbildung 4.4.: Schritt 1 und 2: Dichtefunktion (blau) und Graph G erstellt

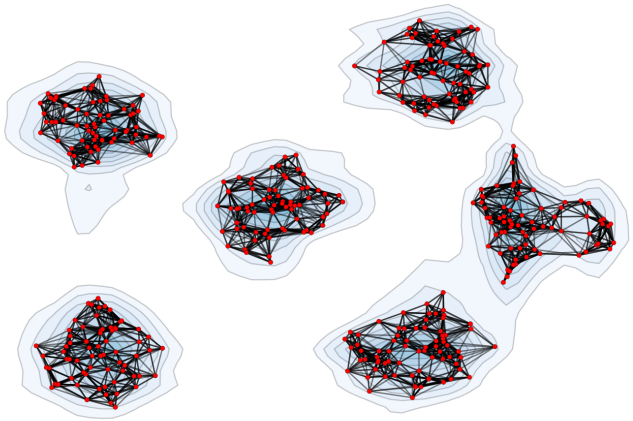


Abbildung 4.5.: Schritt 3: Durch das Entfernen von Knoten und Kanten mit geringer Dichte wurde Graph G' aus G erzeugt

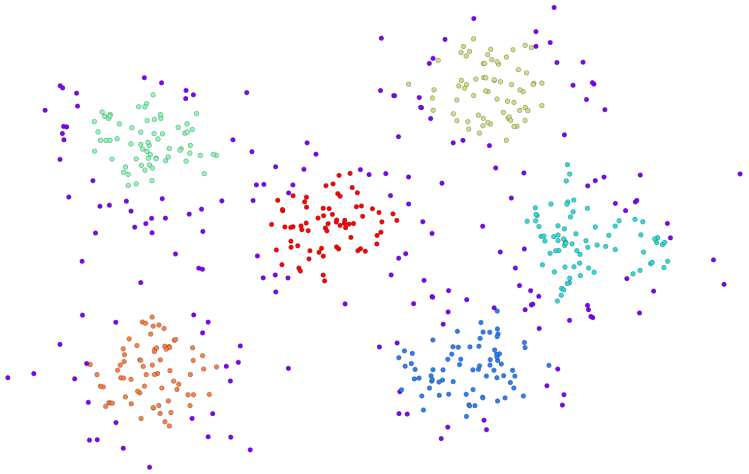


Abbildung 4.6.: Schritt 4: Jedem Datenpunkt wurde ein Cluster zugeordnet. Die Menge der blauen Punkte beinhaltet die entfernten Punkte

5. Paralleles Clustering auf einem Rechenknoten

In diesem Kapitel gehen wir darauf ein, wie der Clustering Algorithmus von [PPB12] mit OpenCL parallelisiert wurde. Hierzu erklären wir kurz allgemein was OpenCL ist, und gehen danach auf die Implementierung der einzelnen Arbeitsschritte ein.

5.1. OpenCL Grundlagen

Die Open Computing Language, oder kurz OpenCL, ist eine Schnittstelle für Parallelrechner. Mittels dieser Schnittstelle kann man Programme schreiben und anschließend auf beliebigen OpenCL Geräten ausführen. Mögliche OpenCL Geräte umfassen vor allem CPUs und Grafikkarten. Die Programme für diese Geräte schreibt man in einer auf C99 basierenden Sprache. Ein solches Programm wird anschließend auf einem OpenCL Gerät mehrfach ausgeführt und kann so ein Problem parallel bearbeiten. Dazu legt eine Hostanwendung die Daten für das Programm in OpenCL Puffern ab und führt danach das Programm aus. Die Ergebnisse können später von der Host Anwendung wieder aus Puffern gelesen werden.

Auch wenn die eigentliche Programmiersprache für diese Programme, die auch Kernels genannt werden, auf C99 basiert, gibt es mehrere Konzepte zu beachten.

Beim Start eines OpenCL Kernels, werden auf dem gewählten Gerät Workitems gestartet. Jedes dieser Workitems wird durch eine ID identifiziert und führt den kompletten Kernel aus. Auf dieses Weise findet die Parallelisierung des Programms statt. Die einzelnen Workitems werden wiederum in Workgroups unterteilt. Das Synchronisieren von Workitems funktioniert nur innerhalb Workgroups. Auf globaler Ebene hat man keine Möglichkeit zur Synchronisierung.

Die Daten, die ein Workitem verarbeitet, können auf mehreren Speicherebenen liegen. Zum einen dem globalen Speicher, in dem die Daten aus der Hostanwendung abgelegt werden. Auf diesen Speicher können alle Workitems zugreifen. Im lokalen Speicher liegen die Daten, die von Workitems der gleichen Workgroup geteilt werden. Im privaten Speicher liegen die Daten eines einzigen Workitems. Diese unterschiedliche Speicherebenen sind unterschiedlich schnell. Man sollte versuchen, die Zugriffe auf den globalen Speicher gering zu halten und

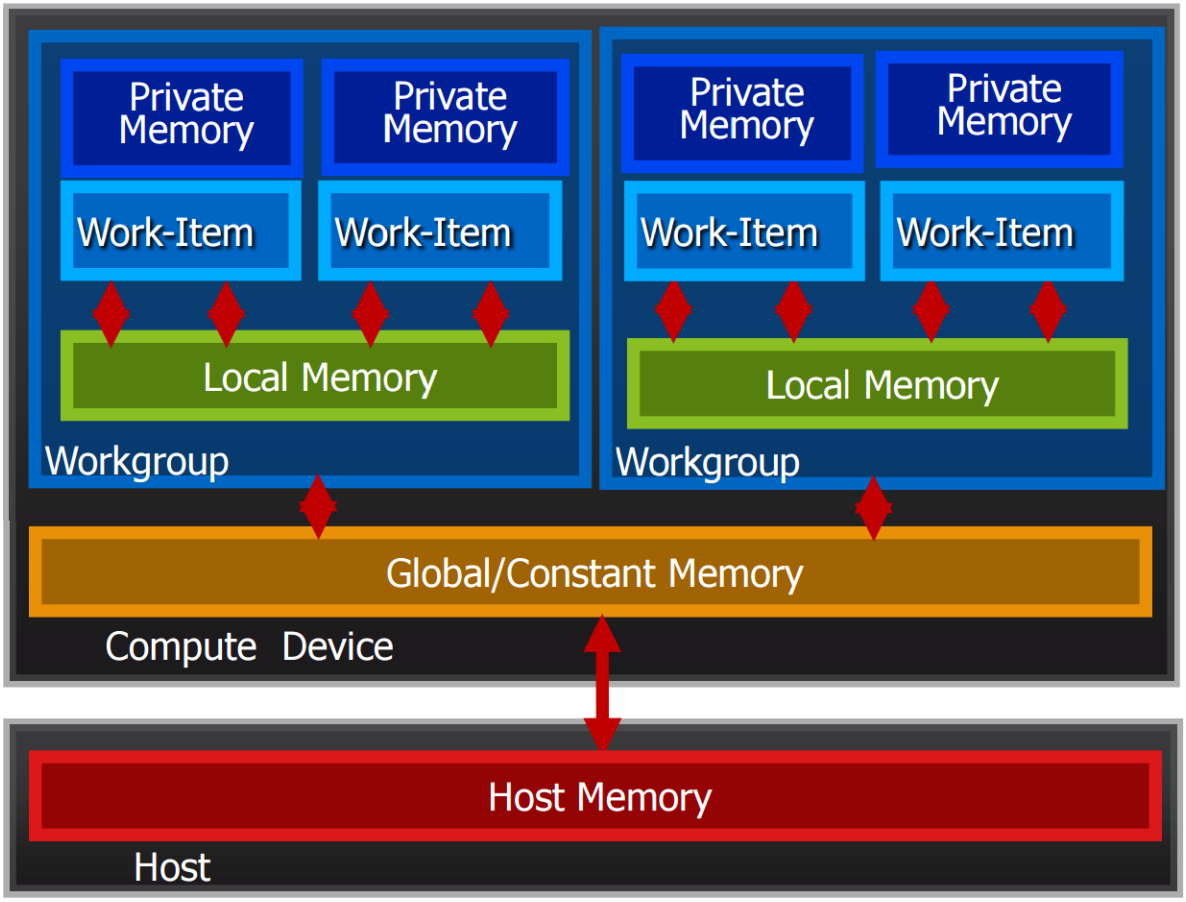


Abbildung 5.1.: Speichermodell von OpenCL (aus [TS12])

stattdessen lieber den lokalen Speicher, oder noch besser den privaten Speicher nutzen. Ein Schema dieser Speicherebenen ist in Abbildung 5.1 zu sehen.

Eine andere Besonderheit sind Verzweigungen. Da Workitems in OpenCL ihre Instruktionen immer parallel abarbeiten, sollte man Verzweigungen im Kontrollfluss eines Kernels vermeiden. Wenn ein Teil der Workitems in eine if-Verzweigung springen, die anderen aber nicht, müssen die anderen Workitems warten, bis die If-Verzweigung fertig berechnet wurde.

Falls bereits bekannt ist, auf welcher Zielplattform ein OpenCL ausgeführt werden soll, sollte man diesen auch direkt an diese Architektur anpassen. Zwar wäre es möglich, einen OpenCL Kernel auf jedem beliebigen OpenCL Device auszuführen. Allerdings können Unterschiede zwischen Speicher der Geräte und unterschiedlichen Compilern für die einzelnen Plattformen dazu führen, dass die vorhandene Leistung einzelner Geräte unterschiedlich gut genutzt wird.

5.2. Implementierung des Clustering Algorithmus mit OpenCL

Um den in Kapitel 4 vorgestellten Clustering Algorithmus mittels OpenCL zu realisieren, benötigen wir mehrere solcher OpenCL Kernel. Wir benötigen ein Kernel, der die rechte Seite der Gleichung 3.11 berechnet. Zur iterativen Lösung derselben Gleichung benötigen wir außerdem einen Kernel, der die angesprochene Matrix-Vektor Multiplikation durchführt. Zusätzlich werden noch zwei weitere OpenCL Kernel benötigt. Einen zum Erstellen des k-nearest-neighbor Graphen und einen zum späteren Entfernen der Knoten und Kanten anhand der Dichteschätzung.

Diese vier OpenCL Kernel implementieren dann die ersten drei der vier Arbeitsschritte des Clustering Algorithmus. Die Suche nach Zusammenhangskomponenten im Graph G' wird nicht als OpenCL Kernel implementiert, da dieser Vorgang im Vergleich zu den anderen drei Arbeitsschritten kaum Rechenleistung benötigt.

Zu Beginn dieser Arbeit waren bereits die zwei benötigten Kernel zum Aufstellen der Dichteschätzung vorhanden. Die Kernels zum Erstellen des Graphen und Entfernen der Knoten und Kanten mussten neu geschrieben werden.

5.2.1. Kernels zum Aufstellen der Dichteschätzung

Zunächst gehen wir einmal auf die zwei Kernel ein, die die Dichteschätzung erstellen sollen. Für eine Abschätzung der Dichte müssen wir die Gleichung 3.11 lösen können. Wie in Kapitel 4 erwähnt nutzen wir dazu das CG-Verfahren. Um dieses Verfahren nutzen zu können, benötigen wir die Möglichkeit eine Matrix-Vektor Multiplikation mit der Matrix $\mathcal{A} + \lambda I$ durchzuführen. Dies wird durch einen der beiden folgenden OpenCL Kernels implementiert. Der andere Kernel soll die rechte Seite der Gleichung 3.11 bestimmen. Mit diesen beiden Kernels können wir das LGS mit dem CG-Verfahren auflösen und so unsere Dichteschätzung bestimmen.

Berechnen der rechten Gleichungsseite

Dieser Kernel soll den Vektor auf der rechten Seite der Gleichung 3.11 berechnen. Da ein Eintrag i dieses Vektors besteht jeweils aus der Aufsummierung aller Funktionswerte $\varphi_i(\vec{x}) : D \to \mathbb{R}$. Jedes Workitem des OpenCL Kernels soll einen dieser Einträge berechnen. Dafür muss es Zugriff auf alle Datenpunkte des Datensatzes D haben und in der Lage sein, die Basisfunktion eines Gitterpunktes auszuwerten. Dazu legen wir das Gitter und den Datensatz in Puffern des globalen Speicher von dem Gerät ab.

Mit diesen Werten kann jedes Workitem nun einfach in einer Schleife die Funktionswerte der Basisfunktion für die Elemente des Datensatzes aufaddieren. Das Ergebnis dieser Summation kann dann in einen Eintrag des Ergebnisvektors im globalen Speicher geschrieben, und später von der Hostanwendung ausgelesen werden.

OpenCL Kernel zur Matrix-Vektor Multiplikation

Dieser Kernel soll eine einzige Matrix-Vektor Multiplikation der Form $(\mathcal{A} + \lambda I)\,\vec{\alpha}$ ausführen. Wie wir in Kapitel 4 bereits erläutert haben, ist die Matrix für die Multiplikation zu groß um sinnvoll abgespeichert zu werden. Daher müssen wir die deren Einträge bei jeder Ausführung dieses Kernels neu berechnen. Zur Berechnung dieser Einträge haben wir zuvor in verschiedene Fälle unterschieden. Je nach Level der Basisfunktionen nutzen wir also eine der Formeln 4.14, 4.15 oder 4.16 für jeden der d eindimensionalen Fälle. Diese eindimensionalen Fälle kann man dann mit der Formel 4.17 kombinieren und erhält so einen einzelnen Eintrag der Matrix.

Jedes Workitem soll nun einen Eintrag des Ergebnisvektors der Multiplikation berechnen. Dazu benötigt es Zugriff auf alle Gitterpunkte, zur Berechnung der Matrix Einträge und Zugriff auf den Vektor $\vec{\alpha}$. Mit diesen Werten kann das Workitem nun die Einträge der Matrix berechnen und diese mit den entsprechenden Einträgen α_i multiplizieren. Es ist von dieser Ausgangslage also möglich, wie üblich einen Eintrag des Matrix-Vektor Produkts zu berechnen und diesen in einen Puffer zu schreiben, der von der Hostanwendung aus lesbar ist.

K-nearest neighbors auf OpenCL

Mit diesem Kernel soll die Adjazenzliste des k-nearest-Neighbors Algorithmus erstellt werden. Jeder Zeile dieser Liste ist einem Datenpunkt $\vec{x} \in D$ zugeordnet und beinhaltet k Einträge. Diese sind Einträge sind die Zeilenindices der k Nachbarn. Es macht also Sinn, pro Workitem jeweils eine solche Zeile zu berechnen. Um den k-nearest neighbors Graph zu erstellen, benötigt ein Workitem zumindest alle Datenpunkte in seiner unmittelbaren Nähe. Man kann also entweder den Datensatz zuvor partitionieren, so dass jedes Workitem später nur einen Teil der Datenpunkte nach den k Nachbarn durchsuchen muss, oder man übergibt direkt den kompletten Datensatz an den Kernel. Jedes Workitem durchsucht seine Datenpunkte dann nach den k nächsten Nachbarn und speichert diese an seine Stelle in der Adjazenzliste.

Entfernen von Knoten und Kanten in OpenCL

Um die Knoten und Kanten mit geringer Dichte aus dem Graphen zu entfernen muss jedes Workitem dieses Kernel dazu in der Lage sein, die Dichte an den jeweiligen Positionen

in Raum Ω auszuwerten. Zuerst wird die Dichte des Knotens, zu dem die Zeile gehört, berechnet. Wenn dessen Dichte unter dem angegebenen Schwellwert liegt, wird er entfernt. Dies geschieht, indem man alle Einträge einfach auf -1 setzt. Falls dies jedoch nicht der Fall ist, überprüft man die Dichte seiner einzelnen Verbindungen. Da man die Positionen der Knoten vorliegen hat, kann man auch einfach den Mittelpunkt dieser der Verbindung berechnen und die Dichte an dieser Stelle auswerten. Falls diese unter dem Schwellwert liegt, wird der Eintrag dieser Verbindung durch -2 ersetzt.

Um danach mit der Adjazenzliste nach Clustern zu suchen, kann man einfach die zuvor angesprochene Tiefensuche verwenden und die negativen Einträge entsprechend berücksichtigen. Dies findet nicht in einem eigenen OpenCL Kernel statt, sondern in der Hostanwendung. Da diese Tiefensuche allerdings lediglich $O(n)$ Rechenzeit in Anspruch nimmt, ist dies kaum bemerkbar. Der Hauptteil der Rechenzeit fließt noch immer in die das Erstellen der Dichteschätzung und des Graphen.

6. Verteiltes Clustering mit MPI

Nun soll der Clustering Algorithmus auf mehrere Rechner verteilt werden. Um dies durchzuführen benötigen wir ein Kommunikationsprotokoll, mit dem diese Rechner in der Lage sind Daten auszutauschen. So können etwa Teilergebnisse oder Zustandsänderungen zwischen den einzelnen Rechnern ausgetauscht werden. Wir gehen in diesem Kapitel zunächst kurz auf das Kommunikationsprotokoll ein, und danach darauf, wie die einzelnen Rechenaufgaben verteilt werden.

6.1. MPI Grundlagen

Im Gegensatz zur vorherigen Parallelisierung teilen sich die Threads nun keinen globalen Speicher mehr, sondern jeder Rechner verfügt über seinen eigenen Speicher, so wie in 6.1 dargestellt. Jeder Rechner benötigt also alle Daten, die zur Lösung seines Teilproblems nötig sind, in seinem lokalen Speicher. Um dies zu gewährleisten, müssen die Rechner in der Lage sein, Daten untereinander auszutauschen. Sie müssen also über eine Art zu kommunizieren verfügen.

Das Message Passing Interface, oder auch MPI, ist eine API, mit der ein solcher Nachrichtenaustausch durchgeführt werden kann. Eine MPI Anwendung besteht normalerweise, wie oben beschrieben, aus mehreren Prozessen, die untereinander kommunizieren. Ein solcher Prozess kann jeweils seinen eigenen Rechner haben. Es ist aber auch möglich, dass sich mehrere Prozesse einen Rechner teilen. Die MPI API selbst ermöglicht, es einfach Nachrichten zwischen Knoten zu schicken, um so die Rechenlast des Problems effizient aufzuteilen. Die Art, wie der Algorithmus verteilt wird, ist nun die eigentliche Herausforderung. Zum einen sollten alle Rechner möglichst durchgehend ausgelastet sein, zum anderen sollte man den Kommunikationsaufwand gering halten, da dieser ansonsten zu einem Flaschenhals werden kann.

6.2. Verteilen des Clustering Algorithmus

Wie bereits bei der Implementierung des Algorithmus mit OpenCL, müssen wir auch hier wieder mehrere Arbeitsschritte implementieren. Es muss wieder ein Graph G und eine

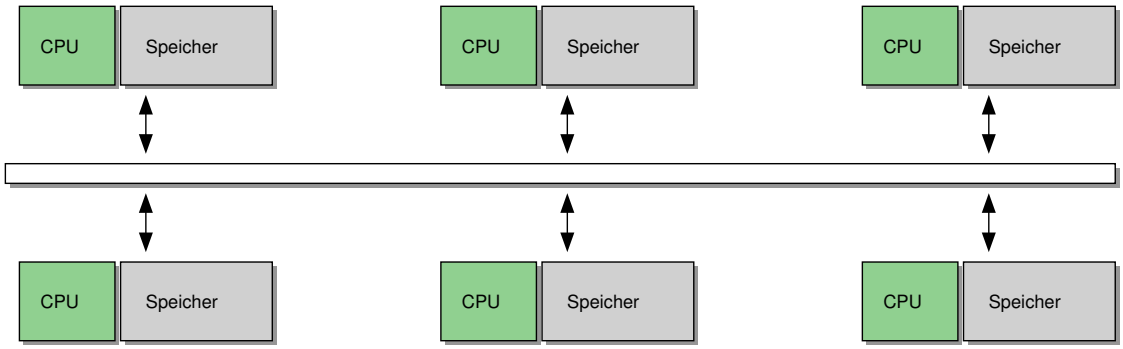


Abbildung 6.1.: Beispielhafte MPI Architektur

Dichteschätzung erstellt werden. Der Graph und die Dichtefunktion werden anschließend benötigt, um die Knoten und Kanten in Gebieten geringer Dichte zu entfernen, so dass man den Graphen G' erzeugen kann. Dieser wird wiederum für die Suche nach den Zusammenhangskomponenten benötigt. Zur Aufteilung des Clustering Algorithmus in mehrere Prozesse benötigen wir mindestens eine Stelle, an der alle Ergebnisse wieder zusammenlaufen, da wir die Ergebnisse eines Arbeitsschritts benötigen, bevor wir den nächsten starten können.

Für diese Aufgabe haben wir einen Masterknoten eingeführt. Der Masterknoten ist ein Prozess, der die Aufgabe hat, die jeweiligen Arbeitsschritte in Teilprobleme zu zerlegen, diese an andere Knoten zur Bearbeitung zu schicken, und am Ende deren Teilergebnisse zu einem endgültigen Resultat zusammenzusetzen. Der Masterknoten selbst bearbeitet keines der Teilprobleme, seine Aufgabe ist lediglich die Verwaltung der Rechenjobs für die anderen Knoten und die Verwaltung der Ergebnisse.

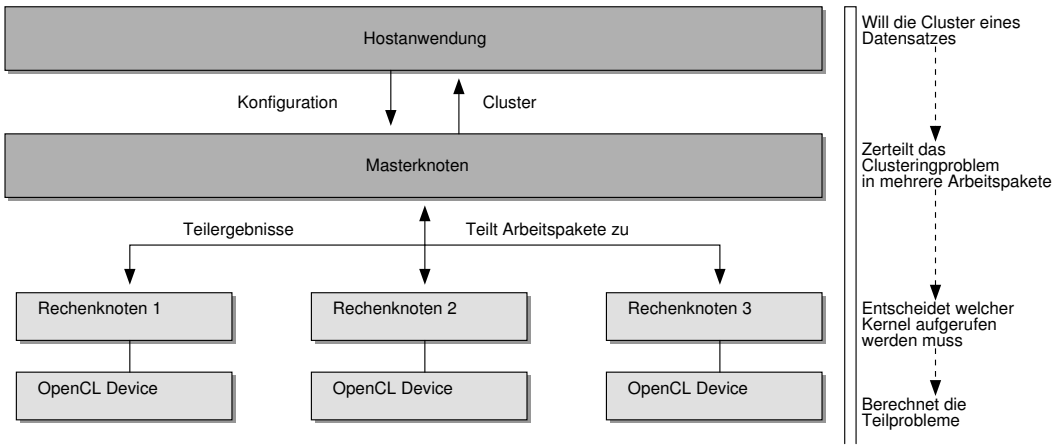
Die eigentliche Rechenarbeit übernehmen die anderen Prozesse, von nun an Rechenknoten genannt. Diese warten auf Nachrichten von dem Masterknoten und bearbeiten die Rechenjobs, die ihnen vom Masterknoten geschickt werden. Diese Rechenknoten müssen in der Lage sein zwischen vier Arten von Rechenjobs zu unterscheiden.

- Berechnung einiger Einträge des Vektors auf der rechten Seite der Gleichung 3.11
- Berechnung einiger Einträge des Matrix-Vektor Produkts
- Erstellen der einiger Zeilen der Adjazenzliste des k-nearest-neighbors Graphen G
- Überprüfung eines Teils der Adjazenzliste auf Knoten und Kanten die entfernt werden sollen

Im Kapitel 5 haben wir bereits OpenCL Kernel für diese Probleme entworfen. Es ist nun naheliegend, dass wir diese Kernel so anpassen, dass sie nun jeweils nur einen Teil der

Aufgabe berechnen, die sie zuvor vollständig gelöst haben. So können wir die Teilprobleme, die wir an die Rechenknoten schicken, per OpenCL bearbeiten.

Die Rechenknoten selbst sind voneinander komplett unabhängig. Das Schema unseres MPI Programms sieht also ungefähr so aus wie in Abbildung 6.2 dargestellt. Aus Sicht der Programmlogik ist es egal, ob der Masterknoten und die unterschiedlichen Rechenknoten, sich auf einem Rechner befinden, oder ob sich jeder Knoten auf einem eigenen Rechner befindet. Jeder Rechenknoten sollte allerdings Zugriff auf exakt ein OpenCL fähiges Gerät haben, um seine Rechenjobs per OpenCL Kernel bearbeiten zu können. Der Masterknoten benötigt außerdem Zugriff auf alle Daten des Problems, da er die Aufgabe hat diese aufzuteilen. Diese Daten umfassen den Datensatz auf den das Clustering angewendet werden soll, das Rechengitter zur Erstellung der Dichtefunktion und die Parameter λ, k, τ und den Abbruchsfehler für das CG-Verfahren. Kurz gesagt: Sie umfassen die gesamte Konfiguration. Bei unserer Implementierung wird diese Konfiguration über mehrere Dateien eingelesen. Diese Dateien können entweder vom Nutzer manuell angelegt werden, oder von einer Hostanwendung stammen, die die Konfiguration in die Dateien schreibt und anschließend das MPI-Programm startet. Auf diese Weise können wir das MPI-Programm sowohl von einer GUI-Hostanwendung nutzen, als auch Skripte schreiben, die das MPI-Programm mehrmals hintereinander mit unterschiedlichen Konfigurationen ausführen. Dies ist besonders für die Durchführung der Tests im nächsten Kapitel nützlich.



6.3. Implementierung der einzelnen Arbeitsschritte

Als nächstes gehen wir kurz auf die Implementierung der Rechenknoten ein und wie diese die einzelnen Teilprobleme, die ihnen geschickt werden, lösen können.

6.3.1. Verteiltes Aufstellen der Dichteschätzung

Zum Aufstellen der Dichteschätzung ist zunächst die Implementierung der Matrix-Vektor Multiplikation nötig. Wenn wir als Basis der Aufteilung den OpenCL Kernel für die Matrix-Vektor Multiplikation heranziehen, können wir das Problem in Arbeitsblöcke unterteilen, die jeweils mehrere Zeilen umfassen. Jeder Rechenknoten könnte dann einen solchen Arbeitsblock berechnen, so wie in 6.2 grob dargestellt. Man kann den OpenCL Kernel für die Multiplikation nehmen und für diesen einen Startindex und eine Größe festlegen und so einen solchen Arbeitsblock zu berechnen. Das Ergebnis hiervon sind die einzelnen Einträge des Ergebnisvektors der Zeilen des Arbeitsblocks. Diese können an den Masterknoten zurückgegeben werden und von diesem zum Gesamtergebnis hinzugefügt werden.

Ein Vorteil dieser Methode ist, dass das Zusammensetzen der Teillösungen trivial ist, da der Masterknoten einfach nur die empfangenen Teillösungen hintereinander setzen muss, um den Ergebnisvektor der Multiplikation zu konstruieren. Zudem ist die Versendung eines Arbeitspakets einfach durchzuführen. Am Anfang einer Multiplikation empfängt jeder Rechenknoten von dem Masterknoten einen neuen Vektor $\vec{\alpha}$, danach kann der Masterknoten einfach Multiplikationsaufträge verteilen, indem er Startindex und Auftragsgröße an die Rechenknoten sendet. Diese haben bereits alle nötigen Daten vorhanden und können sofort den Auftrag beginnen. Problematisch ist allerdings, dass jeder Knoten das komplette Gitter und den kompletten Vektor $\vec{\alpha}$ für die Berechnung eines Arbeitspaketes benötigt. Das bedeutet am Anfang des Programms müssen die nötigen Daten für das Gitter herum geschickt werden, und am Anfang jeder Multiplikation nochmals der Koeffizientenvektor $\vec{\alpha}$.

Zur Durchführung des CG-Verfahrens wird auch wieder der Vektor der rechten Gleichungsseite von 3.11 benötigt. Auch hier ist die Berechnung der einzelnen Ergebniseinträge von einander unabhängig. Es ist also wieder, wie bei der Multiplikation zuvor, möglich die Berechnung in Arbeitsblöcke zu unterteilen, wobei auch hier jeder Arbeitsblock aus einer Abfolge von Zeilen besteht. Sobald ein Rechenknoten die Gitterdaten und den Datensatz zur Verfügung hat, kann er mit seinem OpenCL Kernel einen beliebigen Abschnitt des Ergebnisvektors ausrechnen. Die Versendung der eigentlichen Rechenaufträge besteht somit auch hier nur aus einem Startindex und einer Auftragsgröße.

6.3.2. Verteiltes Erstellen des Graphen

Auch hier wird aus dem Datensatz eine Adjazenzliste gebildet. Man kann wieder das Erstellen des Graphen in Arbeitspakete aufteilen. Jedes Arbeitspaket besteht nun aus einer Menge von Datenpunkten, für die die k nächsten Nachbarn gesucht werden und in die Adjazenzliste eingetragen werden sollen. Auch für dieses Teilproblem kann man den vorhandenen OpenCL Kernel verändern, um nur die Adjazenzliste für eine bestimmte Menge von Datenpunkten zu erzeugen anstatt für den gesamten Datensatz.

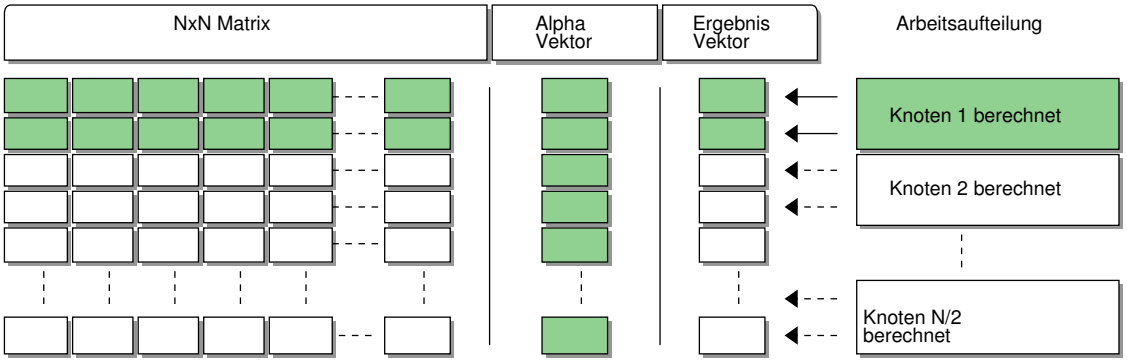


Abbildung 6.2.: Aufteilung der Multiplikation auf mehrere Rechenknoten

6.3.3. Verteiltes Entfernen von Knoten und Kanten

Nach dem verteilten Erstellen des Nachbarschaftsgraphen verfügt jeder Knoten bereits über einen Teil der Adjazenzliste, sowie über Informationen über die Datenpunkte in diesem Gebiet. Wenn man dem Rechenknoten nun noch die benötigten Informationen schickt um die Dichte zu berechnen, kann der vorhandene Teilgraph auch direkt beschnitten werden.

Auf diese Weise kann man die Kommunikation mit dem Masterknoten minimieren, da man die Adjazenzliste des Teilgraphen erst dann schickt, wenn bei dieser schon die Knoten und Kanten geringer Dichte entfernt wurden.

6.4. Load balancing

Bisher haben wir nur darüber gesprochen, wie man die vorhandenen Arbeitsschritte in Arbeitspakete für die Rechenknoten aufteilen kann. Allerdings ist die Größe dieser Arbeitspakete ebenfalls interessant. Falls man einen Arbeitsschritt einfach in gleich große Arbeitspakete für jeden Rechenknoten aufteilt, müssen die schnellen Rechenknoten am Ende auf die langsamen warten, da zum Start des nächsten Arbeitsschritts alle Teilergebnisse benötigt werden.

Eine mögliche Lösung dafür ist, die Größe der Arbeitspakete an die Leistung der jeweiligen Rechenknoten anzupassen. Schnelle Rechenknoten bekommen so größere Arbeitspakete und sollten so gleich schnell terminieren. Um die Größe der Arbeitspakete festzulegen, kann man beispielsweise eine lineare Interpolation verwenden.

6.5. Paketsystem

Eine weitere Möglichkeit die Last sinnvoll aufzuteilen, ist es, einen Arbeitsschritt nicht nur in ein Arbeitspaket pro Rechenknoten aufteilen. Stattdessen könnte man die Last in viele kleinere Schritte aufteilen und in einer Warteschlange speichern. Jedes Mal, wenn ein Rechenknoten fertig wird, kann ihm der Masterknoten so direkt ein neues Arbeitspaket zuweisen. Auf diese Weise sind die Rechenknoten ständig beschäftigt, bis dem Masterknoten zu Arbeitspakete ausgehen. Ein weiterer Vorteil ist, dass der Masterknoten ständig neue Teilergebnisse bekommt, die er sofort zusammensetzen kann und er so nicht auf alle Teilergebnisse warten muss.

Nachteilig ist jedoch, dass sich der Kommunikationsaufwand erhöht. Zum einen müssen zwischen den MPI Knoten mehr Nachrichten ausgetauscht werden. Zusätzlich hat jeder Rechenknoten noch zusätzlichen Aufwand, weil er die erhaltenen Daten auf sein OpenCL Gerät schreiben muss. Dies ist alles Rechenzeit, die nur zur Kommunikation genutzt wird. Es ist allerdings möglich, diesen Kommunikationsaufwand über die Größe der Arbeitspakete zu steuern. Je nach genutztem System können unterschiedliche Größen für die Arbeitspakete Sinn machen. Falls man beispielsweise ein System vorliegen hat, bei dem alle Rechenknoten über die gleiche Leistung verfügen, kann man einfach für jeden Rechenknoten exakt ein Paket bereithalten. So sollten alle Knoten gleichzeitig terminieren. Bei einem System, bei dem die Rechenknoten über unterschiedliche Leistung verfügen, wäre dies hingegen keine gute Idee. Hier würden am Ende alle Rechenknoten auf den langsamsten Knoten warten und so Rechenzeit verschwenden.

In der Abbildung 6.3 sehen wir, wie sich die Laufzeit des Programms bei unterschiedlichen Paketgrößen entwickelt. Hier wurde ein System genutzt, welches über zwei OpenCL Geräte verfügt, eine i7-4770k CPU und einer GeForce GTX 760 GPU. Diese zwei Geräte unterscheiden sich im Bezug auf die Rechenleistung stark. Die CPU kann 200 GFLOP/s mit doppelter Genauigkeit berechnen, nur GPU nur 94 GFLOP/s. Falls hier die Paketgröße zu groß wird, kann es sein, dass ein Rechenknoten auf den anderen warten muss, sobald dem Masterknoten die übrigen Arbeitspakete ausgehen. Dadurch wird potentielle Rechenzeit verschwendet. Bei einem System, bei dem dessen Geräte alle die gleiche Leistung haben, sollte hingegen die Performance bei erhöhter Paketgröße steigen, solange alle Rechenknoten die gleiche Anzahl an Pakete empfangen. Der Kommunikationsaufwand zwischen den Rechenknoten steigt bei kleiner werdender Paketgröße zwar nicht maßgeblich an, aber jeder Knoten muss öfters auf sein jeweiliges OpenCL Gerät zugreifen. Dadurch sinkt dessen reine Rechenzeit, da mehr Zeit mit IO Operationen verbraucht wird.

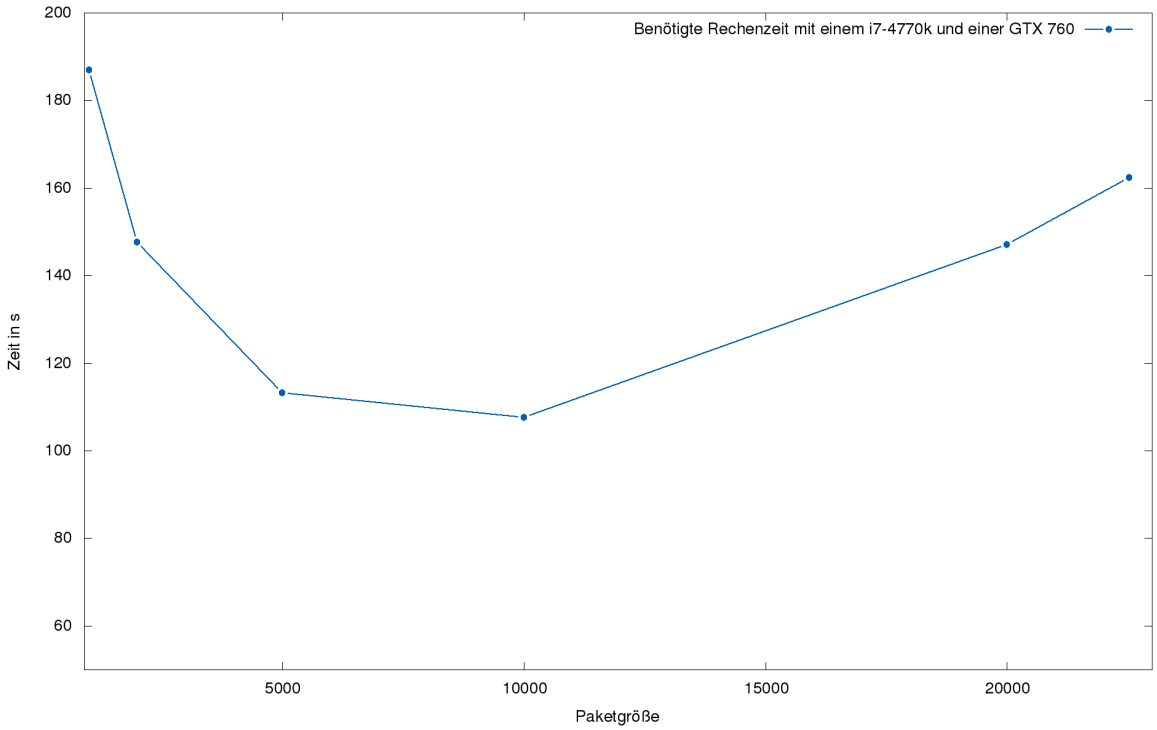


Abbildung 6.3.: Entwicklung der Laufzeit bei verschiedenen Paketgrößen

7. Tests

In diesem Kapitel testen wir das mit MPI verteilte Programm nun auf mehreren Plattformen. Dazu benötigen wir zunächst passende Datensätze zum Clustering und verschiedene Plattformen, um das Programm zu testen. Daher werden diese zuerst beschrieben. Anschließend führen wir auf jeder Plattform verschiedene Tests durch und analysieren deren Ergebnisse.

7.1. Generierung der Testdatensätze

Für die Durchführung der Testläufe auf den verschiedenen Plattformen ist es wünschenswert, beliebig große Datensätze erzeugen zu können. Mit diesen können wir testen, wie lange der Algorithmus zum Bearbeiten unterschiedlich großer Datensätze benötigt. Um das Clustering auf einer Plattform mit größerer Rechenleistung zu testen, wollen wir schließlich auch größere Datensätze verwenden.

Diese selbst erzeugten Datensätze haben allerdings vor allem einen anderen Vorteil. Wir haben ein Soll-Ergebnis, da wir wissen wie viele Cluster in dem Datensatz untergebracht wurden und welche Größe diese Cluster haben. Wir können also für jeden Datenpunkt überprüfen, ob er von dem Algorithmus dem richtigen Cluster zugeordnet wurde. Somit können wir nicht nur die Laufzeit des Algorithmus unter verschiedenen Bedingungen testen, sondern auch die Erkennungsrate überprüfen.

Um einen solchen Datensatz mit d Dimensionen zu erzeugen, benötigen wir zunächst die Positionen der Cluster in dem Raum $\Omega = [0, 1]^d$. Diese werden nacheinander bestimmt. Bei jedem Cluster wird überprüft, ob ein gewisser Abstand zu den Zentren, den bereits bekannten anderen Clusterzentren, eingehalten wird. Wenn dies nicht der Fall ist, wird eine neue, zufällige Position bestimmt. Der Abstand, der eingehalten werden soll, wird in Abhängigkeit der Standardabweichung, der später verwendeten Normalverteilung, bestimmt. Beispielsweise können wir sagen, dass die Clusterzentren 14 Standardabweichungen voneinander entfernt sein sollen, um auch bei großen Datensätze klar abgrenzte Cluster zu haben. Falls der Algorithmus nach mehreren Versuchen keine Position findet, die das Abstandskriterium einhält, wird die Standardabweichungen σ verkleinert.

Sobald die Zentren der Cluster bestimmt sind, werden deren Datenpunkte mit einer Gauss-Verteilung um diese Zentren herum platziert. Zuletzt werden für das Rauschen mit einer Gleichverteilung Datenpunkte in Ω verteilt.

In Abbildung 7.1 sieht man, wie beim zweiten Datensatz die Standardabweichung im Vergleich zum ersten Datensatz automatisch reduziert wurde, um alle Cluster im Raum Ω unterzubringen. Der 3. Datensatz ist ein Beispiel für einen der zweidimensionalen Datensätzen, die mit scikit-learn in Python erstellt wurden. Im Gegensatz zur obigen Methode, lassen sich diese Datensätze mit den Kreisen jedoch nicht für höherdimensionale Räume erstellen.

Die Zuordnung der Datenpunkte zu den Clustern, in Abbildung 7.1 links noch farbig zu sehen, wird gesondert in einer Datei abgespeichert und später zu Analyse verwendet. In derselben Abbildung sieht man rechts jeweils den resultierenden Datensatz.

7.2. Testplattformen

Es wurden drei unterschiedliche Plattformen für die Tests genutzt.

Vgpu1 Plattform

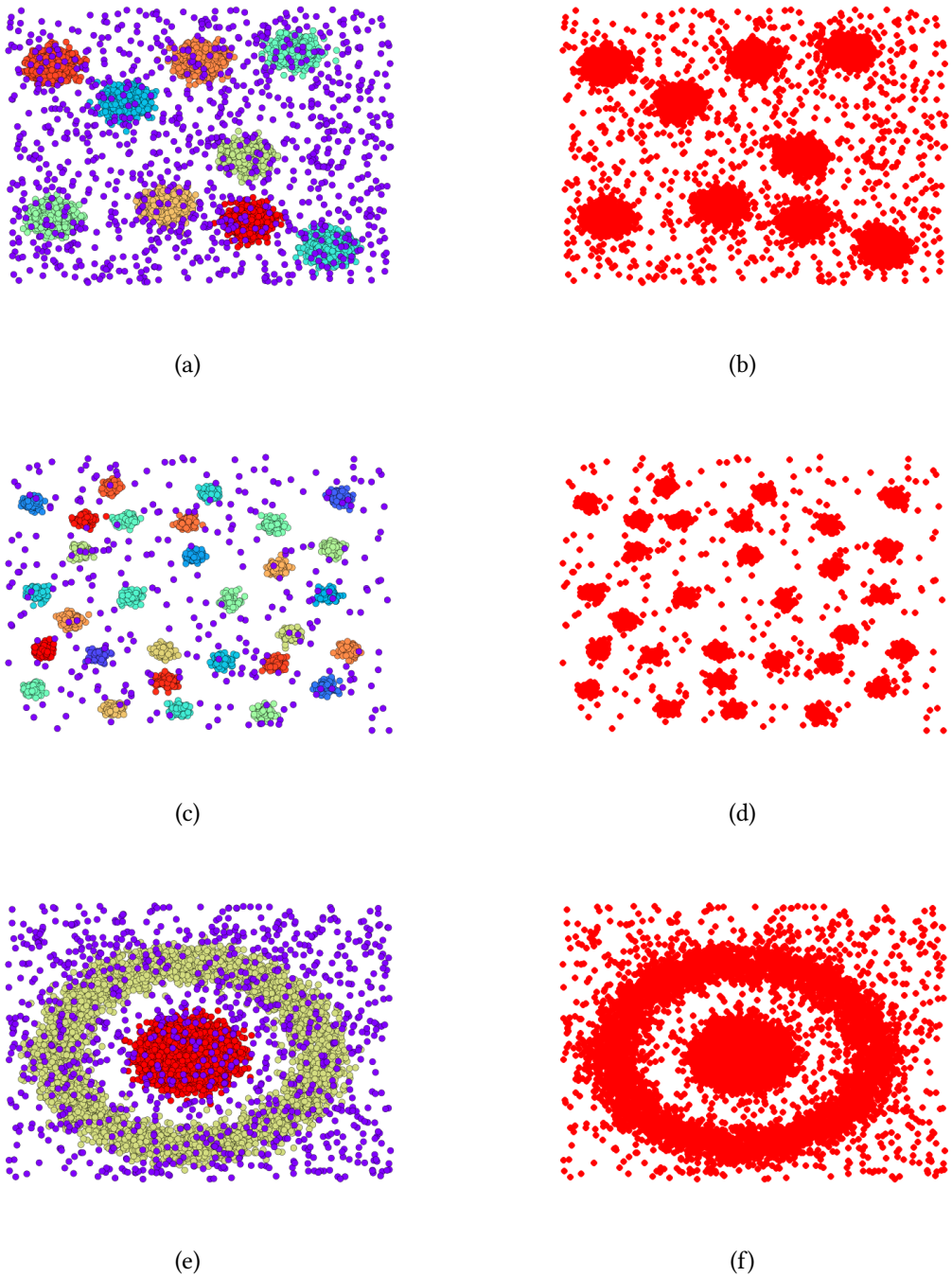
Diese Plattform beinhaltet vier Geforce GTX 680 Grafikkarten, welche jeweils eine Leistung von 128 GFLOP/s bei Operationen mit doppelter Genauigkeit haben.

Kepler Plattform

Die Kepler Plattform beinhaltet eine Xeon Phi E5-2650 mit 2,6 GHz und zwei GeForce K20Xm Grafikkarten.

SuperMIC Plattform

Die SuperMIC Plattform besteht aus 32 Knoten. Jeder dieser Knoten beinhaltet zwei Ivy-Bridge (2 x 8 Kerne) Hostprozessoren E5-2650 @ 2.6 GHz und 2 Intel Xeon Phi (MIC) Coprozessoren 5110P mit 60 Kernen @ 1.1 GHz.



(a) (b)

(c) (d)

(e) (f)

Abbildung 7.1.: Beispiele für generierte Datensätze

7.3. Tests auf der vgpu1 Plattform

Die vgpu1 ist die erste Plattform auf der wir Tests durchführen. Die Tests auf dieser Plattform bestehen aus einem Skalierungstest, und zwei Tests zur Untersuchung der Erkennungsraten.

7.3.1. vgpu1: Skalierungstest

Bei diesem Test wollen wir untersuchen, wie sich die Laufzeit bei unterschiedlichen Größen der Arbeitspakete und unterschiedlicher Anzahl von Rechenknoten entwickelt. Die Laufzeit des reinen OpenCL Programmes ohne eine Verteilung der Last mit MPI wurde zusätzlich noch in der Abbildung 7.2 rot eingezeichnet. Da wir auf dieser Plattform nur maximal vier Rechenknoten sinnvoll betreiben können und diese alle die gleiche Rechenleistung zur Verfügung haben, ist zu erwarten, dass wir bessere Laufzeiten erzielen können, wenn die Paketgröße erhöht wird.

Zusätzlich wurde der gleiche Test noch mit gleichverteilter Last durchgeführt. Es gibt bei diesem Test also für jedes Problem nur ein einziges Arbeitspaket für jeden Rechenknoten.

Konfiguration

Größe des Datensatzes:	Zweidimensionaler Datensatz mit 100k Datenpunkten
Größe des Rechengitters:	Fünfdimensionales dünnes Gitter mit 45057 Gitterpunkten
Größe der Arbeitspakete:	Zwischen 3000 und 4000 Workitems pro Arbeitspaket, oder gleichverteilte Last
k:	12

Resultate

Knoten	Paketgröße 3000	Paketgröße 4000	Last gleichverteilt
1	298.368s	230.641s	180.269s
2	158.13s	122.13s	107.907s
3	108.684s	82.729s	70.636s
4	84.381s	65.064s	65.46s

Tabelle 7.1.: Skalierungstest auf der vgpu1

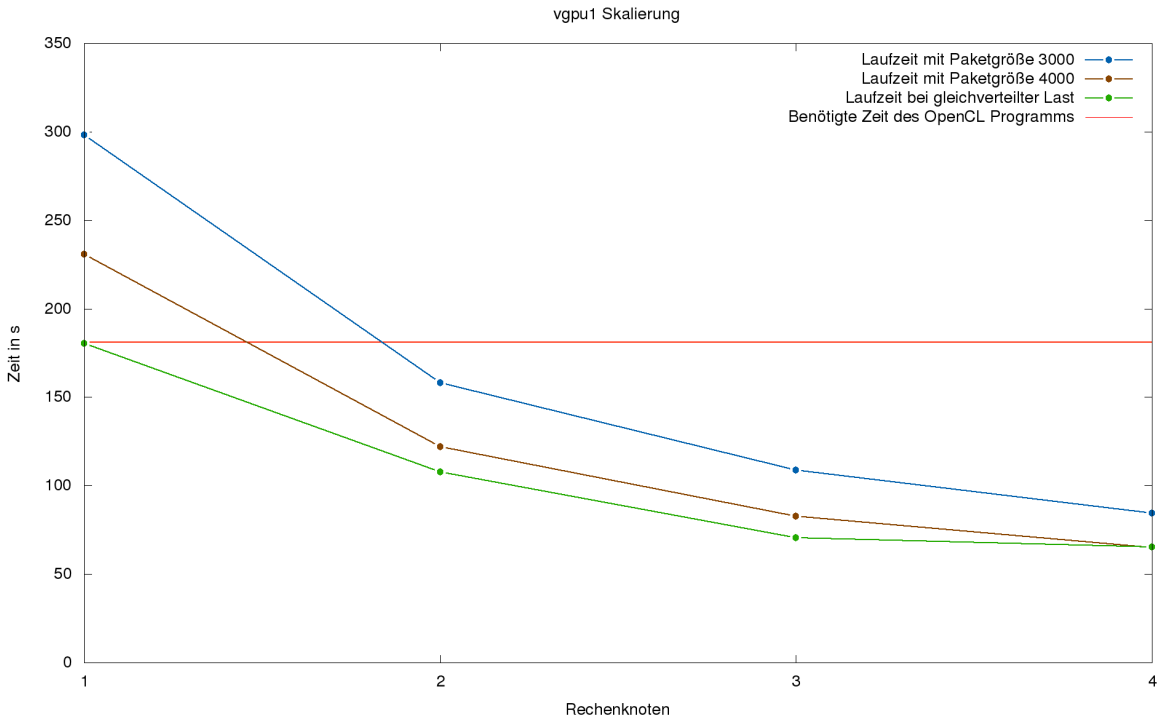


Abbildung 7.2.: Skalierungstest mit unterschiedlichen Paketgrößen auf der vgpu1

Analyse

Wir können deutlich erkennen, dass zu kleine Paketgrößen bei dieser Anzahl an Rechenknoten gleicher Leistung zu langsameren Laufzeiten führt. Abgesehen davon ist zu sehen, dass sich die Laufzeit bei Hinzunahme von mehr Rechenknoten ungefähr wie erwartet und die Rechenzeit mit p Knoten sich nahe an $\frac{T_{1\ Rechenknoten}}{p}$ befindet. Der Sprung von einem Rechenknoten auf vier führt bei einer Paketgröße von 4000 zum Beispiel zu einer Verringerung der Laufzeit auf 65s.

$$S(4) = \frac{T_{1\ Rechenknoten}}{T_{4\ Rechenknoten}} = \frac{230.641s}{65.064} \approx 3.55 \tag{7.1}$$

Der Speedup ist also nicht weit von dem idealen Speedup 4 entfernt. Die parallele Effizienz E(4) bei Paketgröße 4000 ist also

$$E(4) = \frac{S(4)}{4} = \frac{3.55}{4} = 0.8875 \tag{7.2}$$

Bei gleichverteilter Last ist außerdem zu sehen, dass die Leistung bei einem Rechenknoten äquivalent zu dem des OpenCL Programm ohne MPI ist.

7.3.2. vgpu1: Test 1 mit zehndimensionalen Datensatz

Für diesen Test haben wir einen Datensatz mit 10 Cluster erzeugt und zusätzlich noch 10000 Datenpunkte gleichverteiltes Rauschen hinzugefügt. Wir untersuchen nun, ob alle Cluster gefunden werden. Die Abstände zwischen den Clustern sind mindestens je 14 σ.

Konfiguration

Größe des Datensatzes:	Zehndimensionaler Datensatz mit 100k Datenpunkten
Art der Datenpunkte:	10 Cluster mit je 9000 Datenpunkten 10000 Datenpunkte als Rauschen
Größe des Rechengitters:	Zehndimensionales dünnes Gitter mit 77505 Gitterpunkten
Größe der Arbeitspakete:	4000 Workitems pro Arbeitspaket
Regularisierungsparameter λ:	10^{-7}
k:	12
Schwellwert κ:	0.35

Resultate

Laufzeit	923.58s mit 4 Rechenknoten
Anzahl der richtig erkannten Datenpunkte	97820 von 100000

Cluster	Gefundene Anzahl an Datenpunkten im Cluster	Korrekt erkannte Datenpunkte	Anteil des Originalclusters korrekt erkannt
1	8938	8935	99.28%
2	8993	8985	99.83%
3	9007	8990	99.89%
4	8990	8983	99.81%
5	9012	8995	99.94%
6	7420	7420	82.44%
7	8980	8971	99.68%
8	8913	8909	98.99%
9	8719	8719	96.88%
10	9016	8997	99.97%
Rauschen	12012	9916	99.16%

Tabelle 7.2.: Erkennungsrate eines einfachen, zehndimensionalen Datensatzes (Plattform: vgpu1)

Analyse

Die gute Erkennungsrate bei diesem Datensatz ist durch den Abstand der Cluster zu erklären. Da die Zentren der Cluster bei diesem Datensatz mindestens durch 14σ getrennt sind, wird bei der Generierung des Datensatzes automatisch eine sehr kleine Standardabweichung σ gewählt, was dazu führt, dass die Cluster sehr klein werden. Daher sind die meisten Datenpunkte, die innerhalb eines Clusters gefunden werden, tatsächlich auch Datenpunkte des Originalclusters, da nur wenige Datenpunkte des Rauschens direkt auf dem Cluster liegen.
Dieser Datensatz ist also relativ einfach zu clustern. Dies erklärt auch die schnelle Laufzeit, da man die Lösung des Gleichungssystems bereits früh abbrechen konnte und trotzdem gute Ergebnisse erhält.

7.3.3. vgpu1: Test 2 mit zehndimensionalen Datensatz

Da der Datensatz des letzten Tests sehr einfach zu clustern war, führen wir diesen Test hier noch einmal mit einem schwierigeren Datensatz durch. Die Zentren der Cluster sind nun nur noch 6 Standardabweichung voneinander getrennt was bedeutet, dass die Cluster selbst mehr Raum einnehmen. Es ist zu erwarten, dass wir nun verglichen mit dem ersten Test, weniger Datenpunkte richtig erkennen werden.

Konfiguration

Größe des Datensatzes:	Zehndimensionaler Datensatz mit 100k Datenpunkten
Art der Datenpunkte:	10 Cluster mit je 9000 Datenpunkten
	10000 Datenpunkte als Rauschen
	Abstände zwischen den Clustern sind mindestens je 6 σ
Größe des Rechengitters:	Zehndimensionales dünnes Gitter mit 77505 Gitterpunkten
Größe der Arbeitspakete:	4000 Workitems pro Arbeitspaket
Regularisierungsparameter λ:	10^{-7}
k:	6
Schwellwert:	0.48

Resultate

Laufzeit	1320.55s mit 4 Rechenknoten
Anzahl der richtig erkannten Datenpunkte	64232 von 10000
Zu kleine Cluster	20 Datenpunkten ignoriert, da deren Cluster je weniger als 50 Datenpunkte beinhalten

Cluster	Gefundene Anzahl an Datenpunkten im Cluster	Korrekt erkannte Datenpunkte	Anteil des Originalclusters korrekt erkannt
1	6834	6830	75,89%
2	2048	2048	22,76%
3	6544	6538	72,64%
4	7734	7733	85,92%
5	5650	5650	62,78%
6	7106	7105	78,94%
7	2578	2577	28,63%
8	2117	2117	23,52%
9	7010	7010	77,89%
10	6640	6638	73,76%
Rauschen	45719	9986	99,86%

Tabelle 7.3.: Erkennungsrate eines schwierigen, zehndimensionalen Datensatzes (Plattform: vgpu1)

Analyse

Es werden zwar noch alle 10 ursprünglichen Cluster gefunden, aber um dieses Ergebnis zu erreichen, mussten einige Parameter verändert werden. Man muss fast doppelt soviele Iterationen des CG-Verfahrens durchführen und sowohl k auf 6 herabsetzen als auch den Schwellwert auf 0.48 erhöhen.
Die erhöhte Anzahl an Iteration erklärt auch den Anstieg der Laufzeit auf 1320.55 Sekunden.
Die Erhöhung des Schwellwertes führt leider auch dazu, dass die erkannten Cluster nun wesentlich kleiner sind, da viele Datenpunkte entfernt wurden. Zudem werden nun einige kleine Cluster erkannt, die eigentlich nicht im Datensatz vorhanden sind. Diese kann man einfach herausfiltern, indem man eine Mindestgröße für die Cluster einführt.

7.4. Tests auf der Kepler Plattform

7.4.1. Kepler: Skalierungstest

Bei diesem Test soll die Entwicklung der Laufzeit bei unterschiedlichen Paketgrößen und unterschiedlicher Anzahl an Rechenknoten untersucht werdem.

Konfiguration

Größe des Datensatzes:	Zweidimensionaler Datensatz mit 100k Datenpunkten
Größe des Rechengitters:	Zweidimensionales dünnes Gitter mit 98305 Gitterpunkten
Größe der Arbeitspakete:	Zwischen 4000 und 7000 Workitems pro Arbeitspaket
k:	12

Resultate

Knoten	Paketgröße 4000	Paketgröße 5000	Paketgröße 6000	Paketgröße 7000
1	961.544s	768.462s	656.577s	576.981s
2	497.298s	388.152s	345.989s	304.346s
3	232.301s	212.336s	196.644s	170.558s

Tabelle 7.4.: Skalierungstest auf der Kepler

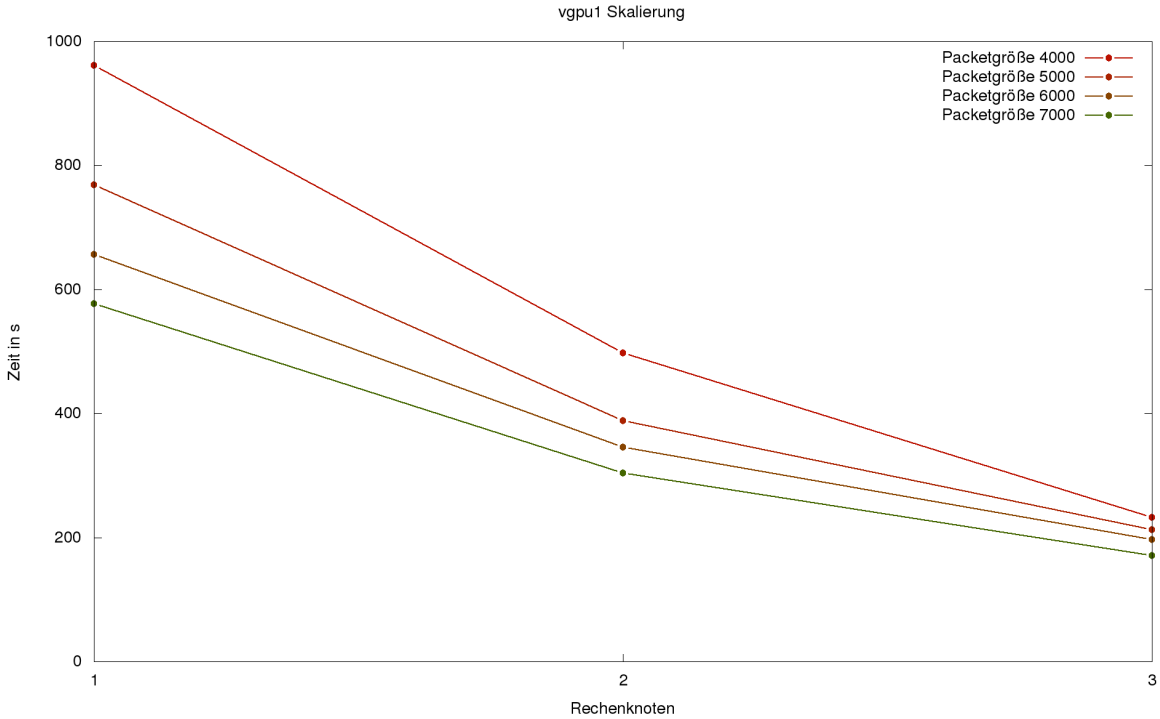


Abbildung 7.3.: Skalierungstest auf der Kepler

7.4.2. Analyse

Die Laufzeit sinkt bei erhöhter Paketgröße und erhöhter Anzahl an Rechenknoten. Der erste und der zweite Rechenknoten nutzen jeweils eine K20Xm, der dritte Rechenknoten die Intel(R) Xeon(R) CPU E5-2650 v2. Man erkennt, dass die Erhöhung von zwei auf drei Rechenknoten bei der Paketgröße 4000 mehr als halbiert. Auch bei den anderen Paketgrößen bringt der dritte Rechenknoten eine größere Reduktion der Laufzeit als zu erwarten wäre, gerade da dieser Knoten die CPU verwendet. Diese hat eine schwächere Rechenleistung, als die K20Xm, führt aber die OpenCL Kernel für das Programm dennoch schneller aus. Falls man die OpenCL Kernel explizit für die K20Xm Grafikkarten anpassen würde, könnte man also auf dieser Plattform die Leistung noch verbessern.

7.5. Tests auf der SuperMIC Plattform

7.5.1. SuperMIC Skalierungstest mit festem Datensatz und festem Gitter

Da wir nun eine Plattform zur Verfügung haben, bei der sich große Anzahl von Rechenknoten lohnen, untersuchen wir hier die Laufzeit des Programmes für 1 bis 39 Rechenknoten. Jeder dieser Rechenknoten verwendet einen MIC Coprozessor (Many Integrated Cores).

Konfiguration

Größe des Datensatzes:	Zweidimensionaler Datensatz mit 100k Datenpunkten
Größe des Rechengitters:	Zweidimensionales dünnes Gitter mit 458753 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket
k:	12

Resultate

Genutztes MICs	Benötigte Zeit	Genutztes MICs	Benötigte Zeit
1	5571.01s	13	507.853s
3	1925.41s	19	438.266s
5	1212.94s	23	366.131s
7	862.993s	29	300.212s
9	698.549s	39	218.858s

Tabelle 7.5.: SuperMIC Skalierungstest

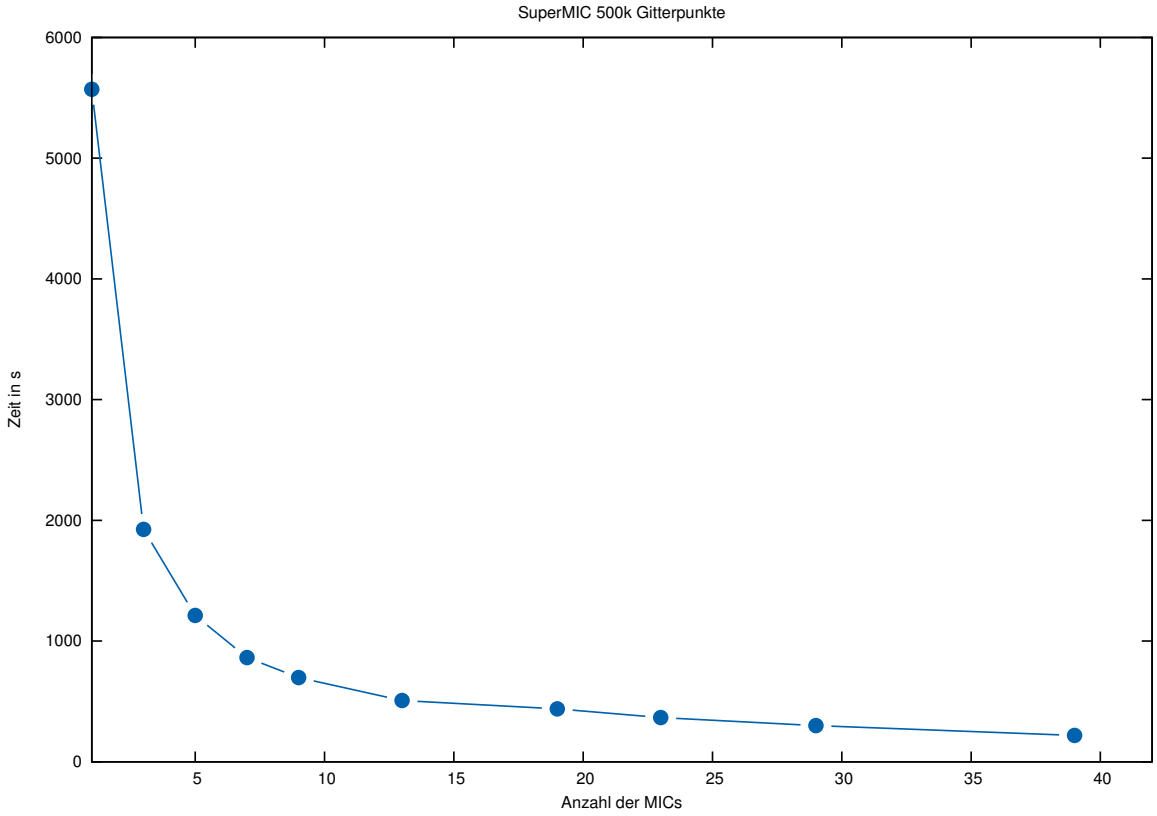


Abbildung 7.4.: SuperMIC Skalierungstest

Analyse

Der Speedup und die parallele Effizienz bei der Nutzung von 3 Rechenknoten ist

$$S(3) = \frac{T_{1\ Rechenknoten}}{T_{3\ Rechenknoten}} = \frac{5571.01s}{1925.41s} = 2.89 \tag{7.3}$$

$$E(3) = \frac{S(3)}{3} = \frac{2.89}{3} = 0.97 \tag{7.4}$$

Bei Nutzung von 39 Rechenknoten erhalten für den Speedup und die parallele Effizienz folgende Werte:

$$S(39) = \frac{T_{1\ Rechenknoten}}{T_{39\ Rechenknoten}} = \frac{5571.01s}{218.858s} = 24.46 \tag{7.5}$$

$$E(39) = \frac{S(39)}{39} = \frac{24.46}{39} = 0.63 \tag{7.6}$$

Der Speedup S(29) ist 18.56, der Speedup von S(39) ist 24,46. Also wird selbst bei der letzten untersuchten Erhöhung der Rechenknoten, noch eine Verringerung der Laufzeit erreicht. Wie man erkennen kann, bringt das Hinzufügen von weiteren Rechenknoten keinen so

großen Vorteil mehr, falls man bereits viele Rechenknoten verwendet, ist aber dennoch noch bemerkbar.

7.5.2. SuperMIC Skalierungstest mit variablen zweidimensionalen Datensatz und festem Gitter

Hier testen wir, wie sich die Laufzeit des Programms bei Vergrößerung der Datensätze entwickelt. Im Appendix sind drei weitere dieser Tests, mit drei-, vier- und fünfdimensionalen Datensätze, zu finden.

Konfiguration

Größe des Datensatzes:	Zweidimensionaler Datensatz zwischen 500k und 1000k Datenpunkten
Größe des Rechengitters:	Zweidimensionales dünnes Gitter mit 458753 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket
k:	12

Resultate

Anzahl der Datenpunkte	9 MICs	19 MICs	29 MICs	39 MICs
500000	1157.63s	635.68s	406.271s	339.46s
600000	1312.07s	712.10s	474.163s	352.75s
700000	1428.26s	800.45s	550.479s	367.67s
800000	1611.59s	862.43s	573.862s	442.97s
900000	1810.76s	931.01s	662.098s	462.34s
1000000	2029.62s	1037.82s	691.652s	550.45s

Tabelle 7.6.: SuperMIC Test mit zweidimensionalen Datensätzen variabler Größe

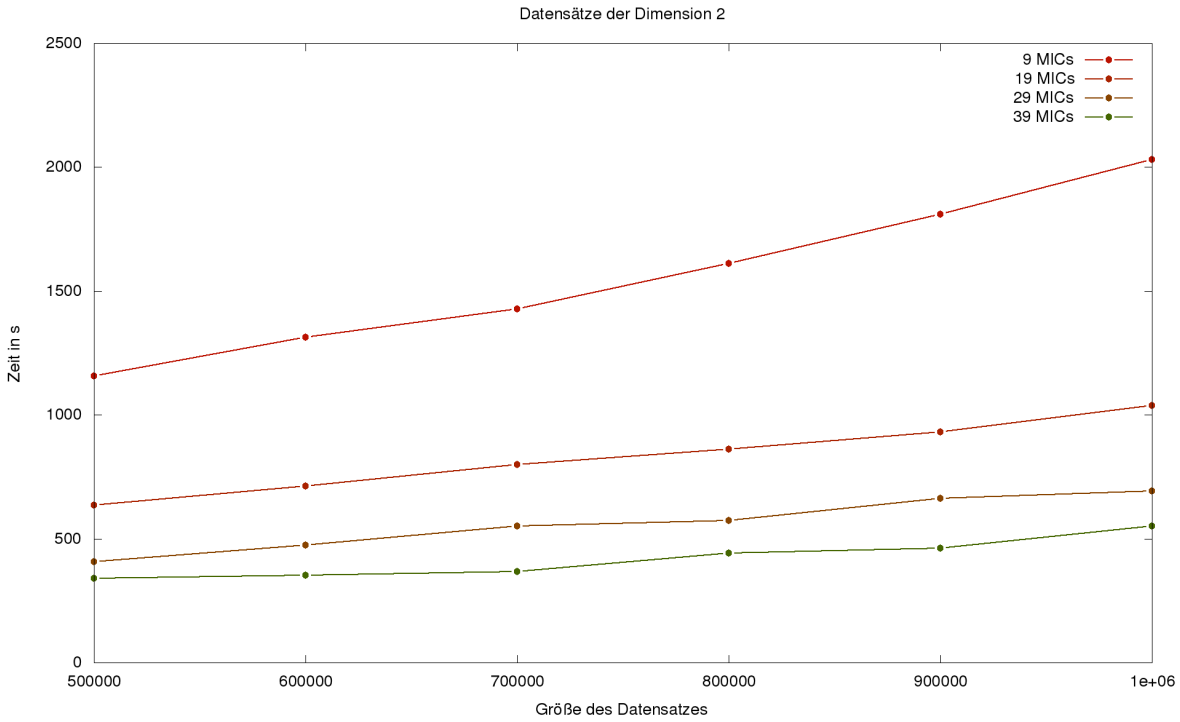


Abbildung 7.5.: SuperMIC Test mit zweidimensionalen Datensätzen variabler Größe

Analyse

Wie man erkennen kann entwickelt sich die Laufzeit, bei größer werdenden Datensätzen, nahezu linear. Dies ist dadurch zu erklären, dass das Erstellen der Dichteschätzung die Laufzeit dominiert. Dieses Erstellen der Dichteschätzung ist nur linear von der Anzahl der Datenpunkte abhängig.
Für die, im Appendix hinzugefügten, Tests mit höheren Dimensionen ergibt dieselbe Entwicklung.

7.5.3. SuperMIC zehndimensionaler Datensatz mit wenig Rauschen

Hier testen wir die Erkennungsrate für einen Zehndimensionalen Datensatz mit einer Million Datenpunkte. Bei diesem Datensatz sind nur 1% der Punkte Rauschen, die Erkennungrate sollte also recht hoch sein.

Konfiguration

Größe des Datensatzes:	Zehndimensionaler Datensatz mit 1000k Datenpunkten
Art der Datenpunkte:	10 Cluster mit je 99000 Datenpunkten 10000 Datenpunkte als Rauschen Abstände zwischen den Clustern sind mindestens je 10 σ
Größe des Rechengitters:	Zehndimensionales dünnes Gitter mit 397825 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket
Regularisierungsparameter λ:	$8(10^{-8})$

Resultate

Laufzeit	1634s mit 59 MICs
Anzahl der richtig erkannten Datenpunkte	982509 von 1000000

Cluster	Gefundene Anzahl an Datenpunkten im Cluster	Korrekt erkannte Datenpunkte	Anteil des Originalclusters korrekt erkannt
1	96983	96980	97.96%
2	98820	98729	99.73%
3	96731	96728	97.71%
4	97824	97819	98.81%
5	97104	97099	98.08%
6	98252	98244	99.24%
7	96457	96444	97.42%
8	98528	98491	99.49%
9	96597	95563	97.56%
10	95570	96588	96.53%
Rauschen	27134	9824	98.24%

Tabelle 7.7.: Erkennungsrate eines großen zehndimensionalen Datensatzes (Plattform: SuperMIC)

Analyse

Wie erwartet ist die Erkennungsrate für diesen Datensatz sehr hoch. Es wurden alle 10 Cluster erkannt und die meisten Datenpunkte richtig zugeordnet. Da dieser Datensatz sehr wenig Rauschen beinhaltet, können wir einen recht geringer Schwellwert für die Erstellung des Graphen G' verwenden. Dies führt dazu, dass nur wenige Knoten aus dem Graph entfernt werden müssen und die meisten Datenpunkte der originalen Cluster erkannt werden.

7.5.4. SuperMIC zehndimensionaler Datensatz mit starkem Rauschen

Da wir bei dem letzten Test einen Datensatz vorliegen hatten, bei dem nur 1% der Datenpunkte Rauschen waren, nutzen wir dieses Mal einen gleich großen Datensatz mit 20% Rauschen. Bei diesem Datensatz haben wir außerdem den Mindestabstand zwischen den Clustern auf 8 σ verringert, die Datenpunkte liegen also nicht mehr so konzentriert auf einiges Gebieten wie zuvor.

Konfiguration

Größe des Datensatzes:	Zehndimensionaler Datensatz mit 1000k Datenpunkten
Art der Datenpunkte:	10 Cluster mit je 80000 Datenpunkten
	200000 Datenpunkte als Rauschen
	Abstände zwischen den Clustern sind mindestens je 8 σ
Größe des Rechengitters:	Zehndimensionales dünnes Gitter mit 397825 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket
Regularisierungsparameter λ:	$8(10^{-8})$

Resultate

Laufzeit	1434.11s mit 59 MICs
Anzahl der richtig erkannten Datenpunkte	889906 von 1000000
Zu kleine Cluster	87 Datenpunkten ignoriert, da deren Cluster je weniger als 50 Datenpunkte beinhalten

Cluster	Gefundene Anzahl an Datenpunkten im Cluster	Korrekt erkannte Datenpunkte	Anteil des Originalclusters korrekt erkannt
1	77097	77002	96.25%
2	68145	68095	85.12%
3	65767	65727	82.16%
4	78062	77816	97.27%
5	60419	60411	75.51%
6	73949	73861	92.33%
7	75376	75299	94.12%
8	44092	44090	55.11%
9	72621	72594	90.74%
10	75654	75645	94.56%
Rauschen	308731	199348	99.67%

Tabelle 7.8.: Erkennungsrate eines großen verrauschten zehndimensionalen Datensatzes (Plattform: SuperMIC)

Analyse

Es wurden alle 10 Cluster korrekt erkannt. Allerdings mussten wir für dieses Resultat den Parameter k von 12 auf 4 herabsetzen. Dies hat die Erkennungsrate deutlich verbessert, da man ansonsten den Schwellwert hätte erhöhen müssen.
Die Verringerung von k hatte zwei weitere Nebeneffekte. Die Laufzeit wurde im Vergleich zum vorherigen Test um fast 200 Sekunden verringert, da die Adjazenzliste kleiner ist und wir auf weniger Kanten die Dichte prüfen müssen. Der zweite Nebeneffekt ist, dass 87 sehr kleine Cluster gefunden wurden. Diese lassen sich jedoch sehr einfach herausfiltern.

8. Zusammenfassung und Ausblick

In dieser Arbeit haben wir den Clustering Algorithmus von Peherstorfer ([PPB12]) mit OpenCL implementiert. Mit dieser Implementierung sind wir in der Lage beliebige OpenCL-fähige Geräte zur Ausführung des Clusterings zu verwenden. Zusätzlich haben wir den Algorithmus mit MPI auf mehrere Rechner, oder Rechenknoten, verteilt, von denen jeder die OpenCL Implementierung nutzt, um einen Teil des Clustering Problems zu lösen.
Um die Last auf die einzelnen Rechner zu verteilen, nutzen wir ein Paketsystem mit einer Warteschlange. Mit diesem System der Lastverteilung können wir die einzelnen Rechenknoten möglichst konstant auslasten, auch wenn die Knoten über unterschiedliche Rechenleistung verfügen.
Dies erlaubt uns, den Clustering Algorithmus auf einer Vielzahl von unterschiedlichen Plattformen auszuführen und dabei gute Laufzeiten erzielen. Die einzige Voraussetzung ist, dass jeder Rechenknoten ein eigenes OpenCL-fähiges Gerät zur Verfügung hat.
Eine der Plattformen, die in dieser Arbeit genutzt wurde, ist die SuperMIC. Bei Tests auf dieser Plattform haben wir mit Nutzung von 39 Rechenknoten einen Speedup von 24,46 und parallele Effizienz von 0.62 erreicht. Bei Nutzung von weniger Rechenknoten ist die parallele Effizienz erwartungsgemäß höher. Auf der vgpu1 Plattform mit vier Grafikkarten, konnte bei Nutzung von vier Rechenknoten ein Speedup von 3.55 und eine parallele Effizienz von 0.8875 erreicht werden.
Weiterhin haben wir die Erkennungsrate mit mehreren synthetischen Datensätzen getestet. Wir sind mit unserer Implementierung in der Lage, die Cluster eines zehndimensionalen Datensatz mit einer Million Datenpunkten in 1434.11 Sekunden zu finden. In diesem Datensatz waren 20% der Punkte lediglich durch gleichverteiltes Rauschen gegeben, aber es wurden dennoch alle 10 Cluster gefunden und 88.99 % der Datenpunkte richtig zugeordnet.
Um diese Tests auszuführen und zu analysieren, sind im Zuge dieser Arbeit, neben dem eigentlichen MPI Clustering Programm, mehrere andere Anwendungen entwickelt worden. Zum einen wurde ein Programm entwickelt, zum Erstellen der synthetischen Datensätzen und zum Analysieren der Erkennungsrate. Zum anderen wurde noch eine GUI in Python entwickelt, mit der schnell Gitter und Datensätze unterschiedlicher Größe erstellt werden können und man sowohl die reine OpenCL Implementierung des Clustering Algorithmus, als auch die MPI Implementierung zum Clustering verwenden kann. Mit den Ergebnissen des Clusterings kann man in der GUI eine Visualisierung der Dichtefunktion und des k-nearest-neighbor Graphen G erstellen. Beispielsweise wurden die viele Abbildungen in dieser Arbeit mit dieser Anwendung erzeugt.
Um die Performance dieser Implementierung weiter zu erhöhen, ist es möglich, die OpenCL Kernel für die einzelnen Arbeitsschritte zu verbessern. Wie wir in dem Test auf Kepler

Plattform sehen konnten, ist der OpenCL Code zwar für sich genommen plattformunabhängig, aber man kann die Rechenleistung verbessern, wenn man seinen Code an die Hardware anpasst, um beispielsweise schnelleren Speicher zu verwenden.
Um die Laufzeit weiter zu verbessern, kann man ausnutzen, dass die Matrix, mit der wir unsere Multiplikation durchführen, symmetrisch ist. Da Workitems in OpenCL innerhalb von Workgroups synchronisiert werden können, ist es dadurch möglich, durch geschickte Aufteilung zuvor berechnete Einträge wieder zu verwenden.
Um auch mit Datensätzen umgehen zu können, die zu groß für den Speicher der einzelnen Rechenknoten sind, ist es nötig, den Datensatz vor dem Clustering zu partitionieren. Auf diese Weise kann man den Rechenknoten immer nur einen Teil des Datensatzes schicken. Eine Partitionierung des Datensatzes in Gebiete verbessert außerdem die Laufzeit des k-nearest-neighbors Algorithmus, da man für jeden Knoten nur einen Teil des Datensatzes nach den k nächsten Nachbarn untersuchen muss, anstatt wie bisher den gesamten Datensatz. In dieser Arbeit war allerdings kein Datensatz groß genug, um auf den einzelnen Rechenknoten zu einer Speicherknappheit zu führen.

A. Sonstige Tests

A.1. SuperMIC Skalierungstest mit variablen dreidimensionalen Datensatz und festem Gitter

Konfiguration

Größe des Datensatzes:	Dreidimensionaler Datensatz zwischen 500k und 1000k Datenpunkten
Größe des Rechengitters:	Dreidimensionales dünnes Gitter mit 471041 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket

Resultate

Anzahl der Datenpunkte	9 MICs	19 MICs	29 MICs	39 MICs
500000	2520.4s	1385.64s	979.42s	864.575s
600000	2776.86s	1513.68s	1095.15s	886.147s
700000	2963.51s	1656.75s	1223.05s	908.715s
800000	3263.84s	1758.16s	1259.59s	1034.0s
900000	3590.2s	1867.75s	1406.41s	1065.41s
1000000	3939.63	2040.53s	1447.88s	1208.53s

Tabelle A.1.: SuperMIC Test mit dreidimensionalen Datensätzen variabler Größe

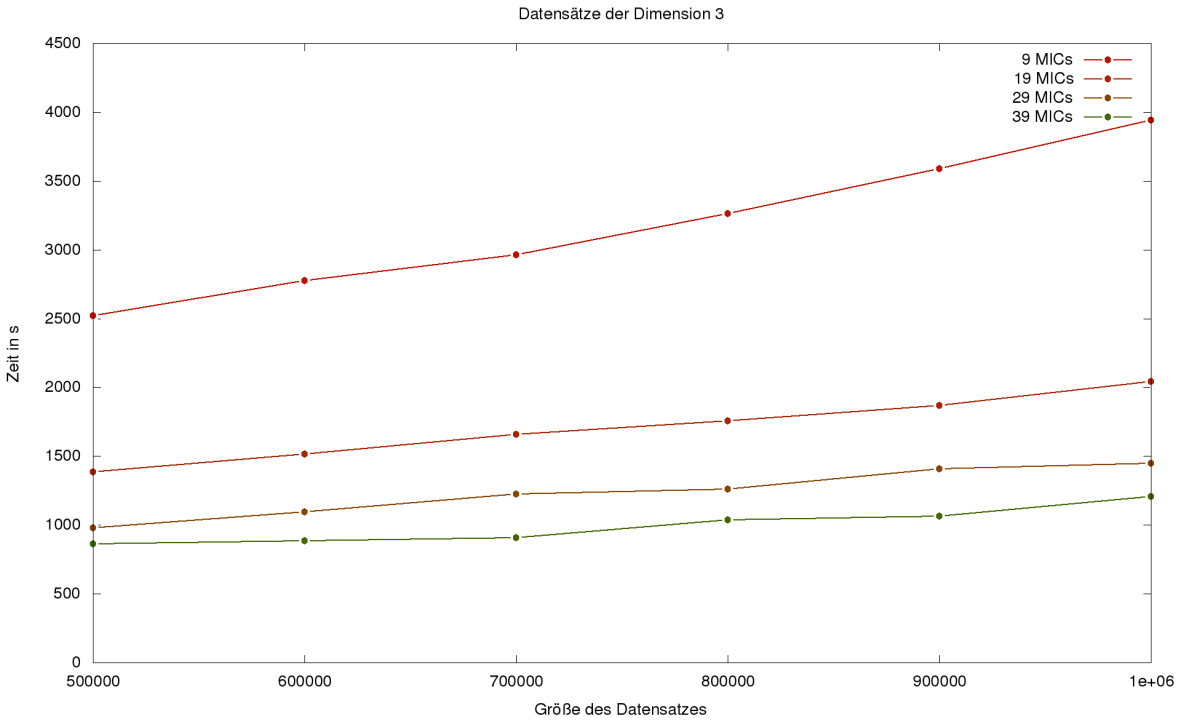


Abbildung A.1.: SuperMIC Test mit dreidimensionalen Datensätzen variabler Größe

A.2. SuperMIC Skalierungstest mit variablen vierdimensionalen Datensatz und festem Gitter

Konfiguration

Größe des Datensatzes:	Vierdimensionaler Datensatz zwischen 500k und 1000k Datenpunkten
Größe des Rechengitters:	Vierdimensionales dünnes Gitter mit 647167 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket

Resultate

Anzahl der Datenpunkte	9 MICs	19 MICs	29 MICs	39 MICs
500000	1873.74s	1031.96s	659.627s	653.49s
600000	2133.92s	1162.43s	772.97s	678.08s
700000	2327.48s	1308.22s	846.697s	703s
800000	2634.69s	1361.82s	940.133s	830.173s
900000	2972.1s	1528.76s	1089.19s	863.303s
1000000	3335.82s	1707.84s	1136.65s	1010.42s

Tabelle A.2.: SuperMIC Test mit vierdimensionalen Datensätzen variabler Größe

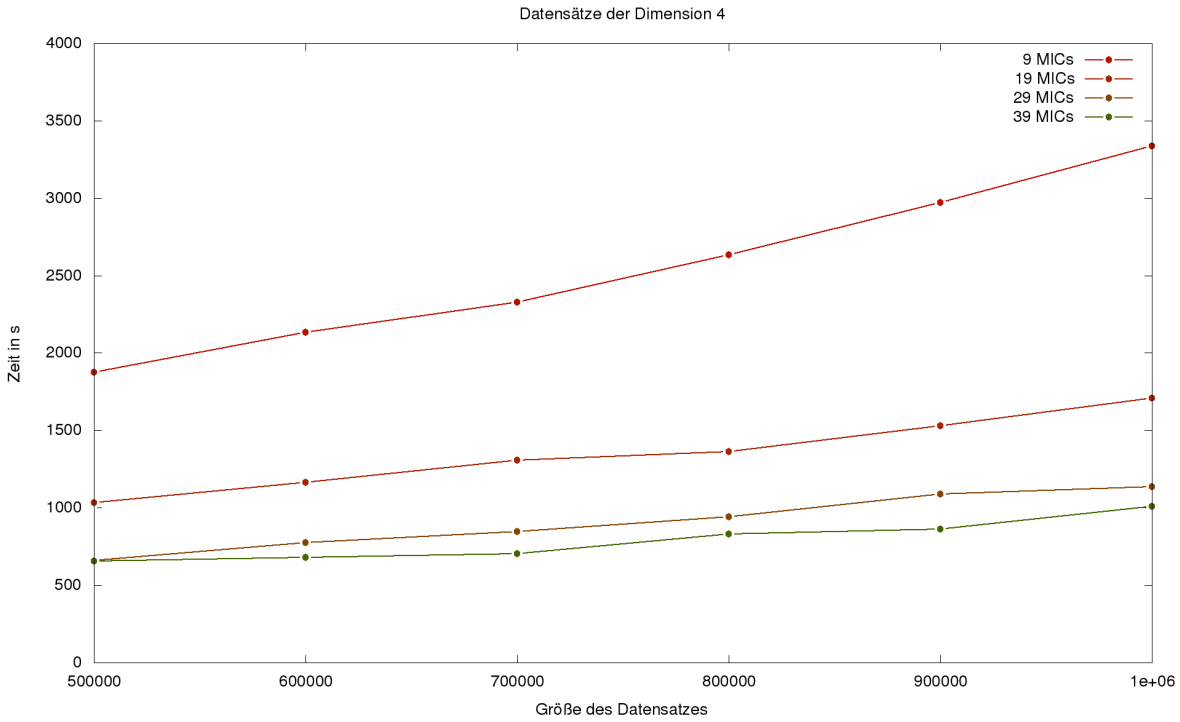


Abbildung A.2.: SuperMIC Test mit vierdimensionalen Datensätzen variabler Größe

A.3. SuperMIC Skalierungstest mit variablen fünfdimensionalen Datensatz und festem Gitter

Konfiguration

Größe des Datensatzes:	Fünfdimensionaler Datensatz zwischen 500k und 1000k Datenpunkten
Größe des Rechengitters:	Fünfdimensionales dünnes Gitter mit 553983 Gitterpunkten
Größe der Arbeitspakete:	6000 Workitems pro Arbeitspaket

Resultate

Anzahl der Datenpunkte	9 MICs	19 MICs	29 MICs	39 MICs
500000	2744.22s	1319.85s	1024.8s	797.649s
600000	3043.59s	1472.24s	1156.26s	809.183s
700000	3324.74s	1683.65s	1201.53s	843.738s
800000	3862.09s	1876.03s	1438.5s	1054.99s
900000	4355.91s	2041.22	1655.29s	1099.55s
1000000	4893.75s	2307.79	1726.41s	1318.56s

Tabelle A.3.: SuperMIC Test mit Fünfdimensionalen Datensätzen variabler Größe

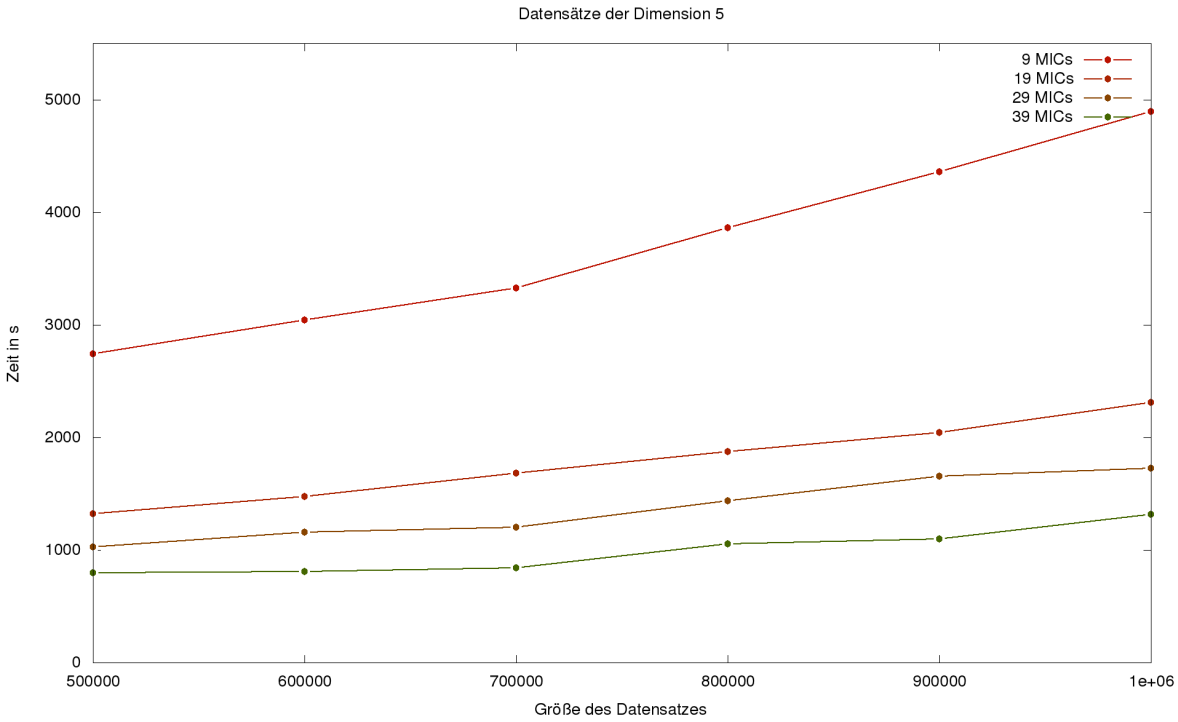


Abbildung A.3.: SuperMIC Test mit Fünfdimensionalen Datensätzen variabler Größe

Literaturverzeichnis

[BG04] H.-J. Bungartz und M. Griebel. „Sparse grids“. In: *Acta numerica* 13 (2004), S. 147–269 (Zitiert auf S. 13, 16, 21).

[CND14] S. Chakraborty, N. K. Nagwani und L. Dey. „Weather Forecasting using Incremental K-means Clustering“. In: *CoRR* abs/1406.4756 (2014). URL: http://arxiv.org/abs/1406.4756 (Zitiert auf S. 11).

[CSD11] A. Cuzzocrea, I.-Y. Song und K. C. Davis. „Analytics over Large-scale Multidimensional Data: The Big Data Revolution!“ In: *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*. DOLAP '11. Glasgow, Scotland, UK: ACM, 2011, S. 101–104. ISBN: 978-1-4503-0963-9. DOI: 10.1145/2064676.2064695. URL: http://doi.acm.org/10.1145/2064676.2064695 (Zitiert auf S. 11).

[DH04] C. Ding und X. He. „K-nearest-neighbor Consistency in Data Clustering: Incorporating Local Information into Global Optimization“. In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. Nicosia, Cyprus: ACM, 2004, S. 584–589. ISBN: 1-58113-812-1. DOI: 10.1145/967900.968021. URL: http://doi.acm.org/10.1145/967900.968021 (Zitiert auf S. 33).

[HHR00] M. Hegland, G. Hooker und S. Roberts. *Finite Element Thin Plate Splines in Density Estimation*. 2000 (Zitiert auf S. 25, 26).

[HJ97] P. Hansen und B. Jaumard. „Cluster Analysis and Mathematical Programming“. In: *Math. Program.* 79.1-3 (Okt. 1997), S. 191–215. ISSN: 0025-5610. DOI: 10.1007/BF02614317. URL: http://dx.doi.org/10.1007/BF02614317 (Zitiert auf S. 29).

[JMF99] A. K. Jain, M. N. Murty und P. J. Flynn. „Data Clustering: A Review“. In: *ACM Comput. Surv.* 31.3 (Sep. 1999), S. 264–323. ISSN: 0360-0300. DOI: 10.1145/331499.331504. URL: http://doi.acm.org/10.1145/331499.331504 (Zitiert auf S. 11).

[JT97] P. Jimack und N. Touheed. „An introduction to MPI for computational mechanics“. In: *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools* (1997), S. 24–25 (Zitiert auf S. 12).

[Mad12] T. S. Madhulatha. „An overview on clustering methods“. In: *arXiv preprint arXiv:1205.1117* (2012) (Zitiert auf S. 29).

[OES07] M. G. Omran, A. P. Engelbrecht und A. Salman. „An overview of clustering methods“. In: *Intelligent Data Analysis* 11.6 (2007), S. 583–605 (Zitiert auf S. 29).

[PPB10] D. Pflüger, B. Peherstorfer und H.-J. Bungartz. „Spatially adaptive sparse grids for high-dimensional data-driven problems“. In: *Journal of Complexity* 26.5 (2010). SI: {HDA} 2009, S. 508–522. ISSN: 0885-064X. DOI: http://dx.doi.org/10.1016/j.jco.2010.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0885064X10000257 (Zitiert auf S. 16, 23, 37).

[PPB12] B. Peherstorfer, D. Pflüger und H.-J. Bungartz. „Clustering Based on Density Estimation with Sparse Grids“. In: *Proceedings of the 35th Annual German Conference on Advances in Artificial Intelligence*. KI'12. Saarbrücken, Germany: Springer-Verlag, 2012, S. 131–142. ISBN: 978-3-642-33346-0. DOI: 10.1007/978-3-642-33347-7_12. URL: http://dx.doi.org/10.1007/978-3-642-33347-7_12 (Zitiert auf S. 11, 13, 26, 27, 30, 34, 43, 73).

[Raj11] S. Rajagopal. „Customer Data Clustering using Data Mining Technique“. In: *CoRR* abs/1112.2663 (2011). URL: http://arxiv.org/abs/1112.2663 (Zitiert auf S. 11).

[RGB14] P. Ramachandran, N. Girija und T. Bhuvaneswari. „Article: Early Detection and Prevention of Cancer using Data Mining Techniques“. In: *International Journal of Computer Applications* 97.13 (Juli 2014). Full text available, S. 48–53 (Zitiert auf S. 11).

[She94] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Techn. Ber. Pittsburgh, PA, USA, 1994 (Zitiert auf S. 34).

[TS12] J. Tompson und K. Schlachter. „An Introduction to the OpenCL Programming Model“. In: (2012) (Zitiert auf S. 12, 44).

[XW+05] R. Xu, D. Wunsch et al. „Survey of clustering algorithms“. In: *Neural Networks, IEEE Transactions on* 16.3 (2005), S. 645–678 (Zitiert auf S. 29).

Alle URLs wurden zuletzt am 24. 11. 2015 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