Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Verteilte Dünngitter-Regression mit SG++ und HPX

Philipp Wundrack

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Dirk Pflüger
Betreuer/in:	M.Sc. Gregor Daiß
Beginn am:	20. September 2018
Beendet am:	20. März 2019

Kurzfassung

Datamining und Big Data werden immer wichtiger für viele Forschungsgebiete und die Menge an Daten, die gesammelt werden steigt stetig an. Für besonders große Datensätzen ist Dünngitter-Regression ein geeignetes Verfahren, da es linear mit der Anzahl der Datenpunkte skaliert und es sich zudem einfach parallelisieren lässt. Bei verteilter Ausführung von Programmen wird üblicherweise das Message Passing Interface (MPI) zur Kommunikation eingesetzt, doch dieser inzwischen zwei Jahrzehnte alte Standard ist den neuen Herausforderungen wie heterogener Hardware und Exascale Computing nicht mehr gewachsen. Die High Performance ParalleX (HPX) Bibliothek versucht diese Probleme zu lösen, um eine zukunftsfähige Runtime für parallele und verteilte Ausführung bereitzustellen. Wir haben HPX hier genutzt, um die Dünngitter Bibliothek SG++ mit einem verteilt ausführbaren Regressions-Algorithmus zu erweitern. Dabei wurde besonderer Wert auf gute Skalierbarkeit gelegt, für eine große Anzahl an Rechenknoten. Es hat sich gezeigt, dass sich mithilfe von HPX die verteilte, parallele Ausführung und asynchrone Kommunikation zwischen den Rechenkonten unkompliziert umsetzen lässt. Außerdem lässt sich das Programm effizient auf viele Knoten skalieren, dank der latenzversteckenden Eigenschaften von HPX.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Unter Regression versteht man das Schätzen oder Vorhersagen eines kontinuierlichen Zielwertes [Lar05]. Dazu bekommt ein Regression-Algorithmus einen Datensatz von Datenpunkten und Zielwerten und soll eine Funktion finden, die die Zielwerte möglichst genau Schätzen kann [FPS96]. Ein möglicher Funktionsraum, in dem eine solche Funktion gefunden werden kann ist der, der von Dünngittern aufgespannt wird [HKPB16; Pfl10]. Wir haben für diese Arbeit die Dünngitter Bibliothek SG++ erweitert, sodass man mit ihr Dünngitter-Regression verteilt auf vielen Rechenknoten ausführen kann. Dazu haben wir die High Performance ParalleX (HPX) Bibliothek verwendet. Üblicherweise wird dafür das Message Passing Interface (MPI) genutzt, aber HPX verspricht mehrere Vorteile gegenüber diesem [KHA+14].

In Kapitel 2 wird zunächst definiert, was Regression ist. Dann werden volle Gitter im eindimensionalen und danach im mehrdimensionalen Fall beschrieben. Daraufhin wird die hierarchische Darstellung von Gittern beschrieben und wie Dünngitter diese nutzen, um die Laufzeitkomplexität zu reduzieren. Zuletzt wird erläutert, wie Regression mithilfe von Dünngittern ausgeführt werden kann.

In Kapitel 3 geht es um die HPX Bibliothek. Es wird die Architektur dieses Laufzeitsystems und die API beschrieben.

In Kapitel 4 beschreiben wir, wie sich die Dünngitter-Regression parallelisieren lässt auf einem einzelnen Rechenknoten. Wir zeigen beispielhaft, wie man das mit HPX erreichen kann und es wird kurz OpenCL eingeführt.

In Kapitel 5 zeigen wir, wie wir HPX genutzt haben für die verteilte Ausführung der Dünngitter-Regression.

In Kapitel 6 präsentieren wir die Ergebnisse unserer Experimente. In einem geht es um das Skalierungsverhalten auf vielen Rechenknoten und in dem anderen werden die Ergebnisse der Dünngitter-Regression mit anderen Regressions-Algorithmen verglichen.

In Kapitel 7 gibt es eine Zusammenfassung und einen Ausblick darauf, was noch verbessert werden kann.

2 Regression mit Gittern

Bei Regression geht es darum aus einem Datensatz die zugrunde liegende Funktion annähernd zu rekonstruieren. Das kann dazu genutzt werden, um einen Wert zu schätzen oder vorherzusagen basierend auf gemessenen Werten [FPS96]. Der Datensatz besteht dabei aus Datenpunkten und Zielwerten. Ein Datenpunkt kann zum Beispiel die Temperaturen von sieben aufeinanderfolgenden Tagen enthalten. Der Datenpunkt ist damit ein Vektor mit sieben Dimensionen. Der Zielwert ist dann die Temperatur am achten Tag, die vom Regressionsalgorithmus vorhergesagt werden soll. Man möchte nun eine Funktion haben, die einen Datenpunkt als Eingabe bekommt und den zugehörigen Zielwert als Ergebnis hat. Das Ziel der Regression ist es eine solche Funktion zu finden, deren Ausgabe für alle Datenpunkte im Datensatz möglichst nahe am Zielwert liegt. Formal ausgedrückt ist der Datensatz definiert als

$$S = \{(\vec{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}\}_{i=1}^{m}. \tag{2.1}$$

$\vec{x}_i$ sind die Datenpunkte, y_i die Zielwerte und m ist die Anzahl der Elemente im Datensatz S [Pfl10].

In den folgenden Abschnitten wird zunächst Regression genauer definiert. Dann definieren wir den Funktionsraum, den wir hierfür verwenden werden. Zuerst zeigen wir, wie ein volles Gitter den Funktionsraum mit Basisfunktionen aufspannt. Anschließend zeigen wir, wie Dünngitter dabei helfen Gitterpunkte einzusparen. Zum Schluss wird beschrieben, wie Gitter für Regression verwendet werden können.

2.1 Regression

Bei der Regression wird eine Funktion t gesucht, die die zugrundeliegende Funktion eines Datensatzes S annähernd rekonstruiert. Da die Daten in dem Datensatz im Regelfall verrauscht sind, da sie auf Messungen in der realen Welt basieren, wird nicht nur versucht ein Fehlerterm zu minimieren, sondern auch ein Regularisierungsterm. Die Minimierung des Fehlerterms dient dazu, dass die Funktion t möglichst nahe an den Daten des Datensatzes liegt. Die Minimierung des Regularisierungsterms hingegen dient dazu, dass die Funktion t glatt ist und sich nicht zu sehr an das Rauschen in den Daten anpasst [Pfl10]. Außerdem soll sich die Funktion t dadurch auch nicht allgemein zu sehr an den Datensatz anpassen, da sie sonst nicht vernünftig approximieren kann [FPS96]. Als Fehlerterm wird hierfür gerne der mittlere quadratische Fehler genutzt [Lar05]. Zusammen mit dem Regularisierungsterm ergibt dies nach Pflüger [Pfl10]

$$t = \underset{t \in V}{arg\,min} \left(\frac{1}{m} \sum_{i=1}^{m} (y_i - t(\vec{x}_i))^2 + \lambda C(t) \right). \tag{2.2}$$

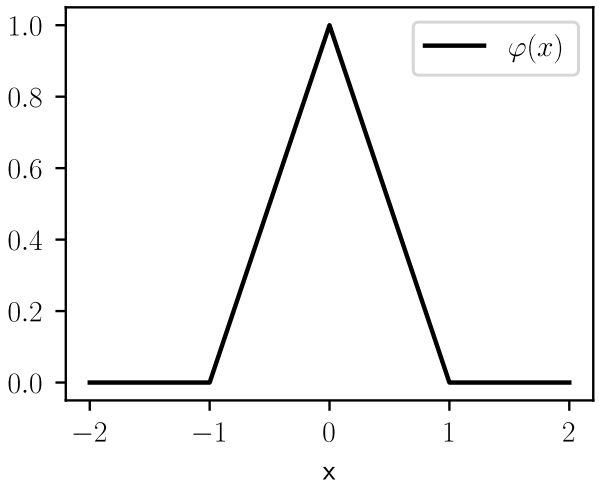

Abbildung 2.1: Hut-Funktion

$C(t)$ ist der Regularisierungsterm, für dessen Wahl es mehrere Möglichkeiten gibt. Eine gute Wahl für Gitter wird später präsentiert. Der Regularisierungsparameter λ bestimmt, wie groß der Einfluss des Regularisierungsterms ist. V ist der Funktionsraum, in dem die Funktion t gefunden werden soll. Gitter spannen mit Basisfunktionen einen Funktionsraum auf, der hierfür verwendet werden kann. Im Folgenden werden Gitter und Basisfunktionen eingeführt.

2.2 Eindimensional

Eine mögliche Wahl für die Basisfunktion ist die in Abbildung 2.1 dargestellte Hut-Funktion. In ihrer Grundform lautet die Formel

$$\varphi(x) = max(1 - |x|, 0). \tag{2.3}$$

Eingesetzt in die folgende Formel, kann die Hut-Funktion dann als Basisfunktion verwendet werden.

$$\varphi_{n,i}(x) = \varphi(2^n x - i) \tag{2.4}$$

mit dem Diskretisierungslevel n und dem Index i. Das Diskretisierungslevel n definiert die Maschenweite $h_n = 2^{-n}$ des Gitters. Der Index i ist aus der Indexmenge

$$I_n = \{i : 0 < i < 2^n\}. \tag{2.5}$$

Hier werden nun volle Gitter eingeführt anhand eines Beispiels, nämlich der Interpolation einer Funktion. Wie Gitter für Regression verwendet werden, wird in Abschnitt 2.6 erläutert.

Möchte man eine Funktion $f : \Omega \to \mathbb{R}$ mit einem vollen Gitter interpolieren, muss man diese Funktion an beliebigen Stellen auswerten können [Pfl10]. Außerdem beschränken wir die Domäne Ω auf ein Teilvolumen von $\mathbb{R}^d$ nämlich $\Omega := [0, 1]^d$. Dies stellt im Allgemeinen keine Einschränkung dar, da man die Domäne der zu interpolierenden Funktion passend skalieren kann [Pfl10].

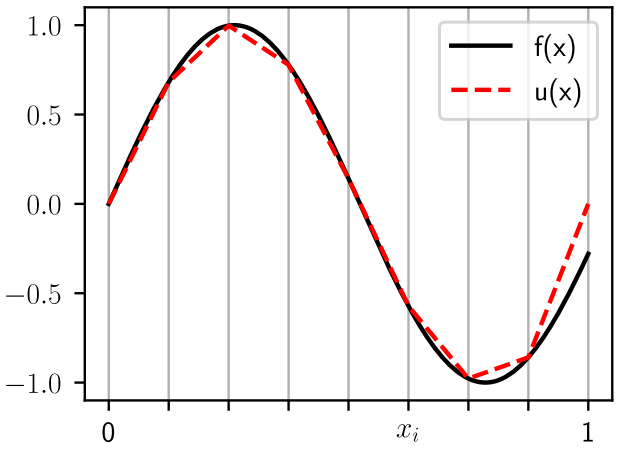

Abbildung 2.2: Interpolation einer eindimensionalen Funktion f mit dem Interpolanten u. Das Diskretisierungslevel beträgt hierbei $n = 3$ und die Gitterpunkte x_i haben einen Abstand von $h_3 = \frac{1}{8}$.

Man wählt ein Diskretisierungslevel n und wertet die Funktion f dann an den Gitterpunkten x_i aus mit einem Abstand von h_n.

Außerdem befindet sich der Gitterpunkt x_i an der Stelle $x_i = i * h_n$. Zu jedem Gitterpunkt gehört genau eine Basisfunktion und hat im Falle der Hut-Funktion genau einen Hut. Damit hat die zu einem Gitterpunkt gehörende Basisfunktion nur in einem beschränkten Bereich um den Gitterpunkt herum einen Funktionswert ungleich null. Durch Linearkombination von Basisfunktionen mit Koeffizienten α_i spannt das Gitter einen Funktionsraum auf. Aus diesem Funktionsraum kann ein Interpolant u für die Funktion f gewählt werden. Der Interpolant u wird durch folgende Funktion beschrieben

$$f(x) \approx u(x) := \sum_j \alpha_j \varphi_j(x). \tag{2.6}$$

Abbildung 2.2 zeigt eine Funktion f, die durch den Interpolaten u angenähert wird. Mit einer Funktion u aus dem Funktionsraum eines Gitters lässt sich dann zum Beispiel die Funktion f aus Abbildung 2.2 interpolieren, so wie in Abbildung 2.3 veranschaulicht.

2.3 Mehrdimensional

Eine eindimensionale Basisfunktion lässt sich über ein Tensor Produkt auf den mehrdimensionalen Fall erweitern [Pfl10]. Für Gleichung (2.4) ergibt dies

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

mit d Dimensionen und dem mehrdimensionalen Index $\vec{i}$. Abbildung 2.4 zeigt wie die Hut-Funktion mit zwei-dimensionaler Domäne aussieht.

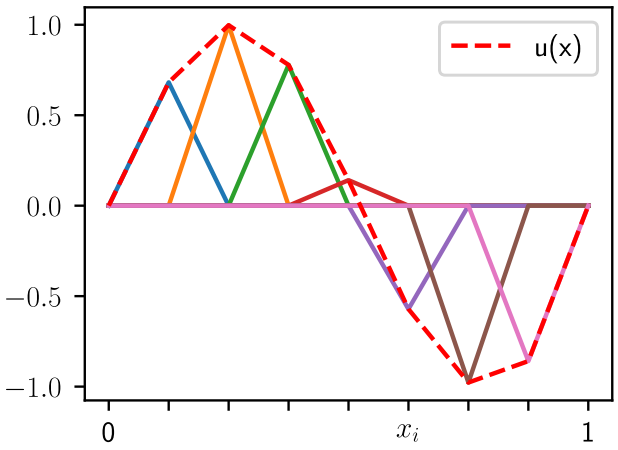

Abbildung 2.3: Linearkombination von Hut-Funktionen als Interpolant

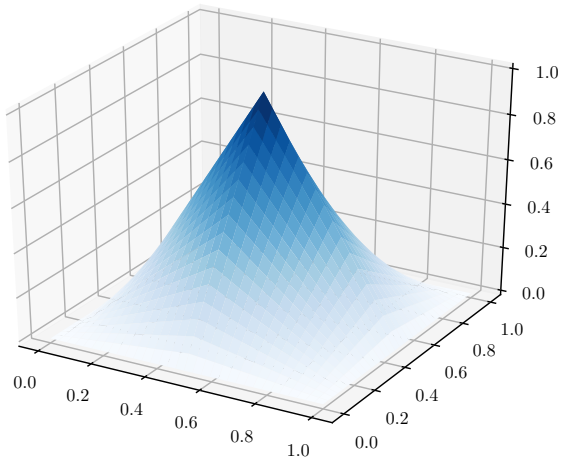

Abbildung 2.4: Hut-Funktion mit zwei-dimensionaler Domäne

Die Formel für den Interpolant u lässt sich damit ebenfalls auf eine mehrdimensionale Domäne erweitern.

$$u(\vec{x}) = \sum_{\vec{i} \in I_n} \alpha_{\vec{i}} \varphi_{n,\vec{i}}(\vec{x}) \tag{2.8}$$

Die Indexmenge I_n ist dabei definiert als

$$I_n = \{\vec{i} : 0 < i_j < 2^n, 1 \leq j \leq d\} \tag{2.9}$$

im d-dimensionalen Fall. Damit liegt die Anzahl der Gitterpunkte, an denen die zu interpolierende Funktion ausgewertet werden muss, im Bereich von

$$O(2^{nd}). \tag{2.10}$$

Aufgrund der exponentiell steigenden Komplexität sind volle Gitter nur für Probleme mit maximal 4 Dimensionen und angemessenem n praktisch nutzbar [GG01].

Im Folgenden zeigen wir zuerst, wie ein Gitter in hierarchischer Basis dargestellt werden kann. Anschließend zeigen wir, wie einige der Gitterpunkte weggelassen werden können, um die asymptotische Komplexität zu verringern.

2.4 Volle Gitter in hierarchischer Basis

Basisfunktionen wie die Hut-Funktion lassen sich über Skalierung und Verschiebung auch als hierarchische Basisfunktion verwenden. Dies ergibt im eindimensionalen [Pfl10]

$$\varphi_{l,i}(x) = \varphi(2^l x - i). \tag{2.11}$$

Dies ist die gleiche Formel wie in Gleichung (2.4), nur das das feste Diskretisierungslevel n durch das variable Level l ersetzt wurde. Die Menge der Indizes i ist dabei [Pfl10]

$$I_l := \{i \in \mathbb{N} : 0 < i < 2^l, i \; ungerade\}. \tag{2.12}$$

In Abbildung 2.5 ist veranschaulicht, wie die Hut-Funktion als hierarchische Basisfunktion aussieht in Level 1 bis 3. Je höher das Level, desto feiner die Diskretisierung. Da hier nur Basisfunktionen mit ungeradem Index verwendet werden, überlappen sich die Träger der Hut-Funktionen auf demselben Level nicht. Würde man nur Basisfunktionen mit demselben Level verwenden, um einen Funktionsraum aufzuspannen, hätten Funktionen aus diesem Funktionsraum an Gitterpunkten mit geradem Index immer einen Funktionswert von 0. Daher werden auch die Basisfunktionen von niedrigeren Levels benötigt. Die hierarchischen Unterräume W_l, die für die unterschiedlichen Level l aufgespannt werden, sind definiert als [Pfl10]

$$W_l := \left\langle \varphi_{l,i}(x) : i \in I_l \right\rangle. \tag{2.13}$$

Wie die Interpolation einer eindimensionalen Funktion mit hierarchischer Basisfunktion aussieht, ist in Abbildung 2.6 dargestellt. Der Funktionsraum, der von einem vollen Gitter aufgespannt wird mit hierarchischen Basisfunktionen und einer Maschenweite von h_n, lässt sich als direkte Summe der hierarchischen Unterräume W_l darstellen [Pfl10].

$$V_n = \bigoplus_{l \leq n} W_l \tag{2.14}$$

Wie in Gleichung (2.7) geschehen, lässt sich auch Gleichung (2.11) als mehrdimensionale Basisfunktion verwenden mit

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j) \tag{2.15}$$

wie in [Pfl10] beschrieben. Mit d Dimensionen und dem mehrdimensionalen Level $\vec{l}$ und Index $\vec{i}$. Die Index-Menge ist nun [Pfl10]

$$I_{\vec{l}} := \{\vec{i} : 0 < i_j < 2^{l_j}, i_j \; ungerade, 1 \leq j \leq d\}. \tag{2.16}$$

Analog werden auch die Unterräume $W_{\vec{l}}$ erweitert [Pfl10]

$$W_{\vec{l}} := \left\langle \varphi_{\vec{l},\vec{i}}(\vec{x}) : \vec{i} \in I_{\vec{l}} \right\rangle, \tag{2.17}$$

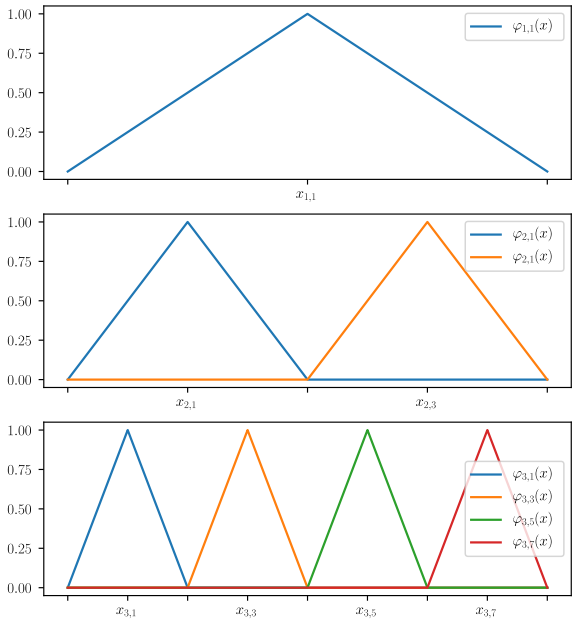

Abbildung 2.5: Eindimensionale Hut-Funktionen $\varphi_{l,i}$ mit zugehörigen Gitterpunkten $x_{l,i}$ als hierarchische Basis [Pfl10].

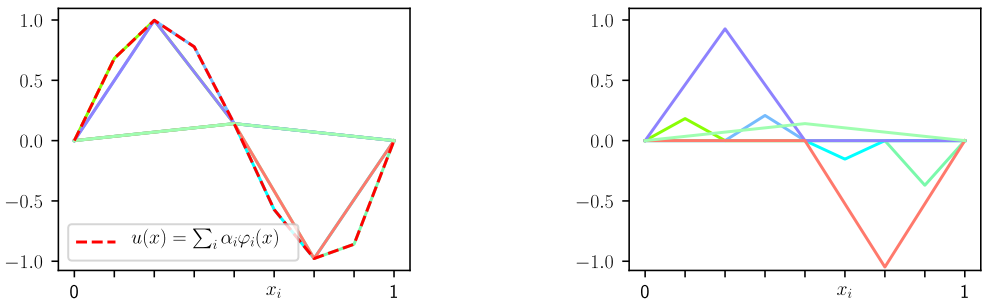

Abbildung 2.6: Interpolation der gleichen Funktion wie in Abbildung 2.3, dieses Mal mit hierarchischen Basisfunktionen (links) und die einzelnen skalierten Basisfunktionen, die der Interpolant verwendet (rechts).

ebenso der Funktionsraum, der durch das mehrdimensionale volle Gitter aufgespannt wird [Pfl10]

$$V_n = \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}}. \tag{2.18}$$

Die hier verwendete Maximum-Norm ist definiert als

$$|\vec{x}|_\infty := max(|x_1|, ..., |x_n|). \tag{2.19}$$

In Abbildung 2.7 sind die Basisfunktionen visualisiert, die in den Unterräumen $W_{\vec{l}}$ des Funktionsraumes V_3 verwendet werden in zwei Dimensionen. Der Interpolant u wird aus dem Funktionsraum V_n gewählt und hat dadurch die Form [Pfl10]

$$u(\vec{x}) = \sum_{|\vec{l}|_\infty \leq n, \vec{i} \in I_{\vec{l}}} = \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.20}$$

Die Anzahl der Gitterpunkte liegt auch mit hierarchischer Basisfunktion in

$$O(h_n^{-d}) = O(2^{nd}). \tag{2.21}$$

2.5 Dünngitter

Bei Dünngittern werden die hierarchischen Basisfunktionen verwendet, die im vorherigen Abschnitt beschrieben wurden. Durch die hierarchische Abdeckung der Domäne können Gitterpunkte weggelassen werden, die nur wenig zum Gesamtergebnis beitragen [Pfl10]. Das hilft bei der Reduzierung der Anzahl der nötigen Gitterpunkte und damit auch der Reduzierung der Rechenleistung und des benötigten Speicherplatzes.

Man verwendet nur Unterräume $W_{\vec{l}}$, für die $|\vec{l}|_1 \leq n + d - 1$ gilt [Pfl10]. Daraus folgt der Dünngitter Funktionsraum [Pfl10]

$$V_n^{(1)} := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}. \tag{2.22}$$

Die hier verwendete l_1-Norm ist wie folgt definiert

$$|\vec{l}|_1 := \sum_{j=1}^{d} l_j. \tag{2.23}$$

Welche Unterräume $W_{\vec{l}}$ im Funktionsraum $V_3^{(1)}$ verwendet werden ist in Abbildung 2.8 dargestellt.

Der Dünngitter-Interpolant u mit mehrdimensionaler hierarchischer Basisfunktion lässt sich folgendermaßen aufstellen

$$u(\vec{x}) = \sum_{|\vec{l}|_1 \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}) \tag{2.24}$$

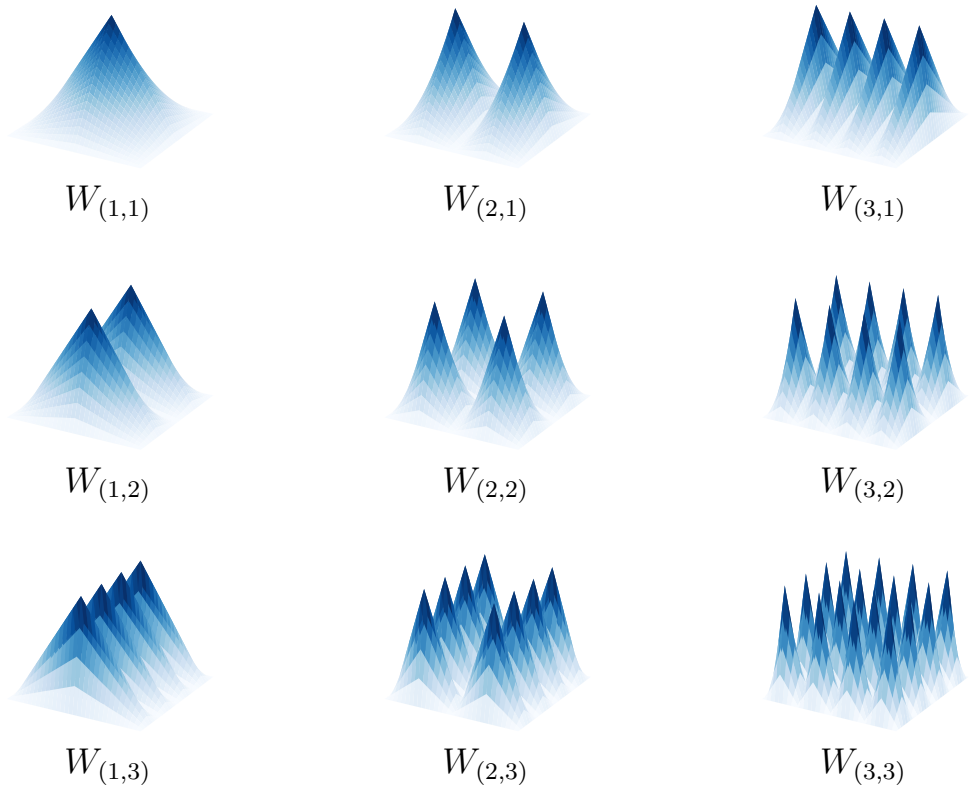

Abbildung 2.7: Basisfunktionen der Unterräume $W_{\vec{l}}$ des Funktionsraumes V_3 in zwei Dimensionen [Pfl10]

mit Indexmenge

$$I_{\vec{l}} := \{\vec{i} : 0 < i_j < 2^{l_j}, i_j \, ungerade, 1 \leq j \leq d\}. \tag{2.25}$$

In Abbildung 2.9 sind die Gitterpunkte von zwei- und drei-dimensionalen Dünngittern dargestellt. Während die vollen Gitter noch eine Anzahl von Gitterpunkten in der Größenordnung von $O(2^{nd})$ hatten, ist es bei Dünngittern nur noch

$$O(h_n^{-1}(log(h_n^{-1}))^{d-1}) = O(2^n n^{d-1}) \tag{2.26}$$

wie in [GG01] beschrieben. Zur Erinnerung h_n ist die Maschenweite, also der Abstand zwischen den Gitterpunkten im Fall eines vollen Gitters, n das Diskretisierungslevel und d die Anzahl an Dimensionen.

In Tabelle 2.1 ist nochmals die Anzahl der Gitterpunkte in vollen Gittern und Dünngittern aufgelistet. Außerdem ist jeweils der asymptotische Fehler angegeben, der bei Dünngittern etwas schlechter ist als bei vollen Gittern.

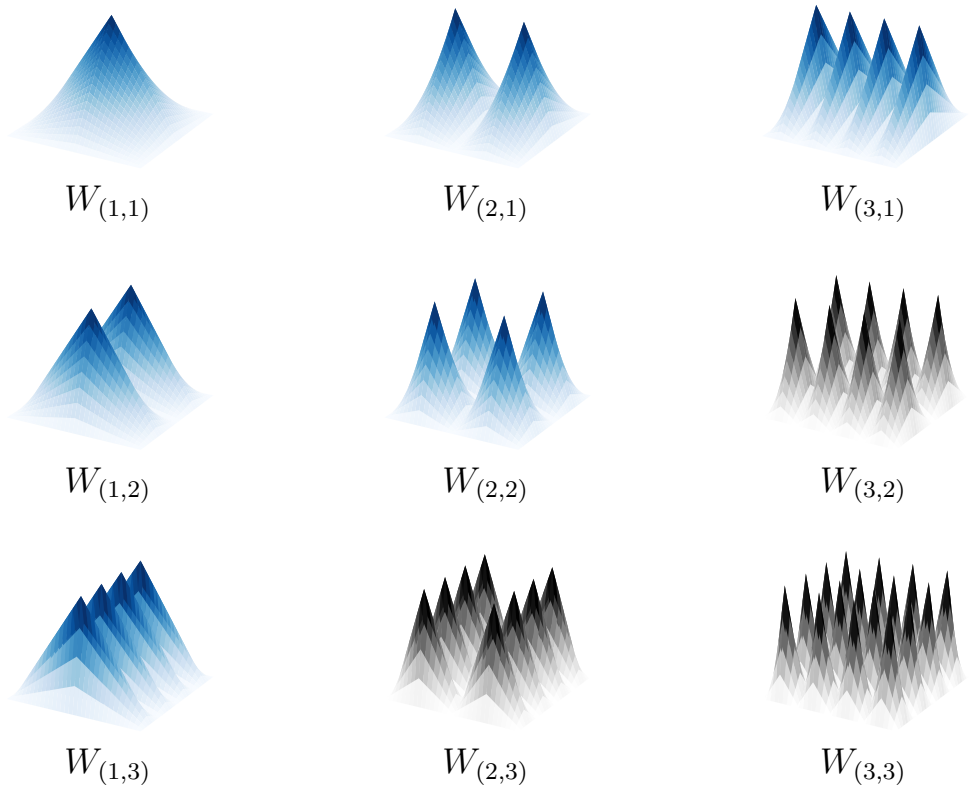

Abbildung 2.8: Basisfunktionen der Unterräume $W_{\vec{l}}$ des Funktionsraumes $V_3^{(1)}$ (blau). Basisfunktionen der Unterräume $W_{\vec{l}}$ des Funktionsraumes V_3, die nicht in $V_3^{(1)}$ verwendet werden (grau) [Pfl10].

	Gitterpunkte	Fehler
Volle Gitter	$O(h_n^{-d})$	$O(h_n^2)$
Dünngitter	$O(h_n^{-1}(log(h_n^{-1}))^{d-1})$	$O(h_n^2(log(h_n^{-1}))^{d-1})$

Tabelle 2.1: Vergleich der Komplexität von vollen Gittern und Dünngittern [Pfl10]

2.6 Regression mit Dünngittern

Dünngitter sind gut für Regression mit großen Datensätzen geeignet, da die Komplexität linear mit der Anzahl der Datenpunkte steigt. Außerdem verwenden wir keine vollen Gitter, da diese nur für Probleme mit maximal etwa 4 Dimensionen einsetzbar sind [GG01]. Um Regression effizient mit Gittern durchzuführen, verwenden wir als Regularisierungsterm die euklidische Norm über den Vektor der Koeffizienten wie von Pflüger [Pfl10] beschrieben

$$C(t) := ||\vec{\alpha}||_2^2 . \tag{2.27}$$

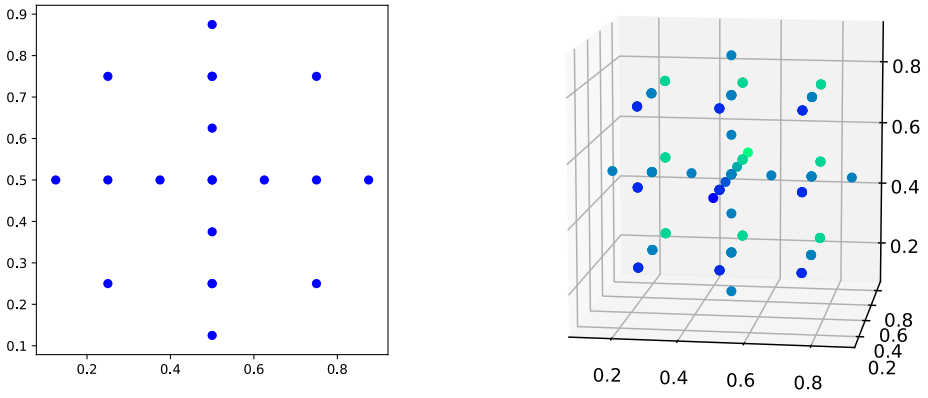

Abbildung 2.9: Gitterpunkte eines zwei-dimensionalen (links) und drei-dimensionalen (rechts) Dünngitters.

Eingesetzt in Gleichung (2.2) ergibt dies das Gleichungssystem [Pfl10]

$$\left(\frac{1}{m}BB^T + \lambda I\right)\vec{\alpha} = \frac{1}{m}B\vec{y}. \tag{2.28}$$

Die Elemente der Matrix B sind dabei definiert als

$$(B)_{i,j} = \varphi_i(\vec{x}_j). \tag{2.29}$$

$\vec{x}_j$ sind die Datenpunkte und $\vec{y}$ enthält die Zielwerte y_i aus dem Datensatz. I ist die Einheitsmatrix.

Dieses Gleichungssystem kann dann mit einem numerischen Verfahren wie zum Beispiel dem CG-Verfahren gelöst werden [Pfl10], um die Koeffizienten $\vec{\alpha}$ zu erhalten. Diese Koeffizienten können dann in die jeweilige Gitter-Gleichung eingesetzt werden, also in Gleichung (2.6) oder Gleichung (2.24), je nachdem welche Art von Gitter beim Aufstellen des Gleichungssystems verwendet wurde um die Regressionsfunktion zu erhalten.

3 HPX

Für die verteilte Ausführung von Programmen auf Clustern und Supercomputern wird heutzutage oftmals das *Message Passing Interface (MPI)* verwendet. Bei diesem kommunizieren sequenzielle Prozesse miteinander über Nachrichten. Dieses Ausführungsmodell hat allerdings Probleme bestimmte Klassen von Anwendungen zu skalieren. Selbst auf tausend Prozessorkerne zu skalieren ist dabei oft nicht möglich, selbst bei einer Ausführungsdauer von mehreren Wochen. Zu diesen Anwendungen gehören zum Beispiel die Simulation von Gammablitzen und die Erkennung von Gravitationswellen mit dem *Laser Interferometer Gravitational Oberservatory (LIGO)* [KBS09].

Das ParalleX Modell, das von dem Laufzeitsystem High Performance ParraleX (HPX) umgesetzt wird, versucht auch solche Anwendungen besser skalierbar zu machen. Dabei verwendet es aktive Nachrichten für eine Nachrichten-gesteuerte Ausführung, die dynamisch und asynchron arbeitet [KBS09]. Die Ziele von HPX sind: Skalierbarkeit, auch für Supercomputer mit einer Rechenleistung im Exaflop Bereich; Programmierbarkeit, um die Entwicklung für parallele und verteilte Programme zu erleichtern; Minimierung des Aufwands bei der Portierung auf neue Plattformen; Robustheit gegenüber von Ausfällen; Energieeffizienz durch effiziente Ausnutzung der Ressourcen [KHA+14].

3.1 Architekur

Da HPX ein Laufzeitsystem ist, wird es mit der Anwendung gestartet und mit dieser auch wieder beendet. Ein Laufzeitsystem hat die Aufgabe, die Ausführung einer Anwendung zu verbessern, zum Beispiel durch Verbesserung der Leistung, der Energieeffizienz oder der Skalierbarkeit. Dabei baut es auf den Diensten des Betriebssystems auf [KHA+14].

Im Folgenden werden die wichtigsten Bestandteile der Architektur von HPX beschrieben.

3.1.1 Das *Parcel Subsystem*

Das *Parcel Subsystem* ist für die gesamte Kommunikation in HPX zuständig. Ein *Parcel* ist eine Art von aktiver Nachricht und entspricht einem entfernten Funktionsaufruf. Es enthält das Ziel, eine Referenz auf eine Aufzurufende Funktion, die Argumente für diese Funktion und optional Informationen, wie fortgefahren werden soll nach Beendigung der Funktion. Die Übertragung erfolgt immer asynchron und alle Daten werden als Argumente für die Funktion mitgeliefert. *Parcles* werden zwischen *Localities* verschickt, die im Normalfall einem Rechenknoten entsprechen. Auf der *Locality*, auf der ein *Parcel* empfangen wird, wird die referenzierte Funktion ausgeführt mit den enthaltenen Argumenten und schließlich wird optional das Ergebnis zurückgeschickt und/oder andere Teile des Programms über die Fertigstellung informiert [KHA+14].

3.1.2 Das *Active Global Address Space (AGAS)*

Das AGAS stellte einen globalen Adressraum bereit, der sämtliche *Localities* umspannt. Ein Objekt im AGAS hat immer dieselbe Adresse, selbst wenn es auf eine andere *Locality* verschoben wird. Dafür passt sich das AGAS während der Laufzeit dynamisch an [KHA+14].

3.1.3 HPX-Threads

Wird ein *Parcel* empfangen, wird es in ein HPX-Thread konvertiert und vom *Threading Subsystem* verwaltet. Die Anzahl der HPX-Threads, die ausgeführt werden ist in der Regel größer als die Anzahl der OS-Threads. Pro Prozessorkern wird typischerweise ein OS-thread verwendet. Die HPX-Threads werden auf die OS-Threads verteilt und abwechselnd ausgeführt. Dies geschieht sehr effizient, da bei einem Kontextwechsel von HPX-Threads kein Kernel-Aufruf nötig ist. Ziel ist es, dass dadurch Millionen von Threads pro Sekunde auf einem Prozessorkern ausgeführt werden können. HPX-Threads werden zwar nie vom Scheduler pausiert, aber sie können sich selbst freiwillig pausieren während sie zum Beispiel auf Daten oder Synchronisation warten, damit währenddessen andere HPX-Threads ausgeführt werden können [KHA+14].

3.1.4 *Local Control Objects (LCOs)*

LCOs werden zur Kontrolle der Parallelisierung und Synchronisation verwendet. Außerdem helfen sie beim Verhindern von Verzögerungen. Jedes Objekt, dass einen HPX-Thread erzeugt oder einen pausierten HPX-Thread fortführt, fungiert als LCO. Sie helfen bei der Ereignisgesteuerten Ausführung und ersetzen dadurch globale Synchronisation. Statt das darauf gewartet werden muss, bis alle Threads an einer bestimmten Stelle in der Ausführung sind, wartet jeder Thread nur darauf, dass die Ergebnisse berechnet oder übertragen wurden, die es selbst benötigt um fortzufahren. Das ganze geschieht ohne aktives Blockieren oder Warten. Wie von Kaiser et al. [KHA+14] beschrieben zählen zu den LCOs unter anderem:

- *Futures* stellen Ergebnisse dar, die noch nicht bekannt sind, weil sie noch nicht übertragen oder berechnet wurden. Wenn ein HPX-Thread auf ein Future wartet, wird er pausiert, bis das Ergebnis zur Verfügung steht und dann automatisch fortgesetzt.
- *Dataflow Objects* helfen Abhängigkeiten zu managen ohne globale Synchronisation. Sie warten auf bestimmte *Futures* und führen dann eine vordefinierte Funktion aus mit den Ergebnissen der *Futures*.
- Herkömmliche Mechanismen zur Kontrolle von nebenläufiger Ausführung wie Mutexes, Semaphores, Spinlocks und Barrieren.

R f(p...)	Synchrone Ausführung (gibt R zurück)	Asynchrone Ausführung (gibt future<R> zurück)	*Fire and Forget* Ausführung (gibt void zurück)
Funktionen (direkter Aufruf)	f(p...)	async(f, p...)	apply(f, p...)
Funktionen (lazy Aufruf)	bind(f, p...)(...)	async(bind(f, p...), ...)	apply(bind(f, p...), ...)
Aktionen (direkter Aufruf)	HPX_ACTION(f, action) a(id, p...)	HPX_ACTION(f, action) async(a, id, p...)	HPX_ACTION(f, action) apply(a, id, p...)
Aktionen (lazy Aufruf)	HPX_ACTION(f, action) bind(a, id, p...)(...)	HPX_ACTION(f, action) async(bind(a, id, p...), ...)	HPX_ACTION(f, action) apply(bind(a, id, p...), ...)

Tabelle 3.1: Arten eine Funktion mit HPX aufzurufen. Der dunkelgraue Bereich enthält die von C++ definierte Art eine Funktion aufzurufen. Der mittelgraue Bereich enthält die in der C++ Standard Library definierten Arten eine Funktion aufzurufen. Der dunkelgraue Bereich enthält die von HPX hinzugefügten Arten eine Funktion aufzurufen [KHA+14].

3.2 API

Die HPX API hält sich so gut wie möglich an den C++ Standard. Alle Schnittstellen, die im C++ Standard definiert sind bezüglich Multi-Threading, wie `future`, `thread`, `mutex` und `async` sind implementiert. Darauf aufbauend wurden diese Schnittstellen erweitert für die verteilte Ausführung von Anwendungen [KHA+14].

Jede Operation, die mit dem Netzwerk arbeitet, gibt ein `future` zurück. Dadurch kann andere Arbeit verrichtet werden, solange auf das Netzwerk gewartet wird. Wartet man auf das `future` wird der HPX-Thread pausiert, bis das `future` bereit ist. In diesem Fall also bis der Netzwerkzugriff beendet ist, beziehungsweise bis das Ergebnis verfügbar ist. Es können aber auch andere Operationen an das `future` angehängt werden, die ausgeführt werden, sobald das `future` bereit ist [KHA+14].

Die API bietet drei verschiedene Arten Funktionen auszuführen. Diese können entweder lokal oder entfernt aufgerufen werden. In Tabelle 3.1 sind Möglichkeiten dargestellt, wie eine Funktion mit C++ und HPX aufgerufen werden kann. Kaiser et al. [KHA+14] beschreiben diese Arten als:

- Synchroner Funktionsaufruf: die herkömmliche Art eine Funktion aufzurufen. Der Aufrufende Thread wartet, bis die Funktion ihr Ergebnis zurückgibt. In HPX wird dieser HPX-Thread pausiert, bis die Funktion fertig ist, damit in der Zeit andere HPX-Threads arbeiten können.
- Asynchroner Funktionsaufruf: ein neuer HPX-Thread wird gestartet, in welchem die Funktion ausgeführt wird. Es wird sofort ein `future` zurückgegeben, das zur Synchronisation genutzt werden kann oder an das weitere Operationen angehängt werden können.
- *Fire and Forget* Funktionsaufruf: wie ein asynchroner Funktionsaufruf, nur das nichts zurückgegeben wird, mit dem man synchronisieren kann. Die Funktion wird in einem neuen HPX-Thread ausgeführt und ein möglicher Rückgabewert verworfen.

4 Parallele Regression auf einem Rechenknoten

Da die Ausführung der Regression auf einem einzelnen Kern unserer Erfahrung nach schon bei Datensätzen und Gittern moderater Größe mehrere Stunden dauern kann, ist es sinnvoll die Arbeit auf alle Kerne aufzuteilen. In diesem Kapitel geht es darum, wie die Regression mit Dünngittern parallel auf einem Rechenknoten ausgeführt werden kann. Zunächst wird beschrieben, wie die Berechnungen aufgeteilt werden können und unabhängig voneinander bearbeitet werden können. Anschließend geht es um die Nutzung von HPX und OpenCL für die parallel Ausführung.

4.1 Parallelisierung von Dünngittern

Wir wollen das in Abschnitt 2.6 beschriebene Gleichungssystem

$$\left(\frac{1}{m}BB^T + \lambda I\right)\vec{\alpha} = \frac{1}{m}B\vec{y}. \tag{4.1}$$

parallel mit dem CG-Verfahren lösen. Dabei werden pro Iteration drei Matrix-Vector Multiplikationen ausgeführt [Pfl10]. Diese benötigen unseren Tests nach fast 100% der gesamten Laufzeit, daher haben wir uns nur auf deren Parallelisierung konzentriert. Die Matrix B als auch ihre transponierte Matrix B^T müssen mit Vektoren multipliziert werden. Die Elemente von B sind dabei definiert als

$$(B)_{i,j} = \varphi_i(\vec{x}_j). \tag{4.2}$$

Wenn man $\vec{b} = B^T\vec{\alpha}$ parallel berechnen will, kann man jede Zeile der Matrix B^T getrennt voneinander mit $\vec{\alpha}$ multiplizieren und anschließen die Ergebnisse in den Ergebnisvektor $\vec{b}$ eintragen. Für das i-te Element in $\vec{b}$ berechnet man

$$b_i = \sum_{j=1}^{N} \alpha_j \varphi_j(\vec{x}_i). \tag{4.3}$$

N ist hierbei die Anzahl der Gitterpunkte und damit auch die Anzahl der Basisfunktionen. Alle b_i lassen sich gleichzeitig parallel berechnen, da sie völlig unabhängig voneinander sind. i liegt dabei im Bereich $1, ..., m$. m ist dabei die Anzahl der Datenpunkte x_i.

Wenn man $\vec{b} = B\vec{z}$ parallel berechnen will, kann man dies wieder für jeden Datenpunkt x_i getrennt machen, mit dem Unterschied, dass man zum Schluss alle Zwischenergebnisse $\vec{b}^{(i)}$ noch aufsummieren muss.

$$\vec{b}^{(i)} = \sum_{j=1}^{N} z_i \varphi_j(\vec{x}_i) \tag{4.4}$$

$$\vec{b} = \sum_{i=0}^{m} \vec{b}^{(i)} \tag{4.5}$$

Listing 4.1 Beispiel wie die Matrix-Vektor Multiplikation mit HPX parallel ausgeführt werden kann in Pseudo-C++ Code.

```
1   std::vector<hpx::future<Result>> resultFutures;
2
3   for (int i = 0; i < numOfThreads; i++) {
4     resultFutures.push_back(hpx::async(multFunction, params));
5   }
6
7   Result combinedResult;
8
9   for (int i = 0; i < numOfThreads; i++) {
10    combinedResult += resultFutures[i].get();
11  }
12
```

Die Berechnung der Zwischenergebnisse $\vec{b}^{(i)}$ kann parallel geschehen, aber bei der Aufsummierung der Zwischenergebnisse muss man aufpassen, dass es zu keiner *Race Condition* kommt [Pfl10].

4.2 Parallele Ausführung mit HPX

In Listing 4.1 ist ein Beispiel, wie man die Matrix-Vektor Multiplikation mit HPX parallel ausführen kann. Dabei ist vorausgesetzt, dass es eine Funktion `multFunction` gibt, die die Matrix-Vektor Multiplikation auf einem Teil des Datensatzes ausführt und das Ergebnis vom Typ `Result` zurückgibt.

In Zeile 4 wird `numOfThreads`-Mal die `multFunction` asynchron aufgerufen mit `params` als Parameter. `numOfThreads` ist dabei die Anzahl der Threads, die man nutzen möchte und `params` enthält die Parameter für `multFunction`. Zu den Parametern gehören der Teil des Datensatzes, der von diesem Thread verarbeitet werden soll, der zugehörige Teil des Vektors, mit dem die Matrix B multipliziert werden soll und Informationen über die Gitterpunkte, also die jeweiligen Level $\vec{l}$ und Indizes $\vec{i}$. Von `hpx::async` wird dann ein *Future* zurückgegeben, welches die Rückgabe der `multFunction` enthält, sobald diese berechnet wurde.

In Zeile 10 wird die `get()` Funktion auf den `Futures` aufgerufen, um auf die Ergebnisse zu warten und diese dann aufzusummieren. Dies wird synchron ausgeführt, um *Race Conditions* zu verhindern.

4.3 OpenCL

OpenCL ist ein offener Standard für parallele Berechnung auf heterogenen Plattformen mit verschiedenen Prozessoren wie CPUs, GPUs und DSPs [SGS10]. OpenCL bietet eine einheitliche Programmiersprache, Schnittstellen und Abstraktionen um das Arbeiteten auf verschiedenen Typen von Prozessoren von verschiedenen Herstellern so einheitlich wie möglich zu gestalten [SGS10].

Bei OpenCL werden *Kernel* genannte Funktionen kompiliert und auf dem ausgewählten *Device* ausgeführt. Das *Device* kann dabei die CPU des Systems sein oder ein anderes angeschlossenes Gerät, dass OpenCL unterstützt. Der *Kernel* wird dabei mehrfach aufgerufen und er erfährt über einen Index, auf welchem Teil der Daten er arbeiten soll [SGS10].

In der Dünngitter Bibliothek SG++, die wir für diese Arbeit mit HPX erweitert haben, waren bereits OpenCL Kernel enthalten. Da diese bereits parallel arbeiten, wurde HPX hier nicht zur Parallelisierung genutzt, sondern nur für die verteilte Ausführung, die im nächsten Kapitel beschrieben wird.

5 Verteilte Regression mit HPX

Zur verteilten Ausführung der Regression mit HPX auf mehreren Rechenknoten verwenden wir mehrere Abstraktionen, die von HPX zur Verfügung gestellt werden. Dazu gehören neben den schon erwähnten *Futures* auch *Components*, *Actions* und *Channels*. *Components* sind C++ Klassen, von denen Instanzen auf entfernten Rechenknoten erstellt werden können und deren Funktionen auch entfernt ausgeführt werden können [HPX18c]. *Actions* sind Funktionen, die auch entfernt ausgeführt werden können und sind damit für die *Components* wichtig [HPX18a]. *Channels* bieten Kommunikation und Synchronisation. Über sie können Werte verschickt werden, zu anderen Threads oder auch zu anderen Rechenknoten. Sie können auch zur Synchronisation verwendet werden, da der Empfänger auf die Werte warten kann um erst fortzufahren, wenn ein Wert empfangen wurde [HPX18b].

Die Matrix-Vektor Multiplikation lässt sich verteilt ausführen, indem auf jedem Rechenknoten der Vektor y nur mit einem Teil der Matrix B multipliziert wird und die Teilergebnisse dann zusammengefasst werden. In Abbildung 5.1 ist dies veranschaulicht. Die Teilmatrizen $B^{(i)}$ werden auf den Rechenknoten mit jeweils einem Teil des Datensatzes berechnet und anschließend mit dem Vektor y multipliziert. Zum Schluss werden sie zu dem Vektor z aufsummiert. Analog wird so auch die Multiplikation mit der transponierten Matrix ausgeführt.

In Listing 5.1 wird beispielhaft gezeigt, wie wir mit HPX die Matrix-Vektor Multiplikation auf mehreren Rechenknoten verteilt berechnen. In Zeile 4 wird auf jedem Rechenknoten eine Component-Instanz von der Klasse `Comp` erzeugt. Diese Instanz wird jeweils auf der *Locality* `loc` erzeugt mit `params` als Parameter für den Konstruktor. Zu den Parametern gehören zum Beispiel der Pfad zu der Datei mit dem Datensatz und Informationen, welcher Teil des Datensatzes von diesem Rechenknoten verwendet werden soll. In Zeile 10 wird definiert, welche *Action* wir auf der Instanz aufrufen möchten. In Zeile 11 wird ein *Channel* erstellt, der Daten vom Typ `Result` verschicken und empfangen kann. In Zeile 13 wird die *Action* `Comp::MultAction` auf der Instanz `inst` ausgeführt nach

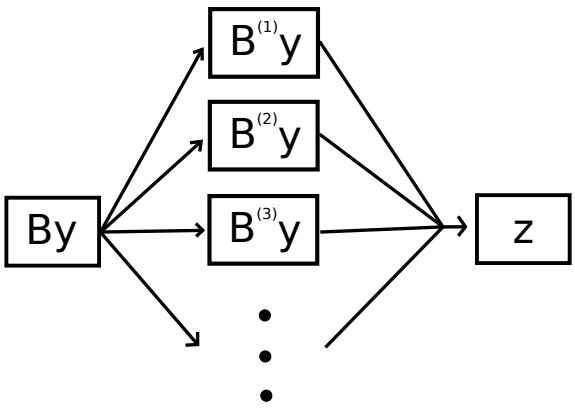

Abbildung 5.1: Verteilte Berechnung der Matrix-Vektor Multiplikation.

Listing 5.1 Beispiel wie die Matrix-Vektor Multiplikation mit HPX verteilt ausgeführt werden kann in Pseudo-C++ Code.

```
1  std::vector<hpx::id_type> instances;
2
3  for (auto loc : hpx::find_all_localities()) {
4    instances.push_back(hpx::new_<Comp>(loc, params).get());
5  }
6
7  std::vector<hpx::lcos::channel<Result>> channels;
8
9  for (auto inst : instances) {
10   Comp::MultAction act;
11   hpx::lcos::channel<Result> channel(hpx::find_here());
12   channels.push_back(channel);
13   hpx::apply(act, inst, channel, params);
14 }
15
16 Result combinedResult;
17
18 for (auto channel : channels) {
19   for (auto result : channel) {
20     combinedResult += result;
21   }
22 }
23
```

dem *Fire and Forget* Prinzip. Die *Action* bekommt einen *Channel* und weitere Parameter. In den Zeilen 19-21 wird über alle Ergebnisse iteriert, die über die *Channel* empfangen werden und zum Endergebnis aufsummiert.

Die gleiche Funktionalität wäre auch mit MPI machbar gewesen, aber deutlich aufwendiger und komplizierter. Die Abstraktionen von HPX ermöglichen es einfacheren und übersichtlicheren Code zu schreiben, der dadurch auch weniger fehleranfällig ist.

6 Experimente

Wir haben die Dünngitter Bibliothek SG++ mit HPX erweitert, um Dünngitter-Regression auch auf verteilt auf einem Cluster ausführen zu können. In diesem Kapitel geht es um Experimente, die wir ausgeführt haben, um einerseits zu prüfen wie gut unsere Implementierung mit der Anzahl der Rechenknoten skaliert und andererseits um die Ergebnisse der Dünngitter-Regression mit anderen Regressions-Algorithmen zu vergleichen. Der Vergleich mit den anderen Algorithmen dient auch dazu zu prüfen, ob SG++ nach unseren Erweiterungen immer noch korrekte Ergebnisse berechnet.

6.1 Skalierung

Zum testen des Skalierungsverhaltens haben wir synthetische Datensätze generiert. Die Datenpunkte waren Vektoren mit jeweils 100 zufälligen Werten zwischen 0 und 1. Die Zielwerte waren jeweils der geometrische Abstand des mehrdimensionalen Datenpunktes zum Mittelpunkt der Domäne $\Omega := [0, 1]^d$.

Der erste Test war auf dem SGSCL1 Cluster, der aus 16 Rechenknoten besteht mit jeweils einem Intel® Xeon™ E3-1585 v5 und 32GB RAM. Für diesen Test bestand der Datensatz aus 100.000 Datenpunkte und hatte 100 Dimensionen. Das Testprogramm `RegressionRefinementExample`[1] wurde dafür mit den Parametern `--dim 100 --num_refine 4 --no_points 2` ausgeführt. Für den Test mit OpenCL wurde zusätzlich der Parameter `--ocl detectedPlatform.cfg` hinzugefügt um OpenCL zu aktivieren und die Konfigurationsdatei anzugeben. In Abbildung 6.1 ist das Skalierungsverhalten zu sehen, einmal für die Ausführung von reinem C++ und einmal mit OpenCL. Für die reine C++ Ausführung benötigt ein Rechenknoten knapp 32 Minuten und auf 16 Rechenknoten etwas über 2 Minuten. Mit den genauen Zeiten ergibt das einen Skalierungsfaktor von etwa 15,1 und eine parallele Effizienz von etwa 94,4%. Damit liegt sie nahe am theoretischen Optimum. Bei der Ausführung mit OpenCL benötigt ein Rechenknoten knapp 24 Minuten und auf 16 Knoten 2 Minuten. Das ergibt einen Skalierungsfaktor von etwa 12 und eine parallele Effizienz von etwa 74,8%. Das etwas schlechtere Skalierungsverhalten ist mit dem zusätzlichen Overhead von OpenCL zu erklären.

Der zweite Test war auf dem Supercomputer Cori vom National Energy Research Scientific Computing Center. Wir haben die Haswell Rechenknoten genutzt, von denen Cori insgesamt 2388 Stück besitzt mit jeweils zwei Intel® Xeon™ E5-2698 v3 und 128GB RAM. Da wir diesesmal bis zu 128 Rechenknoten verwendet haben, haben wir einen größeren Datensatz generiert mit 1.000.000 Datenpunkten und wieder 100 Dimensionen. Das Testprogramm wurde mit den gleichen Parametern ausgeführt, wie zuvor. In Abbildung 6.2 ist das Skalierungsverhalten für die Ausführung mit OpenCL zu sehen. Es wurde nur OpenCL getestet, da dies den Worstcase für die Skalierbarkeit

[1] https://simsgs.informatik.uni-stuttgart.de:8444/SGpp/SGpp/blob/HPX_regression/datadriven/examplesHPX/RegressionRefinementExample.cpp Commit fdfb6931

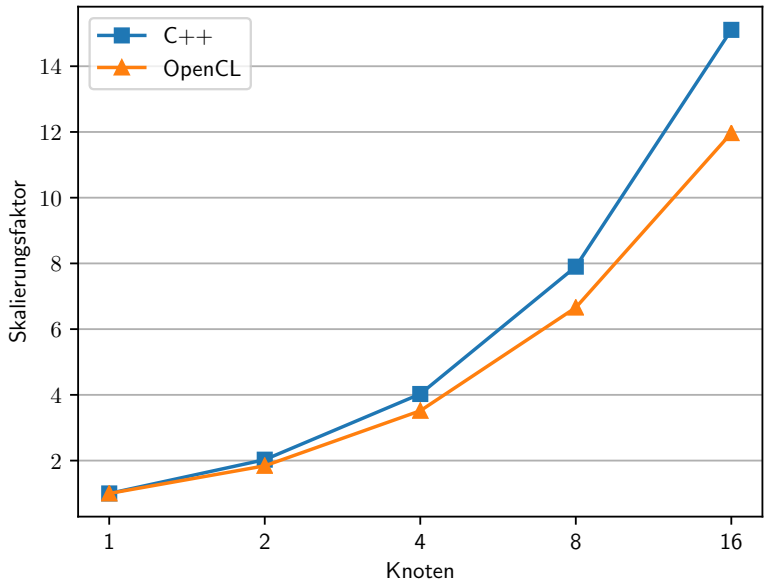

Abbildung 6.1: Skalierungsverhalten auf dem SGSCL1 Cluster einer reiner C++ Implementierung und einer Implementierung mit OpenCL.

darstellt, da OpenCL zusätzlichen Overhead hinzufügt und schneller ist, wodurch der Overhead stärker ins Gewicht fällt. Auf einem Rechenknoten werden knapp 73 Minuten benötigt und auf 128 Rechenknoten knapp 2 Minuten. Das entspricht einem Skalierungsfaktor von knapp 44 und einer parallelen Effizienz von etwa 34%. Wie allerdings zu sehen ist, ist das Skalierungsverhalten bis etwa 32 Rechenknoten nicht weit entfernt vom Optimum mit einer parallelen Effizienz von etwa 84%. Bei mehr als 32 Rechenknoten wird der Overhead zu groß um noch gut zu skalieren. Bei 32 Rechenknoten beträgt die Ausführungszeit lediglich knapp 3 Minuten, weshalb das Hinzufügen von mehr Rechenknoten ohnehin in der Praxis nicht nötig ist. Hat man einen größeren Datensatz, der entsprechend auch eine längere Ausführungszeit benötigt, würde das Skalierungsverhalten voraussichtlich besser sein, da der Overhead etwa gleichgroß bleibt, aber die Rechenzeit pro Rechenknoten größer wird.

6.2 Vergleich zu anderen Algorithmen

Beim Vergleich zu anderen Algorithmen, in Bezug auf die Genauigkeit der Ergebnisse, wurden alle Regressions-Datensätze der Penn Machine Learning Benchmarks [OLO+17] verwendet, die maximal 11 Dimensionen und maximal 1000 Datenpunkte besitzen. Wir haben das *scikit-learn* Framework genutzt, um 10 weitere Regression-Algorithmen zu testen. Wir haben die gleichen Parameter genutzt wie Orzechowski et al. [OLM18]. Die von uns erweiterte SG++ Bibliothek wurde für die Dünngitter-Regression genutzt. Für diese 11 Algorithmen haben wir insgesamt über 600 verschiedene Parameterkombinationen getestet. Alle wurden mit 5-facher Kreuzvaliderung auf den 59 verwendeten Datensätzen getestet mit quadratischem mittleren Fehler. Die getesteten Parameter sind in Tabelle 6.1 aufgelistet und die durchschnittliche Platzierung der Algorithmen ist in Abbildung 6.3 zu sehen. Für die Platzierung wurden bei jedem Algorithmus nur die besten gefunden

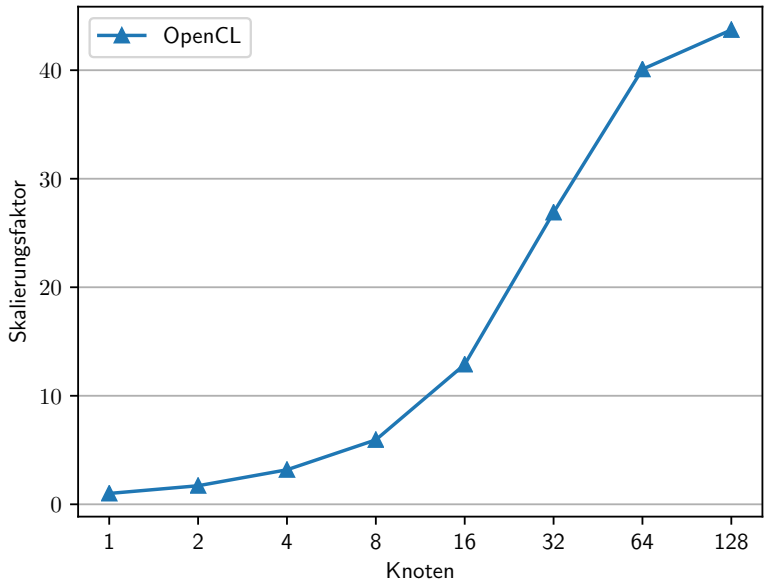

Abbildung 6.2: Skalierungsverhalten der Implementierung mit OpenCL auf den Haswell-Knoten des Cori Supercomputers.

Parameter berücksichtigt. Wie zu sehen ist, hat die Dünngitter-Regression sehr gut abgeschnitten. Sie hat die beste durchschnittliche Platzierung unter den getesteten Algorithmen. Für jeden Algorithmus sind sicherlich besser Parameter findbar, aber für einen groben Vergleich reichen diese aus. Das Ziel war hauptsächlich zu prüfen, ob SG++ nach unseren Erweiterungen immer noch korrekte Ergebnisse liefert, was hiermit bestätigt ist.

Algorithmus Name	Parameter Namen	Werte
Dünngitter	'num_refine' 'no_points' 'lambda'	{0, 1, 2, 3, 4, 5} {5, 10, 20} {1e-4, 0.001, 0.01}
AdaBoostRegressor	'n_estimators' 'learning_rate'	{10, 100, 1000} {0.01, 0.1, 1, 10}
GradientBoostingRegressor	'n_estimators' 'min_weight_fraction_leaf' 'max_features'	{10, 100, 1000} {0.0, 0.25, 0.5} {'sqrt', 'log2', None}
KernelRidge	'kernel' 'alpha' 'gamma'	{'linear', 'poly', 'rbf', 'sigmoid'} {1e-4, 0.01, 0.1, 1} {0.01, 0.1, 1, 10}
LassoLARS	'alpha'	{1e-4, 0.001, 0.01, 0.1, 1}
LinearRegression	default	default
MLPRegressor	'activation' 'solver' 'learning_rate'	{'logistic', 'tanh', 'relu'} {'lbfgs', 'adam', 'sgd'} {'constant', 'invscaling', 'adaptive'}
RandomForestRegressor	'n_estimators' 'min_weight_fraction_leaf' 'max_features'	{10, 100, 1000} {0.0, 0.25, 0.5} {'sqrt', 'log2', None}
SGDRegressor	'alpha' 'penalty'	{1e-6, 1e-4, 0.01, 1} {'l2', 'l1', 'elasticnet'}
LinearSVR	'C' 'loss'	{1e-6, 1e-4, 0.1, 1} {'epsilon_insensitive', 'squared_epsilon_insensitive'}
XGBoost	'n_estimators' 'learning_rate' 'gamma' 'max_depth' 'subsample'	{10, 50, 100, 250, 500, 1000} {1e-4, 0.01, 0.05, 0.1, 0.2} {6} {0.5, 0.75, 1}

Tabelle 6.1: Verwendete Parameter für die Regressions-Algorithmen. Für alle Algorithmen, bis auf die Dünngitter, wurden die gleichen Parameter verwendet wie bei Orzechowski et al. [OLM18]

.

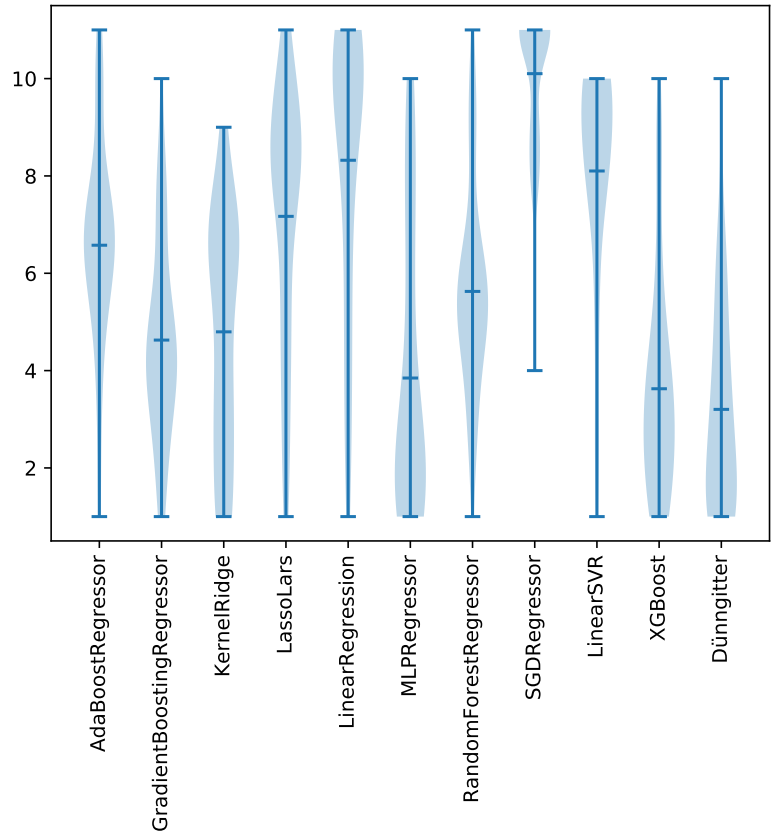

Abbildung 6.3: Durchschnittliche Platzierung der Dünngitter-Regression beim Vergleich mit anderen Regressions-Algorithmen. Es wurde der mittlere quadratische Fehler auf 59 verschiedenen Datensätzen getestet mit 5-facher Kreuzvalidierung.

7 Zusammenfassung und Ausblick

Für diese Arbeit wurde die Dünngitter Bibliothek SG++ mit HPX erweitert, sodass sich die Dünngitter-Regression verteilt auf vielen Rechenknoten ausführen lässt. Wir haben HPX auch genutzt für die Parallelisierung auf den einzelnen Rechenknoten und haben dies verglichen mit der OpenCL Implementierung, die bereits in SG++ vorhanden war. Wir haben gesehen, dass die OpenCL Implementierung etwas schneller war, aber durch den höheren Overhead etwas schlechter skaliert hat.

Auf dem SGSCL1 Cluster haben wir auf 16 Rechenknoten einen Skalierungsfaktor von etwa 15,1 erreicht und eine parallele Effizienz von etwa 94,4% mit der reinen C++ Implementierung. Mit der OpenCL Implementierung hatten wir noch einen Skalierungsfaktor von etwa 12 und eine parallel Effizienz von etwa 74,8%.

Auf dem Supercomputer Cori haben wir dann nur noch die OpenCL Implementierung getestet, da sie den Worstcase für die Skalierbarkeit darstellt. Dabei erreichten wir einen Skalierungsfaktor von 26,9 bei 32 Rechenknoten mit einer parallelen Effizienz von etwa 84%. Auf 128 Rechenknoten erreichten wir dann nur noch einen Skalierungsfaktor von 43,7 und eine parallele Effizienz von etwa 34,2%. Schon bei 32 Rechenknoten war die Ausführungszeit mit knapp 3 Minuten so kurz, dass man in der Praxis sowieso nicht mehr Rechenknoten hinzufügen würde.

Beim Vergleich mit anderen Regressions-Algorithem ging es dann hauptsächlich darum sicherzustellen, dass nach unseren Erweiterungen die Ergebnisse immer noch korrekt sind. Wir haben die Dünngitter-Regression dazu mit 10 anderen Algorithmen verglichen auf 59 verschiedenen Datensätzen. Dabei wurden für die meisten Algorithmen verschiedene Parameter durchprobiert und jeweils die besten Parameter beim Vergleich genutzt. Die Dünngitter hatten dabei die beste durchschnittliche Platzierung, woraus wir schließen können, dass nach unserer Erweiterung immer noch korrekte Ergebnisse berechnet werden.

Es gibt natürlich noch Potenzial für Verbesserungen. Wir haben zum Beispiel nur für die reine C++ Implementierung Kommunikation und Berechnungen überlappt. Bei der OpenCL Implementierung wäre das nicht so einfach möglich gewesen, da die OpenCL Kernel die Daten in sehr großen Blöcken abarbeiten, um den Overhead so gering wie möglich zu halten. Das hat aber dazu geführt, dass die Kernel in den meisten Fällen alle Daten aufeinmal verarbeitet haben und es damit keine Möglichkeit gab Zwischenergebnisse zu verschicken. Ein anderer Punkt, der noch verbessert werden kann, ist das alle Rechenknoten ihre Ergebnisse an den gleichen Rechenknoten zurückschicken. Das kann bei einer großen Anzahl an Rechenknoten zu einem Flaschenhals führen. Dies ließe sich zum Beispiel verhindern, indem die Ergebnisse über *Binary Fan-In* zusammengeführt werden, damit nicht mehr jede Kommunikation über einen einzelnen Rechenknoten läuft.

Literaturverzeichnis

[FPS96] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth. „From Data Mining to Knowledge Discovery in Databases“. In: *AI Magazine* 17.3 (15. März 1996), S. 37–37. ISSN: 2371-9621. DOI: 10.1609/aimag.v17i3.1230. URL: https://www.aaai.org/ojs/index.php/aimagazine/article/view/1230 (besucht am 01. 12. 2018) (zitiert auf S. 9, 11).

[GG01] J. Garcke, M. Griebel. „Data Mining with Sparse Grids Using Simplicial Basis Functions“. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '01. New York, NY, USA: ACM, 2001, S. 87–96. ISBN: 978-1-58113-391-2. DOI: 10.1145/502512.502528. URL: http://doi.acm.org/10.1145/502512.502528 (besucht am 17. 11. 2018) (zitiert auf S. 14, 18, 19).

[HKPB16] A. Heinecke, R. Karlstetter, D. Pflüger, H.-J. Bungartz. „Data mining on vast data sets as a cluster system benchmark“. In: *Concurrency and Computation: Practice and Experience* 28.7 (1. Mai 2016), S. 2145–2165. ISSN: 1532-0634. DOI: 10.1002/cpe.3514. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3514 (besucht am 17. 11. 2018) (zitiert auf S. 9).

[HPX18a] *Action Type Definition*. 2018. URL: http://stellar.cct.lsu.edu/files/hpx-1.1.0/html/hpx/manual/applying_actions/action_type_definition.html (besucht am 13. 03. 2019) (zitiert auf S. 29).

[HPX18b] *Using LCOs*. 2018. URL: http://stellar.cct.lsu.edu/files/hpx-1.1.0/html/hpx/manual/lcos.html (besucht am 13. 03. 2019) (zitiert auf S. 29).

[HPX18c] *Writing Components*. 2018. URL: http://stellar.cct.lsu.edu/files/hpx-1.1.0/html/hpx/manual/components.html (besucht am 13. 03. 2019) (zitiert auf S. 29).

[KBS09] H. Kaiser, M. Brodowicz, T. Sterling. „ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications“. In: *2009 International Conference on Parallel Processing Workshops*. 2009 International Conference on Parallel Processing Workshops. Sep. 2009, S. 394–401. DOI: 10.1109/ICPPW.2009.14 (zitiert auf S. 21).

[KHA+14] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey. „HPX: A Task Based Programming Model in a Global Address Space“. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. New York, NY, USA: ACM, 2014, 6:1–6:11. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883. URL: http://doi.acm.org/10.1145/2676870.2676883 (besucht am 17. 11. 2018) (zitiert auf S. 9, 21–23).

[Lar05] D. T. Larose. *Data Mining Methods and Models*. John Wiley & Sons, Inc., Nov. 2005. DOI: 10.1002/0471756482 (zitiert auf S. 9, 11).

[OLM18] P. Orzechowski, W. La Cava, J. H. Moore. „Where are we now? A large benchmark study of recent symbolic regression methods“. In: *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18* (2018), S. 1183–1190. DOI: 10.1145/3205455.3205539. arXiv: 1804.09331. URL: http://arxiv.org/abs/1804.09331 (besucht am 18. 02. 2019) (zitiert auf S. 32, 34).

[OLO+17] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, J. H. Moore. „PMLB: a large benchmark suite for machine learning evaluation and comparison“. In: *BioData Mining* 10.1 (11. Dez. 2017), S. 36. ISSN: 1756-0381. DOI: 10.1186/s13040-017-0154-4. URL: https://doi.org/10.1186/s13040-017-0154-4 (besucht am 05. 12. 2018) (zitiert auf S. 32).

[Pfl10] D. Pflüger. *Spatially adaptive sparse grids for high-dimensional problems*. 1. Aufl. OCLC: 700066168. München: Verl. Dr. Hut, 2010. 181 S. ISBN: 978-3-86853-555-6 (zitiert auf S. 9, 11–13, 15–20, 25, 26).

[PPB10] D. Pflüger, B. Peherstorfer, H.-J. Bungartz. „Spatially adaptive sparse grids for high-dimensional data-driven problems“. In: *Journal of Complexity*. SI: HDA 2009 26.5 (1. Okt. 2010), S. 508–522. ISSN: 0885-064X. DOI: 10.1016/j.jco.2010.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0885064X10000257 (besucht am 17. 11. 2018).

[SGS10] J. E. Stone, D. Gohara, G. Shi. „OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems“. In: *Computing in science & engineering* 12.3 (Mai 2010), S. 66–72. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2964860/ (besucht am 12. 03. 2019) (zitiert auf S. 26, 27).

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