LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

# Praktikum Rechnerarchitektur

## Parallele Berechnung der Mandelbrotmenge

Wintersemester 2018/19

Maximilian Frühauf Tobias Klasuen Florian Lercher Niels Mündler

# 1 Einleitung

Die Leistung und Geschwindigkeit des individuellen Rechenkerns stagniert seit einigen Jahren. Moderne Computer erlangen einen Großteil ihrer erhöhten Rechenleistung seit einiger Zeit nur noch durch Parallelisierung. Diese sollte jedoch geschickt gestaltet werden, um unerwünschte Seiteneffekte wie Leerlauf zu vermeiden.

## 1.1 Didaktische Ziele

Das zugrundeliegende Problem ist, dass bei der Lastaufteilung einer unabhängigen Menge von Berechnungen in einem Cluster eine fixe Zuordnung von zu berechnden Bereichen auf Rechenkerne erzeugt wird. Dauert die Bearbeitung eines Bereiches jedoch deutlich kürzer als diejenige anderer Abschnitte, so verbringt der reservierte Rechenkern die Zeit bis zum Abschluss der anderen Berechnungen ohne Arbeit und verbraucht Strom und Platz im Idle-Mode. Da die Kerne eines Clusters jedoch darauf ausgelegt sind, ständig zu arbeiten, sollte dieser Zustand vermieden werden, um Zeit, Kosten und Energie zu sparen.

Dies kann erreicht werden, indem die Einteilung der Rechenbereiche die vorraussichtliche Rechendauer berücksichtigt. Dazu werden rechenintensive Bereiche verkleinert und umgekehrt Bereiche mit geringerer Rechenlast vergrößert. Ziel sollte sein, dass alle Knoten für die Bearbeitung in etwa gleich lang brauchen, sodass die gegenseitige Wartezeit minimiert wird.

Dieses Projekt soll intuitiv vermitteln, dass bei der Aufteilung unabhängiger Berechnungen auf ein Cluster eine Abschätzung der benötigten Rechenlast die Gesamtrechendauer deutlich verringern kann. Außerdem soll ersichtlich sein, wie die verwendete Aufteilung bestimmt wird.

## 1.2 Verwendung der Mandelbrotmenge

Die Mandelbrotmenge ist eine Teilmenge der komplexen Zahlen. Um sie zu berechnen wendet man folgende Formel wiederholt auf jede komplexe Zahl $c$ an:

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0 \tag{1}$$

In der Mandelbrotmenge befinden sich alle $c$, für die der Betrag von $z_n$ für beliebig große $n$ endlich bleibt. Wenn der Betrag von $z$ nach einer Iteration größer als 2 ist, so strebt $z$ gegen unendlich, das zugehörige $c$ liegt also nicht in der Menge. Sobald $|z_n| > 2$ kann die Berechnung daher abgebrochen werden.

Um nun für eine beliebige Zahl zu bestimmen, ob diese in der Mandelbrotmenge liegt, müssen theoretisch unendlich viele Rechenschritte durchgeführt werden. Zur computergestützten Bestimmung werden die Rechenschritte nach einer bestimmten Iteration abgebrochen die Zahl als in der Menge liegend betrachtet.

Es handelt sich also um eine Berechnung, die sehr zeitaufwändig ist, wobei die benötigte Zeit durch Erhöhen der Iterationszahl beliebig erhöht werden kann. Zusätzlich ist die Berechnung für jede einzelne komplexe Zahl unabhängig von jeder anderen Zahl.

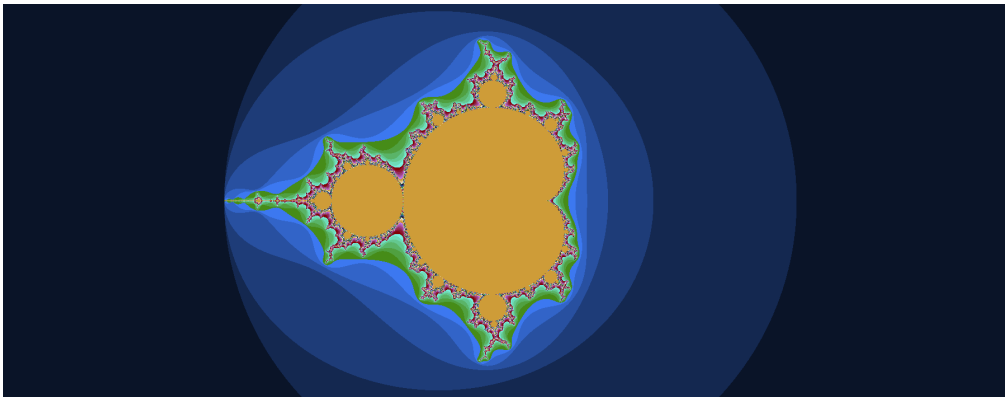

Abbildung 1: Die Mandelbrotmenge, visualisiert in einem Ausschnitt des komplexen Zahlenraumes.

Diese Eigenschaften ermöglichen es, zweierlei Dinge zu kontrollieren:

- Die Dauer der Berechnung
- Die Aufteilung der Berechnung auf unterschiedliche Rechenkerne

Somit kann gesichert werden, dass eine wahrnehmbare Zeit (100-200 ms) zur Berechnung benötigt wird. Zudem kann die Unterteilung des zu berechnden Raumes frei gewählt werden, sodass für verschiedenste Aufteilungen die Gesamtrechenzeit visualisiert werden kann.

## 1.3 Darstellung der Mandelbrotmenge

Komplexe Zahlen lassen sich auch grafisch darstellen, indem man sie in ein Koordinatensystem einträgt. Dabei entspricht die x-Koordinate dem Realteil und die y-Koordinate dem Imaginärteil der Zahl. Für das Projekt wird ein Ausschnitt des Bildschirmes als zweidimensionale Darstellung des komplexen Raumes betrachtet und für jeden darin liegenden Punkt die Zugehörigkeit zur Mandelbrotmenge bestimmt. Dabei wird der Raum jedoch diskretisiert, indem jedem Pixel des Bildschirmes die komplexen Koordinaten $c$ der linken oberen Ecke zugeordnet werden.

Die grafische Darstellung der Mandelbrotmenge wird durch Einfärbung des zu $c$ gehörigen Pixels erhalten. Die Zahl der benötigten Iterationen bis zum Abbruch der Berechnung bestimmt dabei die Farbe, sodass alle Pixel innerhalb der Menge und alle Pixel außerhalb jeweils gleichfarbig sind.

Das entstehende Fraktal ist aufgrund seiner Form auch als “Apfelmännchen” bekannt (siehe Abbildung 1). Die Menge ist zusammenhängend, jedoch bilden sich an ihren Rändern viele kleine und sehr komplexe Formen, die visuell ansprechend sind. Es eignet sich daher gut, um optisch Interesse am Projekt zu wecken.

## 1.4 MPI

Das Message Passing Interface[1] ist eine weit verbreitete Spezifikation, für die Kommunikation zwischen unabhängigen Rechenkernen. Dadurch existieren viele gut funktionierende Umsetzungen in einer Vielzahl von Programmiersprachen. Für dieses Projekt wichtig ist, dass es echte Parallelisierung mit geringem Overhead ermöglicht. So können die einzelnen Berechnungen auf jeweils eigenen unabhängigen Rechenkernen laufen und die Art der Aufteilung erhält größtmögliche Bedeutung. Die Gestaltung von MPI erlaubt dabei beliebige Zuordnungen, von Kernen auf einem Prozessor bis hin zu unabhängigen Clusterknoten, die lediglich eine SSH-Verbindung besitzen.

## 1.5 Qualitätsanforderungen

Die Benutzeroberfläche soll so leicht und intuitiv wie möglich zu bedienen sein. Hierbei soll zudem darauf geachtet werden, dass alle Funktionen nur mit einer minimalen Anzahl an Mausklicks auszuführen sind und die Oberfläche nicht überladen wird.

Zudem soll das System robust gestaltet werden. Dies wird durch die Verwendung von Buttons, Listen, Drop-Down Menüs und Slider gewährleistet, die die Möglichkeit der Eingabe von ungültigen Werten verhindern.

Es sind keine Sicherheitsfeatures (Benutzerauthentisierung, Verschlüsselung) geplant, da keine sensiblen Daten verarbeitet werden und die Anwendung nicht uneingeschränkt über das Internet zugänglich ist.

## 1.6 Einschränkungen

Die Benutzeroberfläche soll in einem Webbrowser lauffähig sein, sodass sie auf beliebigen Endgeräten zugänglich gemacht werden kann. Um die erwartete Performanzsteigerung an einem echten Beispiel zu demonstrieren, soll die Berechnung der Mandelbrotmenge parallel auf mehreren Raspberry Pi's[2] oder ähnlichen unabhängigen Kleincomputern oder Rechenkernen zum Einsatz kommen soll.

[1] `https://www.mpi-forum.org/`

[2] Für ein Beispiel, siehe `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`

# 2 Problemstellung und Motivation

- Fachliche Spezifikation in Anhang
- NFRs in Einleitung schreiben

# 3 Dokumentation der Implementierung

- Sollte größter Teil werden
- 1. High level overview?
  2. Wie läuft das auf meinem System?
  3. Code Dokumentation / Entwicklerdokumentation

## 3.1 Aufsetzen des Backends

### 3.1.1 Schnellstart

Um das System schnell zum laufen zu bekommen, muss das Repository auf das eigene System geklont werden. Im Unterverzeichnis => README aus backend/himmuc übertragen

## 3.2 Implementierung der Mandelbrotberechnung

Zur hardwarenahe Berechnung der Mandelbrotmenge wird ein sogenanntes Backend gestartet. Das in C++ programmierte Teilprojekt nimmt Rechenaufträge von einem Nutzer durch ein Frontend entgegen (auch ein solches wird bereitgestellt), zerlegt sie und verteilt sie per MPI auf dedizierte Rechenknoten. Dazu besteht das Backend aus zwei ausführbaren Dateien, host und worker.

## 3.3 Inkludierte Header und CMake Anweisungen

Die zusammenstellung der ausführbaren Dateien wird in CMake definiert. Dabei unterscheiden sich diese lediglich in den eingebundenen Quelldateien: In die Datei host werden host.main.cpp und actors/Host.cpp eingebunden, während in worker worker.main.cpp und actors/Worker.cpp eingebunden werden.

Diese und alle weiteren Build-Vorgaben werden in der Datei CMakeLists.txt für cmake[3] in der hier beschriebenen Reihenfolge spezifiert.

Es sollte hierbeit eine CMake-Version über 3.7.0 gewählt werden und die C++11 Standarts[4] werden vorrausgesetzt. Zudem werden für das Projekt "Mandelbrot"werden alle Dateien im Order include eingebunden. In diesem Ordner liegen die Header-Dateien für alle projektinternen C++-Quelldateien. Anschließend werden alle C++-Quelldateien (Endung ".cpp") aus dem Ordner src in einer Liste gesammelt, mit Ausnahme jedoch der oben genannten, exklusiven Quelldateien. Die erzeugte Liste und die jeweils exklusiven Dateien werden dann den ausführbaren Dateien host und worker zugeordnet.

Um die verwendeten Biblioteken verfügbar zu machen werden anschließend die Header der installierten MPI-Bibliotek sowie die Header der Bibliotheken rapidjson[5],

[3] Ein Programm, welches die Erstellung von Makefiles vereinfacht in dem es sie automatisch an die Umgebung des Build-Systems anpasst. `https://cmake.org/`

[4] `https://isocpp.org/wiki/faq/cpp11`

[5] `http://rapidjson.org`

websocketpp[6] und boost[7] Diese werden respektive verwendet um JSON zu parsen und enkodieren, Websocket-Verbindungen aufzubauen und darüber zu kommunieren sowie um diese Bibliothek zu unterstützen. Da für die boost Bibliothek dabei Header nicht genügen und die systemweite Verfügbarkeit der kompilierten boost-Bibliothek nicht garantiert werden kann, wird die Teilbibliothek boost_system statisch in die ausführbaren Datei host eingebunden.

Zuletzt werden über Compilerflags alle Kompilierfehler und -warnungen aktiviert sowie die POSIX-Thread-Bibliothek eingebunden und spezielle Flags für die Websocketlibrary und MPI gesetzt.

## 3.4 Mainfunktion und Initialisierung

Zur Initialisierung der Prozesse muss zunächst die MPI-Umgebung aktiviert und abgerufen werden. Dies geschieht für beide Programme gleich, über die Initialisierungsfunktion in Quelltext 1. Sie erwartet lediglich eine Beschreibung des Prozesses für den Log und eine Initialisierungsfunktion, die erst zurückkehrt, wenn das Programm abgeschlossen ist und MPI beendet werden soll. Die Funktion muss als Parameter den Rang bzw. die Id des aktuellen MPI-Prozesses und die Anzahl der initialisierten Prozesse entgegen nehmen.

Ein beispielhafter Aufruf ist in Quelltext 2 zu sehen.

## 3.5 Lastbalancierung

Um die Mandelbrotmenge effizient parallel zu berechnen, muss die Last gleichmäßig auf die Worker verteilt werden. Die Aufgabe der Lastbalancierung besteht darin zu einer gegebenen Region und einer Anzahl von Workern eine solche Unterteilung in sogenannte Teilregionen zu finden. Wichtig dabei ist, dass der garantierte Teiler von Höhe und Breite der Teilregionen dem der angeforderten Region entspricht, da es sonst im Frontend zu Schwierigkeiten bei der Darstellung kommt. Die Klassenstruktur der Lastbalancierer entspricht dem Strategy-Pattern. So kann der Balancierer zur Laufzeit leicht gewechselt werden und auch die Erweiterung des Projekts um eine weitere Strategie gestaltet sich einfach. Was dabei genau beachtet werden muss findet sich im Teil Erweiterung.

Damit die Unterschiede zwischen guter und schlechter Lastverteilung deutlich werden, wurden hier verschiedene Strategien zur Lastbalancierung implementiert.

### 3.5.1 Naive Strategie

Bei der naiven Strategie (zu finden in den Klassen NaiveBalancer und RecursiveNaiveBalancer) wird versucht den einzelnen Workern etwa gleich große Teilregionen zuzuweisen. Dies geschieht allerdings ohne Beachtung der eventuell unterschiedlichen Rechenzeiten

[6] `https://github.com/zaphoyd/websocketpp`
[7] `https://www.boost.org/`

```
int init(int argc, char **argv, const char* type, void (*initFunc) (int
    world_rank, int world_size)) {

    MPI_Init(&argc, &argv);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Retreive the processor name to check if host and
    // worker share a node
    char* proc_name = new char[MPI_MAX_PROCESSOR_NAME];
    int proc_name_length;
    MPI_Get_processor_name(proc_name, &proc_name_length);
  std::cout << type << ": " << world_rank << " of " << world_size <<
        " on node " << proc_name << std::endl;

    if (world_size < 2) {
       std:: cerr << "Need at least 2 processes to run. Currently have "
          << world_size << std::endl;
        MPI_Finalize();
        return -1;  // return with error
    }
    initFunc(world_rank, world_size);
    MPI_Finalize();

    return 0;
}
```

Quelltext 1: Initialisierung der MPI-Prozesse in init.cpp

```
#include "init.h"
#include "Host.h"

int main(int argc, char *argv[])
{
    return init(argc, argv, "Host", Host::init);
}
```

Quelltext 2: Initialisierung des Host-Prozesses in host.main.cpp

innerhalb der Teilregionen. Die naive Strategie wurde hier in einer nicht-rekursiven und einer rekursiven Variante implementiert.

Zur nicht-rekursiven Aufteilung wird zuerst die Anzahl der zu erstellenden Spalten und Zeilen berechnet. Dazu wird der größte Teiler der Anzahl der Worker bestimmt. Dieser gibt die Anzahl der Spalten an, das Ergebnis der Division ist die Anzahl der Zeilen. Damit ist sichergestellt, dass die Region in die richtige Menge von Teilregionen unterteilt wird.

Als nächstes wird die Breite und Höhe der Teilregionen berechnet. Hierbei ist wichtig, dass der garantierte Teiler erhalten wird. Die Breite berechnet sich also durch:

$$\frac{region.width}{region.guaranteedDivisor * nodeCount} * region.guaranteedDivisor$$

Dabei ist $nodeCount$ die Anzahl der Worker und $region$ die zu unterteilende Region. Es ist zu beachten, dass es sich hier um eine Ganzzahldivision handelt, deren Rest angibt, wie viele Teilregionen um $region.guaranteedDivisor$ breiter sind. Die Höhe berechnet sich analog.

Bevor die eigentliche Aufteilung beginnt werden noch die Deltas für Real- und Imaginärteil bestimmt. Diese geben an, wie breit bzw. hoch der Bereich der komplexen Ebene ist, den ein Pixel der Region überdeckt. Die Deltas können über Methoden der Klasse Fractal berechnet werden.

Zur Aufteilung wird nun mittels zweier verschachtelter Schleifen über die Zeilen und Spalten iteriert. Die benötigten Start- und Endpunkt der Teilregionen (also die linke obere Ecke und die rechte untere Ecke auf der komplexen Ebene) können nun mithilfe der Schleifenzähler und der Deltas bestimmt werden. Zusätzlich wird noch der vertikale und horizontale Offset der Teilregion vom Startpunkt der Eingaberegion abgespeichert. Diese Information ermöglicht es die Region im Frontend einfach anzuzeigen.

Falls die aktuell betrachtete Teiregionen zu den Breiteren und/oder Höheren (s.o.) gehört, müssen alle Werte entsprechend angepasst werden. Die letzte Teilregion einer Spalte bzw. Zeile wird immer so gewählt, dass sie auf jeden Fall mit dem Rand der Eingaberegion abschließt.

Die Teilregionen werden in einem Ergebnisarray gespeichert, welches dann zurückgegeben wird.

Die Grundidee der rekursiven Balancierung ist, die Region solange zu halbieren, bis man genug Teile für jeden Worker hat. Dies funktioniert sehr gut, wenn die Anzahl der Worker eine 2er-Potenz ist. Wenn das nicht der Fall ist, so müssen für einige Worker die Regionen öfters geteilt werden als für andere. Die Anzahl der Blätter des Rekursionsbaumes (hier ein Binärbaum) muss also der Anzahl der Worker entsprechen. Die Rekursionstiefe kann man wie folgt berechnen:

$$recCounter = \lfloor \log_2 nodeCount \rfloor + 1 \tag{2}$$

Und die Anzahl der Teilregionen auf der untersten Ebene des Rekursionsbaumes ergibt sich als:

$$missing = nodeCount - 2^{\lfloor \log_2 nodeCount \rfloor}$$
$$onLowestLevel = missing * 2 \tag{3}$$

Auch hier ist $nodeCount$ die Anzahl der Worker. Die Multiplikation mit 2 ist nötig, da nochmal $missing$ Blätter aufgeteilt werden müssen, um $missing$ zusätzliche Blätter zu erzeugen.

Um diese Werte einfach durch die Rekursionsebenen zu reichen wurde die Struktur BalancingContext definiert, welche zusätzlich noch die beiden Deltas, den Index in das Ergebnisarray und einen Zeiger auf das Array selbst abspeichert.

Aus $recCounter$ und $onLowestLevel$ kann auch die Abbruchbedingung der Rekursion gefolgert werden:

$$recCounter = 0 \vee (recCounter = 1 \wedge resultIndex \geq onLowestLevel) \equiv true \tag{4}$$

$resultIndex$ ist hierbei der Index in das Ergebnisarray, beschreibt also die Anzahl der bereits erstellten Teilregionen.

Ist die Abbruchbedingung (4) erfüllt, so genügt es die übergebene Region in das Ergebnisarray einzutragen und $resultIndex$ zu inkrementieren. Ansonsten muss die Region halbiert werden. Ist die Region vertikal oder horizontal nicht mehr teilbar (d.h. $region.width$ bzw. $region.height \leq region.guaranteedDivisor$), so wird in die andere Richtung geteilt. Kann die Region in beide Richtungen geteilt werden, so wird abwechselnd vertikal und horizontal geteilt. Dies gewährleistet, dass der Balancierer in beide Richtungen aufteilt, sofern das möglich ist. Wenn die Region unteilbar ist, so muss eine leere Region erzeugt werden.

Die beiden Hälften berechnen sich wie bei der Aufteilung auf zwei Worker mit der nicht-rekursiven Strategie. Dann wird die Funktion für jede Hälfte rekursiv aufgerufen.

### 3.5.2 Strategie mit Vorhersage

Bei dieser Strategie (zu finden in den Klassen PredictionBalancer und RecursivePredictionBalancer) basiert die Aufteilung der Region auf einer Vorhersage über die Rechenzeit. Die Teilregionen werden so gewählt, dass sie, entsprechend der Vorhersage, etwa einen ähnlichen Rechenaufwand haben.

Die Vorhersage (struct Prediction) wird von der Klasse Predicter angestellt. Dazu wird die Region in einer sehr viel geringeren Auflösung berechnet. Die benötigte Anzahl an Iterationen wird jeweils pro Kachel (Breite und Höhe sind der garantierte Teiler) abgespeichert. So wird sichergestellt, dass der garantierte Teiler auch nach der Aufteilung noch gilt, da die Balancierer die Vorhersage Eintrag für Eintrag verarbeiten. Die Genauigkeit der Vorhersage kann über das Attribut $predictionAccuracy$ gesteuert werden:

- $predictionAccuracy > 0$: $(predictionAccuracy)^2$ Pixel werden pro Kachel berechnet. Die Summe der Iterationen für die einzelnen Pixel ergibt die Vorhersage für die Kachel.

- $predictionAccuracy < 0$: Für $(predictionAccuracy)^2$ Kacheln wird ein Pixel in der Vorhersage berechnet. Es erhalten also mehrere Kacheln diesselbe Vorhersage.
- $predictionAccuracy = 0$: Unzulässig, es wird ein Null-Pointer zurückgegeben.

Es ist wichtig eine gute Balance zwischen Güte und Geschwindigkeit der Vorhersage zu finden. Zusätzlich beinhaltet die Vorhersage die Summen der benötigten Iterationen pro Spalte und Zeile, sowie die Gesamtsumme. Auch die Deltas für Real- und Imaginärteil pro Kachel und die Anzahl der Zeilen und Spalten werden angegeben.

Auch die Strategie mit Vorhersage wurde in einer rekursiven und in einer nicht-rekursiven Variante implementiert.

Für die nicht-rekursive Variante wird zuerst die benötigte Anzahl an Zeilen und Spalten bestimmt. Dies geschieht genauso wie bei der naiven Strategie.

Die erzeugten Teilregionen sollen in etwa den gleichen Rechenaufwand haben. Dieser berechnet sich durch:

$$desiredN = \frac{nSum}{nodeCount} \tag{5}$$

Dabei ist $nSum$ die Gesamtsumme der Vorhersage und $nodeCount$ wieder die Anzahl der Worker.

Danach wird die Region erst in Spalten aufgeteilt und in einem zweiten Schritt wird dann die horizontale Unterteilung in Teilregionen vorgenommen.

Zur Aufteilung wird über die Spaltensummen in der Vorhersage iteriert. Diese werden aufaddiert und bilden so den Zähler $currentN$. Sobald $currentN \geq desiredN$ gilt oder für alle restlichen Spalten nur noch je ein Eintrag in den Spaltensummen vorhanden ist, wird eine Spalte abgeschlossen. Dazu werden $maxReal$ und $width$ aus den Zählern berechnet. Es ist wichtig $maxReal$ immer neu aus $region.minReal$ zu berechnen, anstatt nur das Delta aufzuaddieren, da sich sonst der Fehler, der bei Fließkommaaddition unvermeidbar ist, auch mit aufaddiert. $minReal$ und $hOffset$ stehen bereits in $tmp$, die Werte wurden bei der Berechnung der vorhergehenden Spalte bereits gesetzt. Jetzt wird $tmp$ für die nächste Spalte vorbereitet, das heißt $tmp.minReal$ wird auf $tmp.maxReal$ gesetzt und $tmp.hOffset$ wird um $tmp.width$ erhöht. Ersteres vermeidet das Entstehen von Lücken zuverlässig. Außerdem wird $desiredN$ für die verbleibenden Spalten nach (5) neu berechnet und die Zähler werden zurückgesetzt. Bevor die Aufteilung der Spalte in Teilregionen startet, wird eine Kopie der Vorhersage erstellt, die nur die Werte für die aktuelle Spalte enthält. Das Aufteilen einer Spalte in Teilregion geschieht analog zum Aufteilen in Spalten.

Die letzte Spalte wird gesondert behandelt: Sie muss so gewählt werden, dass sie den gesamten Rest der Eingaberegion abdeckt, ansonsten können Lücken entstehen. Dies gilt genauso bei der Unterteilung der einzelnen Spalten.

Die rekursive Variante der Strategie mit Vorhersage verwendet dasselbe Rekursionsschema wie ihr naives Gegenstück. Die Werte in BalancingContext berechnen sich also wie in (2) und (3). Auch die Abbruchbedingung ist wie in (4). Die Entscheidung, ob

horizontal oder vertikal geteilt werden soll, wird auch auf die gleiche Art und Weise gefällt.

Bei dieser Strategie werden die Regionen allerdings nicht einfach halbiert, sondern in zwei Teile aufgeteilt, die laut der Vorhersage ähnlich rechenintensiv sind. Dazu wird so vorgegangen, wie bei der nicht-rekursiven Variante für die Aufteilung auf zwei Worker. Es ist hierbei auch wichtig die Vorhersage so zu teilen, dass es für jede Hälfte eine Vorhersage gibt, die dann an den rekursiven Aufruf übergeben werden kann.

#### 3.5.3 Leere Regionen

#### 3.5.4 Erweiterung

### 3.6 Berechnung der Mandelbrotmenge

#### 3.6.1 Berechnung mithilfe von SIMD

Um die Berechnungen intern noch zu beschleunigen, kann SIMD zur parallelen Bearbeitung mehrerer Punkte verwendet werden. Dazu ist es hilfreich, sich zunächst vor Augen zu führen, wie ein Vektor komplexer Koordinaten ohne SIMD verarbeitet würde. Dies ist in Quelltext 3 zu sehen. Die Berechnung der einzelnen Kooridnaten bleibt gleich, nur die Abbruchbedingung wird auf alle bearbeiteten Koordinaten erweitert. Sofern nur eine Koordinate nicht abschließend bearbeitet wurde, muss über den gesamten Vektor weiter iteriert werden. Da jedoch $z_n$, deren Iteration abgebrochen wurde nicht erneut Betragsmäßig unter $2$ kommen[1], steigt ihre Iterationszahl nicht weiter, denn die Iterationszahl wird lediglich erhöht wenn der Betrag des z-Wertes wieder unter 2 liegt.

Damit kann die Berechnung relativ simpel via Arm NEON Compiler Intrinsics[8] implementiert werden (siehe dazu Quelltext 4). Dabei wurden Optimierungen mithilfe der NEON-nativen multiply-add und multiply-subtract Befehle vorgenommen. Zudem kann der parallele Vergleich zweier Vektoren ausgenutzt werden, wobei als Ergebnis jedoch nicht 1 und 0 ausgegeben werden, sondern alle Bits des Ergebnisvektors auf 1 gesetzt werden sofern die Bedingung erfüllt ist und sonst auf 0. Um diese im weiteren Verlauf einfach aufzuaddieren, sodass die Summe der Anzahl nicht abgeschlossener Berechnungen entspricht, wird daher der Ergebnisvektor des Vergleiches mit 1 verundet.

[8] Details im Abschnitt "Compiler Intrinsicsünter `https://developer.arm.com/technologies/neon`

```
1  void MandelbrotVect::calculateFractal(precision_t* cReal, precision_t*
       cImaginary, unsigned short int maxIteration, int vectorLength,
       unsigned short int* dest) {
2      if(vectorLength <= 0){
3          throw std::invalid_argument("vectorLength may not be less than 1.
               ");
4      }
5      std::fill_n(dest, vectorLength, 0);
6      precision_t* zReal = new precision_t[vectorLength];
7      precision_t* zImaginary = new precision_t[vectorLength];
8      precision_t* nextZReal = new precision_t[vectorLength];
9      precision_t* nextZImaginary = new precision_t[vectorLength];
10     // Factor that is multiplied on iteration count and computation
11     // is 0 or 1 and determines whether that point is still being
           computed
12     bool* factor = new bool[vectorLength];
13     std::fill_n(factor, vectorLength, 1);
14     // Bool storing information about whether any abs value that is being
           computed
15     // is still below two => continue computation
16     unsigned int lessThanTwo = vectorLength; // as we begin with ZReal/
           ZImag as 0
17     int i = 0;
18     while (i < maxIteration && lessThanTwo > 0){
19         lessThanTwo = 0;
20         for(int k = 0; k < vectorLength; k++){
21             // Compute next step in iteration
22             nextZReal[k] = (zReal[k] * zReal[k] - zImaginary[k] *
                   zImaginary[k]) + cReal[k];
23             nextZImaginary[k] = 2 * (zReal[k] * zImaginary[k]) +
                   cImaginary[k];
24             zReal[k] = nextZReal[k];
25             zImaginary[k] = nextZImaginary[k];
26             // Determine whether to stop
27             factor[k] = (factor[k] && zReal[k] * zReal[k] + zImaginary[k]
                   * zImaginary[k] < 4.0) ? 1 : 0;
28             // sum => if any number is still less than two, we need to
                   continue
29             lessThanTwo += factor[k];
30             // increase number of iterations if this number hasnt aborted
                   yet
31             dest[k] += factor[k];
32         }
33     }
34     delete[] zReal;
35     delete[] zImaginary;
36     delete[] nextZReal;
37     delete[] nextZImaginary;
38     delete[] factor;
39 }
```

Quelltext 3: Bearbeitung eines Vektors komplexer Koordinaten in C++

```
    // General form of vector commands
    // v<cmd>q_f<pr>
    // v -> vector command
    // q -> double amount of used registers (quad=>4)
    // f -> float
    // 32 bit vectorization
    // Load values from array to simd vector
    float32x4_t cReal = vdupq_n_f32(0);// = vld1q_f32(cRealArray); if
        casting weren't necessary this would work
    cReal = vsetq_lane_f32((float32_t) cRealArray[0], cReal, 0);
    cReal = vsetq_lane_f32((float32_t) cRealArray[1], cReal, 1);
    cReal = vsetq_lane_f32((float32_t) cRealArray[2], cReal, 2);
    cReal = vsetq_lane_f32((float32_t) cRealArray[3], cReal, 3);
    float32x4_t cImaginary = vdupq_n_f32(0);// = vld1q_f32(
        cImaginaryArray);
    cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[0],
        cImaginary, 0);
    cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[1],
        cImaginary, 1);
    cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[2],
        cImaginary, 2);
    cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[3],
        cImaginary, 3);
    // The z values
    float32x4_t zReal = vdupq_n_f32(0);
    float32x4_t zImaginary = vdupq_n_f32(0);
    // determines if computation will be continued
    float32x4_t two = vdupq_n_f32(2);
    float32x4_t four = vdupq_n_f32(4);
    // result iterations
    uint32x4_t one = vdupq_n_u32(1);
    uint32x4_t n = vdupq_n_u32(0);
    uint32x4_t absLesserThanTwo = vdupq_n_u32(1);
    int i = 0;
    while(i < maxIteration && vaddvq_u32(absLesserThanTwo) > 0){
        // add a b -> a+b
        // mls a b c -> a - b*c
        // mul a b -> a*b
        float32x4_t nextZReal = vaddq_f32(vmlsq_f32(vmulq_f32(zReal,
            zReal), zImaginary, zImaginary), cReal);
        // mla a b c -> a + b*c
        float32x4_t nextZImaginary = vmlaq_f32(cImaginary, two, vmulq_f32
            (zReal, zImaginary));
        zReal = nextZReal;
        zImaginary = nextZImaginary;
        // Square of the absolute value -> determine when to stop
        float32x4_t absSquare = vmlaq_f32(vmulq_f32(zReal, zReal),
            zImaginary, zImaginary);
        // If square of the absolute is less than 4, abs<2 holds -> 1
            else 0
        absLesserThanTwo = vandq_u32(vcltq_f32(absSquare, four), one);
        // if any value is 1 in the vector (abs<2) then dont break
        // addv => sum all elements of the vector
        n = vaddq_u32(n, absLesserThanTwo);
        i++;
    }
    // write n to dest
    dest[0] = vgetq_lane_u32(n, 0);
    dest[1] = vgetq_lane_u32(n, 1);
    dest[2] = vgetq_lane_u32(n, 2);
    dest[3] = vgetq_lane_u32(n, 3);
```

Quelltext 4: Parallelisierte Bearbeitung eines Vectors komplexer Koordinaten in C++

# 4 Ergebnisse / Evaluation

- Skalierbarkeitsgraph
- Wie gut ist SIMD / OpenMP / MPI / Mischformen?
- Frontend Overhead messen

# 5 Zusammenfassung

- Zusammenfassung
- Ausblick

## Literatur

[1] mrf (https://math.stackexchange.com/users/19440/mrf). Why is the bailout value of the mandelbrot set 2? Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/424331 (version: 2013-06-24).