

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME  
**Praktikum Rechnerarchitektur**Parallelle Berechnung der Mandelbrotmenge  
Wintersemester 2018/19

Maximilian Frühauf

Tobias Klausen

Florian Lercher

Niels Mündler

**Inhaltsverzeichnis**

<b>1 Hintergrund</b>	<b>3</b>
1.1 Die Mandelbrotmenge . . . . .	3
1.2 Darstellung der Mandelbrotmenge . . . . .	3
1.3 MPI . . . . .	3
1.4 SIMD . . . . .	4
1.5 TypeScript und React . . . . .	5
<b>2 Einleitung</b>	<b>6</b>
2.1 Problemstellung und Motivation . . . . .	6
2.2 Didaktische Ziele . . . . .	6
2.3 Qualitätsanforderungen . . . . .	6
2.4 Einschränkungen . . . . .	7
<b>3 Lösungsansatz</b>	<b>8</b>
3.1 Verwendung der Mandelbrotmenge . . . . .	8
3.2 Architekturübersicht . . . . .	8
3.3 Lastbalancierung . . . . .	8
3.3.1 Naive Strategie . . . . .	9
3.3.2 Strategie mit Vorhersage . . . . .	9
3.3.3 Implementierungsvarianten . . . . .	10
3.4 Konzept der ausgetauschten Nachrichten . . . . .	10
<b>4 Installation der Anwendung</b>	<b>13</b>
4.1 Lokales Backend . . . . .	13
4.2 Backend auf HimMUC Cluster . . . . .	13
4.3 Installation des Frontends . . . . .	14
<b>5 Dokumentation der Implementierung</b>	<b>16</b>
5.1 Implementierung des Backends . . . . .	16
5.1.1 Inkludierte Header und CMake Anweisungen . . . . .	16
5.1.2 Mainfunktion und Initialisierung . . . . .	16
5.2 Implementierung der Lastbalancierung . . . . .	16
5.2.1 Naive Strategie . . . . .	17
5.2.2 Strategie mit Vorhersage . . . . .	17
5.2.3 Erweiterung . . . . .	19

---

5.3	Kommunikation zwischen Host und Worker im Backend . . . . .	20
5.3.1	Beschreibung der verwendeten MPI-Funktionen . . . . .	21
5.3.2	MPI-Designentscheidungen . . . . .	22
5.3.3	Beschreibung der implementierten MPI-Hauptroutinen . . . . .	25
5.4	Berechnung der Mandelbrotmenge . . . . .	25
5.4.1	Berechnung ohne SIMD . . . . .	26
5.4.2	Berechnung mithilfe von SIMD . . . . .	27
5.5	Leistungssteigerung durch Parallelisierung auf Thread Level mithilfe von OpenMP . . . . .	28
5.6	WebSocketverbindung . . . . .	28
5.7	Implementierung des Frontends . . . . .	30
5.7.1	Kommunikation mit dem Backend . . . . .	30
5.7.2	Darstellung der Regionsdaten . . . . .	32
5.7.3	Visualisierung . . . . .	34
5.7.4	Visualisierung der Rechenzeiten . . . . .	35
<b>6</b>	<b>Ergebnisse / Evaluation</b>	<b>37</b>
6.1	Datenerhebung . . . . .	37
6.2	Skalierung . . . . .	38
6.3	OpenMP . . . . .	38
6.4	SIMD . . . . .	38
6.5	Lastbalancierung . . . . .	41
6.6	Zusammenfassung . . . . .	43
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>44</b>
<b>8</b>	<b>Anhang</b>	<b>47</b>
8.1	Detaillierter Start des Backends auf dem HIMMUC . . . . .	47
8.2	Detaillierte Beschreibung der Header und der CMake Instruktionen . . . . .	50
8.3	Beispielhafte Einbindung der MPI-Initialisierungsfunktion . . . . .	50
8.4	Detaillierter Ablauf der Host::handle_region_request Methode . . . . .	50
8.5	Beispiele für versendete Daten . . . . .	51
8.6	Nutzbarkeit einzelner Worker . . . . .	51
8.7	Entwicklung des Beschleunigungsfaktors von SIMD für höhere Iterationszahlen . . . . .	55
8.8	Kommunikation der Prozesse per MPI . . . . .	56
8.8.1	Genereller Aufbau der MPI-Kommunikation . . . . .	57
8.8.2	Übertragung neuer Rechenaufträge vom Host an die Worker . . . . .	58
8.8.3	Übertragung der berechneten Daten von den Workern zum Host	61

---

## 1 Hintergrund

### 1.1 Die Mandelbrotmenge

Die Mandelbrotmenge ist eine Teilmenge der komplexen Zahlen. Um sie zu berechnen wendet man folgende Formel wiederholt auf eine komplexe Zahl  $c \in \mathbb{C}$  an:

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

In der Mandelbrotmenge befinden sich alle solche  $c$ , für die  $\lim_{n \rightarrow \infty} |z_n| < \infty$ . Wenn nach der  $n$ ten Iteration  $|z_n| > 2$  ist, so strebt  $z$  gegen unendlich, das zugehörige  $c$  liegt also nicht in der Menge. Somit sobald  $|z_n| > 2$  die Berechnung daher abgebrochen werden[2].

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.

### 1.2 Darstellung der Mandelbrotmenge

Eine komplexe Zahl  $c \in \mathbb{C}$  lässt sich grafisch darstellen, indem man sie in ein zweidimensionales Koordinatensystem einträgt. Dabei entspricht die x-Koordinate dem Realteil  $Re(c)$  und die y-Koordinate dem Imaginärteil  $Im(c)$ . 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 die komplexe Ebene  $\mathbb{C}$  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). Am Rand der Menge bilden sich viele kleine und sehr komplexe Formen, die visuell ansprechend sind.

### 1.3 MPI

Das Message Passing Interface<sup>1</sup> ist eine weit verbreitete Spezifikation, für die Kommunikation zwischen unabhängigen Rechenkernen. Dadurch existieren viele gut funktionierende Umsetzungen in einer Vielzahl von Programmiersprachen. Es ermöglicht echte Parallelisierung mit geringem Overhead. So können die einzelnen Berechnungen auf jeweils eigenen unabhängigen Rechenkernen laufen und die Art der Aufteilung

---

<sup>1</sup><https://www.mpi-forum.org/>



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

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.

## 1.4 SIMD

„Single Instruction, Multiple Data“ setzt auf Hardwareebene um, was der Name bereits andeutet: Eine Instruktion wird auf verschiedene Daten gleichzeitig angewendet. Bei einem Projekt wie dem Mandelbrot kann diese Prinzip der Parallelisierung auch gut angewendet werden, da die einzelnen Punkte unabhängig voneinander sind.

Für ARMv8-Architektur-Prozessoren wie die Prozessoren des Raspberry Pi 3 B+ oder ODroid existieren zur Implementierung von SIMD Instruktionen in Hochsprachen wie C und C++ sogenannte NEON Compiler Intrinsics<sup>2</sup>.

Diese Intrinsics ermöglichen eine Verwendung der nativen SIMD-Befehle, wobei der Compiler sich um die Verwendung der SIMD-Register kümmert. Dadurch wird lesbarer Code ermöglicht, der sich stärker am zu implementierenden Algorithmus orientiert.

Zu den hier benötigten mathematischen Operationen (z.B. der Addition) wird hierbei das Compiler Intrinsic nach folgendem Schema erzeugt: `vopcq_fpr` mit dem Operationscode *opc* (z.B. *add* für Addition). *v* ist das allgemeine Prefix für Vektoroperationen und *q* bedeutet, dass doppelt so viele Register verwendet werden wie ohne „*q*“. Das Postfix *fpr* bestimmt, dass die Register als Gleitkommazahlen der Präzision *pr* bit (in diesem Fall 32 oder 64) interpretiert werden sollen. Damit werden in jeder Operation 4 mal 32 bit Gleitkommazahlen oder 2 mal 64 bit Gleitkommazahlen verrechnet.

---

<sup>2</sup>Details im Abschnitt ‘Compiler Intrinsics’ unter <https://developer.arm.com/technologies/neon>

## 1.5 TypeScript und React

Um das Frontend zu implementieren, wurde sich für die Programmiersprache TypeScript<sup>3</sup>(Erweiterung von JavaScript um Typisierung) entschieden. Da diese zu JavaScript kompiliert, werden die Vorteile von JavaScript, wie Ausführung im Webbrowser des Benutzers und eine vielzahl verfügbarer Bibliotheken, mit den Vorteilen einer typisierten Programmiersprache vereint.

Für die graphische Benutzeroberfläche wurde ebenfalls TypeScript mit dem React Framework<sup>4</sup> verwendet, welches es ermöglicht, graphische Komponenten nativ in TypeScript zu erstellen und dynamisch zu verändern. Es wurde für jede Komponente eine eigene TypeScript Klasse zu erstellt, welche dann den betreffenden das betreffende Verhalten und dessen Darstellung enthält.

Um das Fraktal darzustellen wird die Leaflet<sup>5</sup> Bibliothek verwendet. Diese, für Onlinekarten konzipierte, Bibliothek stellt den Bereich der komplexen Ebene, auf der die Mandelbrotmenge liegt dar. Dabei wird der momentan sichtbare Ausschnitt der Menge mit Hilfe der WebSockets Verbindung (siehe Unterunterabschnitt 5.7.1) an das Backend versendet und vom ausgewählten Lastbalancierer in Teilregionen unterteilt (siehe Unterabschnitt 5.2). Jeder der vom Backend berechneten Teilregionen wird, sobald diese empfangen wurden, im Frontend angezeigt und die Komponenten zu Visualisierung der Rechenzeit aktualisiert.

---

<sup>3</sup><https://www.typescriptlang.org/>

<sup>4</sup><https://reactjs.org/>

<sup>5</sup><https://leafletjs.com/>

## 2 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.

### 2.1 Problemstellung und Motivation

Das zugrundeliegende Problem ist, dass bei der Lastaufteilung einer unabhängigen Menge von Berechnungen in einem Cluster eine fixe Zuordnung von zu berechnenden 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. Dieser Zustand sollte vermieden werden um Zeit, Kosten und Energie zu sparen.

Dies kann erreicht werden, indem die Einteilung der Rechenbereiche die voraussichtliche Rechendauer berücksichtigt. Dazu werden rechenintensive Bereiche verkleinert und umgekehrt Bereiche mit geringerer Rechenlast vergrößert. Ziel ist, dass alle Knoten für die Bearbeitung in etwa gleich lang brauchen, sodass die gegenseitige Wartezeit minimiert wird. Da nun zu jedem Zeitpunkt möglichst viele Knoten involviert sind wird die Qualität der Parallelisierung deutlich erhöht und die Maximalrechendauer gesenkt.

### 2.2 Didaktische Ziele

Dieses Projekt soll eine Oberfläche bereitstellen mit der Endnutzer intuitiv erfahren können, wie eine Abschätzung der benötigten Rechenlast bei der Aufteilung unabhängiger Berechnungen auf einem Cluster die Gesamtrechendauer deutlich verringern kann. Außerdem soll ersichtlich sein, wie die verwendete Aufteilung bestimmt wird.

Die Qualität der Parallelisierung durch MPI soll zudem noch mit anderen Parallelisierungskonzepten wie OpenMP und SIMD verglichen und vergleichbar gemacht werden.

### 2.3 Qualitätsanforderungen

Die Benutzeroberfläche soll so leicht und intuitiv wie möglich zu bedienen sein. Hierbei wird ein Fokus auf folgende Eigenschaften gelegt:

- Alle Funktionen sollen mit einer minimalen Anzahl an Mausklicks auszuführen, sowie durch eine minimalistisch Designte Benutzeroberfläche erreichbar sein.
  - Zudem soll das System robust gestaltet werden. Dies wird durch die Verwendung von Buttons, Listen, Drop-Down Menüs und Slidern gewährleistet, die die Möglichkeit der Eingabe von ungültigen Werten verhindern.
-

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

## 2.4 Einschränkungen

Die Benutzeroberfläche soll in einem Webbrower 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 einem Rechencluster lauffähig sein.

### 3 Lösungsansatz

#### 3.1 Verwendung der Mandelbrotmenge

Es handelt bei der Bestimmung der Mandelbrotmenge um eine rechenintensive Operation, 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.

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 der Punkte innerhalb der Mandelbrotmenge benötigt wird. Aus didaktischer Sicht ist dies wichtig, um Differenzen zwischen den Balancierungsstrategien spürbar zu machen. Zudem kann die Unterteilung des zu berechnenden Raumes frei gewählt werden, sodass verschiedenste Aufteilungen möglich sind und insbesondere bei der Lastbalancierung frei eingeteilt werden kann.

Als Projekt, das an nicht technisch versierten Anwendern spricht auch der ästhetische Faktor des Mandelbrotfraktales für eine Verwendung. Er kann ein erstes Interesse für die Anwendung wecken.

#### 3.2 Architekturübersicht

Die hohe Rechenintensivität sorgt dafür, dass der Berechnungsteil des Projektes möglichst in einer hardwarenahen Sprache umgesetzt wird. Andererseits sollte die Benutzeroberfläche einfach zu bedienen und auf möglichst vielen verschiedenen Geräten lauffähig sein. Daher wurde sich für eine Unterteilung in ein Frontend, im Browser aufrufbar, und ein Backend, auf einem Cluster laufend und hardwarenah programmiert.

Zudem soll unterschieden werden, zwischen einem Host-Prozess im Backend, welcher für die Kommunikation mit dem Frontend zuständig ist und Workerprozessen, welche die tatsächliche Berechnung der Mandelbrotmenge durchführen. Der Hostprozess nimmt damit Rechenaufträge vom Nutzerfrontend entgegen und führt die Lastbalancierung durch. Die kleineren Regionen der unterteilten Anfrage werden dann an die jeweiligen Workerprozesse gesendet, welche sie berechnen sollen. Damit erhalten wir eine Unterteilung in drei wesentliche Bausteine, wie sie in Abbildung 2 zu sehen sind.

#### 3.3 Lastbalancierung

Um die Effizienz der parallelen Berechnung der Mandelbrotmenge zu erhöhen, sollte die Rechenlast möglichst 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. Damit die Unterschiede zwischen guter und schlechter Lastverteilung deutlich werden, stehen in

---

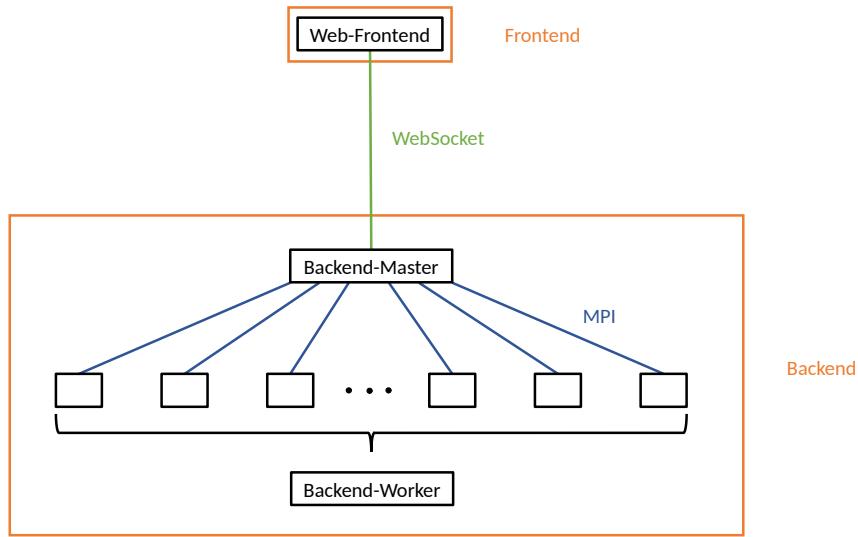


Abbildung 2: Architekturübersicht

diesem Projekt verschiedene Strategien der Lastbalancierung zur Wahl. Die Strategien lassen sich zur Laufzeit austauschen, um einen direkten Vergleich zu ermöglichen.

### 3.3.1 Naive Strategie

Bei der naiven Strategie wird versucht den einzelnen Workern etwa gleich große Teilregionen zuzuweisen. Dies geschieht allerdings ohne Beachtung der eventuell unterschiedlichen Rechenzeiten innerhalb der Teilregionen. Es kann bei Verwendung dieser Strategie also durchaus passieren, dass ein Worker noch rechnet, während alle anderen bereits fertig sind.

### 3.3.2 Strategie mit Vorhersage

Bei dieser Strategie 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 optimale Rechenlast für einen Worker berechnet sich also durch:

$$\frac{\text{Gesamtrechenlast}}{\text{AnzahlWorker}} \quad (2)$$

Wenn die Vorhersage hinreichend exakt ist, kann dieser Wert gut angenähert werden. Dadurch wird die Last gleichmäßiger auf die Worker verteilt als bei der naiven Strategie.

Die Zugehörigkeit eines Punktes zur Mandelbrotmenge wird nach Gleichung 1 iterativ berechnet. Deshalb kann die Anzahl der benötigten Iterationen als Abschätzung der Rechenzeit für diesen Punkt verwendet werden. Zur Anstellung der Vorhersage wird also die angeforderte Region in deutlich geringerer Auflösung berechnet. Dies ist eine gute Annäherung an die tatsächlich Rechendauer, da benachbarte Punkte meist eine ähnliche Anzahl an Iterationen benötigen. Einzig am Rand der Mandelbrotmenge kommt es zu Ungenauigkeiten, weil dort Punkte innerhalb und außerhalb der Menge für die Vorhersage zusammenfallen. Die Genauigkeit der Vorhersage zu erhöhen bedeutet zusätzlichen Rechenaufwand während der Balancierung. Dieser sollte in einem sinnvollen Verhältnis zum Aufwand der Berechnung der Region selbst stehen. Es ist also wichtig eine Balance zwischen Güte und Geschwindigkeit der Vorhersage zu finden.

### 3.3.3 Implementierungsvarianten

Sowohl die naive Strategie als auch die Strategie mit Vorhersage lassen sich in zwei Varianten umsetzen. Man kann hierbei einen ganzheitlichen oder einen rekursiven Ansatz wählen. Bei ersterem wird die gesamte Region in einem Schritt in die gewünschte Anzahl an Teilregionen geteilt. Dazu werden Zeilen und Spalten gebildet. Die Grundidee eines rekursiven Ansatzes ist es das Problem so lange in einfachere Teilprobleme aufzuteilen, bis die Lösung offensichtlich ist (Basisfall). Hier ist der Basisfall die Aufteilung einer Region auf genau einen Worker. Um diesen zu erreichen wird die Region solange halbiert, bis genug Teilregionen für jeden Worker entstanden sind.

Wo die Grenzen zwischen den Zeilen und Spalten (nicht-rekursiv) bzw. den Hälften (rekursiv) liegen, wird von der zugrundeliegenden Lastbalancierungsstrategie bestimmt.

## 3.4 Konzept der ausgetauschten Nachrichten

Um eine zwischen den unabhängigen Systemen eine einheitliche Kommunikation zu ermöglichen, wurde ein Protokoll spezifiziert um Flächen in der komplexen Ebene und ihre Auflösung eindeutig zu bestimmen. Der grobe Inhalt und die Richtung der Nachrichten ist Abbildung 3 zu entnehmen, die exakte Spezifikation in der jeweiligen Sprache ist den angegebenen Dateien zu entnehmen.

Ein Beispiel für eine gültige Regionsanfrage ist in Quelltext 1 zu finden. Einerseits wird hierbei eine Region in komplexen Koordinaten beschrieben, wobei der obere linke Punkt ( $maxImag, minReal$ ) und der rechte untere Punkt ( $minImag, maxReal$ ) in der komplexen Ebene einen zu berechnenden Bereich aufspannen. Da die reelle Ebene jedoch beliebig genau aufgelöst werden kann, muss zudem noch die Anzahl an Pixeln pro Seite des Rechteckes definiert werden, `width` und `height`. Wie in Abbildung 4 zu sehen, ist zudem der horizontale Offset und vertikale Offset die linke obere Koordinate der Region bezüglich der gesamten sichtbaren Anfrage in Pixeln (diese Werte gewinnen in den Regionsaufteilungen an Bedeutung). Der Wert `validation` ist technisch gesehen nicht mehr notwendig, wird aber mit dem Zoom-wert der Leafletkarte gefüllt um zu vermeiden dass Regionsdaten von zuvor berechneten Regionen falsch verwendet werden.

---

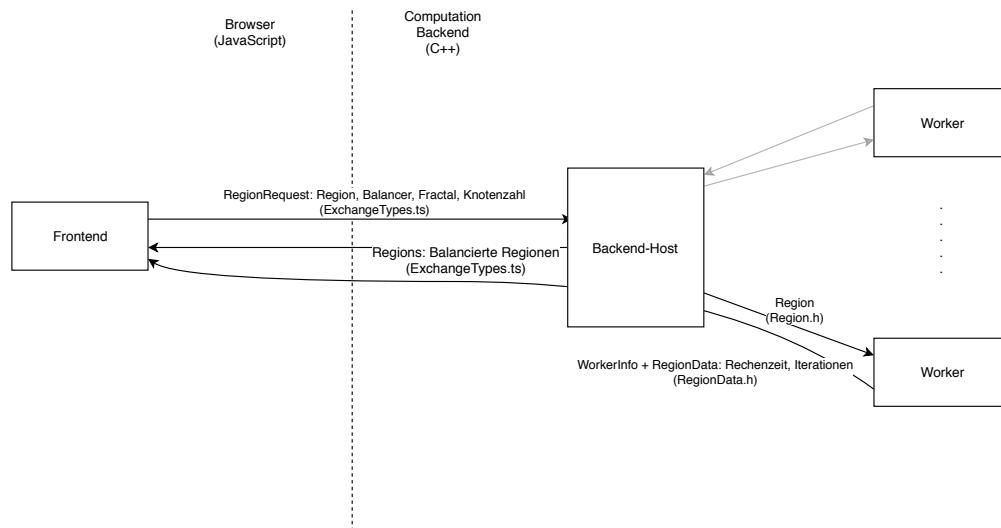


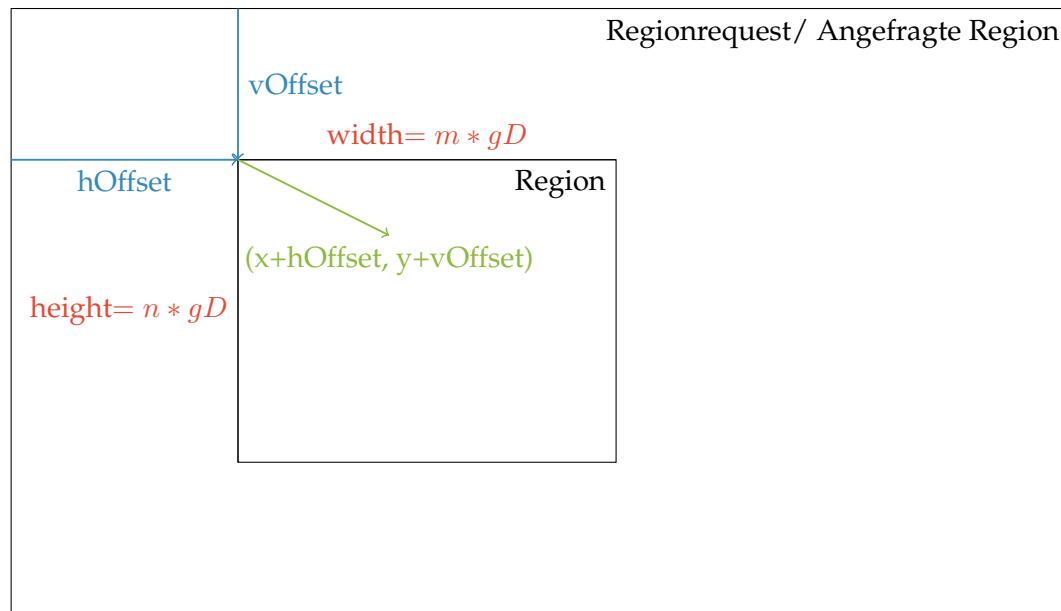
Abbildung 3: Konzept der versendeten Nachrichten. Die genauen Definitionen der Nachrichten sind in den angegebenen Dateien nachzusehen.

```

1 {
2     "type": "regionRequest",
3     "region": {
4         "minReal": -0.251953125,
5         "maxImag": -0.8388671875,
6         "maxReal": -0.2216796875,
7         "minImag": -0.8505859375,
8         "width": 1984,
9         "height": 768,
10        "hOffset": 0,
11        "vOffset": 0,
12        "validation": 8,
13        "guaranteedDivisor": 64,
14        "maxIteration": 1019
15    },
16    "balancer": "naiveRecursive",
17    "fractal": "mandelbrot"
18}

```

Quelltext 1: Eine gültige Anfrage einer Region in JSON



$$gD = \text{guaranteedDivisor}, n, m \in \mathbb{N}^+, x < width, y < height$$

Abbildung 4: Konzept der Koordinaten in den Regionsobjekten. Alle Koordinaten beziehen sich auf die Darstellungsebene und sind daher in Pixeln.

In zurückkehrenden `RegionData`-Nachrichten sind Arrays der berechneten Iterationszahlen eingebunden. Dabei wird der Punkt  $(x, y)$  in der gesendeten Region (Punkt  $(x+hOffset, y+vOffset)$  in der angefragten Region) im Datenarray an Index  $i = x + y * width$  gespeichert.

## 4 Installation der Anwendung

Um das System zu installieren, muss das Repository mit git<sup>6</sup> lokal geklont werden. Dabei werden die Quelldateien für das Front- sowie Backend heruntergeladen.

```
1 git clone https://gitlab.lrz.de/lrr-tum/students/eragp-mandelbrot.git
```

Quelltext 2: Klonen des Repositorys

### 4.1 Lokales Backend

Eine lokale Installation des Backends zu Entwicklungszwecken ist durch einen Docker<sup>7</sup> Container möglich. Dieser bietet eine ähnliche Umgebung zu der des Clusters und ermöglicht schnellere Feedbackzyklen.

```
1 # Systemabhängige Installation der Docker Anwendung
2 $ sudo apt install docker
3 cd backend/ && ./run_docker.sh
4 Starting the Build Process
5 ...
6 Host: Core 37 ready!
7 # ^C beendet das Backend und verbindet sich mit der shell des Containers
8 ^C[mpiexec@9cc2d5ac2cd1] Sending Ctrl-C to processes as requested
9 [mpiexec@9cc2d5ac2cd1] Press Ctrl-C again to force abort
10 # exit schliesst die shell des Containers
11 root@9cc2d5ac2cd1:~/eragp-mandelbrot/backend# exit
```

Quelltext 3: Starten der Entwicklungsumgebung des Backends

Das run\_docker.sh Skript lädt das benötigte Basis Image, welches alle benötigten Bibliotheken bereits enthält, herunter und erstellt basierend darauf den Entwicklungscontainer. In diesen werden dann die aktuellen Quelldateien hinein kopiert und kompiliert, wonach das Backend mit Adresse ws://localhost:9002 gestartet wird.

### 4.2 Backend auf HimMUC Cluster

[Der] HimMUC ist ein flexibler Cluster von ARM-Geräten, bestehend aus 40 Raspberry Pi 3 sowie 40 ODroid C2 Single-Board-Computers (SBC).<sup>8</sup>

**Schnellstart** Um das Programm auf dem HimMUC Cluster zu starten, wurde ein Python Skript erstellt, das alle notwendigen Schritte übernimmt. Es führt die Befehle aus Unterabschnitt 8.1 aus, es kann daher bei Problemen zur Fehlerbehebung herangezogen werden.

---

<sup>6</sup><https://git-scm.com/>

<sup>7</sup><https://www.docker.com/>

<sup>8</sup><http://www.caps.in.tum.de/himmuc/>

Stellen sie zunächst sicher, dass sie ein Konto mit Zugangsberechtigungen auf dem HimMUC Cluster besitzen. Um den eigenen Quellcode auf dem Cluster zu kompilieren muss für die korrekte Funktionsweise des Skriptes zudem ihr SSH-Key auf dem Cluster abgelegt sein<sup>9</sup>.

Außerdem sollten folgende Programme lokal installiert sein:

- rsync
- ssh
- python3 (3.5 oder neuer)

Starten sie anschließend aus dem Ordner backend/ den Befehl aus Quelltext 4

```
1 python3 himmuc/start_himmuc.py <Rechnerkennung> <Anzahl Prozesse> <Anzahl Rechenknoten>
```

Quelltext 4: Start der Entwicklungsumgebung auf dem HimMUC

Das Ergebnis wird ähnlich zu Quelltext 5 aussehen. Details zu weiteren Optionen des Skripts sind via --help verfügbar. Für eine detailliertere Beschreibung der Installation auf dem „HimMUC“ Cluster, siehe Unterabschnitt 8.1

### 4.3 Installation des Frontends

Das Frontend ist in TypeScript<sup>10</sup> (Erweiterung von JavaScript<sup>11</sup>) geschrieben und kann somit auf einem beliebigen Endgerät mit einem modernen Webbrowser ausgeführt werden. Um eine Version lokal zu starten, muss die Paketverwaltung npm<sup>12</sup> installiert werden. Diese verwaltet alle für das Frontend benötigten Bibliotheken und installiert diese lokal.

Das Kommando npm start startet dabei einen lokalen WebServer, welcher eine kompilierte Version des Frontends unter der Adresse http://localhost:3000 anbietet. Danach wird der Standardwebbrowser des Systems verwendet, um diese URL zu öffnen.

Das Frontend verbindet sich automatisch mit der lokalen Adresse ws://localhost:9002. Dies kann über den Get-Parameter ws in der URL auf die Adresse des eigenen Backends gesetzt werden.

---

<sup>9</sup>siehe ssh-copy-id

<sup>10</sup><https://www.typescriptlang.org/>

<sup>11</sup><https://en.wikipedia.org/wiki/JavaScript>

<sup>12</sup><https://www.npmjs.com/>

```

1 $ eragp-mandelbrot/backend$ python3 himmuc/start_himmuc.py muendler 10 9
2 Uploading backend... sending incremental file list
3 backend/himmuc/start_backend.py
4           3,897 100%    3.05MB/s   0:00:00 (xfr#1, to-chk=35/62)
5 done
6 Start mandelbrot with 1 host and 9 workers on 9 nodes... started
7     mandelbrot
8 Search host node... srun: error: Could not find executable worker
9 odr00 found
10 Establish port 9002 forwarding to host node odr00:9002 ... established
11 System running. WebSocket connection to backend is now available at
12     ws://himmuc.caps.in.tum.de:9002
13 Press enter (in doubt, twice) to stop Warning: Permanently added the
14     ED25519 host key for IP address '10.42.0.54' to the list of known
15     hosts.
16 # Enter
17
18 Stopping port forwarding... stopped (-9)
19 Stopping mandelbrot host and workers... stopped (-9)

```

Quelltext 5: Beispielausgabe bei Start der Entwicklungsumgebung auf dem HimMUC

```

1 # Systemabhängige Installation der npm Paketverwaltung
2 sudo apt install npm
3 # Installiert benötigte Bibliotheken und startet WebServer
4 cd frontend/ && npm install ; npm start
5 ...
6 Version: webpack 4.25.1
7 Time: 7230ms
8 Built at: 12/28/2018 10:48:32 PM
9          Asset      Size  Chunks      Chunk Names
10        index.html  1.65 KiB          [emitted]
11  mandelbrot.js  11.7 MiB  main  [emitted]  main
12    style.css    519 KiB  main  [emitted]  main
13 Entrypoint main = style.css mandelbrot.js
14 ...

```

Quelltext 6: Starten des Frontends mit beispielhafter Ausgabe

## 5 Dokumentation der Implementierung

### 5.1 Implementierung des Backends

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.

#### 5.1.1 Inkludierte Header und CMake Anweisungen

Sämtliche Headerdateien sind ohne Untergruppierung im Ordner `include` des Projektes abgelegt.

Sie werden zusammen mit den Header-Bibliotheken `rapidjson` und `websocketpp` sowie der vorkompilierten Bibliothek `boost_static` von CMake eingebunden um das Projekt zu bauen.

Für einen erfolgreichen Build wird CMake einer Version von mindestens 3.7.0 und die C++11 Standards vorrausgesetzt. Ergebnis des Builds sind die ausführbaren Dateien `host` und `worker` welche die beschriebenen Funktionen innerhalb des Backends umsetzen.

Eine detailliertere Beschreibung ist im Anhang zu finden.

#### 5.1.2 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 20. 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.

### 5.2 Implementierung der Lastbalancierung

Der Implementierung der Lastbalancierung liegen die in Unterabschnitt 3.3 beschriebenen Konzepte zugrunde. Wichtig bei der Umsetzung dieser ist, dass der garantierte Teiler (`guaranteedDivisor`) von Höhe und Breite der Teilregionen dem der angeforderten Region entspricht (vgl. Abbildung 4). Eine Tile (ein Bereich mit Breite und Höhe gleich dem garantierte Teiler<sup>13</sup>, die Bezeichnung *Tile* kommt aus dem Frontend) muss also als atomare Einheit betrachtet werden, da es sonst im Frontend zu Schwierigkeiten bei der Darstellung kommt.

Die Klassenstruktur der Lastbalancierer entspricht dem Strategy-Pattern (vgl. [1] S. 315-323). So kann der Balancierer zur Laufzeit leicht gewechselt werden und auch die

---

<sup>13</sup>Eine Region lässt sich also immer in eine ganzzahlige Anzahl von Tiles aufteilen, vgl. Abbildung 6

Erweiterung des Projekts um eine weitere Strategie gestaltet sich einfach. Was dabei genau beachtet werden muss findet sich im Teil Erweiterung (5.2.3).

### 5.2.1 Naive Strategie

Die naive Strategie (`NaiveBalancer` und `RecursiveNaiveBalancer`) lässt sich in beiden Varianten recht einfach nach dem oben beschriebenen Konzept umsetzen. Zusätzlich wurde noch `ColumnBalancer` implementiert, dabei handelt es sich um eine Variante des nicht-rekursiven Ansatzes, die nur Spalten erzeugt. Die Erhaltung des garantierten Teilers wird dadurch erreicht, dass die Höhe und Breite der Teilregionen auf ein Vielfaches dieses Teilers gesetzt werden. Diese können bei der nicht-rekursiven Variante vor der eigentlichen Aufteilung bestimmt werden. Da sich  $\frac{\text{width}}{\text{guaranteedDivisor}}$  nicht unbedingt durch die Anzahl an Workern teilen lässt, kann es sein, dass einige Teilregionen um `guaranteedDivisor` Pixel breiter sind. Selbiges gilt für die Höhe.

Bei der rekursiven Variante wird mithilfe folgender Kriterien entschieden, ob horizontal oder vertikal geteilt wird:

- Region vertikal oder horizontal nicht mehr teilbar (d.h. `width` bzw. `height`  $\leq \text{guaranteedDivisor}$ ): Teile in die andere Richtung.
- Region vertikal und horizontal unteilbar: Erzeuge eine leere Region für die zweite Hälfte.
- Sonst: Teile parallel zur kürzeren Seite. Dies gibt dem Lastbalancierer mehr Möglichkeiten die Trennlinie zwischen den Teilregionen zu setzen, was zu einer genauereren Teilung führt.

Die Teilung an sich funktioniert wie die nicht-rekursive Aufteilung auf zwei Worker. Der Rekursionskontext (`struct BalancingContext`) wurde extern definiert, da dieser für die Strategie mit Vorhersage wiederverwendet wird.

### 5.2.2 Strategie mit Vorhersage

Auch diese Strategie (`PredictionBalancer` und `RecursivePredictionBalancer`) folgt den oben beschriebenen Konzept. Allerdings wurde die Berechnung der Vorhersage in eine eigene Klasse ausgelagert, da die Berechnung der Vorhersage für die rekursive und die nicht-rekursive Variante gleich ist. Die Struktur der Vorhersage sorgt auch dafür, dass der garantierte Teiler erhalten bleibt. Für die Berechnung der Vorhersage ist es notwendig, dass bei der Erstellung eine Referenz auf ein `Fractal`-Objekt (siehe 5.4) erhalten.

**Bestimmung der Vorhersage** Die Vorhersage (`struct Prediction`) wird von der Klasse `Predictor` angestellt. Dazu wird die Region in einer sehr viel geringeren Auflösung berechnet. Die benötigte Anzahl an Iterationen wird jeweils pro Tile 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:

- $\text{predictionAccuracy} > 0$ :  $(\text{predictionAccuracy})^2$  Pixel werden pro Tile berechnet. Die Summe der Iterationen für die einzelnen Pixel ergibt die Vorhersage für die Tile.
- $\text{predictionAccuracy} < 0$ : Für  $(\text{predictionAccuracy})^2$  Tiles wird ein Pixel in der Vorhersage berechnet. Es erhalten also mehrere Tiles dieselbe Vorhersage.
- $\text{predictionAccuracy} = 0$ : Unzulässig, es wird ein Null-Pointer zurückgegeben.

Zusätzlich beinhaltet die Vorhersage die Summen der benötigten Iterationen pro Spalte und Zeile, sowie die Gesamtsumme. So wird vermieden, dass diese während des Balancierens immer neu berechnet werden müssen.

**Nicht-rekursive Variante** Für die nicht-rekursive Aufteilung wird die Region erst in Spalten aufgeteilt und in einem zweiten Schritt wird dann die horizontale Unterteilung in Teilregionen vorgenommen. Da die beiden Schritte analog zueinander sind wird hier nur das Aufteilen in Spalten anhand des Pseudocodes in Quelltext 7 beschrieben.

Die optimale Rechenlast pro Spalte (`desiredN`) berechnet sich nach Gleichung 2 wobei die Anzahl der Worker durch die Anzahl an Spalten ersetzt werden muss. Als Abschätzung der Gesamtrechenlast wird die Gesamtsumme der Vorhersage verwendet. Eine Spalte, die als leere Region beginnt, wird nun solange um eine Spalte von Tiles vergrößert bis die optimale Rechenlast erreicht oder überschritten wird. Dazu müssen die Spaltensummen der Vorhersage aufaddiert werden. Um Lücken auszuschließen werden die Grenzen der Spalten explizit aufeinander gesetzt, d.h. bei zwei benachbarten Teilregionen entspricht die obere Grenze der Einen der unteren Grenze der Anderen. Damit es auch für die Aufteilung in Teilregionen eine Vorhersage gibt, wird eine Kopie der Vorhersage erstellt, welche nur die Werte für die aktuelle Spalte enthält (nicht im Pseudocode).

Im eigentlichen Code wurden ein paar Optimierungen am Pseudocode vorgenommen. Um float-Ungenauigkeiten beim wiederholten Addieren zu minimieren werden die Werte in `cur` nicht aufaddiert, sondern aus den Zählern berechnet. Außerdem werden die Spalten ohne Zwischenspeicherung in die nötigen Teilregionen aufgeteilt. Zusätzlich wird `desiredN` nach jeder abgeschlossenen Spalte für die Verbleibenden neu berechnet, da es sehr unwahrscheinlich ist, dass dieser Wert genau erreicht wird.

**Rekursive Variante** Die rekursive Variante der Strategie mit Vorhersage verwendet dasselbe Rekursionsschema wie ihr naives Gegenstück. Also wird auch die Entscheidung, ob horizontal oder vertikal geteilt werden soll, auf die gleiche Art und Weise gefällt. Der Unterschied zwischen den beiden Strategien liegt also hauptsächlich in den beiden Methoden zur Aufteilung.

Bei dieser Strategie werden die Regionen nicht einfach halbiert, sondern in zwei Teile aufgeteilt, die laut der Vorhersage ähnlich rechenintensiv sind. Dazu wird ähnlich

```

1  def balanceLoad (region, nodeCount)
2      # Will be per tile
3      prediction = sampleFractal(region)
4      cols = computeColCount(nodeCount)
5      deltaRes = region.guaranteedDivisor
6      deltaReal = deltaReal(region)
7      desiredN = computeOptimalLoad(prediction, cols)
8      cur = region
9      curN = 0
10     colsMade = 0
11     for i = 0 to prediction.cols.vectorLength
12         if colsMade + 1 == cols
13             cur = rest of region # Part of region thats not already
14                 assigned to a col
15             result.append(cur)
16             break
17             curN += prediction.cols[i]
18             cur.width += deltaRes
19             cur.maxReal += deltaReal
20             # Make sure that each col has at least width = deltaRes
21             if curN >= desiredN OR prediction.cols.length - i - 1 < cols
22                 result.append(cur) # Copy of cur
23                 cur.minReal = cur.maxReal
24                 cur.width = 0
25                 curN = 0
26                 colsMade++
27                 continue
28     return splitColsInParts(result)

```

Quelltext 7: Aufteilung in Spalten im Pseudocode

vorgegangen, wie bei der nicht-rekursiven Variante für die Aufteilung auf zwei Worker. Deshalb profitiert diese Strategie auch besonders davon, dass immer parallel zur kürzeren Seite geteilt wird. Die Vorhersage ist in diese Richtung nämlich feingliedriger, da weniger Tiles der Vorhersage zu einer Spalte bzw. Zeile zusammengefasst werden müssen als wenn parallel zur längeren Seite aufgeteilt werden würde. Allerdings wird hier, wenn möglich, sichergestellt, dass beide Teilregionen groß genug sind um jeweils auf die Hälfte der Worker aufgeteilt zu werden, da ansonsten unnötigerweise leere Regionen entstehen. 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.

### 5.2.3 Erweiterung

Da die Lastbalancierung nach dem Strategy-Pattern realisiert ist, gestaltet sich die Erweiterung um eine neue Balancierungsstrategie recht einfach. Zuerst muss eine Unterklasse von Balancer (Quelltext 8) erstellt werden, um ein gemeinsames Interface zu erzwingen und somit die polymorphe Nutzung der neuen Klasse zu ermöglichen.

Danach wird die neue Strategie über die Methode `BalancerPolicy::chooseBalancer`

```

1 class Balancer {
2     public:
3         virtual Region* balanceLoad(Region region, int nodeCount) = 0;
4         virtual ~Balancer();
5 };

```

Quelltext 8: Das gemeinsame Interface der Lastbalancierung

verfügbar gemacht. Dazu muss sie mithilfe der statischen Variablen Klassenname::NAME benannt werden. BalancerPolicy muss nun so erweitert werden, dass, bei Eingabe des vorher festgelegten Namens, ein neues Objekt der entsprechenden Klasse zurückgegeben wird. In BalancerPolicy kann auch die Genauigkeit der Vorhersage für die oben aufgeführten Strategien festgelegt werden.

**Bedingungen an Balancer::balanceLoad** Bei der Eingabe von region und nodeCount erfüllt ein korrekter Rückgabewert subregions die folgenden Bedingungen:

- subregions ist ein Zeiger auf ein Array mit nodeCount Elementen vom Typ Region
- Die Regionen in subregions sind eine Partitionierung von region, d.h. sie überschneiden sich nicht und ihre Vereinigung ergibt genau region
- Für jede Region subregion in subregions gilt:
  - guaranteedDivisor, validation, maxIteration und fractal sind in region und subregion gleich
  - subregion.guaranteedDivisor teilt subregion.width und subregion.height ohne Rest
  - subregion.hOffset und subregion.vOffset sind so gesetzt, dass sie den Abstand der oberen linken Ecke von subregion zur oberen linken Ecke von region in Pixeln angeben
  - Die Deltas für die komplexen Werte sind unverändert, d.h. die Größe des Bereiches der komplexen Ebene, der von einem Pixel abgedeckt wird, ist unverändert

Ob diese Bedingungen erfüllt sind kann mit dem Test in BalancerTest überprüft werden. Dazu muss der neue Testfall (struct TestCase) durch Angabe von Name, Anzahl an Workern, Balancierungsstrategie und Testregion spezifiziert werden. Dann kann er an der im Quellcode markierten Stelle zum Vektor der Testfälle hinzugefügt werden. Anschließend muss der Test mittels cmake in backend/tests neu kompiliert werden.

### 5.3 Kommunikation zwischen Host und Worker im Backend

Zur Kommunikation zwischen dem Host und den Workern im Backend wird ausschließlich das Message Passing Interface (MPI) genutzt. Als MPI-Implementierung wird

OpenMPI<sup>14</sup> verwendet, da dies sehr verbreitet ist und auch die Nutzung von Threads unterstützt, was laut der offiziellen MPI-Dokumentation<sup>15</sup> ein optionales Feature ist.

### 5.3.1 Beschreibung der verwendeten MPI-Funktionen

In der Implementierung kommen folgende MPI-Funktionen zum Einsatz:

- Initialisierung der MPI-Umgebung mit dem passenden Thread-Level

Die MPI-Umgebung wird mit einem Aufruf von `MPI_Init_thread()` initialisiert, wobei das als Parameter übergebene Thread-Level mit dem Thread-Modell des eigenen Programms zusammen passen muss. In Abhängigkeit vom verwendeten Thread-Level kann die MPI-Implementierung verschiedene Performanceoptimierungen durchführen.

- MPI Sendeoperationen

- Blockierende Sendeoperation

Mithilfe der blockierenden Sendeoperation (`MPI_Send()`) lassen sich Nachrichten variabler Länge übertragen wobei die Funktion erst zurückkehrt, wenn die Daten komplett an MPI übergeben sind, d.h. der Sendepuffer vollständig ausgelesen wurde.

- Nicht-blockierende Sendeoperation als Persistent Communication Request

Der Persistent Communication Request (`MPI_Send_init()` und `MPI_Start()`) ist nur für die Übertragung von Nachrichten mit exakt gleicher Konfiguration geeignet. Insbesondere sind dies Länge der Nachricht, verwendeter Datenpuffer, Übertragungstyp, Nachrichtentag und Empfänger. Dadurch wird der Overhead für die Kommunikation mit dem MPI-Kommunikationscontroller reduziert und damit eine Performanceoptimierung erreicht. Der Übertragungsmodus ist bei dem Persistent Communication Request immer nicht-blockierend.

- MPI Empfangsoperationen

- Test, ob Daten Empfangen werden können

Die Funktion `MPI_Probe()` prüft, ob Daten zum Empfangen bereit sind und stellt Statusinformationen (z.B. Länge der Nachricht) über die Nachricht bereit. Diese Informationen sind für die eigentlichen Empfangsoperationen wichtig. Diese Funktion gibt es in einer blockierenden (`MPI_Probe()`) und nicht-blockierenden (`MPI_Iprobe()`) Ausprägung.

- Blockierende Empfangsoperation

Die blockierenden Empfangsoperation (`MPI_Recv()`) verhält sich analog zur blockierenden Sendeoperation.

---

<sup>14</sup><https://www.open-mpi.org/doc/v3.1/>

<sup>15</sup><https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

- Nicht-blockierende Empfangsoperation als Persistent Communication Request

Analog der Sendeoperation Persistent Communication Request gibt es eine nicht-blockierende Empfangsoperation (`MPI_Recv_init()` und `MPI_Start()`) für den Empfang von Nachrichten fester Größe mit identischer Konfiguration. Der Übertragungsmodus ist immer nicht-blockierend. Die Verwendung ist unabhängig von der Sendeoperation mit der die Daten übertragen werden.

- Abschluss nicht-blockierender Kommunikation

Nicht-blockierende MPI-Kommunikationsoperationen müssen immer mit einem Aufruf von `MPI_Wait()` oder `MPI_Test()` bzw. deren abgeleiteten Funktionen (z.B. `MPI_Waitall()`) abgeschlossen werden.

- `MPI_Test()` prüft nicht-blockierend ob eine Sende- oder Empfangsoperation abgeschlossen ist.
- `MPI_Wait()` wartet blockierend auf den Abschluss einer Sende- oder Empfangsoperation.

### 5.3.2 MPI-Designentscheidungen

Im Folgenden wird die Kommunikation zwischen dem MPI-Host-Thread und den MPI-Worker-Threads beschrieben. Zur Veranschaulichung dient folgendes Sequenzdiagramm:

Wie oben (TODO Link) beschrieben, werden über die WebSocketverbindung und den WebSocket-Recv-Thread die zu berechnenden Subregionen in der gemeinsamen Datenstruktur `WebSocket_Request_To_MPI` abgelegt. Aufgabe des MPI-Host-Threads ist es, die einzelnen Subregionen an die verfügbaren MPI-Worker-Threads zu übertragen, die Berechnungsergebnisse von den MPI-Worker-Threads entgegen zu nehmen und in der gemeinsamen Datenstruktur `MPI_To_WebSocket_Result` abzulegen, sodass der WebSocket-Result-Thread die Daten an das Frontend übertragen kann.

Wird während einer laufenden Berechnung eine neue Berechnung angefordert, so werden die auf den Workern noch laufenden Berechnungen abgebrochen und mit der neuen Berechnung begonnen. Ein entsprechendes Sequenzdiagramm ist im Anhang (TODO Link) zu finden.

Die Kommunikation soll performanceoptimiert und parallelisiert erfolgen.

Für die Implementierung der MPI-Kommunikation wurden daher folgende Designentscheidungen getroffen:

- Initialisierung der MPI-Umgebung mit `MPI_THREAD_FUNNELED`
-

In der gewählten Architektur arbeiten sowohl Host als auch Worker mit mehreren Threads, wobei nur der Hauptthread MPI-Aufrufe tätigt. Für die MPI-Konfiguration bedeutet dies, dass sie mit MPI\_THREAD\_FUNNELED initialisiert werden muss. Durch die Einschränkung auf einen MPI-Thread ist MPI in der Lage Optimierungen durchzuführen, was der Performance der Kommunikation zu Gute kommt. Es wurde sich bewusst gegen den Einsatz von MPI in mehreren Threads (MPI\_THREAD\_MULTIPLE) entschieden, da OpenMPI hierfür zwar eine korrekte aber nicht performanceoptimierte Implementierung anbietet<sup>16</sup>.

- Datenübertragung als uninterpretierte Byte-Datenströme

Alle per MPI übertragenen Daten werden uninterpretiert als Byte-Datenströme geschickt. Das bedeutet, dass eine erfolgreiche Interpretation der Daten nur auf einem Cluster völlig gleichartiger Rechner garantiert werden kann. Allerdings macht dies die Übertragung denkbar einfach und performant, da im gesamten Backend mit den selben Structs ohne eine Umformatierung gearbeitet werden kann.

- Nicht-blockierendes Senden der Subregionen vom Host an die Worker

Der MPI-Host-Thread verwendet Persistent Communication Requests für das performante Senden der Arbeitsaufträge zu den Workern, da hier die Nachrichtengröße konstant ist und die Sendeparameter identisch sind. Der gewählte Übertragungsmodus ist nicht-blockierend, da dieser Modus es erlaubt, alle Sendeanoperationen an alle Worker zu starten und erst danach auf deren Abschluss mit einem MPI\_Waitall() zu warten. Dies gewährleistet, dass die Subregionen möglichst schnell an die Worker verteilt werden und diese somit mit minimaler Verzögerung die Berechnung beginnen können. Dieser Aufruf ist in der Grafik mit "MPI\_PCR\_Start()" bezeichnet.

- Nicht-blockierendes Empfangen der vom Host gesendeten Subregionen in den Workern

Analog zum MPI-Host-Thread nutzt auch der MPI-Worker-Thread Persistent Communication Requests um eine Subregion möglichst performant, d.h. mit möglichst wenig Overhead, zu empfangen. Es wurde der nicht-blockierende Übertragungsmodus gewählt, um laufende Berechnungen abbrechen zu können, falls ein neuer Rechenauftrag empfangen wurde (siehe Implementierung der MPI-Kommunikation im Worker).

- Struktur der von den Workern an den Host gesendeten Daten

Es müssen generelle Informationen in Form eines WorkerInfo-Structs (allgemeine Daten zur Subregion, Nummer des Workers, Berechnungsdauer) und ein Array der berechneten Daten von den Workern an den Host gesendet werden. Da die

---

<sup>16</sup>siehe Abschnitt 'MPI\_THREAD\_MULTIPLE Support' unter [https://www.open-mpi.org/doc/v3.1/man3/MPI\\_Init\\_thread.3.php](https://www.open-mpi.org/doc/v3.1/man3/MPI_Init_thread.3.php)

Anzahl der berechneten Daten und damit die Länge des Datenarrays variiert, kann das Array nicht ohne weiteres Teil des WorkerInfo-Structs sein. Um alle Daten mit einer Übertragungsoperation abzuarbeiten, wird das Datenarray direkt hinter das WorkerInfo-Struct kopiert.

- Blockierendes Senden der berechneten Subregion von den Workern an den Host  
Diese flexible Operation wird für das Versenden der Rechenergebnisse vom Worker zum Host verwendet, da die zu übertragende Datenmenge von der Größe der berechneten Region abhängt. Nicht-blockierendes Senden ist hier nicht notwendig, da das Ergebnis nach der abgeschlossenen Berechnung in jeden Fall an den Host gesendet werden soll, bevor eine neue Berechnung startet. Im Sequenzdiagramm ist dieser Aufruf mit "MPI\_Send()" bezeichnet.
- Blockierendes Empfangen der von den Workern berechneten Subregionen im Host  
Wenn Daten von den Workern zu empfangen sind (dies wird nicht-blockierend mit MPI\_Iprobe() getestet), werden die Daten blockierend Empfangen und sofort im richtigen Format in die MPI\_To\_WebSocket\_Result Datenstruktur abgelegt. Eine nicht-blockierende Empfangsoperation bietet in diesem Anwendungsfall keine Vorteile.
- Busy Waiting

- Busy waiting im MPI-Host-Thread

Im Host ist busy waiting unumgänglich, da einerseits auf neue Arbeitsaufträge und andererseits auf Rechenergebnisse reagiert werden muss. Um busy waiting zu umgehen, wäre eine Lösung, einen Sendethread und einen Empfangsthread zu implementieren, was aber die MPI-Kommunikation insgesamt verlangsamen würde, da ein höherer Isolation-Level gesetzt werden muss (siehe Initialisierung der MPI-Umgebung mit MPI\_THREAD\_FUNNELED). Um die CPU zu entlasten, wird der MPI-Host-Thread für 100 Mikrosekunden schlafen gelegt, falls im aktuellen Schleifendurchlauf weder eine Sendeoperation noch eine Empfangsoperation durchgeführt wurde. Diese Pause hat keine signifikante Auswirkung auf die Gesamtperformance aufgrund der im Vergleich deutlich längeren Berechnungsdauer einer Subregion.

- Busy waiting im MPI-Worker-Thread

Solange keine Berechnungen durchzuführen sind, wäre es ausreichend blockierend auf das Eintreffen neuer Berechnungsaufträge zu warten, um die CPU nicht unnötig zu beladen. Da die blockierende Empfangsoperation MPI\_Recv() aber selbst mit busy waiting implementiert ist, wird auf deren Nutzung verzichtet. Stattdessen wird der MPI-Worker-Thread im Schleifendurchlauf für 1 Millisekunde schlafen gelegt wenn keine Nachrichten zu empfangen sind. Im Vergleich zur Berechnungsdauer einer Subregion (vgl. Evaluationstimer ... TODO) fällt diese 1 Millisekunde nicht ins Gewicht, entlastet aber spürbar die CPU.

---

### 5.3.3 Beschreibung der implementierten MPI-Hauptroutinen

In diesem Abschnitt werden die Implementierungen der Hauptroutinen des MPI-Host-Threads und des MPI-Worker-Threads genauer beschrieben.

**Ablauf der MPI-Kommunikation im Host** Im folgenden wird das Kommunikationsverhalten des MPI-Host-Thread genauer beschrieben.

Die Kommunikation zwischen Host (MPI-Host-Thread) und Workern (MPI-Worker-Thread) ist im MPI-Host-Thread als Endlosschleife (busy waiting) implementiert. Die dabei stattfindenden Operationen inklusive ausführlicher Kommentare sind in Quelltext 9 einzusehen.

```

1 while true
2   # Check for new computation tasks (subregions) in shared datastructure
3   if websocket_request_to_mpi not empty
4     # Send subregions to worker using non-blocking PCR
5     MPI_Pcr_start(subregions)
6   # Check non-blocking for incoming computation results from worker
7   if MPI_Iprobe() is true
8     # Receive computation result using blocking receive operation
9     MPI_Recv(result)
10    # Store result in shared datastructure
11    mpi_to_websocket_result.add(result)

```

Quelltext 9: MPI Kommunikation im Host in Pseudocode

**Ablauf der MPI-Kommunikation im Worker** Im folgenden wird das Kommunikationsverhalten des MPI-Worker-Threads genauer beschrieben.

Analog zum Host ist auch der Worker (MPI-Worker-Thread) als Endlosschleife mit busy waiting implementiert. Die Berechnung einer Subregion erfolgt dabei zeilenweise durch einen compute() Aufruf. Der entsprechende Pseudocode inklusive ausführlicher Kommentare ist in Quelltext 10 zu finden.

## 5.4 Berechnung der Mandelbrotmenge

Die Berechnung der einzelnen Punkte der Mandelbrotmenge ist in eine eigene Klasse ausgelagert. Wie bei der Lastbalancierung sind die Klassen nach dem Strategy-Pattern strukturiert, um eine spätere Erweiterung um andere Fraktale zu vereinfachen. Das Interface der Strategien ist hier in der Klasse `Fractal` (Quelltext 11) definiert.

Zusätzlich beinhaltet `Fractal` statische Methoden zur Berechnung der Deltas für Real- und Imaginärteil. Diese geben die Größe des Bereichs der komplexen Ebene an, der von einem Pixel überdeckt wird. So berechnet sich zum Beispiel das reelle Delta wie folgt:

$$\text{deltaReal} = \frac{\maxReal - \minReal}{\text{width}}$$

Die Berechnung des imaginären Deltas erfolgt analog. Die Deltas werden u.a. für die Berechnung des Fraktals in den Workern verwendet.

```

1 # Start non-blocking PCR to receive computation task (subregion)
2 MPI_Pcr_Start()
3 while true
4   # Check non-blocking if receive operation of computation task is
5   # complete
6   if MPI_Test() is true
7     # Start non-blocking PCR receive for possible new computation task
8     MPI_Pcr_Start()
9     for every column of subregion
10       # Compute current row
11       compute(row)
12       # Check non-blocking if receive operation of new computation task
13       # is complete
14       if MPI_Test() is true
15         # Abort running computation and start with new one
16         goto while loop
17       # Send results using blocking send operation
18       MPI_Send(result)
19     else
# Sleep for 1 ms to reduce CPU workload
thread.sleep(1)

```

Quelltext 10: MPI Kommunikation im Worker in Pseudocode

#### 5.4.1 Berechnung ohne SIMD

Die Klasse Mandelbrot stellt die Methode calculate\_fractal bereit, mithilfe derer für ein übergebenes Array an Punkten die Iterationszahl bestimmt wird. Diese Art und Weise der Übergabe macht ein gemeinsames Interface mit den vektorisierten Versionen möglich.

Die Berechnung eines Punktes ist nun nicht weiter schwer, es muss lediglich Gleichung 1 als C++-Code umgesetzt werden. Bei  $|z_n| > 2$  die Berechnung abgebrochen werden, da der Punkt sicher nicht in der Mandelbrotmenge liegt. Um Rechenzeit zu sparen wird dabei in allen Implementierungen die äquivalente Formel  $Re(z_n)^2 + Im(z_n)^2 > 4$  evaluiert. Die Berechnung wird ebenfalls abgebrochen, wenn die maximale Anzahl an

```

1 mandelbrotOpenMP64 ,
2 mandelbrotOpenMPSIMD32 ,
3 mandelbrotOpenMPSIMD64 ,
4 };
5
6 class Fractal {
7   public:
8   /**
9    * Calculates the fractal iteration value for a given complex (real,
10      imaginary) pair
11    * @param cReal real coordinate
12    * @param cImaginary imaginary coordinate

```

Quelltext 11: Das gemeinsame Interface der Fraktale

Iterationen erreicht wurde. Die benötigte Anzahl an Iterationen wird in ein übergebenes Array geschrieben.

#### 5.4.2 Berechnung mithilfe von SIMD

Um das Parallelisierungskonzept mithilfe von SIMD zu erklären, ist es hilfreich zunächst vor Augen zu führen, wie ein Vektor komplexer Koordinaten ohne SIMD verarbeitet würde. Dies ist in Quelltext 12 zu sehen. Die Berechnung der einzelnen Koordinaten bleibt gleich, nur die Abbruchbedingung wird auf alle bearbeiteten Koordinaten erweitert.

Es muss dabei solange weiter iteriert werden, bis für alle Komponenten die Berechnung abgebrochen werden darf. Hierbei ist es kein Problem mit den abgebrochenen Punkten weiter zu rechnen, sofern die Iterationszahl nur hochgezählt wird wenn das  $z_n$  der Koordinate Betragmäßig kleiner gleich 2 ist. Dies gilt, da alle  $|z_{n+i}| > 2$  sofern  $|z_n| > 2$  [2].

Außerdem ist zu beachten, dass die Iterationszahl vor der Berechnung der nächsten Iteration erhöht werden muss. Dies liegt darin begründet die Iterationszahl die Anzahl der Rechenschritte repräsentieren soll und der nächste Schritt sicher ausgeführt wird. Auch dann, wenn in auf Basis der neuen Berechnung bereits abgebrochen wäre.

```

1 z := array(length){0}
2 n := array(length){0}
3 i := 0
4
5 lessThanTwo := array(length)
6 for k in [0, length]
7     lessThanTwo[k] = 1 if |z[k]| > 2 else 0
8
9 while(i < maxIteration && |lessThanTwo| > 0)
10    for k in [0, length]
11        n[k] += lessThanTwo[k]
12        z[k] = z[k]^2 + c
13        lessThanTwo[k] = 1 if |z[k]| > 2 else 0

```

Quelltext 12: Bearbeitung eines Vektors komplexer Koordinaten in Pseudocode

Mithilfe der in Unterabschnitt 1.4 beschriebenen SIMD-Intrinsics ist die Umsetzung dieses Codes gut möglich. Bei der Hardware-beschleunigten Variante mit SIMD, wurde stets das Postfix q eingebunden. Damit werden alle verfügbaren SIMD-Register des ARMv8-A Prozessors des ODroids und Raspberry Pi 3 B+ herangezogen.

Zusätzlich wurden Optimierungen mithilfe der NEON-nativen multiply-add (*mla*) und multiply-subtract (*mls*) Befehle vorgenommen. Zudem kann der parallele Vergleich zweier Vektoren (*clt*) ausgenutzt werden, wobei als Ergebnis jedoch nicht 1 und 0 ausgegeben werden, sondern alle Bits der Ergebnisvektorkomponente auf 1 gesetzt werden sofern die Bedingung erfüllt ist und sonst auf 0. Um im weiteren Verlauf effizient die Vektorkomponenten aufzuaddieren (*addv*) wird die ursprünglich vorzeichenlose Zahl als vorzeichenbehaftet interpretiert. Jede Komponente für die die Bedingung zu wahr evaluiert hat, kann nun als -1 interpretiert werden. Die Summe über die Komponenten

entspricht damit der negierten Anzahl an noch nicht abgeschlossenen Punkten.

## 5.5 Leistungssteigerung durch Parallelisierung auf Thread Level mithilfe von OpenMP

OpenMP<sup>17</sup> (Open Multi-Processing) ist ein API, das auf die Parallelisierung von Schleifen und Programmabschnitten auf Shared Memory Systemen spezialisiert ist. Parallel ausführbare Programmteile werden durch eine spezielle Präprozessor Anweisung für die parallele Ausführung in mehreren Threads gekennzeichnet.

Bei der Berechnung der Mandelbrotmenge wird OpenMP im Worker verwendet um die for-Berechnungsschleife parallel auf den zur Verfügung stehenden Threads auszuführen. Dies dient dazu, die Leistung des Workers signifikant zu steigern. Die dabei genutzte Anweisung setzt sich folgendermaßen zusammen:

- `#pragma omp parallel for` kennzeichnet die darauf folgende for-Schleife für die parallele Ausführung.

Folgende Parameter werden dabei noch gesetzt:

- `shared(<Datenstrukturen>)` kennzeichnet diejenigen Datenstrukturen die parallel beschreibbar sind.
- `schedule(<Parameter>)` setzt die gewählte Schedulingstrategie fest. Als Parameter wurde genutzt:
  - `nonmonotonic` besagt, dass die Schleifeniterationen in beliebiger Reihenfolge ausgeführt werden können.
  - `dynamic, 10` bedeutet, dass jeder Thread zunächst 10 Schleifeniterationen zur Berechnung zugewiesen bekommt. Hat ein Thread diese Aufgabe abgeschlossen, bekommt er weitere 10 Iterationen falls noch welche verfügbar sind. Diese Strategie wurde gewählt, um die Rechenlast möglichst gleichmäßig auf alle Threads zu verteilen und so die Gesamtrechenzeit der Schleife zu minimieren ohne ein erneutes Balancing vornehmen zu müssen.

## 5.6 Websocketverbindung

Direkt nach der Initialisierung des Host-Programms wird ein separater Thread gestartet, der über Websocket Anfragen zur Berechnung einer Region entgegennimmt sowie ein Thread, der berechnete Regionen an den verbundenen Client übergibt. Der Websocketserver wird mit `server.init_asio()` mit der Transport Policy "transport::asio" konfiguriert, sodass Multithreadzugriffe auf Sende- und Empfangsmethoden problemlos möglich sind [3].

---

<sup>17</sup><https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

Der Server behandelt zu jedem Zeitpunkt maximal eine Verbindung und speichert lediglich die zuletzt geöffnete Verbindung.

Empfangene Nachrichten werden mit der Methode `Host::handle_region_request` behandelt.

Der zweite bei der Initialisierung gestartete Thread führt die Methode `Host::send` aus.

**Host::handle\_region\_request** Die Methode dekodiert einen empfangene Nachricht als JSON Regionsanfrage und behandelt sie nach folgendem Schema:

1. Parse das empfangene RegionRequest-Objekt. Bei Fehlern wird die Funktion abgebrochen.
2. Involviere den spezifizierten Lastbalancierer um eine Aufteilung der Region zu erhalten
3. Bestimme den Rang des Workers, der eine Region bearbeiten wird. Hierzu wird der Algorithmus aus Quelltext 21 verwendet.
4. Sende die Aufteilung inklusive der Ränge aller beteiligten Worker als Regions-Objekt an das Frontend
5. Übergebe die aufgeteilten Regionen an den Thread, der diese per MPI an die Worker sendet.

In dem JSON wird unter dem Schlüssel "balancer" ein String erwartet, der den zu wählenden Lastbalancierer bestimmt. Mögliche Zeichenketten hierfür sind in den Klassen der Balancierer unter `backend/src/balancer` in der globalen Variable `Klassenname::NAME` gespeichert.

Es ist hierbei wichtig, dass das senden an die Worker erst nach der Antwort an das Frontend geschieht und wird daher garantiert. Dadurch kann im Frontend sichergestellt sein, dass alle eintreffenden RegionData-Objekte zu einer Regionsaufteilung gehören, die bereits empfangen wurde.

**Host::send** Während ein Thread das Empfangen von Nachrichten übernimmt, behandelt diese Methode von den Workern fertig berechnete Regionen. Die Methode setzt dabei eine Dauerschleife nach folgendem Schema um:

1. Überprüfe auf das Vorhandensein fertig berechneter Regionen
2. Ist keine Region verfügbar
  - a) Warte blockierend auf eine Änderung
  - b) Springe zu Schritt 1
3. Ist eine Region verfügbar

- a) Locke die geteilte Datenstruktur
  - b) Entnimm eine Region daraus
  - c) Löse das Lock
4. Codiere die Regionsdaten in JSON und versende sie über WebSocket an den aktuell verbundenen Client
5. Springe anschließend zu Schritt 1

Ein Beispiel für versendete Regionsdaten kann Quelltext 23 entnommen werden. Mithilfe einer `condition_variable`<sup>18</sup> nutzt sie die C++11 nativen mutex-Mechanismen um über das Vorhandensein neuer Regionen informiert zu werden.

## 5.7 Implementierung des Frontends

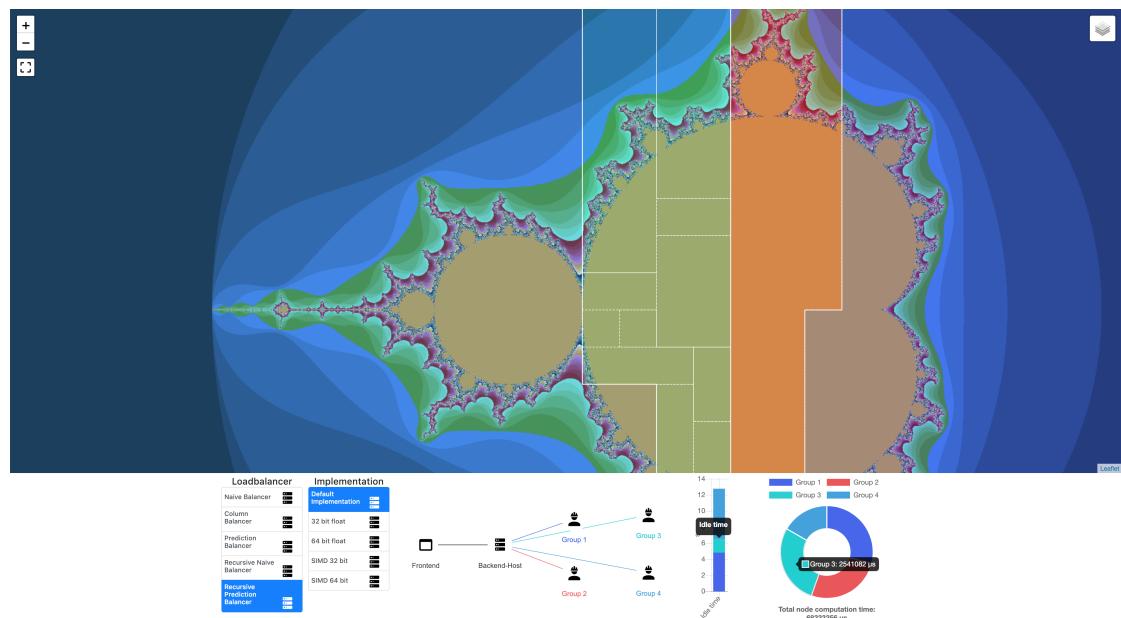


Abbildung 5: Benutzeroberfläche der Mandelbrot Anwendung

### 5.7.1 Kommunikation mit dem Backend

Zur Kommunikation mit dem Backend wird im Frontend ein Objekt der Klasse `WebSocketClient` erzeugt. Alle zur Verbindung mit dem Backend verwendeten Codestücke liegen dabei in dem Ordner `src/connection/`.

Die dort definierte Klasse `WebSocketClient` abstrahiert von dem JavaScript-nativen `WebSocket`-Interface<sup>19</sup>. Bei der Initialisierung baut das erzeugte Objekt eine

<sup>18</sup>[https://en.cppreference.com/w/cpp/thread/condition\\_variable](https://en.cppreference.com/w/cpp/thread/condition_variable)

<sup>19</sup><https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>

Verbindung zu der lokalen Adresse `ws://localhost:9002` auf<sup>20</sup>. Dort muss das Backend bereit sein, eine WebSocketverbindung anzunehmen.

Die Klasse `WebSocketClient` bietet dabei folgende Methoden:

- `sendRequest(request)`

Diese Methode versendet das übergebene `RegionRequest`-Objekt codiert als JSON<sup>21</sup>-Objekt an das Backend.

- `registerRegion(fun), registerRegionData(fun)`

An diese Methoden können Callbacks übergeben werden, die aufgerufen werden, wenn das Frontend über die WebSocketverbindung respektive ein `Region`-Objekt oder ein `RegionData`-Objekt empfängt. Diese sind die Aufteilung einer angefragten Region (siehe Quelltext 22) oder die berechneten Iterationswerte einer Region (siehe Quelltext 23).

Die übergebenen Callbacks erhalten als Parameter respektive die vorgruppier-te Aufteilung als ein Array von Workergruppen (`RegionGroup`) oder das JSON-dekodierte `RegionData`-Objekt.

Zudem wird hierbei bereits eine Filterung der eingehenden Regionsdaten vorgenommen. Bei Empfang einer Regionsaufteilung, wird diese zwischengespeichert. Jedes empfange-ne Regionsdatenobjekt wird dann bei Empfang daraufhin überprüft, ob der darin darge-stellte Ausschnitt einer Regionsaufteilung entspricht (dazu wird das `region`-Attribut verglichen) und das dargestellte Fraktal der zuletzt übergebenen Auswahl entspricht. Diese Filtrierung ist notwendig, da Worker Regionsdaten auch „verspätet“ absenden können, falls bei dem zuvorgehenden Bereich nicht gewartet wurde bis die Berech-nungen aller Worker entgegengenommen wurden. Die Definitionen der zugehörigen Objekt-Interfaces finden sich in den Dateien `RegionGroup.ts` und `ExchangeTypes.ts` finden.

Anfragen an das Backend werden dabei mit der folgenden Funktion in `RegionRequest.ts` erstellt:

- `request(map, balancer, implementation):`

Diese extrahiert aus der übergebenen Sicht auf die Mandelbrotmenge, die in der Leaflet-Karte gespeichert ist, die Parameter zum Anfragen einer Region. Dazu werden mithilfe des aktuellen Zooms der linke obere und rechte untere Punkt des Sichtbereiches in dem Leaflet-internen Koordinatensystem auf entsprechende Punkte in der komplexen Ebene projiziert. Da zum Erzeugen des passenden Objektes auch der gewünschte Lastbalancierer und der Fraktaltyp notwendig sind, werden diese als weitere Parameter übergeben.

Die Funktion gibt direkt ein Objekt zurück, dass das Interface `RegionRequest` erfüllt.

---

<sup>20</sup> Faktisch wird eine Verbindung geöffnet, geschlossen und erneut geöffnet. Dies ist durch ein ungelöstes Problem beim Verbindungsauflaufbau bedingt, das dafür sorgt, dass ein Verbindungsauflaufbau erst bei der zweiten erzeugten WebSocketverbindung fehlerfrei gelingt.

<sup>21</sup> <https://www.json.org/>

### 5.7.2 Darstellung der Regionsdaten

Die für die Darstellung der Mandelbrotmenge verwendete Bibliothek (leaflet) hat die Einschränkung, dass nur Quadrate einer vordefinierten Größe (Tiles) angezeigt werden können. Ebenfalls verwendet diese ein eigenes Koordinatensystem, unter welchem jede angezeigte Tile eindeutig mit dem Tripel  $(x, y, zoom)$  identifiziert wird. Somit ist eine Übersetzung zwischen den Daten des Backends und den von leaflet erwarteten nötig.

**MatrixView.ts, RegionOfInterest.ts** Die Klasse `MatrixView.ts` implementiert die Umsetzung einer vom Backend versendeten Region zu den von leaflet erwarteten Tiles.

- `registerTile(point, draw)`

Alle sichtbaren Tiles registrieren sich mit dem Callback `draw`, welcher ausgeführt wird, sobald die anzuzeigenden Daten für die entsprechende Tile verfügbar sind. Diese Iterationswerte des Backends werden dabei der `draw` Funktion als Parameter in Form eines `RegionOfInterest` Objekts übergeben.

Ein `RegionOfInterest` Objekt implementiert wiederum die Übersetzung von lokalen  $(x, y)$  Pixel-Werten einer Tile zu Indizes in das vom Backend gesendete Array an Regionsdaten (siehe Quelltext 23).

- `get(x, y)`

Gibt, für einen Tile  $(x, y)$  Pixel-Wert die benötigte Iterationsanzahl zurück.

**TileDisplay.tsx** Die Klasse `TileDisplay.tsx` verwendet die leaflet Bibliothek direkt, um die Regionsdaten darzustellen. Dabei wird dieser die WebSocket Verbindung in Form eines `WebSocketClient`, sowie der vom Nutzer gewählte Balancer, die Implementierung und Gruppierung durch `Observable` Klassen (siehe ??) übergeben. Ebenfalls wird der anzuzeigende Ausschnitt der Mandelbrotmenge übergeben.

Da das Backend die berechneten Teilbereiche der Mandelbrotmenge als Regionen zurück gibt, dessen Höhe und Breite Vielfache der leaflet Tile-Größe sind, wird eine Region dem Benutzer durch mehrere Tiles dargestellt. Dieses Verhältnis wird in Abbildung 6 dargestellt.

Zudem ist die Darstellung der Iterationswerte, sowie Regionsaufteilung durch unterschiedlichen Ebenen innerhalb der leaflet Bibliothek realisiert:

- `MandelbrotLayer` in `TileDisplay.tsx`

In dieser eigenen Ebene der leaflet Karte werden die Tiles dargestellt. Dieser erstellt für den sichtbaren Bereich<sup>22</sup> alle benötigten Tiles. Für jede der Tiles wird ein HTML5 `canvas`<sup>23</sup> Objekt erstellt, welches es ermöglicht, für jeden Pixel einen  $(r, g, b)$  Farbwert zu definierten und anzuzeigen. Wobei die Farbwerte aus den berechneten

---

<sup>22</sup> Da die Fenstergröße des sichtbaren Bereichs kein Vielfaches der Tilegröße sein muss, können Tiles erzeugt werden, welche teilweise außerhalb des sichtbaren Bereichs liegen

<sup>23</sup> [https://developer.mozilla.org/kab/docs/Web/API/Canvas\\_API](https://developer.mozilla.org/kab/docs/Web/API/Canvas_API)

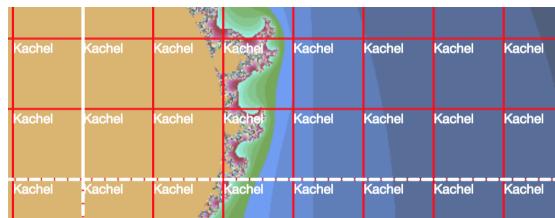


Abbildung 6: Relation von Backend Regionen zu leaflet Tiles (Kachel). Dabei ist beispielhaft eine Region des Backends weiß eingezeichnet, alle leaflet Tiles sind rot umrandet angegeben.

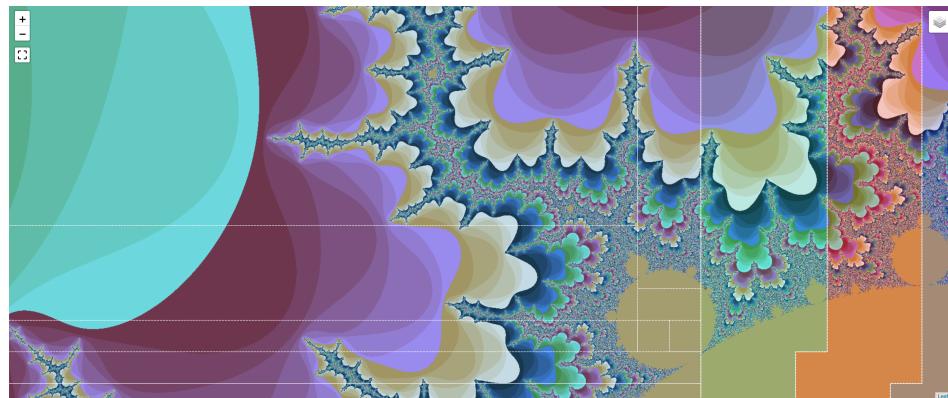


Abbildung 7: WorkerLayer für eine Gruppierung der Aufteilung des Recursive PredictionBalancers mit 37 Workern

Iterationswerte des Backends mit einem Shader (siehe Abschnitt 5.7.2) ermittelt werden. Die Iterationswerte können wiederum mit einem MatrixView Objekt aus den Regionsdaten des Backends gelesen werden.

- **WorkerLayer in WorkerLayer.ts**

Diese Klasse visualisiert die Regionsaufteilung des Lastbalancierers als Overlay, welche über den Iterationswerten dem Benutzer angezeigt wird. Dafür wird eine in leaflet bestehende GeoJSON API verwendet, mit welcher es möglich ist, beliebige Polygone auf den bestehenden Kartendaten anzuzeigen. Die Knoten dieser Polygone werden dabei jede RegionGroup mit Hilfe der Funktionen aus Project von Koordinaten der komplexen Ebene, welche im Backend verwendet werden, zu leaflet Koordinaten umgerechnet.

Da es wie in Abschnitt 5.7.2 beschrieben zu einer Gruppierung kommt, falls die Anzahl der Worker im Backend zu groß ist, werden ebenfalls alle Untergruppen einer Gruppe angezeigt (siehe Abbildung 7), falls der Benutzer mit der Maus über eine der dargestellten Gruppierungen geht.

- **DebugLayer in TileDisplay.tsx**

Diese Ebene gibt Informationen zur Aufteilung der angezeigten leaflet Tiles und den Projektionen von leaflet Koordinaten zu komplexen Koordinaten.

**Shader.ts** Berechnet für einen Iterationswert der Mandelbrotmenge ein Tripel  $(r, g, b)$  von Farbwerten. Implementiert ist eine simple Funktion, welche den Iterationswert für jeden Farbkanal mit einer Konstante multipliziert und auf den zulässigen ganzzahligen Wertebereich  $[0, 255]$  abbildet (siehe Quelltext 13).

```

2  public static default(n: number, maxIteration: number): number[] {
3    const r = (n * 10) % 256;
4    const g = (n * 20) % 256;
5    const b = (n * 40) % 256;
6    return [r, g, b];
7 }
```

Quelltext 13: Berechnung der  $(r, g, b)$  Farbwerte

**Project.ts** Da die leaflet Bibliothek Tiles verwendet, um die Regionen des Backends anzuzeigen, liegen diese in einem neuen Koordinatensystem, welches jeder Tile auf für eine Zoomstufe ein Triple  $(tileX, tileY, zoom)$  zuordnet. Weiterhin besitzt leaflet ein internes Koordinatensystem (CRS<sup>24</sup>), welches jeden Punkt auf der Karte durch das Paar  $(latitude, longitude)$  identifiziert. Projekt.ts enthält Funktionen, um zwischen diesen 3 Koordinatensystemen (Tile-Koordinaten, leaflet-Koordinaten und Koordinaten der komplexen Ebene) zu konvertieren.

### Regionsgruppierung (RegionGroup.ts)

#### 5.7.3 Visualisierung

Die Struktur der Parallelisierung wird in einem Netzwerkgraphen unter dem Fraktal dargestellt. Mithilfe der Bibliothek visjs<sup>25</sup> wird hierzu ein Graph (siehe Abbildung 8) mit 3 Ebenen erzeugt:

Auf der linken Seite befindet sich ein Knoten in Form eines Programmfensters, der das Browserfrontend darstellt. In der Mitte wird ein Knoten für den Backend-Host mit Serverrack als Symbol dargestellt, der mit dem Frontendknoten verbunden ist. Auf der rechten Seite befindet sich schließlich zwischen 2 und 4 Knoten, die jeweils eine Gruppe an Arbeitern darstellen. Das Symbol hierfür ist ein Oberkörper mit Arbeitshelm. Sie sind wiederum direkt mit dem Host verbunden.

Der Code für diese Darstellung findet sich in components/NetworkView.tsx. Dort wird ein Baumgraph von links nach rechts mit manueller Ebenenverteilung als Grundstruktur für die Darstellung des Graphen definiert. Daher wird das Frontend auf der höchsten Ebene und der Backend-Host auf der Ebene darunter spezifiziert. Die Arbeiter

<sup>24</sup>[https://en.wikipedia.org/wiki/Spatial\\_reference\\_system](https://en.wikipedia.org/wiki/Spatial_reference_system)

<sup>25</sup><http://visjs.org/>

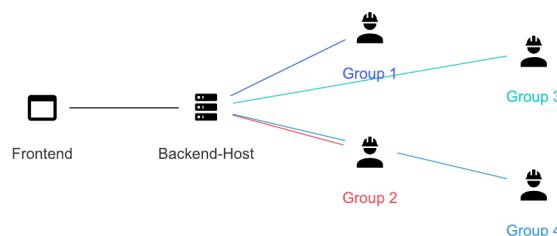


Abbildung 8: Darstellung der Architektur der Anwendung als Graph



Abbildung 9: Idle Time für 5 Gruppen

Abbildung 10: Computation time für 4 Gruppen

werden auf Basis der erhaltenen Regionsaufteilung erzeugt und jeweils zu zweit nacheinander auf Ebenen verteilt. Diese Aufteilung wird nach Erhalt jeder Regionsaufteilung vorgenommen und der Graph neu aufgebaut, sowie die Größe der Leinwand an die Anzahl an Knoten angepasst.

#### 5.7.4 Visualisierung der Rechenzeiten

Die zeitbezüglichen Komponenten der Visualisierung werden mithilfe der Charts der Bibliothek chart.js<sup>26</sup> dargestellt.

**Idle Time** Für die kritische Information der verschwendeten Zeit wird ein Balkengraph verwendet. Dieser besitzt einen Balken, der die Differenz zwischen der Rechenzeit aller Knoten und dem am längsten rechnenden Knoten aufsummiert und nach Gruppe sortiert darstellt (siehe Abbildung 9).

Die Darstellung wird bei Empfang einer Regionsaufteilung (mithilfe registerRegion in WebSocketClient) initialisiert indem die Anzahl an Knoten und die Gruppen gespeichert, die Rechenzeit der Knoten auf 0 gesetzt und alle Knoten als aktiv markiert werden. Da die tatsächliche Rechenzeit erst am Ende der Berechnung mit den Regionsdaten selbst verfügbar ist, wird mithilfe eines Intervalls die Rechenzeit live abgeschätzt, indem sie alle 50ms um 50ms hochgezählt wird, falls die Regionsdaten des Knotens noch nicht empfangen wurden.

<sup>26</sup><http://www.chartjs.org/>

Da die Charts nicht zur Animation von Übergängen fähig sind, wird das Objekt mit jedem Update neu gerendert. Bei Empfang von Regionsdaten (hierzu wird die erwähnte Methode `registerRegionData` verwendet) wird die tatsächlich gemessene Rechenzeit eingetragen und der Rechenknoten als inaktiv gekennzeichnet. Der Intervall wird gestoppt, sobald alle Arbeiterknoten die Berechnung abgeschlossen haben.

**Computation Time** Zur Darstellung des relativen Verhältnisses der Rechenzeiten wird in einem Kuchengraph die absolute Rechenzeit der Gruppen dargestellt (siehe Abbildung 10).

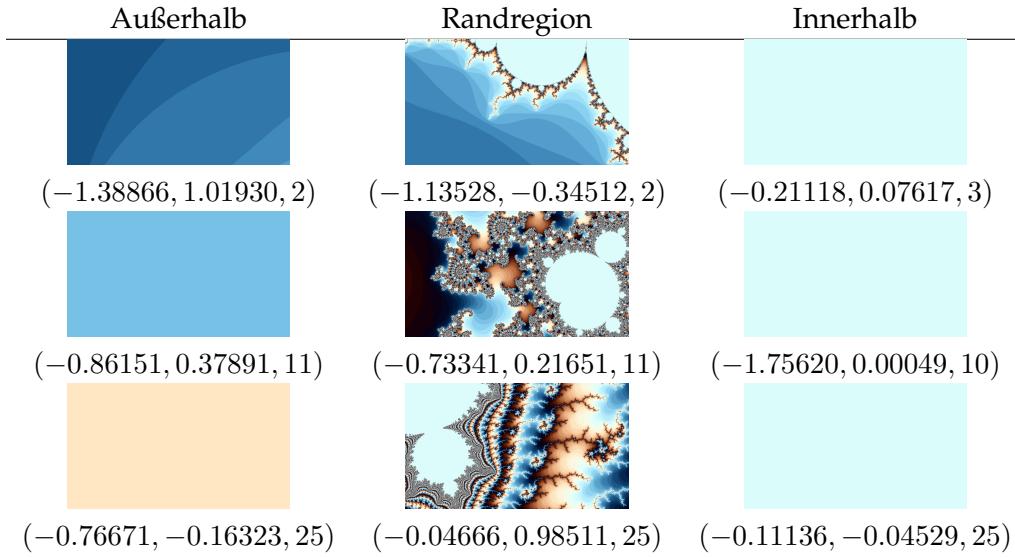


Abbildung 11: Testregionen der Evaluierung. Jeder der Punkte  $z \in \mathbb{C}$  entspricht dem Mittelpunkt der Region im Format  $(Re(z), Im(z), zoom)$

## 6 Ergebnisse / Evaluation

### 6.1 Datenerhebung

Die in den folgenden Abschnitten ausgewerteten Rechenzeiten für die Komponenten des Back- und Frontends wurden in den entsprechenden Teilanwendungen in  $\mu\text{s}$  unter Verwendung der jeweiligen Systembibliotheken (`std::chrono`<sup>27</sup> in C++ und Performance API<sup>28</sup> in `node.js`) gemessen. Dabei stellt das so modifizierte Backend eine Erweiterung dar, welche die gemessenen Zeiten zusätzlich in der Antwort der Regionsdaten an das Frontend versendet. Dort werden diese aggregiert, mit den Zeiten des Frontend verknüpft und ausgegeben.

Um die Performance der folgenden Algorithmen auf der Mandelbrotmenge zu evaluieren, müssen Regionen einzeln ausgewählt und die Performanz ihrer Berechnung verglichen werden. Dafür haben wir die Mandelbrotmenge in drei Klassen unterteilt: Regionen in, am Rand und außerhalb der Menge. Diese Aufteilung ist so gewählt, dass sich die Anzahl Iterationen pro Punkt innerhalb einer Klasse ähneln. Dazu haben wir ebenfalls für jede der Klassen eine niedrige, mittlere und hohe Zoomstufe gewählt (siehe Abbildung 11).

TODO: einarbeiten Als Cluster unabhängiger Rechenknoten wurde freundlicherweise vom Lehrstuhl für Rechnerarchitektur und Verteilte System der TU München das HiMMUC Cluster<sup>29</sup> bereitgestellt. Es besteht aus jeweils 40 Raspberry Pi B+<sup>30</sup> und ODroids

<sup>27</sup><https://en.cppreference.com/w/cpp/chrono>

<sup>28</sup><https://developer.mozilla.org/en-US/docs/Web/API/Performance>

<sup>29</sup><http://www.caps.in.tum.de/himmuc/>

<sup>30</sup>Für ein Beispiel, siehe <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

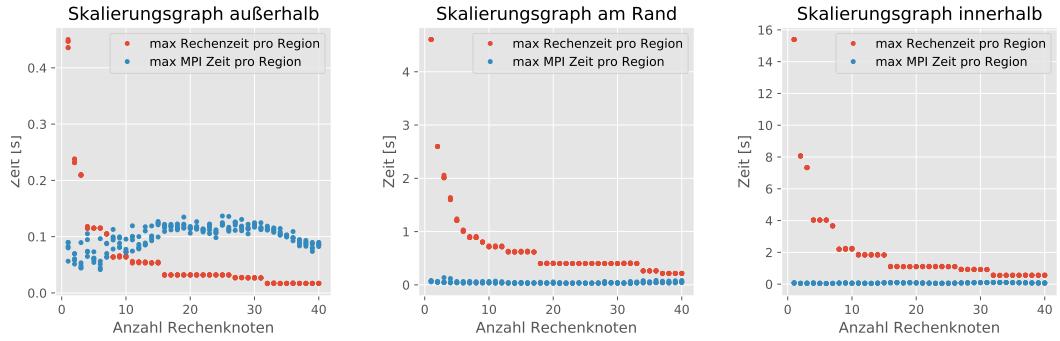


Abbildung 12: Skalierungsgraph von 1 – 40 Rechenknoten, Strategie mit Vorhersage (rekursiv)

C2<sup>31</sup>.

## 6.2 Skalierung

Eine wichtige Performanzmetrik, welche durch das Projekt evaluiert werden solle, ist wie gut sich eine unabhängige Berechnung, wie die der Mandelbrotfunktion parallelisieren lässt. Dabei ist vor allem von Interesse, wie sich die Rechenzeit der einzelnen Teilreigonen des Balancers im Vergleich zur Anzahl der an der Berechnung beteiligten Worker verhält. Dies ist in Abbildung 12 aufgetragen.

Dabei ist jeweils ein Skalierungsgraph für jede der drei Regionstypen (innerhalb, am Rand und außerhalb) angegeben, was nötig wird, da Regionen abhängig von der Anzahl der Pixel, welche innerhalb der Mandelbrotmenge liegen, sehr unterschiedliche Rechenzeiten haben. Erwartet ist dabei, dass sich die Rechenzeit  $t$  ungefähr nach  $t \approx c_1/n + c_2$  verhält, wobei  $n$  die Anzahl der Rechenknoten und  $c_1, c_2$  geeignet gewählte Konstanten zur Abschätzung des Overheads sind. Diese Relation lässt sich dabei auch in den gemessenen Daten in Abbildung 12 wieder finden.

Es ist jedoch zu beachten, dass somit der

## 6.3 OpenMP

## 6.4 SIMD

SIMD unterstützt in den Präzisionen 32 und 64 bit Parallelisierung von einer Rechenoperationen auf 4 beziehungsweise 2 unterschiedlichen Werten. Der Effekt beläuft sich dabei, wie in Abbildung 14 zu sehen, bei einer Beschränkung auf 1019 Iterationen auf eine durchschnittliche Beschleunigung um den Faktor 2, 3 und 1, 2.

Dass die Performanzerhöhung nicht genau 4 oder 2 ist, lässt sich mit dem erhöhten Aufwand der Verwendung der SIMD Instruktionen erklären. Da hierzu die Werte aus den normalen Registern in spezielle SIMD-Register und zurück kopiert werden müssen,

<sup>31</sup><https://www.hardkernel.com/shop/odroid-c2/>

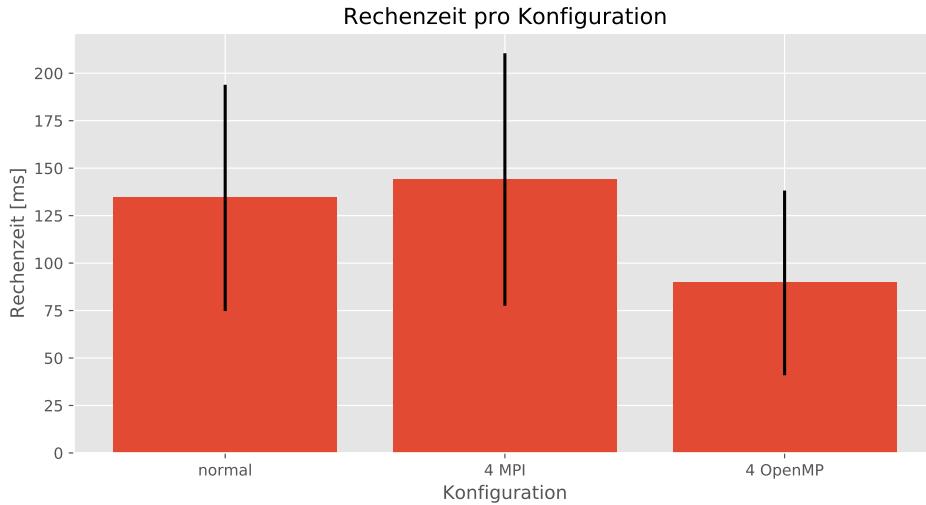


Abbildung 13: Parallelisierungsstrategien: ein MPI Prozess pro Node, 4 MPI Prozesse pro Node & 4 OpenMP threads pro Node

entsteht eine gewisse Verzögerung durch zusätzliche Transportoperationen. Die tatsächliche Rechenzeit für eine Vektorisierung von  $n$  Punkten sollte also  $t_{SIMD} \approx \frac{t_{normal}}{n} + const$  entsprechen.

Je größer die benötigte Zahl an Operationen, desto weniger fällt dieser konstante Zusatzaufwand ins Gewicht. Dies kann zum Beispiel in Abbildung 14 beobachtet werden. Steigert man die maximale Iterationszahl von 1019 auf bis zu 10000 steigt dieser Speed-Up nicht wesentlich weiter. Er erreicht sein vorläufiges Maximum bei 2.5 und 1.25. Details zu dieser Entwicklung sind in Unterabschnitt 8.7 zu finden.

Zudem werden für eine Menge an Punkten stets die Zahl an Iterationen durchgeführt, die das Maximum aller Iterationszahlen der Punkte ist. Bei Iterationszahlen  $i_k$  des Punktvektors  $p$  ist daher anstatt einer Rechenzeit von  $\sum_{k \in p} \frac{i_k}{|p|}$  eine Rechenzeit von  $\sum_{k \in p} \frac{\max(i_k)_{k \in p}}{|p|} = \max(i_k)_{k \in p}$  pro Punktvektor zu erwarten. Damit wird die Beschleunigung durch SIMD kleiner, je inhomogener die Iterationszahlen sind. Dass die Iterationszahlen im Allgemeinen am Rand der Mandelbrotmenge inhomogener sind, wo auch die Gesamtrechenzeit geringer ist, sollte bei der Betrachtung von Abbildung 14 berücksichtigt werden.

Insgesamt kann mit SIMD eine deutliche Beschleunigung des Systems bewirkt werden. Stehen jedoch mehr Rechenkerne zur Verfügung, schneidet wie in Abbildung 15 zu sehen eine Erhöhung der MPI-Prozesse besser ab. Da die Parallelisierungsmethoden jedoch unabhängig voneinander sind, führt die Anwendung von SIMD generell zu einer Erhöhung der Performanz und empfiehlt sich bei einfacher Umsetzbarkeit.

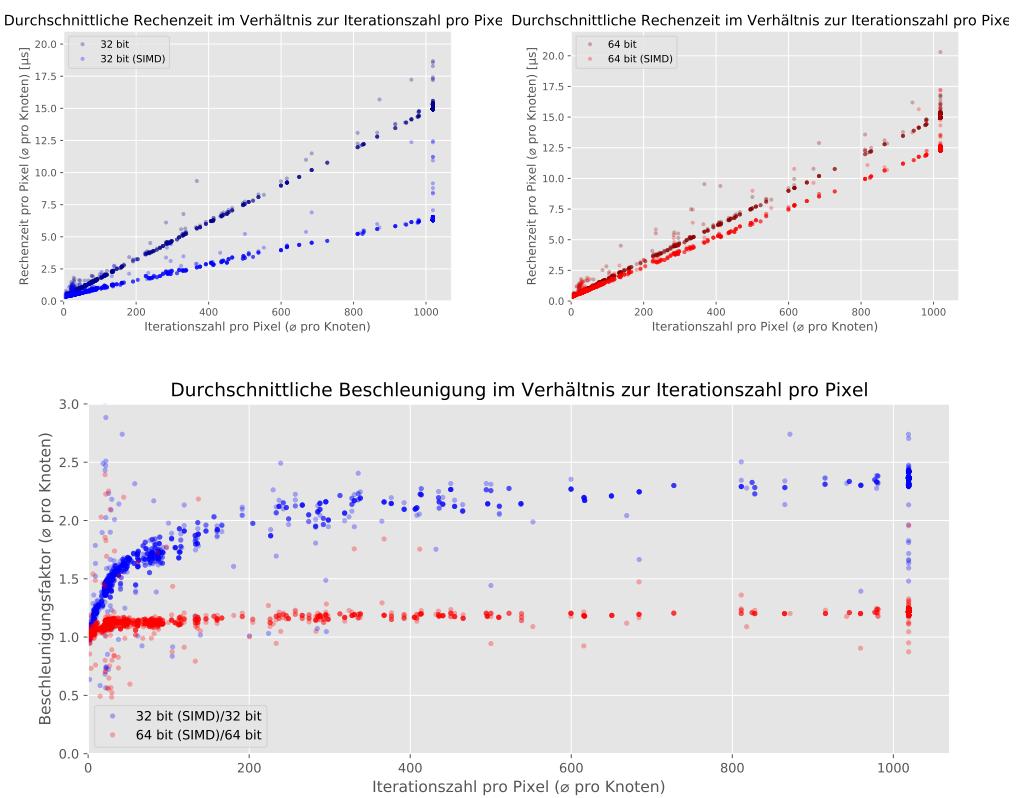


Abbildung 14: Vergleich der Performanzen der Implementierungen mit und ohne SIMD bei bis zu 1019 Iterationen über 360 Regionen in 10 Wiederholungen.

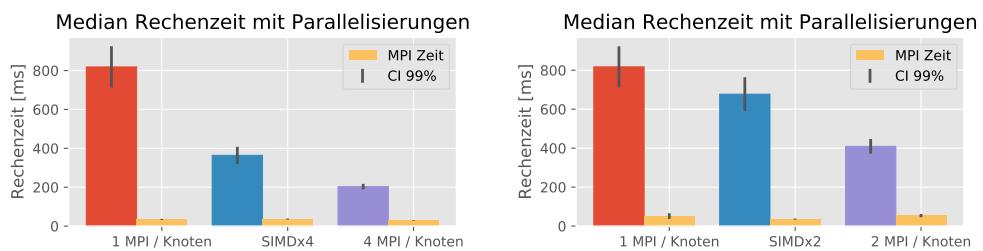


Abbildung 15: Vergleich zwischen der Performanz mit mehr MPI-Prozessen und SIMD bei naiver Lastbalancierung. Eine Erhöhung der Anzahl an Rechenknoten führt zu einer deutlich größeren Beschleunigung.

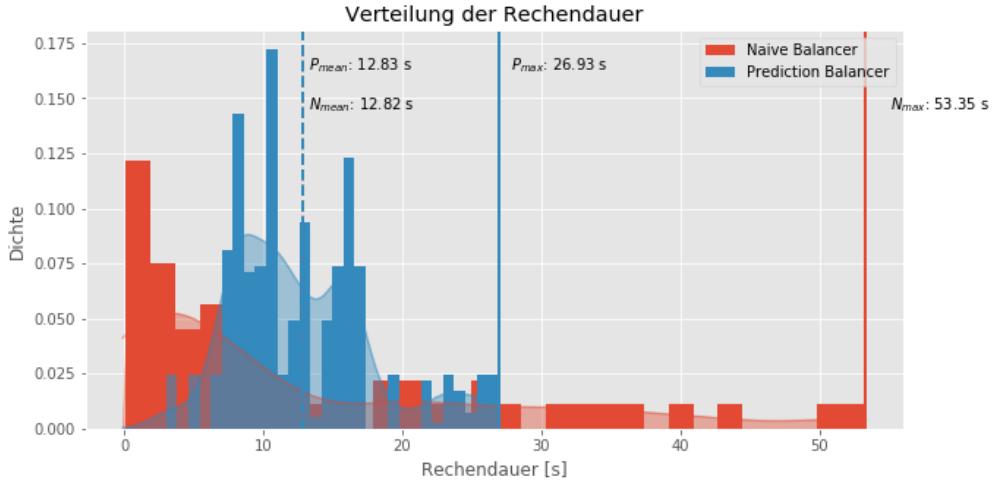


Abbildung 16: Verteilung der Rechenzeiten für Randregionen

## 6.5 Lastbalancierung

Zur Lastbalancierung gibt es zwei Strategien in jeweils zwei Varianten, zur Evaluierung wird allerdings nur die rekursive Variante der beiden Strategien betrachtet. Man stellt leicht fest, dass die nicht-rekursive Variante für Primzahlen als Worker-Anzahl schlecht ist, weil dabei nur eine Aufteilung in Zeilen oder Spalten möglich ist. Da die Aufteilungsmöglichkeiten zusätzlich durch die Betrachtung von Tiles als atomaren Einheiten beschränkt werden, erzeugt diese Variante bereits für eine geringe Anzahl an Workern verhältnismäßig viele Leerregionen (d.h. untätige Worker).

Um die naive Strategie und die Strategie mit Vorhersage zu vergleichen werden drei Klassen von Regionen betrachtet (vgl. Abbildung 11), für die es unterschiedliche Erwartungen gibt:

- innere Regionen: Die Rechenzeiten sind bei beiden Strategien gleichmäßig hoch
- äußere Regionen: Die Rechenzeiten sind bei beiden Strategien gleichmäßig niedrig
- Randregionen: Für die naive Strategie wird hier eine große Streuung der Rechenzeiten erwartet. Die Strategie mit Vorhersage sollte für eine Ballung um den Mittelwert sorgen.

Weiterhin wird erwartet, dass der Mittelwert der Rechenzeit pro Regionsklasse für die beiden Strategien gleichbleibt, da die selbe Region mit der selben Anzahl an Workern berechnet wird. Für die Messungen hier wurden immer 17 Worker verwendet.

In den folgenden Diagrammen wird die statistische Dichte der Rechenzeiten für die naive Strategie und die Strategie mit Vorhersage dargestellt. Bei einer guten Lastbalancierung sollten sich die Rechenzeiten also um den Mittelwert ballen und die maximale Rechenzeit gering sein.

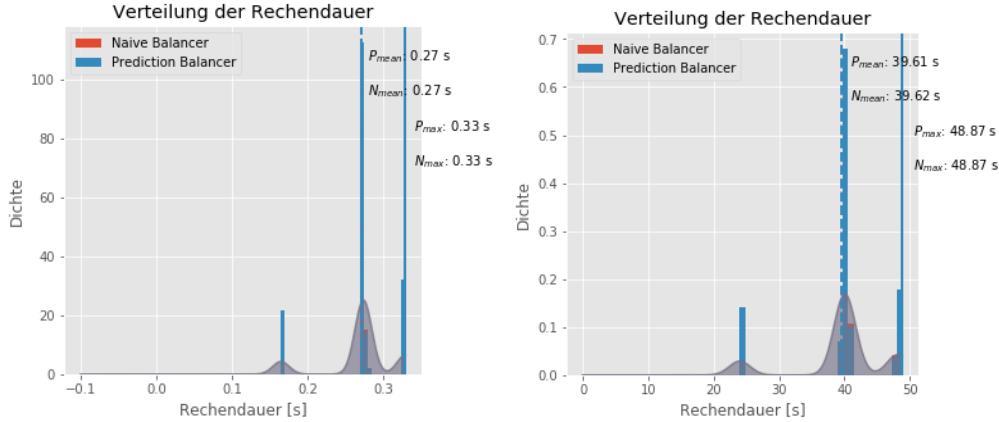


Abbildung 17: Verteilung der Rechenzeiten für innere und äußere Regionen

Den Unterschied zwischen naiver Strategie (rot) und Strategie mit Vorhersage (blau, leicht durchsichtig) sieht man besonders gut in der Verteilung für Randregionen (Abbildung 16). Die blauen Balken haben ihr Maximum in der Nähe des Mittelwerts, während die roten Balken viel mehr über den Wertebereich verteilt sind. Dass die Strategie mit Vorhersage vom Mittelwert abweicht und auch einige Ausreißer nach oben und unten hat lässt sich auf Ungenauigkeiten in der Vorhersage zurückführen, welche am Rand der Mandelbrotmenge besonders ausgeprägt sind (vgl. Abschnitt 3.3.2).

Auch bei den inneren und äußeren Regionen wurde das erwartete Ergebnis erzielt. In Abbildung 17 sieht man, dass die Verteilung für naive Strategie und Strategie mit Vorhersage nahezu deckungsgleich sind. Die auffällige Abweichung einiger Regionen nach oben oder unten ist durch die unterschiedliche Größe begründet. Theoretisch sollte die Größe aller Teilregionen gleich sein, allerdings ist dies im Diskreten nicht immer möglich, da Breite und Höhe nicht unbedingt glatt teilbar sind. Durch die Betrachtung von Tiles (in diesem Test  $64 \times 64$  Pixel) als atomare Einheit wird dieser Effekt noch verstärkt.

Bei der Strategie mit Vorhersage lässt sich die Genauigkeit der Vorhersage einstellen (vgl. Abschnitt 5.2.2). Deshalb wird hier eine Kosten-Nutzen Analyse durchgeführt. Dazu werden für verschiedene Genauigkeiten die Verteilung der Rechenzeiten und die Dauer der Vorhersage aufgetragen. Die Messungen fanden unter den selben Bedingungen wie die Obigen statt, allerdings wurden für die Implementierung hier 64-bit floats verwendet, was die absoluten Rechenzeiten durch die geringere Genauigkeit verkürzt.

In Abbildung 18 sieht man, dass sich die maximale Rechenzeit schon ab einer Vorhersagegenauigkeit von 4 nicht mehr signifikant verbessert, davor ist die Beschleunigung aber deutlich sichtbar. Die für die Lastbalancierung benötigte Zeit steigt exponentiell an, was bei der Definition der Genauigkeit zu erwarten war. Ab Genauigkeit 22 übersteigt die Zeit für die Vorhersage sogar die maximale Berechnungsdauer einer Region, was dem Sinn einer Lastbalancierung widerspricht.

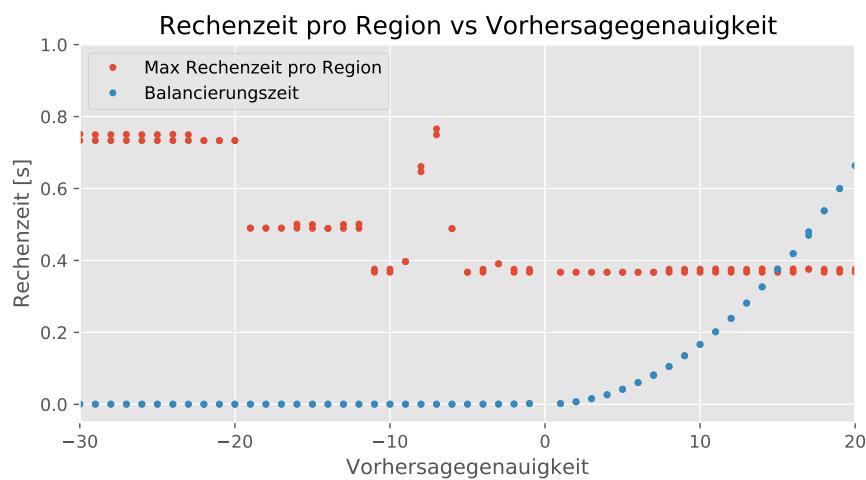


Abbildung 18: Maximale Rechenzeit pro Region und für die Lastbalancierung benötigte Zeit für verschiedene Vorhersagegenauigkeiten

## 6.6 Zusammenfassung

## 7 Zusammenfassung und Ausblick

Ziel des Projektes war es, ein Programm zur parallelen Berechnung der Mandelbrotmenge auf einem Rechnercluster zu erstellen. Dazu wurden drei unterschiedliche Ansätze der Parallelisierung genutzt:

- **MPI:** Verteilung der Rechenlast auf die Hardwareknoten des Clusters, Kommunikation über Nachrichten zwischen den Knoten
- **OpenMP:** Verteilung der Rechenlast auf die CPU-Kerne eines Knotens, Kommunikation über geteilten Speicher
- **SIMD:** Verteilung der Rechenlast innerhalb eines CPU-Kernes durch Nutzung von Vektorregistern

Die Verteilung auf verschiedene Rechenknoten erfordert eine Lastbalancierung, damit die Knoten gleichmäßig ausgelastet werden. Hier wurden dafür zwei verschiedene Strategien realisiert:

- **Naive Strategie:** Aufteilung in gleich große Bereiche
- **Strategie mit Vorhersage:** Aufteilung in Bereiche mit gleichem Rechenaufwand mithilfe einer Heuristik

Weiterhin sollte der Effekt der Parallelisierung für den Nutzer deutlich werden. Dazu gibt es eine Web-Schnittstelle in der die Mandelbrotmenge optisch ansprechend dargestellt wird und von der Nutzerin erkundet werden kann. Der Benutzer hat die Möglichkeit die Parallelisierung mittels OpenMP und SIMD an- oder abzuschalten, sowie die Strategien zur Lastbalancierung ändern. Außerdem werden Rechenzeit und die Zeit in der Knoten untätig waren in Diagrammen angezeigt. Die Aufteilung der Lastbalancierung kann als Overlay über die Darstellung Mandelbrotmenge gelegt werden.

---

## Abbildungsverzeichnis

1	Die Mandelbrotmenge, visualisiert in einem Ausschnitt des komplexen Zahlenraumes. . . . .	4
2	Architekturübersicht . . . . .	9
3	Konzept der versendeten Nachrichten. Die genauen Definitionen der Nachrichten sind in den angegeben Dateien nachzusehen. . . . .	11
4	Konzept der Koordinaten in den Regionsobjekten. Alle Koordinaten beziehen sich auf die Darstellungsebene und sind daher in Pixeln. . . . .	12
5	Benutzeroberfläche der Mandelbrot Anwendung . . . . .	30
6	Relation von Backend Regionen zu leaflet Tiles (Kachel). Dabei ist bei-spielhaft eine Region des Backends weiß eingezeichnet, alle leaflet Tiles sind rot umrandet angegeben. . . . .	33
7	WorkerLayer für eine Gruppierung der Aufteilung des Recursive PredictionBalancers mit 37 Workern . . . . .	33
8	Darstellung der Architektur der Anwendung als Graph . . . . .	35
9	Idle Time für 5 Gruppen . . . . .	35
10	Computation time für 4 Gruppen . . . . .	35
11	Testregionen der Evaluierung. Jeder der Punkte $z \in \mathbb{C}$ entspricht dem Mittelpunkt der Region im Format $(Re(z), Im(z), zoom)$ . . . . .	37
12	Skalierungsgraph von 1 – 40 Rechenknoten, Strategie mit Vorhersage (rekursiv) . . . . .	38
13	Parallelisierungsstrategien: ein MPI Prozess pro Node, 4 MPI Prozesse pro Node & 4 OpenMP threads pro Node . . . . .	39
14	Vergleich der Performanzen der Implementierungen mit und ohne SIMD bei bis zu 1019 Iterationen über 360 Regionen in 10 Wiederholungen. . . . .	40
15	Vergleich zwischen der Performanz mit mehr MPI-Prozessen und SIMD bei naiver Lastbalancierung. Eine Erhöhung der Anzahl an Rechenknoten führt zu einer deutlich größeren Beschleunigung. . . . .	40
16	Verteilung der Rechenzeiten für Randregionen . . . . .	41
17	Verteilung der Rechenzeiten für innere und äußere Regionen . . . . .	42
18	Maximale Rechenzeit pro Region und für die Lastbalancierung benötigte Zeit für verschiedene Vorhersagegenauigkeiten . . . . .	43
19	Beschleunigungsfaktor durch SIMD in Abhängigkeit von der durch-schnittlichen Iterationszahl. Auswertung von 360 Regionen mit zwei Durchläufen. Die Iterationszahl für einen Abbruch war 10000. . . . .	55
20	Beschleunigungsfaktor durch SIMD in Abhängigkeit von der durch-schnittlichen Iterationszahl. Auswertung von 360 Regionen mit zwei Durchläufen. Die Iterationszahl für einen Abbruch war 150. . . . .	56

## Tabellenverzeichnis

## Literatur

- [1] Erich Gamma, editor. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995.
- [2] 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).
- [3] zaphoyd (<https://www.zaphoyd.com/>). Websocket++ user manual. Zaphoyd.com. URL:<https://www.zaphoyd.com/websocketpp/manual/reference/thread-safety> (version: 2013-03-02).

## 8 Anhang

### 8.1 Detaillierter Start des Backends auf dem HIMMUC

Um Mandelbrot manuell auf dem Host-System zu installieren, müssen zunächst die notwendigen Bibliotheken installiert werden. Eine Anleitung dazu findet sich in Abschnitt 8.1. Dies muss nur einmal ausgeführt werden, anschließend können die Programme wie in Abschnitt 8.1 beschrieben kompiliert werden. Ist dies nicht gewünscht oder erledigt muss das Backend lediglich noch wie in Abschnitt 8.1 beschrieben gestartet werden.

**Lokale Installation der Bibliotheken** Da hierbei davon ausgegangen wird, dass keine root-Rechte auf dem Server existieren, werden die Bibliotheken hier lokal in `~/.eragp-mandelbrot` installiert. Achten Sie darauf, dass sie Schreibrechte auf dem Ordner haben und falls sie einen anderen Ordner verwenden wollen, ersetzen sie jedes Vorkommen des Pfades durch ihren Pfad (insbesondere in der Datei `CMakeLists.txt`). Die MPI-Bibliothek ist auf dem HimMUC Cluster bereits vorinstalliert und muss daher nicht mehr aufgesetzt werden.

```
1 mkdir ~/.eragp-mandelbrot
```

Quelltext 14: Erstellen des Installationsordners

Die 'Header-only' Libraries `websocketpp` und `rapidjson` müssen lediglich an einen fixen Ort kopiert werden. Dies erledigen die Befehle aus Quelltext 15.

```
1 mkdir "~/.eragp-mandelbrot/install"
2 cd "~/.eragp-mandelbrot/install"
3 # Installation von websocketpp
4 git clone --branch 0.7.0 https://github.com/zaphoyd/websocketpp.git
   websocketpp --depth 1
5 # Installation von rapidjson
6 git clone https://github.com/Tencent/rapidjson/
```

Quelltext 15: Lokale Installation der Bibliotheken `websocketpp` und `rapidjson`.

Aus der Bibliothek `boost` muss die Teilbibliothek `boost_system` lokal kompiliert werden. Dazu werden die Befehle aus Quelltext 16 ausgeführt, um die Version 1.67.0 herunterzuladen, zu entpacken und lokal zu installieren. Beachten Sie, dass das kompilieren auch wie in Abschnitt 8.1 beschrieben von einem der Boards ausgeführt werden muss.

**Kompilieren des Backends** Stellen Sie zunächst sicher, dass auf dem Cluster die Quelldateien des Backends (im Ordner `backend/`) liegen (zum Beispiel über `rsync` oder indem sie das Repository dort auch klonen). Zum Kompilieren des Backends sollte sich

```

1 # Erstellen der notwendigen Ordnerstrukturen
2 mkdir "~/.eragp-mandelbrot/install"
3 mkdir "~/.eragp-mandelbrot/local"
4 # Einrichten des Internetproxys
5 export http_proxy=proxy.in.tum.de:8080
6 export https_proxy=proxy.in.tum.de:8080
7 # Herunterladen und kompilieren der Boost-Bibliothek
8 cd "~/.eragp-mandelbrot/install"
9 wget "https://dl.bintray.com/boostorg/release/1.67.0/source/boost_1_67_0.tar.bz2"
10 tar --bzip2 -xf boost_1_67_0.tar.bz2
11 cd boost_1_67_0
12 ./bootstrap.sh --prefix="$HOME/.eragp-mandelbrot/local/" --with-libraries=system
13 ./b2 install

```

Quelltext 16: Lokale Installation der Bibliothek boost.

auf einen Raspberry Pi oder ODroid per ssh eingeloggt werden<sup>32</sup>

Auf dem Board, aus dem Ordner des Backendquellcodes müssen Sie zum kompilieren des Backendes die Befehle aus Quelltext 17 ausführen.

```

1 # Erstellen und betreten eines build Ordners
2 mkdir build
3 cd build
4 # Aktivieren der MPI Bibliothek
5 module load mpi
6 # Kompilieren
7 cmake ..
8 make

```

Quelltext 17: Kompilieren des Backends

**Ausführen des Backends** Um das Backend auf dem HimMUC Cluster laufen zu lassen, muss sich zunächst darauf per ssh eingeloggt werden. Damit für das Frontend kein Unterschied dazwischen besteht, ob das Backend im Dockercontainer , oder auf einem externen Server ausgeführt wird, ist bei der ssh-Verbindung der Port 9002 des himmuc.caps.in.tum.de-Servers an den lokalen Port 9002 gebunden. So ist das Backend stets unter localhost:9002 verfügbar. Der zugehörige Befehl zum Login lautet demnach:

```
1 ssh <rechnerkennung>@himmuc.caps.in.tum.de -L localhost:9002:localhost:9002
```

Anschließend muss aus dem Ordner, in dem die ausführbaren Dateien liegen, für gewöhnlich also der ~/.eragp-mandelbrot/build/ Ordner, folgender Befehl ausgeführt werden:

---

<sup>32</sup>Es existiert ein Entwicklerzugang zu einem geteilten Raspberry Pi über die Adresse sshgate-gepasp.in.tum.de. Dieser wird auch vom Pythonskript genutzt

```

1 srun -p <odr|rpi> -n <number of workers+1> -N <number of nodes/raspis> -l
   --multi-prog <path to eragp-mandelbrot/backend>/himmuc/run.conf &
2 ssh -L 0.0.0.0:9002:localhost:9002 -fN -M -S .tunnel.ssh <odr|rpi><host
   number>
```

Dabei bestimmt `-n` die Anzahl der laufenden Prozesse (Also Hostprozess und Workerprozesse) und `-N` die Anzahl zu verwendender Rechenknoten. Damit anschließend noch alle Anfragen an den WebSocketserver auf dem Hostknoten weitergeleitet werden, muss noch der Port 9002 des `himmuc.in.caps.tum.de`-Servers an den Port 9002 des Rechenknotens gebunden werden, auf dem der Hostprozess läuft. Der korrekte Knoten ist dabei der Ausgabe des `srun`-Befehles zu entnehmen. Eine beispielhafte Ausgabe ist in Quelltext 18 zu sehen.

```

1 muendlar@vmschulz8:~/eragp-mandelbrot/backend/build$ srun -N4 -n5 -l --
   multi-prog ..../himmuc/run.conf
2 srun: error: Could not find executable worker
3 4: Worker: 4 of 5 on node rpi06
4 2: Worker: 2 of 5 on node rpi04
5 3: Worker: 3 of 5 on node rpi05
6 0: Host: 0 of 5 on node rpi03
7 0: Host init 5
8 1: Worker: 1 of 5 on node rpi03
9 0: Core 1 ready!
10 1: Worker 1 is ready to receive Data.
11 2: Worker 2 is ready to receive Data.
12 0: Listening for connections on to websocket server on 9002
13 0: Core 2 ready!
14 3: Worker 3 is ready to receive Data.
15 0: Core 3 ready!
16 4: Worker 4 is ready to receive Data.
17 0: Core 4 ready!
18 muendlar@vmschulz8:~/eragp-mandelbrot/backend/build$ ssh ssh -L
   0.0.0.0:9002:localhost:9002 -fN -M -S .tunnel.ssh rpi03
```

Quelltext 18: Beispielhafter Start des Backends. Hierbei ist der Knoten des Hostprozesses `rpi03`.

**Stoppen des Backends** Um das Backend wieder zu stoppen, müssen der ssh-Tunnel zur Verbindung der Ports und der `srun`-Prozess gestoppt werden. Letzterer lässt sich nach dem dämonisieren im vorigen Aufruf nur über die Prozess-ID finden. Diese zeigt das Tool `ps` an.

```

1 ssh -S .tunnel.ssh -O exit rpi<host number>
2 # To stop the node allocation
3 scancel -u <user name>
```

## 8.2 Detaillierte Beschreibung der Header und der CMake Instruktionen

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<sup>33</sup> in der hier beschriebenen Reihenfolge spezifiziert. Es sollte hierbei eine CMake-Version über 3.7.0 gewählt werden und die C++11 Standards<sup>34</sup> werden vorausgesetzt. 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 Bibliotheken verfügbar zu machen werden anschließend die Header der installierten MPI-Bibliothek sowie die Header der Bibliotheken rapidjson<sup>35</sup>, websocketpp<sup>36</sup> und boost<sup>37</sup>. Diese werden respektive verwendet um JSON zu parsen und enkodieren, WebSocket-Verbindungen aufzubauen und darüber zu kommunizieren 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, die Optimierung auf die höchste Stufe gesetzt und spezielle Flags für die Websocketlibrary und MPI gesetzt.

## 8.3 Beispielhafte Einbindung der MPI-Initialisierungsfunktion

Ein beispielhafter Aufruf der Prozessinitialisierung von einer Mainfunktion aus ist in Quelltext 19 zu sehen. Damit wird MPI initialisiert und nach der erfolgreichen Initialisierung der eigentliche Host-Prozess über Host::init gestartet.

## 8.4 Detaillierter Ablauf der Host::handle\_region\_request Methode

Nach dem Parsen der Nachricht wird die Region global festgelegt und beim korrekten Lastbalancierer eine Zerlegung der angefragten Region über Balancer::balanceLoad gestartet.

<sup>33</sup>Ein Programm, welches die Erstellung von Makefiles vereinfacht in dem es sie automatisch an die Umgebung des Build-Systems anpasst. <https://cmake.org/>

<sup>34</sup><https://isocpp.org/wiki/faq/cpp11>

<sup>35</sup><http://rapidjson.org>

<sup>36</sup><https://github.com/zaphoyd/websocketpp>

<sup>37</sup><https://www.boost.org/>

```

1 #include "init.h"
2 #include "Host.h"
3
4 int main(int argc, char *argv[])
5 {
6     return init(argc, argv, "Host", Host::init);
7 }

```

Quelltext 19: Initialisierung des Host-Prozesses in host.main.cpp

Jeder Region wird ein Worker zugewiesen. Dabei ist der Rang des Workers, der eine Region berechnen soll genau der Index der Region in dem zurückgegebenen Array. Ist ein Worker nicht verfügbar, so werden sein und alle folgenden Ränge um eins erhöht (siehe Quelltext 21). Damit kann der WebSocketprozess unabhängig von der tatsächlichen Verteilung den Rang des berechnenden Worker-Prozesses bestimmen und in der Antwort an den Client zu der Aufteilung der Prozesse einfügen. Die Struktur und eine gültige Antwort des Websocketservers kann Quelltext 22 entnommen werden.

Nachdem die Aufteilung als Antwort auf die Anfrage dem WebSocketprozess bereitgestellt wurde, werden die Teilregionen in eine per Mutexlock-gesicherte Datenstruktur gelegt. Der MPI verwendende Thread, der diese Regionen anschließend an die Worker sendet wird über das Setzen des booleschen Wertes `mpi_set_regions` auf `true` darüber informiert, dass neue Regionen zum versenden zur Verfügung stehen.

Indem die Regionaufteilung zuerst an das Frontend und anschließend an die Worker gesendet wird, wird sichergestellt, dass das Frontend alle empfangenen Regionsdaten korrekt der angefragten Region zuordnen kann. Dabei wird zwar Rechenzeit verschenkt, jedoch steigt durch die Erstellung sehr kleiner oder leerer Teilregionen die Wahrscheinlichkeit, dass ein Arbeiter mit seiner Region zu früh abgeschlossen ist. Würde dann dessen Region vor der Gesamtaufteilung beim Frontend ankommen, besteht die Gefahr, dass es die Daten verwirft.

## 8.5 Beispiele für versendete Daten

## 8.6 Nutzbarkeit einzelner Worker

Es besteht die Möglichkeit, explizit zu definieren ob ein Worker Rechenaufträge bekommen soll oder nicht. Hierzu existiert im Host das globale Boolean-Array `usable_nodes`. Ein Prozess mit Rang  $x$  ist benutzbar, wenn `usable_nodes` am Index  $x$  `true` ist. Nicht benutzbar ist er, wenn die Variable auf `false` gesetzt ist. Zudem existiert noch die globale Variable `usable_nodes_count`, die angibt wie viele Worker benutzbar sind. Genutzt wird dieses Feature, falls beim Initialen Test aller Worker `Worker` als nicht benutzbar eingestuft werden.

**Initialer Test aller Worker** Bevor Rechenaufträge an die Worker weitergeleitet werden, wird zunächst jeder Worker einzeln getestet. Dies dient auch dazu, den Workern den Rang des Hosts zu übermitteln.

```

15 int init(int argc, char **argv, const char* type, void (*initFunc) (int
16   world_rank, int world_size)) {
17
18   // Set thread level to MPI_THREAD_FUNNELED. We use MPI in exactly one
19   // thread per actor.
20   int providedMPIThreadLevel;
21   MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &
22     providedMPIThreadLevel);
23
24   // std::cout << "Provided thread level: " << providedMPIThreadLevel
25   // << " ; Needed thread level: " << MPI_THREAD_FUNNELED << std::endl;
26
27   if (providedMPIThreadLevel < MPI_THREAD_FUNNELED) {
28     std::cerr << "MPI thread level support is insufficient." << std::endl;
29     MPI_Abort(MPI_COMM_WORLD, 1);
30     return -1; // Return with error
31   }
32
33   int world_rank;
34   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
35   int world_size;
36   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
37   // Retreive the processor name to check if host and
38   // worker share a node
39   char* proc_name = new char[MPI_MAX_PROCESSOR_NAME];
40   int proc_name_length;
41   MPI_Get_processor_name(proc_name, &proc_name_length);
42   std::cout << type << ":" << world_rank << " of " << world_size <<
43   " on node " << proc_name << std::endl;

```

Quelltext 20: Initialisierung der MPI-Prozesse in init.cpp

```

222 Balancer *b = BalancerPolicy::chooseBalancer(balancer, fractal_bal);
223 // Measure time needed for balancing - Start
224 std::chrono::high_resolution_clock::time_point balancerTimeStart =
225   std::chrono::high_resolution_clock::now();
226 // Call balanceLoad
227 Region *blocks = b->balanceLoad(region, regionCount); // Blocks is
228   array with regionCount members
229 // Measure time needed for balancing - End
230 std::chrono::high_resolution_clock::time_point balancerTimeEnd = std
231   ::chrono::high_resolution_clock::now();
232 unsigned long balancerTime = std::chrono::duration_cast<std::chrono::
233   microseconds>(balancerTimeEnd - balancerTimeStart).count();
234 std::cout << "Balancing took " << balancerTime << " microseconds." <<
235   std::endl;

```

Quelltext 21: Algorithmus zur Zuordnung von Regionen auf Worker in Host.cpp

```
1  {
2      "type": "region",
3      "regionCount": 37,
4      "regions": [
5          {
6              "rank": 1,
7              "computationTime": 0,
8              "region": {
9                  "minReal": -0.251953125,
10                 "maxImag": -0.8408203125,
11                 "maxReal": -0.24609375,
12                 "minImag": -0.8427734375,
13                 "width": 384,
14                 "height": 128,
15                 "hOffset": 0,
16                 "vOffset": 0,
17                 "maxIteration": 1019,
18                 "validation": 8,
19                 "guaranteedDivisor": 64
20             }
21         },
22         {
23             "rank": 2,
24             "computationTime": 0,
25             "region": {
26                 "minReal": -0.251953125,
27                 .....
28             },
29             {
30                 "rank": 37,
31                 "computationTime": 0,
32                 "region": {
33                     "minReal": -0.2275390625,
34                     "maxImag": -0.84765625,
35                     "maxReal": -0.2216796875,
36                     "minImag": -0.8486328125,
37                     "width": 384,
38                     "height": 64,
39                     "hOffset": 1600,
40                     "vOffset": 448,
41                     "maxIteration": 1019,
42                     "validation": 8,
43                     "guaranteedDivisor": 64
44                 }
45             }
46         }
47     ]
48 }
```

Quelltext 22: Ausschnitt aus einer gültigen Antwort auf die Region aus Quelltext 1 in JSON

```

1  {
2      "type": "regionData",
3      "workerInfo": {
4          "rank": 3,
5          "computationTime": 1120852,
6          "region": {
7              "minReal": 0.33984375,
8              "maxImag": -0.583984375,
9              "maxReal": 0.400390625,
10             "minImag": -0.5859375,
11             "width": 1984,
12             "height": 64,
13             "hOffset": 0,
14             "vOffset": 128,
15             "maxIteration": 1019,
16             "validation": 7,
17             "guaranteedDivisor": 64
18         }
19     },
20     "data": [22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
21         22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
24, 24, 24, 24, 24, 24, ...]
21 }

```

Quelltext 23: Ausschnitt aus den Daten einer versendeten Teilregion. Punkt (x,y) liegt in "data" an Index  $i = x + y * width$ .

Dazu sendet der Host mittels der nicht-blockierenden, synchronen Sendeoperation MPI\_Issend einen Test-Wert an alle Worker. Durch die synchrone Eigenschaft wird die Operation erst abgeschlossen, wenn eine passende Empfangsoperation gestartet und der Sendeoperation zugeordnet wurde. Es wurde eine nicht-blockierende Sendeoperation gewählt, um den Test für alle Worker parallel ausführen zu können.

Wurden alle Sendeoperationen gestartet, muss auf deren Abschluss gewartet werden. Falls nun ein Worker nicht wie erwartet reagiert und die Empfangsoperation nicht durchführt, muss das Warten auf den Abschluss der Sendeoperation abgebrochen werden können. Deshalb wird MPI\_Testsome in einer Endlosschleife genutzt, die

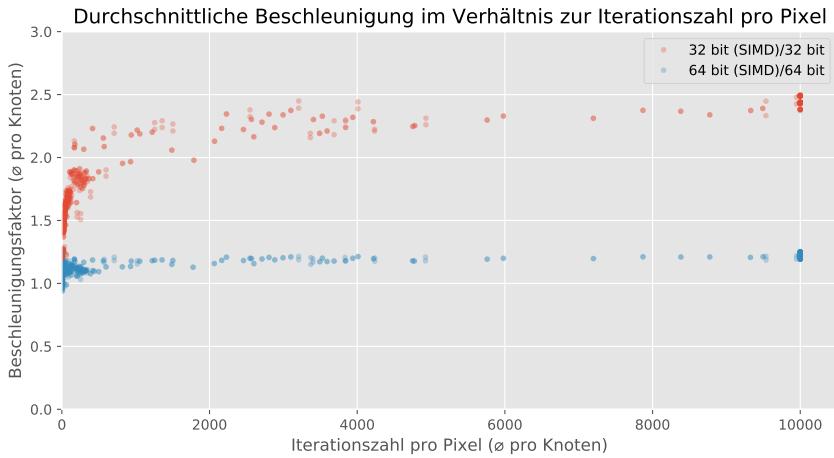


Abbildung 19: Beschleunigungsfaktor durch SIMD in Abhängigkeit von der durchschnittlichen Iterationszahl. Auswertung von 360 Regionen mit zwei Durchläufen. Die Iterationszahl für einen Abbruch war 10000.

nur abbricht falls alle Sendeoperationen erfolgreich waren (alle MPI\_Request sind MPI\_REQUEST\_NULL, wodurch MPI\_Testsome in outcount MPI\_UNDEFINED zurückgibt) oder ein Timer abgelaufen ist. Falls eine oder mehrere Sendeoperationen innerhalb eines Schleifendurchlaufs erfolgreich abgeschlossen wurden, wird deren MPI\_Request auf MPI\_REQUEST\_NULL gesetzt um dies explizit zu zeigen. Sie werden in den nächsten Schleifendurchläufen von MPI\_Testsome ignoriert.

Nachdem die Schleife abgebrochen wurde, muss noch überprüft werden, welche Sendeoperationen erfolgreich waren und welche nicht. War die Operation erfolgreich, wird der entsprechende Worker als nutzbar eingestuft. Falls nicht, wird die noch laufende Sendeoperation mit MPI\_Cancel abgebrochen und der Worker als nicht benutzbar eingestuft.

## 8.7 Entwicklung des Beschleunigungsfaktors von SIMD für höhere Iterationszahlen

Wie durch den konstanten Mehraufwand der SIMD-Verwendung zu erwarten, steigt der Beschleunigungsfaktor monoton und wächst ab circa 6000 Iterationen pro Pixel nicht mehr wesentlich. Dies kann Abbildung 19 entnommen werden.

Dass hier auch nicht in der Homogenität der Iterationszahlen in Bereichen hoher Rechenintensität liegt, zeigt Abbildung 20. Sie hat jedoch einen deutlich erkennbaren Einfluss auf die Beschleunigung. Während in bei einem frühzeitigen Abbruch bei 1019 Iterationen eine Beschleunigung von über 2 erst ab etwa 200 Iterationen gemessen wird, tritt sie bei maximal 150 Iterationen bereits ab etwa 140 auf.

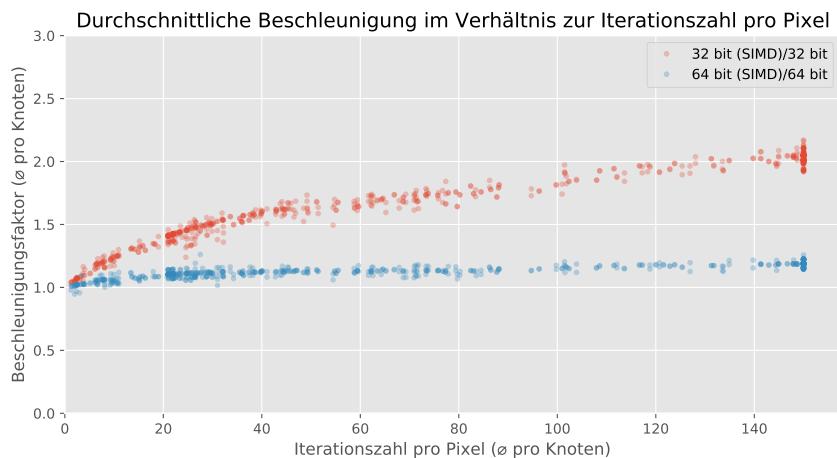


Abbildung 20: Beschleunigungsfaktor durch SIMD in Abhängigkeit von der durchschnittlichen Iterationszahl. Auswertung von 360 Regionen mit zwei Durchläufen. Die Iterationszahl für einen Abbruch war 150.

## 8.8 Kommunikation der Prozesse per MPI

Das Message Passing Interface (MPI) wird ausschließlich im Backend zur Kommunikation zwischen dem Host und den Workern verwendet.

**MPI und Threads** Da sowohl im Host, als auch in den Workern mehrere Threads arbeiten, muss die Kompatibilität von MPI mit Threads genau betrachtet werden. Laut der offiziellen MPI Dokumentation<sup>38</sup> müssen die konkreten Implementierungen von MPI keinerlei Threads unterstützen, die meisten bekannten Implementierungen (wie beispielsweise Open MPI<sup>39</sup>) tun dies aber, falls sie entsprechend konfiguriert wurden.

Es wurde besonders darauf geachtet, dass nur ein Thread pro Prozess MPI-Aufrufe tätigt, wodurch die Thread-Umgebung mit MPI\_THREAD\_FUNNELED initialisiert werden kann. Dadurch kann MPI einige Optimierungen durchführen, die nicht möglich wären, wenn mehrere Threads MPI-Aufrufe tätigen. Zudem sind die meisten MPI Implementierungen noch nicht auf eine performante Umsetzung von mehreren Threads ausgelegt, weshalb das Thread-Level so niedrig wie möglich gehalten werden sollte.

**Busy-Waiting von MPI** Bei den verwendeten Implementierungen von MPI, MPICH und OpenMP, wird bei blockierenden Nachrichtensende und -empfangsoperationen Busy-Waiting umgesetzt. Das führt einerseits dazu, dass die Verzögerung beim Empfangen von Nachrichten minimiert wird und die Bearbeitung der Daten sofort starten kann. Andererseits führt das aber zu einer 100-Prozentigen Auslastung des Rechenkerns,

<sup>38</sup><https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

<sup>39</sup>[https://www.open-mpi.org/doc/v3.1/man3/MPI\\_Init\\_thread.3.php](https://www.open-mpi.org/doc/v3.1/man3/MPI_Init_thread.3.php)

was andere Prozesse von der Durchführung ihrer Arbeit abhalten kann und zu einem erhöhten Stromverbrauch führt.

Im Fall des blockierenden MPI\_Recv wird solange aktiv ohne Pause getestet, ob eine Nachricht empfangen werden kann, bis dies der Fall ist. Falls eine nicht-blockierende Empfangsoperation mit MPI\_Irecv gestartet wird, muss diese anschließend auch abgeschlossen werden. Dazu kann einerseits MPI\_Wait dienen, das solange wartet, bis die Empfangsoperation abgeschlossen ist. Hierzu wird ebenfalls Busy-Waiting nutzt. Andererseits kann auch MPI\_Test genutzt werden, was nur überprüft, ob die Empfangsoperation abgeschlossen ist ohne auf deren Abschluss zu warten. Auch hier gibt es keine andere Möglichkeit, als immer wieder (beispielsweise in einer Schleife) MPI\_Test aufzurufen. Der einzige Vorteil besteht darin, dass zwischen diesen wiederholten Aufrufen andere Arbeit erledigt oder der Prozess für eine gewisse Zeit schlafen gelegt werden kann. Mit MPI\_Probe bzw. MPI\_Iprobe verhält es sich ähnlich wie mit MPI\_Recv bzw. MPI\_Test. Also wird auch hier Busy-Waiting eingesetzt.

In der Implementierung wurde deshalb im Host und im Worker eine nicht-blockierende Empfangsmethode gewählt, die den Thread für x Millisekunden schlafen legt, bevor Überprüft wird, ob Daten empfangen wurden. Die damit einhergehende Verzögerung verfälscht das Ergebnis nicht schwerwiegend, da die Berechnung einer Region für gewöhnlich deutlich länger als y Millisekunden benötigt.

### 8.8.1 Genereller Aufbau der MPI-Kommunikation

Im Folgenden wird kurz der generelle Aufbau der MPI-Kommunikationsroutinen im Host und in den Workern erklärt.

**MPI-Kommunikation im Host** Die essenziellen Teile der MPI-Kommunikation des Hosts, also das Senden von Rechenaufträgen und das Empfangen der berechneten Daten, befinden sich in einer gemeinsamen Endlosschleife.

1. Ist das Flag gesetzt, dass neue Rechenaufträge an die Worker zu senden sind
    - a) Sende alle Aufträge per MPI an die jeweiligen Worker. Der Rang bestimmt sich aus dem Index in der Datenstruktur für die Rechenaufträge.
  2. Können berechnete Daten von einem der Worker empfangen werden
    - a) Formatiere die empfangenen Daten, um Metainformationen hinzuzufügen
    - b) Locke die mit dem WebSocket-Sende Thread geteilte Datenstruktur.
    - c) Lege die Daten in die Datenstruktur
    - d) Löse das Lock
  3. Wurde keine Operation durchgeführt, schlafe  $z$  ms, um die Prozessorlast zu verringern
  4. Springe zu Schritt 1
-

Dabei wird immer zuerst überprüft, ob neue Rechenaufträge an die Worker zu senden sind. Gegebenenfalls werden die Sendeoperationen durchgeführt. Anschließend wird überprüft, ob berechnete Daten empfangen werden können. Ist das der Fall, so wird die Empfangsoperation durchgeführt, die Daten reorganisiert und über eine gemeinsame Datenstruktur an einen WebSocket-Thread weitergeleitet, der die Informationen an das Frontend reicht. Nun wird die Schleife von neuem begonnen. Es wird also sowohl für das Überprüfen, ob neue Rechenaufträge vorhanden sind, als auch für das Empfangen der berechneten Daten Busy-Waiting eingesetzt.

**MPI-Kommunikation im Worker** Auch hier wird das Empfangen der Rechenaufträge und das Senden der berechneten Daten in einer Endlosschleife bewerkstelligt. Zu Beginn wird so lange gewartet, bis ein neuer Rechenauftrag empfangen wurde. Direkt im Anschluss wird die Berechnung der empfangenen Region gestartet, wobei während der Kalkulation auf neu ankommende Rechenaufträge gelauscht wird. Ist ein neuer Auftrag zu empfangen, so wird die laufende Berechnung abgebrochen und die Schleife von vorne gestartet. Falls die Berechnung nicht unterbrochen wurde, werden die Daten organisiert und an den Host geschickt. Nun beginnt die Schleife von vorne.

Es ist auch hier zu erkennen, dass Busy-Waiting für das Empfangen von neuen Rechenaufträgen eingesetzt wird.

### 8.8.2 Übertragung neuer Rechenaufträge vom Host an die Worker

Um neue Rechenaufträge an die Worker zu übertragen, werden im Host und in den Workern Persistent Communication Requests verwendet. Das hat den Vorteil, dass der Kommunikationsoverhead zwischen dem Prozess und dem Kommunikationscontroller reduziert wird. Die Nutzung dieser Persistent Communication Requests ist möglich, da immer die selbe Datenmenge an die selben Empfänger gesendet wird bzw. vom selben Absender (das ist hier der Host) empfangen wird.

Zur Übertragung der neuen Rechenaufträge wird das Region-Struct verwendet, welches bereits durch einen WebSocket-Thread ausgefüllt wurde.

**Initialisierung der Persistent Communication Requests** Um die Persistent Communication Requests nutzen zu können, müssen diese zunächst initialisiert werden.

Im Host werden hierzu ein Array mit MPI\_Request, ein Array mit MPI\_Status und ein Array mit Region erstellt. So bekommt jeder Prozess seinen eigenen Request, Status und Puffer für die zu sendende Region. Der Index, an dem sich diese Objekte befinden ist immer der von MPI zugewiesene Rang des Prozesses. Anschließend wird die eigentliche Initialisierung der Persistent Communication Requests durch einen Aufruf von MPI\_Send\_init für jeden Prozess durchgeführt, wobei als Tag der Kommunikation die Zahl 1 gewählt wurde. Als Kommunikationsmodus wird eine nicht-blockierende Standardsendeoperation gewählt. Durch diesen Kommunikationsmodus wird die Sendeoperation nur gestartet, d.h. der Aufruf kehrt unter Umständen zurück, bevor die Sendeoperation abgeschlossen ist. Dadurch wird ein paralleles, möglichst schnelles

---

Senden aller Rechenaufträge gewährleistet. Mehr zur Durchführung des Sendens ist im nächsten Paragraphen zu finden. Die genaue Implementierung der Initialisierung ist in Quelltext 24 zu sehen.

Im Worker werden ebenfalls jeweils ein MPI\_Request, ein MPI\_Status und ein Region Objekt erstellt. Dann wird wieder der Persistent Communication Request durch einen Aufruf von MPI\_Recv\_init initialisiert, wobei der Tag wieder 1 und die Empfangsoperation nicht-blockierend ist. Der Einsatz der nicht-blockierenden Empfangsoperation stellt sicher, dass der Worker laufende Berechnungen bei Erhalt eines neuen Rechenauftrags abbricht und sofort mit der Berechnung der neuen Anfrage beginnt. Mehr hierzu im Paragraphen Empfangen der Rechenaufträge im Worker. Die Implementierung ist in Quelltext 25 zu sehen.

```

525     usable_nodes_count--;
526     // Cancel uncompleted send operations
527     MPI_Cancel(&test_requests[rank - acc]);
528     MPI_Status cancel_status;
529     MPI_Wait(&test_requests[rank - acc], &cancel_status);
530     int cancel_flag;
531     MPI_Test_cancelled(&cancel_status, &cancel_flag);

```

Quelltext 24: Initialisierung des Persistent Communication Requests im Host

```

56     // Init persistent asynchronous receive
57     Region newRegion;
58     MPI_Request request;
59     MPI_Recv_init(&newRegion, sizeof(Region), MPI_BYTE, host_rank, 1,
      MPI_COMM_WORLD, &request);

```

Quelltext 25: Initialisierung des Persistent Communication Requests im Worker

**Senden der Rechenaufträge im Host** Im Host sind neue Rechenaufträge immer dann an die Worker zu übertragen, wenn das Flag mpi\_send\_regions vom entsprechenden WebSocket-Thread gesetzt wurde. Hierbei wird Busy-Waiting eingesetzt, da das Empfangen von fertig berechnenden Regionen auch nur mit Busy-Waiting funktioniert und sich das Senden und Empfangen in der selben Schleife befindet. Mehr hierzu in Abschnitten Busy-Waiting und Genereller Aufbau der MPI-Kommunikation im Host.

Sind neue Rechenaufträge verfügbar, wird jede einzelne Region einem verfügbaren Worker zugeordnet, wobei jeder Worker maximal eine Region bekommt. Hierzu wird die Region aus der gemeinsamen Datenstruktur websocket\_request\_to\_mpi in den Sendedpuffer des Workers kopiert. Anschließend wird das nicht-blockierende standard Senden mit MPI\_Start gestartet. Die Zuordnung zwischen Rang des Workers und Index der Region wird dabei Deterministisch bestimmt so wie in Quelltext 21.

Falls nicht genügend Rechenprozesse zur Verfügung stehen (es also mehr zu berechnende Regionen als Rechenprozesse gibt), wird nur ein Fehler ausgegeben, da dieser Fall in der aktuellen Version unmöglich ist. Die restlichen Regionen, denen noch kein Rechenprozess zugeteilt wurde, werden nicht berechnet.

Um sicherzustellen, dass alle Sendeoperationen abgeschlossen sind, wird MPI\_Waitall eingesetzt, welches den Thread des Hosts so lange blockiert, bis die Daten aus allen Sendepuffern ausgelesen wurden. Es wird jedoch nicht gewartet, bis das entsprechende Empfangen in den Workern gestartet bzw. beendet wurde.

Der Code hierzu ist in Quelltext 26 zu sehen.

```

537         } else {
538             usable_nodes[rank] = true;
539             std::cout << "Host: Worker " << rank << " is usable." <<
540                 std::endl;
541         }
542     }
543     std::cout << "Host: There are " << usable_nodes_count << " usable
544         Worker." << std::endl;
545     // Test if all cores are available - end
546
547     // Approximately the time that MPI communication with one Worker has
548     // taken in microseconds
549     std::chrono::high_resolution_clock::time_point *mpiCommunicationStart
550         = new std::chrono::high_resolution_clock::time_point[world_size];
551
552     // Init persistent asynchronous send. Each process gets his own
553     // Request, Status and Buffer
554     MPI_Request region_requests[world_size];
555     MPI_Status region_status[world_size];
556     Region persistent_send_buffer[world_size];
557     for (int rank = 0 ; rank < world_size ; rank++) {
558         MPI_Send_init(&persistent_send_buffer[rank], sizeof(Region),
559                         MPI_BYTE, rank, 1, MPI_COMM_WORLD, &region_requests[rank]);
560     }
561
562     // MPI communication between Host and Workers (send region requests
563     // and receive computed data) and start websocket send
564     bool change = false;
565     while (true) {

```

Quelltext 26: Senden neuer Rechenaufträge im Host

**Empfangen der Rechenaufträge im Worker** Da der Worker laufende Berechnungen abbrechen soll, falls er einen neuen Rechenauftrag bekommt, wird eine nicht-blockierende Empfangsoperation verwendet (dessen Initialisierung wurde hier behandelt). Direkt nach der Initialisierung wird die Empfangsoperation mittels MPI\_Start gestartet, der Worker lauscht also nach neuen Rechenaufträgen und empfängt diese im Hintergrund während das Programm weiterläuft.

Direkt danach beginnt eine Endlosschleife (sie stellt den wichtigsten Teil des Workers dar), die zunächst MPI\_Test auruft, um zu überprüfen, ob ein neuer Rechenauftrag erfolgreich empfangen wurde.

Ist das nicht der Fall, so wird der Worker für eine Millisekunde gestoppt um die

Prozessorlast zu reduzieren. Anschließend wird die Schleife von neuem begonnen.

Konnte etwas empfangen werden, so wird die empfangene Region aus dem Empfangspuffer des Persistent Communication Requests (newRegion) in einen neuen Speicherplatz (region) kopiert. Nun wird wieder MPI\_Start aufgerufen, um nach neuen Rechenaufträgen lauschen zu können. Jetzt wird auch klar, warum der Puffer kopiert werden musste. Der Empfangspuffer wird nämlich wiederverwendet. Anschließend wird die Berechnung der empfangenen Region gestartet, wobei nach jedem berechneten Pixel mittels MPI\_Test überprüft wird, ob eine neue Region empfangen wurde. Ist das der Fall, so wird die Berechnung abgebrochen und die Endlosschleife von vorne begonnen. Falls nicht, wird mit der Berechnung des nächsten Pixels fortgefahrene.

Der entsprechende Code ist in Quelltext 27 zu finden.

```

59 // Listen for incoming messages
60 MPI_Start(&request);
61 // Start with actual work of this worker
62 while (true) {
63 // Test if receive operation is complete
64 MPI_Test(&request, &flag, &status);
65 if (flag != 0) {
66 // Set current region to newRegion, copy value explicitly
67 std::memcpy(&region, &newRegion, sizeof(Region));
68 // Listen for incoming messages
69 MPI_Start(&request);
70 // Loop over every pixel
71 for (...) {
72 // Compute pixel
73 f->calculateFractal(...)
74 // Test if receive operation is complete
75 MPI_Test(&request, &flag, &status);
76 if (flag != 0) {
77 // Start while loop again
78 }
79 }
80 } else {
81 // Reduce processor usage on idle
82 std::this_thread::sleep_for(std::chrono::milliseconds(1));
83 }
84 }
```

Quelltext 27: Empfangen neuer Rechenaufträge im Worker

### 8.8.3 Übertragung der berechneten Daten von den Workern zum Host

Um dem vom Worker berechneten Teil der Mandelbrotmenge und zusätzliche Informationen wie beispielsweise die Rechendauer an den Host zu schicken, wird ebenfalls wieder MPI benutzt.

**Struktur der zu übertragenden Daten** Die Daten des Workers setzen sich aus zwei Teilen zusammen. Zum einen werden allgemeine Informationen in Form eines WorkerInfo-

Structs an den Host geschickt. Dieses Struct beinhaltet den Rang des Workers, die Dauer der Berechnung und das zuvor empfangene Region-Struct, dass den Arbeitsauftrag enthält. Zum anderen müssen natürlich noch die berechneten Daten in Form eines unsigned short int Arrays gesendet werden.

Es ist dabei nicht praktikabel das Array mit den Daten in das WorkerInfo-Struct zu integrieren, da die Länge des Arrays von Region zu Region aufgrund des vorgenommenen Balancings unterschiedlich sein kann. Demnach wäre nur ein Pointer auf den wirklichen Speicherbereich (der außerhalb des Structs liegt) möglich. Wenn die Daten aber per MPI an einen anderen Prozess oder sogar an einen anderen Computer im Cluster gesendet werden sollen, dann ist es nicht zielführend, wenn man nur den Pointer überträgt und nicht die Daten selbst. Auf den vom Pointer referenzierten Speicherbereich kann nämlich gar nicht zugegriffen werden.

Zur effizienten Lösung dieses Problems werden die beiden Datensätze hintereinander in den Speicher kopiert, wobei zuerst das WorkerInfo-Struct und dann die berechneten Daten kommen. Der Quellcode dazu ist in Quelltext 28 zu sehen.

Das Array mit den Daten ist eine eindimensionale Repräsentation der Fraktalwerte der Region. Die Umrechnung von x und y Koordinate innerhalb des Bereiches auf den Index i im Array erfolgt dabei nach folgender Regel, wobei region die berechnete Region ist     $i = y * \text{region.width} + x$

```

154         for( int k = 0; k < vectorLength; k++){
155             projReal[k] = region.projectReal(x+k);
156             projImag[k] = region.projectImag(reverseY);
157         }
158         // Directly write them into our n array

```

Quelltext 28: Erstellen der Struktur aus WorkerInfo und den berechneten Daten

**Senden der berechneten Daten im Worker** Sobald die Berechnungen abgeschlossen sind, werden die Daten mit der blockierenden Standardsendeoperation MPI\_Send mit Tag 2 übertragen. Es wurde die blockierende Variante gewählt, da die Sendeoperation der komplett berechneten Region abgeschlossen sein soll, bevor die Berechnung einer neuen Region startet. Zudem benötigt das Senden im Vergleich zum Berechnen sehr wenig Zeit, so dass die Komplexität, die eine nicht-blockierende Behandlung mitbringen würde in keinem Verhältnis zum Nutzen steht.

Der Code ist in Quelltext 29 zu finden.

```

161         // Send "ret" to the Host using one MPI_Send operation
162         MPI_Send(ret, ret_len, MPI_BYTE, host_rank, 2, MPI_COMM_WORLD);

```

Quelltext 29: Senden der berechneten Daten im Worker

**Empfangen der berechneten Daten im Host** Das Empfangen der Daten gestaltet sich etwas schwieriger als das Senden. Zunächst wird mit dem nicht-blockierenden

MPI\_Iprobe überprüft, ob neue Daten mit dem Tag 2 empfangen werden können. Da mit Daten von jedem Worker empfangen werden können, wird als Source-Argument MPI\_ANY\_SOURCE angegeben. Liegen Daten zum empfangen an, so wird ein MPI\_Status-Objekt von MPI\_Iprobe ausgefüllt. Nun muss die Länge der zu empfangenden Daten ermittelt werden. Dazu wird MPI\_Get\_count mit dem gerade ausgefüllten Status-Objekt aufgerufen. Nachdem der Empfangspuffer in der entsprechenden Größe allokiert wurde, wird die Nachricht mit einem blockierendem MPI\_Recv empfangen. Es wurde der blockierende Empfangsmodus gewählt, da die Daten direkt im Anschluss weiterverarbeitet werden müssen und damit keine Arbeit zwischen dem Starten und Abschließen der Empfangsoperation erledigt werden kann. Nachdem MPI\_Recv zurückgekehrt ist, werden die Daten wieder entsprechend der Vorgabe entpackt. Dann wird das WorkerInfo-Struct zusammen mit den berechneten Daten und der gestoppten Zeit der MPI-Kommunikation im Struct RegionData gespeichert und an einen WebSocket-Thread übergeben.

Die Implementierung ist in Quelltext 30 zu sehen.

```
562         std::lock_guard<std::mutex> lock(
563             websocket_request_to_mpi_lock);
564         if (mpi_send_regions == true) {
565             unsigned int transmit_counter = 0;
566             for (int rank = 0 ; rank < world_size && transmit_counter
567                 < websocket_request_to_mpi.size() ; rank++) {
568                 if (usable_nodes[rank] == true) {
569                     std::cout << "Host: Start invoking core " << rank
570                     << std::endl;
571                     // Copy requested Region from joint datastructure
572                     // "websocket_request_to_mpi" to MPI Send buffer
573                     // "persistent_send_buffer"
574                     std::memcpy(&persistent_send_buffer[rank], &
575                         websocket_request_to_mpi[transmit_counter++],
576                         sizeof(Region));
577                     // Start the clock for MPI communication
578                     mpiCommunicationStart[rank] = std::chrono::
579                         high_resolution_clock::now();
580                     // Start send to one Worker using persistent
581                     // asynchronous send
582                     MPI_Start(&region_requests[rank]);
583                 }
584             }
585             if (transmit_counter != websocket_request_to_mpi.size())
586             {
587                 std::cerr << "Not enough Workers to compute all
588                     subregions." << std::endl;
589             }
590             std::cout << "Host: Start invoking all cores done." <<
591                 std::endl;
592             // Wait to complete all send operations
593             MPI_Waitall(world_size, region_requests, region_status);
594             std::cout << "Host: Waitall returned. All send operations
595                 are complete." << std::endl;
596             mpi_send_regions = false;
597             change = true;
598         }
599     }
```

Quelltext 30: Empfangen der berechneten Daten im Worker