

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

Maximilian Frühauf

Tobias Klausen

Florian Lercher

Niels Mündler

**Inhaltsverzeichnis**

<b>1 Einleitung</b>	<b>3</b>
1.1 Didaktische Ziele . . . . .	3
1.2 Verwendung der Mandelbrotmenge . . . . .	3
1.3 Darstellung der Mandelbrotmenge . . . . .	4
1.4 MPI . . . . .	5
1.5 SIMD . . . . .	5
1.6 TypeScript und React . . . . .	6
1.7 Qualitätsanforderungen . . . . .	6
1.8 Einschränkungen . . . . .	7
<b>2 Problemstellung und Motivation</b>	<b>8</b>
<b>3 Übersicht der Implementierung</b>	<b>9</b>
3.1 Lastbalancierung . . . . .	9
3.1.1 Naive Strategie . . . . .	10
3.1.2 Strategie mit Vorhersage . . . . .	10
3.1.3 Implementierungsvarianten . . . . .	10
<b>4 Installation der Anwendung</b>	<b>11</b>
4.1 Lokales Backend . . . . .	11
4.2 Backend auf HimMUC Cluster . . . . .	11
4.3 Installation des Frontends . . . . .	12
<b>5 Dokumentation der Implementierung</b>	<b>12</b>
5.1 Konzept der ausgetauschten Nachrichten . . . . .	12
5.2 Implementierung des Backends . . . . .	15
5.2.1 Inkludierte Header und CMake Anweisungen . . . . .	15
5.2.2 Mainfunktion und Initialisierung . . . . .	16
5.3 Host Funktionalitäten . . . . .	18
5.3.1 Websocketverbindung . . . . .	18
5.4 Implementierung der Lastbalancierung . . . . .	21
5.4.1 Naive Strategie . . . . .	21
5.4.2 Strategie mit Vorhersage . . . . .	23
5.4.3 Erweiterung . . . . .	24

---

5.5	Kommunikation der Prozesse per MPI . . . . .	25
5.5.1	Genereller Aufbau der MPI-Kommunikation . . . . .	27
5.5.2	Übertragung neuer Rechenaufträge vom Host an die Worker . . . . .	28
5.5.3	Übertragung der berechneten Daten von den Workern zum Host . . . . .	34
5.6	Berechnung der Mandelbrotmenge . . . . .	35
5.6.1	Berechnung ohne SIMD . . . . .	35
5.6.2	Berechnung mithilfe von SIMD . . . . .	37
5.7	Implementierung des Frontends . . . . .	37
5.7.1	Kommunikation mit dem Backend . . . . .	37
5.7.2	Darstellung der Regionsdaten . . . . .	40
5.7.3	Visualisierung der Architektur des Backends . . . . .	43
5.7.4	Visualisierung der Rechenzeiten . . . . .	43
5.7.5	MISC . . . . .	44
<b>6</b>	<b>Ergebnisse / Evaluation</b>	<b>45</b>
6.1	Performanzerhöhung alternative Parallelisierungsmechanismen . . . . .	45
6.1.1	SIMD . . . . .	45
<b>7</b>	<b>Zusammenfassung</b>	<b>47</b>
<b>8</b>	<b>Anhang</b>	<b>49</b>
8.1	Detaillierter Start des Backends auf dem HIMMUC . . . . .	49

## 1 Einleitung

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

### 1.1 Didaktische Ziele

Das zugrundeliegende Problem ist, dass bei der Lastaufteilung einer unabhängigen Menge von Berechnungen in einem Cluster eine fixe Zuordnung von zu 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. Da die Kerne eines Clusters jedoch darauf ausgelegt sind, ständig zu arbeiten, sollte dieser Zustand vermieden werden, um Zeit, Kosten und Energie zu sparen.

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

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

### 1.2 Verwendung der Mandelbrotmenge

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

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

In der Mandelbrotmenge befinden sich alle  $c$ , für die der Betrag von  $z_n$  für beliebig große  $n$  endlich bleibt. Wenn der Betrag von  $z$  nach einer Iteration größer als 2 ist, so strebt  $z$  gegen unendlich, das zugehörige  $c$  liegt also nicht in der Menge. Sobald  $|z_n| > 2$  kann die Berechnung daher abgebrochen werden [1].

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

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

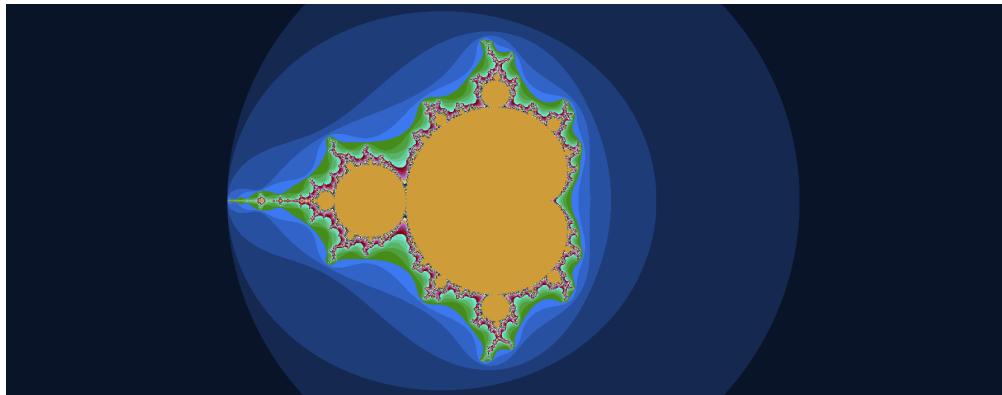


Abbildung 1: Die Mandelbrotmenge, visualisiert in einem Ausschnitt des komplexen Zahlenraumes.

Diese Eigenschaften ermöglichen es, zweierlei Dinge zu kontrollieren:

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

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

### 1.3 Darstellung der Mandelbrotmenge

Eine Komplexe Zahl  $c \in \mathbb{C}$  lässt sich auch 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 der Raum jedoch diskretisiert, indem jedem Pixel des Bildschirmes die komplexen Koordinaten  $c$  der linken oberen Ecke zugeordnet werden.

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

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

---

## 1.4 MPI

Das Message Passing Interface<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. Für dieses Projekt wichtig ist, dass es echte Parallelisierung mit geringem Overhead ermöglicht. So können die einzelnen Berechnungen auf jeweils eigenen unabhängigen Rechenkernen laufen und die Art der Aufteilung erhält größtmögliche Bedeutung. Die Gestaltung von MPI erlaubt dabei beliebige Zuordnungen, von Kernen auf einem Prozessor bis hin zu unabhängigen Clusterknoten, die lediglich eine SSH-Verbindung besitzen.

## 1.5 SIMD

„Simultaneous 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.

Dazu ist es hilfreich, sich zunächst vor Augen zu führen, wie ein Vektor komplexer Koordinaten ohne SIMD verarbeitet würde. Dies ist in Quelltext 1 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 solange das  $z_n$  der Koordinate Betragmäßig kleiner gleich 2 ist. Dies gilt, da alle  $|z_{n+i}| > 2$  sofern  $|z_n| > 2$  [1].

```

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

```

Quelltext 1: Bearbeitung eines Vektors komplexer Koordinaten in Pseudocode

Damit kann die Berechnung relativ simpel via Arm NEON Compiler Intrinsics<sup>2</sup> implementiert werden (siehe dazu Quelltext 25). 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.

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

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

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*“. Damit werden alle verfügbaren SIMD-Register des ARMv8-A Prozessors des ODroids und Raspberry Pi 3 B+ herangezogen. 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.

## 1.6 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 Webbrower 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 5.4). Jeder der vom Backend berechneten Teilregionen wird, sobald diese empfangen wurden, im Frontend angezeigt und die Komponenten zu Visualisierung der Rechenzeit aktualisiert.

## 1.7 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 Slider gewährleistet, die die Möglichkeit der Eingabe von ungültigen Werten verhindern.

---

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

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

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

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

## 1.8 Einschränkungen

Die Benutzeroberfläche soll in einem Webbrowser lauffähig sein, sodass sie auf beliebigen Endgeräten zugänglich gemacht werden kann. Um die erwartete Performancesteigerung an einem echten Beispiel zu demonstrieren, soll die Berechnung der Mandelbrotmenge parallel auf mehreren Raspberry Pi's<sup>6</sup> oder ähnlichen unabhängigen Kleincomputern oder Rechenkernen zum Einsatz kommen soll.

---

<sup>6</sup>Für ein Beispiel, siehe <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

## **2 Problemstellung und Motivation**

- Fachliche Spezifikation in Anhang
- NFRs in Einleitung schreiben

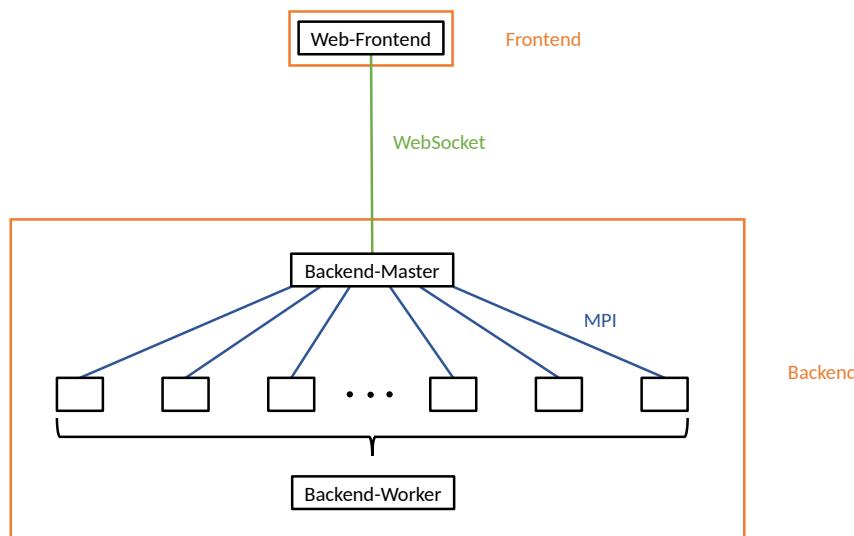


Abbildung 2: Architekturübersicht

### 3 Übersicht der Implementierung

Das Problem fordert eine Unterteilung in drei wesentliche Bausteine, wie sie in Abbildung 2 zu sehen sind. Eine Benutzeroberfläche in einem Web Browser des Benutzers (“Frontend”), welches die Benutzerinteraktionen entgegennimmt und mit dem Backend kommuniziert. Ein Backend-Host übernimmt dazu die Kommunikationsfunktion, verwaltet die eingehenden Rechenaufträge und verteilt sie an die Backend-Worker. Das Ergebnis dieser Berechnung sendet der Host dann an das Frontend zurück. Die Backend-Worker nehmen die zugewiesenen Rechenaufträge entgegen und führen die eigentlichen Berechnungen verteilt aus.

#### 3.1 Lastbalancierung

Um die Mandelbrotmenge effizient parallel zu berechnen, 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 diesem Projekt verschiedene Strategien der Lastbalancierung zur Wahl. Die Strategien lassen sich zur Laufzeit austauschen, um einen direkten Vergleich zu ermöglichen.

### 3.1.1 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.

### 3.1.2 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:

$$\frac{\text{Gesamtrechenlast}}{\text{AnzahlWorker}}$$

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.

**Bestimmung der Vorhersage** 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.

### 3.1.3 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 einfache 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.

## 4 Installation der Anwendung

Um das System zu installieren, muss das Repository mit git<sup>7</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>8</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>9</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>7</sup><https://git-scm.com/>

<sup>8</sup><https://www.docker.com/>

<sup>9</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>10</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>11</sup> (Erweiterung von JavaScript<sup>12</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>13</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.

## 5 Dokumentation der Implementierung

### 5.1 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.

---

<sup>10</sup>siehe ssh-copy-id

<sup>11</sup><https://www.typescriptlang.org/>

<sup>12</sup><https://en.wikipedia.org/wiki/JavaScript>

<sup>13</sup><https://www.npmjs.com/>

```

1 $ eragp-mandelbrot/backend$ python3 himmuc/start_himmuc.py muendlar 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
    mandelbrot
7 Search host node... srun: error: Could not find executable worker
odr00 found
8 Establish port 9002 forwarding to host node odr00:9002 ... established
9 System running. Websocket connection to backend is now available at
10 ws://himmuc.caps.in.tum.de:9002
11 Press enter (in doubt, twice) to stop Warning: Permanently added the
    ED25519 host key for IP address '10.42.0.54' to the list of known
    hosts.
12 # Enter
13
14 Stopping port forwarding... stopped (-9)
15 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

Ein Beispiel für eine gültige Regionsanfrage ist in Quelltext 7 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.

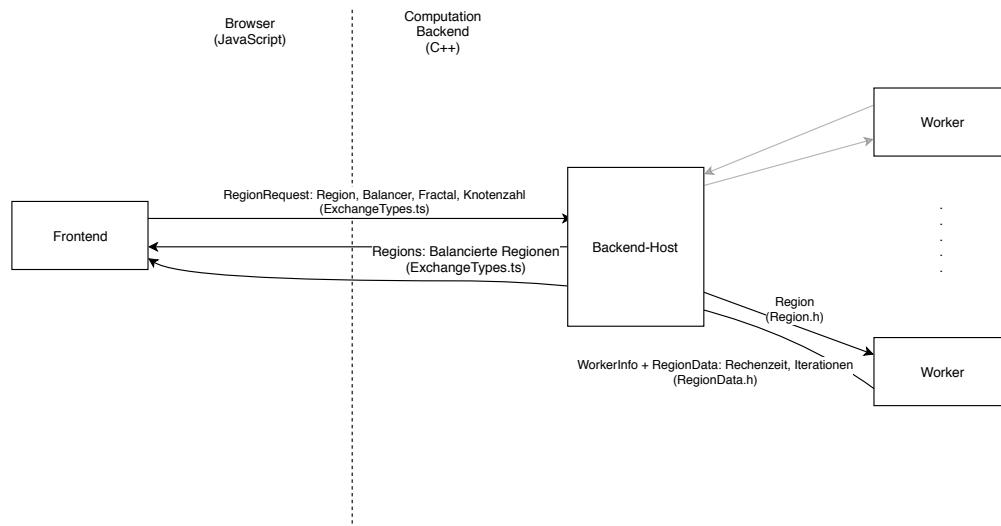


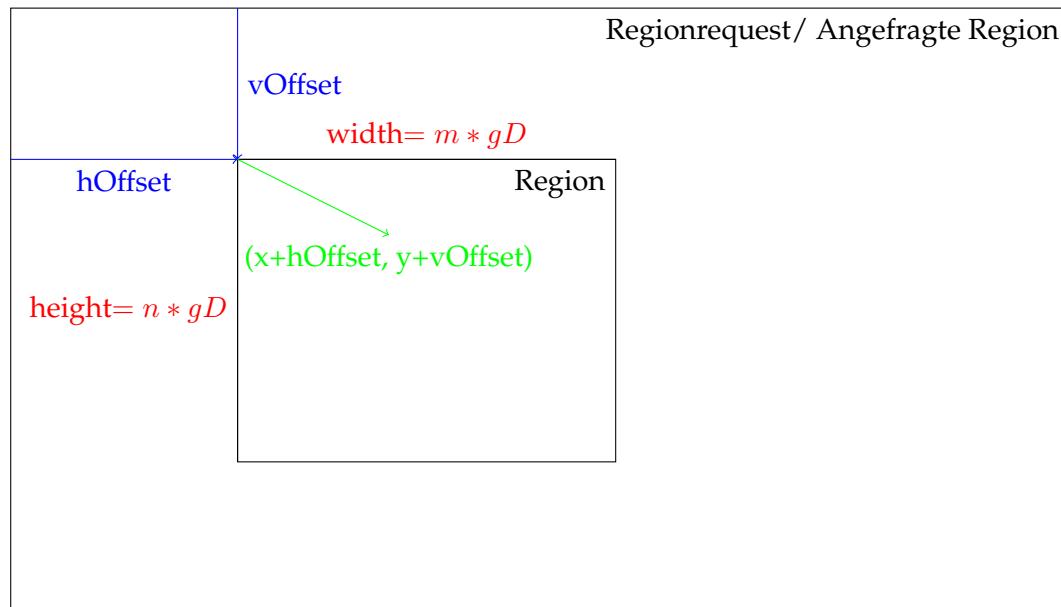
Abbildung 3: Konzept der versendeten Nachrichten. Die genauen Definitionen der Nachrichten sind in den angegebenen Dateien nachzusehen.

```

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 7: 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.

## 5.2 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.

### 5.2.1 Inkludierte Header und CMake Anweisungen

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

Diese und alle weiteren Build-Vorgaben werden in der Datei CMakeLists.txt für cmake<sup>14</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>15</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

<sup>14</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>15</sup><https://isocpp.org/wiki/faq/cpp11>

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>16</sup>, websocketpp<sup>17</sup> und boost<sup>18</sup>. Diese werden respektive verwendet um JSON zu parsen und encodieren, 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.

### 5.2.2 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 9. Sie erwartet lediglich eine Beschreibung des Prozesses für den Log und eine Initialisierungsfunktion, die erst zurückkehrt, wenn das Programm abgeschlossen ist und MPI beendet werden soll. Die Funktion muss als Parameter den Rang bzw. die Id des aktuellen MPI-Prozesses und die Anzahl der initialisierten Prozesse entgegen nehmen.

Ein beispielhafter Aufruf ist in Quelltext 8 zu sehen. Damit wird MPI initialisiert und nach der erfolgreichen Initialisierung der eigentliche Host-Prozess über Host::init gestartet.

```

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 8: Initialisierung des Host-Prozesses in host.main.cpp

---

<sup>16</sup><http://rapidjson.org>

<sup>17</sup><https://github.com/zaphoyd/websocketpp>

<sup>18</sup><https://www.boost.org/>

```
15 int init(int argc, char **argv, const char* type, void (*initFunc) (int
16   world_rank, int world_size)) {
17
18   MPI_Init(&argc, &argv);
19   int world_rank;
20   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
21   int world_size;
22   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
23   // Retreive the processor name to check if host and
24   // worker share a node
25   char* proc_name = new char[MPI_MAX_PROCESSOR_NAME];
26   int proc_name_length;
27   MPI_Get_processor_name(proc_name, &proc_name_length);
28   std::cout << type << ":" << world_rank << " of " << world_size <<
29     " on node " << proc_name << std::endl;
30
31   if (world_size < 2) {
32     std::cerr << "Need at least 2 processes to run. Currently have "
33       << world_size << std::endl;
34     MPI_Finalize();
35     return -1; // return with error
36   }
37   initFunc(world_rank, world_size);
38   MPI_Finalize();
39
40   return 0;
41 }
```

Quelltext 9: Initialisierung der MPI-Prozesse in init.cpp

## 5.3 Host Funktionalitäten

### 5.3.1 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. Die Methode `Host::start_server` initialisiert dabei lediglich den Websocketserver. Über den Typ des Servers wird eine Kompression versendeter Nachrichten über die im Standard akzeptierte Extension `perMessageDeflate` aktiviert. Zudem wird der Websocketserver mit `server.init_asio()` mit der Transport Policy "transport::asio" konfiguriert, sodass Multithreadzugriff auf Sende- und Empfangsmethoden problemlos möglich ist [2].

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: 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.

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 10 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.

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.

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

---

```

222 // Determine which Worker gets which Region
223 int region_to_worker[regionCount];
224 int counter = 0;
225 for (int rank = 0 ; rank < world_size && counter < regionCount ; rank
226    ++) {
227     if (usable_nodes[rank] == true) {
228         std::cout << "Region " << counter << " will be computed by
229             Worker " << rank << std::endl;
230         region_to_worker[counter] = rank;
231         counter++;
232     }
233 }
```

Quelltext 10: Algorithmus zur Zuordnung von Regionen auf Worker in Host.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 10). 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 11 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.

**Host::send** Während ein Thread das Empfangen von Nachrichten übernimmt, behandelt diese Methode von den Workern fertig berechnete Regionen. Er versendet in einer Dauerschleife Regionsdaten an den aktuell verbundenen Client. Nach jedem Durchlauf oder wenn der Thread per `notify` (oder auch durch mögliche andere Nebeneffekte) aufgeweckt wird, überprüft er auf das Vorhandensein neuer Regionen, wählt die erste aus und entfernt sie aus der Datenstruktur. Nachdem das Lock auf die geteilte Datenstruktur gelöst wurde, werden die Regionsdaten JSON codiert versendet. Ein Beispiel für versendete Regionsdaten kann Quelltext 12 entnommen werden. 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. Ist keine Region in der Datenstruktur abgelegt wird per `wait` auf eine Änderung gewartet. Mit-

```
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 11: Ausschnitt aus einer gültigen Antwort auf die Region aus Quelltext 7 in JSON

hilfe der `condition_variable`<sup>19</sup> `Host::mpi_to_websocket_result_available` nutzt sie die C++11 nativen mutex-Mechanismen um über das Vorhandensein neuer Regionen informiert zu werden.

## 5.4 Implementierung der Lastbalancierung

Der Implementierung der Lastbalancierung liegen die in Unterabschnitt 3.1 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>20</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<sup>21</sup>. So kann der Balancierer zur Laufzeit leicht gewechselt werden und auch die Erweiterung des Projekts um eine weitere Strategie gestaltet sich einfach. Was dabei genau beachtet werden muss findet sich im Teil Erweiterung (5.4.3).

### 5.4.1 Naive Strategie

Die naive Strategie (`NaiveBalancer` und `RecursiveNaiveBalancer`) lässt sich in beiden Varianten einfach nach dem oben beschriebenen Konzept umsetzen. 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 genaueren Teilung führt.

Die Teilung ansich 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.

<sup>19</sup>[https://en.cppreference.com/w/cpp/thread/condition\\_variable](https://en.cppreference.com/w/cpp/thread/condition_variable)

<sup>20</sup>Eine Region lässt sich also immer in eine ganzzahlige Anzahl von Tiles aufteilen, vgl. Abbildung 6

<sup>21</sup>[https://sourcemaking.com/design\\_patterns/strategy](https://sourcemaking.com/design_patterns/strategy)

Quelltext 12: Ausschnitt aus den Daten einer versendeten Teilregion. Punkt (x,y) liegt in "data" an Index  $i = x + y * width$ .

### 5.4.2 Strategie mit Vorhersage

Bei dieser Strategie (zu finden in den Klassen `PredictionBalancer` und `RecursivePredictionBalancer`) basiert die Aufteilung der Region auf einer Vorhersage über die Rechenzeit.

**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. Auch die Deltas für Real- und Imaginärteil pro Tile und die Anzahl der Zeilen und Spalten werden angegeben.

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

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

$$\text{desiredN} = \frac{nSum}{nodeCount} \quad (2)$$

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

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

Zur Aufteilung wird über die Spaltensummen in der Vorhersage iteriert (vgl. ??). Diese werden aufaddiert und bilden so den Zähler `currentN`. Sobald  $\text{currentN} \geq \text{desiredN}$  gilt oder für alle restlichen Spalten nur noch je ein Eintrag in den Spaltensummen vorhanden ist, wird eine Spalte abgeschlossen. Dazu werden `maxReal` und `width` aus den Zählern berechnet. Es ist wichtig `maxReal` immer neu aus `region.minReal` zu berechnen, anstatt nur das Delta aufzuaddieren, da sich sonst der Fehler, der bei Fließkomma-addition unvermeidbar ist, auch mit aufaddiert. `minReal` und `hOffset` stehen bereits in der Region `tmp`, die Werte wurden bei der Berechnung der vorhergehenden Spalte

```

1 #include "Region.h"
2
3 class Balancer {
4     public:
5         virtual Region* balanceLoad(Region region, int nodeCount) = 0;
6         virtual ~Balancer();
7 };

```

Quelltext 13: Das gemeinsame Interface der Lastbalancierung

bereits gesetzt. Jetzt wird tmp für die nächste Spalte vorbereitet, das heißt tmp.minReal wird auf tmp.maxReal gesetzt und tmp.hOffset wird um tmp.width erhöht. Erstes vermeidet das Entstehen von Lücken zuverlässig. Außerdem wird desiredN für die verbleibenden Spalten nach (2) neu berechnet und die Zähler werden zurückgesetzt. Bevor die Aufteilung der Spalte in Teilregionen startet, wird eine Kopie der Vorhersage erstellt, die nur die Werte für die aktuelle Spalte enthält. Das Aufteilen einer Spalte in Teilregion geschieht analog zum Aufteilen in Spalten.

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

**Rekursive Variante** Die rekursive Variante der Strategie mit Vorhersage verwendet dasselbe Rekursionsschema wie ihr naives Gegenstück. Die Abbruchbedingung ist also wie in (??). Auch die Entscheidung, ob horizontal oder vertikal geteilt werden soll, wird 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 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  $\frac{\text{context.partsLeft}}{2}$  Worker aufgeteilt zu werden. 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.

#### 5.4.3 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 13) 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

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.

## 5.5 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>22</sup> müssen die konkreten Implementierungen von MPI keinerlei Threads unterstützen, die meisten bekannten Implementierungen (wie beispielsweise Open MPI<sup>23</sup>) tun dies aber, falls sie entsprechend konfiguriert wurden.

<sup>22</sup><https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

<sup>23</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)

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** Da MPI grundsätzlich auf Höchstleistung ausgelegt ist, werden die Empfangsoperationen mit Busy-Waiting umgesetzt. Das führt einerseits dazu, dass ankommende Nachrichten ohne Verzögerung empfangen werden und die Bearbeitung der Daten sofort starten kann. Andererseits führt das aber zu einer 100-Prozentigen Auslastung des Rechenkerns, 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 wurde, bis dies auch wirklich geschehen 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 ein erneuter Empfangsversuch gestartet wird. Die damit einhergehende Verzögerung verfälscht das Ergebnis nicht schwerwiegend, da die Berechnung einer Region für gewöhnlich deutlich länger als y Millisekunden benötigt.

**Nutzbarkeit der Worker definieren** 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.

---

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. Der Source-Code ist in Quelltext 14 zu finden.

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 nur abbricht falls alle Sendeoperationen erfolgreich waren (alle MPI\_Requests 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. In Quelltext 15 ist der entsprechende Code zu finden.

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. Der Code kann in Quelltext 16 eingesehen werden.

### 5.5.1 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. 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

---

```

428 // Test if all cores are available
429 // This also allows the Workers to get the rank of the Host
430 MPI_Request test_requests[usable_nodes_count];
431 MPI_Status test_status[usable_nodes_count];
432 // Send test messages to every Worker
433 for (int rank = 0 ; rank < world_size ; rank++) {
434     if (rank != world_rank) {
435         // Skip Host
436         int acc = 0;
437         if (rank > world_rank) {
438             acc = 1;
439         }
440         int test_send = rank;
441         int ierr = MPI_Issend((const void *) &test_send, 1, MPI_INT,
442                               rank, 10, MPI_COMM_WORLD, &test_requests[rank - acc]);
443         // Error handling
444         if (ierr != MPI_SUCCESS){
445             std::cerr << "Error on MPI_Issend to worker " << rank <<
446                         " on test send: " << std::endl;
447             char err_buffer[MPI_MAX_ERROR_STRING];
448             int resultlen;
449             MPI_Error_string(ierr, err_buffer, &resultlen);
450             fprintf(stderr, err_buffer);
451         }
452         // Error handling - end
453     }
454 }

```

Quelltext 14: Starten der Sendeoperation zum Testen der Worker

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.

### 5.5.2 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.

```

453 // Start timer
454 std::chrono::high_resolution_clock::time_point test_time_start = std
455   ::chrono::high_resolution_clock::now();
456 unsigned long test_time = 0;
457 // Check if every Worker has started its receive operation
458 int *array_of_indices = new int[usable_nodes_count];
459 do {
460     int outcount;
461     int ierr = MPI_Testsome(usable_nodes_count, test_requests, &
462       outcount, array_of_indices, test_status);
463     // Error handling
464     if (ierr != MPI_SUCCESS){
465         for (int i = 0 ; i < usable_nodes_count ; i++) {
466             ierr = test_status[i].MPI_ERROR;
467             if (ierr != MPI_SUCCESS) {
468                 std::cerr << "Error on MPI_Testsome with index " << i
469                   << " on test send: " << std::endl;
470                 char err_buffer[MPI_MAX_ERROR_STRING];
471                 int resultlen;
472                 MPI_Error_string(ierr, err_buffer, &resultlen);
473                 fprintf(stderr, err_buffer);
474             }
475         }
476     }
477     // Error handling - end
478     // Every send operation was successful
479     if (outcount == MPI_UNDEFINED) {
480         std::cout << "Tests successful. Break test loop." << std::
481           endl;
482         break;
483     }
484     // One or more send operations were successful. Set their
485     // MPI_Request to MPI_REQUEST_NULL
486     if (outcount != 0) {
487         for (int i = 0 ; i < outcount ; i++) {
488             test_requests[array_of_indices[i]] = MPI_REQUEST_NULL;
489         }
490     }
491     // Check how much time has passed
492     std::chrono::high_resolution_clock::time_point test_time_end =
493       std::chrono::high_resolution_clock::now();
494     test_time = std::chrono::duration_cast<std::chrono::microseconds>(
495       (test_time_end - test_time_start).count());
496 } while (test_time < 1000000);
497 delete[] array_of_indices;

```

Quelltext 15: Warten auf den Abschluss aller Sendeoperationen des Workertests

**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

```

491     // Update usable_nodes and usable_nodes_count according to the test
492     // results
493     for (int rank = 0 ; rank < world_size ; rank++) {
494         if (rank != world_rank) {
495             // Skip Host.
496             int acc = 0;
497             if (rank > world_rank) {
498                 acc = 1;
499             }
500             if (test_requests[rank - acc] != MPI_REQUEST_NULL) {
501                 usable_nodes[rank] = false;
502                 usable_nodes_count--;
503                 // Cancel uncompleted send operations
504                 MPI_Cancel(&test_requests[rank - acc]);
505                 MPI_Status cancel_status;
506                 MPI_Wait(&test_requests[rank - acc], &cancel_status);
507                 int cancel_flag;
508                 MPI_Test_cancelled(&cancel_status, &cancel_flag);
509                 if (cancel_flag == true) {
510                     std::cout << "Host: Worker " << rank << " has NOT
511                         started his receive operation. This Worker is NOT
512                         usable. Cancel was successful." << std::endl;
513                 } else {
514                     std::cout << "Host: Worker " << rank << " has
515                         started his receive operation to late. This
516                         Worker is NOT usable. Cancel was NOT successful."
517                         << std::endl;
518                 }
519             std::cout << "Host: There are " << usable_nodes_count << " usable
520             Worker." << std::endl;

```

Quelltext 16: Einstufung der Worker als nutzbar bzw. nicht nutzbar entsprechend des Testergebnisses

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 Senden-Operation 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 17 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 18 zu sehen.

```

525 // Init persistent asynchronous send. Each process gets his own
526   Request, Status and Buffer
527   MPI_Request region_requests[world_size];
528   MPI_Status region_status[world_size];
529   Region persistent_send_buffer[world_size];
530   for (int rank = 0 ; rank < world_size ; rank++) {
531     MPI_Send_init(&persistent_send_buffer[rank], sizeof(Region),
532                   MPI_BYTE, rank, 1, MPI_COMM_WORLD, &region_requests[rank]);
531 }
```

Quelltext 17: 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,
60               MPI_COMM_WORLD, &request);
```

Quelltext 18: 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 Sendepuffer 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 10.

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 19 zu sehen.

```

537         std::lock_guard<std::mutex> lock(
538             websocket_request_to_mpi_lock);
539         if (mpi_send_regions == true) {
540             unsigned int transmit_counter = 0;
541             for (int rank = 0 ; rank < world_size && transmit_counter
542                 < websocket_request_to_mpi.size() ; rank++) {
543                 if (usable_nodes[rank] == true) {
544                     std::cout << "Host: Start invoking core " << rank
545                     << std::endl;
546                     // Copy requested Region from joint datastructure
547                     // "websocket_request_to_mpi" to MPI Send buffer
548                     // "persistent_send_buffer"
549                     std::memcpy(&persistent_send_buffer[rank], &
550                         websocket_request_to_mpi[transmit_counter++],
551                         sizeof(Region));
552                     // Start the clock for MPI communication
553                     mpiCommunicationStart[rank] = std::chrono::
554                         high_resolution_clock::now();
555                     // Start send to one Worker using persistent
556                     // asynchronous send
557                     MPI_Start(&region_requests[rank]);
558                 }
559             }
560             if (transmit_counter != websocket_request_to_mpi.size())
561             {
562                 std::cerr << "Not enough Workers to compute all
563                     subregions." << std::endl;
564             }
565             std::cout << "Host: Start invoking all cores done." <<
566                     std::endl;
567             // Wait to complete all send operations
568             MPI_Waitall(world_size, region_requests, region_status);
569             std::cout << "Host: Waitall returned. All send operations
570                     are complete." << std::endl;
571             mpi_send_regions = false;
572         }

```

Quelltext 19: 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 aufruft, 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 berechnetem 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 20 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 20: Empfangen neuer Rechenaufträge im Worker

### 5.5.3 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 21 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     workerInfo.region = region;
155
156     // Pack "workerInfo" and the computed "data" in one
157     // coherent storage area "ret"
158     const unsigned int ret_len = sizeof(unsigned short int) *
159         data_len + sizeof(WorkerInfo);
160     uint8_t* ret = new uint8_t[ret_len];

```

Quelltext 21: 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 22 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 22: 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. Damit 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 23 zu sehen.

## 5.6 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 24) 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.

### 5.6.1 Berechnung ohne SIMD

Die Klasse Mandelbrot dient dazu jeweils einen Punkt der Mandelbrotmenge zu berechnen. Da hierbei keine Vektorisierung stattfindet ist es sinnvoll vectorLength auf 0 zu setzen. Es ist zu beachten, dass cReal und cImaginary als Zeiger auf diese Werte zu

```

562     // Listen for incoming complete computations from workers (MPI)
563     // Receive one message of dynamic length containing "workerInfo"
564     // and the computed "worker_data"
565     MPI_Status status;
566     int probe_flag;
567     // Check if it is possible to receive a finished computation
568     MPI_Iprobe(MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &probe_flag, &
569     status); // Tag: 2
570     if (probe_flag == true) {
571         int recv_len;
572         // Determine the length of the incoming message
573         MPI_Get_count(&status, MPI_BYTE, &recv_len);
574         std::cout << "Host is receiving Data from Worker " << status.
575             MPI_SOURCE << " Total length: " << recv_len << " Bytes."
576             << std::endl;
577         // Allocate the receive buffer
578         uint8_t* recv = new uint8_t[recv_len];
579         // Execute the actual receive operation
580         int ierr = MPI_Recv(recv, recv_len, MPI_BYTE, status.
581             MPI_SOURCE, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // Tag:
582             2
583         // Error handling
584         if(ierr != MPI_SUCCESS){
585             std::cerr << "Error on receiving data from worker: " <<
586                 std::endl;
587             char err_buffer[MPI_MAX_ERROR_STRING];
588             int resultlen;
589             MPI_Error_string(ierr, err_buffer, &resultlen);
590             fprintf(stderr, err_buffer);
591             continue;
592         }
593         // Error handling - end

```

Quelltext 23: Empfangen der berechneten Daten im Worker

übergeben sind. Außerdem muss eine Speicherstelle für den Rückgabewert bereitgestellt und als dest übergeben werden. Diese Art und Weise der Übergabe macht ein gemeinsames Interface mit den vektorisierten Versionen möglich. Für die nicht parallele Berechnung wäre dies nicht nötig, da einzelne Werte auch direkt übergeben werden könnten.

Die Berechnung eines Punktes ist nun nicht weiter schwer, es muss lediglich Gleichung 1 als C++-Code umgesetzt werden. Bei  $|z_n| > 2$  oder äquivalent  $Re(z_n)^2 + Im(z_n)^2$  kann die Berechnung abgebrochen werden, da der Punkt sicher nicht in der Mandelbrotmenge liegt. Dies ist ebenfalls nötig wenn die maximale Anzahl an Iterationen erreicht wurde. Die benötigte Anzahl an Iterationen kann dann als Ergebnis zurückgegeben werden.

```

1 class Fractal {
2     public:
3     /**
4      * Calculates the fractal iteration value for a given complex (real,
5      *           imaginary) pair
6      * @param cReal real coordinate
7      * @param cImaginary imaginary coordinate
8      * @param maxIteration maximum amount of iterations to perform
9      * @param vectorLength number of coordinates to be calculated at once
10     * @param dest memory position where result(number of iterations for
11       coordinates) should be stored
12     */
13     virtual void calculateFractal(precision_t* cReal, precision_t*
14         cImaginary, unsigned short int maxIteration, int vectorLength,
15         unsigned short int* dest) = 0;

```

Quelltext 24: Das gemeinsame Interface der Fraktale

### 5.6.2 Berechnung mithilfe von SIMD

Bei der in Unterabschnitt 1.5 beschriebenen Implementierung einer Hardware-beschleunigten Variante mit SIMD, 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*), sodass die Summe der Anzahl nicht abgebrochener Berechnungen entspricht, werden daher alle Komponenten des Ergebnisvektors des Vergleiches mit 1 verundet. Das Ergebnis des Vergleiches und der Verundung ist ein Vektor vorzeichenloser 32 oder 64 bit Ganzzahlen, weshalb das Postfix für diese Operationen *u32* oder *u64* lautet.

## 5.7 Implementierung des Frontends

### 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 Websocketinterface `WebSocket`<sup>24</sup>. Bei der Initialisierung baut das erzeugte Objekt eine Verbindung zu der lokalen Adresse `ws://localhost:9002` auf<sup>25</sup>. Dort muss das Backend bereit sein, eine Websocketverbindung anzunehmen.

Die Klasse `WebSocketClient` bietet dabei folgende Methoden:

<sup>24</sup><https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>

<sup>25</sup>Faktisch wird eine Verbindung geöffnet, geschlossen und erneut geöffnet. Dies ist durch ein ungelöstes Problem beim Verbindungsauftbau bedingt, das dafür sorgt, dass ein Verbindungsauftbau erst bei der zweiten erzeugten Websocketverbindung fehlerfrei gelingt.

```

29 // Load casted values from array to SIMD vector
30 float32x4_t cReal = vdupq_n_f32(0); // = vld1q_f32(cRealArray); if
   casting weren't necessary this would work
31 cReal = vsetq_lane_f32((float32_t) cRealArray[0], cReal, 0);
32 cReal = vsetq_lane_f32((float32_t) cRealArray[1], cReal, 1);
33 cReal = vsetq_lane_f32((float32_t) cRealArray[2], cReal, 2);
34 cReal = vsetq_lane_f32((float32_t) cRealArray[3], cReal, 3);
35 float32x4_t cImaginary = vdupq_n_f32(0);
36 cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[0],
   cImaginary, 0);
37 cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[1],
   cImaginary, 1);
38 cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[2],
   cImaginary, 2);
39 cImaginary = vsetq_lane_f32((float32_t) cImaginaryArray[3],
   cImaginary, 3);
40 // The z values
41 float32x4_t zReal = vdupq_n_f32(0);
42 float32x4_t zImaginary = vdupq_n_f32(0);
43 // Helper variables
44 float32x4_t two = vdupq_n_f32(2);
45 float32x4_t four = vdupq_n_f32(4);
46 uint32x4_t one = vdupq_n_u32(1);
47 // result iterations
48 uint32x4_t n = vdupq_n_u32(0);
49 // vector with 1 if absolute value of component is less than two
50 uint32x4_t absLesserThanTwo = vdupq_n_u32(1);
51 int i = 0;
52 // addv => sum all elements of the vector
53 while(i < maxIteration && vaddvq_u32(absLesserThanTwo) > 0){
54     // mls a b c -> a - b*c
55     float32x4_t nextZReal = vaddq_f32(vmlsq_f32(vmulq_f32(zReal,
      zReal), zImaginary, zImaginary), cReal);
56     // mla a b c -> a + b*c
57     float32x4_t nextZImaginary = vmlaq_f32(cImaginary, two, vmulq_f32
      (zReal, zImaginary));
58     zReal = nextZReal;
59     zImaginary = nextZImaginary;
60     // Square of the absolute value -> determine when to stop
61     float32x4_t absSquare = vmlaq_f32(vmulq_f32(zReal, zReal),
      zImaginary, zImaginary);
62     // If square of the absolute is less than 4, abs<2 holds -> 1
       else 0
63     absLesserThanTwo = vandq_u32(vcsltq_f32(absSquare, four), one);
64     // if any value is 1 in the vector (abs<2) then dont break
65     n = vaddq_u32(n, absLesserThanTwo);
66     i++;
67 }
68 // write n to dest
69 dest[0] = vgetq_lane_u32(n, 0);
70 dest[1] = vgetq_lane_u32(n, 1);
71 dest[2] = vgetq_lane_u32(n, 2);
72 dest[3] = vgetq_lane_u32(n, 3);

```

Quelltext 25: Parallelisierte Bearbeitung eines Vektors komplexer Koordinaten in 32 bit Gleitkommapräzision in C++ mit Arm NEON Compiler Intrinsics für SIMD

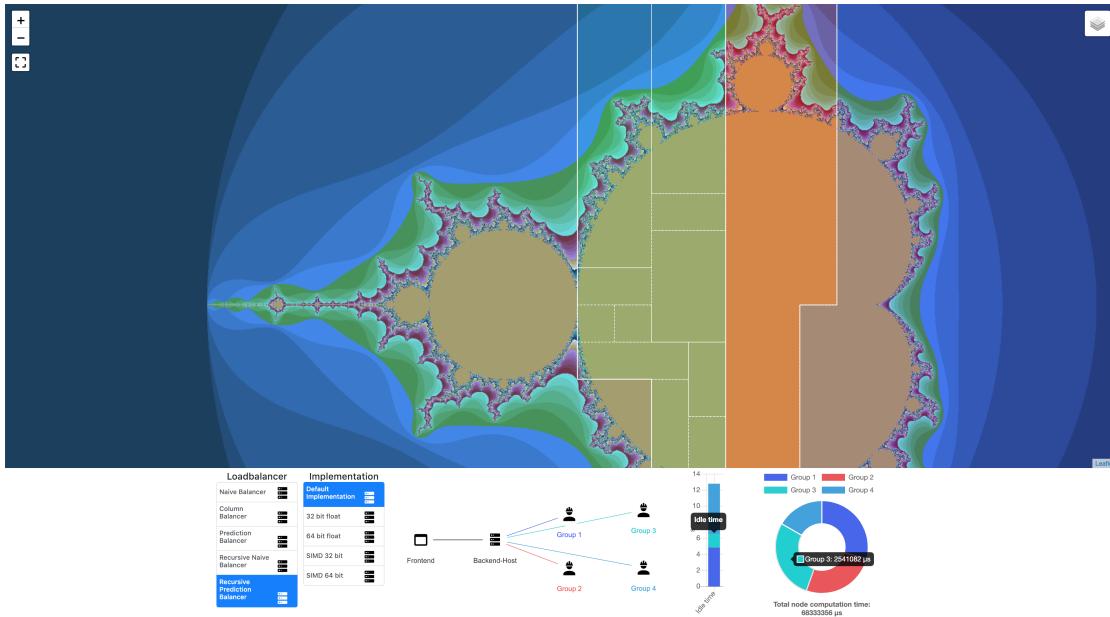


Abbildung 5: Benutzeroberfläche der Mandelbrot Anwendung

- `sendRequest(request)`

Diese Methode versendet das übergebene RegionRequest-Objekt codiert als JSON<sup>26</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 11) oder die berechneten Iterationswerte einer Region (siehe Quelltext 12).

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 empfangene Regionsdatenobjekt wird dann bei Empfang daraufhin überprüft, ob der darin dargestellte 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 Berechnungen aller Worker entgegengenommen wurden. Die Definitionen der zugehörigen

<sup>26</sup><https://www.json.org/>

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.

### 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 12).

- `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

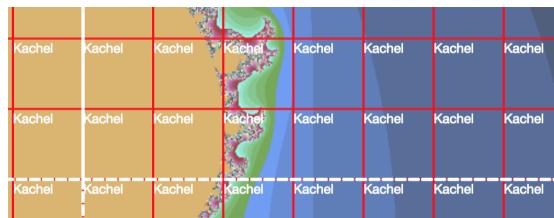


Abbildung 6: Relation von Backend Regionen zu leaflet Tiles. Dabei ist beispielhaft eine Region des Backends weiß eingezeichnet, alle leaflet Tiles sind rot umrandet angegeben.

und Gruppierung durch Observable Klassen (siehe Abschnitt 5.7.5) übergeben. Ebenfalls wird der anzugebende 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>27</sup> alle benötigten Tiles. Für jede der Tiles wird ein HTML5 canvas<sup>28</sup> Objekt erstellt, welches es ermöglicht, für jeden Pixel einen ( $r, g, b$ ) Farbwert zu definieren und anzuzeigen. Wobei die Farbwerte aus den berechneten Iterationswerten 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

<sup>27</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>28</sup>[https://developer.mozilla.org/kab/docs/Web/API/Canvas\\_API](https://developer.mozilla.org/kab/docs/Web/API/Canvas_API)

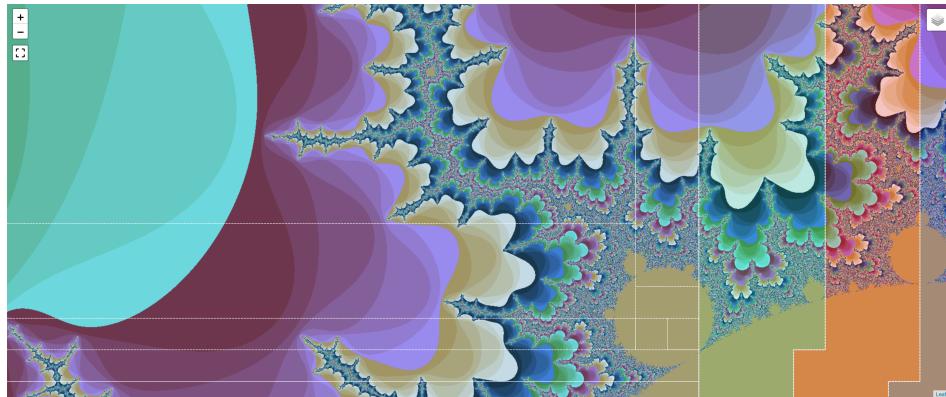


Abbildung 7: WorkerLayer für eine Gruppierung der Aufteilung des Recursive PredictionBalancers mit 37 Workern

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 26).

```

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 26: 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>29</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.

<sup>29</sup>[https://en.wikipedia.org/wiki/Spatial\\_reference\\_system](https://en.wikipedia.org/wiki/Spatial_reference_system)

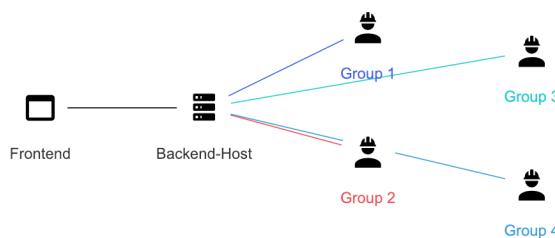


Abbildung 8: Darstellung der Architektur der Anwendung als Graph

### Regionsgruppierung (RegionGroup.ts)

#### 5.7.3 Visualisierung der Architektur des Backends

Die Struktur der Parallelisierung wird in einem Netzwerkgraphen unter dem Fraktal dargestellt. Mithilfe der Bibliothek visjs<sup>30</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 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>31</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).

<sup>30</sup><http://visjs.org/>

<sup>31</sup><http://www.chartjs.org/>

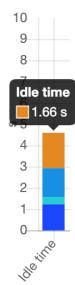


Abbildung 9: Idle Time für 5 Gruppen

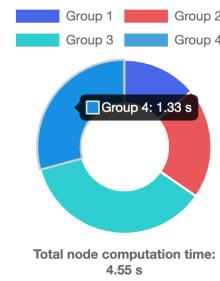


Abbildung 10: Computation time für 4 Gruppen

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.

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.

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

### 5.7.5 MISC

#### Observable.ts

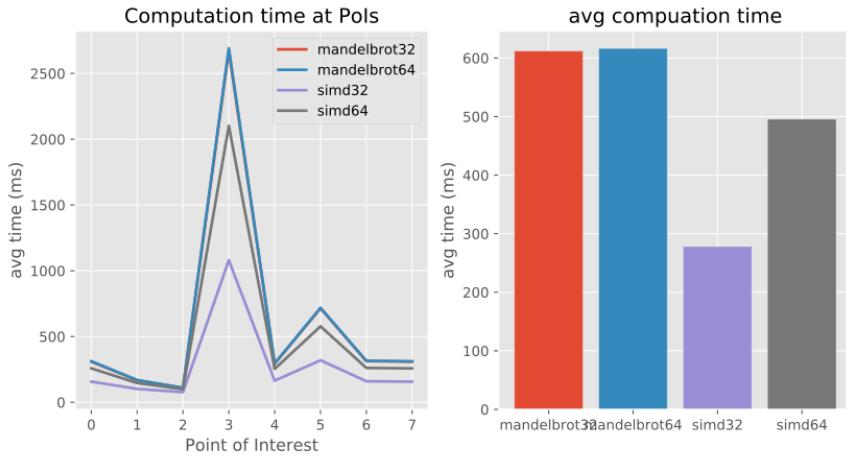


Abbildung 11: Vergleich der Performanzen der Implementierungen mit und ohne SIMD

## 6 Ergebnisse / Evaluation

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

### 6.1 Performanzerhöhung alternative Parallelisierungsmechanismen

#### 6.1.1 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 11 zu sehen, auf eine durchschnittliche Beschleunigung um den Faktor 1,9 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, entsteht eine gewisse Verzögerung durch zusätzliche Schiebeoperationen. Zudem werden für eine Menge an Punkten stets die Zahl an Iterationen durchgeführt, die das Maximum aller Iterationszahlen der Punkte ist. Dies wird dadurch bestätigt, dass die Beschleunigung am größten ist (ca. 2,5 und 1,3), wenn alle Punkte gleich große Iterationszahlen haben.

**Notiz, entdeckt durch das In-Betracht-Ziehen der Leerregionen** Probleme bei der Verwendung des Rekursiven Lastbalanierers: Da es stets im Diskreten eine Minimalgröße für die aufgeteilten Regionen gibt, kann es sein, dass eine Region für die eine hohe Last

vorhergesagt wird viele Worker reserviert - diese jedoch nicht vollständig ausreizen kann, da die Maximalaufteilung schon erreicht wurde. Durch die dadurch entstehenden Leerregionen von nicht verwendbaren Workern kann eine suboptimale Aufteilung entstehen, schlechter noch als die des naiven rekursiven Balancierers.

Dies kann abgeschwächt werden, indem eine Region stets nur soviel Workerressourcen erhält, wie sie maximal auslasten könnte (Fläche/Fläche minimaler Aufteilung)

## 7 Zusammenfassung

- Zusammenfassung
- Ausblick

## 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. . . . .	14
4	Konzept der Koordinaten in den Regionsobjekten. Alle Koordinaten beziehen sich auf die Darstellungsebene und sind daher in Pixeln. . . . .	15
5	Benutzeroberfläche der Mandelbrot Anwendung . . . . .	39
6	Relation von Backend Regionen zu leaflet Tiles. Dabei ist beispielhaft eine Region des Backends weiß eingezeichnet, alle leaflet Tiles sind rot umrandet angegeben. . . . .	41
7	WorkerLayer für eine Gruppierung der Aufteilung des Recursive PredictionBalancers mit 37 Workern . . . . .	42
8	Darstellung der Architektur der Anwendung als Graph . . . . .	43
9	Idle Time für 5 Gruppen . . . . .	44
10	Computation time für 4 Gruppen . . . . .	44
11	Vergleich der Performanzen der Implementierungen mit und ohne SIMD	45

## Tabellenverzeichnis

## Literatur

- [1] mrf (<https://math.stackexchange.com/users/19440/mrf>). Why is the bailout value of the mandelbrot set 2? Mathematics Stack Exchange. URL:<https://math.stackexchange.com/q/424331> (version: 2013-06-24).
  - [2] 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 27: Erstellen des Installationsordners

Die 'Header-only' Libraries `websocketpp` und `rapidjson` müssen lediglich an einen fixen Ort kopiert werden. Dies erledigen die Befehle aus Quelltext 28.

```
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 28: 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 29 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 29: 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 30 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 30: 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 31 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 31: Beispielhafter Start des Backends. Hierbei ist der Knoten des Hostprozesses `rpi03`.

**Stoppend 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 ps -eo comm,pid | grep srun
4 kill <srun pid>
```