Einführung in die Rechnerarchitektur Großpraktikum Parallele Berechnung der Mandelbrotmenge

Zweite Ausarbeitung

Maximilian Frühauf[1], Tobias Klausen[2],
Florian Lercher[3], Niels Mündler[4]

[1] max.fruehauf@tum.de
[2] tobias.klausen@tum.de
[3] florian.lercher@tum.de
[4] n.muendler@tum.de

1 Überblick über das Projekt

Es soll ein Programm zur Berechnung und Darstellung der Mandelbrotmenge entwickelt werden. Hierbei soll der Nutzer die Möglichkeit haben, beliebig in das Fraktal hinein und heraus zu zoomen, sowie den sichtbaren Ausschnitt zu verschieben. Dabei soll die Berechnung der Menge auf einem Cluster (bestehend aus mehreren Kleincomputern) durchgeführt werden (“Backend”), während die Darstellung in einem Webbrowser geschieht (“Frontend”). Die Berechnung im Backend soll dabei parallel erfolgen, und nach verschiedenen Kriterien auf die Worker balanciert sein, um die Vorzüge unterschiedlicher Strategien visuell darzustellen.

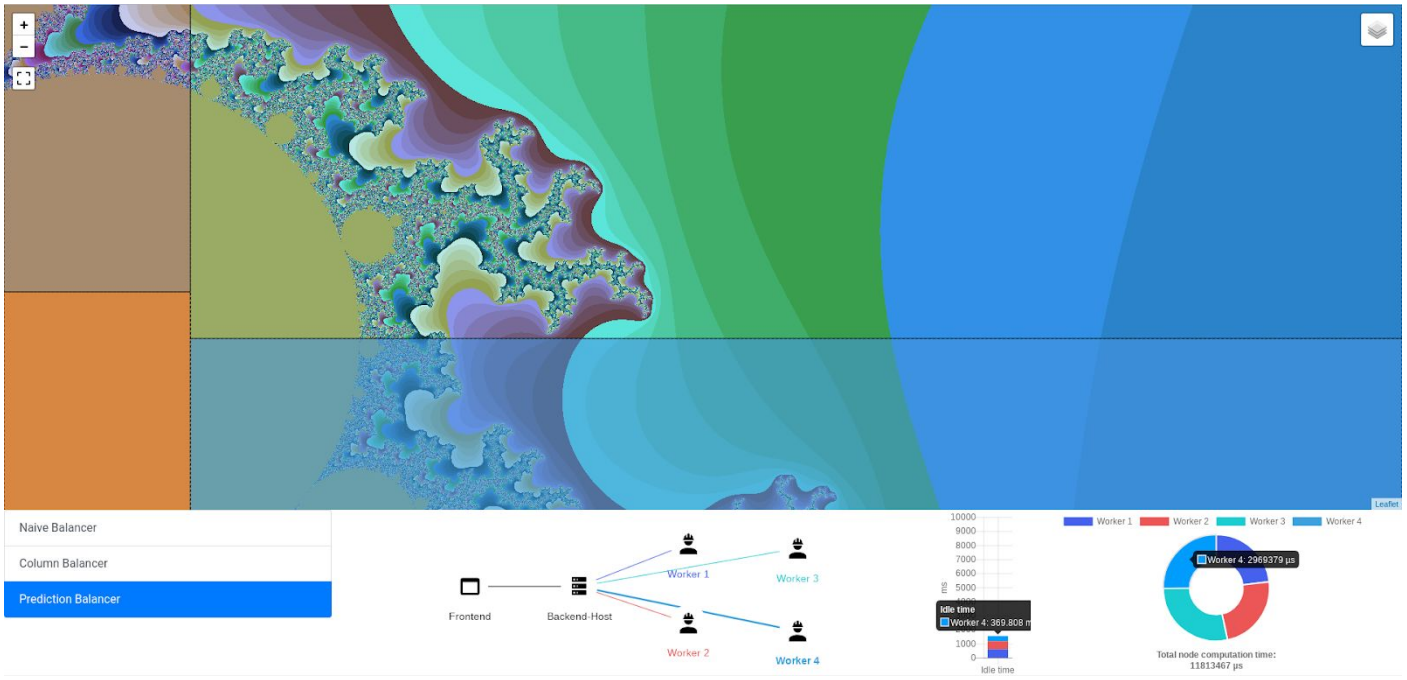

Abb. 1.1: Aktuelle Benutzeroberfläche

2 Definition: Mandelbrotmenge[5]

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

$$z_{n+1} = z_n^2 + c$$

Dabei ist $z_0 = 0$. In der Mandelbrotmenge befinden sich alle *c*, für die der Betrag von *z* nach beliebig vielen Iterationen 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. Die Elemente der Mandelbrotmenge besitzen einen Realteil zwischen -2 und 1 sowie einen Imaginärteil zwischen -1,5 und 1,5.

Komplexe Zahlen lassen sich auch grafisch darstellen, indem man sie in ein Koordinatensystem einträgt. Dabei entspricht die x-Koordinate dem Realteil und die y-Koordinate dem Imaginärteil der Zahl. Die grafische Darstellung der Mandelbrotmenge erhält man durch Einfärbung des Punktes zu *c* entsprechend dem zugehörigen Iterationswert, ab dem der Betrag von *z* 2 übersteigt.

Da man bei einem Programm zur Berechnung der Mandelbrotmenge nicht unendlich oft iterieren kann, wird vorher eine Maximalanzahl von Iterationen festgelegt. Überschreitet der Betrag von *z* auch nach der letzten Iteration nicht 2, so geht man davon aus, dass das zugehörige *c* in der Menge liegt. Bei steigender Iterationszahl kann so genauer differenziert werden, ab wann ein *c* divergiert, sodass sich ein genaueres Bild der Mandelbrotmenge ergibt. Falls der Betrag von *z* 2 überschreitet, wird die Berechnung abgebrochen. Damit kann man jedem *c* anhand der nötigen Iterationen eine eindeutige Farbe zuweisen und zudem entscheiden, ob es in der Menge liegt. Überschreitet der Betrag von *z* auch nach der letzten Iteration 2 nicht, so geht man davon aus, dass das zugehörige *c* in der Menge liegt.

Das entstehende Fraktal ist aufgrund seiner Form auch als “Apfelmännchen” bekannt. 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 Interesse am Projekt zu wecken.

Zudem ist die Berechnung der Farbe jedes Pixels sehr rechenaufwendig, da theoretisch unendlich oft, praktisch bis zu einer festgelegten Grenze (Maximalanzahl an Iterationen) das Ergebnis der voherigen Iteration quadriert werden muss. Es sind jedoch alle Pixel paarweise voneinander unabhängig. Deshalb handelt sich hierbei um ein rechenaufwendiges Problem, dessen Lösung einfach parallelisiert werden kann, da keine Abhängigkeiten zwischen Pixeln vorhanden sind.

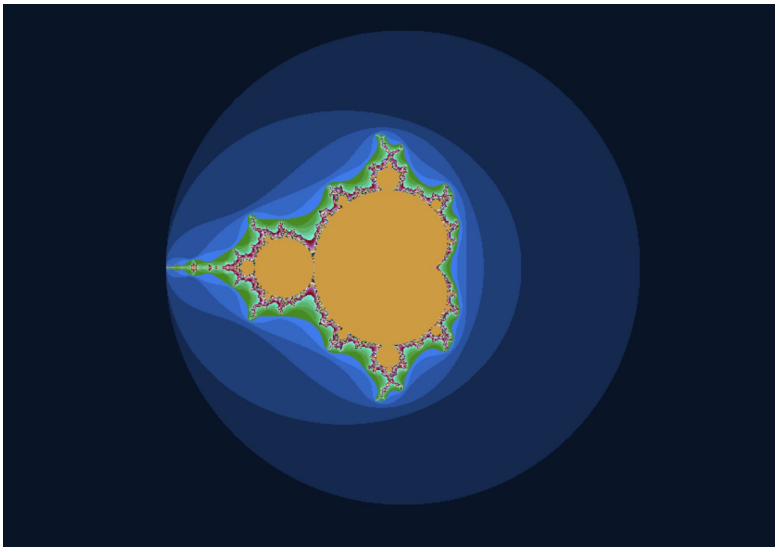

Abb. 2.1: Visualisierung der Mandelbrotmenge

[5] vgl. *Computer-Kurzweil 1* (1988). Heidelberg: Spektrum d. Wiss.

3 Begriffsdefinitionen

- Backend:
  Serverseitige Anwendung, die das Fraktal berechnet. Die Daten werden zur Weiterverarbeitung bereitgestellt und parallelisiert berechnet.

- Frontend:
  Webanwendung, die die vom Backend berechneten Daten visuell aufbereitet und für die Interaktion mit dem Nutzer zuständig ist.

- Region:
  Ein Ausschnitt des Fraktals. Im Frontend existiert stets eine sichtbare Region, der gesamte dargestellte Ausschnitt des Fraktals. Im Backend wird sie in kleinere Regionen unterteilt.

- Teilregion:
  Jede Region kann in Teilregionen unterteilt werden. Hierbei wird jede Teilregion von einem Rechenkern berechnet. Es gibt also pro Region genau so viele Teilregionen, wie es Rechenkerne gibt.

- Kachel:
  Aufteilung der Region, die im Frontend nur zur Darstellung verwendet wird.

4 Fachliche Spezifikation

4.1 Funktionale Anforderungen

Die funktionalen Anforderungen sind als Use-Cases beschrieben. Die folgenden vier Basis Use-Cases wurden im Rahmen des Projekts umgesetzt. Fünf weitere Use-Cases werden im folgenden Semester umgesetzt.

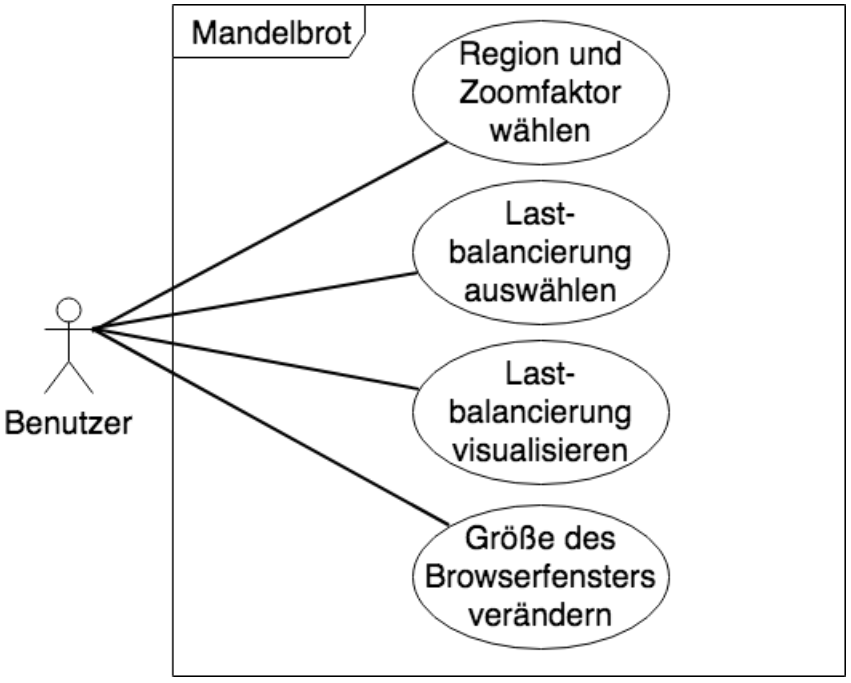

Abb. 4.1.2: Use Cases der Basisversion

4.1.1 Use Case: Region und Zoomfaktor wählen

Name	Region und Zoomfaktor wählen
Aktoren	Benutzer
Eintritts-bedingungen	Das Fraktal wird angezeigt.
Austritts-bedingungen	Die gewählte Region des Fraktals wird angezeigt.
Event-behandlung	Der Benutzer kann mit gedrückter linker Maustaste die Bildregion des Fraktals verschieben. Zum zoomen hat der Benutzer insgesamt 3 Möglichkeiten: ● Nutzung der Zoom-Knöpfe links oben im Bild ● Zoomen mit Hilfe des Mausrads ● Nach aktivieren des Zoombox-Modus mittels dem zugehörigen Knopf links oben im Bild, kann mit gedrückter linker Maustaste ein Rechteck über die Region gezogen werden, die berechnet werden soll. Wird die linke Maustaste losgelassen, startet die Berechnung. Ein Verschieben der Region ist bis zur Deaktivierung des Zoombox-Modus nicht mehr möglich.

4.1.2 Use Case: Lastbalancierung auswählen

Name	Lastbalancierung auswählen
Aktoren	Benutzer
Eintritts-bedingungen	Die Website ist geladen, d.h. die Startseite oder ein zuvor berechnetes Fraktal wird angezeigt.
Austritts-bedingungen	In der Liste der Balancer ist die gewünschte Option ausgewählt, welche bei den folgenden Berechnung angewendet wird.
Event-behandlung	Der Benutzer wählt in der Liste der Balancer die gewünschte Lastbalancierung aus. Wird nun durch eine andere Funktion das Fraktal neu berechnet, so wird die neue Methode zur Lastbalancierung angewendet.

4.1.3 Use Case: Lastbalancierung visualisieren

Name	Lastbalancierung visualisieren
Aktoren	Benutzer
Eintritts-bedingungen	Das Fraktal wird bereits angezeigt.
Austritts-bedingungen	Es wird ein Overlay über dem Fraktal angezeigt, welches dem Benutzer visualisiert, welcher Rechenknoten welche Teilregion berechnet hat.
Event-behandlung	Der Benutzer aktiviert das Lastbalancierungsoverlay in der Liste der Overlays. Die Teilregionen werden in verschiedenen Farben markiert. Der Benutzer kann das Overlay über die Liste wieder deaktivieren.

2.1.4 Use Case: Größe des Browserfensters verändern

Name	Größe des Browserfensters verändern
Aktoren	Benutzer
Eintritts-bedingungen	Das Fraktal wird bereits angezeigt oder aktuell berechnet.
Austritts-bedingungen	Das Fraktal wird in der neuen Größe angezeigt.
Event-behandlung	Der Benutzer verändert die Größe des Browserfensters. Daraufhin wird das Fraktal in der neuen Auflösung komplett neu berechnet. Laufende Berechnungen werden abgebrochen.

4.2 Funktionale Anforderungen - Erweiterte Version

Über die Basisversion hinaus ist geplant das Projekt im Wintersemester 2018 um weitere Features zu erweitern. Das Ziel der meisten dieser Erweiterungen ist es, die Interaktion mit dem Benutzer zu verbessern. Zudem ist geplant, die Performance des Back- und Frontends weiter zu verbessern.

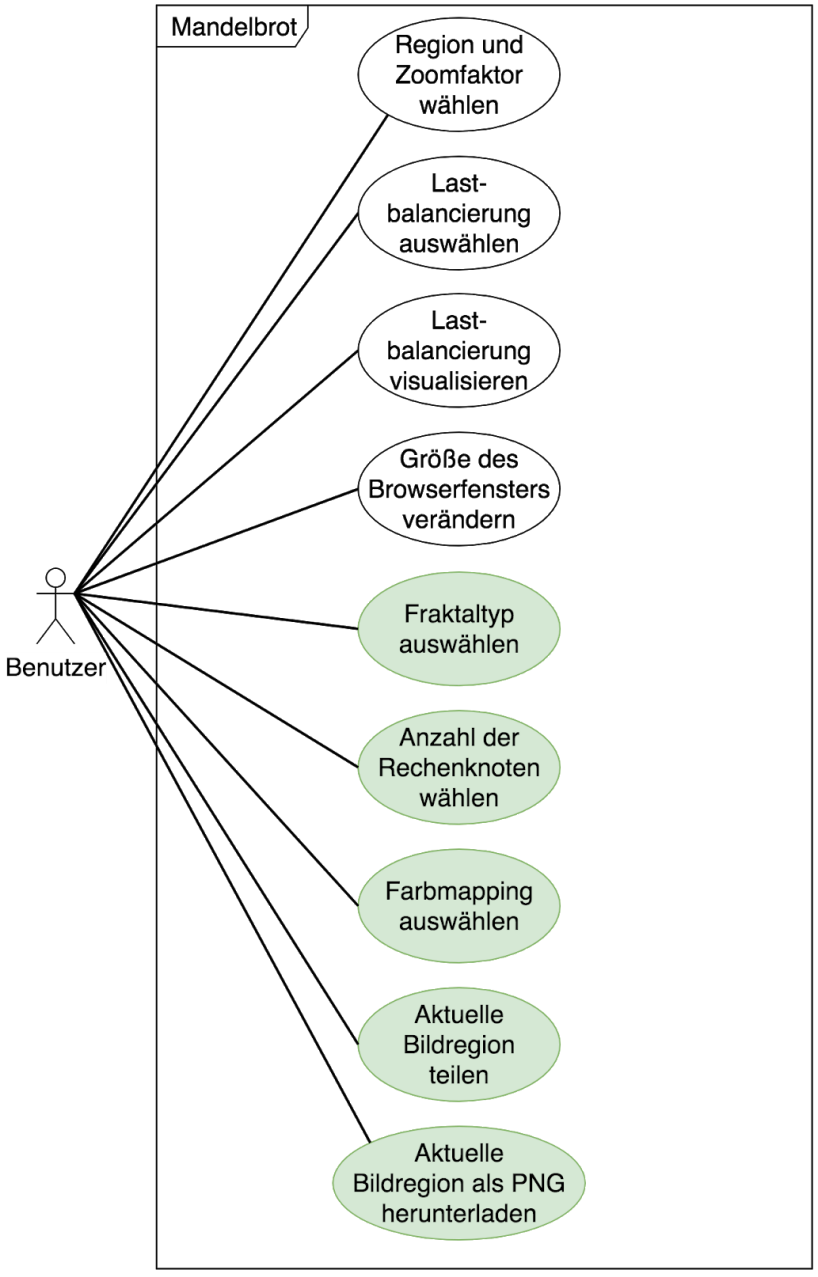

Abb. 4.2.1: Use Cases der erweiterten Version

4.2.1 Use Case: Fraktaltyp auswählen

Name	Fraktaltyp auswählen
Aktoren	Benutzer
Eintritts-bedingungen	Die Website ist geladen.
Austritts-bedingungen	Das gewünschte Fraktal wird in der vordefinierten Ausgangsposition angezeigt.

Event-behandlung	Der Benutzer wählt aus dem Drop-Down Menü das gewünschte Fraktal aus, welches anschließend berechnet wird. Zur Auswahl stehen mindestens die Mandelbrotmenge und die Juliamenge[6].

4.2.2 Use Case: Anzahl der Rechenknoten wählen

Name	Anzahl der Rechenknoten wählen
Aktoren	Benutzer
Eintritts-bedingungen	Die Website ist geladen. Die Anzahl der maximal zur Verfügung stehenden Rechenknoten ist vom Backend vorgegeben.
Austritts-bedingungen	Die Anzahl der Rechenknoten, welche verwendet werden, um die parallele Berechnung der Mandelbrotmenge durchzuführen wird verändert. Die Lastbalancierung ist auf die neue Rechenknotenanzahl angepasst.
Event-behandlung	Durch einen Slider kann der Benutzer interaktiv die Anzahl der Rechenknoten einstellen, die im Backend verwendet werden, um die Teilregionen zu berechnen. Wird die Anzahl aktiver Rechenknoten verringert, wird mehr Zeit benötigt, um alle Teilregionen zu berechnen.

4.2.3 Use Case: Farbmapping auswählen

Name	Farbmapping auswählen
Aktoren	Benutzer
Eintritts-bedingungen	Das Fraktal wird bereits angezeigt.
Austritts-bedingungen	Das Fraktal wird mit dem gewählten Farbmapping angezeigt.
Event-behandlung	Der Benutzer wählt entweder eines der vorgefertigten Mappings aus einem Drop-Down Menü oder kann eine eigenes Mapping erstellen, indem er Iterationen Farbwerte zuweist. Ein benutzerdefiniertes Mapping kann lokal gespeichert werden.

4.2.4 Use Case: Aktuelle Bildregion teilen

Name	Aktuelle Bildregion teilen
Aktoren	Benutzer
Eintritts-	Das Fraktal wird bereits angezeigt.

[6] https://de.wikipedia.org/wiki/Julia-Menge

bedingungen	
Austritts-bedingungen	Ein Dialog, der dem Benutzer die Möglichkeit gibt, eine für die aktuelle Bildregion spezifische URL zu kopieren, wird angezeigt.
Event-behandlung	Der Benutzer klickt auf den “Teilen”-Knopf. Daraufhin wird der Dialog angezeigt.

4.2.5 Use Case: Aktuelle Bildregion als PNG herunterladen

Name	Aktuelle Bildregion als PNG herunterladen
Aktoren	Benutzer
Eintritts-bedingungen	Das Fraktal wird bereits angezeigt.
Austritts-bedingungen	Die aktuelle Bildregion wird als PNG angezeigt. Über das Kontextmenü kann der Benutzer das Bild auf seinem PC speichern.
Event-behandlung	Der Benutzer klickt auf den “Herunterladen”-Knopf. Das Fraktal wird berechnet, als PNG gerendert und bereitgestellt.

4.3 Nicht-funktionale Anforderungen

Da die Basisversion vorrangig auf Funktionalität fokussiert ist, wird das System erst während der Erweiterungsphase hinsichtlich der nicht-funktionalen Anforderungen optimiert. In der Basisversion soll selbstverständlich trotzdem ein Mindestmaß dieser Anforderungen erfüllt sein.

4.3.1 Leistungsanforderungen

Eine zentrale Anforderung ist es, bei der Berechnung des Fraktales den Effekt von sinnvoller Parallelisierung darzustellen. Damit ein Unterschied festgestellt werden kann, muss also die Rechenzeit der Knoten für den Nutzer spürbar sein. Insgesamt sollte die Rechenzeit nicht über 3-5 Sekunden betragen, um die Geduld der Nutzer nicht zu sehr zu strapazieren.
Mehrere Benutzer müssen nicht unterstützt werden.

4.3.2 Qualitätsanforderungen

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

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

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

4.3.3 Einschränkungen

Das Frontend soll in einem Webbrowser lauffähig sein, während das Backend parallel auf mehreren Raspberry Pi’s oder ähnlichen unabhängigen Kleincomputern oder Rechenkernen zum Einsatz kommen soll.

5 Technische Spezifikation / Architektur

Das Problem fordert eine Unterteilung in drei wesentliche Bausteine:
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.

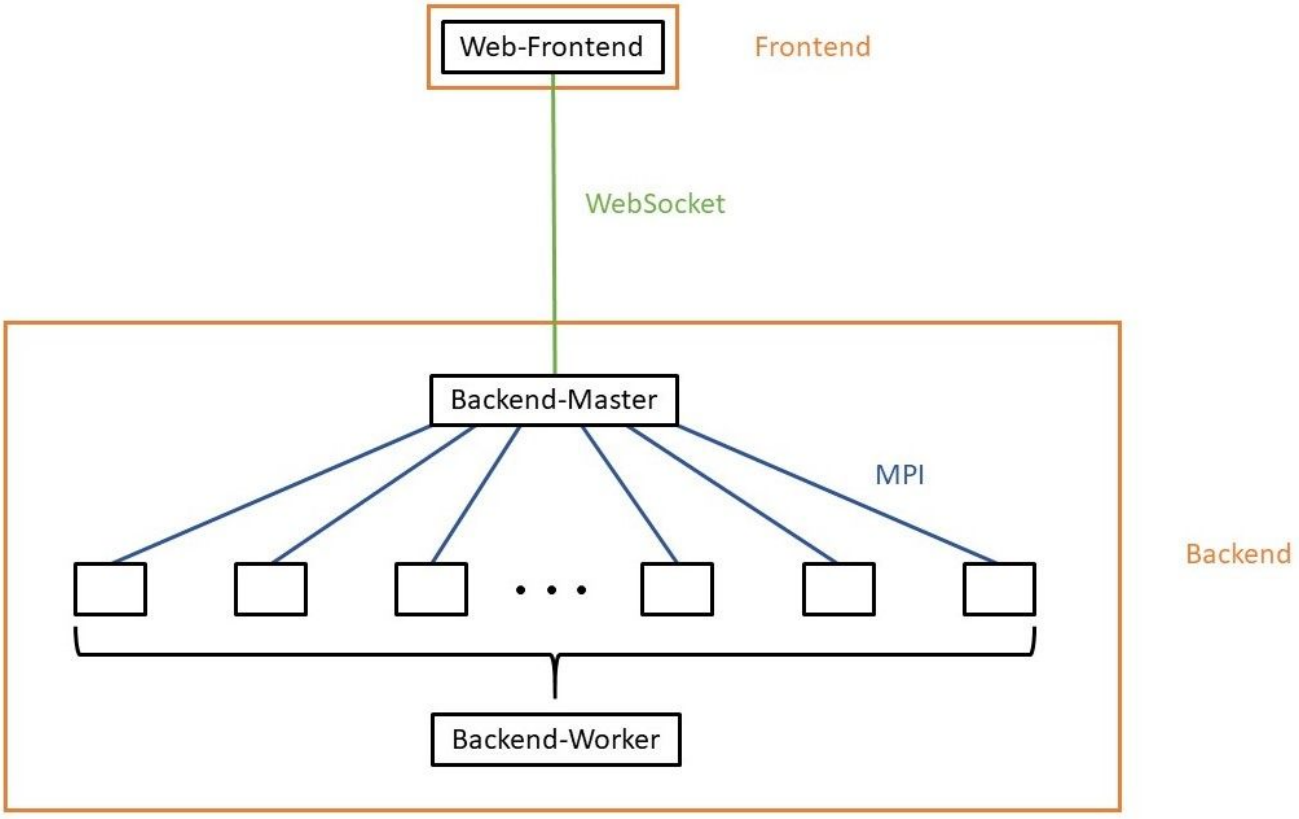

Abb. 5.1: Architekturübersicht

5.1 Frontend

5.1.1 Eingesetzte Technologien

Um das Frontend zu implementieren, wurde sich für die Programmiersprache JavaScript (ECMAScript 6 Standard[7]) entschieden. Diese erlaubt uns, Berechnungen innerhalb des Browsers auszuführen und erlaubt gestalterische Freiheit. Zudem wird ermöglicht dynamisch auf Eingaben des Nutzers zu reagieren.

Moderne Browser verwenden zum Anzeigen von Webseiten HTML (Hyper-Text-Markup-Language) Code, welcher die hierarchische Struktur einer Seite definiert und CSS (Cascading-Style-Sheets), um diese zu layouten. Um den benötigten HTML Code dynamisch in Reaktion auf eine Eingabe des Benutzers verändern zu können wird das React[8] Framework verwendet. Dieses erlaubt für jede logische Komponente eine eigene JavaScript Klasse zu erstellen, welche dann den betreffenden HTML Code generiert. Somit wird eine logische Trennung der einzelnen Domänen der Frontend Applikation erreicht, wodurch die Wartbarkeit des Systems erhöht wird.

Um das Fraktal darzustellen wird die Leaflet[9] Bibliothek verwendet. Diese, eigentlich für Onlinekarten konzipierte, Bibliothek stellt einen Bereich der komplexen Ebene, auf der die Mandelbrotmenge liegt dar. Diese sichtbare Region wird dann an das Backend versendet und vom Lastbalancierer in Teilregionen aufgeteilt. Die so entstandenen Teilregionen werden im Backend bearbeitet und die resultierenden Daten an das Frontend versendet. Dort wird jede Teilregion in kleine Kacheln von konstanter Größe aufgeteilt. Das nachfolgende Bild zeigt beispielhaft eine solche Aufteilung. Hier sind die Regionen der Lastbalancierung des Backends weiß gestrichelt und die von Leaflet erzeugten Kacheln rot umrandet dargestellt.

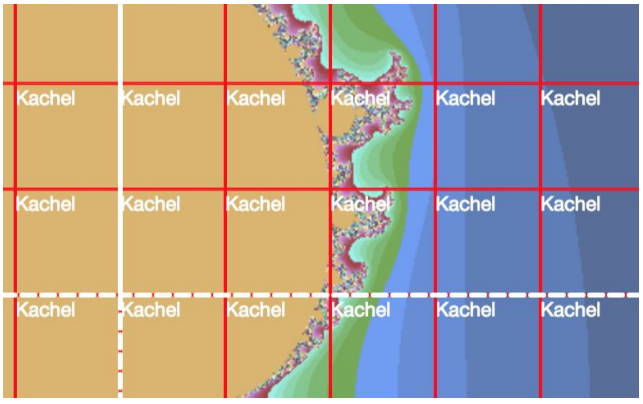

Abb. 5.1.1.1: Visualisierung von Kacheln und Teilregionen

Die Leaflet Bibliothek lässt sich einfach in den generierten HTML Code einbinden und weiter für spezifische Features erweitern.
Neben der Visualisierung der Mandelbrotmenge selbst wird ebenfalls die Bibliothek chart.js[10] für Balken- und Kreisdiagramme verwendet.

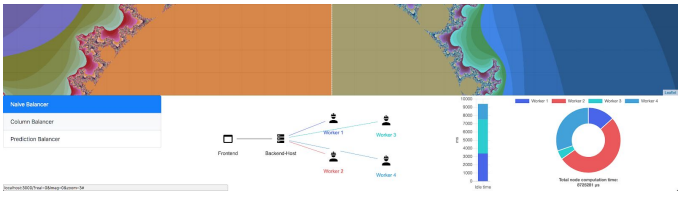

Abb. 5.1.1.2: Übersicht der Benutzeroberfläche

[7] https://www.ecma-international.org/publications/standards/Ecma-262.htm
[8] https://reactjs.org/
[9] http://leafletjs.com/
[10] https://www.chartjs.org/

Weiterhin wird vis.js[11] als Netzwerkgraphenbibliothek verwendet, um den Aufbau des Systems zu visualisieren.

Damit eine konsistente Version aller verwendeten Libraries über die Lebensdauer des Projekts gewährleistet werden kann, verwenden wir npm[12] als Paketmanager der verwendeten Libraries. Dieses ermöglicht es, genaue Versionen der verwendeten Packages zu spezifizieren und bietet eine entwicklerfreundliche Kommandozeilenanwendung um Packages zu installieren und zu aktualisieren.

5.1.2 Funktionsweise

Das Frontend der Applikation ist im Kern als monolithische Applikation realisiert, welche Daten aus dem Backend empfängt und diese dem Benutzer anzeigt. Daher ergibt sich eine logische Trennung der Komponenten, auch im Code. Das System (Window) besteht dabei aus den Komponenten Display, Visualization, Interaction und WebSocket Client.

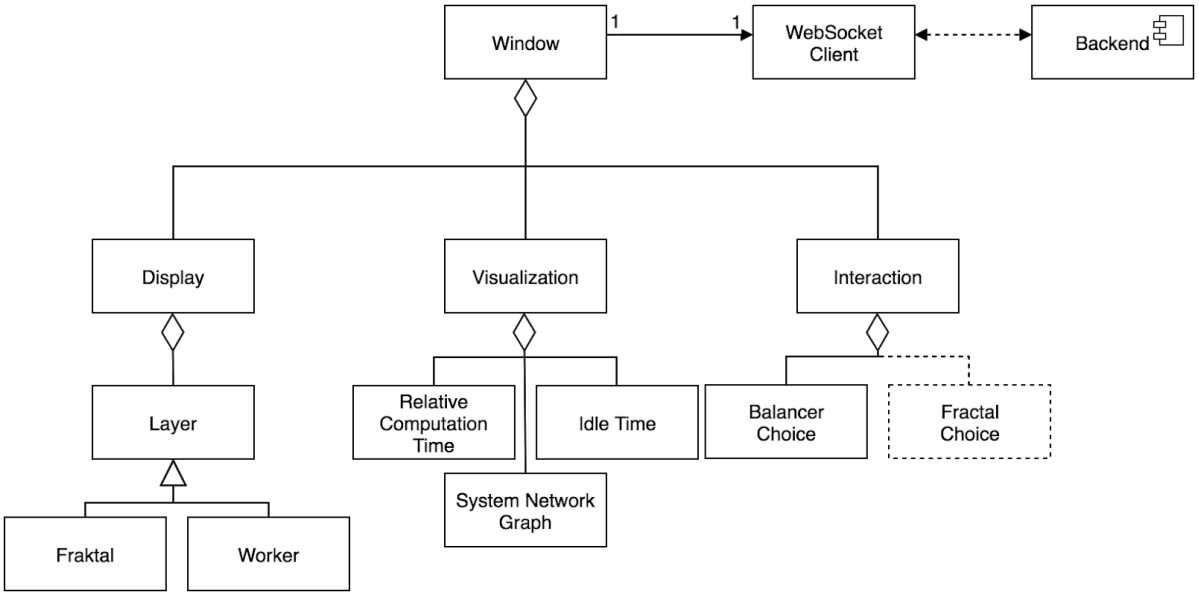

Abb. 5.1.2.1: Logische Aufteilung des Frontends

Durch eine hierarchische Dekomposition besteht das Fenster (Window) unter anderem aus der Display Komponente, welche die verschiedenen Layer der Leaflet Bibliothek enthält. Dieser wird in einen Mandelbrot Layer, welcher das Fraktal dem Benutzer anzeigt und einen Worker Layer der die Lastbalancierung des Backends darstellt differenziert.
Ebenfalls werden die Komponenten Visualization und Interaction erzeugt, welche Diagramme über die relative Verteilung der Rechenzeit, die Netzwerkdarstellung der Kommunikation des verteilten Systems und Komponenten zur Auswahl der Fraktale sowie Lastbalancierer durch den Benutzer enthalten.

Das Fenster erzeugt dabei initial eine Websocket Verbindung zum Backend, welche durch Anwendung des Observer Patterns den Komponenten des Windows ermöglicht auf ankommende Daten zu reagieren. Dabei registrieren alle beteiligten Objekte einen Callback bei der WebSocket Verbindung, welcher evaluiert wird, sobald Daten vorliegen. Dieses Pattern wird auch auf den Austausch der Informationen über den momentan ausgewählten Lastbalancierer und Fraktal angewendet, um eine responsive Anwendung zu ermöglichen.

[11] http://visjs.org/
[12] https://www.npmjs.com/

5.1.3 Design der Benutzeroberfläche

Das Design der Weboberfläche ist schlicht und modern gehalten und fokussiert sich im wesentlichen auf die Visualisierung der Mandelbrotmenge. Um jedoch weiterhin die vorgenommene Balancierung und den damit resultierenden Einfluss auf die Rechenzeit der einzelnen Bereiche darzustellen werden weitere Diagramme am unteren Rand des Fensters eingeblendet.
Dabei ist es immer die oberste Priorität, dem Benutzer möglichst einfach den Effekt unterschiedlicher Lastbalancierungs-strategien für die paralelle Berechnung der Mandelbrotmenge zu präsentieren.

Die Anordnung der Komponenten in unserem Design ist logisch nach deren Relevanz im laufenden System sortiert. Deshalb befinden sich von links nach rechts gelesen in dem unteren Banner Auswahl des Lastbalancieres, Netzwerkdarstellung, absolute Wartezeiten der Worker und eine relative Darstellung der Rechenzeit.

Die Auswahl des Balancers ist als Liste gestaltet, welche die Auswahl eines der verfügbaren Lastbalancierers erlaubt. Sobald der Benutzer länger die Maus über einen der Einträge hält, wird zusätzlich noch eine Beschreibung über die Funktionalität des Lastbalancierers angezeigt.

Der Systemaufbau wird mit einem Graph visualisiert, dessen Knoten die am System beteiligten, unabhängigen Komponenten (Frontend, Host, Worker) darstellen. Eine Kante bedeutet eine Kommunikationsverbindung zwischen den Komponenten.

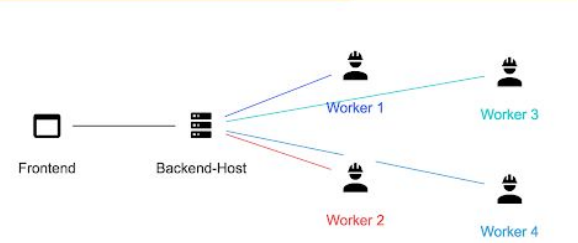

Abb. 5.1.3.1: Visualisierung des Systemaufbaus

Das Ziel einer optimalen Lastbalancierung ist es den gesamten sichtbaren Bereich der Mandelbrotmenge möglichst gleich zwischen den Workern aufzuteilen. Bei einer solchen optimalen Aufteilung werden somit auch die entstehenden Wartezeiten der Worker (Idle Times) minimiert.
Diese Wartezeiten werden in dem Balkendiagramm der Idle Time dargestellt. Eine Möglichst gute Aufteilung zeigt deshalb eine kleine Gesamt-Ruhezeit aller beteiligten Worker.

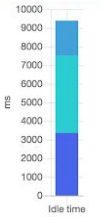

Abb. 5.1.3.2:
Wartezeiten der Worker

In der absoluten Wartezeit werden immer alle Worker außer demjenigen dargestellt, welcher zuletzt fertig wird, da dieser keine Wartezeit erzeugt.
Um auch eine relative Darstellung der Rechenzeiten der Worker zu geben, ist ebenfalls ein Kuchendiagramm eingebunden, welches die Rechenzeit aller Worker im Verhältnis zueinander darstellt. Eine möglichst gute Aufteilung teilt die Rechenzeit möglichst gleich auf, wodurch die farbigen Flächen im Kuchendiagramm der einzelnen Worker möglichst gleich groß werden.

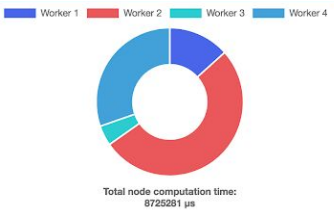

Abb. 5.1.3.3:
Relative Rechenzeiten der Worker

Um ebenfalls darzustellen, welcher Worker, einen gegebenen Teil des sichtbaren Ausschnitts der Mandelbrotmenge berechnet hat, wird hierfür ein Overlay eingeführt. Dieses wird halbtransparent über der Visualisierung der Mandelbrotmenge selbst angezeigt und färbt alle disjunkten Bereiche des

Lastbalancierers in einer eigenen Farbe ein. Damit zeigt dieses Overlay ebenfalls exakt die Aufteilung, welche der ausgewählte Lastbalancierer vorgenommen, und vom Backend parallel berechnet wurden. Förderlich ist dies für das Design Ziel des Frontends, da damit dem Benutzer dargestellt werden kann, welche Bereiche der Mandelbrotmenge rechenintensiv darzustellen sind und deshalb mehr Zeit benötigen, und welche sich schnell berechnen lassen.

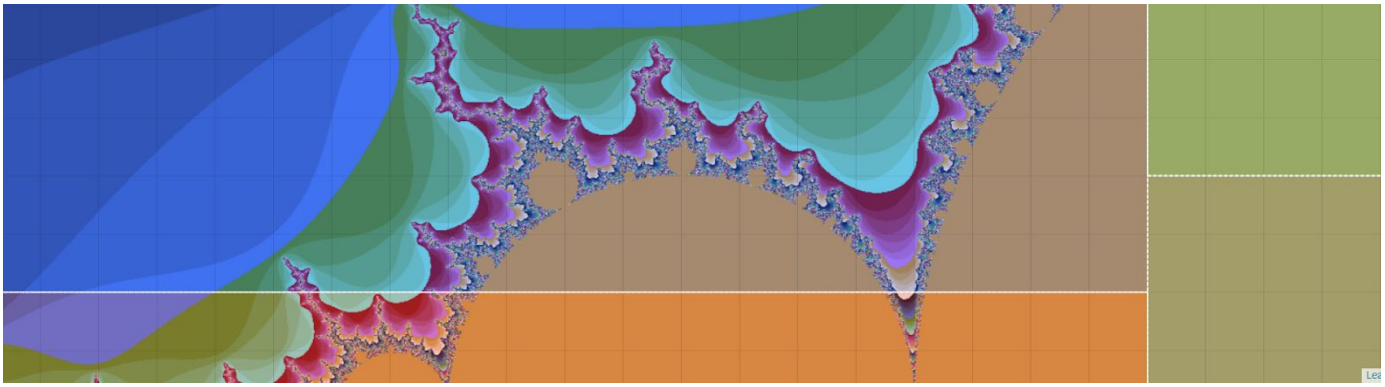

Abb. 5.1.3.4: Overlay einer Lastbalanceriung

Um der Nutzerin eine Zuordnung zwischen Region, Name, Wartezeit und Rechenzeit zu ermöglichen, erhalten in jeder der Komponenten die Knoten jeweils die gleiche Farbe. Die Bereitstellung der entsprechenden Information wird über ein Objekt der Klasse *WorkerContext* bereitgestellt, welches ebenfalls synchronisiert, welcher der Knoten aktuell im Fokus der Nutzerin steht. Diese Synchronisation wird ebenfalls durch das Observer-Pattern durchgeführt.

5.2 Backend

5.2.1 Eingesetzte Technologien

Als Programmiersprache im Backend wird C++ verwendet. Zum einen bietet C++ hochsprachliche Konstrukte, wie die Möglichkeit Objekte in Klassen zu organisieren. Auch Vererbung und Polymorphie, zwei wichtige Konzepte der objektorientierten Programmierung, können genutzt werden, um die Wartung und Erweiterung des Systems zu erleichtern. Zum anderen ist C++ eine vergleichsweise maschinennahe und somit performante Sprache.

Für die Websocket-Verbindung zum Frontend wird die websocketpp[13] Bibliothek als leichtgewichtige und performante Implementierung eingesetzt.

Die Kommunikation der verschiedenen Prozesse des Backends wird mittels MPI realisiert. Das Message Passing Interface[14] ist eine weit verbreitete Spezifikation, die Kommunikation zwischen unabhängigen Rechenkernen regelt. Dadurch existieren viele gut funktionierende Umsetzungen in einer Vielzahl von Programmiersprachen.

Als konkrete Implementierung wurde sich für MPICH entschieden. Ein Grund dafür ist, dass MPICH stetig weiterentwickelt und verbessert wird, es also regelmäßig Veröffentlichungen mit neuen Features und Bugfixes gibt. Zudem ist diese Implementierung sehr gut dokumentiert, weit verbreitet und es existiert eine sehr aktive Community mit Blogs und Tutorials. Dies macht es für alle Beteiligten leicht, sich in die Materie einzulesen und kann in Zukunft für eine schnelle und zuverlässige Lösung von Problemen sorgen. Außerdem wird MPICH zusammen mit C++ auf einer Vielzahl von Systemen unterstützt, wozu auch der Raspberry Pi gehört.
Da für dieses Projekt nur die grundlegendsten Funktionen von MPI verwendet werden, sollte das Backend aber auch mit allen anderen gängigen MPI-Implementierungen (z.B. OpenMPI) kompilierbar sein.

5.2.2 Funktionsweise

Um die Mandelbrotmenge parallel zu berechnen werden mehrere Worker-Prozesse gestartet, welche jeweils einen Teil der zu berechnenden Region erhalten. Um diese Aufteilung zu koordinieren existiert ein zentraler Host-Prozess, welcher Aufgaben an die Worker verteilt und Rechenergebnisse entgegen nimmt. Ebenfalls ist dieser Prozess die zentrale Kommunikationseinheit mit dem Frontend. Dort findet also die Behandlung der Anfragen und das Versenden der berechneten Regionen statt.

Die Aufgabe des Hosts ist es, zunächst Anfragen des Frontends für die komplette Region via Websocket entgegenzunehmen und in Teilregionen aufzuteilen (siehe Lastbalancierung). Die Teilregionen werden nun vom Host per MPI an die Worker verteilt, wobei jedem Worker genau eine Teilregion zugeteilt wird (siehe Kommunikation mittels MPI).

Hat ein Worker seine Teilregion erhalten, so wird jeder Pixel auf einen entsprechenden Punkt in der komplexen Ebene projiziert. Der Worker iteriert über alle Pixel und erstellt ein Ergebnisarray. Dabei wird die tatsächliche Berechnung des Fraktals für einen Pixel an eine Instanz einer Unterklasse von *Fractal* delegiert. Da die Bindung an eine konkrete Unterklasse dynamisch passiert, kann der Fraktaltyp zur Laufzeit gewechselt werden.

[13] https://github.com/zaphoyd/websocketpp
[14] https://www.mpi-forum.org/

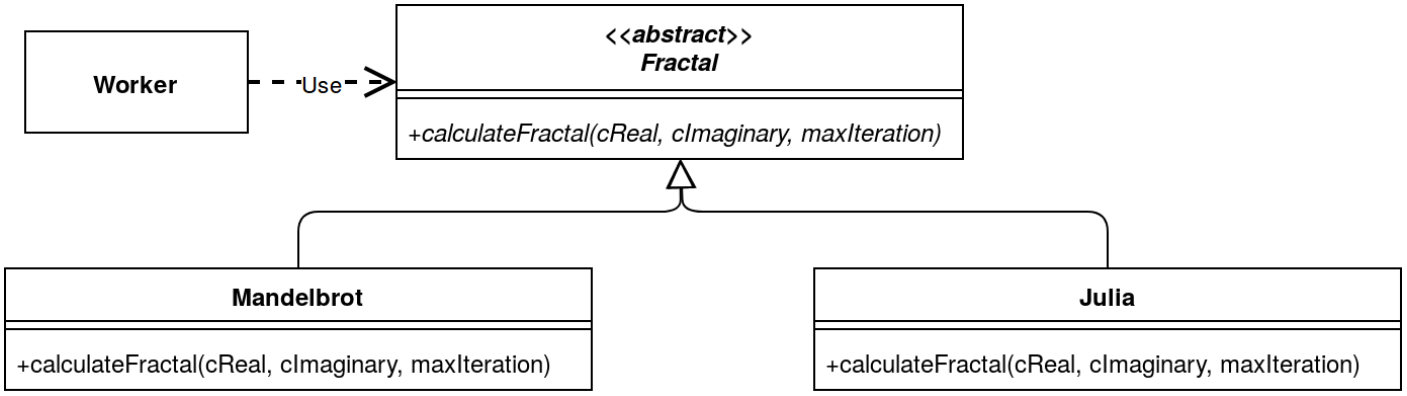

Abb. 5.2.2.1: Klassendiagramm für *Fractal*

Nachdem alle Worker ihre Teilregion erhalten haben, wird die getätigte Aufteilung vom Host per Websocket an das Frontend weitergeleitet. Dies ist für eine Visualisierung der Aufteilung zwingend nötig.

Sobald die Berechnungen eines Workers abgeschlossen sind, werden die Ergebnisse wieder mittels MPI an den Host übermittelt.

Damit der Zeitunterschied (vor allem bzgl. der Option ohne Lastbalancierung) bei der Berechnung der verschiedenen Teilregionen für den Nutzer zu sehen ist, müssen die Ergebnisse der Teilregionen mit möglichst geringer Verzögerung vom Backend-Host an das Frontend weitergeleitet werden. Dies geschieht wieder per Websocket.

5.2.3 Lastbalancierung

Ziel der Lastbalancierung ist es, die Rechenlast so gut wie möglich auf die zur Verfügung stehenden Rechenkerne aufzuteilen. Hierzu wird die Region in Teilregionen aufgeteilt.

Es stehen momentan eine naive Strategie und eine Strategie mit Vorhersage zur Verfügung.

Die naive Strategie (Naive Balancer) teilt die Region in möglichst gleich große Teilregionen auf. Da dies ohne Rücksicht auf eventuell unterschiedliche Rechenzeiten in den einzelnen Teilregionen geschieht, kann es dabei zu einer schlechten Lastverteilung kommen.

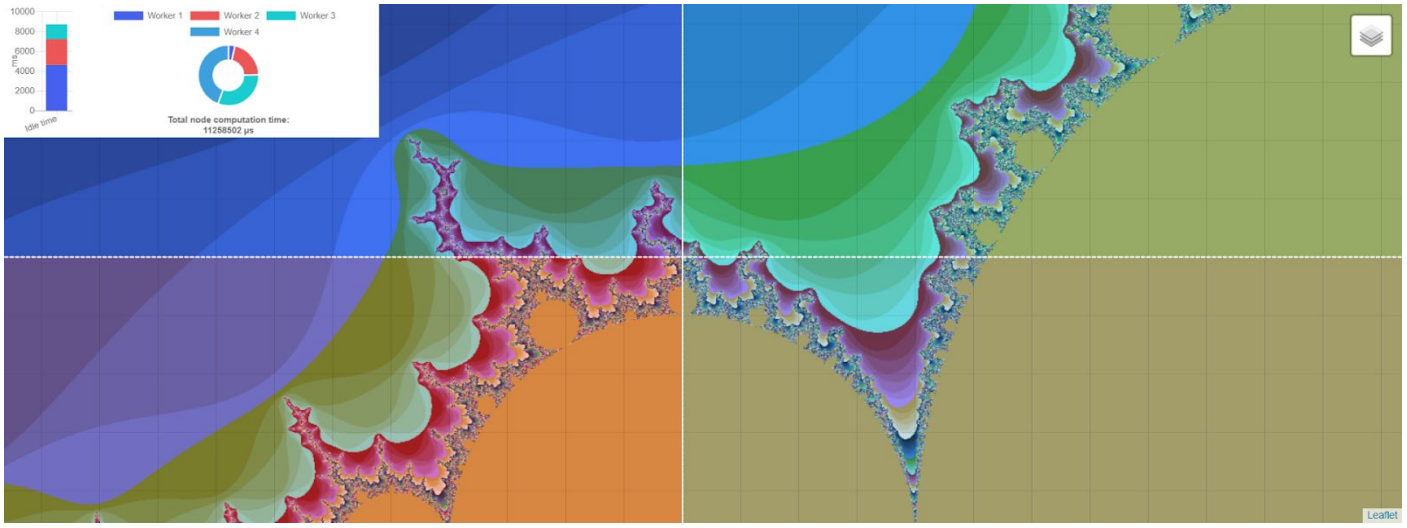

Abb. 5.2.3.1: Naive Strategie zur Lastbalancierung

Bei der Strategie mit Vorhersage (Prediction Balancer) wird die zu berechnende Region vor der Aufteilung

in geringerer Auflösung im Host vorberechnet. Die Teilregionen werden dann so gewählt, dass der durch die Vorhersage abgeschätzte Rechenaufwand möglichst gleich unter den Workern verteilt ist. Aber auch diese Strategie ist nicht perfekt. Die Vorhersage ist durch die geringere Auflösung vor allem am Rand des Fraktals ungenau. Die Genauigkeit der Vorhersage zu erhöhen bedeutet zusätzlichen Rechenaufwand während der Balancierung. Dieser sollte in einem sinnvollen Verhältnis zum Aufwand der Berechnung des Fraktals stehen. Einen perfekten Ausgleich zu finden ist wesentliche Aufgabe ernsthafter Optimierungsansätze.

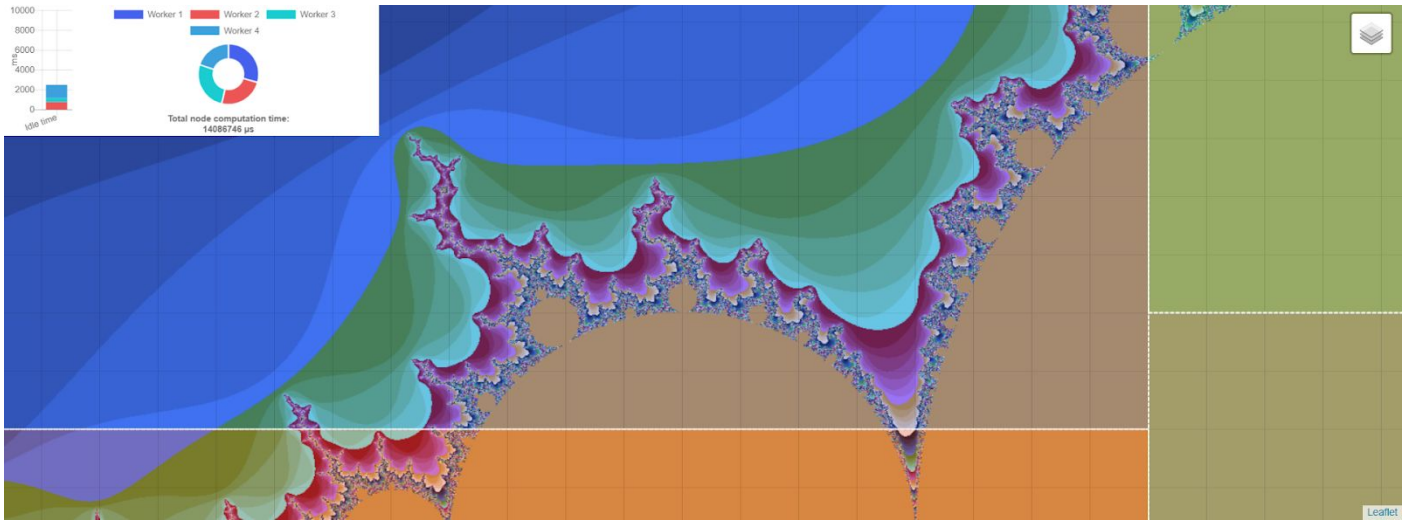

Abb. 5.2.3.2: Strategie mit Vorhersage zur Lastbalancierung

5.2.4 Kommunikation der Prozesse per MPI

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

Um die Berechnung einer Teilregion anzufragen, sendet der Host ein entsprechendes *Region*-Objekt per MPI mit Tag 1 an den Worker mit dem zuvor bestimmten Rang. Der Worker erwartet:

- Koordinaten des Bildausschnitts (bzgl. der komplexen Ebene) und Auflösung (= Breite und Höhe der Region)
- Einen Teiler der Auflösung, der bei der Aufteilung erhalten werden soll (im Fall des Leaflet Frontends ist das die Auflösung einer Kachel, ansonsten kann man dies auf 1 setzen)
- Die maximale Anzahl an Iterationen
- Kennzahl des Fraktals
- Validierungsnummer der jeweiligen Region (zur Zuordnung der Aufteilung im Frontend)

Region
minReal, maxImag: Double
maxReal, minImag: Double
width, height: Integer
hOffset, vOffset: Integer
maxIteration: Integer
validation: Integer
guaranteedDivisor: Integer

WorkerInfo
rank: Integer
computationTime: Integer
region: Region

Abb 5.2.4.1: Objekte, deren Austausch über MPI spezifiziert ist

Der Worker beginnt sofort nach Empfang der Nachricht mit der Bearbeitung der Region. Ist die Region fertig berechnet, werden als Antwort die gestoppte Zeit und die berechneten Daten gesendet.

Um laufende Berechnungen abbrechen zu können und um die Leistung zu steigern, wurde sich dazu entschieden, eine asynchrone Kommunikation zwischen dem Host und den Workern aufzubauen. Diese wird genutzt, um neue Rechenaufträge für Teilregionen an die Worker zu verteilen.

Der Host sendet dabei jede Teilregion an einen anderen Worker, sodass jeder Worker genau eine Teilregion zugeteilt bekommt. Dieses senden passiert parallel für alle Worker, indem das nicht-blockierende *MPI_Isend* genutzt wird. Es startet die Senden-Operation, und kehrt unter Umständen zurück, bevor die Senden-Operation abgeschlossen ist. Dadurch wird ein möglichst schnelles Senden der Aufträge erzielt. Um sicherzustellen, dass die Senden-Operation abgeschlossen ist, also alle Senden-Buffer vollständig ausgelesen wurden, wird *MPI_Waitall* eingesetzt, das den Thread des Hosts so lange blockiert, bis alle Anfragen gesendet wurden. Es wird nicht gewartet, bis die Anfragen von allen Workern empfangen wurden.

Um sicherzustellen, dass der Worker laufende Berechnungen bei Erhalt einer neuen Teilregionsanfrage abbricht und sofort mit der Berechnung der neuen Anfrage beginnt, wird hier ebenfalls ein nicht-blockierendes Empfangen eingesetzt. Das bedeutet, dass zu Beginn der Berechnung *MPI_Irecv* aufgerufen wird, was ausdrückt, dass eine Nachricht in bestimmter Form erwartet wird und in einen spezifizierten Buffer geschrieben werden soll. In diesem Fall wird ein Region-Objekt als Anfrage erwartet. Vor der Berechnung jedes einzelnen Pixels der alten Anfrage wird per *MPI_Test* geprüft, ob eine neue Nachricht empfangen wurde. Ist dies der Fall, wird die Berechnung sofort abgebrochen und mit der Bearbeitung der neuen, empfangenen Region begonnen.

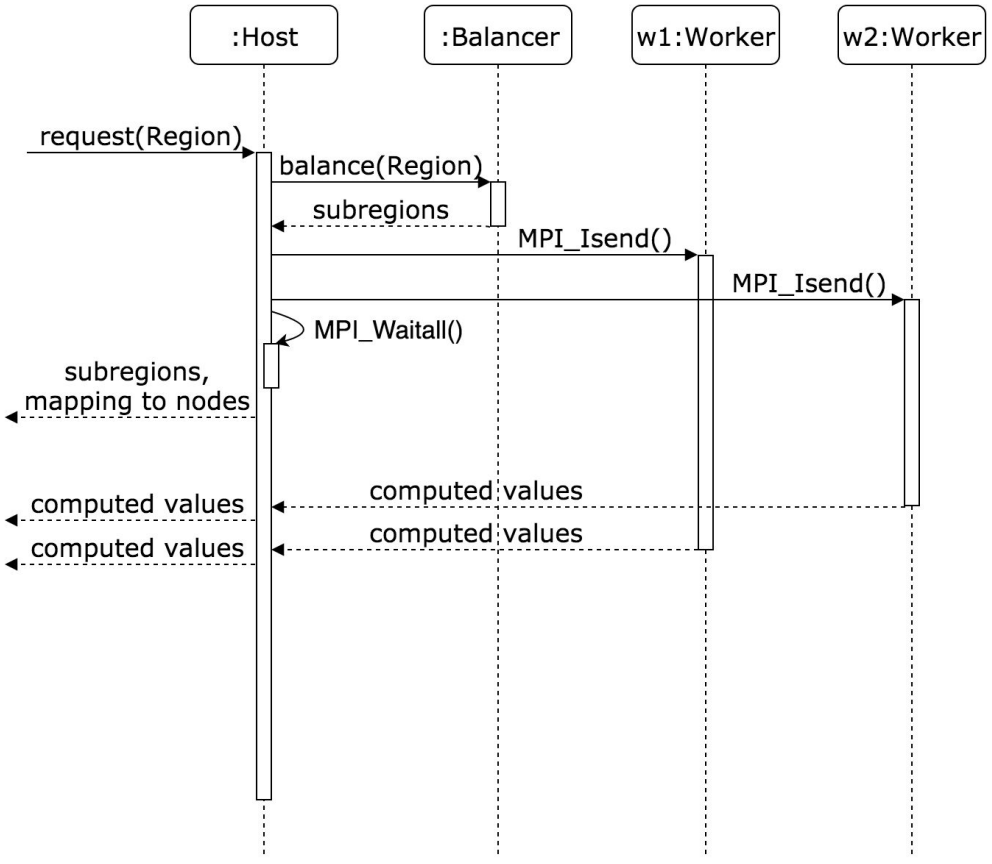

Abb 5.2.4.2: Beispielhafte Anfrage einer Region.

Das Senden der berechneten Region vom Worker zum Host funktioniert hingegen mit blockierenden Sende- bzw. Empfang-Operationen (*MPI_Send* und *MPI_Recv*). Diese Variante wurde gewählt, da das Senden im Vergleich zum Berechnen sehr wenig Zeit benötigt, so dass die Komplexität, die eine nicht-blockierende Behandlung mitbringen würde in keinem Verhältnis zum Nutzen steht.

Tatsächlich besteht das Senden der berechneten Daten aus zwei Schritten.
Zunächst wird ein *WorkerInfo*-Objekt mit der gestoppten Zeit der Berechnung in µs, dem Rang des Workers und dem zu Beginn empfangenen *Region*-Objekt vom Worker gesendet und vom Host empfangen. Hierzu wird der Tag 3 benutzt.
Anschließend erwartet der Host von dem selben Worker ein Array an Integern der Größe $region.width * region.height$, wobei *region* in dem gerade erhaltenen *WorkerInfo*-Objekt enthalten war. Dieses Array enthält die eigentlichen Daten, die der Worker berechnet hat. Für diese Operation wird der Tag 2 benutzt.

Dieses eben beschriebene Senden bzw. Empfangen muss in zwei Schritte aufgeteilt werden, da es nicht ohne Probleme möglich ist, ein struct mit einem dynamischen Array über MPI zu senden. Ein dynamisches Array für die Ergebnisse der Berechnungen wird benötigt, da nicht zu Compile-Zeit festgelegt werden kann, wie viele Pixel eine Teilregion enthält. Das ist abhängig vom Balancer. Dieses dynamische Array wird in einem struct aber als Pointer dargestellt. Das heißt, dass die Daten nicht direkt bei den anderen Daten des structs liegen. Dies hat zur Folge, dass nur ein Pointer mit MPI geschickt werden würde. Das ist aber nicht zielführend, da die Daten ja immer noch auf einem anderen Hardwareknoten liegen und dadurch ein Zugriff nicht möglich wird. Deshalb erschienen zwei separate Operationen als die beste Lösung.

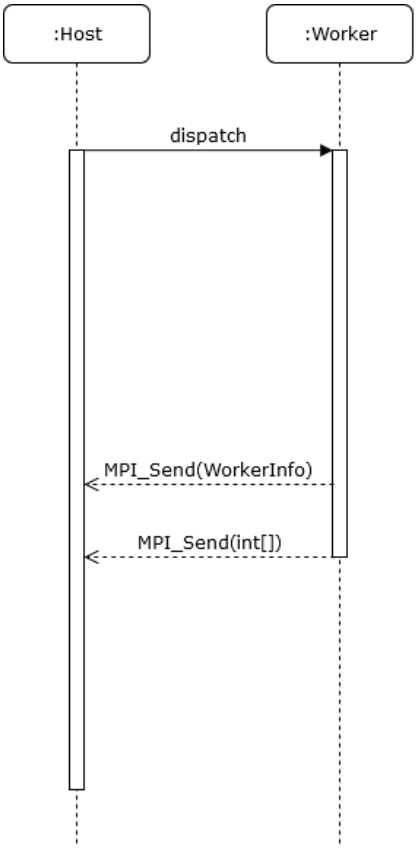

Abb 5.2.4.3:Tatsächlicher Ablauf der Rückgabe der berechneten Daten

Das Array 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:

$$i = y * region.width + x$$

Noch eine Bemerkung zum Kommunikationsverhalten von MPI:
Da MPI grundsätzlich auf Höchstleistung ausgelegt ist, wartet ein blockierendes MPI_Recv aktiv, indem es ständig testet ob eine Anfrage gesendet wird (busy waiting). Das hat zur Folge, dass ankommende Nachrichten sofort bearbeitet werden können, führt aber auch zu einer 100-Prozentigen Auslastung des Rechenkerns. Mit dem nicht-blockierenden *MPI_Irecv* kann dieses Verhalten angepasst werden. In der Implementierung wurde sich dazu entschieden, den Thread bei dem Ausbleiben eines neuen Auftrages und wenn der alte Auftrag schon abgeschlossen ist für eine Millisekunde zu pausieren bevor das nächste mal das Empfangen eines neuen Auftrages überprüft wird. Die damit einhergehende Verzögerung verfälscht das Ergebnis nicht schwerwiegend, da die Berechnung einer Region für gewöhnlich deutlich länger als 100 Millisekunden benötigt. Es führt allerdings zu einer drastischen Entlastung des Rechenkerns und damit zu einer Senkung des Stromverbrauchs.

5.3 Kommunikation

Im Folgenden noch einmal das Zusammenspiel der einzelnen Komponenten im Überblick als Sequenzdiagramm:

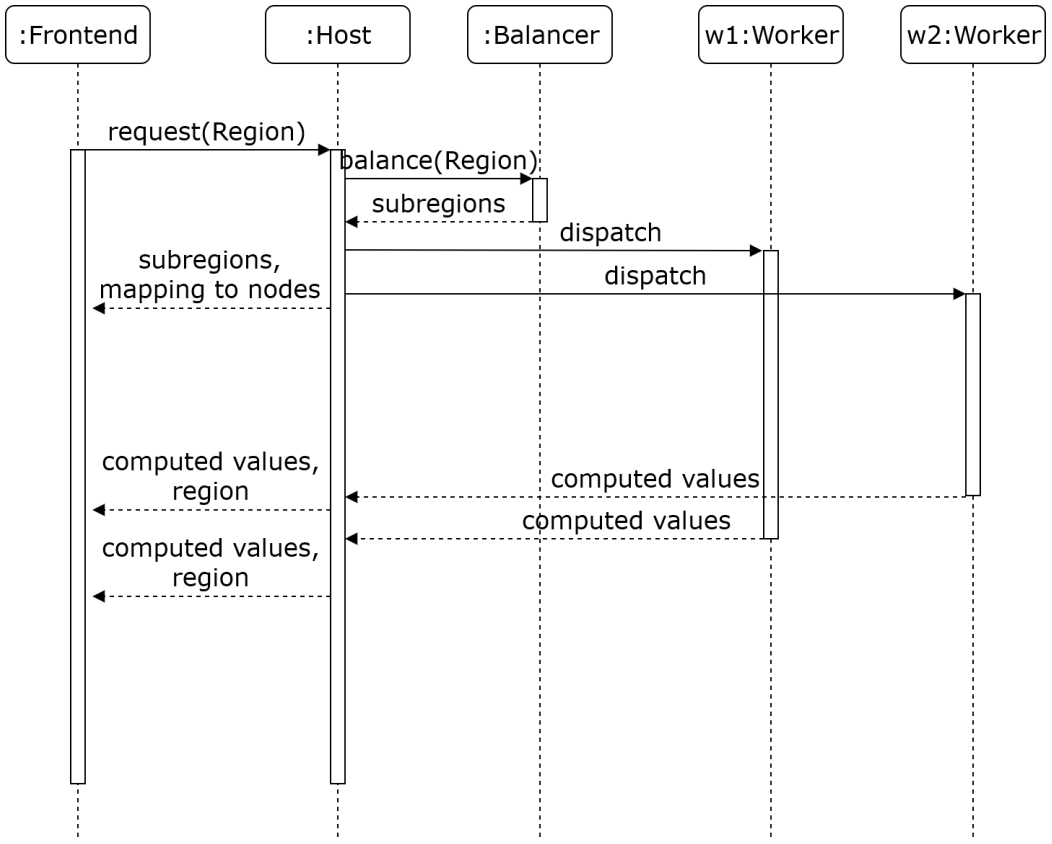

Abb 5.3.1: Beispielhafte Anfrage einer Region. Hierbei ist zufällig Rechenknoten 1 langsamer, was jedoch das System nicht blockiert.

Die Trennung der verschiedenen Komponenten erfordert eine Kommunikationsebene zwischen dem Back- und Frontend der Anwendung. Es wird ein Kommunikationsprotokoll spezifiziert, sodass serialisierte Objekte ausgetauscht werden können. Hierzu werden JSON-Kodierte Daten über Websockets übertragen.

5.3.1 Eingesetzte Technologien

JSON implementiert eine Serialisierungsvorschrift für beliebige Objekte[15]. Unter anderem können Arrays, Objekte mit Memberattributen (gleichend einer Map) sowie diverse primitive Datentypen encodiert werden. In diesem Fall werden drei Objekte spezifiziert, die JSON-Serialisiert ausgetauscht werden. Es handelt sich dabei um ein Objekt zur Anfrage der Unterteilung einer bestimmten Region in Teilregionen, sowie die Antwort darauf, die die Aufteilung mitsamt einer Zuordnung auf die aktivierten Rechenknoten enthält. Zuletzt können vom Backend aktiv Datenblöcke gesendet werden, die zusätzlich Informationen darüber enthalten, welcher Teilregion sie angehören und Metadaten zu Rechenknoten und -dauer bieten.

Die Daten werden hierbei über Websockets versendet. Websockets ist ein relativ neues OSI Layer 5-7 Protokoll, welches anders als HTTP Vollduplexkommunikation zwischen Browseranwendung und Server über TCP erlaubt. Dies ist wichtig, da nicht nur Anfragen vom Webfrontend beantwortet werden müssen, sondern möglichst ohne Verzögerung ein fertig vom Host berechneter Bereich an das Frontend gesendet

[15] Die exakte Spezifikation ist hier zu finden https://www.json.org/json-de.html

werden soll[16]. Zur Implementierung bietet das Backend einen Websocketserver über die Library websocketpp[17] an, mit dem sich das Frontend über JavaScript-native Websocketclients[18] verbindet.

5.3.2 Spezifizierte Objekte

Der Host erwartet vom Frontend zur Festlegung des nächsten Bildausschnitts folgende Daten:

- Koordinaten des Bildausschnitts (bzgl. der komplexen Ebene) und Auflösung (Breite und Höhe in Pixel der Region)
- Einen Teiler der Auflösung, der bei der Aufteilung erhalten werden soll (im Fall des Leaflet Frontends ist das die Auflösung einer Kachel, ansonsten kann man dies auf 1 setzen)
- Die maximale Anzahl an Iterationen
- Kennung des zu berechnenden Fraktals
- Kennung der anzuwendenden Lastbalancierung
- Anzahl der Rechenprozesse
- Validierungsnummer der jeweiligen Region (zur Zuordnung einer Region im Frontend)

Nach der Lastbalancierung wird als Antwort an das Frontend zurückgegeben:

- Ein Array mit Regionen und dem zugewiesenen Worker, jeweils
    - Informationen über die Region
    - Rang des Workers, der die Region berechnen wird
- Die Anzahl der Regionen

Sobald die Berechnungen eines Workers abgeschlossen sind werden folgende Daten an das Frontend gesendet:

- Array mit den Ergebnissen der Berechnung als Rohdaten
- Informationen über die berechnete Region
- Benötigte Rechenzeit
- Kennung des Rechenknotens

Abb. 4.3.2.1 kann hierzu entnommen werden, wie die exakten Typen und Namen der Objekte und Attribute benannt werden. Über den Websocket-Kanal werden dabei stets nur serialisierte *SharedObject* Subklassen versendet, da JSON die Klassennamen nicht berücksichtigt. Diese werden jedoch für eine korrekte Zuordnung und Behandlung der Objekte benötigt, weshalb das Feld *type:String* die genauere Spezifikation übernimmt.

Das in einem *RegionData*-Objekt gesendete Array mit Ergebnissen ist eine eindimensionale Repräsentation der in der Region berechneten Fraktalwerte. Der Index *i* im Array lässt sich nach der oben schon für die MPI-Kommunikation spezifizierten Regel aus *x* und *y* Koordinate berechnen, wobei *region* die berechnete Region ist:

$$i = y * region.width + x$$

[16] Zwar könnte dies auch über wiederholte Anfragen vom Frontend oder jeweils eine gepufferte HTTP-Anfrage pro Region beim Backend gelöst werden. Ersteres würde jedoch zu einer Verzögerung beim Empfangen der Region oder unnötige verschwendete Rechenleistung durch busy waiting führen. Letzteres kann nicht ohne weiteres umgesetzt werden, da moderne Browser nur maximal 8-10 HTTP-Anfragen gleichzeitig zulassen, wodurch bei mehr als 10 Regionen nicht alle Regionsanfragen parallel bearbeitet werden und sich eine erhebliche Verzögerung ergibt.

[17] https://www.zaphoyd.com/websocketpp/manual/reference/roles/websocketppserver

[18] https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

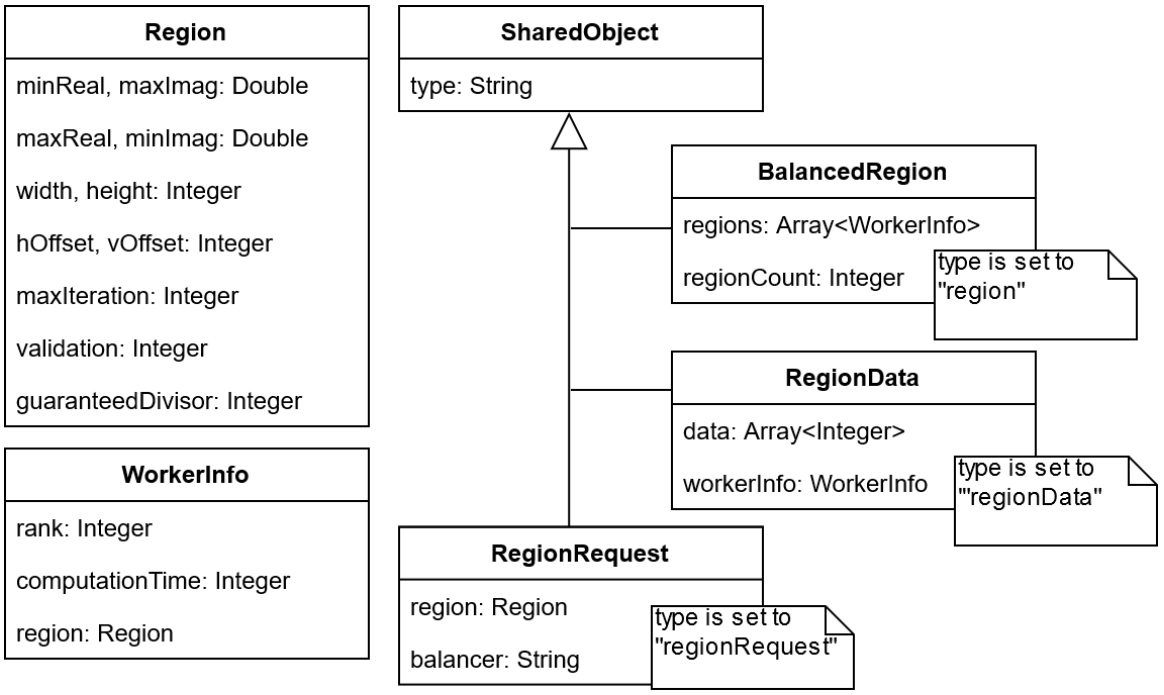

Abb 5.3.2.1: Benennung und Spezifikation der Objekte, die zwischen Systemen gesendet werden

5.3.3 Ablauf der Kommunikation

Um die Berechnung eines Bereiches anzufragen sendet das Frontend die gewünschte Region in einem *RegionRequest* an das Backend. Dieses liefert als Antwort zunächst die Aufteilung der Region sowie die zugewiesenen Worker als Menge in einem *BalancedRegion*-Objekt zurück, sobald der Sendeprozess an alle Worker gestartet wurde. Anschließend werden, sobald eine Teilregion ausgerechnet wurde, die zugehörigen Werte als *RegionData* an das Frontend gesendet.

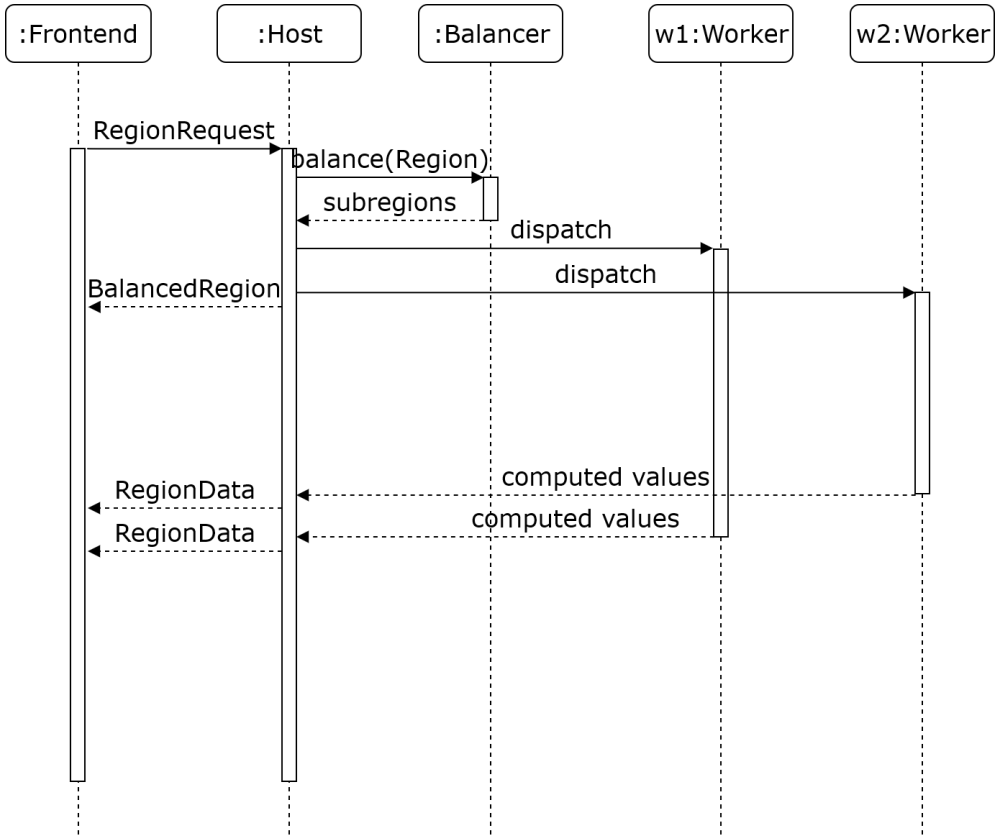

Abb 5.3.3.1: Ablauf der Anfrage mit Angabe der spezifizierten Objekte. Zu beachten ist, dass die Frakalwerte ohne Verzögerung weitergeleitet werden und die balancierte Region erst gesendet wird, nachdem die zugehörigen Worker sie empfangen haben.

5.4 Zielplattformen

Während der Entwicklung wird der C++ Code des Backends in einer Dockerumgebung[19] mit Linux kompiliert und ausgeführt. Dies ermöglicht ein problemloses Arbeiten über mehrere Systeme mit unterschiedlichen Betriebssystemen und Entwicklungsumgebungen.
In der finalen Version soll das Backend auf einem Raspberry Pi oder ähnlichen unabhängigen Kleincomputern lauffähig sein. Zudem kann es durch Docker auch auf anderen Zielplattformen problemlos laufen. Allerdings muss hierbei bedacht werden, dass dies die zu erwartende Leistung senkt.

[19] https://www.docker.com/

6 Teamkommunikation

Die Kommunikation zwischen den einzelnen Teammitgliedern haben wir, abhängig vom Kontext des Gesprächs auf zwei Onlineplattformen aufgespalten. Einerseits verwenden wir WhatsApp für eine schnelle und informelle Kommunikation, welche sich um grundlegende Konzepte oder Organisatorisches dreht. Andererseits verwenden wir GitLab für detaillierte Gespräche zu dem verwendeten Code oder der Diskussion unterschiedlicher Lösungsmöglichkeiten für ein aufgetretenes Problem.

Dabei wird für jedes auftretende Problem oder eine vorgeschlagene Erweiterung von dem entsprechenden Teammitglied in GitLab ein Issue erstellt, welcher das Problem beschreibt und optional ein paar mögliche Lösungen aufführt.
Dieser wird dann von einem Teammitglied auf einem eigenen Branch von *dev* mit dem gleichen Namen wie der Issue bearbeitet. Sobald eine Lösung gefunden wurde oder weiterer Diskussionsbedarf besteht, wird dies entweder weiter in dem zugehörigen Issue besprochen oder ein Merge Request mit *dev* erstellt.
Dann ist angedacht, dass ein anderes Mitglied sich den fertigen Code und den zugehörigen Issue durchliest. Dieses gibt dann Feedback zu der implementierten Lösung und womit eine hohe Codequalität innerhalb des Repositories gesichert wird. Sobald alle Verbesserungsvorschläge eingearbeitet sind, wird das Merge Request geschlossen und in *dev* eingefasst.
Sobald genug fertige Features für eine Version auf *dev* sind und das Programm einwandfrei funktioniert werden die Änderungen in *master* übernommen. So wird sichergestellt, dass auf *master* nur komplett lauffähige und auslieferbare Versionen gespeichert sind.

Außerhalb der online Kommunikation, treffen wir uns ca. einmal pro Woche persönlich. Diese Treffen haben meist weder eine festgelegte Länge, noch vordefinierte Themen. Der Zweck ist es, aufgekommene Fragen zu diskutieren insbesondere wenn persönliche Besprechung sinnvoll ist um Details zur Implementierung oder Modellideen auszutauschen.

Zur gemeinsamen Arbeit an Textdokumenten wie dieser Spezifikation, wird Google Docs verwendet. Google Docs bietet ein gemeinsames Dokument, in dem alle Änderungen von allen Teilnehmern durch den integrierten Versionsverlauf gut nachvollziehbar sind. Kontroverse Stellen und Änderungsvorschläge können durch die Kommentarfunktion direkt im Dokument diskutiert werden.

7 Ablauf des Projekts

Um eine gewisse Spezialisierung auf Teilgebiete in diesem recht breite Ansprüche stellenden Projekt wurde eine Aufgabenteilung vorgenommen.

Tobias Klausen und Florian Lercher beschäftigen sich mit der Umsetzung des Backends und setzen sich dazu verstärkt mit C++ und MPI auseinander. Maximilian Frühauf kümmert sich unterdessen um die Entwicklung der Weboberfläche. Zudem stellt er das Grundgerüst für die plattformunabhängige Entwicklung des Backends auf. Niels Mündler kümmert sich um das Zusammenspiel von Front- und Backend und ist für die funktionierende Kommunikation der Systeme verantwortlich.

Wir gestalten das Projekt überwiegend agil mit folgendem Projektplan:

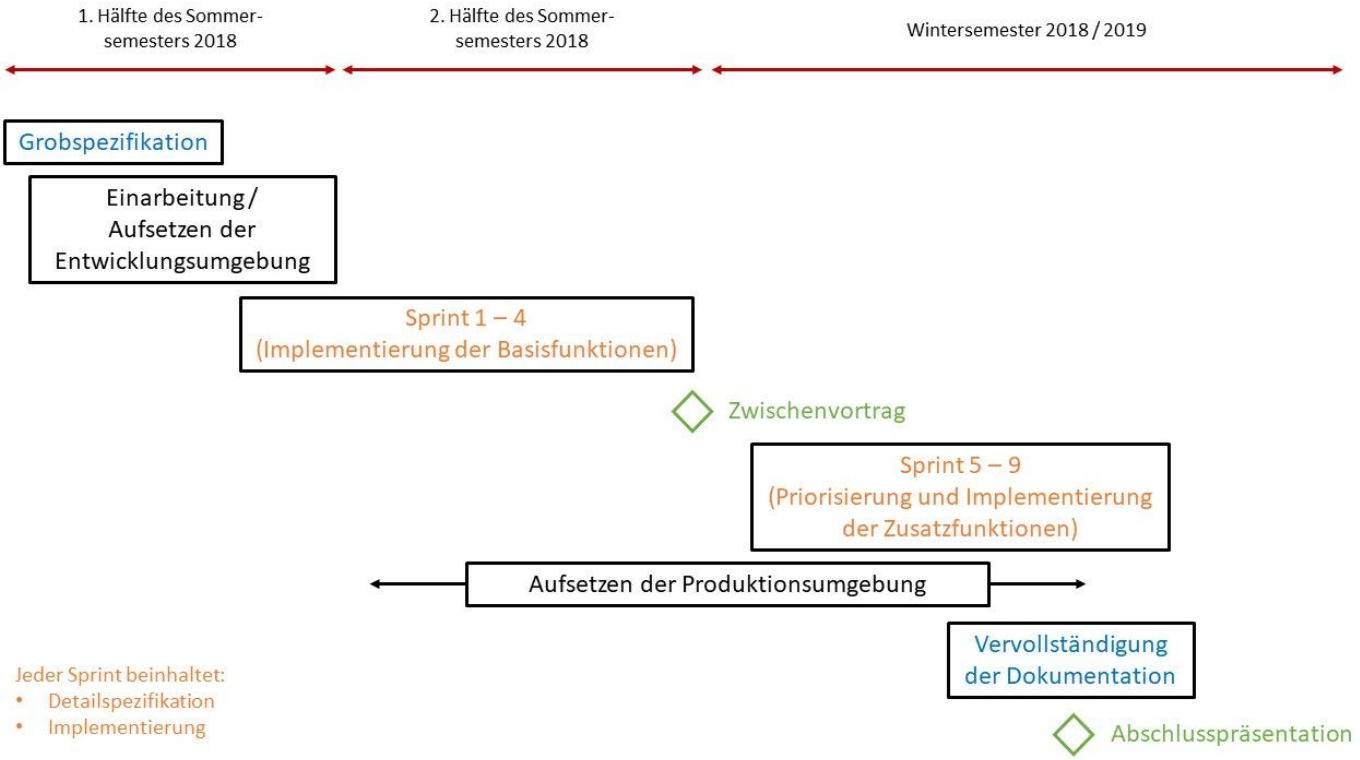

Abb. 7.1: Projektplan