LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Parallele Berechnung der Mandelbrotmenge

Wintersemester 2018/19

Maximilian Frühauf Tobias Klasuen Florian Lercher Niels Mündler

1 Einleitung

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

1.1 Didaktische Ziele

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

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

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

1.2 Verwendung der Mandelbrotmenge

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

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

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

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

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

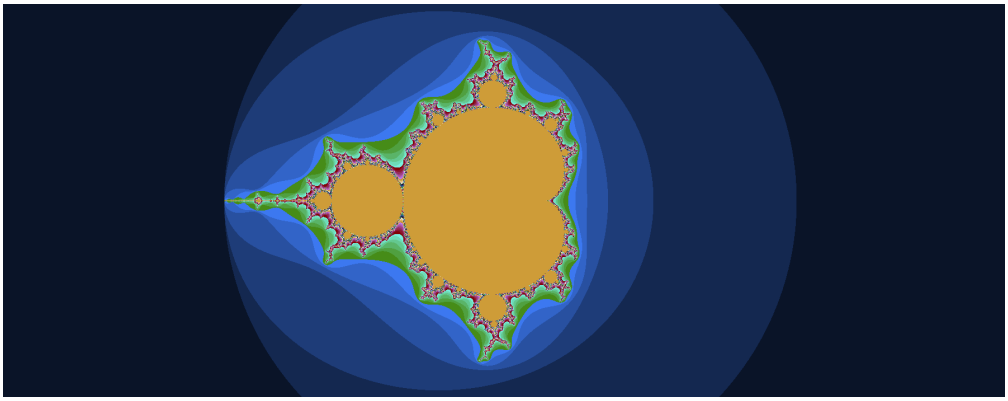

Abbildung 1: Die Mandelbrotmenge, visualisiert in einem Ausschnitt des komplexen Zahlenraumes.

Diese Eigenschaften ermöglichen es, zweierlei Dinge zu kontrollieren:

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

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

1.3 Darstellung der Mandelbrotmenge

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

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

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

1.4 MPI

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

1.5 Qualitätsanforderungen

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

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

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

1.6 Einschränkungen

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

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

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

2 Problemstellung und Spezifikation

- Fachliche Spezifikation in Anhang

Abbildung 2: Visualisierung des Systemaufbaus

3 Dokumentation der Implementierung

3.1 Überblick

Das Problem fordert eine Unterteilung in drei wesentliche Bausteine, wie sie in Abbildung 6 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.

3.1.1 Frontend

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 Fluss und Erzeugung der Informationen sortiert. Deshalb befinden sich von links nach rechts gelesen in dem unteren Banner Auswahl des Lastbalancieres, Netzwerkdarstellung mit Frontend links über Host zu den Workern, 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.

Das Ziel einer optimalen Lastbalancierung ist es, den gesamten sichtbaren Bereich der Mandelbrotmenge möglichst gleich bezüglich des Rechenaufwandes 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 Gesamtwartezeit aller beteiligten Worker.

In der absoluten Wartezeit werden immer alle Worker außer demjenigen dargestellt,

Abbildung 3: Wartezeiten der Worker

Abbildung 4: Relative Rechenzeiten der Worker

welcher zuletzt fertig wird, da dieser keine Wartezeit erzeugt. Um auch einen relativen Vergleich der Rechenzeiten der Worker darzustellen, 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.

Um ebenfalls darzustellen, welcher Worker, einen gegebenen Teil des sichtbaren Ausschnitts der Mandelbrotmenge berechnet hat, wird 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 bestimmt hat und die vom Backend parallel berechnet wurde. 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.

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.

3.1.2 Backend

Um die Mandelbrotmenge parallel zu berechnen, werden mehrere Worker-Prozesse per MPI auf unterschiedlichen Rechenknoten gestartet. Jeder davon soll jeweils einen Teil der zu berechnenden Region erhalten. Um diese Aufteilung zu koordinieren existiert ein zentraler Host-Prozess. Er teilt sich mit einem der Worker-Prozesse einen Rechenknoten, verteilt Aufgaben an die Worker und nimmt Rechenergebnisse entgegen. Dieser Host-Prozess ist außerdem die zentrale Kommunikationseinheit für das Frontend. Dort findet also die Behandlung der Anfragen und das Versenden der berechneten Regionen an die Benutzeroberfläche statt.

Die Aufgabe des Hosts ist, zunächst Anfragen des Frontends für die komplette Region via Websocket entgegenzunehmen und in Teilregionen aufzuteilen (siehe Lastbalancierung). Die Teilregionen werden anschließend vom Host per MPI an die Worker verteilt,

Abbildung 5: Overlay einer Lastbalanceriung

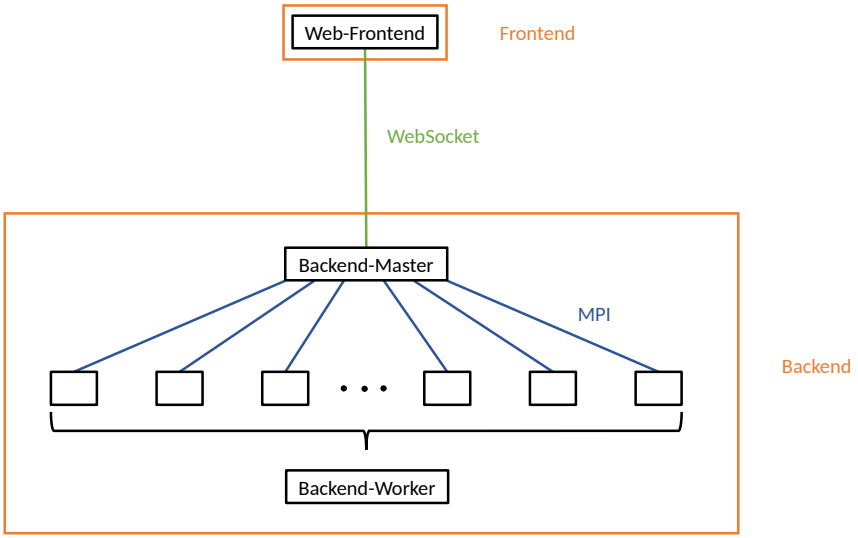

Abbildung 6: Architekturübersicht

wobei jedem Worker genau eine Teilregion zugeteilt wird (siehe Kommunikation mittels MPI).

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

Nach dem Empfangen einer einzelnen Region beginnen die Worker mit der Berechnung des Bereiches. Dazu wird jeder Pixel auf einen entsprechenden Punkt in der komplexen Ebene projiziert. Die tatsächliche Berechnung des Fraktals für einen Pixel an eine Instanz einer Unterklasse von Fractal delegiert. Da die Bindung an eine konkrete Unterklasse dynamisch geschieht, kann der Fraktaltyp zur Laufzeit gewechselt werden. Der Worker iteriert so über alle Pixel und erstellt ein Ergebnisarray.

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 wird durch die Verwendung von Websockets ermöglicht.

3.1.3 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.

Abbildung 7: Naive Strategie zur Lastbalancierung

Abbildung 8: Strategie mit Vorhersage zur Lastbalancierung

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.

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.

3.2 Installation

3.2.1 Frontend

Um das Frontend zu implementieren, wurde sich für die Programmiersprache JavaScript (ECMAScript 6 Standard[3]) 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 Framework[4] 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 Bibliothek[5] 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

[3] `https://www.ecma-international.org/publications/standards/Ecma-262.htm`
[4] `https://reactjs.org/`
[5] `http://leafletjs.com/`

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.

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[6] für Balken- und Kreisdiagramme verwendet.

Weiterhin wird vis.js[7] 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[8] als Paketmanager der verwendeten Libraries. Dieses ermöglicht es, genaue Versionen der verwendeten Pakete zu spezifizieren und bietet eine entwicklerfreundliche Kommandozeilenanwendung um Pakete zu installieren und zu aktualisieren.

3.3 Backend

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 Bibliothek[9] als leichtgewichtige und performante Implementierung eingesetzt.

Die Kommunikation der verschiedenen Prozesse des Backends wird mittels MPI realisiert. Das Message Passing Interface[10] ist eine weit verbreitete Spezifikation, die Kommunikation zwischen unabhängigen Rechenkernen regelt. Dadurch existieren viele gut funktionierende Umsetzungen in einer Vielzahl von Programmiersprachen. Durch MPI wird echte Parallelisierung mit geringem Overhead ermöglicht. Es können so die einzelnen Workerprozesse auf jeweils einem eigenen unabhängigen Rechenkernen laufen. Die Gestaltung von MPI erlaubt dabei alle möglichen Aufteilungen, von Kernen auf einem Prozessor bis hin zu unabhängigen Clustern, die lediglich eine SSH-Verbindung besitzen.

Als konkrete Implementierung wurde sich für MPICH entschieden. Ein Grund dafür ist, dass MPICH stetig weiterentwickelt und verbessert wird, es also regelmäßig Veröf-

[6] `https://www.chartjs.org/`
[7] `http://visjs.org/`
[8] `https://www.npmjs.com/`
[9] `https://github.com/zaphoyd/websocketpp`
[10] `https://www.mpi-forum.org/`

fentlichungen 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 allgemein spezifizierten Funktionen von MPI verwendet werden, sollte das Backend aber auch mit allen anderen gängigen MPI-Implementierungen (z.B. OpenMPI) ausführbar sein.

3.4 Technische Spezifikation

3.4.1 Frontend

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

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 Fraktal 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 empfangene 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.

3.4.2 Backend

3.4.3 Kommunikation 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)

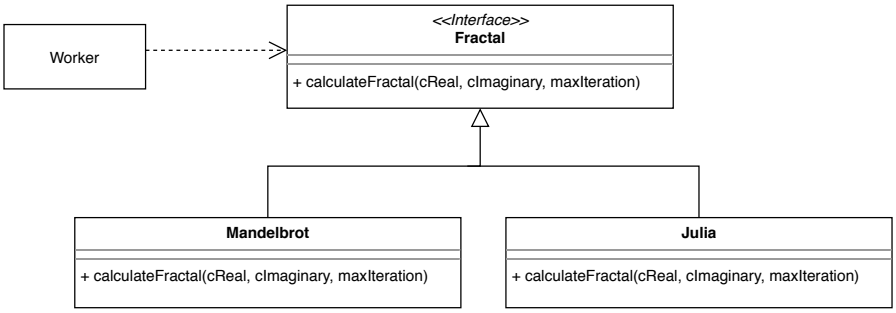

Abbildung 9: Klassendiagramm für Fractal

- 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)

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.

Das Senden der berechneten Region vom Worker zum Host funktioniert hingegen mit blockierenden Sende- und Empfangs-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 Mikrosekunden, 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.

4 Ergebnisse / Evaluation

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

5 Zusammenfassung

- Zusammenfassung
- Ausblick