

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

1.1 Didaktische Ziele

Die Leistung und Geschwindigkeit individueller Rechenkerns stagniert seit einigen Jahren. Moderne Computer erlangen einen Großteil ihrer erhöhten Rechenleistung seit einiger Zeit nur noch durch Parallelisierung. Da dies stets einen geringen Overhead verursacht werden gut durchdachte Konzepte zur Ausnutzung mehrerer Kerne immer wichtiger.

Dieses Projekt soll Nutzern ermöglichen intuitiv ein Gefühl dafür zu gewinnen, welche wichtige Rolle hierbei die Aufteilung der Last zwischen den Rechenknoten spielt.

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:

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

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.

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 können die Berechnungen frei auf verfügbare Rechenknoten verteilt werden, sodass für verschiedenste Aufteilungen eine Rechenzeit 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 Bildschirms als



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

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 Bildschirms 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 1.3). 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

2 Problemstellung und Motivation

- Fachliche Spezifikation in Anhang
- NFRs in Einleitung schreiben

3 Dokumentation der Implementierung

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

3.1 Lastbalancierung

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

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

3.1.1 Naive Strategie

Bei der naiven Strategie (zu finden in den Klassen `NaiveBalancer` und `RecursiveNaiveBalancer`) wird versucht den einzelnen Workern etwa gleich große Teilregionen zuzuweisen. Dies geschieht allerdings ohne Beachtung der eventuell unterschiedlichen Rechenzeiten innerhalb der Teilregionen. Die naive Strategie wurde hier in einer nicht-rekursiven und einer rekursiven Variante implementiert.

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

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

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

Dabei ist `nodeCount` die Anzahl der Worker und `region` die zu unterteilende Region. Es ist zu beachten, dass es sich hier um eine Ganzzahldivision handelt, deren Rest angibt,

wie viele Teilregionen um *region.guaranteedDivisor* breiter sind. Die Höhe berechnet sich analog.

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

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

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

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

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

$$recCounter = \lfloor \log_2 nodeCount \rfloor + 1 \quad (2)$$

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

$$missing = nodeCount - 2^{\lfloor \log_2 nodeCount \rfloor}$$

$$onLowestLevel = missing * 2 \quad (3)$$

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

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

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

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

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

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

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

3.1.2 Strategie mit Vorhersage

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

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

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

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

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

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

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

$$desiredN = \frac{nSum}{nodeCount} \quad (5)$$

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

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

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

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

Die rekursive Variante der Strategie mit Vorhersage verwendet dasselbe Rekursionschema wie ihr naives Gegenstück. Die Werte in `BalancingContext` berechnen sich also wie in (2) und (3). Auch die Abbruchbedingung ist wie in (4). Die Entscheidung, ob horizontal oder vertikal geteilt werden soll, wird auch auf die gleiche Art und Weise gefällt.

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

3.1.3 Leere Regionen

3.1.4 Erweiterung

4 Ergebnisse / Evaluation

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

5 Zusammenfassung

- Zusammenfassung
- Ausblick