LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Parallele Berechnung der Mandelbrotmenge

Wintersemester 2018/19

Maximilian Frühauf Tobias Klasuen Florian Lercher Niels Mündler

Inhaltsverzeichnis

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.

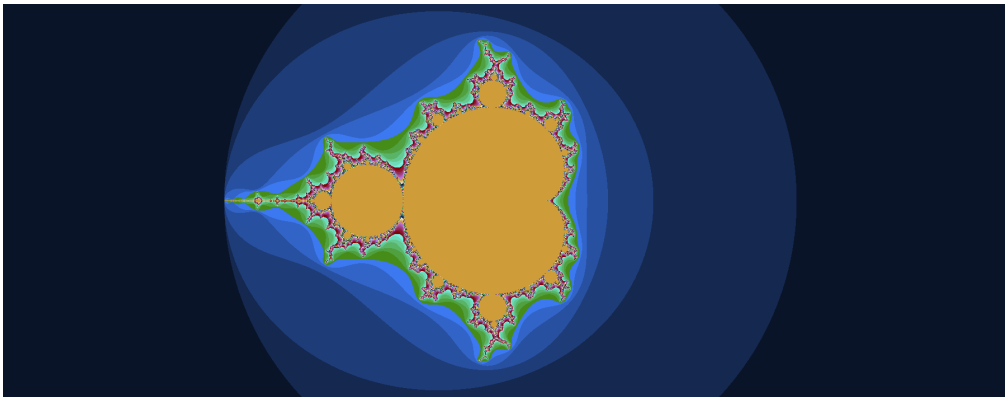

Abbildung 1: Die Mandelbrotmenge, visualisiert in einem Ausschnitt des komplexen Zahlenraumes.

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 Motivation

- Fachliche Spezifikation in Anhang
- NFRs in Einleitung schreiben

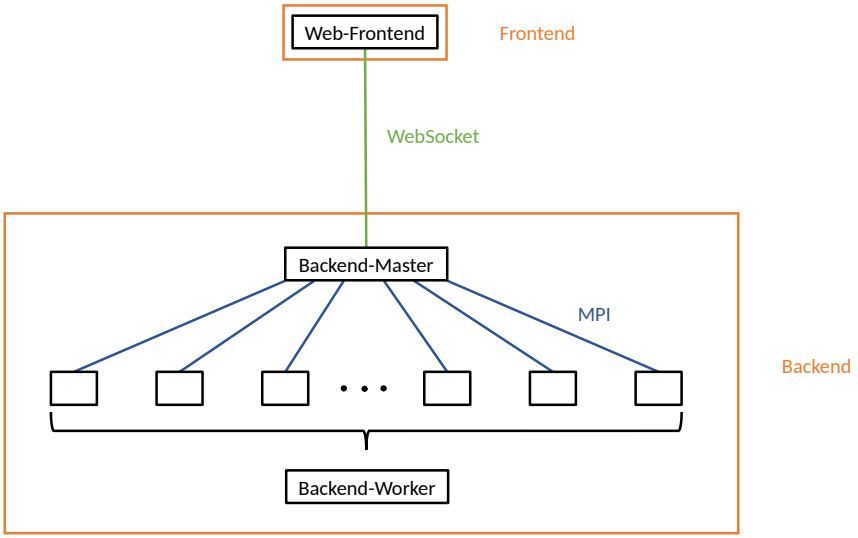

Abbildung 2: Architekturübersicht

3 Dokumentation der Implementierung

3.1 Übersicht

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.

3.2 Installation der Anwendung

Um das System zu installieren, muss das Repository mit git 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 1: Klonen des Repositorys

3.2.1 Lokales Backend

Eine lokale Installation des Backends zu Entwicklungszwecken ist durch einen Docker [3] Container möglich. Dieser bietet eine ähnliche Umgebung zu der des Clusters und ermöglicht schnellere Feedbackzyklen.

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

Quelltext 2: Starten der Entwicklungsumbegung des Backends

Das `run_docker.sh` Skript lädt das benötigte Basis Image, welches alle benötigen 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.

3.2.2 Backend auf HIMMUC Cluster

3.2.3 Installation des Frontends

Das Frontend ist in TypeScript[4] (erweiterung von JavaScript[5]) geschrien und kann somit auf einem beliebigen Endgerät mit einem modernen Webbrowser ausgeführt werden. Um eine Version lokal zu starten, muss die Paketverwaltung npm[6] 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 standard Webbrowser des System verwendet, um diese URL zu öffnen.

3.3 Erläuterung des Backends

3.4 Implementierung der Mandelbrotberechnung

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

[3] `https://www.docker.com/`
[4] `https://www.typescriptlang.org/`
[5] `https://en.wikipedia.org/wiki/JavaScript`
[6] `https://www.npmjs.com/`

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

Quelltext 3: Starten des Frontends

und verteilt sie per MPI auf dedizierte Rechenknoten. Dazu besteht das Backend aus zwei ausführbaren Dateien, `host` und `worker`.

3.4.1 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`[7] in der hier beschriebenen Reihenfolge spezifiert. Es sollte hierbeit eine CMake-Version über 3.7.0 gewählt werden und die C++11 Standarts[8] werden vorrausgesetzt. Zudem werden für das Projekt "Mandelbrot"werden alle Dateien im Order `include` eingebunden. In diesem Ordner liegen die Header-Dateien für alle projektinternen C++-Quelldateien. Anschließend werden alle C++-Quelldateien (Endung "`.cpp`") aus dem Ordner `src` in einer Liste gesammelt, mit Ausnahme jedoch der oben genannten, exklusiven Quelldateien. Die erzeugte Liste und die jeweils exklusiven Dateien werden dann den ausführbaren Dateien `host` und `worker` zugeordnet.

Um die verwendeten Biblioteken verfügbar zu machen werden anschließend die Header der installierten MPI-Bibliotek sowie die Header der Bibliotheken rapidjson[9], websocketpp[10] und boost[11] Diese werden respektive verwendet um JSON zu parsen und enkodieren, Websocket-Verbindungen aufzubauen und darüber zu kommunizieren sowie um diese Bibliothek zu unterstützen. Da für die boost Bibliothek dabei Header nicht genügen und die systemweite Verfügbarkeit der kompilierten boost-Bibliothek nicht ga-

[7] Ein Programm, welches die Erstellung von Makefiles vereinfacht in dem es sie automatisch an die Umgebung des Build-Systems anpasst. `https://cmake.org/`

[8] `https://isocpp.org/wiki/faq/cpp11`

[9] `http://rapidjson.org`

[10] `https://github.com/zaphoyd/websocketpp`

[11] `https://www.boost.org/`

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

Quelltext 4: Initialisierung der MPI-Prozesse in init.cpp

rantiert 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 und spezielle Flags für die Websocket-library und MPI gesetzt.

3.4.2 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 4. 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 5 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 5: Initialisierung des Host-Prozesses in host.main.cpp

```
387    // Websockets
388    // Start a thread that hosts the server
389    std::thread websocket_server(start_server);
390
391    // Start Websocket-Result-Thread (sends RegionData filled with
           computed mandelbrot data to frontend)
392    std::thread websocket_result(send);
393
394    // Init usable_nodes and set Host as not usable
395    usable_nodes = new bool[world_size];
396    usable_nodes[world_rank] = false;
397
398    // Test if all cores are available
```

Quelltext 6: Starten des Websocketservers bei der Initialisierung des Host-Programmes in Host.cpp

3.5 Host Funktionalitäten

3.5.1 Websocketverbindung

Direkt nach der Initialisierung des Host-Programmes in Quelltext 6, 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 mit den Methoden zum Behandeln geöffneter und geschlossener Verbindungen und einer Methode um Nachrichten des Clients zu behandeln, `Host::handle_region_request`.

Diese Methode versucht, den Inhalt der empfangenen Nachricht als JSON zu dekodieren und entnimmt die für das struct Region notwendigen Werte unter gleichem Namen dem Objekt, das unter dem Schlüssel `"region"` in der geparsten Anfrage gespeichert ist. Außerdem erwartet es unter dem Schlüssel `"balancer"` einen String, der den zu wählenden Lastbalancierer bestimmt und unter dem Schlüssel `"type"` den String `"regionRequest"`. Mögliche Zeichenketten hierfür sind in den Klassen der Balancierer unter `backend/src/balancer` in der globalen Variable `Klassenname::NAME` gespeichert. Ein Beispiel für einen Namen kann in Quelltext 7 gefunden werden:

```
11 const std::string PredictionBalancer::NAME = "prediction";
```

Quelltext 7: Namensdefinition des PredictionBalancers

```
{
    "type": "regionRequest",
    "region": {
        "minReal": -0.251953125,
        "maxImag": -0.8388671875,
        "maxReal": -0.2216796875,
        "minImag": -0.8505859375,
        "width": 1984,
        "height": 768,
        "hOffset": 0,
        "vOffset": 0,
        "validation": 8,
        "guaranteedDivisor": 64,
        "maxIteration": 1019
    },
    "balancer": "naiveRecursive",
    "fractal": "mandelbrot"
}
```

Quelltext 8: Eine gültige Anfrage einer Region in JSON

Ein Beispiel für eine gültige Regionsanfrage ist in Quelltext 8 zu finden.

Nach dem Parsen der Nachricht wird die Region global festgelegt und beim korrekten Lastbalancierer eine Zerlegung der angefragten Region über `Balancer::balanceLoad` gestartet. Aus dem Ergebnis werden leere Regionen werden anschließend aussortiert und alle anderen in eine per mutex Thread-gesicherte Datenstruktur gelegt. Der MPI verwendende Thread, der diese Regionen anschließend an die Worker sendet wird über das Setzen eines boolschen Wertes auf `true` darüber informiert, dass neue Regionen zum versenden zur Verfügung stehen. Dabei ist der Rang des Workers, der eine Region berechnen soll genau der Index der Region in der Datenstruktur sein. Ist ein Worker nicht verfügbar, so werden alle folgenden Ränge um eins erhöht. 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 9 entnommen werden.

3.6 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

```
{
    "type": "region",
    "regionCount": 37,
    "regions": [{
        "rank": 1,
        "computationTime": 0,
        "region": {
            "minReal": -0.251953125,
            "maxImag": -0.8408203125,
            "maxReal": -0.24609375,
            "minImag": -0.8427734375,
            "width": 384,
            "height": 128,
            "hOffset": 0,
            "vOffset": 0,
            "maxIteration": 1019,
            "validation": 8,
            "guaranteedDivisor": 64
        }
    }, {
        "rank": 2,
        "computationTime": 0,
        "region": {
            "minReal": -0.251953125,
            ......
    }, {
        "rank": 37,
        "computationTime": 0,
        "region": {
            "minReal": -0.2275390625,
            "maxImag": -0.84765625,
            "maxReal": -0.2216796875,
            "minImag": -0.8486328125,
            "width": 384,
            "height": 64,
            "hOffset": 1600,
            "vOffset": 448,
            "maxIteration": 1019,
            "validation": 8,
            "guaranteedDivisor": 64
        }
    }]
}
```

Quelltext 9: Eine gültige Antwort auf die Region aus Quelltext 8 in JSON

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.

3.6.1 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 Teiregionen 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:

$$recCounter = \lfloor \log_2 nodeCount \rfloor + 1 \tag{2}$$

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

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

$$onLowestLevel = missing * 2 \tag{3}$$

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:

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

$resultIndex$ 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 Balancierer 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.

3.6.2 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 Predicter 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:

$$desiredN = \frac{nSum}{nodeCount} \tag{5}$$

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 Rekursionsschema 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.

3.6.3 Leere Regionen

3.6.4 Erweiterung

3.7 Berechnung der Mandelbrotmenge

3.7.1 Berechnung mithilfe von SIMD

Um die Berechnungen intern noch zu beschleunigen, kann SIMD zur parallelen Bearbeitung mehrerer Punkte verwendet werden. Dazu ist es hilfreich, sich zunächst vor Augen zu führen, wie ein Vektor komplexer Koordinaten ohne SIMD verarbeitet würde. Dies ist in Quelltext 10 zu sehen. Die Berechnung der einzelnen Kooridnaten 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 Betragsmäßig kleiner gleich 2 ist. Dies gilt, da alle $|z_{n+i}| > 2$ sofern $|z_n| > 2$ [1].

Damit kann die Berechnung relativ simpel via Arm NEON Compiler Intrinsics[12] implementiert werden (siehe dazu Quelltext 11). Diese Intrinsics ermöglichen eine Verwendung der nativen SIMD-Befehle, wobei der Compiler sich um die Verwendung

[12]Details im Abschnitt "Compiler Intrinsicsünter `https://developer.arm.com/technologies/neon`

```
8  void MandelbrotVect::calculateFractal(precision_t* cReal, precision_t*
       cImaginary, unsigned short int maxIteration, int vectorLength,
       unsigned short int* dest) {
9      if(vectorLength <= 0){
10         throw std::invalid_argument("vectorLength may not be less than 1.
               ");
11     }
12     std::fill_n(dest, vectorLength, 0);
13     precision_t* zReal = new precision_t[vectorLength];
14     precision_t* zImaginary = new precision_t[vectorLength];
15     precision_t* nextZReal = new precision_t[vectorLength];
16     precision_t* nextZImaginary = new precision_t[vectorLength];
17     // Factor that is added on iteration count
18     // is 0 or 1 and determines whether that point is still being
           computed
19     bool* factor = new bool[vectorLength];
20     std::fill_n(factor, vectorLength, 1);
21     // Integer storing number of Z components with absolute value
22     // below two => continue computation
23     unsigned int lessThanTwo = vectorLength; // as we begin with ZReal/
           ZImag as 0
24     int i = 0;
25     while (i < maxIteration && lessThanTwo > 0){
26         lessThanTwo = 0;
27         for(int k = 0; k < vectorLength; k++){
28             // Compute next step in iteration
29             nextZReal[k] = (zReal[k] * zReal[k] - zImaginary[k] *
                   zImaginary[k]) + cReal[k];
30             nextZImaginary[k] = 2 * (zReal[k] * zImaginary[k]) +
                   cImaginary[k];
31             zReal[k] = nextZReal[k];
32             zImaginary[k] = nextZImaginary[k];
33             // Determine whether to stop
34             factor[k] = (zReal[k] * zReal[k] + zImaginary[k] * zImaginary
                   [k] < 4.0) ? 1 : 0;
35             // sum => if any number is still less than two, we need to
                   continue
36             lessThanTwo += factor[k];
37             // increase number of iterations if this number wasnt aborted
                    yet
38             dest[k] += factor[k];
39         }
40     }
41     delete[] zReal;
42     delete[] zImaginary;
43     delete[] nextZReal;
44     delete[] nextZImaginary;
45     delete[] factor;
46 }
```

Quelltext 10: Bearbeitung eines Vektors komplexer Koordinaten in C++

der SIMD-Register kümmert. Dadurch wird lesbarer Code ermöglicht, der sich stärker am zu implementierenden Algorithmus orientiert.

Zu den hier benötigten mathematischen Operationen (z.B. der Addition) wird hierbei das Compiler Intrinsic nach folgendem Schema erzeugt: "v*opc*q_f*pr*"mit dem Operationscode *opc* (z.B. *add* für Addition). "vïst 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 f*pr* 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.

Bei der Implementierung 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 Ergbenis des Vergleiches und der Verundung ist ein Vektor vorzeichenloser 32 oder 64 bit Ganzzahlen, weshalb das Postfix für diese Operationen *u32* oder *u64* lautet.

3.8 Erläuterung des Frontends

```
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(vcltq_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 11: Parallelisierte Bearbeitung eines Vektors komplexer Koordinaten in 32 bit Gleitkommapräzision in C++ mit Arm NEON Compiler Intrinsics für SIMD

4 Ergebnisse / Evaluation

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

5 Zusammenfassung

- Zusammenfassung
- Ausblick

Abbildungsverzeichnis

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).