Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Fachbereich 4: INFORMATIK, KOMMUNIKATION UND WIRTSCHAFT

Beleg

Wellengleichung

Studiengang:	**Angewandte Informatik**
Lehrveranstaltung:	**Parallel Systems**
Autor:	Chris Rebbelin, s0548921
Dozent:	Sebastian Bauer
Datum:	Berlin, 02. August 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Kapitel 1

Einleitung

In diesem Kapitel werden zunächst die Aufgabenstellung vorgestellt und der weitere Aufbau erläutert.

1.1 Aufgabe

Die Aufgabe dieses Beleges war es die Amplituden einer vibrierenden Saite auf Grundlage der Wellengleichung im eindimensionalen Fall zu visualisieren. Eine Approximation kann die folgende Gleichung darstellen:

$$A(i, t+1) = 2A(i, t) - A(i, t-1) + c(A(i-1, t) - 2A(i, t) + A(i+1, t)) \tag{1.1}$$

Neben einer sequentiellen Berechnung der Visualisierung sollten auch mindestens zwei unterschiedliche Frameworks verwendet werden, welche die Rechenkraft mehrerer Prozessorkerne oder Grafikprozessoren in Anspruch nehmen. Ein mögliches Aussehen einer so berechneten Kurve ist in Abbildung 1.1 zu sehen.

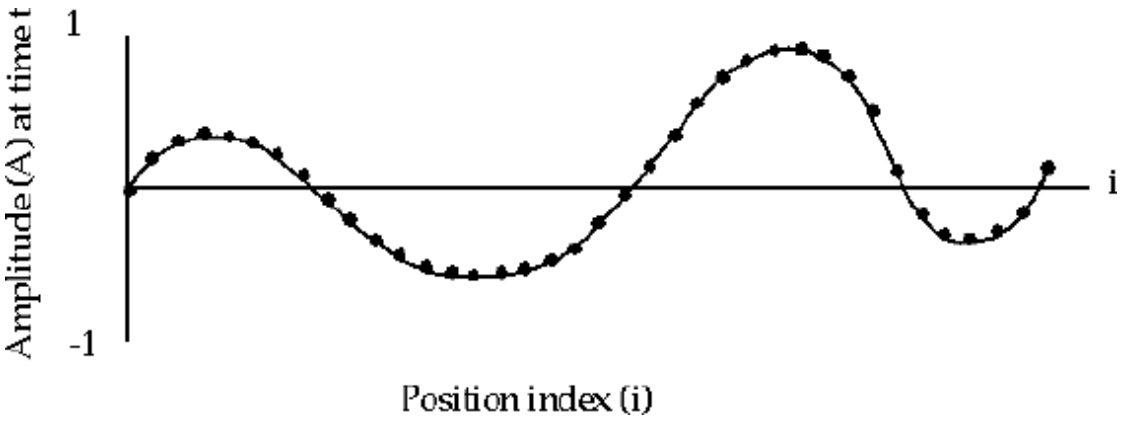

Abbildung 1.1: diskrete Punkte einer Welle

Weitere Rahmenbedingungen waren:

- Visualisierung (Benutzeroberfläche)
- Trennen von Logik und Darstellung
- Konfiguration per Benutzeroberfläche und/oder Konfigurationsdatei
- Perturbationen ("Festhalten"der Saite)
- Performancevergleich der verschiedenen Varianten
- Präsentation des Standes im Semester
- Beleg als PDF
- Quellcode-Dokumentation

1.2 Aufbau

Gegenstand dieser Arbeit sind die Ergebnisse des Belegs und der Weg dorthin. Dies beinhaltet insbesondere den Performancevergleich und die Benutzung der Programme (siehe Rahmenbedingungen). Zunächst werden einige Grundlagen zur hier verwendeten Parallelisierung und zu den Frameworks erläutert. Des Weiteren wird auf die mathematischen Herleitung der Aufgabe eingegangen. Anschließend wird die Struktur und der grobe Ablauf der drei einzelnen Programme dargestellt. Abschließend wird das Benchmarkverfahren erläutert und ein Fazit gezogen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die nötigen Grundlagen zur Wellengleichung und den verwendeten Frameworks zur Parallelisierung kurz dargestellt.

2.1 Wellengleichung

Die Wellengleichung im eindimensionalen Fall ist die partielle Differentialgleichung 2.1. [1, 2]

$$u_{tt} = c^2 u_{xx} \tag{2.1}$$

Die Differentiale können dabei auch in anderer Form notiert werden. Sei $A(i,t)$ eine Funktion, die für die Stelle i zur Zeit t die Auslenkung der Welle berechnet. Dann ergibt sich Formel 2.2.

$$\frac{\partial^2}{\partial t^2} A(i,t) = c^2 \frac{\partial^2}{\partial i^2} A(i,t) \tag{2.2}$$

Die zweiten Ableitungen können dann mit Formel 2.3 approximiert werden. Dabei wird vorausgesetzt, dass die Abstände zwischen den Punkten jeweils δh beträgt, also eine Diskretisierung des Definitionsbereiches stattfindet.

$$\frac{\partial^2}{\partial x^2} f(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{2.3}$$

Die Approximation aus 2.3 in 2.2 eingesetzt ergibt Formel 2.4.

$$\frac{A(i,t+1) - 2A(i,t) + A(i,t-1)}{t^2} = c^2 \left(\frac{A(i+1,t) - 2A(i,t) + A(i-1,t)}{i^2} \right) \tag{2.4}$$

Durch geeignetes Umstellen von Formel 2.4 ergibt sich dann die gegebene Approximationsformel 2.5. [1,2]

$$A(i,t+1) = 2A(i,t) - A(i,t-1) + C(A(i-1,t) - 2A(i,t) + A(i+1,t)) \tag{2.5}$$

Dabei bedeutet $A(i,t+1)$ die Auslenkung der Welle an der Stelle i zum Zeitpunkt $t+1$.

Die Variable C ist dabei eine Konstante, welche die Geschwindigkeit der Wellenbewegung beeinflusst. (2.6)

$$C = \frac{c^2 t^2}{i^2} \tag{2.6}$$

Für Werte von C größer als 1 und kleiner als 0 wird die entstehenden Welle instabil und liefert keine genauen Ergebnisse mehr.

Es ist zu erkennen, das der neu berechnete Wert jeweils von dem Wert aus dem vorherigen Zeitschritt ($A(i,t-1)$) und von den Nachbarwerten des momentanen Zeitschrittes ($A(i-1,t)$ und $A(i+1,t)$) abhängt.
Damit kann eine eventuelle Parallelisierung also nicht über die Zeitschritte hinweg stattfinden, da es hier aufgrund der Abhängigkeiten wahrscheinlich zu keinem Performancegewinn kommen würde (es müsste immer auf einen Prozess gewartet werden).
Stattdessen bietet sich die Parallelisierung der Werteberechnung an. Diese kann unabhängig voneinander für jeden Punkt gleichzeitig geschehen.

2.2 Parallelisierung

Ich habe mich für die Frameworks OpenMP und OpenMPI entschieden. Einerseits hatte ich aus einem früheren Veranstaltungen bereits einige Erfahrungen mit diesen gesammelten, anderseits wurden speziell diese auch in den Übungen dieser Veranstaltung detailliert besprochen. Im folgenden werden beide kurz dargestellt.

2.2.1 OpenMP

Auf einem Computer (PC) befindet sich in der Regel ein eng gekoppeltes Speichersystem, d.h. alle Prozessoren greifen auf den gleichen gemeinsamen globalen Speicher zu, besitzen jedoch auch einen eigenen schnelleren lokalen Speicher (Cache). *OpenMP* wird auf diesen Systemen eingesetzt. Es steht für Open Multi-Processing und ist eine Programmierschnittstelle für C/C++ und Fortran.
Die Idee hinter *OpenMP* ist es, Schleifen, die normalerweise sequentiell abgearbeitet werden, aufzuteilen und von mehreren Prozessoren ausführen zu lassen. Dies ist natürlich nicht für jede Schleife möglich, weshalb der Entwickler entscheiden muss welche Schleifen parallelisiert ausgeführt werden können, welche nicht und welche vielleicht mit Anpassungen parallelisiert werden könnte. Eine Schleife wird über Pragma-Direktiven für *OpenMP* markiert und konfiguriert. [3]

2.2.2 OpenMPI

Neben eng gekoppelten Speichersystemen gibt es auch lose gekoppelte Speichersystem, d.h. es gibt keinen globalen Speicher, sondern jeder Prozessor hat nur seinen eigenen Speicher. Eine Kommunikation erfolgt hier über ein Netzwerk.

Ein Standard, der diesen Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Systemen beschreibt, ist das Message Passing Interface (MPI). Eine Implementierung davon ist das hier verwendete *OpenMPI*.

Der Kern von *OpenMPI* ist das Austauschen von Informationen zwischen den Prozessoren über Nachrichten (Message Passing). In der Regel wird die Anwendung von einem Master-Prozess initialisiert, welcher dann die Daten an alle Prozesse verteilt und nach der Berechnung wieder einsammelt.

Um den Nachrichtenaustausch zwischen Prozessen in *OpenMPI* zu realisieren, werden unter anderem die Funktionen *Send*, *Receive* und *Broadcast* verwendet. Die Funktion *Send* ist in der Lage Daten in Form von Variablen zu anderen Prozessen zu schicken (siehe Listing 2.1). Hier ist das Senden eine blockierende Funktion, das heißt die Programmausführung erfolgt erst, wenn die Nachricht entsprechend empfangen wurde durch ein *Recv*. [4]

Listing 2.1: MPI Send

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
    MPI_Comm comm)
```

Die Funktion *Receive* ist das Gegenstück zu *Send*, sie ermöglicht das Empfangen von Daten zu einer Variable (siehe Listing 2.2). Diese Variante von *Recieve* ist ebenfalls blockierend, das heißt es muss mit der Programmausführung erst auf die Daten gewartet werden. [4].

Listing 2.2: MPI Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status *status)
```

Damit muss bei der Verwendung dieser Funktion darauf geachtet werden, keinen Deadlock zu produzieren. [5]

Listing 2.3: MPI Deadlock

```
 if (rank == 0) {
  MPI_Send(..., 1, tag, MPI_COMM_WORLD);
  MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
  MPI_Send(..., 0, tag, MPI_COMM_WORLD);
  MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
}
```

Hier senden der Master und der Worker gleichzeitig Daten aneinander und warten dann auf Empfangsbestätigung, die aber nie eintrifft, da der jeweils andere auch wartet (siehe Listing 2.3). Eine Lösung wäre es, die Reihenfolge der Funktionen zu vertauschen (Listing 2.4).

Listing 2.4: Deadlock vermieden

```
 if (rank == 0) {
  MPI_Send(..., 1, tag, MPI_COMM_WORLD);
  MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
  MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
  MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

Die Funktion Broadcast schließlich vereint beide Funktionen. Der Prozess, dessen *ID root* entspricht, führt einen *Send* aus, alle anderen Prozesse werden dann zu einem *Recv* quasi. Am Ende besitzen alle Prozesse den gleichen Wert (siehe Listing 2.5). [4].

Listing 2.5: MPI Bcast

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
    comm )
```

Kapitel 3

Entwurf

Hier soll der Entwurf für die Implementierung der Programme erläutert werden.

3.1 Struktur

Um die verschiedenen Implementierungen möglichst unabhängig voneinander zu testen, wurden drei separate Programme entwickelt. Die sequentielle Variante unterscheidet sich dabei nur gering von der *OpenMP*-Variante, was an der einfachen Handhabung über die Pragma-Direktiven liegt. Die Arbeitsweise von *OpenMPI* führte jedoch zu deutlichen Unterschieden, vor allem auch durch Verwendung eines speziellen Compilers und Ausführungsprogrammes. Dies war auch ein Grund für die Trennung der einzelnen Varianten.
Da die verwendeten Frameworks primär in der Programmiersprache C vorliegen und diese auch in der Vorlesung verwendet wurde, ist auch hier diese Sprache zum Einsatz gekommen.

Die Struktur der Implementierung entspricht der Beispielstruktur aus der Übung. Dabei wird jedes Programm in zwei Teile unterteilt, welche aus jeweils zwei Dateien bestehen (c-Datei und Header-Datei): *myWave* und *core*.

3.1.1 myWave

Hier findet die Programminitialisierung statt und hier sind auch alle Methoden und Variablen, die für die grafische Benutzeroberfläche verwendet werden, angelegt. Für die Visualisierung habe ich mich für das *SDL*-Framework entschieden. Als Vorgabe für die Visualisierung wurde das Beispielbild aus der Projektbeschreibung verwendet. Da es in dieser Arbeit um die Parallelisierung gehen sollte, wird an dieser Stelle nicht weiter auf die Visualisierung eingegangen.
Die Programmlogik der Wellenberechnung wird von hier aus aufgerufen, das heißt der gesamte Programmfluss wird von hier gesteuert. Damit soll der Vorgabe entsprochen werden, die Logik und die Darstellung zu trennen (siehe Kapitel 1.1).

3.1.2 core

Hier ist die Logik der Wellenberechnung implementiert. Es werden drei Arrays allokiert, die den Verlauf der Welle darstellen. Dabei stellt eins die Werte in der Vergangenheit, eins die der Gegenwart und eins die der Zukunft dar. Die Arrays der Vergangenheit und der Gegenwart werden initial mit einer Sinuskurve befüllt (siehe Kapitel 4).

Die Werte der Zukunft werden dann immer mithilfe der in Abschnitt 2.1 vorgestellten Gleichung und den Werten der Gegenwart und der Vergangenheit berechnet. Nach einer Berechnung müssen alle Werte eine Schritt in die Vergangenheit kopiert werden. Im Folgenden wird solch eine Berechnung als ein Zeitschritt bezeichnet.

Ebenfalls hier implementiert ist die Konfiguration des Programmes. Die Parameter können sowohl über eine Datei `settings.txt` oder über die Kommandozeile eingelesen werden. Für jeden Parameter existiert ein Standardwert, der verwendet wird, wenn kein Wert übergeben wurde. Nach dem Einlesen werden alle Parameter auf Korrektheit überprüft, bevor der Logikteil aufgerufen wird. Im folgenden sind alle Parameter aufgeführt. Eine genauere Übersicht inklusive der Kommandozeilenargumente kann über `--help` aufgerufen werden. Die Kommandozeilenargumente existieren jeweils in der Kurz- und der Langform (z.B. `-h` oder `--help`).
Sowohl die Einstellungsdatei als auch die Hilfeausgabe der Programme geben Auskunft über die Standardwerte.

- SPEED: Geschwindigkeit der Welle
- NUMBER_OF_TIME_STEPS: Anzahl Zeitschritte
- LINE_INTERVAL_END: Rechte Intervalgrenze der Welle
- NUMBER_OF_POINTS: Anzahl diskreter Punkte
- SHOW_GUI: Ob die Welle visualisiert werden soll
- NUMBER_OF_PERIODS: Anzahl der Perioden der Sinuskurve
- AMPLITUDE: Amplitude der Sinuskurve
- LAMBDA: Dämpfungsfaktor für die Welle, siehe Kapitel 4.4.2
- PRINT_VALUES: Ob die Endergebnisse ausgegeben werden sollen

3.1.3 Besonderheit OpenMPI

Für *OpenMPI* wurden aufgrund der unterschiedliche Arbeitsweise einige Änderungen vorgenommen, die im Folgenden dargestellt werden sollen.

Die parallele Simulation der Wellengleichung mit *OpenMPI* wird hier über den Grundansatz des Divide-and-Conquer-Prinzips angegangen.
Dabei wird das Problem in mehrere Teilprobleme aufgeteilt. Anschließend werden die Teilprobleme separat gelöst und wieder zusammengefasst. Die Zusammenführung aller Teillösungen führt damit zur Lösung des Gesamtproblems.

Abbildung 3.1 zeigt die Vorgehensweise.

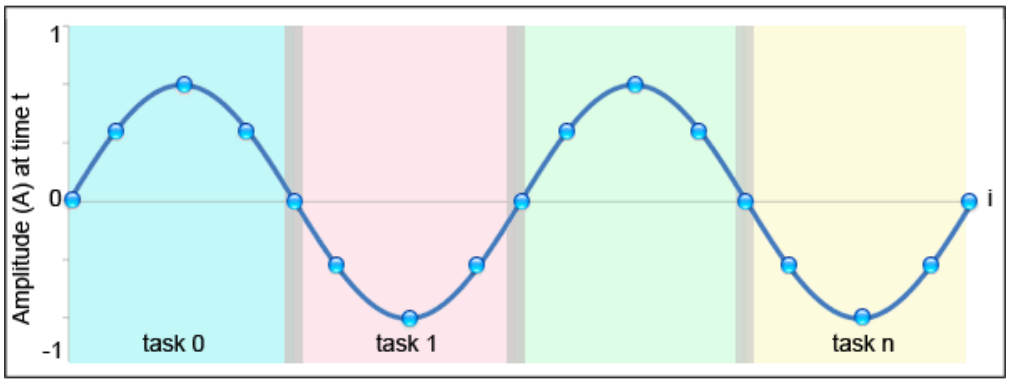

Abbildung 3.1: Aufteilung des Problems in Teilprobleme

Zunächst muss die Problemgröße effizient aufgeteilt werden. Jeder Prozess soll gleich viele Punkte bearbeiten. Bei der anschließende Bearbeitung der Punkte muss an den Rändern eine Kommunikation mittels MPI-Funktionen erfolgen. Abschließend muss ein Prozess die Daten geordnet einsammeln.

Weiterhin muss für jede Abbruchbedingung, gerade bei dem Einlesen der Konfiguration darauf geachtet werden, dass immer alle Prozesse gleichzeitig terminieren über die Funktion `MPI_Finalize()`. Bei Printausgaben muss darauf geachtet werden, dass jeweils nur ein Prozess diese ausführt.

Kapitel 4

Implementierung

Es wurden drei einzelne Programme geschrieben, die jeweils die im vorherigen Kapitel genannte Unterteilung des Codes besitzen. Weiterhin existiert für jedes Programm ein eigenes `makefile` und `doxyfile`. Alle Programme nutzen als Standardeinstellungsdatei die `settings.txt`.

Drei Arrays vom Datentype `double` repräsentieren die Werte in der Vergangenheit ($A(i, t-1)$) `previousStep`, in der Gegenwart ($A(i, t)$) `currentStep` und der Zukunft ($A(i, t+1)$) `nextStep`.

Für den ersten Zeitschritt werden `previousStep` und `currentStep` mit den Werten der Sinuskurve vorbelegt. Die Formel ist in 4.1 zu sehen.

$$A * sin(2 * x * \pi * P/(I-1)) \tag{4.1}$$

Dabei ist A die Amplitude der Sinuskurve und P die Anzahl der Perioden. I gibt an bis wohin soll die Welle berechnet werden soll. Alle drei Werte können über ihren entsprechenden Einstellungsparameter verändert werden.

4.1 Sequentiell

Die Umsetzung der Formel kann beispielsweise so geschrieben werden.

Listing 4.1: sequentielle Umsetzung

```
for (i = 1; i < nPoints - 1; i++)
{
 nextStep[i] = 2.0 * currentStep[i] - previousStep[i] + cSquared * (currentStep[i -
      1] - (2.0 * currentStep[i]) + currentStep[i + 1]);
}

nextStep[0] = 0.0;
nextStep[nPoints - 1] = 0.0;

for (i = 1; i < nPoints - 1; i++)
{
previousStep[i] = currentStep[i];
currentStep[i] = nextStep[i];
}
```

Dabei wird erst die Berechnung nach der Formel durchgeführt und anschließend die Werte jeweils einen Schritt in die ''Vergangenheit'' kopiert.

Die Ausführungszeit kann noch verkürzt werden, wenn statt des kostspieligen Kopierens der Werte einfach die Zeiger auf diese getauscht werden.

Listing 4.2: Effizienter Zeigertausch

```
double *tempStep = previousStep;
previousStep = currentStep;
currentStep = nextStep;
nextStep = tempStep;
```

4.2 OpenMP

Das Parallelisieren der Berechnung aus dem sequentiellen Programm erfolgt über ein Pragma von *OpenMP*. Weiterhin muss im `makefile` die Compileroption `-fopenmp` gesetzt werden. Ansonsten entspricht dieses Programm exakt dem sequentiellen.
Um die Anzahl der Threads zu verändern, muss vor Aufruf des Programs die Umgebungsvariable `OMP_NUM_THREADS` geändert werden (bspw. `OMP_NUM_THREADS=4 ./myWaveMP`).

Listing 4.3: sequentieller Code mit OpenMP Pragma

```
#pragma omp parallel for shared(nextStep, currentStep, previousStep, cSquared,
    nPoints) private(i)
for (i = 1; i < nPoints - 1; i++)
{
nextStep[i] = 2.0 * currentStep[i] - previousStep[i] + cSquared * (currentStep[i -
    1] - (2.0 * currentStep[i]) + currentStep[i + 1]);
}

// ...
```

Nur `i` ist hier als privat deklariert, da alle anderen Variablen unabhängig voneinander jedem Thread zur Verfügung stehen.

4.3 OpenMPI

Die Implementierung der verteilten Wellensimulation hat einige Ergänzungen am Code des sequentiellen Programmes nötig gemacht. Beim Allokieren der Arrays muss zunächst die Größe der Dateimenge für jeden Prozess festgelegt werden. Am Ende der Berechnung muss weiterhin eine zusätzliche Funktion alle berechneten Werte aller Prozesse wieder zusammenführen.
Die Funktion `CHECK()` dient jeweils dem Überprüfen des zurückgegebene Fehlercodes von den MPI-Funktionen.

Listing 4.4: OpenMPI Zeitschritt

```
for (i = 1; i < right - left; i++)
{
nextStep[i] = 2.0 * currentStep[i] - previousStep[i] + cSquared * (currentStep[i -
    1] - (2.0 * currentStep[i]) + currentStep[i + 1]);
}

if (id != FIRST)
{
CHECK(MPI_Send(&nextStep[1], 1, MPI_DOUBLE, id - 1, R_TO_L, MPI_COMM_WORLD));
CHECK(MPI_Recv(&nextStep[0], 1, MPI_DOUBLE, id - 1, L_TO_R, MPI_COMM_WORLD, &status
    ));
}
```

```
else
{
nextStep[0] = 0.0;
}

if (id != LAST)
{
CHECK(MPI_Send(&nextStep[right - left - 1], 1, MPI_DOUBLE, id + 1, L_TO_R,
    MPI_COMM_WORLD));
CHECK(MPI_Recv(&nextStep[right - left], 1, MPI_DOUBLE, id + 1, R_TO_L,
    MPI_COMM_WORLD, &status));
}
else
{
nextStep[right - left] = 0.0;
}

double *tempStep = previousStep;
previousStep = currentStep;
currentStep = nextStep;
nextStep = tempStep;
```

Im Folgenden soll diese Implementierung an einem vereinfachten Beispiel dargestellt werden. Angenommen wird ein Array mit 20 Punkten (Abbildung 4.1). Es sind vier Prozesse (*P0, P1, P2* und *P3*) vorhanden.

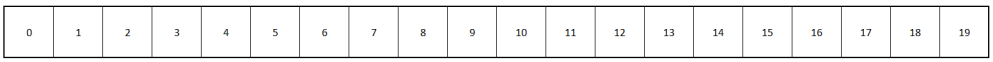

Abbildung 4.1: Globales Array mit 20 Werten

Jeder Prozess bekommt ungefähr die gleiche Menge an Punkten zugewiesen, unter der Bedingung das sich jeweils zwei Werte an den Grenzen überschneiden. (Abbildung 4.2)

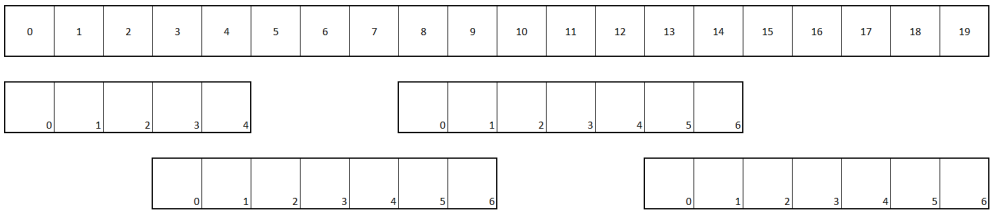

Abbildung 4.2: Aufteilung auf vier Prozesse mit Überschneidung an den Rändern

Entsprechend dem vorangegangenen Listing, berechnet jeder Prozess zunächst parallel die Werte von Index *1* bis *Ende - 1*. Offen bleiben damit für jeden Prozess nur die Randwerte. Global betrachten sind hier bereits alle Werte bis auf den ersten und letzten berechnet worden (Abbildung 4.3).

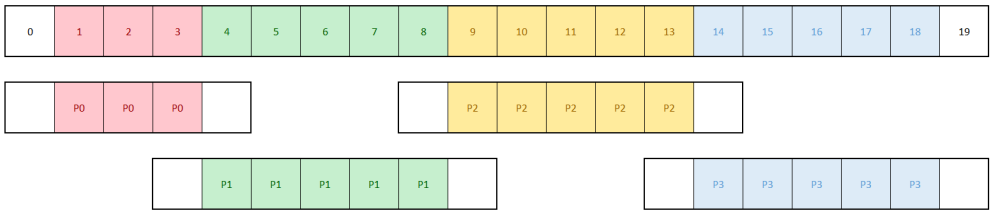

Abbildung 4.3: Parallele Berechnung der inneren Werte

Jetzt muss zweimal eine Abfrage erfolgen. Wenn der Prozess der erste ist (hier *P0*), besitzt er keinen linken Nachbarn und kann seinen linken Randwert mit der Randbedingung der Wellengleichung belegen ($A(0,t) = 0.0$).
Genauso kann der letzte Prozess, der keine rechten Nachbarn mehr hat (hier *P3*), seinen rechten Randwert ebenfalls auf 0.0 setzen (Abbildung 4.4).

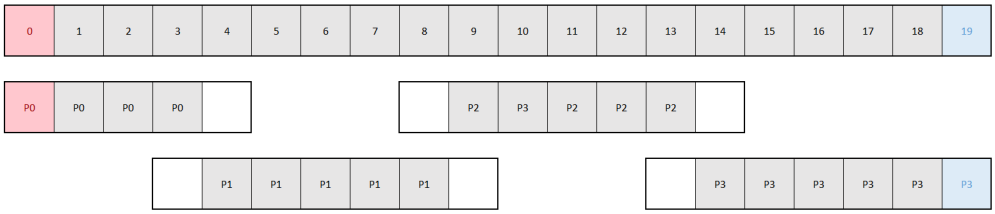

Abbildung 4.4: Setzen der äußersten Randwerte

Zu diesem Zeitpunkt wäre das globale Array vollständig belegt. Allerdings fehlen den inneren Prozessen die Randwerte für den nächsten Zeitschritt. Für die Berechnung des Wertes am Index *1* von *P1* muss dieser die Werte an Index *0* und Index *2* kennen. Der Wert an Index *0* ist zwar berechnet worden, aber nicht von *P1*, sondern von *P0*. Damit ist der Wert für *P1* noch nicht benutzbar.
Deshalb muss eine Kommunikation zwischen den Prozessen erfolgen.
Jeder Prozess, der einen linken Nachbarn besitzt, sendet diesem seinen Wert an Index *1* und erwartet einen Wert, den er bei sich an Index *0* einträgt. Genauso sendet jeder Prozess mit einem rechten Nachbarn diesem seinen Wert an Index *Ende - 1* und und erwartet einen Wert, den er bei sich an Index *Ende* einträgt (Abbildung 4.5).

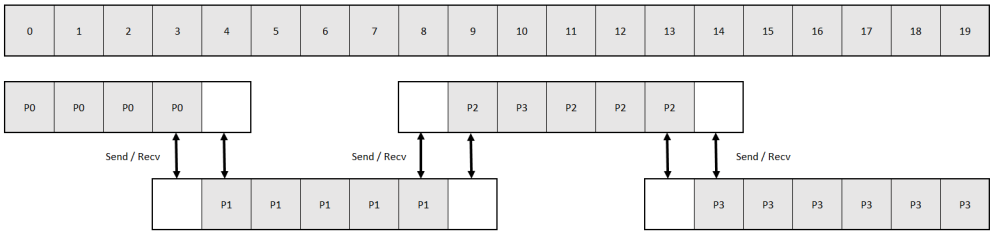

Abbildung 4.5: MPI Prozesskommmunikation der inneren Randwerte

Jetzt besitzt jeder Prozess alle Werte, die er für die Berechnung im nächsten Zeitschritt benötigt (Abbildung 4.6). Abschließend werden die Arrays getauscht über ihre Zeiger und der nächste Zeitschritt beginnt.

Abbildung 4.6: Zeitschritt abgeschlossen

Das folgende Listing 4.5 zeigt das abschließende Einsammeln aller Werte durch den Master-Prozess.
Als erstes trägt der Master selbst seine Ergebnisse in das globale Array `globalStep` ein.
Jeder Prozess der nicht der Master ist, sendet zunächst zwei Werte an den Master.:

- Startwert im globalen Array
- Anzahl der bearbeiteten Punkte

Dieser `Send` trägt das Tag *INFO*. Das ist nötig, damit der Master für den nachfolgenden `Send` den passenden `Recv` bereitstellen kann.
Dann werden die eigentlichen Werte gesendet. Dieser `Send` trägt dann das Tag *ACTUAL*.
Der Master warten für jeden Prozess auf die Nachricht mit dem Tag *INFO* und weiß dann wie viele Werte gleich mit dem Tag *ACTUAL* gesendet werden und an welcher Stelle in `globalStep` diese eingetragen werden müssen.
Am Ende besitzt der Master alle Werte in dem globalen Array und kann diese Bedarfsweise ausgeben oder visualisieren.

Listing 4.5: OpenMPI Werte einsammeln

```
if (id == FIRST)
{

for (int i = 0; i <= right; i++)
{
globalStep[i] = currentStep[i];
}

int startIdx, cnt = 0;

for (int i = 1; i < numberOfProcesses; i++)
{
CHECK(MPI_Recv(buffer, 2, MPI_INT, i, INFO, MPI_COMM_WORLD, &status));

startIdx = buffer[0];
cnt = buffer[1];

CHECK(MPI_Recv(&globalStep[startIdx], cnt, MPI_DOUBLE, i, ACTUAL, MPI_COMM_WORLD, &
    status));
}
```

```
}
else
{

buffer[0] = left;
buffer[1] = nPointsLocal;

CHECK(MPI_Send(buffer, 2, MPI_INT, 0, INFO, MPI_COMM_WORLD));
CHECK(MPI_Send(currentStep, nPointsLocal, MPI_DOUBLE, 0, ACTUAL, MPI_COMM_WORLD));
}
```

4.4 Optionale Ziele

Ein optionales Ziel des Belegs war das Implementieren von zusätzlichen Perturbationen. Im Folgenden sollen zwei ausgewählte davon kurz beschrieben werden.

4.4.1 Festhalten einer Stelle

Als zusätzliche Perturbation wurde die Fähigkeit eingebaut, die Welle an einer Stelle festhalten zu können. Aufgrund der Gleichheit von sequentiellem und *OpenMP* Programm, is dieses in beiden Programmen möglich. Aufgrund der Komplexität von *OpenMPI* wurde dies hier nicht implementiert.

Um ein Festhalten zu ermöglichen, wurde der Funktion, die einen Zeitschritt simuliert, eine zusätzliche Variable `holdflag` übergeben. Über eine einfache Bedingungsabfrage wird das Halteverhalten geregelt. Im Normalfall ist diese 0 und das Programm läuft wie bereits beschreiben. Wenn in der Visualisierung der Nutzer mit der Maus auf die Saite klickt, wird `holdflag = x` gesetzt, weobei `x` der x-Wert des Wellenpunktes an dieser Stelle ist. Dann wird der Wert an dieser Stelle nicht mit der Formel berechnet, sondern einfach kopiert.

Listing 4.6: Wellenpunkt festhalten

```
for (i = 1; i < nPoints - 1; i++)
{
if (holdflag == i)
{
nextStep[holdflag] = currentStep[holdflag];
}
else
{
nextStep[i] = 2.0 * currentStep[i] - previousStep[i] + cSquared * (currentStep[i -
    1] - (2.0 * currentStep[i]) + currentStep[i + 1]);
}
}

// ...
```

4.4.2 Dämpfung der Welle

Über einen zusätzlichen Parameter `LAMBDA` kann ein Dämpfungsfaktor angegeben werden. Dieser sorgt für ein Abschwächen der Wellenauslenkung.
Zu beachten ist hierbei, das die Dämpfung nicht Teil der Logikberechnung in *core* ist. Dort werden wie gehabt die Werte berechnet. In *myWave* wird hingegen wird jeder Wert mit diesem Faktor abgeschwächt. Damit ist das Verändern dieses Parameters eigentlich nur im Fall einer

Visualisierung sinnvoll. Ohne Visualisierung ändert sich der Programmablauf nicht.
Die Abschwächung erfolgt über die e-Funktion (4.2).
Je nachdem, wie die Werte gewählt wurden, erreicht eine gedämpfte Welle relativ schnell die Nulllage, das heißt es ist nur noch eine Linie auf der x-Achse zu sehen. In diesem Fall kann die Welle in den Ausgangszustand zurückgesetzt werden. (siehe Kapitel 4.7)

$$z = e^{-t*\lambda} \tag{4.2}$$

Dabei ist z der Faktor, mit dem jeder Wert multipliziert wird. t ist hier der momentane Zeitschritt in der Simulation. Über den Parameter λ kann die Geschwindigkeit der Abschwächung eingestellt werden.

4.5 Makefile

Das `makefile` wurde aus der Übung übernommen. Für die drei Programme ist es quasi gleich, bis auf unterschiedliche benötigte Compiler-Optionen und Programmnamen. Um die Programme jeweils neu zu bauen, kann der Befehl `make all` verwendet werden.

4.6 Doxyfile

Die Dokumentation des Quelltextes wird über das Kommandozeilen-Tool *doxygen* generiert und in einem neu erzeugtem Ordner abgelegt. Diese Funktionalität wird durch die Datei `doxyfile` ermöglicht, die die Konfiguration für *doxygen* enthält. Diese Datei ist für jedes Programm enthalten. Die Dokumentation liegt dann im HTML Format vor und kann über den Webbrowser angesehen werden.
Die Funktionen sind jeweils in der entsprechende Header-Datei ausführlich kommentiert. Zusätzlich enthalten die `c`-Dateien weitere Anmerkungen und Kommentare an einzelnen Stellen.

4.7 Visualisierung

Für die Visualisierung habe ich mich für *SDL* entschieden. *SDL* bedeutet Simple DirectMedia Layer und ist eine Bibliothek für Visualisierung mit Handling für diverse Events wie Maus, Tastatur und weitere. *SDL* kommt unter anderem in vielen bekannten Spielen zum Einsatz und ist plattformübergreifend verfügbar. Es ist ursprünglich in C implementiert und eignet sich damit gut für die Visualisierung diese Belegs.
Ein weiterer Grund, warum sich für *SDL* entscheiden wurde, ist die relative einfache Umsetzung von Dingen wie Fenstererzeugung und Event-Handling. [6]

Im folgenden wird die Benutzung der Visualisierung kurz erläutert.
Mit *q* und *ESC* wird das Fenster geschlossen und die Simulation beendet.
Mit *a* können die Achsen ein- und ausgeschaltet werden.
Mit *p* kann die Simulation der Welle angehalten und fortgesetzt werden. Dazu ändert sich der Statustext in der rechten unteren Bildschirmecke.
Mit *r* kann die Welle in den originalen Startzustand zurückversetzt werden. Dabei werden die momentanen Werte zunächst mit Nullen überschrieben und dann mit den Sinuswerten beschrieben.
Mit der linken Maustaste kann auf die Welle geklickt und diese damit an dieser Position "festgehalten"werden. Ein erneuter Mausklick lässt die Saite wieder los. Wie in Kapitel 4.4.1 beschrieben, ist dies nicht in der *OpenMPI* Variante verfügbar.

Die Framerate wurde konstant auf 21 festgelegt. Damit sollte ein durchschnittlicher PC in der Lage sein, die Visualisierung flüssig abzuspielen. Die visualisierte Welle ist in den folgeneden Abbildungen zu sehen. Dabei wurden durchgängig die Standardwerte verwendet.

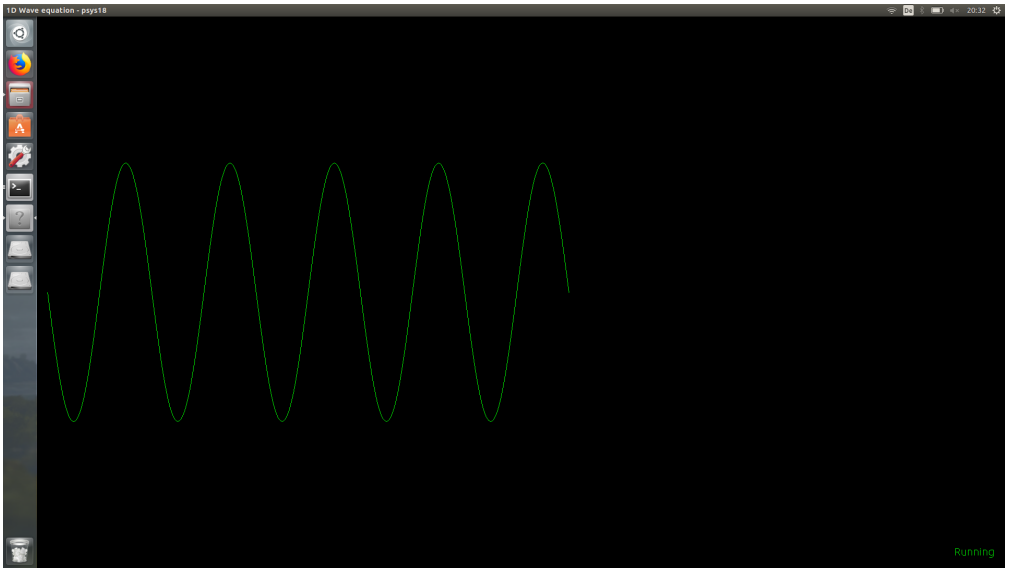

Abbildung 4.7: Visualisierung mit SDL

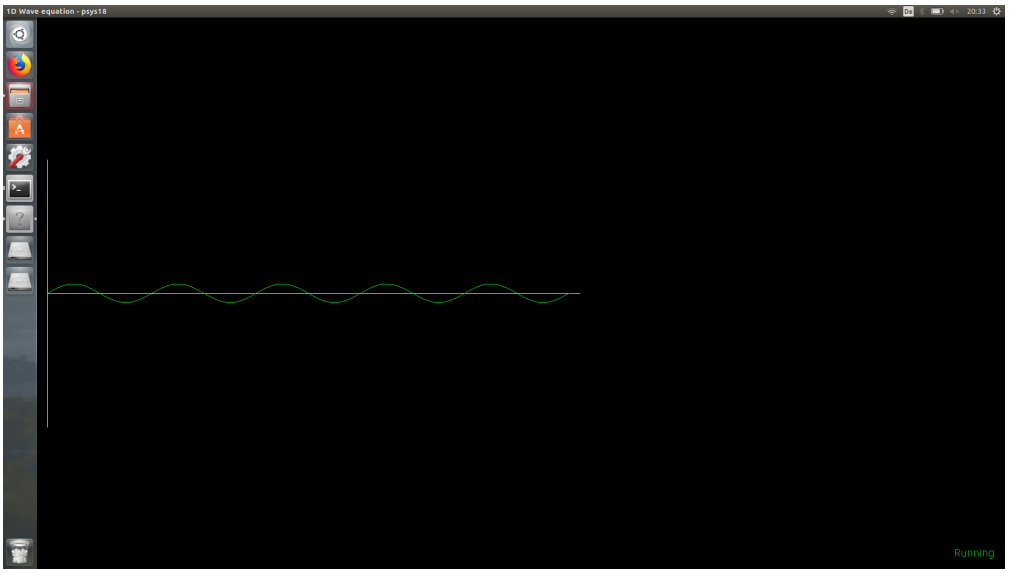

Abbildung 4.8: Achsen eingeschaltet

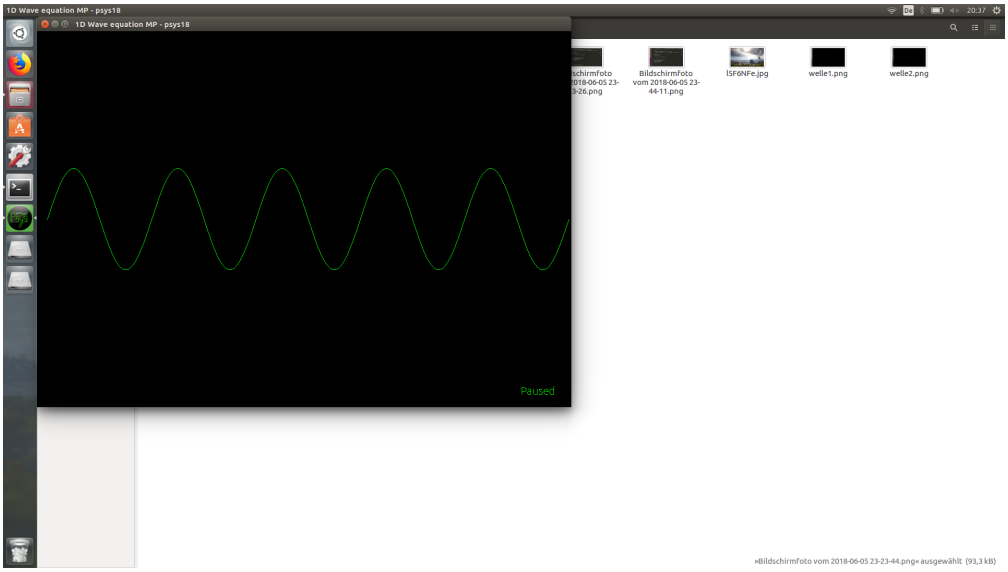

Abbildung 4.9: Simulation pausiert

Kapitel 5

Benchmark

Der Benchmark wurde auf einem Laptop mit 8 GiB RAM und einem Intel I5-7200U Prozessor ausgeführt, das verwendete Betriebssystem war Ubuntu 16.04.

5.1 Tests

Da die Programme keine hohe Komplexität aufweisen, gibt es keine direkten Testdateien. Der Fokus wurde auch nicht darauf gelegt, die Ergebnisse der approximierten Wellengleichung zu bestätigen. Stattdessen wurden die Programme über die Konfigurationsmöglichkeiten von Kommandozeilenargumenten und der Einstellungsdatei verändert und ausgeführt.

Mit `make test` existiert eine Möglichkeit, die Programme mithilfe des Tools *Valgrind* zu überprüfen.

5.2 Durchführung

Die Durchführung des Benchmark läuft über die eigene Funktion, die aufgerufen wird mit dem Kommandozeilenparameter `--benchmark`. Die Zeitschritte und Punkte müssen danach spezifiziert werden. Die Wellenberechnung mit dieser Konfiguration wird dann zehn mal berechnet und jeweils die Ausführungszeit gemessen. Dann werden Mittelwert und die Standardabweichung berechnet. Die Ergebnisse werden in die Datei `benchResults.txt` im Ordner `benchmark` geschrieben. Um alles einmal zu testen wurde ein Script `benchScript.sh` geschrieben. Diese ruft jedes Programm mit unterschiedlich vielen Punkten auf. Die Zeitschritte wurden dabei konstant auf 1000 gelegt. Für *OpenMP* und *OpenMPI* wurden jeweils die Laufzeiten für unterschiedliche Anzahlen von Threads gemessen. Für die Dauer des Benchmarks wird die CPU-Performance auf hoch gesetzt, um die Geschwindigkeit des Prozessor voll ausnutzen zu können. Dazu wird der Befehl `cpufreq -set -g performance` verwendet (benötigt `root`).
Zum Messen der Ausführungszeit wurden verschiedene Funktionen verwendet. Für das sequentielle Programm wurde die `c`-Funktion `gettimeofday()` verwendet. *OpenMP* bietet eine eigene Timer-Funktion `omp_get_wtime()`, die hier verwendet wurde. Bei *OpenMPI* kam die ebenfalls hauseigenen Funktion `MPI_Wtime()` zum Einsatz.

5.3 Ergebnisse

Mit steigender Problemgröße wird der Unterschied zwischen Parallelisierung und Sequentiell immer größer (Abbildung 5.1). Die CPU ist dabei immer langsamer, weil hier eben alles nacheinander gemacht wird.

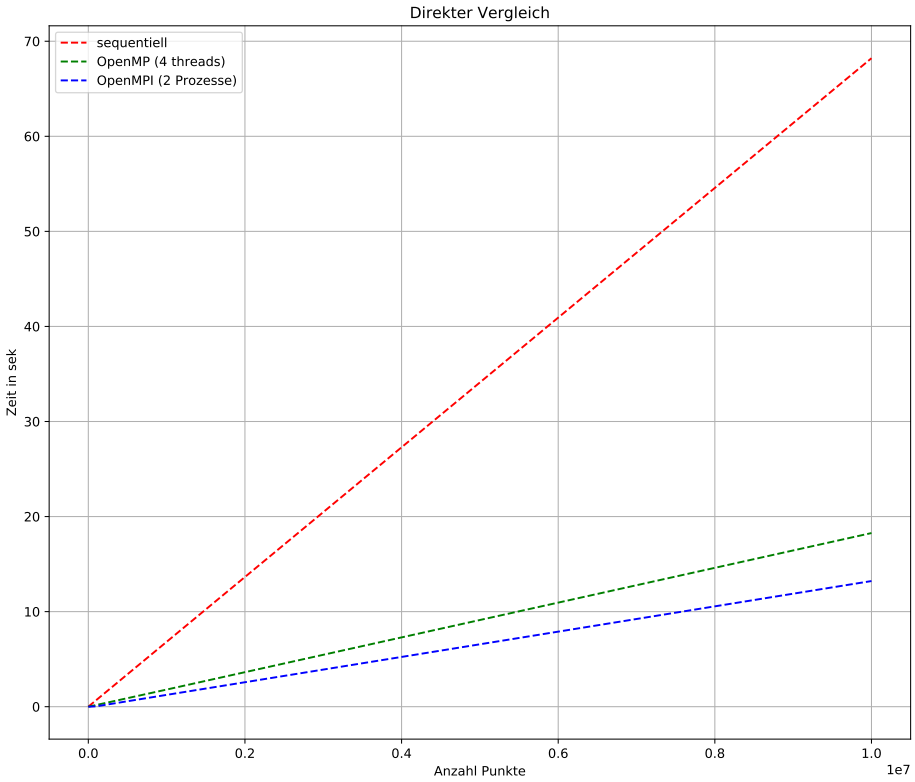

Abbildung 5.1: Ausführungszeiten für alle Programme

OpenMP kann davon profitieren und wird mit steigender Thread-Anzahl immer schneller (Abbildung 5.2). Mit fünf nimmt es dann allerdings wieder ab, weil jetzt einer der Threads immer warten muss.
OpenMPI ist mit zwei Prozessen nochmal schneller als OpenMP. Dieser Vorteile nimmt mit mehreren Prozessen sehr geringfügig ab. Trotzdem ist selbst die langsamste Konfiguration (Fünf Prozesse mit *OpenMPI*) immer noch schneller als die schnellste *OpenMP* Lösung. (Abbildung 5.3). Die vorherige Vermutung, dass eine erhöhte Prozesskommunikation die Geschwindigkeit stark beeinflussen würde, hat sich damit nicht bestätigt.

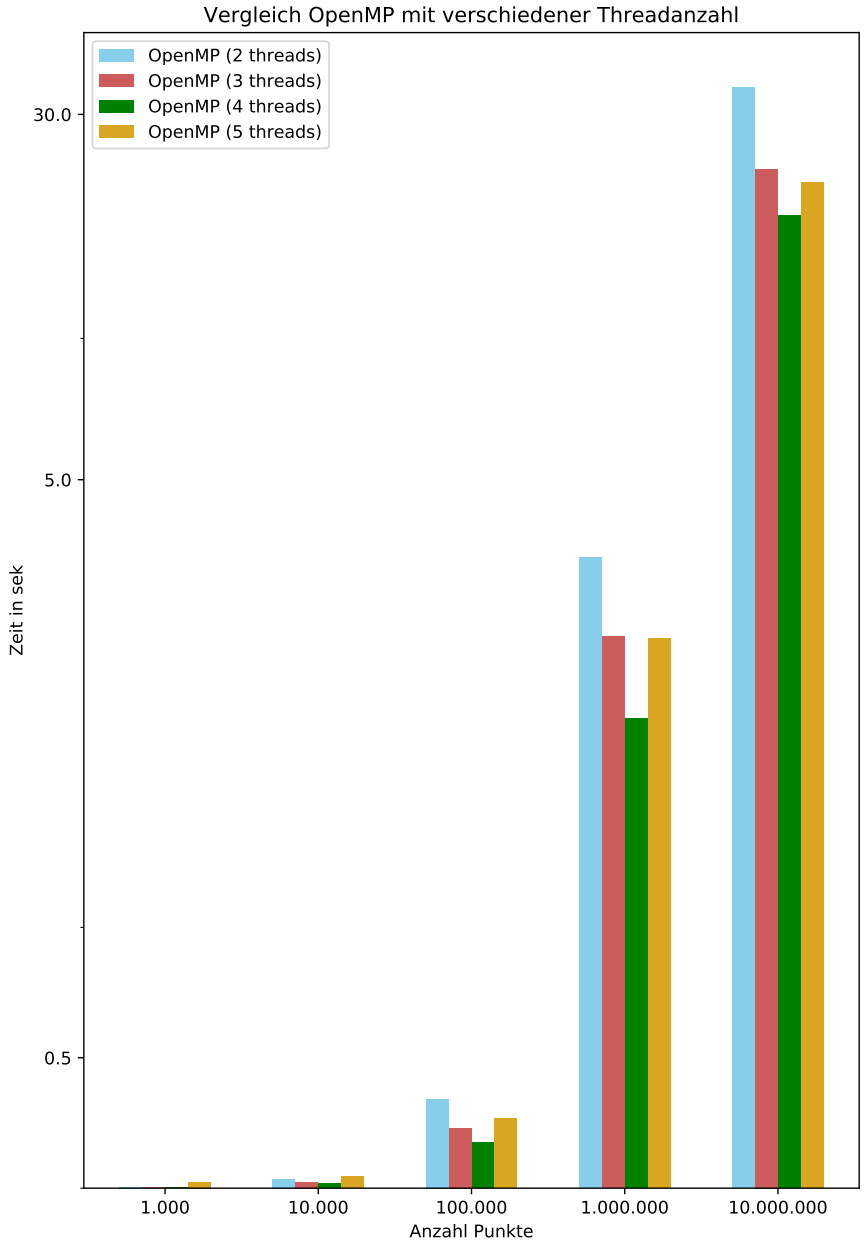

Abbildung 5.2: Ausführungszeiten für MP Programm mit unterschiedlicher Threadanzahl

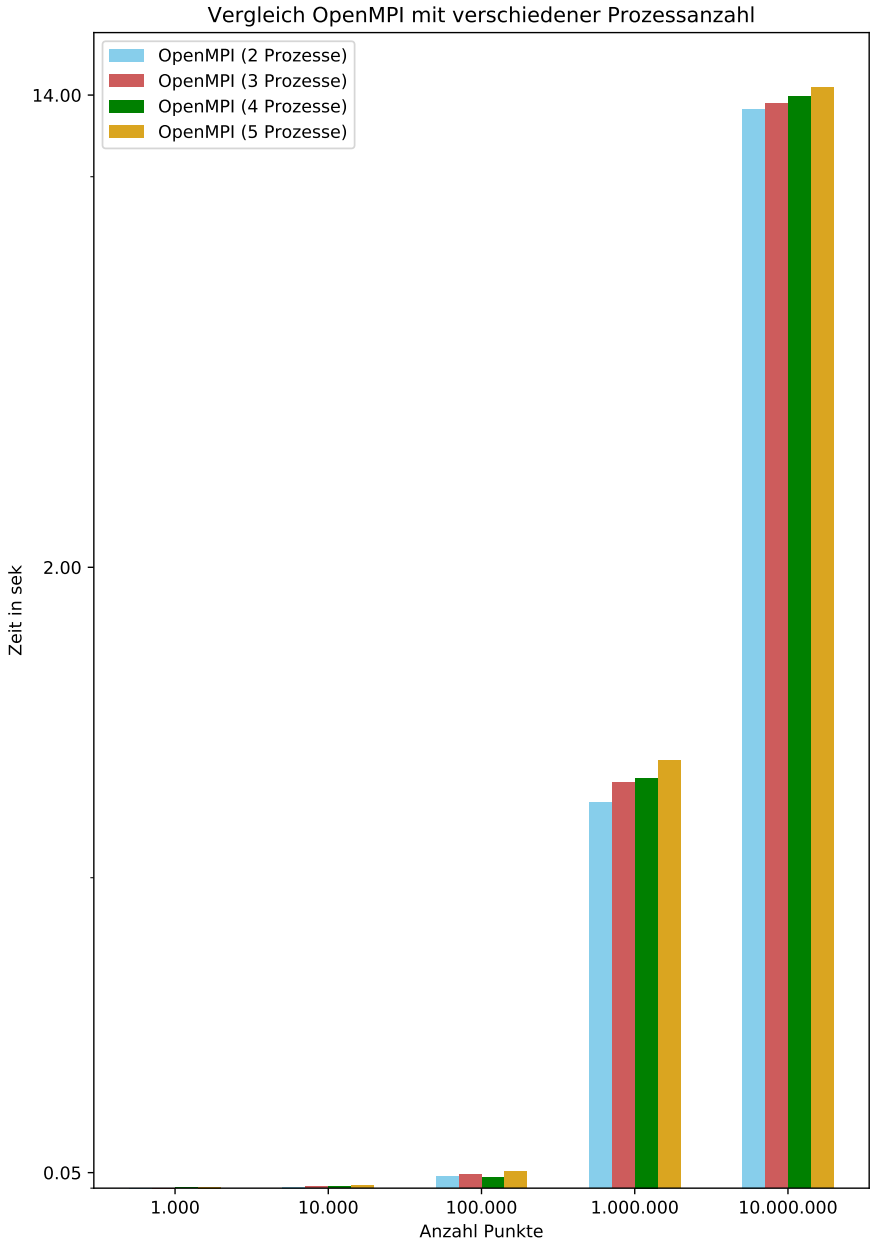

Abbildung 5.3: Ausführungszeiten für MPI Programm mit unterschiedlicher Prozessanzahl

Tabelle 5.1: Ausführungszeiten (in sek) für verschiedene Punktanzahlen für alle Programme

Typ	Anzahl Punkte				
	1000	10000	100000	1000000	10000000
Sequentiell	0.0067	0.0679	0.6820	6.8216	68.2133
OpenMP 2	0.0041	0.0346	0.3423	3.4184	34.2136
OpenMP 3	0.0030	0.0234	0.2305	2.3180	22.9427
OpenMP 4	0.0027	0.0181	0.1741	1.8010	18.2648
OpenMP 5	0.0231	0.0444	0.2683	2.2943	21.5163
OpenMPI 2	0.0005	0.0041	0.0370	1.2423	13.2166
OpenMPI 3	0.0010	0.0049	0.0437	1.3064	13.5145
OpenMPI 4	0.0014	0.0057	0.0336	1.3208	13.9271
OpenMPI 5	0.0039	0.0078	0.0536	1.3770	14.4364

Interessanterweise ist *OpenMP* am schnellsten mit vier Threads, *OpenMPI* dagegen mit zwei Prozessen, was auf den Aufbau meines Prozessors zurückzuführen ist. Dieser besitzt nur zwei echte Kerne, kann aber dank Hyperthreading vier Kerne "simulieren".
Ein Versuch, die Geschwindigkeit von *OpenMPI* weiter zu verbessern, indem man die Schleifen mit *OpenMP* zusätzlich parallelisiert brachte den komplett gegenteiligen Effekt, was sich ebenfalls mit der Kernanzahl erklären lassen kann. *OpenMPI* mit zwei Prozessen und per *OpenMP* parallelisierten Schleifen mit zwei *OpenMP* Threads ist noch quasi genauso schnell wie die reine *OpenMPI* Lösung.
Werden jedoch mehrere *OpenMP* Threads Prozesse verwendet, steigt die Ausführungszeit stark nach oben an und erreicht schnell sogar die der Sequentiellen. Das lässt sich damit erklären, das hier dann mehr Prozesse als Kerne zur Verfügung stehen und dauernd ein Prozess warten muss. Hier geht dann der Vorteil der Parallelisierung verloren.

Tabelle 5.1 zeigt abschließend alle Ausführungszeiten für fünf verschiedene Punktanzahlen für alle Programme. Damit lässt sich der Speedup für die parallelisierte Programme berechnen. Dieser liegt für *OpenMP* mit 4 Threads bei ≈ 3.7 und für *OpenMPI* mit 2 Prozessen bei ≈ 5.1. Diese Werte wurden für die Zeiten bei 10000000 Punkten berechnet.
Dabei ist zu beachten, dass der Speedup bei *OpenMP* für alle Punktanzahlen ungefähr auf dem gleichen Niveau bleibt, während er bei *OpenMPI* gerade für die kleineren Punktanzahlen nochmal deutlich höher ist als für größere (bspw. *OpenMPI* mit zwei Prozessen erreicht bei 100000 Punkten einen Speedup von ≈ 18.4).
Das ist wahrscheinlich darauf zurückzuführen, dass hier die Ausführungszeit so klein ist, das schon kleinste Unterschiede einen großen Einfluss auf den Speedup haben. Damit ist die Berechnung des Speedups erst für längere Zeiten aussagekräftig.

Kapitel 6

Fazit

Zusammenfassend kann gesagt werden, dass die Parallelisierung der Wellensimulation eine deutlichen Zeiteinsparung gebracht hat. Bei der Verwendung der zwei Softwarebibliotheken *OpenMP* und *OpenMPI* konnten jeweils starke Verbesserungen der Laufzeit beobachtet werden. Umso größer die Punkteanzahl ist, umso stärker wird die Notwendigkeit der Parallelisierung deutlich. Damit konnte die initiale Vermutung bestätigt werden.

Die Umsetzung und Implementierung der Parallelisierung gestaltete sich mit *OpenMP* wesentlich einfacher als mit *OpenMPI*. Die Einarbeitungszeit in *OpenMPI* dauert wesentlich länger, da man sich erst auf das Konzept einstellen musste, das der geschriene Code jeweils einzelnen von einem Prozess gesehen werden. In *OpenMP* mussten lediglich ein Pragma konfiguriert werden.

Aufgrund der Unerfahrenheit musste zunächst eine Einarbeitung in die Verwendung von der *SDL* Bibliothek erfolgen. Da diese aber recht benutzerfreundlich gestaltet ist, gab es hier insgesamt betrachtet keine Schwierigkeiten.

Literaturverzeichnis

[1] H. P. Langtangen, “Finite difference methods for wave motion,” pp. 6–11, Nov. 2016.

[2] A. Peirce, “Lecture 8: Solving the heat, laplace and wave equations using finite difference methods,” pp. 6–8, Jan. 2018.

[3] Guide into openmp. [Online]. Available: https://bisqwit.iki.fi/story/howto/openmp/

[4] Man pages for mpich. [Online]. Available: https://www.mpich.org/static/docs/v3.1/

[5] Avoiding mpi deadlock or race conditions. [Online]. Available: https://www.dartmouth.edu/~rc/classes/intro_mpi/mpi_race_conditions.html/

[6] About sdl. [Online]. Available: https://www.libsdl.org/