Semesterprojekt - Parallel Systems numerische implementierung der Wellengleichung

Christian Hugo
Sommersemester 2018
s0562410@htw-berlin.de

HTW Berlin

Einleitung

In folgender Arbeit werden die Ergebnisse vom Semesterprojekt Wellengleichung aus der Parallel Systems Vorlesung im Sommersemester 2018 vorgestellt. Das Ziel war es Amplituden einer Welle, die sich auf einer vibrierenden Seite fortbewegt parallel zu berechnen und zu visualisieren. Die Berechnung wurde sowohl sequentiell, wie auch parallelisiert mit OpenMP, OpenMPI und Threads aus C++ Standard Library implementiert.

1 Wellengleichung

Die Wellengleichung beschreibt vertikale Verschiebungen $u(x,t)$ einer Welle, die sich auf einem Medium (z. B. Luft, Wasser oder auf einer Gitarrensaite) ausbreitet. Es wird davon ausgegangen, dass die Saite zum Startzeitpunkt ($t = 0$) in der Ruheposition liegt (Fig. 1) und im ersten Zeitschritt am linken Ende ($x = 0$) einen Schubs in vertikaler Richtung erhält. In den folgenden Zeitschritten breitet sich die Welle aus (Fig. 2) und bleibt am rechten Ende (x=L) eingespannt. D. h. $u(L,t)$ ist immer 0.

Fig. 1: Welle am Ruhepunkt

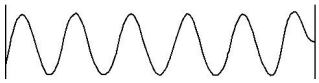


Fig. 2: Ausgebreitet

Die Gleichung ist eine partielle Differenzialgleichung, die die partiellen Ableitungen 2. Grades nach Raum (x) und Zeit (t) benutzt. D. h. sie beschreibt implizit, dass alle $u(x,t)$ Werte mit zweiten partiellen Ableitungen, die zur unten stehenden Formel passen ein Teil der Welle sind. c ist die Ausbreitungsgeschwindigkeit mit der sich die Welle auf dem Medium fortbewegt.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

1.1 initial conditions und boundary conditions

Allein die Wellengleichung beschreibt $u(x,t)$ nicht eindeutig, denn theoretisch gibt es unendlich viele passenden Ableitungen. Für die Eindeutigkeit sind Startbedingungen bei $t = 0$ sowie Randbedingungen links und rechts ($x = 0$, $x = L$) notwendig. Startbedingungen werden initial conditions (kurz IC) genannt, die Randbedingungen heißen boundary conditions (kurz BC). In diesem Fall ist die Welle bei $t = 0$ in der Ruheposition (Fig. 1) und rechts fest eingespannt. D. h. $u(L,t)$ und $u(x,0)$ ergeben immer 0. Die linke Randbedingungen (bei $x = 0$) entsprechen dem Schubs in vertikaler Richtung und wird mit folgen Formel berechnet:

$$u(0,t) = 3sin(2\pi f t)exp(-t)$$

Wobei f die Wellenfrequenz ist.

2 numerische Implementierung

In der analytischen Variante werden x und t Parameter mit unendlicher Genauigkeit erwartet, jedoch ist dies in einer numerischen Implementierung nicht möglich. Aus diesem Grund werden die Ergebnisse mit der finite difference Methode an diskreten x und t Stellen approximiert. Die Abstände Δx und Δt werden eingeführt und der unendlich genaue Definitionsbereich durch sie in endliche Positionen unterteilt. In jedem Δt Schritt wird für jedes Δx die Auslenkung der Welle approximiert (Fig. 3) und das Ergebnis in die Matrix M überführt (Fig. 4).

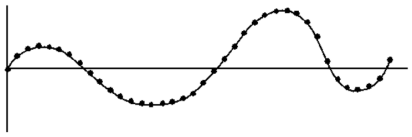


Fig. 3: sampling

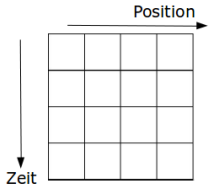


Fig. 4: Matrix M

2.1 Approximierung und Abhängigkeiten

Die numerische Variante berechnet die partiellen Ableitungen mithilfe von Taylor Reihen. Taylor Reihen können für eine Zelle aus M die Ableitungen anhand der nächsten Nachbarn approximieren. Der Nachteil von diesem Verfahren ist, dass sich dadurch Abhängigkeiten zu den benachbarten Zellen ergeben, was vor allem bei der Parallelisierung Problemen bereiten kann. Insgesamt braucht die Approximierung vier Nachbarn aus den letzten zwei Zeilen. Angenommen für $M(t+1,x)$ ist ein neuer Wert zu berechnen, dann hängt die Approximierung ab von:

$$M(t,x), M(t,x-1), M(t,x+1), M(t-1,x)$$

Folgende Abbildungen zeigen die Verteilung grafisch. Für die rote Zelle soll approximiert werden, Fig. 6 zeigt die Abhängigkeiten blau eingefärbt.

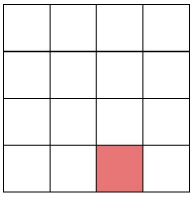


Fig. 5: zu berechnende Zelle

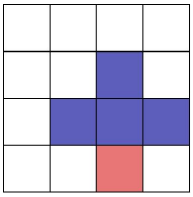


Fig. 6: Abhängigkeiten

Schlussendlich kann folgende Formel einen Wert an Stelle $M(t+1,x)$ anhand der Abhängigkeiten approximieren:

$$... = 2M(t,x) - M(t-1,x) + r(M(t,x-1) - 2M(t,x) + M(t,x+1))$$

Wobei r sich aus der Multiplikation der Ausbreitungsgeschwindigkeit c mit dem Quotienten aus Δt und Δx ergibt:

$$r = (c\frac{\Delta t}{\Delta x})$$

2.2 initial conditions und boundary conditions numerisch

Bevor mit der eigentlichen Berechnung losgelegt werden kann, müssen die IC und BC in die Matrix M eingetragen werden.

0	0	0	0
BC	IC	IC	0
BC			0
BC			0

Fig. 7: IC und BC eingetragen in M

Für $t = 0$ und $x = L$ reicht es Zeile 0 sowie Spalte L mit Nullen aufzufüllen. Die BC der ersten Spalte ($x = 0$) kann mit der Formel aus 1.1 ohne numerische Approximierung berechnet werden. Im ersten Zeitschritt ($t = 1$) werden die IC bei $M(1,x)$ nur anhand der Nachbarn aus Zeile 0 mit folgender Formel approximiert:

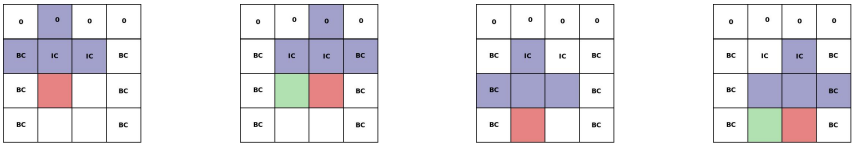
$$M(1,x) = M(0,x) + 0,5r[M(0,x+1) - 2M(0,x) + M(0,x-1)]$$

3 Sequentiell und Parallelisiert

Folgende Abschnitte beschreiben die Implementierung der Wellengleichung. Zuerst werden allgemein Konzepte der sequentiellen und parallelisierten Variante eingeführt und danach allgemeine Strukturen vom Programm mit Codeausschnitten gezeigt. Abschließend folgt für jede Ausführungsstrategie (Sequentiell, OpenMP, std::threads sowie OpenMPI) ein separater Abschnitt.

3.1 Sequentiell

Die sequentielle Variante iteriert in einer geschachtelten Schleife über alle Zeitschritte Δt und für jeden Zeitschritt über alle Δx. In den folgenden Abbildungen ist zu sehen, dass bei jeder Iteration in allen abhängigen Zellen Werte eingetragen wurden und die Abhängigkeiten somit erfüllt sind. D. h. in der sequentiellen Implementierung müssen sie nicht gesondert behandelt werden.



3.2 Parallelisiert

Bei der parallelen Ausführung wird pro Δt Schritt die Zeile partitioniert und auf die Threads verteilt. Fig. 8 zeigt die Partitionen bei 4 Spalten und 4 Threads. Jedoch besteht jetzt die Gefahr, dass ein Thread auf die nächste Zeile springt, bevor die anderen ihren Bereich abgearbeitet haben. In Fig. 9 ist zu sehen, dass wenn Thread 2 versucht in der nächsten Zeile zu approximieren, die abhängigen Zellen von Thread 1 und 2 noch nicht beschrieben sein müssen. D. h. beim Schreiben von $M(t+1,x)$ treten race conditions an den Abhängigkeiten $M(t,x+1), M(t,x-1)$ auf.

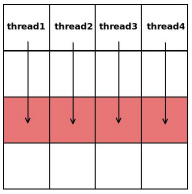


Fig. 8: 4 Threads pro Zeile

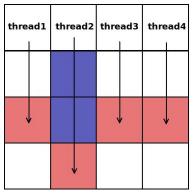


Fig. 9: race conditions

Eine barrier Aufruf am Zeilenende kann das Problem lösen. Eine barrier Synchronisierung versetzt Thread 2 in den WAIT Status bis alle Threads mit ihren Partitionen fertig sind und weckt ihn dann wieder auf.

4 allgemeine Strukturen

Die Ausführungsstrategien sind durch das enum STRATEGY_TYPE unterscheidbar:

```
enum class STRATEGY_TYPE
{
    START = 0,
    SEQUENTIAL = START,
    OPEN_MP,
    STD_THREAD,
    END
};
```

Fig. 10: STRATEGY_TYPE identifiziert eine implementierte Ausführungsstrategie

Damit Implementierungen hinter einer Schnittstelle austauschbar sind, wurde das Strategy Pattern umgesetzt. Jede Ausführungsstrategie leitet von der Schnittstelle StrategyBase ab und muss den abstrakten () Operator überschreiben.

```
class StrategyBase {
public:
    StrategyBase( const WaveProcessingParams & );
    virtual ~StrategyBase() = default;

    virtual double * operator()() = 0;
```

Fig. 11: Basisklasse für jede Ausführungsstrategie

Der () Operator wird aufgerufen, um eine Strategie zu starten und liefert die Matrix M als double * array zurück. Im Datentyp WaveProcessingParams sind alle für die Berechnung notwendigen Parameter zusammenfasst.

```
struct WaveProcessingParams
{
    double wave_velocity = 6;
    double frequency = 3;
    double length = 10.0;
    double time_duration = 10.0;
    int nt = 1000;
    int nx = 400;
    double dx;
    double dt;
    double r;
```

Fig. 12: struct um alle Parameter für die Berechnung zu kapseln

Die Standardwerte können mit einer Konfigurationsdatei oder Kommandozeilenparametern überschrieben werden (siehe Abschnitt Aufruf und Konfiguration).

Die Klasse StrateyContext startet eine Strategie über den () Operator und bietet nach außen die Methode executeStrategy() an. Ein Aufrufer muss also nicht konkrete Strategien kennen, sondern gibt einen enum Wert, plus einem WaveProcessingParams Objekt herein.

```
class StrategyContext
{
public:
    StrategyContext() = default;

    double * executeStrategy( STRATEGY_TYPE, const WaveProcessingParams & );
```

Fig. 13: StrateyContext kennt konkrete Implementierungen und hat executeStrategy()

Die Formeln zum approximieren aus 2.1. sowie zum Berechnen der IC und BC aus 2.2. sind in der Klasse WaveProcessing implementiert. WaveProcessing wird mit einem WaveProcessingParams Objekt initialisiert, woraufhin process() für das approximieren und doICAndBC() zum setzen der IC und BC aufgerufen werden kann.

```
class WavePocessing
{
public:
    WavePocessing( const WaveProcessingParams & );
    ~WavePocessing() = default;
    double process( double, double, double, double );
```

Fig. 14: WaveProcessing implementiert die Formeln für das Approximieren

Die Methode process() nimmt die Abhängigkeiten als separate double's entgegen und gibt das approximierte Ergebnis zurück. Darüber hinaus existieren weitere Methoden, die bspw. (t,x) Koordinaten sowie die Matrix M entgegennehmen, die Abhängigkeiten selbst auslesen und zu process() delegieren.

```
double WavePocessing::process( double M_t_xMinus1, double M_t_x,
                               double M_t_xPlus1, double M_tMinus1_x )
{
    return 2 * M_t_x - M_tMinus1_x + _processParams.r *
        ( M_t_xPlus1 - 2 * M_t_x + M_t_xMinus1 );
}
```

Fig. 15: zentrale Implementierung der Formel aus 2.1.

5 Sequentielle Implementierung

Die sequentielle Berechnung der Wellengleichung ist in der Klasse SequentialStrategy implementiert. Sie leitet von StrategyBase ab und überschreibt den () Operator.

```
class SequentialStrategy : public StrategyBase
{
public:
    SequentialStrategy( const WaveProcessingParams & );

    virtual double * operator()() override;
```

Fig. 16: Subklasse von StrategyBase zum sequentiellen berechnen

In der Implementierung vom () Operator wird zuerst der Speicher vorbereitet, die IC und BC gesetzt und danach zeilenweise für jede Position process() mit (t,x) Koordinaten aufgerufen.

```
double * SequentialStrategy::operator()()
{
    // Speicher vorbereiten
    double *waveData = new double[ _params.nt * _params.nx ];
    fill( waveData, waveData + _params.nt * _params.nx, 0.0 );

    _proc.doICAndBC( waveData );

    // pro Zeitschritt über alle Positionen
    for ( int i = 2; i < _params.nt; i++ )
        for ( int j = 1; j < _params.nx - 1; j++ )
            _proc.process( i, j, waveData );
    return waveData;
}
```

Fig. 17: Implementierung vom () Operator

Der Speicher der 2 dimensionalen Matrix M liegt im flachen double * array waveData. Das hat den Vorteil, dass für eine (t,x) Koordinate nur einmal in den Hauptspeicher gegriffen werden muss. Das Auflösen von (t,x) Koordinaten zu flachen waveData Koordinaten erledigt WaveProcessing intern. Die Funktion fill aus der C++ Standard Bibliothek initialisiert den Speicher mit nullen vor.

Das setzten der IC and und BC läuft bei allen Strategien identisch ab. Der Einfachheit halber wurde doICAndBC() nicht parallelisiert.

6 OpenMP Implementierung

OpenMP (Open Multi Processing) ist ein Standard um parallele Programme in den Sprachen C/C++ oder Fortran zu entwickeln. Er besteht u. a. aus einer Ansammlung aus precompiler Direktiven die direkt über die Zeile geschrieben werden, die parallelisiert werden soll. Bspw. sorgt die direktive

```
#pragma omp_parallel
{
  ...
}
```

dafür, dass der Block innerhalb der geschweiften Klammern von mehreren Threads gleichzeitig betreten wird. Die direktive

```
#pragma omp_parallel for
for ( int i = 0; i < 100; i++ )
{ ... }
```

hingegen kann über einer for Schleife stehen und partitioniert die Iterationen auf mehrere Threads. Die Anzahl der Threads kann bspw. mit

```
#pragma omp_parallel for numthreads( 2 )
for ( ... )
```

angegeben werden.

Der Code für die Parallelisierung wird zur compile-Zeit vom pre Compiler generiert. Das hat den Vorteil, dass die technische Umsetzung der Parallelisierung dem Entwickler verborgen bleibt und ein sequentielles Programm leicht um parallele Abschnitte erweitert werden kann. Die Direktiven sind kein integraler Bestandteil des Programms, d.h. ignoriert der Precomiler sie oder löscht der Entwickler sie, ensteht ein sequentielles Programm, dass korrekt abläuft.

OpenMP Parallelisiert nach dem fork join Modell, d.h. zum Programmstart existiert nur ein Thread (auch Master oder initialer Thread genannt). Wird ein paralleler Block erreicht, teilt er sich in mehrere Threads auf und wird am Blockende mit einem impliziten barrier zum initialen Thread zusammengeführt.

6.1 Implementierung

Wie auch bei der sequentiellen Strategie gibt es eine Klasse (OpenMPStrategy), die von StrategyBase ableitet und den () Operator überschreibt.

```
class OpenMPStrategy : public StrategyBase
{
public:
    OpenMPStrategy( const WaveProcessingParams & );

    virtual double * operator()() override;
```

Fig. 18: Subklasse von StrategyBase zum parallelisierten berechnen

Auch die Implementierung entspricht größtenteils der sequentiellen Variante, jedoch erweitert um OpenMP Direktiven. Die Äußere Direktive eröffnet einen Block, der von mehreren Threads betreten wird, die innere Direktive steuert die Portionierung der for Schleife über alle Δx Positionen.

```
double * OpenMPStrategy::operator()()
{
    double *waveData = new double[ _params.nt * _params.nx ];
    fill( waveData, waveData + _params.nt * _params.nx, 0.0 );

    _proc.doICAndBC( waveData );

    #pragma omp parallel num_threads( getNumThreads() )
    for ( int i = 2; i < _params.nt; i++ )
    {
        #pragma omp for schedule( static )
        for ( int j = 1; j < _params.nx - 1; j++ )
            _proc.process( i, j, waveData );
    }
    return waveData;
}
```

Fig. 19: Implementierung vom () Operator parallelisiert mit OpenMP

Die Anzahl der Threads stammt aus dem WaveProcessingParams Objekt und ist somit konfigurierbar. Die Anweisung schedule(static) sorgt dafür, dass die Iterationen auf möglichst gleichgroße Partitionen verteilt werden. Angenommen zwei Threads betreten die Schleife gleichzeitig und es existieren 100 Δx Schritte, dann hat Thread 1 Partitionsgrenzen bei 0 bis 49 und Thread 2 bei 50 bis 99.

7 std Threads Implementierung

Die C++11 Spezifikation hat für die C++ Standard Library eine Multi-threading library eingeführt. Die Klasse thread startet paralele Stränge, die Header mutex und conditionvariable definieren Strukturen für die Synchronisierung. Anders als bei OpenMP sind die Funktionen ein integraler Bestandteil vom Programm und müssen explizit aufgerufen werden. Darüber hinaus existiert kein Konstrukt, dass Iterationen einer for Schleife partitioniert oder eine barrier Synchronisierung erzwingt.

7.1 Implementierung

Erneut existiert eine Klasse (StdThreadsStrategy), die von StrategyBase ableitet und den () Operator überschreibt. Der StdThreadsStrategy Header unterscheidet sich kaum von den anderen und wird deshalb nicht gezeigt.
Das explizite Steuern der Parallelisierung sorgt dafür, dass nach doICAndBC() die Implementierung vom stark von dev vorherigen Strategien abweicht.

```
double * StdThreadsStrategy::operator()()
{
    // Speicher vorbereiten
    double *waveData = new double[ _params.nt * _params.nx ];
    fill( waveData, waveData + _params.nt * _params.nx, 0.0 );

    _proc.doICAndBC( waveData );

    int intervall = _params.nx / _params.numThreads;
    vector < thread > threadList;
    int numThreads = getNumThreads();
    for ( int i = 0; i < numThreads; i++ )
    {
        // partitionsgrenzen berechnen
        int xStart = i > 0 ? i * intervall + 1 : 1;
        int xEnd = i < ( numThreads - 1 ) ?
            xStart + intervall : _params.nx - 1;
        // thread in processPartial() starten
        threadList.push_back( thread(
            &StdThreadsStrategy::processPartial, this,
            xStart, xEnd, waveData ) );
    }
    for ( auto &thread : threadList )
        thread.join();
    return waveData;
}
```

Fig. 20: Implementierung vom () Operator der Klasse StdThreadStrategy

Für jeden Thread müssen die Partitionsgrenzen händisch berechnet werden, woraufhin der Thread mit der Einstiegsfunktion processPartial() startet. Der this Parameter teilt

dem thread Konstruktor mit, dass processPartial() eine Methode vom aktuellen Objekt ist, alle weiteren Argumente werden direkt an processPartial() weitergegeben.

```
void StdThreadsStrategy::processPartial( int xStart, int xEnd, double *waveData )
{
    for ( int i = 2; i < _params.nt; i++ )
    {
        for ( int j = xStart; j < xEnd; j++ )
            _proc.process( i, j, waveData );
        _barrier.wait();
    }
}
```

Fig. 21: Einstiegsfunktion von einem std Thread

In der Einstiegsfunktion processPartial() läuft der Thread über seine Partitionsgrenzen, ruft eine der process() Methoden aus WaveProcessing auf und Synchronisiert sich, durch ein explizites barrier.

7.2 barrier

Um die Threads zeilenweise zu synchronisieren wurde die barrier Klasse (StdThreadBarrier) geschrieben. Im C++14 Standard hat die Standard Library keine barrier Implementierung aber alle Mittel für ein eigenes barrier.

```
void StdThreadBarrier::wait()
{
    unique_lock< mutex > lock( _mutex );
    int curBarrierGroup = _barrierGroup;
    // von allen threads aufgerufen ?
    if ( ++_callCount == _threadCount )
    {
        _callCount = 0;
        _barrierGroup++;
        // Versuch alle aufzuwecken
        _condVar.notify_all();
    }
    else
    {
        _condVar.wait( lock, [ this, curBarrierGroup ]
        {
            // threads nicht in "curBarrierGroup" warten
            return curBarrierGroup == _barrierGroup - 1;
        });
    }
}
```

Fig. 22: wait() Methode für eine barrier synchroniserung von std Threads

Beim Betreten von wait() sorgt uniquelock dafür, dass nur ein Thread gleichzeitig aktiv sein kann. Die Variable umwickelt den mutex Member und erzeugt einen lock, der entweder im Destruktor (beim Verlassen von wait()) oder im Übergang zum Status WAIT (Aufruf condVar.wait()) freigegeben wird. Wurde wait() von allen Threads aufgerufen, versucht notifyAll() alle wartenden Threads aufzuwecken. Sobald ein Thread den WAIT Status verlässt, wird sein lock wieder aktiv, bis er wait() verlässt.

8 OpenMPI Implementierung

OpenMPI ist eine Implementierung vom Message passing Interface Protokoll (MPI) für C/C++ und Fortran. MPI definiert u. a. Routinen um Speicher auf Prozesse zu verteilen und wieder zusammenzuführen. Die Kommunikation ist nachrichtenbasiert und kann über den internen Speicherbus oder das Netzwerk laufen. Hierdurch ist MPI gut für den Einsatz in Clustern oder Hochgeschwindigkeitsrechnern geeignet. OpenMPI hat Konzepte der Projekte LAM/MPI, LA-MPI sowie FT-MPI übernommen und zusammengeführt, möchte aber nicht nur als merge dieser Projekte gelten, sondern eine auf die Produktion ausgerichtete MPI Implementierung schaffen.

8.1 Implementierung

Es hat sich gezeigt, dass OpenMPI sich schlecht in das Strategie-Muster integriert. Deswegen liegt die Implementierung in einem eigenen Binary, nutzt aber die Klasse WaveProcessing und den Parser für die Konfigurationsdatei. Das Verarbeiten der Matrix M besteht im groben aus vier Schritten, die sich zeilenweise wiederholen:
1. Speicherpartitionen auf Prozesse verteilen, 2. Wellengleichung anwenden, 3. Speicher zusammenführen und 4. explizite barrier Synchronisierung.

```
for ( int i = 2; i < params.nt; i++ )
{
    // 1. Speicher aufteilen
    int previousRowOffset = ( i - 1 ) * params.nx;
    sendScatterPrevRow( waveData, prevRow,
                        previousRowOffset, elements_per_process );
    // 2. Wellengleichung anwenden
    calcCurPartition( curRow, prevRow, prevPrevRow,
                      proc, elemtsWithDeps );
    // 3. Partitionen im Master zusammenführen
    int currentRowOffset = i * params.nx;
    gather( waveData, curRow,
            elements_per_process, currentRowOffset );
    // im nächsten Durchlauf wird prev zu prev-prev
    // der Speicher von prevRow wird neu befüllt
    swap( prevPrevRow, prevRow );
    // 4. Synchronisierung
    MPI_Barrier( MPI_COMM_WORLD );
}
```

Fig. 23: Schleife über alle Zeilen der Matrix M

Eine für MPI typische Vorgehensweise wäre es in Schritt 1 den Speicher mit:

MPI_Scatter (...)

an die Subprozesse zu verteilen und ihn in Schritt 3 im Master mit:

MPI_Gather (...)

zusammenzuführen. Jedoch sind die Abhängigkeiten überlappend und nicht adjazent:

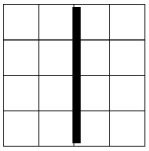


Fig. 24: Partitionen

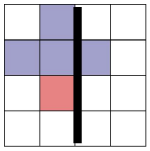


Fig. 25: Prozess 1

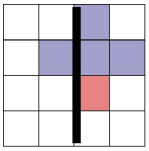


Fig. 26: Prozess 2

Oben braucht Prozess 1 eine Zelle aus der Partition von Prozess 2 und umgekehrt. MPIScatter() kann nur sequentiell unterteilen ohne Überlappungen. Deshalb verschickt der Master die Abhängigkeiten mit MPI_Send() und die Subprozesse empfangen sie mit MPI_Recv():

```
// über alle Prozesse bis auf den Master
for ( int dest = 1; dest < comm_size; dest++ )
{
    // Partition versenden
    int partitionStart = prevRowOffset + elementsPerProcess * dest;
    int errRet = MPI_Send( waveData + partitionStart, elemtsWithDeps,
                           MPI_DOUBLE, dest, 0, MPI_COMM_WORLD );
    if ( errRet != MPI_SUCCESS )
    {
```

Fig. 27: Auszug aus sendScatterPrevRow() für den Masterprozess

```
// Partition empfangen und nach prevRow schreiben
MPI_Status status;
int errRet = MPI_Recv( prevRow, elemtsWithDeps,
                       MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status );
if ( errRet != MPI_SUCCESS )
{
```

Fig. 28: Auszug aus sendScatterPrevRow() für die Subprozesse

Die Partitionen, die in Schritt 2 beschrieben wurden überlappen nicht und könnten theoretisch mit MPI_Scatter() aufgeteilt werden. In diese Zellen wurden jedoch vollkommen neue Werte eingetragen, die alleine von den vorherigen Zeilen abhängen. Deshalb reicht es, wenn die Subprozesse den Speicher selbst allokieren und mit nullen vorinitialisieren.

In Schritt 3 führt MPI_Gather() die Partitionen zusammen. Erneut unterstützt MPI_Gather() nur sequentielle Bereiche, jedoch ist dies hier kein Problem, denn die Partitionen, in die geschrieben wird, sind adjazent.

```
int errRet = MPI_Gather( curRow + 1, elements_per_process, MPI_DOUBLE,
                         waveData + currentRowOffset + 1,
                         elements_per_process, MPI_DOUBLE,
                         0, MPI_COMM_WORLD );
if ( errRet != MPI_SUCCESS )
{…
}
```

Fig. 29: Auszug aus gather()

Bevor sich alles in der nächsten Zeile wiederholt, folgt die Synchronisierung mit

```
MPI_Barrier()
```

9 Aufruf und Konfiguration

Das Programm ist über Parameter konfigurierbar, die entweder aus einer Konfigurationsdatei oder den Kommandozeilenparametern gelesen werden. Erscheint in den Kommandozeilenparametern ein

```
cfg_file=[string]
```

Parameter, mit dem Pfad zur Konfigurationsdatei, dann werden die Werte aus der Datei gelesen und alle weiteren Kommandozeilenparameter ignoriert. Erscheint keins von beiden oder wurde ein Parameter ausgelassen, werden Standardwerte benutzt.
Die Implementierung unterstützt keine Leerzeichen.
Folgende Parameter stehen zur Verfügung:

Name	Wirkung	Standardwert
executionstrategy	spezifiziert die Ausführungssrategie: Die Werte entsprechen dem enum STRATEGY_TYPE	0
wavevelocity	gibt die Ausbreitungsgeschwindigkeit c an	6
frequency	steuert die Frequenz der Welle	3
length	spezifiziert die Länge der Saite	10
timeduration	spezifiziert die Zeitdauer der Simulation	10
nt	steuer die Anzahl der Δt Schritte	1000
nx	steuert die Anzahl der Δx Schritte	400

Die Klasse CfgParser parst die Parameter. Sie erwartet einen String, in dem entweder der Inhalt der Datei oder eine Kombination der Kommandozeilenparameter steht und ist somit unabhängig von der Parameterquelle. Für jeden Parameter hat sie einen regulären Ausdruck, für das Format:

ParameterName=ParameterWert

Bspw. hat der Ausdruck für executionstrategy folgenden Aufbau:

```
std::regex execStratRegex = std::regex( "execution_strategy=(.*)",
                                        std::regex_constants::icase );
```

Der Einfachheit halber matcht der Ausdruck alle Zeichenfolgen, die auf execution-strategy= folgen. Greift der Ausdruck, ist es möglich auf die Zeichenfolge $.*$ zwischen den Klammern zuzugreifen.
Schlussendlich liefert der Parser den Wert in einem boost::optional zurück:

```
boost::optional< STRATEGY_TYPE > getStrategyType() const;
```

Dies hat den Vorteil, dass falls ein Parameter nicht gesetzt ist, boost::None zurückgegeben werden kann und der Aufrufer entscheidet auf welchen Wert execution-strategy zu setzen ist. In diesem Fall ist der default=0 (Sequentiell).

10 grafische Benutzeroberfläche

Die Visualiserung wurde mit OpenGL und dem OpenGL Utility Toolkit (glut) geschrieben. OpenGL hat den Vorteil, dass auch wenn mit großen Daten simuliert wird, die Welle

problemlos gezeichnet werden kann. Initalisiert wird die GUI, nachdem die Matrix M berechnet wurde, durch die Funktion initGUI() aus dem Namensraum open_gl_gui_n.

Links ist die Welle in der Ruheposition zu sehen, rechts wie sie sich zum Rand ($x = L$) ausgebreitet hat.

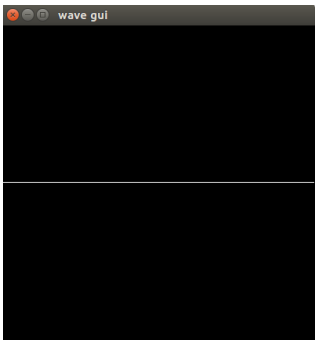


Fig. 30: Welle am Ruhepunkt

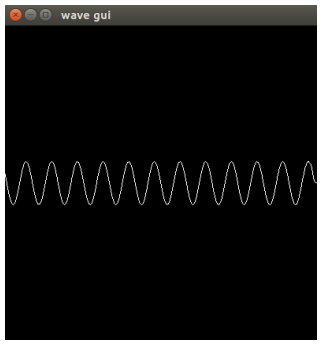


Fig. 31: Ausgebreitet

Nachdem die Welle sich zum rechten Rand ausgebreitet hat, bewegt sie sich wieder in Richtung Start ($x = 0$) und trifft auf die entgegenkommenden Vibrationen.

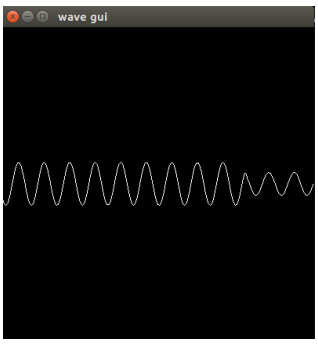


Fig. 32: wieder von rechts nach links

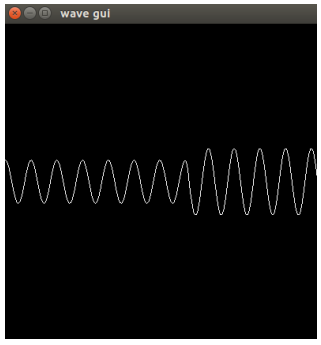


Fig. 33: Welle trifft auf Vibrationen

Am Start angekommen, prallt sie erneut am linken Ende $x = 0$ ab und breitet sich wieder in Richtung $x = L$ aus:

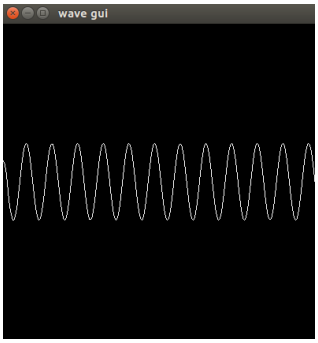


Fig. 34: Welle wieder links angekommen

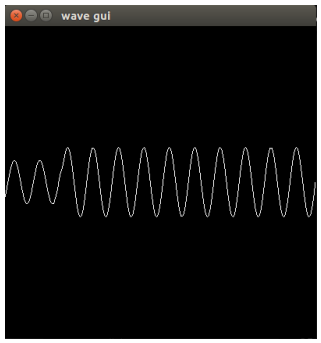


Fig. 35: wieder von links nach rechts

11 Unit Tests

Die Unit Tests wurden mit dem Boost Unit Test Framework geschrieben. Sie stellen sicher, dass die parallelisierten Varianten und das Parsen der Konfiguration funktionieren. Ein Testfall kann mit

```
BOOST_AUTO_TEST_CASE( test_name )
{
    BOOST_CHECK( 2 > 1 );
}
```

definiert werden.

Der gesamte Test ist hierarchisch in die test suites parallelization suite und cfg parser suite unterteilt. Eine test suite kann durch

```
BOOST_AUTO_TEST_SUITE( suite_name )
```

eingeleitet und mit

```
BOOST_AUTO_TEST_SUITE_END()
```

beendet werden.

Um zu prüfen, dass die Matrix M korrekt ist, wird zuerst die sequentielle Variante ausgeführt und das Ergebnis mit dem einer Parallelisierten Implementierung verglichen. D.h. es wird davon ausgegangen, dass die sequentielle Variante korrekt arbeitet. Das Parsen der Konfiguration wird getestet, indem bspw.

```
"wave_velocity=1\n"
"frequency=2\n"
"length=3\n"
"time_duration=4\n"
"nt=5\n"
"nx=6\n"
```

an den Parser übergeben wird und die Werte im WaveProcessingParams Objekt geprüft werden.

12 Projektstruktur

Auf der obersten Ebene verteilt sich der Sourcecode auf die Ordner include und src (Fig. 36). Im Unterordner cfg liegt der Konfigurationsparser sowie eine util, um ihn leichter anzusprechen. Im Verzeichnis gui liegt die OpenGL gui. Das Verzeichnis parallelization hat für jede implementierte Strategie einen eigenen Unterordner [1]. Im Unterordner strategy liegen Basistrukturen für das Strategie Muster. Der Ordner processing beherbergt die Typen WaveProcessing und WaveProcessingParams.

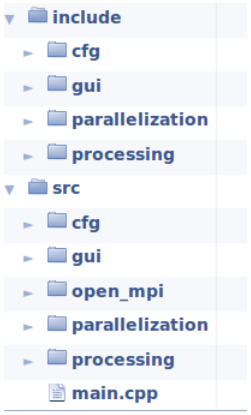


Fig. 36: Projektstruktur auf der obersten Ebene

Fig. 37: include Ordner

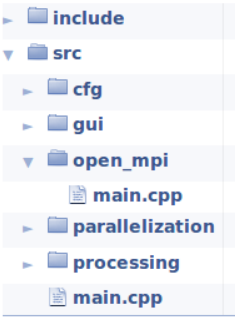


Fig. 38: src Ordner

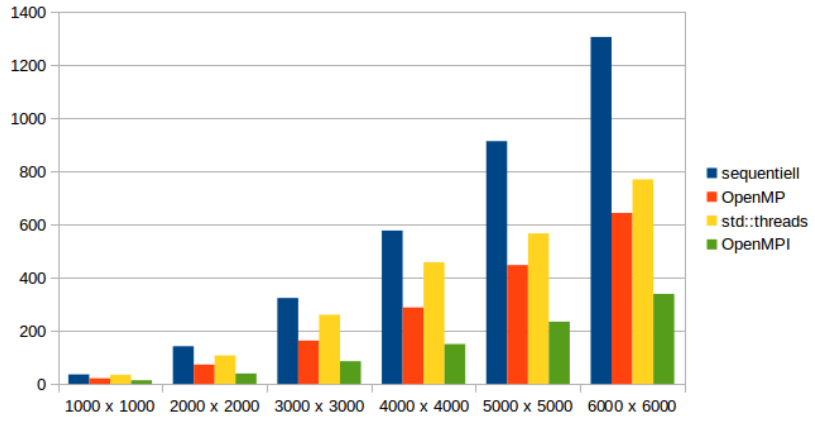
13 Laufzeiten und Speedup

Die Laufzeiten wurden mit 2, 4 und 8 Threads gemessen. Das Programm wurde auf einem Ubuntu 16.04 Betriebssystem, mit insgesamt 4 Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz Prozessoren ausgeführt. Die Zeiten wurden automatisiert durch das Skript bench.sh gemessen und aus je vier läufen der Mittelwert gebildet. Der Einfachheit halber wird immer die komplette Größe der Matrix M angegeben, die Messung wurde nur für den parallelisierten Anteil (also nicht für die IC und BC) gemacht.

13.1 2 Threads

Folgendes Balkendiagramm zeigt die Laufzeiten bei einer Parallelisierung mit 2 Threads. Die X Achse zeigt die Matrix-Größe (nt, xt), die Y Achse die Zeit in Millisekunden.

[1] OpenMPI hat nur den Unterodner src/parallelization mit einer main.cpp.

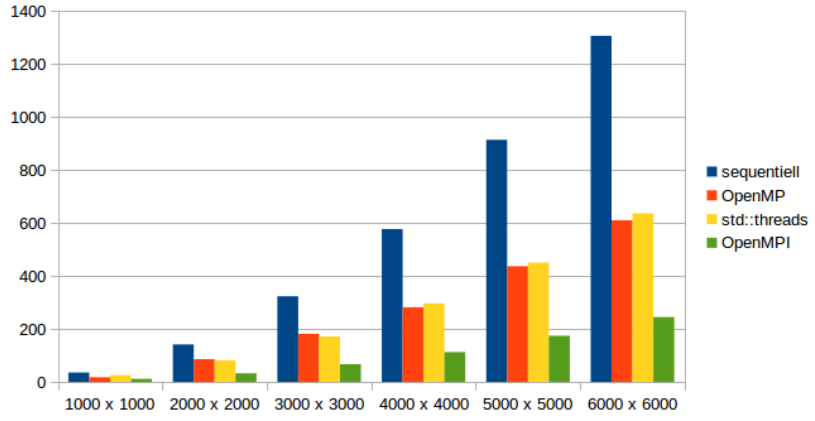


In folgender Tabelle steht der globale Speedup für die Laufzeiten:

	1000 x 1000	**2000 x 2000**	**3000 x 3000**	**4000 x 4000**	**5000 x 5000**	**6000 x 6000**
Speedup OpenMP	1.69	1,94	1,98	2	2,04	2.03
Speedup std	1.04	1.32	1.24	1.26	1.61	1.7
Speedup OpenMPI	2.56	3.61	3.78	3.85	3.9	3.86

13.2 4 Threads

Bei einer Parallelisierung mit 4 Threads ergeben sich folgende Laufzeiten:

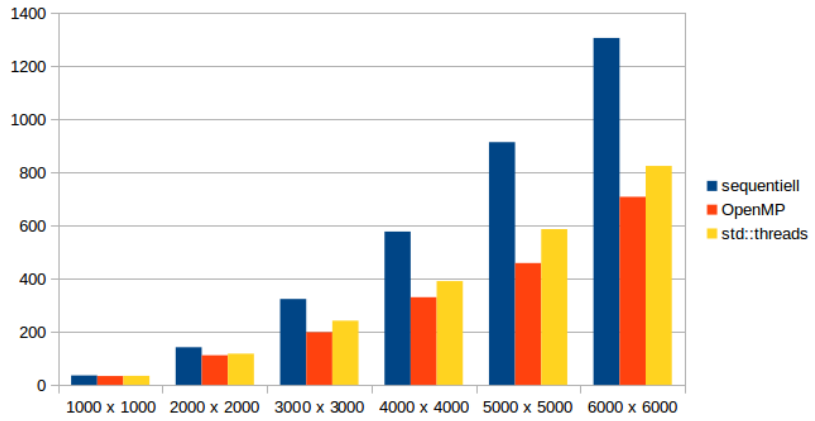


Folgende Tabelle zeigt den globalen Speedup bei 4 Threads:

	1000 x 1000	2000 x 2000	3000 x 3000	4000 x 4000	5000 x 5000	6000 x 6000
Speedup OpenMP	1.98	1.64	1.78	2.05	2.09	2.14
Speedup std	1.38	1.72	1.88	1.95	2.03	2.05
Speedup OpenMPI	3	4.29	4.81	5.1	5.23	5.33

13.3 8 Threads

Bei einer Parallelisierung mit 8 Threads, auf einer Maschine mit 4 Prozessoren, hat OpenMPI zu lange Zeiten produziert, um sie in das Diagramm aufzunehmen.



	1000 x 1000	2000 x 2000	3000 x 3000	4000 x 4000	5000 x 5000	6000 x 6000
Speedup OpenMP	1.07	1.27	1.63	1.75	1.99	1.84
Speedup std	1.05	1.21	1.33	1.48	1.56	1.58

14 Fazit

Die Parallelisierung der Wellengleichung hat die Berechnung wie erwartet deutlich beschleunigt. Bei wachsenden Daten waren bei den OpenMP und std Varianten Speedup Werte > 2 zu sehen. In so gut wie jedem Durchlauf war OpenMP schneller als die std Variante und hat sich weiter von ihr entfernt. Darüber hinaus war die OpenMP Implementierung leichter zu entwickeln. Sie entspricht größtenteils der Sequentiellen, mit Direktiven für die Parallelisierung. Auf einer Maschine mit vier Prozessoren lag der größte Speedup bei einer Parallelisierung mit 4 Threads. Aber auch bei 2 und 8 Threads lief die parallelisierten Varianten schneller durch.

Bei 2 und 4 Threads hat die OpenMPI Implementierung unerwarteterweise alle anderen Strategien um Längen geschlagen. Bei 4 Threads waren Speedup Werte um 5 zu sehen und bei 2 Threads annähernd 4. Jedoch sind die Laufzeiten bei 8 Threads dermaßen in die Höhe geschnellt, dass sie nicht mehr aufgeführt sind. Vermutlich liegen diese Zeiten an dem Speichermodell von OpenMPI. Bei den std und OpenMP Varianten arbeitet ein geshartes Speichermodell hinter den Kulissen, die OpenMPI Prozesse haben alle ihren eigenen Speicher und tauschen ihn mit MPI_Send() sowie MPI_Gather() aus.