Parallele Programmierung in C++ mit OpenMP

Lukas Wendt

Hochschule für Angewandte Wissenschaften Hamburg
lukas.wendt@haw-hamburg.de

31. Januar 2017

Zusammenfassung

In den letzten Jahren haben sich Mehrkernsystem am Markt erfolgreich etabliert. Im Server Segment gibt es einzelne Prozessoren, die bis zu 72 Kerne besitzen. Um diese Hardware voll auszunutzen, muss bei der Software Entwicklung Software aufwändig angepasst werden. Gleichzeitig verschlechtert sich die Möglichkeit zur Portierbarkeit auf andere Systeme. Um den Aufwand für den Entwickler gering zu halten und gleichzeitig eine dynamische Zuweisung von Kernen zu verschiedenen Berechnungen zu ermöglichen, wurde OpenMP entwickelt. OpenMP kann für die Programmiersprachen C, C++ und Fortran verwendet werden. Es nutzt gemeinsamen Speicher und die Pragmas können einfach in vorhandenen Code eingefügt werden. Die potentiellen Leistungsverbesserungen, die durch die Verwendung von OpenMP gegenüber einer Single-Thread und einer MPI Lösung entstehen, werden dargestellt. Mithilfe einiger Beispiele wird eine Einführung in die Programmierung mit OpenMP gegeben.

I. Einleitung

Durch parallele Programmierung können Berechnungen auf einem oder mehreren Computersystemen von einem oder mehreren Kernen gleichzeitig ausgeführt werden. Im Optimalfall führt dies zu einer geringeren Ausführungsdauer (Tp) und zu einem Geschwindigkeitszuwachs (SpeedUp).

$$Tp = \frac{DauerDerSeriellenAusfuehrung}{AnzahlDerBeteiligtenKerne}$$
$$SpeedUp = \frac{DauerDerSeriellenAusfuehrung}{Tp}$$

Für die Umsetzung gibt es für den Entwickler verschiedene Möglichkeiten, z.B. indem direkt verschieden Prozesse und Threads entwickelt werden, die die Berechnungen parallel ausführen. Eine weitere Möglichkeit ist die Verwendung einer Programmiersprache, die für die parallele Programmierung entwickelt wurde. Außerdem gibt es libaries, wie OpenMP oder MPI, die bekannte Programmiersprachen wie C++ um Funktionalitäten zur Parallelisierung erweitern. In dieser Ausarbeitung wird der Schwerpunkt auf OpenMP gelegt. Auch in Single-Core Systemen kann Parallelisierung die Performance erhöhen. Liest beispielsweise ein Thread eine Datei von der Festplatte und blockiert eine lange Zeit, kann trotzdem ein anderer Thread in der Zwischenzeit Berechnungen durchführen.

i. Geschichte

Anfang 1960 wurden die ersten integrierten Schaltkreise (IC)s entwickelt. Daraus entwickelte Gordon Moore bereits 1965 ein Gesetz, wonach sich alle 1-2 Jahre die Anzahl der Transistoren in einem solchen IC verdoppeln würde. Zusätzlich stieg bis in die 2000er Jahre die Anzahl an Instruktionen, die ein Prozessor pro Sekunde (IPC) ausführen kann, im gleichen Maße, weil sich auch die Taktfrequenz und der Stromverbrauch erhöhten. Aufgrund physikalischer Probleme lässt sich die Taktfrequenz nicht weiter erhöhen und auch der Stromver-

brauch pro IC lässt sich nicht weiter steigern, da die Wärmeleitkapazität von Kupfer Kühlkörpern ebenfalls eine physikalische Grenze erreicht hat. Daher entstanden Ende der 2000er Jahre Mehrkernsysteme. Im Optimalfall entspricht die Anzahl der Kerne der Beschleunigung des Programms, sodass hier nach wie vor theoretisch eine Leistungssteigerung möglich ist, die entsprechend der Transistorenanzahl skaliert.

ii. Problemstellung

Wenn die Ausführungsdauer eines Programms zu hoch ist, können eigenen Threads für die Parallelisierung entwickelt, Algorithmen verbessert, neue Hardware gekauft oder es kann eine libary verwendet werden, die den Entwickler bei der Parallelisierung des vorhandenen Programms unterstützt. Um eine solche Libary geht es in dieser Arbeit. Die zeigen soll wie mit OpenMP einfach und komfortabel Parallelisierte Software entwickelt werden kann.

II. Grundlagen

Im Kapitel „Grundlagen“ werden OpenMP und MPI kurz vorgestellt.

i. OpenMP

Die Abkürzung OpenMP steht für Open Multi-Processing. Diese Libary arbeitet auf der Basis von Schleifen und parallelisiert die Ausführung in unterschiedlichen Threads. Da der preprocessor die Anweisungen zur Parallelisierung erhält, kann vorhandener Code einfach um OpenMP erweitert werden. Am Code selber sind keine Änderungen nötig und auch Compiler, die OpenMP nicht unterstützen, können den Code kompilieren, da der preproccessor die unbekannten Pragmas ignoriert. OpenMP kann in C, C++ und Fortran eingesetzt werden. Viele übliche Compiler, wie der Open Source Compiler gcc und einige kommerzielle, unterstützen die Erweiterung OpenMP. Der Quellcode, der im weiteren Verlauf dieser Ausarbeitung erklärt und beschrieben wird, wurde mit dem gcc kompiliert. Er sollte aber auch mit anderen Compilern kompiliert werden können.

ii. MPI

MPI steht für Message Passing Interface und kann für die parallele Berechnung auf einem oder mehreren Computern in einem Cluster eingesetzt werden. Die Berechnungen werden in parallel ablaufenden Prozessen durchgeführt. Für die Kommunikation zwischen den Prozessen wird asynchrone Kommunikation verwendet. MPI ist komplizierter zu verwenden als OpenMP, da sich der Entwickler zusätzlich um die Interprocess Kommunikation (IPC) kümmern muss. Daher wird häufig eine Kombination aus MPI und OpenMP eingesetzt. Ein Beispiel findet sich hier: [9]. MPI ist ebenfalls für C, C++ und Fortran verfügbar.

III. OpenMP

i. Installation

OpenMP kann mit verschiedenen Entwicklungsumgebungen und Compilern genutzt werden. Für die Erstellung dieses Textes wurde ein Debian 8 System und der GCC Kompiler in der Version 6.2.1 genutzt, die der der OpenMP Version 4.5 enstpricht. Die neuesten Features (Tasks), die in dieser Ausarbeitung beschrieben werden, sind bereits ab der Version 3.0 verfügbar.

i.1 Linux

Die Installation wurde wie hier beschrieben durchgeführt: [1]. Die Installation auf anderen Debian Derivaten läuft ähnlich ab. Auf anderen Linux Systemen heißen die Pakete im jeweiligen Paketmanager eventuell etwas anders. Eine Suche nach OpenMP und der entsprechenden Distribution sollte Abhilfe schaffen.

i.2 Windows

Mit der Entwicklungsumgebung Visual Studio und dem enthaltenen Compiler für C und

C++ können ebenfalls OpenMP Projekte umgesetzt werden. Hierfür muss der Compilerflag /openmp aktiviert werden. Eine detaillierte Beschreibung befindet sich hier: [2]

ii. Hello World

Für die Einführung wird ein Hello World Beispiel verwendet:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void){
  int iam = 0, np = 1;
  #pragma omp parallel default(shared)
    private(iam, np)
  {
    #if defined (_OPENMP)
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
    #else
      printf("OpenMp not available\n");
      exit(1);
    #endif
    printf("Hello from thread %d out of %d
    \n", iam, np);
  }
}
```

Listing 1: *Hello World*

Damit der Quellcode mit OpenMP Support kompiliert werden kann, muss der Kompileraufruf um das Flag -fopenmp erweitert werden. Außerdem ist die Umgebungsvariable OMP_NUM_THREADS zu setzen. Dies geschieht mit dem Befehl:

```
export OMP_NUM_THREADS=<numberOfThreads>
```

Listing 2: *Anzahl der Threads vorgeben*

Mit dieser Umgebungsvariable wird OpenMP mitgeteilt, wie viele Threads gestartet werden sollen. Wenn dieser Wert über der Anzahl Kernen/Threads des Systems liegt, kann das Betriebssystem diese Threads nur sequentiell ausführen. In manchen Situationen kann auch eine höhere Anzahl Threads, als das System zur Verfügung stellt sinnvoll sein. Beispielsweise wenn Threads in einer Funktion blockieren, können die restlichen Threads nach wie vor parallel ausgeführt werden. Über das #pragma omp parallel wird dem Compiler mitgeteilt, dass der folgende Code parallel auszuführen ist. Compiler, die dieses pragma nicht kennen, ignorieren es und führen den Code sequentiell aus. Der Ausdruck „private(iam,np)“ sorgt dafür, dass jeder Thread eine eigene Instanz von diesen Variablen erhält. Es sollte daher davon ausgegangen werden, dass die Variablen im parallelen Teil zunächst nicht initialisiert sind. Mit dem Keyword default mit dem Parameter shared werden alle nicht anders definierten Variablen für alle Threads nur einmal im Speicher abgelegt. Die Funktion omp_get_num_threads() gibt die Anzahl der Threads zurück und entspricht somit im parallelen Teil der Systemvariable OMP_NUM_THREADS. Im seriellen Teil ist der Wert gleich 1. Die Funktion omp_get_thread_num() gibt die ID des auszuführenden Threads zurück, diese kann von 0 bis omp_get_num_threads()-1 gehen.

Die Ausgabe des Programms mit 6 Threads sieht wie folgt aus:

```
Hello from thread 1 out of 6
Hello from thread 0 out of 6
Hello from thread 3 out of 6
Hello from thread 5 out of 6
Hello from thread 2 out of 6
Hello from thread 4 out of 6
```

Listing 3: *Hello World mit 6 Threads*

Falls die Meldung „OpenMP not available“ erscheint, wurde das Flag -fopenmp nicht gesetzt, der Compiler unterstützt OpenMP nicht oder die OpenMP Libaries sind nicht installiert.

IV. Keywords

Jede OpenMP Anweisung muss mit dem „#pragma omp“ beginnen. Das Keyword **parallel** bedeutet, dass der nachfolgende Code Teil parallel ausgeführt wird. Wenn innerhalb des Parallelen Code Abschnitts ein Teil nur von einem Thread ausgeführt werden soll, kann dies mit dem Keyword **single** erreicht werden.

i. Schleifen

Es gibt verschieden Schleifen bei OpenMP. Zunächst folgt ein Beispiel mit dem For Konstrukt:

```
#include <omp.h>
#include <math.h>
#include <iostream>

int main(void){
  double sinTab[360];
  #pragma omp parallel for
  for(int i = 0; i < 360;i++){
    sinTab[i]=sin((double)i);
    printf("%d = %f from: %d\n",i,sinTab[i
    ],omp_get_thread_num());
  }
  return 0;
}
```

Listing 4: *Sinus Stuetzwerttabelle berechnen*

Mit diesem Code Beispiel wird eine Sinus Stützwerttabelle berechnet. Da jede Berechnung im Schleifendurchlauf unabhängig von der vorhergehenden Berechnung ist, kann die Schleife theoretisch von bis zu 360 Threads parallel berechnet werden. Einige Näherungsverfahren lassen sich so nicht parallelisieren, da sie immer den vorhergehenden Wert benötigen um den nächsten zu berechnen. Die Reihenfolge ist außerdem nicht dieselbe, wie im Falle der sequentiellen Ausführung. Dies zeigt sich in der printf Ausgabe. Die Ausgabe des Schleifenzählers wird nicht mehr sortiert, von 0-359 ausgegeben, sondern erfolgt mit jedem Aufruf, in einer anderen Reihenfolge.

ii. Sections

Ein Codeabschnitt mit Sections sieht wie folgt aus:

```
#include <omp.h>
#include <iostream>
#include <unistd.h>
int main(void){
  printf("Seq. Ausfuehrung vom Master
    Thread: %d\n",omp_get_thread_num());
  #pragma omp parallel
  {
    printf("Wird von jedem Thread
    ausgefuehrt: %d\n" ,
    omp_get_thread_num());
    sleep(1); // Simulate some Workload
    #pragma omp sections
    {
      #pragma omp section
      {
        printf("Ich bin der Eine: %d\n",
    omp_get_thread_num());
        sleep(1);
      }
      #pragma omp section
      {
        printf("Und ich der andere: %d\n",
    omp_get_thread_num());
      }
    }
    printf("Wird von jedem Thread
    ausgefuehrt: %d\n", omp_get_thread_num
    ());
  }
  printf("Wird Seq. vom Master Thread
    ausgefuehrt: %d\n ",
    omp_get_thread_num());
}
```

Listing 5: *Section Beispiel*

Sections sind hilfreich, wenn gleichzeitig unterschiedliche Berechnungen durchgeführt werden müssen und keine automatische Teilung einer Schleife möglich ist. Für die Parallelisierung von Rekursionen sind Sections gut geeignet. Auch für ein Consumer Producer Problem wären Sections gut einsetzbar.

```
Seq. Ausfuehrung vom Master Thread: 0
Wird von jedem Thread ausgehfuehrt: 0
Wird von jedem Thread ausgehfuehrt: 1
Ich bin der Eine: 0
Und ich der andere: 1
Wird von jedem Thread ausgefuehrt: 1
Wird von jedem Thread ausgefuehrt: 0
Wird Seq. vom Master Thread ausgefuehrt: 0
```

Listing 6: *Section Beispiel Ausgabe*

Wie an der Ausgabe zu sehen ist, wird der Code vor dem OpenMP Pragma seriell ausgeführt und zwar von der ID 0. Der Master Thread hat immer diese ID. Der Code inerhalb des „#pragma omp parallel“ wird wiederum von zwei Threads ausgeführt. Da die Anzahl bei diesem Test auf 2 Threads limitiert war, sind das alle Threads. Die Sections innerhalb von „#pragma omp section“ werden parallel ausgeführt. Die Reihenfolge, in der beide Sections beendet werden, variiert je nach Aufruf. Sie ist also nicht vorgegeben. Am Ende des Section Abschnitts befindet sich automatisch eine Barrier. Erst wenn beide Sections beendet sind, wird der parallele Teil Zeile 6 und 7 ausgeführt. Bei Sections kann der Compiler die Aufteilung nicht optimieren. Daher würde dieses Beispiel, beim Aufruf mit 4 Threads nur

2 Threads starten. Auch wenn Sections unterschiedlich groß sind führt das dazu das Kerne nur teilweise für die Berechnung genutzt werden. Nachdem der parallele Teil des Programms verlassen wurde, wird die letzte Zeile wieder nur vom Master Thread ausgeführt. Eine Section wird jeweils nur einmal aufgerufen. Falls dies nicht gewünscht ist, sollte ein Task verwendet werden.

iii. Task

Bei der Verwendung von Task steht keine Barrier am Ende. Außerdem kann mithilfe einer Schleife eine Task mehrfach ausgeführt werden, was mit einer Section nicht möglich ist. Tasks sind wie Sections für rekursive Funktionen geeignet. Jeder Funktionsaufruf startet eine Task. Um Tasks und Sections zu vergleichen wird bei dem Source Code aus dem Section Abschnitt nun aus jeder Section eine Task :

```
#include <omp.h>
#include <iostream>
#include <unistd.h>
int main(void){
  printf("Seq. Ausfuehrung vom Master
    Thread: %d\n",omp_get_thread_num());
  #pragma omp parallel
  {
    printf("Wird von jedem Thread
    ausgefuehrt: %d\n" ,
    omp_get_thread_num());
    sleep(1); // Simulate some Workload
    #pragma omp task
          {
      printf("Ich bin der Eine: %d\n",
    omp_get_thread_num());
      sleep(1);
          }
    #pragma omp task
    {
      printf("Und ich der andere: %d\n",
    omp_get_thread_num());
    }
    printf("Wird von jedem Thread
    ausgefuehrt: %d\n", omp_get_thread_num
    ());
  }
  printf("Wird Seq. vom Master Thread
    ausgefuehrt: %d\n ",
    omp_get_thread_num());
}
```

Listing 7: *Task Beispiel*

Auf der Konsole werden die Ausgaben mit 2 Threads folgendermaßen ausgegeben:

```
Seq. Ausfuehrung vom Master Thread: 0
Wird von jedem Thread ausgefuehrt: 0
Wird von jedem Thread ausgefuehrt: 1
Wird von jedem Thread ausgefuehrt: 0
Ich bin der Eine: 0
Wird von jedem Thread ausgefuehrt: 1
Und ich der andere: 1
Ich bin der Eine: 1
Und ich der andere: 1
Wird Seq. vom Master Thread ausgefuehrt: 0
```

Listing 8: *Task Beispiel Ausgabe*

Die ersten drei Zeilen sind identisch bei Tasks und Sections. Danach wird eine weitere Zeile „Wird von jedem Thread ausgefuehrt“ ausgegeben. Da sich hinter den Tasks keine Barrier befindet, kann der weitere Code ausgeführt werden. Falls trotz der Verwendung von Tasks nach der Ausführung auf noch laufende Tasks gewartet werden soll, kann dies mit dem „#pragma omp taskwait“ bestimmt werden. Ein weiterer Unterschied besteht darin, dass die einzelnen Tasks mehrfach aufgerufen werden und zwar so häufig wie Threads vorhanden sind. Wie oft die Task aufgerufen werden soll, kann vorgegeben werden, indem ein Task in einer For-Schleife aufgerufen wird. Am Ende des parallen Code Abschnittes befindet sich eine Barrier, denn die Ausgabe „Wird Seq. vom Master Thread ausgefuehrt: 0“ erscheint erst nachdem alle Ausgaben aus dem parallen Abschnitt erfolgt sind.

iv. Synchronisation

Die Keywords zur Synchronisation wirken sich nur im parallelen Abschnitt des Programms aus. Das Keyword **master** {Code Abschnitt} bedeutet, dass der Code Abschnitt nur von einem Thread, dem Thread mit der ID 0, ausgeführt wird. Die **barrier** bewirkt, dass an dieser Stelle alle Threads warten, bis alle anderen dort angekommen sind. Mit **critical** {Code Abschnitt} kann eine kritische Sektion gekennzeichnet werden, in der sich nur ein Thread zur gleichen Zeit befinden darf. Wenn nur eine einzelne Operation ausgeführt werden soll, kann das Keyword **atomic** verwendet werden. Hierbei

wird zwischen atomic read, atomic write und atomic unterschieden. Bevor auf eine globale Variable lesend zugegriffen wird, sollte atomic read verwendet werden. Bei einem schreibenden Zugriff kann atomic write verwendet werden. Wird nur das Keyword atomic verwendet wird sowohl read als auch write durchgeführt. Das ist z.B. sinnvoll, wenn mehrere Threads eine Variable inkrementieren und daher eine lesenden und einen schreibenden Zugriff durchführen. Mit dem Keyword **flush** kann dasselbe erreicht werden wie mit dem Keyword „volatile “, das aus der C Programmierung bekannt ist. Der Vorteil am **flush** ist, dass dieser dort eingetragen wird, wo gewünscht ist, dass der Wert der Variable übernommen wird. Bei „volatile“ ist dies immer der Fall, einen Codeabschnitt, der sequentiell ausgeführt wird, könnte der Compiler nicht mehr optimieren ([8]). Bei OpenMP mit flush ist diese Optimierung möglich, da mit flush nur die Optimierungen in der nächsten Zeile abgeschaltet werden. In OpenMP ist auch eine Funktion zum locken von Bereichen vorhanden. Sie funktioniert ähnlich wie die Verwendung von Mutexes oder Semaphoren, indem eine Lock Variable mit „omp_lock_t lock;“ angelegt wird. Mit omp_init_lock(int*) kann sie erstellt und mit omp_destroy_lock(int*) wieder zerstört werden. Ein Lock wird mit omp_set_lock(int*) gesetzt und mit omp_unset_lock(int*) wieder entfernt. Als Beispiel folgt ein Consumer/Producer Beispiel:

```
#include <iostream>
#include <omp.h>
#include <unistd.h>
#include <math.h>
int main(void){
  int data;
  omp_lock_t lock_write;
  omp_lock_t lock_read;
  omp_init_lock(&lock_write);
  omp_init_lock(&lock_read);
  omp_set_lock(&lock_read);

  #pragma omp parallel
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      {
        while(1){
          omp_set_lock(&lock_read);
          printf("Consume :%d, with: %d\n"
    ,data,omp_get_thread_num());
          omp_unset_lock(&lock_write);
          sleep(1); //Workload
        }
      }

    }
    for(int i = 0; i < 3 ; i++){
      #pragma omp task
      {
        while(1){
          omp_set_lock(&lock_write);
          data=rand()>>16;
          #pragma omp flush(data)
          printf("Produce :%d, with: %d\n"
    ,data,omp_get_thread_num());
          omp_unset_lock(&lock_read);
          sleep(3);
        }

      }
    }
  }
}
```

Listing 9: *Consumer / Producer Beispiel*

Um den Zugriff auf eine Variable zu kontrollieren wird eine Lock Variable für den lesenden und eine Lock Variable für den schreibenden Zugriff deklariert. Da nicht vorgegeben ist ob der Producer oder der Consumer zuerst startet, wird die „lock_read“ Variable gelockt. Danach beginnt in Zeile 13 der parallele Teil der Anwendung. Zunächst wird der Section Bereich definiert. Das Keyword „nowait“ ist erforderlich damit die Barrier am Ende des sections Abschnitt entfernt wird und der Programmfluss weiterläuft. In der Section wird zunächst die „lock_read“ Variable geprüft und gelockt. Fall sie nicht gelockt war, wird die Data Variable ausgeben und zum Schluss die Variable „lock_write“ unlockt. Im nächsten Abschnitt wird mithilfe der For-Schleife dreimal derselbe Task gestartet. Die Task prüft die „lock_write“ Variable, erzeugt eine zufällige Zahl und speichert diese im Buffer. Anschließend gibt die Task die „lock_read“ Variable frei. In diesem Beispiel braucht der Producer 3 Sekunden und der Cosumer 1 Sekunde. Aus diesem Grund sollen 3 Producer gleichzeitig einen Consumer versorgen. Die Ausgabe sieht folgendermaßen

aus:

```
Produce :27531, with: 2
Consume :27531, with: 0
Produce :12923, with: 1
Consume :12923, with: 0
Produce :25660, with: 3
Consume :25660, with: 0
Produce :26163, with: 2
Consume :26163, with: 0
Produce :29872, with: 1
Consume :29872, with: 0
```

Listing 10: *Consumer / Producer Beispiel Ausgabe*

Hinter „Produce“ oder „Consume“ steht die Zahl, die der Producer generiert oder der Consumer eingelesen hat. Hinter „with:“ steht, welche Thread ID den jeweiligen Abschnitt ausgeführt hat. Es wird zunächst das erste Element erzeugt und erst danach vom Consumer eingelesen. Der Consumer hat immer dieselbe ID, während die Producer zwischen den IDs 2,1 und 3 abwechseln. Das liegt daran, dass die Task vom Producer durch die For-Schleife dreimal gestartet wurde und der Consumer durch die Verwendung einer Section nur einmal aufgerufen wird. Da der Producer ein längeres, simuliertes Workload von 3 Sekunden hat, kann der Consumer, der ein Workload von 1 Sekunde besitzt, 3 Elemente in der selben Zeit verarbeiten. Wenn 2 Threads verwendet würden, hätte das zur Konsequenz, dass die gesamte Verarbeitungsdauer beider Threads unverändert bei 3 Sekunden (pro Elemeent) liegt, weil der Producer auf den Consumer warten muss. Dieses Beispiel wurde mit 4 Threads ausgeführt, die dann eine Verarbeitungsdauer von nur 1 Sekunde (pro Element) besitzen. So entstehen durch die manuelle Parallelisierung im Software Code Schwierigkeiten bei der Portierung einer Anwendung. Die Software liefe mit 4 Threads optimal, während sie mit 2 Threads dreimal so langsam ist. Bei der Verwendung von mehr als 4 Threads würde sie ebenfalls nicht profitieren.

v. Sichtbarkeiten

Ein Überblick über die Sichtbarkeiten:

Sichbarkeit	Beschreibung
private	Eigene Variable uninitialisiert
shared	alle Threads eine Variable
reduction	zum akkumulieren von Werten

Tabelle 1: *Sichtbarkeiten OpenMP*

Die private Sichtbarkeit hat zwei spezielle Ausprägungen. Bei firstprivate erhält jeder Thread eine eigene Variable, die mit dem Wert der Variable im vorhergegangen Code Abschnitt übereinstimmt. Bei der lastprivate Sichtbarkeit wird versucht, dass die Variable nach dem parallelen Code Abschnitt denselben berechneten Wert wie bei einer Berechnung ohne Parallelisierung erhält. So können Auswirkungen auf weiter folgende Berechnungen möglichst gering gehalten werden. Wenn nur private verwendet wird, ist die Variable zu Beginn uninitialisiert und nachdem parallelen Abschnitt ist der Wert der Variable gleich dem Wert vor dem parallelen Abschnitt. Bei der Sichtbarkeit reduction erhält jeder Thread eine eigene Variable und nach Beendigung der Threads wird eine arithmetische Operation mit allen Ergebnissen durchgeführt. Das sieht folgendermaßen aus: reduction(operator : variable). Mit dem Keyword default(<Sichtbarkeit>) kann der Default Wert für alle nicht angegebenen Sichtbarkeiten angepasst werden.

vi. Funktionen

Die Funktion **omp_get_num_threads()** liefert die Anzahl der aktiven Threads. Im sequentiellen Teil der Anwendung ist diese Zahl immer eins, im parallelen Teil hängt sie ab vom oben beschriebenen Parameter oder dem Wert, der mit der Funktion **omp_set_num_threads(int)** gesetzt wurde. Falls beides nicht der Fall sein sollte, ist der Wert gleich eins. Um die Anzahl an physikalisch vorhandenen Prozessoren zu ermitteln, kann die Funktion **omp_get_num_procs()** verwendet werden. **omp_get_max_threads** gibt die Anzahl an Threads zurück, die OpenMP maximal anlegt. Die Ausgabe ist unabhängig davon, ob sich das Programm beim Aufruf im sequentiellen

oder im parallelen Teil des Programms befindet. Beim Einsatz dieser Funktionen sollte bedacht werden, dass ein Compiler, der OpenMP nicht unterstützt, diese Funktionen nicht kennt. Daher sollte OpenMP in diesem Fall mit dem preprocessor auskommentiert werden, wie oben im Hello World Beispiel mit „#if defined (_OPENMP) “ vorgestellt.

vii. Umgebungsvariablen

Im folgenden Abschnitt werden die Umgebungsvariablen und die Funktionen, mit denen diese gesetzt werden, beschrieben. In **OMP_DYNAMIC** steht, ob die Thread Anzahl für das Programm dynamisch bestimmt werden soll. Der Parameter kann mit der Funktion omp_set_dynamic(bool) gesetzt werden. Der default Wert ist false. Mit **OMP_THREAD_STACK_SIZE** kann vorgegeben werden, wie viel Speicher ein einzelner Thread maximal auf dem Stack allozieren kann. Der Default Wert beträgt 16MB. Dieser Wert kann nicht während der Laufzeit angepasst werden. Daher ist er auch nicht mit einer Funktion zu verändern. Beim Setzen kann K und M für Kilobyte und Megabyte an die Zahl angehängt werden. Das Minimum pro Thread beträgt 16 Kilobyte. Mit der Variable **OMP_SCHEDULE** kann eingestellt werden, wie der Workload in einer Schleife aufgeteilt wird. Eine grobe Beschreibung der Parameter zeigt folgende Website: https://software.intel.com/en-us/articles/openmp-loop-scheduling . Wenn die Anzahl an Threads dynamisch bestimmt wird, kann es praktisch sein, eine obere Grenze festzulegen. Die kann mit der Umgebungsvariable **OMP_THREAD_LIMIT** vorgegeben werden. Zur Laufzeit kann mit der Funktion omp_get_thread_limit() das derzeitige Limit abgefragt werden und mit der Funktion omp_set_thread_limit(int) kann das Limit verändert werden.

V. Performance

Um die Performance zwischen OpenMP, einer Single Thread und einer MPI Lösung zu vergleichen, wurde ein Code entwickelt, der die Anzahl an Prim Zahlen bis zu einer vorgegeben Zahl ermittelt. Diese Berechnung lässt sich gut in unabhängige Teile teilen. So prüft z.B. 1 Thread die Zahlen 0-100 während ein anderer die Zahlen 101-200 prüft. Beide Threads inkrementieren eine eigene Variable. Am Ende werden diese Variablen addiert und als gemeinsames Ergebnis ausgegeben.
Beispielcode OpenMP:

```
#include <iostream>
#include <omp.h>
#define START 2
#define END 200000

int main()
{

  int count=0;
  #pragma omp parallel for reduction ( + :
     count)
  for(int i = START; i < END; i++){
    bool isPrime = true;
    for(int j = START; isPrime && j < i; j
    ++){
      if(i != j){
        isPrime = !!(i % j);
      }
    }
    if (isPrime){
      count++;
    }
  }
  std::cout << "count: " << count << std::
    endl;
}
```

source/par.c

Beispielcode MPI:
Ein Beispiel-Programm findet sich hier : [5]

Anzahl Threads	OpenMP[s]	MPI[s]
1	11,197	11,58
4	4,6	5,82
16	3,0	5,85
128	2,872	6,56

Tabelle 2: *Performance Vergleich MPI und OpenMP*

Die Berechnungsdauer der Single Thread Lösung liegt bei 11,659 s und damit etwas über der Berechnungsdauer, die OpenMP und MPI benötigen, wenn ein Thread verwendet wird. Desto mehr Threads verwendet werden, desto stärker setzt sich OpenMP von MPI ab.

i. Speicherbedarf

Da OpenMP Threads verwendet, muss nur ein Teil der Variablen doppelt im Speicher stehen, während bei MPI jeder Prozess alle Variablen speichert. Für den Datenaustausch wird dort IPC genutzt. Um abschätzen zu können, wie groß dieser Speicherbedarf ist, hier ein tabellarischer Vergleich:

Anzahl Threads	OpenMP[MB]	MPI[MB]
1	<1	6
4	1	15
16	2	58
128	4	579

Tabelle 3: *Speicherverbrauch in Abhängigkeit zur Anzahl der Threads/Prozesse*

Bei einer großen Anzahl an Threads steigt der Speicherverbrauch von MPI stark an, während sich der Speicherverbrauch der Lösung mit MPI nur sehr gering vergrößert. Für die Berechnung der Primzahlen werden eigentlich nur wenige Variablen benötigt und trotzdem steigt hier der Speicherbedarf stark an. Daraus folgt, dass ein Großteil des Speicherbedarfs auf den Overhead von MPI zurückgeht.

ii. Anzahl Threads

Bei der Messung der Laufzeit des Zählens von Primzahlen lag die CPU-Auslastung des 4-Kerne-Testsystems nur teilweise bei 100%. Das legt die Vermutung nahe, dass noch an Performance gewonnen werden kann, wenn mehr Threads gestartet werden als Kerne vorhanden sind.

Anzahl Threads	**Berechnungsdauer in sec**
1	11,197
2	8,218
3	5,901
4	4,599
6	3,404
8	3,268
16	3,004
32	2,890
64	2,848
128	2,872
256	2,892
dynamisch	5,915

Tabelle 4: *Performance in Abhängigkeit zur Threadanzahl*

Wie in der Tabelle zu sehen, steigt die Performance des Programms auch bei einer Anzahl an Threads, die größer ist als die Anzahl der vorhandenen physischen Kerne. Das liegt zum einen daran, dass sich eventuell einzelne Threads im blockierten Zustand befinden. Zum anderen wäre es möglich, dass Threads auf das Ergebnis anderer Threads warten. Da das Testsystem einen verdrängenden Scheduler verwendet, bewirken viele Threads, dass andere Threads, die nicht zum Programm gehören, den Programmfluss seltener unterbrechen können. Während des Tests mit 256 Threads war daher das System nicht mehr zu benutzen (100% Auslastung durch einen Prozess). Das Sytem war mit 32 Threads noch bedienbar und der Prozess verursachte nur eine 97% Auslastung. Um zu sehen, wie sich die dynamische Funktion verhält, wurde diese ebenfalls vermessen. Auf diese Weise kann OpenMP selbst entscheiden, wie viele Threads gestartet werden sollen. Da das Messergebnis fast identisch mit der Messung mit 3 Threads ist, hat die dynamische Funktion wahrscheinlich 3 Threads gestartet. Obwohl das Ergebnis deutlich unter der sequentiellen Ausführung liegt, ist das noch nicht das optimale Ergebnis. Das beste Ergebnis (2,848) liefert die Ausführung mit 64 Threads. Bei einem theoretischen SpeedUp von 4 läge dieser Wert bei 11,197/4 = 2,799 s, der

dicht an der tatsächlich gemessenen Berechnungsdauer von 2,848 liegt. Dies erklärt sich durch zusätzlichen Overhead und andere Programme im Hintergrund.

VI. Probleme

Die Funktionen printf und cout sind nicht thread-safe. Bei den Beispielen aus dieser Ausarbeitung funktionieren die printf Ausgaben. Wenn man diese aber durch cout Ausgaben austauscht, werden die Ausgaben der verschiedenen Threads miteinander vermischt. Damit die Ausgabe durch mehrere Threads sichergestellt ist, muss vor dem Aufruf von printf ein Mutex gelockt werden, sodass nie mehr als ein Thread gleichzeitig in der Funktion ist. Eine schönere Lösung wäre eine thread-safe Lösung. So könnte z.B. ein Thread(A) einen Buffer für jeden weiteren Thread zur Verfügung stellen, die ihre Ausgaben in diese Buffer speichern. Der Thread(A) würde anschließend alle Buffer koordiniert ausgeben.

VII. Fazit

OpenMP ist eine einfache Möglichkeit, ein Programm zu parallelisieren. Die OpenMP Pragmas können in vorhandene Quellcodes eingefügt werden, sodass vorhandene Software ohne großen Aufwand parallelisiert werden kann. Wenn Codeabschnitte im vorhandenen Code hintereinander stehen, können diese über Sections parallelisiert werden. Die Parallelisierung mit Sections birgt aber den Nachteil gegenüber der Parallelisierung mit Schleifen, dass Sections nicht mit der Anzahl der Threads skalieren. Dies lässt sich gut beim Consumer / Producer Beispiel sehen. Eine Schleife lässt sich mit OpenMP besser parallelisieren, wenn sie auf keine vorhergehenden Ergebnisse zurückgreift. So kann jeder Umlauf in einem einzelnen Thread bearbeitet werden. Mit Tasks können gleiche Threads parallel ausgeführt werden. Das kann die Entwicklung einer Server Anwendung vereinfachen. Im Vergleich zu anderen Lösungen, wie MPI, kann auf die Kommunikation zwischen den Prozessen verzichtet werden. Der Performance Vergleich zwischen OpenMP und MPI geht deutlich zu Gunsten von OpenMP aus. Auch der Speicherverbrauch ist bei OpenMP deutlich geringer. Seit der Version 4.0 bietet OpenMP außerdem die Möglichkeit, Berechnungen auf eine graphics processing unit (GPU) auszulagern. Wer darüber hinaus nach einer guten Übersicht für die Programmierung mit OpenMP sucht, wird hier fündig: [3]

Literatur

[1] Huseyin Cakir. *Installing OpenMP in Linux (Debian)*. https://huseyincakir.wordpress.com/2009/11/05/installing-openmp-in-linux-debian/. Nov. 2009. (Besucht am 24. 01. 2017).

[2] Microsoft Corporation. */openmp (Aktivieren der OpenMP 2.0-Unterstützung)*. https://msdn.microsoft.com/de-de/library/fw509c3b.aspx. (Besucht am 24. 01. 2017).

[3] E. Berta. *OpenMP Reference Sheet for C/C++*. http://www.plutospin.com/files/OpenMP_reference.pdf. (Besucht am 24. 01. 2017).

[4] Pawan Ghildiyal. *MPI vs OpenMP: A Short Introduction Plus Comparison (Parallel Computing)*. http://pawangh.blogspot.de/2014/05/mpi-vs-openmp.html. Juni 2014. (Besucht am 10. 01. 2017).

[5] John Burkardt. *PRIME_MPI*. https://people.sc.fsu.edu/ jburkardt/cpp_src/prime_mpi/prime_mpi.cpp. Juni 2016. (Besucht am 24. 01. 2017).

[6] OpenMP.org. *OpenMP Compilers*. http://www.openmp.org/resources/openmp-compilers/. Dez. 2008. (Besucht am 10. 01. 2017).

[7] Prof. Dr. Rita Loogen. *Der OpenMP Standard*. http://www.mathematik.uni-marburg.de/ loogen/Lehre/ws08/ParProg/Folien/ParProg6.pdf. Dez. 2008. (Besucht am 10. 01. 2017).

[8] Michael Suß und Claudia Leopold. *Common Mistakes in OpenMP and How To Avoid Them A Collection of Best Practices*. https://pdfs.semanticscholar.org/4e60/37127e85445079272e1cb5574cbcce2e175e.pdf. (Besucht am 10. 01. 2017).

[9] Alf Wachsmann. *Mixing MPI and OpenMP*. http://www.slac.stanford.edu/comp/unix/farm/mpi_and_openmp.html. März 2006. (Besucht am 10. 01. 2017).

[10] Joel Yliluoma. *Guide into OpenMP: Easy multithreading programming for C++*. http://bisqwit.iki.fi/story/howto/openmp/. Juni 2016. (Besucht am 10. 01. 2017).