Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Parallelisierung mit OpenMP

vorgelegt von

Niklas Wittmer

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6800916
Betreuer:	Dr. Michael Kuhn

Hamburg, 14. März 2018

Abstract

OpenMP ist ein Framework zur Parallelisierung von Programmen auf Shared-Memory-Systemen. Im Gegensatz zu anderen Standards und Frameworks ist OpenMP eine High-Level-Lösung, welche den Einstieg in die Parallelisierung vergleichsweise leicht macht, wodurch sich jedoch eine hohe Leistungsausbeute schwierig gestaltet.

OpenMP verwendet zur Parallelisierung Compiler-Direktiven, wodurch die Verwendung einer Code-Basis für sequentiellen und parallelen Code möglich ist. Neben Problemen, die OpenMP selbst mitbringt, bestehen auch die üblichen Probleme der Parallelisierung, wie Data Races.

Inhaltsverzeichnis

Listings

1. Einleitung

1.1. Motivation

Heutzutage verfügt jedes aktuelle System über mehrere Prozessorkerne, Prozessoren oder ist ein verteiltes System bestehend aus mehreren Maschinen. Um deren Rechenleistung auszureizen, ist es also wünschenswert, Programme zu parallelisieren. Dazu existieren heute mehrere Möglichkeiten.

Optimal ist beispielsweise die Idee, Programme automatisch zu parallelisieren. Dies bedeutet, dass der Compiler, der das Programm übersetzt, mithilfe eigener Heuristiken Programmcode erkennt, welcher parallelisierbar ist und das Programm entsprechend übersetzt. Der Programmierer hätte in diesem Fall nichts mehr zu tun. Er könnte wie gewohnt seriellen Code schreiben und diesen echt parallel ausführen.

Es existieren zwar beispielsweise entsprechende Mechanismen in gcc (siehe [Tea09]). Jedoch sind diese recht eingeschränkt. Das Problem liegt darin, zu erkennen, welche Programmteile parallelisiert werden dürfen, ohne dass das Programm verändert wird. Um einen deutlichen Speedup zu erreichen, ist diese Methode der Parallelisierung nicht geeignet.

Für eine effektive Parallelisierung existieren daher andere Möglichkeiten, zum Beispiel *MPI* und *Posix-Threads*. Abgesehen davon, dass diese unterschiedliche Probleme der Parallelisierung lösen, bringen sie eigene Schwierigkeiten mit.

MPI ist etwa für Systeme mit verteiltem Speicher konzipiert, auch wenn es theoretisch auch auf Systemen gemeinsamem Speichers funktioniert, ist dies nicht der Sinn von MPI.

Posix-Threads sind sehr weit verbreitet um Programme zu parallelisieren. Jedoch ist die Anwendungen - wie auch in MPI - relativ komplex. Code muss um viele Teile ergänzt und teilweise umgeschrieben werden. Dadurch ist die Pflege zweier Code-Basen notwendig, möchte man parallelisierte und sequentielle Versionen vergleichen.

1.2. Überblick

1.2.1. Was ist OpenMP

MPI und Posix-Threads erfordern zur Parallelisierung also einen recht hohen Aufwand. Es lassen sich jedoch deutlich bessere Ergebnisse erzielen als mit automatischer Paralleli-

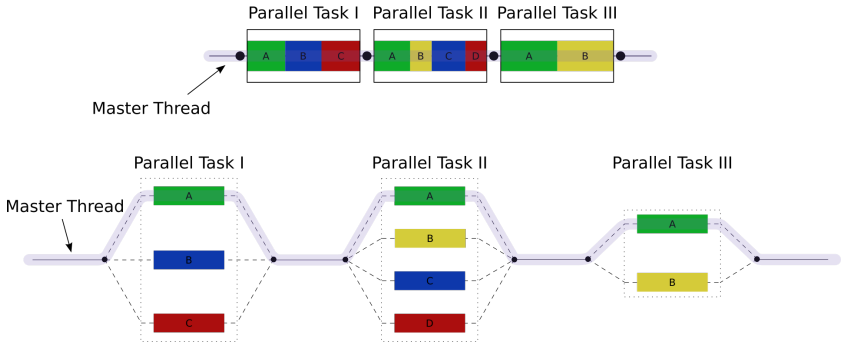

Abbildung 1.1.: Schema des Fork-Join-Modells [Com07]

sierung.

Als Kompromiss zwischen diesen Modellen ließe sich OpenMP verstehen. OpenMP steht für *Open Multi-Processing*. Es beschreibt eine Schnittstelle zur Programmierung paralleler Anwendungen auf Shared-Memory-Systemen. Wie bei MPI und Posix-Threads wird auch bei OpenMP lediglich der Standard vorgegeben. Die letztendliche Funktionsweise ist im Detail von der Implementation abhängig. Solche Implementationen existieren für die Sprachen Fortran sowie C und C++. Diese lassen sich mit allen gängigen Compilern anwenden. In diesem Bericht werde ich mich jedoch auf die Sprache C und den GCC-Compiler beschränken.

OpenMP Version 1.0 wurde im Jahr 1997 veröffentlicht. Damals diente die API noch lediglich als Werkzeug zur Parallelisierung von Schleifen. Heute ist sie um einiges mächtiger. Es lassen sich neben Schleifen ganze Codeblöcke durch Aufteilung von Arbeit auf mehrere Threads parallelisieren, Threads synchronisieren usw. Auch ist OpenMP geeignet für SIMD-Systeme. Das heißt, es lassen sich beispielsweise Vektorrechnungen damit durchführen.

Jedoch hat OpenMP auch seine Grenzen. Es wird einiges an Kontrolle abgegeben. Die Komplexität mag dadurch sinken, jedoch kann in Teilen auch die maximal erreichbare Performance darunter leiden. Von Fall zu Fall kann daher trotzdem die Verwendung von Posix-Threads sinnvoller sein, wenn OpenMP an seine Grenzen stößt. Da OpenMP ein Modell für Systeme mit gemeinsamem Speicher und nicht für verteilte Systeme ist, stellt es auch keinen Ersatz für MPI dar. Im Gegenteil wird in der Praxis tatsächlich häufig MPI in Verbindung mit OpenMP eingesetzt.

1.2.2. Threading Model

Das von OpenMP umgesetzte Modell zur Parallelisierung ist das sogenannte *Fork-Join-Modell*. Eine schematische Darstellung dieses Modells ist in Abbildung 1.1 zu sehen. Die grundlegende Idee ist, dass ein Master-Thread existiert, welcher sequentiell auszuführenden Code abarbeitet. Wird ein parallelisierbarer Bereich erreicht, so soll der

Master-Thread eine bestimmte Anzahl Sub-Threads erzeugen, welche jeweils einen Teil der parallel auszuführenden Tasks übernehmen. Am Ende dieses Bereiches werden die Threads wieder zerstört und nur der Master-Thread rechnet weiter.

2. Features

Hauptbestandteil von OpenMP sind die Compiler-Direktiven, mithilfe derer der Programmierer die Aufteilung von Aufgaben und zusätzliche Optionen angibt. Die Umsetzung der Anweisungen ist dann dem Compiler überlassen. Ein Vorteil dieser Funktionsweise ist, dass ein Programm sowohl mit als auch ohne OpenMP kompiliert werden kann, wodurch sich eine Code-Basis für sequentiellen als auch für parallelen Code nutzen lässt. Direktiven setzen sich grob aus Konstrukten als allgemeine Anweisungen und Klauseln zur näheren Bestimmung paralleler Regionen zusammen. Klauseln geben dabei beispielsweise an, wie viele Threads in einer Region verwendet werden sollen oder wie Daten eines Blockes aufzuteilen sind. Diese Anweisungen stehen zur Compilezeit fest.

Ist die Festlegung beispielsweise der Thread-Anzahl erst zur Laufzeit gewünscht, so lassen sich einige Laufzeitroutinen verwenden. Alternativ ist es auch möglich Umgebungsvariablen zu setzen, welche dann global für eine entsprechende Umgebung gelten.

Es ist wichtig, bei Verwendung von Laufzeitroutinen und Umgebungsvariablen auf die Präzedenz der Anweisungen zu achten. So würde die Verwendung einer Laufzeitroutine, die die Anzahl Threads eines Bereichs bestimmt, eine möglicherweise gesetzte Umgebungsvariable überschreiben. Compiler-Direktiven würden aber gegebenenfalls auch die Laufzeitroutinen ignorieren. Diese haben also die höchste Präzedenz.

2.1. Direktiven

In diesem Abschnitt möchte ich einige der wichtigsten Direktiven in OpenMP vorstellen und zeigen, worauf bei Verwendung dieser zu achten ist.

Direktiven werden aus der Phrase `#pragma omp`, dem Namen der Direktive und einigen optionalen Klauseln zusammengesetzt. `#pragma` leitet dabei eine Anweisung für den Präprozessor ein. Mit `omp` beginnt dann jede Direktive, die mit OpenMP parallelisiert werden soll. Listing 2.1 zeigt den grundlegenden Aufbau eines OpenMP-Pragmas.

```
1 #pragma omp directive-name [clause [clause ...]]
2 {
3 . . .
4 }
```

Listing 2.1: OpenMP-Syntax in C

2.1.1. Parallel regions

Um einen Programmbereich parallel auszuführen, wird ein Block mit der Direktive `parallel` definiert. Hierbei lässt sich über die Klausel `num_threads(n)` die Anzahl der Threads, in denen der Block ausgeführt werden soll auf `n` festlegen. Der folgende Block wird innerhalb geschweiften Klammern geschrieben. Listing 2.2 zeigt ein Beispiel.

```
1 void main() {
2 #pragma omp parallel num_threads(4)
3   {
4   printf("hello world!\n");
5   }
6 }
```

Listing 2.2: Definition einer *Parell Region* mit 4 Threads

Zu beachten hierbei ist, dass der Bereich, der von diesem Pragma umgeben ist, nun von allen Threads einmal ausgeführt wird. Die zu erwartende Ausgabe von Listing 2.2 entspräche dann der in Listing 2.3.

```
1 hello world!
2 hello world!
3 hello world!
4 hello world!
```

Listing 2.3: Ausgabe von Beispiel 2.2

Dies mag in einigen Fällen gewünscht sein. Jedoch ist es in den meisten Programmen wahrscheinlicher, dass dies zu unerwünschtem Verhalten führt. Im Beispiel hier ist es etwa fragwürdig, ob ein und dieselbe Ausgabe tatsächlich viermal ausgeführt werden soll.

2.1.2. Work Sharing

Ziel ist es also, die Arbeit, welche normalerweise von einem Thread ausgeführt wird, auf mehrere Threads aufzuteilen, sodass sich die Rechenzeit für den Codebereich verkürzt. Ein häufiger Anwendungsfall ist das Verteilen von Iterationen einer *For*-Schleife auf alle verfügbaren Threads.

```
1  void main() {
2    int i;
3    #pragma omp parallel
4    {
5      #pragma omp for
6      for(i=0;i<4;++i) {
7        printf("hello world! (%d)\n", i);
8      }
9    }
10 }
```

Listing 2.4: Work Sharing in einer *for-Schleife*

Listing 2.4 zeigt eine parallelisierte *for*-Schleife. Innerhalb der `parallel`-Direktive wird ein zweiter Block mit `#pragma omp for` deklariert. Die `for`-Direktive verlangt, dass der nachfolgende Code-Block eine *for*-Schleife sei. Die Iterationen werden dann gleichmäßig auf die verfügbaren Threads aufgeteilt. Das Ergebnis dieser Parallelisierung lässt sich in Listing 2.5 betrachten. Die Zahlen jeder Zeile geben an, um welche Iteration es sich bei der Ausgabe handelt. Die Reihenfolge, in der Aufgaben von den Threads abgearbeitet werden und anschließend eine mögliche Ausgabe erfolgt, ist jedoch nicht definiert. So ist es ebenso möglich und auch sehr wahrscheinlich, dass eine Ausgabe wie in Listing 2.6 erfolgt.

```
1 hello world! (0)
2 hello world! (1)
3 hello world! (2)
4 hello world! (3)
```

Listing 2.5: Ausgabe von Beispiel 2.4

```
1 hello world! (2)
2 hello world! (0)
3 hello world! (3)
4 hello world! (1)
```

Listing 2.6: Ausgabe von Beispiel 2.4

Die Iterationen werden im Normalfall gleichmäßig über alle Threads aufgeteilt. Dies bedeutet, dass bei n Iterationen und m Threads, jeder Thread $x = \frac{n}{m}$ Iterationen bearbeitet. Aufgeteilt wird *chunk*-weise. Ein Thread erhält also immer x zusammenhängende Iterationen. Der erste Thread bearbeitet dann zum Beispiel die Iterationen `0` bis `x-1`.

Unter Umständen entsteht dabei jedoch eine unfaire Lastverteilung, da die zu verrichtende Arbeit pro Iteration sehr unterschiedlich sein kann. Im Extremfall wäre denkbar, dass ein Thread nahezu nichts zu tun hat, während ein anderer Thread den Großteil der Arbeit einer Schleife übernehmen muss.

```
1 #pragma omp parallel for schedule (kind [, chunk size])
```

Listing 2.7: Syntax für das Scheduling im Work Sharing

Die Art, auf die Iterationen aufgeteilt werden, - das sogenannte *scheduling* - lässt sich daher bei Bedarf einstellen. Listing 2.7 zeigt, wie die Arbeitsaufteilung verändert werden kann. Die Option *static* entspricht dabei dem Normalfall. Der Parameter *chunk size* ist optional. Er gibt die (initiale) Größe der Blöcke an. Im `static`-Modus erhält jedoch jeder Thread maximal einen Block. Bei der Option `dynamic` wird jedem Thread anfangs ein Teil der Iterationen zur Bearbeitung übergeben. Ist ein Thread damit fertig, so erhält er den nächsten zu bearbeitenden Block. Dies soll dafür sorgen, dass die Last fairer über alle Threads verteilt wird. Zu beachten ist jedoch, dass dabei durch die dynamische Aufteilung Overhead entsteht. Um diesen zu reduzieren, empfiehlt es sich unter Umständen auf die Option `guided` auszuweichen. Diese funktioniert ähnlich wie `dynamic`. Jedoch ist hier `chunk size` zu Beginn relativ hoch und nimmt mit fortschreitender Iterationszahl ab. Dies soll bezwecken, dass alle Threads durchgehend ausgelastet sind, die Häufigkeit, mit der neue Chunks zugewiesen werden müssen jedoch im Vergleich zu *dynamic* geringer ist.

Schließlich gibt es noch die Option *auto*, welche zur Laufzeit entscheidet, welches Scheduling am besten zur Situation passt. Wie gut dies in der Praxis funktioniert, lässt sich pauschal schwer sagen.

Grundsätzlich ist es von Fall zu Fall unterschiedlich, welches Scheduling die beste Performance bringt. Häufig funktioniert `static` bereits relativ gut. Für `dynamic` benötigt man meist schon sehr unausgeglichene Schleifen, damit sich hier ein Leistungsschub bemerkbar macht.

2.1.3. Sections

Es ist auch möglich Arbeit außerhalb von *for*-Schleifen aufzuteilen. Dies geschieht mittels der Direktive `sections`. Ein Block, der mit `sections` gekennzeichnet ist, besteht aus mehreren Unterblöcken, welche wiederum mit der Direktive `section` zu kennzeichnen sind. Jeder `section`-Bereich ist von einem Thread auszuführen. Die Syntax dieses Modells

ist beispielhaft in Listing 2.8 zu sehen. Es ist zu beachten, dass kein Code innerhalb eines `Sections`-Blockes außerhalb einer `Section` stehen darf.

```
1  #pragma omp parallel
2  {
3    #pragma omp sections
4    {
5        #pragma omp section
6        {
7            do_something();
8        }
9        #pragma omp section
10       {
11           do_something_else();
12       }
13   }
14 }
```

Listing 2.8: Arbeitsteilung mit Sections

2.1.4. Synchronisation

Die meisten Direktiven in OpenMP besitzen *implizite Barrieren*. Das bedeutet, dass am Ende eines solchen Blockes alle Threads aufeinander warten, bevor sie terminieren bzw. den nächsten Block Code bearbeiten. In vielen Fällen ist dies ohnehin erwünscht, um etwa Race Conditions zu vermeiden. Es gibt jedoch auch Situationen, in denen der Programmierer möchte, dass Threads, die ihre Arbeit bereits erledigt haben, mit dem nächsten Block fortfahren. Der Programmierer sollte in dem Fall sicher sein, dass keine Race Conditions auftreten können.

```
1  int c = 0, int i;
2  #pragma omp parallel
3  {
4      #pragma omp for nowait
5      for(i=0;i<4;++i) {
6          c++;
7      }
8      printf("c ohne Barrier: %d", c);
9      #pragma omp for
10     for(i=0;i<4;++i) {
11         c++;
12     } // implizites barrier
13     printf("c mit Barrier: %d", c);
14 }
```

Listing 2.9: Verwendung der `no wait`-Klausel

Zum Abschalten einer Barrier dient die `no wait`-Klausel. Sie bewirkt folglich, dass ein Thread, der seinen Teil der Arbeit innerhalb eines Blockes erledigt hat, zum nächsten übergeht. Wie sich die Verwendung von `no wait` auf eine *for*-Schleife auswirkt, soll mit Listing 2.9 dargestellt werden. Zu sehen sind 2 Schleifendurchläufe, in denen jeweils die Variable `c` inkrementiert wird. Nach der ersten Schleife warten die Threads nicht aufeinander, sondern gehen sofort zur nächsten Anweisung über. Jeder Thread gibt einmal den Wert in `c` aus.

```
1 c ohne Barrier: 3
2 c ohne Barrier: 4
3 c ohne Barrier: 2
4 c ohne Barrier: 1
5 c mit Barrier: 8
6 c mit Barrier: 8
7 c mit Barrier: 8
8 c mit Barrier: 8
```

Listing 2.10: Ausgabe von Beispiel 2.4

Nach der zweiten Schleife werden die Threads synchronisiert. Anschließend gibt wiederum jeder Thread den Wert in `c` aus. Wie in Listing 2.10 zu sehen, ist die Ausgabe nach der ersten Schleife inkonsistent im Vergleich zu einer Ausgabe, die nach sequentieller Ausführung zu erwarten wäre. Dies ist ein Beispiel für Code, welcher nicht threadsicher

ist. Bei Verwendung von mehr als 4 Threads - die hier impliziert wurden - wäre es sogar wahrscheinlich, dass die Zahlen 1 bis 4 ungleich oft gedruckt werden. Dies hängt davon ab, welchen Wert `c` zur Zeit der Ausgabe eines Threads hält.

Soll hingegen an einer Stelle im Code sichergestellt werden, dass alle Threads aufeinander warten, so wird die Direktive `#pragma omp barrier` verwendet. Diese gibt explizit an, dass kein Thread ab diesem Punkt weiter rechnen darf, bevor jeder Thread ihn erreicht hat.

In Situationen, in denen es sich nicht vermeiden lässt, unterschiedliche Threads auf dieselben Variablen zugreifen zu lassen, können Abschnitte mit `critical` oder `atomic` deklariert werden. Mittels dieser Konstrukte wird sichergestellt, dass jeweils nur ein Thread zur Zeit eine Aufgabe bearbeitet.

Der Einsatz dieser beiden Konstrukte sollte möglichst sparsam erfolgen. Die kritischen Bereiche sollten außerdem möglichst klein gehalten werden, da durch das Warten der Threads auf die jeweilige Ressource Overhead erzeugt, was sich auf die gesamte Performance eines Programms niederschlägt.

Weitere Konstruke zur Synchronisation sind `master` und `single`. Diese kennzeichnen Bereiche, welche nur von einem Thread ausgeführt werden sollen. Dies ist beispielsweise bei der Initialisierung von Variablen sinnvoll oder der Ausgabe. Der Unterschied zwischen `master` und `single` liegt darin, dass ersteres immer vom Master-Thread ausgeführt wird, während alle anderen Threads diesen Bereich überspringen und weiter rechnen. Bei der Verwendung von `single` wird ein beliebiger Thread verwendet und alle anderen Threads warten auf diesen.

2.1.5. Data Sharing

Standardmäßig haben alle Threads innerhalb einer parallelen Region Zugriff auf alle Variablen, die in dessen Gültigkeitsbereich liegen. Dass dies in einigen Situationen zu Data Races und unerwünschtem Verhalten führen kann, haben wir bereits in einigen vorangegangen Beispielen gesehen.

Dabei wurden auch Zählvariablen in *for*-Schleifen verwendet. Diese werden von OpenMP implizit als *private* Variable deklariert. Dies bedeutet, dass jeder Thread eine eigene Kopie dieser Variablen besitzt. Dies stellt sicher, dass die Zahl der durchgeführten Iterationen konsistent bleibt, auch wenn parallelisiert wird.

Diese Regel gilt jedoch nur für die Zählvariablen von *for*-Schleifen. Und auch hier gibt es Einschränkungen. Wird ein Bereich wie in Listing 2.11 deklariert, dann ist nicht vorherbestimmt, welche Werte die Variable `j` annehmen wird. Es werden alle Threads auf dieselbe Variable zugreifen.

```
1 #pragma omp parallel for
2 for(int i=0;i<4;++i) {
3     for(int j=0;j<4;++j) {
4         do_something(i,j);
5     }
6 }
```

Listing 2.11: Vorsicht bei verschachtelten *for*-Schleifen

Um das zu vermeiden, ist es möglich, festzulegen, ob Variablen als private Kopien oder gemeinsame Variablen verwendet werden. Die Konstrukte dafür lauten `private` und `shared` respektive. In Listing 2.12 wird die Variable `i` als privat und `j` als gemeinsam deklariert. Im folgenden Block werden beide Variablen von je zwei Threads inkrementiert und ausgegeben.

```
1 void main() {
2     int i,j = 0;
3     #pragma omp parallel private(i) shared(j) num_threads(2)
4     {
5         i=0;
6         printf("i: %d\n", ++i);
7         printf("j: %d \n", ++j);
8     }
9 }
```

Listing 2.12: shared- und private-Deklarationen

Wie eine mögliche Ausgabe dazu aussieht, ist in Listing 2.13 zu sehen. Da `i` privat ist, wird die Ausgabe bei jeder Ausführung zweimal den Wert `1` drucken. Bei `j` ist dies nicht sicher; hier könnte die Ausgabe wie für `i` aussehen, oder aber einer der beiden Werte könnte `2` betragen.

Das `private`-Konstrukt impliziert, dass eine Variable uninitialisiert verwendet wird. Soll eine Variable aber ihren Wert behalten, welchen Sie vor Beginn des parallelen Blockes hatte, so muss stattdessen das Konstrukt `firstprivate` verwendet werden.

```
1 i: 1
2 i: 1
3 j: 1
4 j: 2
```

Listing 2.13: Ergebnis der shared- und private-Deklarationen

Andersherum behalten private Variablen auch nicht ihren Wert, wenn der Block wieder verlassen wird. Zumindest für *for*-Blöcke lässt sich dies umgehen, indem das `lastprivate`-Konstrukt verwendet wird. Dadurch erhält eine Variable jenen Wert, den sie nach Ausführung des Blockes als sequentiellen Code auch hätte. In Listing 2.14 wird eine als `lastprivate` deklarierte Zählvariable in jeder Iteration ausgegeben und nach Verlassen der Schleife noch einmal gedruckt. Die Ausgabe in der Schleife besteht wie zu erwarten aus den Zahlen `0` bis `9` in beliebiger Reihenfolge. Die anschließende Ausgabe gibt in diesem Fall immer `10` zurück. Wäre `i` dagegen `private`, hätte die Variable den Wert `0`, entsprechend der Zuweisung vor dem parallelen Block.

```
1 int i = 0;
2 #pragma omp parallel for lastprivate(i)
3 for(i=0;i<10;++i) {
4     printf("%d",++i);
5 }
6 printf("\n%d",i);
```

Listing 2.14: Verwendung von lastprivate

Es ist ebenfalls möglich für einen Block einen Defaultwert zu setzen, den alle Variablen erhalten, für die der Scope nicht explizit angegeben wurde. Dies geschieht mittels `default(private|shared|none)`. Der Wert `none` Gibt dabei an, dass für keine Variable ein impliziter Wert gesetzt wird. Es muss dann also jede Variable implizit einer der beiden Kategorien zugeordnet werden.

Die Klausel `reduction(op:var)` setzt die Variable `var` auf privat und reduziert sie am Ende eines Blockes mithilfe des Operators `op`. Dies könnte etwa eine einfache Addition oder andere Grundrechenart sein. Aber es lässt auch das Maximum mit der Angabe von `max` berechnen.

2.1.6. Weitere Konstruke

Abschließend möchte ich hier noch einige weitere Konstrukte nennen.

Um eine Parallelisierung nur unter bestimmten Umständen vorzunehmen, zum Beispiel, wenn die Problemgröße ausreichend hoch ist, bietet sich der Einsatz von `if(bool expr)`.

Ein Beispiel dazu ist in Listing 2.15 zu sehen. Hier wird das jeweilige Pragma nur dann berücksichtigt, wenn der Ausdruck `expr` zu `True` ausgewertet wird.

```
1 void do_something(int n) {
2     #pragma omp parallel for if(n > 20)
3     for(int i=0;i<n;++i) {
4         do_stuff(..);
5     }
6 }
```

Listing 2.15: Die *for*-Schleife wird nur bei ausreichend großem `n` parallelisiert

Es kann auch Situationen geben, in denen es nicht erforderlich ist, einen Block bis zum Ende auszuführen. Wenn beispielsweise in einer Menge ein Element gesucht wird, ist es sinnlos weiterzusuchen, sobald das Element gefunden wurde. Für eben solche Zwecke bietet OpenMP das *Cancellation*-Konstrukt an. Eine Stelle im Code, an der die Threads aus einem Block ausbrechen sollen, wird mit `#pragma omp cancellation point` markiert. Hier prüft jeder Thread, ob `Cancellation` aktiviert wurde und bricht, falls ja, aus. Aktiviert wird diese mittels `#pragma omp cancel`. Erreicht ein Thread eine solche Stelle, wird festgelegt, dass jeder Thread bei der nächsten Möglichkeit ausbricht. Neben `cancellation point` wird die Ausbruchsbedingung auch bei `#pragma omp cancel` selbst sowie an allen Barrieren geprüft.
Für die Korrektheit des Programms ist letztendlich der Programmierer verantwortlich.

Um den Code etwas kürzer schreiben zu können, bietet OpenMP für einige häufig verwendete Direktiven Kurzschreibweisen an. So lässt sich etwa eine *for*-Schleife mit dem Pragma `#pragma omp parallel for` parallelisieren, ohne die gesamte Schleife von einem parallelen Block umgeben zu müssen. Ebenso lässt sich so ein `#pragma omp parallel sections` Bereich erzeugen.

2.2. Laufzeitroutinen

Laufzeitroutinen sind nützlich, wenn es nötig ist, gewisse Dinge wie etwa die Zahl der Threads erst zur Laufzeit zu bestimmen. Die Verwendung setzt das Einbinden des OpenMP-Headers voraus. Unter Linux in `C` lautet der entsprechende Befehl dazu `#include <omp.h>`. Hier nun eine kleine Auswahl an Laufzeitroutinen:

- `omp_get_thread_num()` gibt die ID eines Threads zurück.
- `omp_set_num_threads()` legt die Zahl der Threads eines parallelen Blockes fest. Mit `omp_get_num_threads()` wird entsprechend die Anzahl aktiver Threads ausgegeben.

- `omp_in_parallel()` prüft, ob sich das Programm zur Zeit innerhalb eines parallelen Blockes befindet.
- `omp_set_schedule()` legt die Art des Schedulings, wie es etwa für *for*-Schleifen benötigt wird, fest. Mit `omp_get_schedule()` wird entsprechend der gesetzte Schedule ermittelt.

2.3. Umgebungsvariablen

Es ist auch möglich, gewisse Parameter in einer Ausführungsumgebung zu setzen oder Umgebungsvariablen für einen bestimmten Aufruf zu setzen. So lässt sich mittels `export OMP_NUM_THREADS 8` die Threadanzahl in einer Ausführungsumgebung bestimmen. Verwendet man stattdessen `OMP_NUM_THREADS=8 ./executable`, wird für diese Ausführung der Zahl der Threads festgelegt. Mit `OMP_THREAD_LIMIT 10` ist es möglich die Threadanzahl zu begrenzen. Dies ist in etwa nützlich, wenn häufiger verschachtelte Parallelisierungen in einem Programm auftreten.

Sämtliche Features sind in der offiziellen Dokumentation [ARB15b] zu finden. Eine schnelle Übersicht bietet [ARB15a].

2.4. Präzedenz

Gegeben sei der Code aus Listing 2.16. Dieser verwendet eine Laufzeitroutine, um die Zahl der Threads auf 2 festzulegen. Das folgende Pragma legt die Zahl der Threads jedoch auf 4 fest unter der Bedingung, dass eine Variable `n` größer 2 ist.

```
1 omp_set_num_threads(2);
2 #pragma omp parallel if(n > 2) num_threads(4)
3 {
4     printf("hello world!");
5 }
```

Listing 2.16: Code von omp-test

Der Code soll nun wie folgt ausgeführt werden: `OMP_NUM_THREADS=8 ./omp-test`.

Hier wird also an drei verschiedenen Stellen im Code Einfluss darauf genommen, wie hoch die Zahl der zu verwendenden Threads sein soll.

Tatsächlich würde hier nun vier mal `hello world!` ausgegeben werden. Jedoch nur unter der Bedingung, dass `n` tatsächlich größer 2 ist. OpenMP legt die Präzedenz dieser Statements fest[CJP08]:

1. `IF`-Klausel
2. andere Klauseln einer Direktive
3. Laufzeitroutinen
4. Umgebungsvariablen

Wird ein Wert wie die Threadanzahl nicht explizit bestimmt, so wird ein Defaultwert verwendet. Dieser entspricht meist der Anzahl verfügbarer Rechenkerne.

3. Kompilierung

3.1. Voraussetzungen

Um OpenMP verwenden zu können, ist ein Compiler notwendig, welcher die API implementiert. Heute trifft dies auf jeden gängigen Compiler zu, insbesondere auch *clang* und *gcc*. Falls nötig, muss die Laufzeitbiblithek eingebunden werden. Schließlich ist es notwendig dem Compiler explizit mitzuteilen, dass mit OpenMP parallelisiert werden soll. Bei gcc ist dies zum Beispiel mit dem Flag `-fopenmp` möglich. Die einfachste Möglichkeit, einen Code zu kompilieren, lautet demnach `gcc -fopenmp code.c`.

3.2. Übersetzung

OpenMP-Pragmas werden im Präprozessor des Compilers interpretiert. Dort wird zum Teil Code ersetzt oder erweitert. Letztendlich werden Posix-Threads zur Parallelisierung verwendet.

```
1  int main (void)
2  {
3      int a,b,c;
4
5      #pragma omp parallel private(c)
6      {
7          do_something(a,b,c);
8      }
9
10     return 0;
11 }
```

Listing 3.1: Zu kompilierender C-Code mit parallelem Block

Listing 3.1 zeigt ein kurzes Code-Beispiel, welches kompiliert werden soll. Es werden drei Variablen deklariert `a,b,c`. Diese werden als Parameter an die Funktion `do_something()` übergeben. Diese Funktion befindet sich innerhalb eines parallelen Blockes. Dabei wurde `c` als privat deklariert. Dies bedeutet, dass jeder Thread seine eigene lokale Kopie verwendet.

```
1  static void __omp_func_0(void **__ompc_args) {
2      int *_pp_b,*_pp_a, _p_c;
3
4      _pp_b = (int *) (*__ompc_args);
5      _pp_a = (int *) (*(__ompc_args+1));
6      do_something(*_pp_a,*_pp_b,_p_c);
7  }
8
9  int _ompc_main(void) {
10     int a,b,c;
11     void *__ompc_argv[2];
12
13     *(__ompc_argv) = (void *) (&b);
14     *(__ompc_argv+1) = (void *) (&a);
15
16     _ompc_do_parallel(__ompc_func_0, __ompc_argv);
17     . . .
18 }
```

Listing 3.2: Der Code aus Listing 3.1 nach dem Preprocessing

Listing 3.2 zeigt, wie der Code aus Listing 3.1 nach dem Preprocessing in etwa aussehen würde. Am auffälligsten dürfte zunächst sein, dass der gesamte parallele Block durch eine ausgelagerte Funktion ersetzt wurde. Dieser Funktion werden Pointer auf die Variablen `a` und `b` als Parameter übergeben. Zu beachten ist, dass `c` hier nicht mit übergeben wird, da diese Variable als privat deklariert wurde. So wird für `a` und `b` auf den entsprechenden Speicherbereich zugegriffen, während `c` als nicht initialisierte Variable in der ausgelagerten Funktion verwendet wird.

3.3. Unified Code

OpenMP ermöglicht es, dieselbe Code-Basis für sequentielle wie parallele Programme zu verwenden. Dies erleichtert das Testen von Programmen, die OpenMP verwenden, erheblich. Soll ein mit OpenMP parallelisiertes Programm sequentiell kompiliert werden, dann ist es in vielen Fällen bereits ausreichend, das entsprechende Compiler-Flag nicht zu setzen.

Für den Fall, dass eine oder mehrere Laufzeitroutinen verwendet werden, ist es notwendig, diese Routinen für die Kompilierung der sequentiellen Variante statisch zu definieren, da trotz Einbindung der Laufzeitbibliothek die Kompilierung sonst fehlschlagen würde. Ein Beispiel dazu ist in Listing 3.3 zu sehen.

```
1  #ifdef _OPENMP
2      #include <omp.h>
3  #else
4      #define omp_get_thread_num() (0)
5  #endif
6
7  int main() {
8      #pragma omp parallel num_threads(2)
9      {
10         printf("%d\n", omp_get_thread_num());
11     }
12
13     return 0;
14 }
```

Listing 3.3: Beispiel für Unified Code

Dort wird die Routine `omp_get_thread_num()` für den Fall, dass das Flag nicht gesetzt wurde, statisch als `0` definiert. Das sequentielle Programm gibt dann beim Aufruf dieser Funktion immer 0 zurück, während die OpenMP-Variante entsprechend der Anzahl an Threads deren IDs ausgibt.

4. Peformance und Optimierung

Um die Performance eines Programms nicht negativ zu beeinflussen, empfiehlt es sich, einige paar Punkte zu beachten.

4.1. Speicherzugriff

OpenMP bietet Parallelisierung basierend auf gemeinsamem Speicher. Standardmäßig haben daher alle Threads eines Prozesses Zugriff auf den gleichen Speicherbereich. Dabei kann es zu unterschiedlichen Problemen kommen, welche Auswirkung auf die Performance eines Programms haben.

False Sharing

Ein Problem, das beim Teilen von Speicher auftritt, ist das sogenannte *False Sharing*. Dabei haben Threads jeweils Zugriff auf einen Teil eines zusammenhängenden Speicherbereiches. Es ist etwa möglich, dass in einer Schleife jeder Thread eine Zeile eines zweidimensionalen Arrays bearbeitet. Jeder Thread hält dann den Teil des Arrays, für den er zuständig ist bei sich im Cache vor. Der Mechanismus der Cache-Kohärenz sorgt dafür, dass die Kopien, welche die Threads lokal in ihrem Cache vorhalten über alle Caches hinweg gleich sind, wenn sie vom selben Block im Hauptspeicher stammen. So kann es passieren, dass zwei Threads auf unterschiedliche Zeilen des Arrays zugreifen, die sich jedoch in derselben Cache-Line befinden. Überschreibt nun einer der beiden Threads den von ihm behandelten Bereich, so müssen alle Threads, welche dieselbe Cache-Line halten, ihre Kopie aktualisieren, auch wenn sie niemals auf denselben Bereich zugreifen werden.

Um dies zu verhindern, ist es notwendig, genaueres über die Größen von Cache-Lines zu wissen. In der Regel ist dies eine Potenz von *2*, etwa 2^6 Bytes. Dies hängt jedoch vom Prozessor eines Systems ab.

Ist die Größe von Cache-Lines eines Systems bekannt, dann ist es möglich, die Datenaufteilung auf die Threads so zu gestalten, dass die Speicherbereiche sich nicht überschneiden. Beträgt die Cache-Line-Größe zum Beispiel 64 Byte, ist es sinnvoll, jedem Thread einen Anteil an Daten zuzuteilen, dessen Größe einem vielfachen von 64 Bytes entspricht.

Lastverteilung

Bei der Datenaufteilung ist neben Problemen, die wegen Cache-Kohärenz entstehen, auch auf eine möglichst ausgeglichene Auslastung der Threads zu achten. Ist die Last ungleich verteilt, müssen einige Threads viel Zeit mit Warten verbringen, in der sie keine produktiven Aufgaben haben. Währenddessen müssen andere Threads womöglich den Großteil der Arbeit erledigen, wodurch die Bearbeitung eines Problems länger dauert als nötig.

Wie bereits weiter oben besprochen, kann hier ein alternatives Scheduling helfen, alle Threads gleichmäßig beschäftigt zu halten, damit eine Aufgabe schneller erledigt werden kann.

Overhead

Overhead ist der Teil eines Programms, welcher nicht zur eigentlichen Bearbeitung eines Problems dient, sondern lediglich die korrekte Ausführung sicherstellt. Es ist grundsätzlich wünschenswert, diesen Overhead so klein wie möglich zu halten, da er sich negativ auf Laufzeit oder Speicherplatzbedarf auswirkt.

Ich möchte hier drei verschiedene Arten von Overhead betrachten, welche in direktem Zusammenhang zur Parallelisierung stehen.

Sequentieller Overhead ist der Teil eines Programms, welcher sich nicht parallelisieren lässt, also sequentiell ausgeführt werden muss. In der Regel besitzt jeder Code zumindest einen Teil sequentiellen Codes, der nur von einem oder nur einem Bruchteil aller verfügbaren Threads bearbeitet werden kann.

Parallelisierungs-Overhead: Soll ein Teil eines Programms parallelisiert werden, so sind dafür einige vorbereitende Schritte notwendig, welche das Programm und die Daten so aufbereiten, dass sie von mehreren Threads bearbeitet werden können. Es mag daher nicht immer sinnvoll sein, einen Bereich zu parallelisieren, wenn dabei zu viel Overhead entsteht, als dass durch die Verwendung mehrerer Threads einen Leistungsschub brächte. Darüber hinaus reichen für einen Bereich vielleicht auch schon zwei Threads aus. Nur deswegen 24 Threads zu verwenden, weil es technisch problemlos möglich wäre, ist nicht unbedingt sinnvoll, da alle diese Threads auch gestartet werden müssen. Die Anzahl Threads sollte also möglichst zur Problemgröße passen.

Synchronisations-Overhead entsteht, weil Threads durch Verwendung von Konstrukten wie `barrier` aufeinander warten müssen. Auch implizite Barrieren tragen dazu bei.

4.1.1. Häufige Fehler

Race Conditions

Race Conditions können leicht auftreten, wenn mehrere Threads auf denselben Speicherbereich zugreifen sollen. Bei der Verwendung von OpenMP sind daher folgende Dinge zu beachten. [1]

- Sind die verwendeten Bibliotheken threadsafe?

  Auf die meisten Bibliotheken dürfte dies heute zutreffen. Trotzdem ist dies besonders bei Bibliotheken, die nicht zum Standard gehören, zu überprüfen.

- Fehlerhafte Verwendung des `no wait`-Konstruktes.

  Um die Laufzeit eines Programms zu verbessern, empfiehlt es sich zwar, überall dort, wo es möglich ist, Code-Teile zu parallelisieren. Jedoch ist darauf zu achten, dass parallel laufende Threads nicht (schreibend) auf denselben Speicherbereich zugreifen müssen. Dies kann leicht passieren, beispielsweise bei der Verwendung zu Zählvariablen oder solchen, die nur zum Speichern temporärer Zwischenwerte verwendet werden.

- Wo existieren implizite Barriers und wo nicht?

  Die meisten Konstrukte in OpenMP verwenden implizite Barriers. Das bedeutet, dass Threads automatisch aufeinander warten werden. Umso wichtiger ist es jedoch, zu wissen, wo dies nicht der Fall ist. Bei der Verwendung des `master`-Konstruktes werden Threads nicht auf den Master warten, sondern weiter rechnen.

Gültigkeitsbereich von Variablen

Per Default erhalten die meisten Variablen in parallelen Regionen den Wert *shared*. Hier ist wie oben beschrieben auf Data Races zu achten. Für Variablen, welche als *private* deklariert werden, ist zu beachten, mit welchen Werten Variablen beim Betreten paralleler Regionen initialisiert werden. Auch beim Verlassen paralleler Blöcke können Variablen, je nach Zuweisung unterschiedliche Werte haben.

4.1.2. Best Practices

Zur Verbesserung der Laufzeit und der Vermeidung zumindest häufiger Fehler, empfiehlt es sich, ein paar Richtlinien zu befolgen.

- Optimierung der Barrier-Nutzung

[1] [CJP08, p. 243]

Barrieren sollten überall da verwendet werden, wo sie zwingend erforderlich sind, um etwa Data Races zu vermeiden. Jedoch sollte eine übermäßige Nutzung ebenfalls vermieden werden, da jedes Barrier-Konstrukt Synchronisations-Overhead mit sich bringt, welcher sich negativ auf die Laufzeit auswirkt.

- Vermeidung großer *Critical Regions*

  Critical Regions sind solche, in denen sich zu jedem Zeitpunkt maximal ein Thread befinden kann. Der Einsatz kann sinnvoll sein, wenn gemeinsame Speicherbereiche aktualisiert werden müssen. Da diese Regionen jedoch sequentiell ausgeführt werden und Threads aufeinander warten müssen, sollten Critical Regions so klein wie möglich gehalten werden. Dies vermeidet Leerlauf von Threads.

- Parallele Regionen so groß wie möglich machen.

  Jedes Mal, wenn eine parallele Region betreten wird, müssen Threads zugewiesen und vorher - je nach Implementation - sogar erzeugt werden. Am Ende eine parallelen Region müssen diese wieder beendet (und gegebenenfalls zerstört) werden. Zusätzlich müssen Variablen an Anfang und Ende der Regionen die korrekten Werte zugewiesen werden. All dies fällt unter Parallelisierungs-Overhead und die Häufigkeit, mit der diese Operationen durchgeführt werden müssen, sollte so gering wie möglich gehalten werden. Es ist daher sinnvoll, zu prüfen, ob mehrere parallele Regionen nicht zu einer großen zusammengefasst werden können.

4.2. Anwendungsbeispiel

In diesem Kapitel möchte ich ein simples Beispiel vorstellen, an welchem erkennbar ist, wie eine einfache Parallelisierung einer *for*-Schleife die Laufzeit eines Programms deutlich verbessern kann.

4.2.1. Parallelisierung einer Matrizenmultiplikation

Listing B.1 zeigt eine Funktion `mxn()`,die zwei Matrizen multipliziert. Der vorliegende Code erzeugt dazu zwei Matrizen und füllt diese mit zufälligen Werten. Anschließend wird die benötigte Zeit der Multiplikation in Mikrosekunden ausgegeben.

Die Funktion `mxn()` besteht aus drei verschachtelten *for*-Schleifen. Diese befinden sich in einem parallelen Block, wobei die Zählvariablen jeweils privat und alle anderen gemeinsame Variablen sind. Die äußerste *for*-Schleife wurde außerdem mit dem `for`-Konstrukt von OpenMP versehen.

Das Programm wurde einmal durch `gcc -fopenmp -Wall -O3 mxn.c -o mxn-omp` für die parallelisierte und mit `gcc -Wall -O3 mxn.c -o mxn-seq` für die sequentielle Variante kompiliert.

4.2.2. Analyse

Zunächst einige hier verwendete Definitionen. [CJP08, p. 139]

Speedup: $S_n = t_{seq} \div t_n$, wobei n die Zahl der verwendeten Kerne ist und t die *Wallclock Time*, also die vergangene Zeit bei Ausführung des Programms.

Effizienz: $E_n = S_n \div n$

Das Programm wurde mittels des Skriptes aus Listing B.2 ausgeführt und gemessen.

Version	#CPUs	Wallclock Time	CPU Time	Speedup	Efficiency (%)
Sequentiell	1	49,24	49,18	1	100
Parallel	1	49,69	49,63	0,99	99,08
	2	33,2	66,29	1,48	74,15
	3	23,58	70,5	2,09	69,6
	4	18,13	72,3	2,72	67,9
	5	14,73	73,25	3,34	66,84
	6	12,59	74,83	3,91	65,18
	7	10,72	74,33	4,59	65,63
	8	9,39	74,79	5,24	65,56
	9	10,33	75,61	4,77	52,95
	10	9,89	75,46	4,98	49,78
	11	10,08	75,06	4,88	44,4
	12	9,98	76,06	4,94	41,13

Tabelle 4.1.: Ergebnisse der Laufzeitmessung von mxn.c

Tabelle 4.1 zeigt die Messergebnisse der Laufzeit des Programms bei unterschiedlicher Threadanzahl. In der ersten Zeile ist die Laufzeit der sequentiellen Variante zu sehen. Auffällig ist, dass die parallele Variante bei Verwendung eines Threads langsamer lief als die sequentielle. Dies ist dadurch zu erklären, dass hier durch Parallelisierungs-Overhead entsteht. Da jedoch lediglich ein Thread parallelisiert werden soll, wird die gesamte Berechnung trotzdem sequentiell ausgeführt. Das Ergebnis ist ein Speedup von $0,99$.

Der maximale Speedup liegt bei $5,24$. Insgesamt kommt der Speedup nicht über knapp 5 hinaus, trotz Verwendung von bis zu 12 Kernen. Eine Zusammenfassung der verwendeten Hardware ist in Listing A.1 aufgelistet.

Die Effizienz nimmt etwa monoton ab. Der erste große Sprung ist zwischen zwei und einem Thread erkennbar. Hier nimmt die Effizienz des Programms um 25% ab. Ein weiterer großer Sprung liegt zwischen acht und neun Threads. Dieser ist dadurch zu erklären, dass sich die Laufzeit nicht mehr verbessert, tendenziell sogar verschlechtert. Würden noch mehr Threads verwendet werden, würde sich dieser Trend vermutlich fortsetzen. Zumindest sind keine besseren Laufzeiten mehr zu erwarten.

Ein maximaler Speedup von 5 ist bei 12 Kernen alles andere als optimal. Es stellt sich nun also die Frage, wo hier Performanceverluste entstehen. Ein offensichtlicher Fehler, der jedoch nichts mit der Parallelisierung an sich zu tun hat, ist die Art des Speicherzugriffs. In der Programmiersprache *C* werden Arrays zeilenweise im Speicher abgelegt. Das Programm greift jedoch spaltenweise auf die Matrix *b* zu. Eine Folge sind viele Cache Misses, die sich vermeiden ließen, indem die Matrix transponiert würde, um auch hier den Speicherzugriff zeilenweise zu gestalten.

Da dies jedoch auch die sequentielle Variante betrifft, kann es nicht der Grund für den schlechten Speedup sein.

Eine weitere Möglichkeit ist das Scheduling. Diese entsteht durch ungleich verteilte Last. Jedoch sind im vorliegenden Beispiel alle Threads konstant beschäftigt, da sie auf allen Elementen die gleichen Operationen durchführen. Dynamisches Scheduling dürfte hier also keine Leistungsschübe bringen, könnte die Laufzeit sogar verschlechtern.

Tatsächlich ist es in der Realität so, dass die Leistungsausbeute bei der Parallelisierung von *for*-Schleifen relativ gering ist. Teile des Programms laufen weiterhin sequentiell, wodurch der maximale Speedup dem Gesetz von Amdahl folgend stark beschränkt wird.

5. Zusammenfassung

In dieser Ausarbeitung habe ich die wichtigsten Features von OpenMP vorgestellt und darauf hingewiesen, worauf bei der Verwendung zu achten ist. Anschließend stellte ich anhand eines kurzen Beispieles dar, wie mit wenig Aufwand bereits ein guter Speedup erreicht werden kann; es schien jedoch darüberhinaus nicht trivial zu sein, diesen maximal zu optimieren.

5.1. Fazit

OpenMP bietet einen leichten Einstieg in die parallele Programmierung. Es ist mit relativ wenig Mitteln möglich, die Laufzeit eines Programmes gegenüber einer sequentiellen Ausführung deutlich zu verbessern, ohne sich viele Gedanken über die Datenaufteilung oder die Architektur der verfügbaren Hardware machen zu müssen. Es reicht etwa eine Zeile Code aus, um die Abarbeitung einer *for*-Schleife zu parallelisieren.

Jedoch kann es kein absoluter Ersatz für andere Techniken sein, alleine deshalb, weil OpenMP auf Shared-Memory-Systeme beschränkt ist. Hier kann OpenMP MPI nicht ersetzen. In der Praxis ist es tatsächlich häufig üblich, diese beiden Techniken zu kombinieren, um ein Programm über viele Rechnerknoten zu verteilen, welche wiederum ihre Berechnungen mit OpenMP parallelisieren.

Zudem kann sich auch mit OpenMP eine hohe Optimierung als schwierig gestalten. Es wird ein gewisses Maß an Kontrolle abgegeben, um schnell und einfach höhere Geschwindigkeiten zu erreichen. Um jedoch ein Höchstmaß an Optimierung zu erhalten, könnte der Einsatz von Pthreads nötig sein.

Letzten Endes ist es von der Situation abhängig, ob etwa OpenMP oder eine Low Level-Option zur Parallelisierung verwendet werden sollte.

A. Hardware-Informationen

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             12
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU  X5650 @ 2.67GHz
Stepping:              2
CPU MHz:               2666.760
BogoMIPS:              5333.52
Virtualization:        VT-x
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-11
Flags:                 fpu vme de pse tsc msr pae mce
cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr
sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc
arch_perfmon rep_good nopl pni pclmulqdq vmx ssse3 cx16
pcid sse4_1 sse4_2  x2apic popcnt tsc_deadline_timer aes
hypervisor lahf_lm kaiser tpr_shadow vnmi flexpriority
ept vpid tsc_adjust arat
```

Listing A.1: Die Hardware der verwendeten Maschine

B. Code

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4  #include <sys/time.h>
5  #include <time.h>
6
7  void mxn(int m, int n, int l,
8           double **a, double **b, double **ab);
9
10 int main(int argc, char *argv[])
11 {
12     if(argc != 4) {
13         printf("\nPlease give exactly 3 arguments m,n,l");
14         return 1;
15     }
16     double **a,**b, **ab;
17     int i,j,m,n,l;
18     clock_t t;
19
20     struct timeval tv_start, tv_stop;
21
22     m = atoi(argv[1]);
23     n = atoi(argv[2]);
24     l = atoi(argv[3]);
25
26     a = malloc (sizeof(double*)*m);
27     for(i=0;i<m;++i) {
28         a[i] = malloc(sizeof(double)*n);
29     }
30
31     b = malloc (sizeof(double*)*n);
32     for(i=0;i<n;++i) {
33         b[i] = malloc(sizeof(double)*l);
```

```
34      }
35
36      ab = malloc (sizeof(double*)*m);
37      for(i=0;i<n;++i) {
38          ab[i] = malloc(sizeof(double)*l);
39      }
40
41      for(i=0;i<m;++i) {
42          for(j=0;j<n;++j) {
43              a[i][j] = rand() % 50 + 1;
44          }
45      }
46      for(i=0;i<n;++i) {
47          for(j=0;j<l;++j) {
48              b[i][j] = rand() % 50 + 1;
49          }
50      }
51
52      gettimeofday(&tv_start, NULL);
53      t = clock();
54      mxn(m,n,l,a,b,ab);
55      t = clock() - t;
56      gettimeofday(&tv_stop, NULL);
57
58      printf("\nTime elapsed: %ld usec\n", (tv_stop.tv_sec
59                                          - tv_start.tv_sec)
60                                          * 1000000L
61                                          + tv_stop.tv_usec
62                                          - tv_start.tv_usec);
63      printf("\nCPU Time: %ld usec\n", t);
64
65      free(a);
66      free(b);
67      free(ab);
68      return 0;
69  }
70
71  void mxn (int m, int n, int l,
72            double **a, double **b, double **ab)
73  {
74      int i,j,k;
75
```

```
76      #pragma omp parallel private(i,j,k) shared(ab,a,b,m,n,l)
77      {
78          #pragma omp for
79          for(i=0;i<m;++i) {
80              for(j=0;j<l;++j) {
81                  ab[i][j] = 0;
82                  for(k=0;k<n;++k) {
83                      ab[i][j] += a[i][k]*b[k][j];
84                  }
85              }
86          }
87      }
88
89  }
```

Listing B.1: mxn.c

```
1   #!/bin/bash
2
3   regex='.*Time elapsed: ([0-9]+) usec.*CPU Time: ([0-9]+)
        ↪ usec.*'
4
5   results='./results.dat'
6   touch $results
7   >$results
8
9   echo "Threads Wallclock CPUTime" >> $results
10
11  op='./mxn-seq'
12
13  for i in {0..24}
14  do
15
16      if [ $i -gt 0 ]
17      then
18          op='./mxn-omp'
19      fi
20      echo -n "${i} ">>$results
21      echo "${i} Threads:"
22      export OMP_NUM_THREADS=${i}
```

```
23
24      run1=$($op 2000 2000 2000)
25      [[ $run1 =~ $regex ]]
26      wallTime1=${BASH_REMATCH[1]}
27      cpuTime1=${BASH_REMATCH[2]}
28
29      run2=$($op 2000 2000 2000)
30      [[ $run2 =~ $regex ]]
31      wallTime2=${BASH_REMATCH[1]}
32      cpuTime2=${BASH_REMATCH[2]}
33
34      run3=$($op 2000 2000 2000)
35      [[ $run3 =~ $regex ]]
36      wallTime3=${BASH_REMATCH[1]}
37      cpuTime3=${BASH_REMATCH[2]}
38
39      wallTime=$(((wallTime1+wallTime2+wallTime3)/3))
40      cpuTime=$(((cpuTime1+cpuTime2+cpuTime3)/3))
41
42      echo -n "${wallTime} ">>$results
43      echo "$cpuTime">>$results
44
45  done
```

Listing B.2: jobscript zum Testen der Laufzeit von mxn.c

Literatur

[Cha01] Rohit Chandra, Hrsg. *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann Publishers, 2001. 230 S. ISBN: 978-1-55860-671-5.

[Com07] Wikimedia Commons. *An illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel.* 2007. URL: `https://upload.wikimedia.org/wikipedia/commons/f/f1/Fork_join.svg` (besucht am 30. 10. 2017).

[CJP08] Barbara Chapman, Gabriele Jost und Ruud van der Pas. *Using OpenMP: portable shared memory parallel programming*. Scientific and engineering computation. OCLC: ocn145944336. Cambridge, Mass: MIT Press, 2008. 353 S. ISBN: 978-0-262-53302-7 978-0-262-03377-0.

[Tea09] GCC Team. *Graphite/Parallelization - GCC Wiki*. 2009. URL: `https://gcc.gnu.org/wiki/Graphite/Parallelization` (besucht am 14. 02. 2018).

[ARB15a] OpenMP ARB. *OpenMP 4.5 API C/C++ Syntax Reference Guide*. 2015. URL: `http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf` (besucht am 29. 10. 2017).

[ARB15b] OpenMP ARB. *OpenMP 4.5 API Complete Specification*. Nov. 2015. URL: `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf` (besucht am 29. 10. 2017).

[Tea15] GCC Team. *openmp - GCC Wiki*. 2015. URL: `https://gcc.gnu.org/wiki/openmp` (besucht am 31. 10. 2017).

[ARB16] OpenMP ARB. *OpenMP Technichal Report 4: OpenMP 5.0 Preview 1*. 11. Okt. 2016. URL: `http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf` (besucht am 31. 10. 2017).

[Con17] Wikipedia Contributors. *OpenMP*. In: *Wikipedia*. Page Version ID: 804095087. 6. Okt. 2017. URL: `https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=804095087` (besucht am 30. 10. 2017).