
Viktor Styrbul

In dieser Ausarbeitung geht es um die Programmierschnittstelle OpenMP. Es wird an ihre Eigenschaften und ihre Merkmalle eingegangen. Es werden existierende Kernelemente aufgezählt und Ausführungsmodell beschrieben.

## Inhaltverzeichnis:

# 1 Was ist OpenMP

OpenMP ist eine Programmierschnittstelle, mit deren Hilfe Parallelität in C, C++ und Fortran-Programmen spezifiziert werden kann. OpenMP steht für Open Multi-Processing. Das heißt OpenMP ist ein offener Standard zur Programmierung auf Multiprozeccor-Computern. Wenn man OpenMP mit vielen konkurrierenden Ansätzen zur Parallelisierung vergleicht, dann stellt man fest, dass OpenMP dafür nur minimale Änderungen am ursprünglich sequenziellen Quelltext erfordert. Das führt dazu, dass die Lesbarkeit des resultierenden Quelltextes erheblich verbessert wird. Meistens sind es nur ein paar zusätzliche Anweisungen an den Compiler notwendig, um aus einem sequentiellen Programm ein paralleles Programm zu machen. Das heißt, dass es kaum einen schnelleren Weg gibt C/C++ und Fortran-Programme zu parallelisieren gibt, als OpenMP-Schnittstelle zu verwenden.

OpenMp setzt sich aus einer Menge von Umgebungsvariablen, Bibliotheksfunktionen und Compilerdirektiven[1] zusammen. Durch OpenMP wird es für Shared-Memory-Architekturen[2] verschiedener Hersteller ein portables paralleles Programmiermodell zur Verfügung gestellt. Die zugrundeliegende Programmiersprache wird durch Direktiven um einige Konstrukten erweitert: Konstrukten zur Arbeitsaufteilung zwischen parallellaufenden Threads und Konstrukten zu ihrer Synchronisierung[3] und gestatten ihnen zusammen oder separat auf die Daten zuzugreifen. Umgebungsvariablen und Bibliotheksfunktionen steuern die Parameter der Laufzeitumgebung, in der das parallele Programm ausgeführt wird. OpenMP wird bei vielen Compilerherstellern unterstützt. So verfügen die Compiler meistens über eine Kommandozeilenoption, die für die Interpretation der OpenMP-spezifischen Compileranweisung an- und ausgeschaltet werden kann.

[1] In den Quelltext eingefügte Steueranweisung für den Compiler.
[2] Architekturen, bei denen mehrere CPUs in einem Rechner, die auf einen gemeinsamen Speicher zugreifen können.
[3] Ein Verfahren zum den gemeinsamen Zugriff von Prozessen auf geteilte Ressourcen.

## 2 Warum Parallelisierung

Im Jahr 2002 führt Intel die Hyper-Threading Technologie[4] ein. Ab diesem Zeitpunkt sind auch normale Arbeitsplatzrechner fähig, zwei Programme bzw. zwei Threads parallel[5] auszuführen. Damit ist es möglich die Programme schneller als auf einer einzelner CPU auszuführen, bei der die Programme nur nebenläufig[6] ausgeführt werden können. Dieser Trend hat sich bei den aktuellen Rechnern mit den Dualcore- und Quadcore-CPUs verfestigt. Damit solche Prozessoren optimal ausgenutzt werden können, müssen Anwendungen notwendigerweise parallelisiert werden. Unter der Parallelisierung eines Programms versteht man, dass in einem Programm eine Aufgabe in mehrere Teile, die gleichzeitig nebeneinander ausgeführt werden, zerlegt werden kann. Dadurch wird die Gesamtaufgabe schneller verarbeitet, als bei einer strikt seriellen Verarbeitung. Dabei wird jeder Programmierer von zwei akuten Problemen beim Entwurf von Prozessoren gezwungen sein, sich langfristig mit der parallelen Programmierung zu befassen:

Erstes Problem ist durch das Moore'sches Gesetzt beschrieben. Das Moore'sche Gesetz wurde im Jahr 1965 durch Gordon Moore aufgestellt. Moore'sches Gesetz ist in Wirklichkeit kein wissenschaftliches Naturgesetz, sondern eine Faustregel, die auf einer Beobachtung berührt. Dieses Gesetzt besagt, dass die Anzahl der Transistoren auf den CPUs sich alle 18 Monate verdoppelt. Dabei steigen aber die Zuwächse an der Produktivität bei den CPU-Entwurfswerkzeugen nur mit einer viel geringeren Rate. So haben die CPU-Designer beim Entwurf einer CPU das Problem, dass die Anzahl der zur Verfügung stehenden Bauteile viel schneller wächst als ihre Fähigkeit, diese überzeugend zu verplanen. Die einfachste Möglichkeit dieses Problem zu lösen ist, Funktionseinheiten wie komplette CPUs zu replizieren. Diese Lösung sieht man an den aktuellen Prozessoren, die mehrere CPUs beinhalten.

Das zweite Problem ist, dass es immer schwer fällt, die Geschwindigkeit, mit der einzelne Threads verarbeitet werden, unter Einhaltung einer gegebenen maximalen elektrischen Leistungsaufnahme zu erhöhen. Ein überwiegender Grund dafür ist, dass mit kleineren Transistorstrukturen und höheren Taktraten die sogenannten Leckströme[7] auf der CPU stark

---

[4] Bei Hyper-Threading Technologie können bestimmte Prozessoren nur mit einem Prozessorkern mehrere Programme gleichzeitig bearbeiten.
[5] Prozesse bzw. Threads werden durch mehrere Prozessoren echt gleichzeitig ausgeführt.
[6] Prozesse bzw. Threads können zeitlich unabhängig voneinander ausgeführt werden.
[7] Es sind unerwünschte Stromflüsse innerhalb der Transistoren. Diese können sowohl im gesperrten als auch im leitenden Zustand des Transistors auftreten.

zunehmen. Auch dieses Problem führt dazu, dass die Hardwareindustrie sich stärker auf Parallelprozessoren konzentrieren muss.

## 3 Geschichte

Im Oktober 1997 wurde durch OpenMP ARB[8] erste OpenMP API[9]-Spezifikation für Fortran veröffentlicht. OpenMP ARB ist Non-Profit-Technologie-Konsortium, der sich von einer Gruppe von großen Computer-Hardware- und Softwareherstellern, wie AMD, IBM, Intel, Cray, HP, Fujitsu, Microsoft, NVIDIA, Oracle Corporation, und vieles mehr. zusammensetzt. Im Oktober des folgenden jahres wurde ein C/C++ Standard veröffentlicht. 2000 wurde die Version 2.0 der Fortran-Spezifikation und die C/C++-Spezifikation in 2002 veröffentlicht. Erst bei Version 2.5 wurden die C/C++- und Fortran-Spezifikation zusammen kombiniert. Diese Spezifikation wurde 2005 veröffentlicht. Version 3.0 erscheint im Mai 2008. Eingeschlossen in die neue Funktionen in 3.0 ist das Konzept von Aufgaben und Aufgaben zu konstruieren. Diese neuen Funktionen sind in Anhang F der OpenMP 3.0-Spezifikationen zusammengefasst. Letzte OpenMP Spezifikation wurde in Version 3.1 am 9. Juli 2011 veröffentlicht.

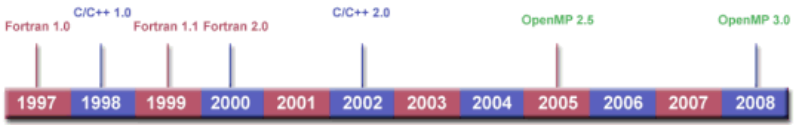

Abbildung 1: Entwicklung von OpenMP

## 4 Merkmale von OpenMP

Im Folgenden wird ein kleines Codeausschnitt in C++ gezeigt, in dem ein Array von ganzen Zahlen parallel von mehreren Thread initialisiert wird. Dieses Beispiel zeigt einige Eigenschaften von OpenMP auf.

```
1. const int size = 1337;
2. int array[size];
3. #pragma omp parallel for
4. for (int i = 0; i < size; i++)
5.         array[i] = i;
```

In den ersten zwei Zeilen des Codeausschnitts wird ein Array von ganzen Zahlen namens array von der Größe 1337 definiert. In der nächsten Zeile sieht man ersten OpenMP-spezifischen Ausdruck. Das ist

[8] Architecture Review Board
[9] Application Programming Interface, Programmierschnittstelle

die Compilerdirektive #pragma omp parallel for. Diese veranlasst, dass die folgende for-Schleife parallel durch mehrere Threads ausgeführt wird. Diese Schleife macht nichts anderes, als oben deklariertes Array mit den ganzen Zahlen zu initiieren. Dabei bleiben solche Details verborgen, wie viele Threads beim der Ausführung des Programms zum Einsatz kommen, oder in welche Bereiche das Array aufgeteilt und von welchem Thread initialisiert wird. Schon durch dieses kleines Beispiel kommen folgende OpenMP-Merkmale hervor:

- OpenMP verschafft einen sehr hohen Abstraktionsgrad, dass man die Threads nicht explizit initialisieren, starten oder beenden muss. Diese Arbeit wird vom Compiler[10] selbst übernommen. Den Codeausschnitt, denn man parallel ausführen möchte, stellt man zwischen sequentiell auszuführenden Anweisungen im Quelltext[11] und nicht, wie bei Pthreads-Bibliotheken, in einer gesonderten Funktion. Sogar muss man sich nicht um die Zuordnung von Arrayindizes zu Threads kümmern. Die Ganze Arbeit übernimmt der Compiler, sofern es vom Programmierer nicht anders erwünscht ist.
- Wenn man versucht ein Programm mit einem nicht OpenMP-spezifischen Compilern zu übersetzten, dann werden die OpenMP-spezifischen Direktiven einfach ignoriert. Höchstens kann es zu einer Warnung kommen. Die Übersetzung wird aber trotzdem und ohne Abbruch durchgeführt. So bleibt die ursprünglich sequentielle Struktur vom Code beibehalten.
- Mit der oben genannter Eigenschaft ist es mit OpenMP möglich ein Programm schrittweise zu parallelisieren, da der Quelltext dafür keine wesentliche Veränderung braucht. Im Prinzip muss der Quelltext dafür nur um einige Zeilen ergänzt werden. Diese Eigenschaft ermöglicht beim Parallelisieren eines Programms, es direkt auf ihre Korrektheit zu überprüfen. Das geschieht in dem man bei der Übersetzung dafür entsprechende Kommandozeilenoptionen weglässt und erhält damit eine lauffähige sequentielle Version des Programms zu Vergleichszwecken.
- Parallelisierung mit OpenMP ist immer lokal begrenzt. Um ein Programm erfolgsvoll zu Parallelisieren ist nur eine kleine Erweiterung mit verhältnismäßig unwesentlichem Umfang notwendig.

---

[10] Ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache umwandelt.

[11] In den Quelltext eingefügte Steueranweisung für den Compiler.

- Mit OpenMP ist es möglich Leistung in „letzter Minute" zu optimieren, weil man die Anwendung zur Parallelisierung nicht neu entwerfen muss.
- Da OpenMP von vielen großen Hardwarehersteller unterstützt wird, ist es sehr leicht eine OpenMP-Anwendung zu portieren.
- OpenMP ist ein herstellerübergreifender und offener Standard. OpenMP ist im Jahr 1997 erschienen und hat sich seit mittlerweile etabliert. Der aktueller Standard liegt seit 9. Juli 2011 in der Version 3.1. Viele verschiedene Hersteller stellen ihre OpenMP-fähige Compiler zur Verfügung.

OpenMP ist sehr einfach anzuwenden. Alles was man braucht sind Direktiven, die auch Pragmas genannt werden. Diese zeigen dem Compiler, welche Codebereiche man parallelisieren möchte. Alle Direktiven bei OpenMP fangen mit #pragma omp. Diese Direktiven werden bei einem nicht OpenMP-fähigen Compiler einfach ignoriert. Allgemein haben alle OpenMP-Direktiven die Form:

#pragma omp <Direktive> [Klausel [[ , ] Klausel] ... ]

Klauseln sind optional. Durch Klauseln wird das Verhalten von Direktiven beeinflusst, auf die sie sich beziehen. Einige Direktiven erfordern eine andere Anzahl an Klauseln und bei einigen sind überhaupt welche Klauseln nicht erlaubt.

OpenMP bringt auch unterschiedliche Bibliothekfunktionen mit sich. Diese Funktionen werden hauptsächlich eingesetzt, um Parameter der Laufzeitumgebung von OpenMP abzufragen bzw. diese zu setzten. Außerdem beinhalten die Bibliotheken auch Funktionen, mit denen man Threads synchronisieren kann. Um die Funktionen aus der Laufzeitbibliothek in einem Programm verwenden zu können, muss man die Headerdatei[12] omp.h einbinden. Wenn man aber keine Funktionen der Laufzeitbibliothek, sondern nur OpenMP-Direktiven verwenden möchte, dann kann man auch theoretisch auf die Einbindung der Headerdatei verzichten. Es ist aber möglich dass nicht jeder Compiler ein ausführbares Programm erstellen kann, wenn man auf die Einbindung der Headerdatei verzichtet hat. Deswegen wird es empfohlen, dass wenn man ein Programm parallelisieren möchte und nur die OpenMP-Direktiven nutzen möchte, trotzdem die Headerdatei einzubinden.

Ein nicht OpenMP-fähiger Compiler wird auch die Headerdatei omp.h nicht kennen. Um dieses Problem zu lösen, kommen die OpenMP-

[12] Eine Text Datei, die Deklarationen und andere Bestandteile des Quelltextes beinhaltet.

spezifische Umgebungsvariablen zum Einsatz. So ist bei einem OpenMP-fähigen Compiler bei aktivierter OpenMP-Option in Eigenschaften eines Projekts die Variable _OPENMP definiert. Der Wert dieser Variablen entspricht dem Datum der umgesetzten OpenMP-Spezifikation im Format JJJJMM. So kann man mit bedingter Klammerung mit #ifdef die Headerdatei omp.h bei einem OpenMP-fähigen Compiler einbinden. Sonst wird diese ignoriert.

## 5 OpenMP-fähige Compiler

Um OpenMP Programmierschnittstelle bei Programmen nutzen zu können braucht man OpenMP-fähige Compiler. OpenMP wird durch folgende C/C++-Compiler unterstützen:

- Visual Studio 2005 und 2008 in der Team Edition- bzw. Professional Variante unterstützen OpenMP, aber nicht die Express-Edition. Bei jedem C/C++-Projekt ist es möglich über dessen Eigenschaften bei Konfigurationseigenschaften > C/C++ > Sprache OpenMP Unterstützung zu aktivieren. Diese Einstellung entspricht der Compileroption „/openmp".
- Intels C++-Compiler ab der Version 8 unterstützt den OpenMP-Standard. Zusätzlich zu den Standard-Direktiven gibt es hier auch Intel-spezifische Direktiven. Der Compiler für Linux steht nur für nicht kommerzielle Zwecke zur Verfügung, für Betriebssysteme wie Windows und MacOS X gibt's nur kostenlose Testversionen.
- GCC unterstützt OpenMP seit der Version 4.2. Wenn man bei kompilieren OpenMP zu Einsatz bringen möchte, so muss man ein Programm mit der Option –fopenmp übersetzen lassen.
- Sun Studio 12 für Solaris OS unterstützt den OpenMP-Standard 3.0.
- IBM XL C/C++ Compiler
- Nanos compiler

## 6 OpenMP Ausführungsmodell

Die parallele Ausführung von Programmen basiert bei OpenMP auf dem Fork/Join – Ausführungsmodell. So startet jedes OpenMP-Programm nur mit einem Thread, der Master Thread. So wird das Programm mit einem Thread ausgeführt bis dieser auf eine #pragma omp parallel-Direktive trifft. Auf dieser Stelle werden weitere Threads erstellt und die Ausführung ab jetzt im mehreren parallelen Threads weiter geführt

(engl. Fork). Wenn der paralleler Abschnitt dann zu Ende ist, wird dann die Programmausführung wiederzusammengefügt (engl. Join).

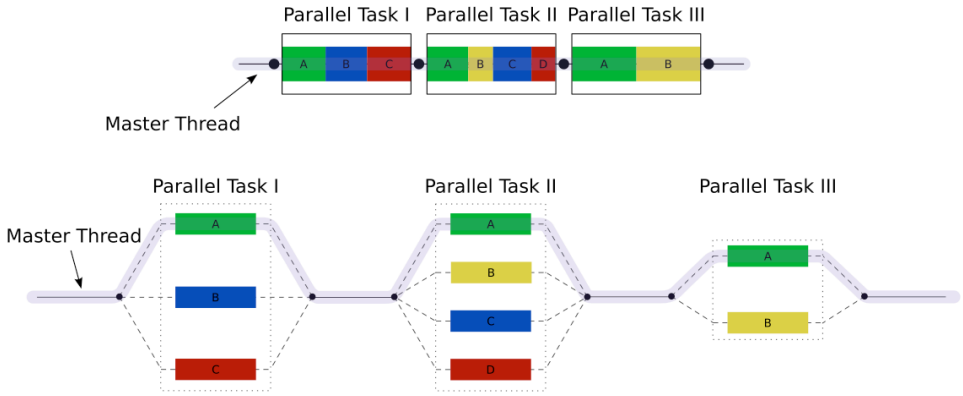


Abbildung 2: OpenMP Ausführungsmodell (fork/join)

Es befindet sich am Ende eines parallelen Abschnitts und auch aller Arbeit aufteilenden Direktiven steht eine implizite Barriere. Eine Barriere ist ein Synchronisationsmechanismus, der dafür sorgt, dass alle zuvor erstellte Threads am Ende eines parallelen Codeabschnitts warten müssen, bis auch der letzte Thread seine Arbeit beendet. Dieser Schritt wird benötigt, um die Semantik vom wieder in die sequentielle Ausführung übergehenden Programms erhalten bleibt. Das kann eine Auswirkung auf die Leistung und Skalierbarkeit vom Programm haben. Das heißt ein paralleler Codeabschnitt kann von einem Bundle von Threads so schnell ausgeführt werden, wie es das langsamste Thread für durchlaufen braucht. Alle andere Threads, die schon ihre Arbeit erledigt haben, müssen auf diesen gemeinsam warten.

Es bleibt der konkreten OpenMP-Implementierung im Compiler überlassen, welche Typs von Threads erzeugt werden. Es können vollwertige Prozesse mit Shared-Memory gestartet werden, oder auch leichtgewichtige Threads auf Basis der Pthreads-Bibliothek. Für OpenMP macht für ein Thread keinen Unterschied. Ein Thread ist einfach ein Kontrollfluss, der zusammen mit den anderen im gemeinsam genutzten Adressraum ausgeführt wird.

# 7 Kernelemente von OpenMP

Die Kernelemente der OpenMP-Spezifikation sind die Konstrukte für die Thread-Erzeugung, Arbeitsverteilung auf mehrere Threads, Verwaltung des Gültigkeitsbereiches von Daten, Thread-Synchronisation, Umgebungsvariablen und Laufzeitroutinen.

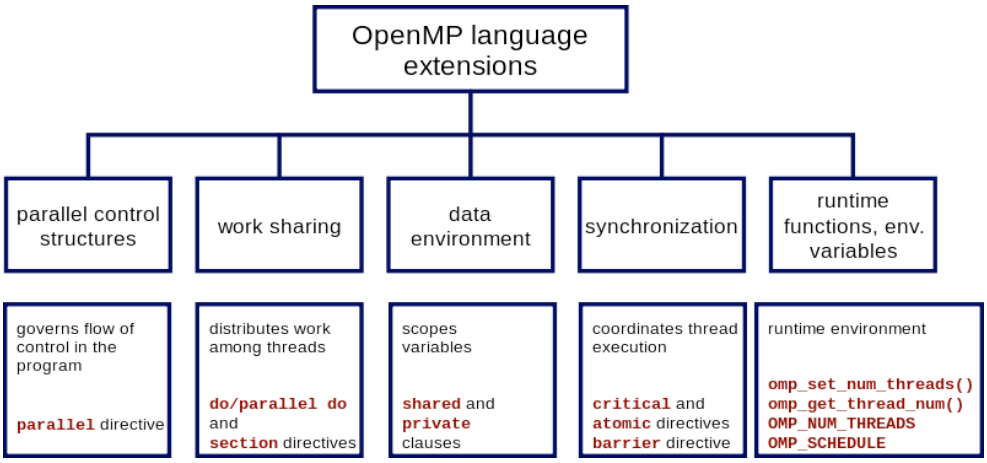

Abbildung 3: Kernelemente von OpenMP

## 7.1 Thread-Erzeugung

Durch omp parallel wird das Programm aufgeteilt. Einschließlich werden zusätzliche Threads[13] zu erzeugen, um die Aufgabe parallel zu berechnen. Der ursprüngliche Prozess wird als Master-Thread bezeichnet und hat 0 als Thread-ID.

## 7.2 Arbeitsverteilung

Durch die Konstrukte der Arbeitsverteilung wird es bestimmt wie Arbeit auf parallele Threads verteilt wird.

- omp for und omp do werden verwendet um die Schleifendurchläufe gleichmäßig auf alle Threads zu verteilen.
- sections verteilt folgende aber unabhängige Programmteile auf unterschiedliche Threads

[13] Ein Thread repräsentiert die sequentielle Ausführung eines Programms

## 7.3 Verwaltung des Gültigkeitsbereichs von Daten

Bei Shared-Memory-Architekturen sind meistens alle Daten in einem Programm für alle Threads sichtbar. Es besteht aber manchmal der Bedarf an privaten, also nur für einen bestimmten Thread sichtbar, und dem expliziten Austauschen von Werten zwischen parallelen und sequentiellen Programmabschnitten. Dafür gibt es in OpenMP die Date-Klausel.

- shared: Wenn die Daten diesen Typ haben, dann sind sie gleichzeitig für alle Threads sichtbar und können von jedem Thread geändert werden, da sie für alle Threads an der gleichen Speicherstelle liegen. Wenn man keine weiteren Angaben macht, so sind das gemeinsame Daten. Die einzige Ausnahme stellen Schleifenvariablen dar.
- private: Es wird für jeden Threadseine eigene Kopie dieser Daten angelegt. So müssen private Daten nicht initialisiert werden. Wenn der paralleler Abschnitt des Programms zu Ende ist, werden diese Daten gelöscht.
- firstprivate: Das sind auch private Daten wie oben. Der Unterschied besteht aber in dem, dass sie mit dem letzten Wert vor dem parallelen Abschnitt initialisiert werden.
- lastprivate: Auch private Daten. Der Unterschied ist, dass der Thread, der die letzte Iteration ausführt, am Ende den Wert aus dem parallelen Abschnitt herauskopiert.
- threadprivate: Diese Daten sind global. Sie werde im parallelen Programmabschnitt aber als private behandelt werden. Der globale Wert wird über den parallelen Abschnitt hinweg behalten.
- copying: mit copying wird der globale Wert explizit an die private Daten übertragen. Ein copyout ist aber unnötig, das der globale Wert beibehalten wird.
- reduction: Die Daten sind privat, aber wenn der Abschnitt zu Ende ist werden alle Daten auf einen globalen Wert zusammengefasst (reduziert). Damit lässt sich zum Beispiel die Summe aller Elemente in einem Array parallel zu berechnen.

## 7.4 Konstrukte zur Synchronisation

- critical section: Ein eingeschlossener Programmabschnitt wird von allen Threads ausgeführt, aber niemals zur gleichen Zeit.
- atomic: ist ähnlich wie critical section, aber mit dem Hinweis an den Compiler spezielle Hardwarefunktionen zu benutzen. Dieser Hinweis kann aber vom Compiler ignoriert werden. Eine sinnvolle Verwendung ist für atomic ist das exklusive Aktualisierung von Daten.

- barrier: setzt eine Barriere. Wenn ein Thread eine Barriere erreicht hat, wartet er bis alle anderen Threads der Gruppe ebenfalls die Barriere erreicht haben.
- ordered: der eingeschlossene Abschnitt wird der Reihenfolge nach abgearbeitet, als würde die Schleife sequentiell abgearbeitet werden.
- flush: setzt einen Synchronisationspunkt fest, bei dem ein konsistentes Speicherabbild erstellt werden muss. So werden privaten Daten in den Arbeitsspeicher zurückgeschrieben.
- single: eingeschlossener Abschnitt wird nur von einem Thread ausgeführt und zwar von dem, der ihn als erstes erreicht.Dies impliziert eine Barriere am Ende des Blocks.
- master: ist ähnlich wie single mit dem Unterschied, dass der eingeschlossene Programmabschnitt nur vom Master-Thread ausgeführt wird. Am Ende des Blockes ist keine Barriere impliziert.

### 7.5 Laufzeitroutinen

Laufzeitroutinen werden eingesetzt, um die Threadanzahl während der Laufzeit zu bestimmen oder um rauszufinden ob das Programm sich aktuell im parallelen oder sequentiellen Zustand befindet.

### 7.6 Umbegungsvariablen

Umgebungsvariablen liefern Information wie die Thread ID. So kann man durch gezieltes Verändern von bestimmten Umgebungsvariablen die Ausführung von OpenMP-Programmen verändern. Durch die Umgebungsvariablen kann man die Anzahl von Threads und die Schleifenparallelisierung zur Laufzeit beeinflusst werden.

## 8 Fazit

Parallelisierung ist heutzutage ein sehr wichtiges Thema, da aktuelle Rechner mehrere Prozessoren enthalten. Das erlaubt der Rechner ein Programm viel schneller auszuführen. Das geschieht dadurch, dass die Ausführung dieses Programms auf die einzelnen Prozessoren aufgeteilt werden kann. Eine Möglichkeit die Programme zu parallelisieren ist OpenMP. OpenMP ist eine Programmierschnittstelle, die es den Programmierern ermöglicht ihre Programme zu parallelisieren. OpenMP ist sehr flexibel. Es ermöglicht Programme ohne großen Aufwand zu

parallelisieren. Außerdem sind es dafür keine großartige Änderungen am Quelltext nötig. So lassen sich schon vorhandene Programme zu parallelisieren.

**Literaturverzeichnis**


[SI08] Simon Hoffmann, Prof. Dr. Rainer Lienhart, Informatik im Fokus OpenMP, 2008

[WD09] Wolfgang Dautermann, Parallele Programmierung mit OpenMP. http://wolfgang.dautermann.at/CLT2009-OpenMP-Vortrag.pdf. Stand: 04.02.2012

[MW07] Michael Westermann, OpenMP, www.wi.uni-muenster.de/pi/lehre/ss07/SeminarPP/ausarbeitungen/OpenMP.pdf. Stand: 04.02.2012

[BR08] Benjamin Rommel, Einführung in OpenMP, http://www.bero-software.de/tutorials_programming/omp.pdf. Stand: 04.02.2012

[WI12] http://de.wikipedia.org/wiki/OpenMP. Stand: 04.02.2012

[MP12] http://openmp.org/wp/. Stand: 04.02.2012