

Einführung

OpenMP ist eine Programmierschnittstelle, mit deren Hilfe Parallelität in C, C++ und Fortran-Programmen spezifiziert werden kann. Anders als viele konkurrierende Ansätze zur Parallelisierung erfordert OpenMP nur minimale Änderungen am ursprünglich sequenziellen Quellcode und trägt so erheblich zu der Lesbarkeit des resultierenden Quelltextes bei. Oft müssen nur ein paar zusätzliche Anweisungen an den Compiler eingefügt werden. Dies bedeutet, dass es – sofern sich OpenMP zur Spezifizierung der gewünschten Parallelität eignet – kaum einen schnelleren Weg gibt, C/C++ und Fortran-Programme zu parallelisieren.

OpenMP setzt sich aus einer Menge von Compilerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen zusammen. Das Ziel von OpenMP ist es, ein portables paralleles Programmiermodell für Shared-Memory-Architekturen verschiedener Hersteller zur Verfügung zu stellen. Die Direktiven erweitern die zugrundeliegende Programmiersprache mit Konstrukten zur Arbeitsaufteilung zwischen parallel laufenden Threads und Konstrukten zur Synchronisierung dieser Threads und ermöglichen ihnen gemeinsa-

men oder getrennten Zugriff auf Daten. Die Bibliotheksfunktionen und Umgebungsvariablen steuern die Parameter der Laufzeitumgebung, in der das parallele Programm läuft. OpenMP wird von zahlreichen Compilerherstellern unterstützt; entsprechende Compiler verfügen oft über eine Kommandozeilenooption, mit Hilfe derer sich die Interpretation der OpenMP-spezifischen Compileranweisungen an- und ausschalten lässt.

Wie das „Open“ im Namen bereits vermuten lässt – das „MP“ steht für *multi processing* – ist OpenMP ein offener Standard. Die erste Adresse im Web für OpenMP ist <http://www.openmp.org>, von wo auch die – für ein technisches Dokument ausgezeichnet lesbare – offizielle OpenMP-Spezifikation des OpenMP Architecture Review Board heruntergeladen werden kann [27]. Ebenfalls erwähnt werden sollte die Webseite der Gemeinschaft der OpenMP-Benutzer, <http://www.compunity.org>.

1.1 Merkmale von OpenMP

Das folgende minimale Codebeispiel in C++, in dem ein Vektor von ganzen Zahlen parallel von mehreren Threads initialisiert wird, soll einen ersten Vorgeschmack bieten und einige der Eigenschaften von OpenMP aufzeigen:

```

1  const int size = 524288;
2  int arr[size];
3  #pragma omp parallel for
4  for(int i = 0; i < size; ++i)
5      arr[i] = i;
```

In den beiden ersten Zeilen wird ein Vektor von ganzen Zahlen namens *arr* der Größe 524288 definiert. In Zeile 3 begegnen wir dem ersten OpenMP-Ausdruck: Die (auch Pragma

genannte) *Compilerdirektive* `#pragma omp parallel for` bewirkt, dass die in Zeile 4 folgende `for`-Schleife parallel von mehreren Threads ausgeführt wird. Der Schleifenkörper dieser Schleife besteht aus der einzelnen Anweisung in Zeile 5, in der jedem Vektorelement sein Index als Wert zugewiesen wird. Diese Anweisung ist es, die parallel ausgeführt wird. Ohne auf Details, wie die genaue Anzahl der zum Einsatz kommenden Threads, oder die Frage, auf welche Bereiche des Vektors von welchem dieser Threads zugegriffen wird, bereits an dieser Stelle näher einzugehen, zeigt sich folgendes:

- OpenMP bietet einen hohen Abstraktionsgrad. Der Programmierer muss die Threads nicht explizit initialisieren, starten oder beenden. Der Code, der von den Threads parallel ausgeführt werden soll, steht zwischen den sequentiell auszuführenden Anweisungen im Quelltext und nicht etwa in einer separaten Funktion, wie es z. B. bei der Pthreads-Bibliothek [25] der Fall ist. Auch die bereits erwähnte Zuordnung von Vektorindizes zu Threads erfolgt implizit, kann jedoch – sofern gewünscht – vom Programmierer beeinflusst werden (siehe Kapitel 3.3).
- Die ursprüngliche sequenzielle Codestruktur bleibt erhalten. Ein nicht OpenMP-fähiger Compiler ignoriert die OpenMP-spezifischen Pragmas oder gibt eine Warnmeldung aus, ohne jedoch die Compilierung zu unterbrechen.
- Aus der obigen Eigenschaft folgt, dass mit OpenMP die schrittweise Parallelisierung eines Programms möglich ist. Da der Quelltext nicht (wesentlich) verändert, sondern nur ergänzt werden muss, bleibt der Code stets lauffähig. Somit sind Parallelisierungen mit OpenMP auch einfach auf Korrektheit zu testen – man schaltet

dazu die entsprechende Kommandozeilenoption ab und erhält eine lauffähige serielle Version zu Vergleichszwecken.

- Parallelisierungen mit OpenMP sind lokal begrenzt. Für eine erfolgreiche Parallelisierung genügt oft schon eine Erweiterung von relativ geringem Umfang.
- Mit OpenMP sind Leistungsoptimierungen „in letzter Minute“ möglich, da kein Neuentwurf der Applikation zur Parallelisierung notwendig ist.
- OpenMP ist portierbar und wird von nahezu allen großen Hardwareherstellern unterstützt.
- OpenMP ist ein offener, herstellerübergreifender Standard. OpenMP existiert seit 1997 und hat sich mittlerweile etabliert. Der Standard liegt in der Version 2.5 vor, und Version 3.0 steht vor der Tür [28]. Es stehen OpenMP-fähige Compiler von verschiedenen Herstellern zur Verfügung. Auch wenn es in der Praxis kleine Unterschiede in der konkreten Umsetzung des Standards gibt, kann jeder dieser Compiler korrekten OpenMP-Code verarbeiten.

OpenMP ist einfach zu verwenden. Die Direktiven, auch Pragmas genannt, weisen den Compiler an, bestimmte Codeabschnitte zu parallelisieren. Alle für OpenMP relevanten Pragmas beginnen mit `#pragma omp`. Nicht OpenMP-fähige Compiler ignorieren diese einfach. OpenMP-Pragmas haben allgemein die Form:

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

Klauseln sind optional und beeinflussen das Verhalten der Direktive, auf die sie sich beziehen. Jede Direktive hat eine andere Menge von gültigen Klauseln. Für einige Direktiven ist diese Menge leer; es sind also keine Klauseln erlaubt.

Alle Anweisungen der Form `#pragma omp <Direktive> [Klausel [[,] Klausel] ...]` müssen mit einem Zeilenumbruch enden. Insbesondere dürfen sie nicht mit der öffnenden Klammer des nachfolgenden Codeblocks enden:

```
// falsch gesetzte Klammer
#pragma omp parallel {
    // kompiliert nicht
}

// richtig gesetzte Klammer
#pragma omp parallel
{
    // Code
}
```

Die Funktionen der Laufzeitbibliothek werden hauptsächlich dazu verwendet, Parameter der Laufzeitumgebung von OpenMP abzufragen bzw. zu setzen. Darüber hinaus enthält die Bibliothek Funktionen zur Synchronisation von Threads. Möchte man Funktionen der Laufzeitbibliothek in einem Programm verwenden, so muss die Headerdatei `omp.h` eingebunden werden. Wenn die Anwendung nur Pragmas verwendet, kann auf das Einbinden von `omp.h` theoretisch verzichtet werden. Allerdings erzeugt beispielsweise der C/C++-Compiler aus Microsoft Visual Studio 2005 kein ausführbares Programm, wenn die Datei nicht eingebunden wurde. Es empfiehlt sich also, `omp.h` für alle zu parallelisierenden Programme zu inkludieren.

Sollte ein Compiler OpenMP nicht unterstützen, so wird er auch `omp.h` nicht kennen. Hier kommen nun die OpenMP-spezifischen Umgebungsvariablen ins Spiel: Für einen OpenMP-fähigen Compiler ist bei aktivierter OpenMP-Option die Variable `_OPENMP` definiert; ihr Wert entspricht dem Datum der umgesetzten OpenMP-Spezifikation im Format *jjjjmm*. Diese kann zur bedingten Klammerung mit

`#ifdef` verwendet werden, so dass ausgewählte, OpenMP-spezifische Codeabschnitte in die Compilierung einbezogen oder von ihr ausgeschlossen werden können. Im folgenden Codestück wird `omp.h` nur inkludiert, wenn `_OPENMP` definiert ist:

```
#ifdef _OPENMP
#include <omp.h>
#endif
// ...
```

1.1.1 OpenMP-fähige Compiler

Folgende C/C++-Compiler unterstützen OpenMP (ohne Anspruch auf Vollständigkeit):

- Visual Studio 2005 und 2008 in der Professional- bzw. Team Edition-Variante, nicht aber die Express-Edition. Für jedes C/C++-Projekt lässt sich über dessen Eigenschaften unter *Configuration Properties (Konfigurationseigenschaften)* \rightarrow *C/C++* \rightarrow *Language (Sprache)* der Schalter „OpenMP support“ aktivieren; dies entspricht der Compileroption „/openmp“.
- Intels C++-Compiler ab der Version 8 implementiert den OpenMP-Standard und zusätzlich weitere Intel-spezifische Direktiven, die über den Standard hinausgehen. Die Version für Linux ist für nicht kommerzielle Zwecke frei erhältlich, die Versionen für Windows und Mac OS X zumindest in kostenlosen Testversionen.
- GCC unterstützt OpenMP seit der Version 4.2. Soll OpenMP zum Einsatz kommen, muss ein Programm mit der Option `-fopenmp` übersetzt werden.
- Sun Studio für Solaris OS implementiert ebenfalls den OpenMP-Standard 2.5.

Werden die o.g. Compileroptionen aktiviert, wird auch die im vorigen Abschnitt beschriebene Variable `_OPENMP` definiert. Möchte man ein mit OpenMP parallelisiertes Programm sequenziell ausführen (etwa zu Debugging- oder Zeitmessungszwecken), so genügt es, den Quellcode ohne die entsprechenden Compileroptionen neu zu übersetzen.

Über dieses Buch

Dieses Buch betrachtet OpenMP aus der Perspektive des C/C++-Programmierers und setzt entsprechende Kenntnisse in diesen Programmiersprachen voraus. OpenMP im Kontext der Programmiersprache Fortran, für die OpenMP ebenfalls spezifiziert ist, wird *nicht* behandelt.

Zum Zeitpunkt der Abfassung waren die OpenMP-Spezifikation 2.5 vom Mai 2005 aktuell und ein Entwurf für die Spezifikation 3.0 gerade der Öffentlichkeit zugänglich gemacht worden [28]. Demnach wird OpenMP 3.0 einige neue Merkmale enthalten, die zum Teil bereits vorab in ähnlicher Form in den OpenMP-fähigen Compilern einzelner Hersteller implementiert waren, wie z. B. Konstrukte zur Spezifikation von Task-Parallelität im C++-Compiler von Intel. Diese Konstrukte finden im vorliegenden Buch bereits Berücksichtigung (siehe z. B. Kapitel 6.2).

1.2 Parallele Programmierung

Der verbleibende Teil dieses einführenden Kapitels gibt einen kurzen Überblick über allgemeine Konzepte paralleler Programmierung wie Prozesse und Threads, Parallelverarbeitung auf Multicoreprozessoren und Leistungsmessung

paralleler Programme. Mit diesen Themen bereits vertraute Leser, die sofort in OpenMP einsteigen wollen, können direkt zu Kapitel 3 weiterblättern. Für eine ausführlichere Behandlung paralleler Architekturen und der Konzepte paralleler Programmierung siehe z. B. [29] aus derselben Reihe.

1.2.1 Prozesse und Threads

Als *Prozess* bezeichnet man ein Programm, das gerade vom Betriebssystem ausgeführt wird. Im Gegensatz zum statischen Programmcode auf der Festplatte handelt es sich also um eine aktive Instanz des Programmcodes. Können in einem System mehrere Prozesse gleichzeitig aktiv sein, bezeichnet man dies als *time-sharing* oder *multitasking* System. Der Ausdruck time-sharing kommt daher, dass – vor allem auf Ein-CPU-Maschinen – diese gleichzeitige Aktivität oft nur dadurch vorgetäuscht wird, dass jeder Prozess innerhalb einer Sekunde mehrfach, aber nur für einen sehr kurzen Zeitraum ausgeführt wird. Da jeder Prozess mehrfach für kurze Zeit ausgeführt wird, ergibt sich der Eindruck der Gleichzeitigkeit der Prozesse. Ein Prozess-Scheduler regelt die genaue zeitliche Ausführung und weist die Prozesse den CPU(s) zu.

Ein Prozess setzt sich aus folgenden Elementen zusammen:

- einer eindeutigen Identifikationsnummer, der Prozess-ID (PID)
- dem Programmcode
- dem aktuellen Wert des Programmschrittzählers
- den aktuellen CPU-Registerwerten
- dem *Stack*, einem Speicherbereich, der lokale Variablen, Rücksprungradressen und Werte von Funktionsparametern enthält

- dem *Datenbereich*, in dem globale Variablen abgelegt sind
- dem *Heap*, einem Speicherbereich für dynamisch angelegte Variablen.

Unterschiedliche Betriebssysteme stellen unterschiedliche Mittel zum Anlegen von Prozessen zur Verfügung. In der UNIX-Welt beispielsweise übernimmt diese Aufgabe der Systemaufruf `fork()`, unter der Win32 API benutzt man `CreateProcess()`.

Entscheidet der Scheduler, dass ein Prozess P auf einer CPU zur Ausführung gelangen soll, so muss das Betriebssystem den aktuellen Status des derzeit dort ausgeführten Prozesses Q sichern, indem alle oben genannten Elemente im sogenannten *Prozesskontrollblock* abgelegt werden. Von dort können sie wiederhergestellt werden, sobald Prozess Q wieder an der Reihe ist, ausgeführt zu werden. Dieses Umschalten zwischen verschiedenen Prozessen bezeichnet man als *Context Switching*.

In modernen Betriebssystemen kann ein Prozess über mehrere Ausführungsstränge oder *Threads* verfügen. Ein Thread kann als „Light-Version“ eines Prozesses betrachtet werden. Neben einer Identifikationsnummer besteht er aus dem aktuellen Wert des Programmzählers, den Registerwerten und einem Stack. Die anderen Elemente eines Prozesses – Programmcode, Datenbereich, Heap und etwaige weitere Ressourcen wie offene Dateien – teilt er sich mit den anderen Threads, die zum selben Prozess wie er gehören. Ein Prozess mit mehreren Threads kann also mehrere Aufgaben auf einmal erledigen. Das erhöht z. B. die Reaktionsgeschwindigkeit auf Benutzereingaben in grafischen Oberflächen. Ein weiterer Vorteil ist der gemeinsame Zugriff auf Ressourcen innerhalb eines gemeinsam genutzten Adressraums; da auch weniger Informationen gesichert und

wiederhergestellt werden müssen, ist es ökonomischer, Context Switching zwischen Threads statt zwischen Prozessen zu vollziehen. Darüber hinaus können in Systemen mit mehreren Prozessoren die Threads tatsächlich gleichzeitig auf unterschiedlichen CPUs laufen und die Parallelität voll ausnutzen [33]. APIs, die dem Programmierer Zugriff auf diese Funktionalität geben, sind beispielsweise die Pthreads-Bibliothek [25] oder die Win32 API.

Hier kommt nun OpenMP ins Spiel, mit Hilfe dessen die Parallelausführung von Programmcode durch unterschiedliche Threads spezifiziert werden kann. Es spielt für den OpenMP-Programmierer keine Rolle, mit welchem Threadmodell die OpenMP-Spezifikation im gerade benutzten Compiler umgesetzt wurde. Um diese und viele weitere Details wie explizites Starten und Beenden von Threads muss er sich nicht kümmern. Im Sinne von OpenMP ist ein Thread ein Kontrollfluss, der zusammen mit anderen Threads einen markierten Codeabschnitt parallel ausführt.

1.2.2 Parallele Hardwarearchitekturen

Als „Moore’sches Gesetz“ bezeichnet man heute die Faustregel, wonach sich die Anzahl an Transistoren auf einem handelsüblichen Prozessor alle achtzehn Monate verdoppelt [38]. Sie wurde (in etwas anderer Form) 1965 von Gordon Moore auf der Basis empirischer Beobachtungen erstmals formuliert [24] und erfuhr über die Jahre einige Abwandlungen, hat aber ihre Gültigkeit behalten¹. In der jüngeren Vergangenheit wird die mit der Erhöhung der Transistorenanzahl einhergehende Leistungssteigerung der Prozessoren nicht mehr durch eine Erhöhung der Taktrate erzielt,

¹ Gordon Moore selbst hat seinem „Gesetz“ 2007 noch eine Gültigkeitsdauer von 10-15 Jahren vorhergesagt, bis eine Grenze erreicht sei.

da technische Beschränkungen dagegen sprechen, sondern durch zunehmende Parallelität auf Prozessorebene [20]. Die Designansätze umfassen u. a. *Hyperthreading*, bei dem mehrere Kontrollflüsse auf einem Prozessor bzw. Prozessorkern ausgeführt werden, sowie *Multicore-Prozessoren*, in denen auf einem Chip mehrere Prozessorkerne mit jeweils eigenen Recheneinheiten integriert sind [16].

In der Vergangenheit konnte man darauf vertrauen, dass alleine durch die mit Erhöhung der Taktrate einhergehende Leistungssteigerung bereits existierende sequenzielle Programme ohne Modifikationen immer schneller ausgeführt werden konnten. Mit zunehmender Parallelität auf Prozessorebene gilt dies nicht mehr. Um die verfügbare Rechenleistung moderner Architekturen auszunutzen, ist es insbesondere durch die wachsende Verbreitung von Multicorechips notwendig geworden, die auszuführenden Programme diesen Gegebenheiten anzupassen und ebenfalls zu parallelisieren. Techniken paralleler Programmierung gehören damit zunehmend zum Rüstzeug eines jeden Programmierers. OpenMP bietet für bestimmte Arten von Architekturen, die im nächsten Abschnitt definiert werden, die Möglichkeit, sequenziellen Code inkrementell zu parallelisieren, und stellt damit eine attraktive Möglichkeit zur Programmierung von Anwendungen für Multicorechips dar.

Allgemein lassen sich Hardwarearchitekturen nach ihrer Anzahl paralleler Daten- und Kontrollflüsse in eine der 1972 von Michael J. Flynn [12] vorgeschlagenen Kategorien einordnen:

- *Single Instruction, Single Data (SISD)*: sequenzielle Rechner, die weder auf der Daten- noch auf der Anweisungsebene parallel arbeiten, wie z. B. der „klassische“ PC.
- *Single Instruction, Multiple Data (SIMD)*: Eine Anweisung kann parallel auf mehreren Datenströmen ausge-

führt werden. Beispiele sind GPUs in Grafikkarten; ein weiteres Beispiel sind die SIMD Anweisungen, die Intel erstmals 1997 mittels der MMX-Technologie beim Pentium Prozessor und AMD 1998 mittels 3DNow!-Technologie beim K6 einführt. Zahlreiche Befehlserweiterungen und Befehlsverbesserungen folgten unter der Namensfamilie der Streaming SIMD Extensions (SSE, SSE2, SSE3, SSE4).

- *Multiple Instruction, Single Data (MISD)*: Ein eher theoretisches Konzept, das bislang nicht für die Massenfertigung von Hardware Verwendung gefunden hat, da mehrere Kontrollflüsse meist auch mehrere Datenströme benötigen, um effektiv rechnen zu können.
- *Multiple Instruction, Multiple Data (MIMD)*: Systeme aus mehreren, voneinander unabhängig auf verschiedenen Daten arbeitenden Prozessoren. Das Standardbeispiel hierfür sind verteilte Systeme wie PC-Cluster, aber auch heutige Multicore-Prozessoren. Diese Klasse lässt sich daher wiederum nach der verwendeten Speicherorganisation in zwei Unterklassen einteilen: Jene mit verteiltem Speicher (z. B. Cluster) und jene mit gemeinsam genutztem Speicher (*shared memory*) wie Multicore-Prozessoren, bei denen jeder auf dem Chip vorhandene Kern Zugriff auf den gemeinsamen physikalischen Speicher hat. In letzterem Fall spricht man auch von *Symmetric Multiprocessing (SMP)*. Parallele Programmierung solcher Shared-Memory-Architekturen ist die Domäne von OpenMP.

Es sei noch angemerkt, dass in einer Anwendung verschiedene Parallelitätsebenen kombiniert sein können: Ein Programm kann auf einem Shared-Memory-Rechner R auf mehreren Prozessoren mit OpenMP parallelisiert ausgeführt werden. Auf jedem dieser Prozessoren können wie-

derum SIMD-Konzepte wie SSE zum Einsatz kommen. Zusätzlich könnte auf einer äußeren Organisationsebene das Programm Teil eines verteilten Systems sein, d. h. weitere Programminstanzen könnten auf einem Cluster verteilt ausgeführt werden.

1.2.3 Leistungsmessung

Bleibt die Frage, welchen Laufzeitvorteil man durch die Parallelisierung abzüglich des zusätzlichen Mehraufwandes bei der Verwaltung der parallelen Threads erzielen kann. Zum Vergleich von seriellen und parallelen Programmen zieht man die beiden Kenngrößen Beschleunigung (engl. *speedup*) und Effizienz (engl. *efficiency*) heran.

Die Beschleunigung s_n ergibt sich als Quotient der Laufzeiten t_1 der sequenziellen und t_n der parallelen Version eines Programms, d. h. aus dem Verhältnis der Ausführungsdauer t_1 auf einem Prozessor zur Ausführungszeit t_n auf n Prozessoren. Die Beschleunigung s_n misst also, wie gut sich ein Programm durch Parallelverarbeitung beschleunigen lässt:

$$s_n = \frac{t_1}{t_n}$$

Die Beschleunigung alleine sagt aber noch nichts darüber aus, wie gut die zusätzlichen Prozessoren im Parallelbetrieb ausgenutzt werden. Die Effizienz e_n eines parallelen Algorithmus ist definiert als der Quotient aus der Beschleunigung s_n und der Anzahl der verwendeten Prozessoren n :

$$e_n = \frac{s_n}{n}$$

Ein Algorithmus, dessen parallele Ausführung auf sechs Prozessoren nur ein Viertel so lange dauert wie die sequenzielle, käme so auf eine Beschleunigung von $s_6 = 4$. Die Effi-

zienz betrüge allerdings nur $e_6 = \frac{4}{6} = 0, \bar{6}$. Ein idealer paralleler Algorithmus, der die zusätzlichen Prozessoren voll ausnutzte und dessen Beschleunigung linear in der Anzahl der genutzten Prozessoren wäre, hätte eine Effizienz von 1. Dennoch stellt eine Beschleunigung von n bei n benutzten Prozessoren nicht die Obergrenze dar. In seltenen Fällen kann eine „superlineare Beschleunigung“ auftreten, bei der eine Anwendung um einen Faktor größer als n beschleunigt wird. Dieser Effekt läßt sich durch Cache-Speichereffekte erklären: Mit der Anzahl der Prozessoren steigt in modernen Computerarchitekturen auch die Gesamtgröße des vorhandenen Cache-Speichers, so dass mehr Daten im Cache gehalten werden können, was die Speicherzugriffszeiten reduziert. Threads können davon profitieren, dass ihre Mitläufer bereits Daten in den Cache geladen haben, auf denen sie nun unmittelbar ihre Berechnungen ausführen können. Dadurch kann die Ausführung schneller erfolgen als ohne diese „Vorarbeit“ der anderen Threads [9].

1.2.4 Das Amdahl'sche Gesetz

Bereits im Jahre 1967 hat sich Gene Amdahl Gedanken zu den Grundlagen parallelen Rechnens gemacht [2]. Er stellte fest, dass der von Multiprozessorarchitekturen benötigte Mehraufwand zur Verwaltung der Daten die mögliche Beschleunigung eines parallelen Algorithmus nach oben beschränkt. Darüber hinaus sei dieser Mehraufwand grundsätzlich sequenzieller Natur und damit selbst nicht parallelisierbar. Er schloss daraus, dass eine Steigerung der Leistung paralleler Rechner nur möglich sei, wenn die Leistung sequenzieller Berechnungen im gleichen Maß mitwachsen würde.

Das nach ihm benannte Amdahl'sche Gesetz beschränkt die theoretisch mögliche Beschleunigung eines parallelen

Algorithmus durch die im Code verbleibenden seriellen Anteile nach oben. Sei σ der (unveränderliche) Anteil der seriellen Berechnungen im Programmcode. Die serielle auf 1 normierte Ausführungszeit t_1 lässt sich beschreiben als die Summe der Zeitanteile für die sequenzielle Ausführung des sequenziellen Programmcodes σ und des parallelen Programmcodes $(1 - \sigma)$:

$$t_1 = 1 = \sigma + (1 - \sigma)$$

Für die parallele Ausführung auf n Prozessoren bleibt der sequenzielle Anteil σ unverändert, während der parallele Anteil $(1 - \sigma)$ auf n Prozessoren verteilt wird:

$$t_n = \sigma + \frac{(1 - \sigma)}{n}$$

Damit gilt nach Amdahl für die maximale Beschleunigung (unter Vernachlässigung des Parallelisierungsmehraufwandes):

$$s_{max,n} = \frac{1}{\sigma + \frac{1-\sigma}{n}}$$

und damit für $n \rightarrow \infty$

$$s_{max,n} \leq s_{max} = \frac{1}{\sigma},$$

da $\frac{1}{a+b} \leq \frac{1}{a}$ bzw. $\frac{1}{a+b} \leq \frac{1}{b}$ für $a, b \geq 0$ gilt. Die maximal erreichbare Beschleunigung ist also unabhängig von der Anzahl der zum Einsatz kommenden Prozessoren durch $\frac{1}{\sigma}$ nach oben beschränkt. Werden beispielsweise 10% eines Programms sequenziell ausgeführt (also $\sigma = 0,1$), besagt das Amdahl'sche Gesetz, dass die maximal mögliche Beschleunigung $\frac{1}{\sigma} = 10$ beträgt. Abbildung 1.1 zeigt die durch das Amdahl'sche Gesetz vorhergesagte maximal mögliche Beschleunigung $s_{max,n}$ als Funktion der Anzahl der zum Einsatz kommenden Prozessoren in Abhängigkeit von verschiedenen sequenziellen Codeanteilen σ an.

OpenMP

Eine Einführung in die parallele Programmierung mit
C/C++

Hoffmann, S.; Lienhart, R.
2008, X, 162 S., Softcover
ISBN: 978-3-540-73122-1