1 Einführung

Die meisten Computer können heute verschiedene Anweisungen parallel abarbeiten. Um diese zur Verfügung stehende Ressource auszunutzen, müssen wir sie bei der Softwareentwicklung entsprechend berücksichtigen. Die nebenläufige Programmierung wird deshalb häufiger eingesetzt. Der Umgang und die Koordinierung von *Threads* gehören heute zum Grundhandwerk eines guten Entwicklers.

1.1 Dimensionen der Parallelität

Bei Softwaresystemen gibt es verschiedene Ebenen, auf denen Parallelisierung eingesetzt werden kann bzw. bereits eingesetzt wird. Grundlegend kann zwischen Parallelität auf der Prozessorebene und der Systemebene unterschieden werden [26, 15]. Auf der Prozessorebene lassen sich die drei Bereiche *Pipelining* (Fließbandverarbeitung), superskalare Ausführung und Vektorisierung für die Parallelisierung identifizieren.

Auf der Systemebene können je nach Prozessoranordnung und Zugriffsart auf gemeinsam benutzte Daten folgende Varianten unterschieden werden:

- Bei *Multinode-Systemen* wird die Aufgabe über verschiedene Rechner hinweg verteilt. Jeder einzelne Knoten (in der Regel ein eigenständiger Rechner) hat seinen eigenen Speicher und Prozessor. Man spricht in diesem Zusammenhang von verteilten Anwendungen.
- Bei *Multiprocessor-Systemen* ist die Anwendung auf verschiedene Prozessoren verteilt, die sich in der Regel alle auf demselben Rechner (Mainboard) befinden und die alle auf denselben Hauptspeicher zugreifen, wobei die Zugriffszeiten nicht einheitlich sind. Jeder Prozessor hat darüber hinaus auch noch verschiedene Cache-Levels. Solche Systeme besitzen häufig eine sogenannte NUMA-Architektur (*Non-Uniform Memory Access*).
- Bei *Multicore-Systemen* befinden sich verschiedene Rechenkerne in einem Prozessor, die sich den Hauptspeicher und zum Teil auch Caches teilen. Der Zugriff auf den Hauptspeicher ist von allen Kernen

gleich schnell. Man spricht in diesem Zusammenhang von einer UMA-Architektur (*Uniform Memory Access*).

Neben den hier aufgeführten allgemeinen Unterscheidungsmerkmalen gibt es noch weitere, herstellerspezifische Erweiterungsebenen. Genannt sei hier z. B. das von Intel eingeführte Hyper-Threading. Dabei werden Lücken in der Fließbandverarbeitung mit Befehlen von anderen Prozessen möglichst aufgefüllt.

Hinweis

In dem vorliegenden Buch werden wir uns ausschließlich mit den Konzepten und Programmiermodellen für Multicore- bzw. Multiprocessor-Systeme mit Zugriff auf einen gemeinsam benutzten Hauptspeicher befassen, wobei wir auf die Besonderheiten der NUMA-Architektur nicht eingehen. Bei Java hat man außer der Verwendung der beiden VM-Flags `-XX:+UseNUMA` und `-XX:+UseParallelGC` kaum Einfluss auf das Speichermanagement.

1.2 Parallelität und Nebenläufigkeit

Zwei oder mehrere Aktivitäten (*Tasks*) heißen *nebenläufig*, wenn sie zeitgleich bearbeitet werden können. Dabei ist es unwichtig, ob zuerst der eine und dann der andere ausgeführt wird, ob sie in umgekehrter Reihenfolge oder gleichzeitig erledigt werden. Sie haben keine kausale Abhängigkeit, d.h., das Ergebnis einer Aktivität hat keine Wirkung auf das Ergebnis einer anderen und umgekehrt. Das Abstraktionskonzept für Nebenläufigkeit ist bei Java der *Thread*, der einem eigenständigen Kontrollfluss entspricht.

Besitzt ein Rechner mehr als eine CPU bzw. mehrere Rechenkerne, kann die Nebenläufigkeit parallel auf Hardwareebene realisiert werden. Dadurch besteht die Möglichkeit, die Abarbeitung eines Programms zu beschleunigen, wenn der zugehörige Kontrollfluss nebenläufige Tasks (Aktivitäten) beinhaltet. Dabei können moderne Hardware und Übersetzer nur bis zu einem gewissen Grad automatisch ermitteln, ob Anweisungen sequenziell oder parallel (gleichzeitig) ausgeführt werden können. Damit Programme die Möglichkeiten der Multicore-Prozessoren voll ausnutzen können, müssen wir die Parallelität explizit im Code berücksichtigen.

Die nebenläufige bzw. parallele Programmierung beschäftigt sich zum einen mit Techniken, wie ein Programm in einzelne, nebenläufige Abschnitte/Teilaktivitäten zerlegt werden kann, zum anderen mit den verschiedenen Mechanismen, mit denen nebenläufige Abläufe synchronisiert und gesteu-

ert werden können. So schlagen z. B. Mattson et al. in [37] ein »patternbasiertes« Vorgehen für das Design paralleler Anwendungen vor. Ähnliche Wege werden auch in [7] oder [38] aufgezeigt. Spezielle Design-Patterns für die nebenläufige Programmierung findet man in [15, 38, 42, 45].

1.2.1 Die Vorteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit ermöglicht die Anwendung verschiedener neuer Programmierkonzepte. Der offensichtlichste Vorteil ist die Steigerung der Performance. Auf Maschinen mit mehreren CPUs kann zum Beispiel das Sortieren eines großen Arrays auf mehrere Threads verteilt werden. Dadurch kann die zur Verfügung stehende Rechenleistung voll ausgenutzt und somit die Leistungsfähigkeit der Anwendung verbessert werden. Ein weiterer Aspekt ist, dass Threads ihre Aktivitäten unterbrechen und wiederaufnehmen können. Durch Auslagerung der blockierenden Tätigkeiten in separate Threads kann die CPU in der Zwischenzeit andere Aufgaben erledigen. Hierdurch ist es möglich, asynchrone Schnittstellen zu implementieren und somit die Anwendung reaktiv zu halten. Dieser Gesichtspunkt gewinnt immer mehr an Bedeutung.

1.2.2 Die Nachteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit hat aber nicht nur Vorteile. Er kann unter Umständen sogar mehr Probleme verursachen, als damit gelöst werden. Programmcode mit Multithreading-Konzepten ist nämlich oft schwer zu verstehen und mit hohem Aufwand zu warten. Insbesondere wird das Debugging erschwert, da die CPU-Zuteilung an die Threads nicht deterministisch ist und ein Programm somit jedes Mal verschieden verzahnt abläuft.

Parallel ablaufende Threads müssen koordiniert werden, sodass man immer mehrere Programmflüsse im Auge haben muss, insbesondere wenn sie auf gemeinsame Daten zugreifen. Wenn eine Variable von einem Thread geschrieben wird, während der andere sie liest, kann das dazu führen, dass das System in einen falschen Zustand gerät. Für gemeinsam verwendete Objekte müssen gesondert Synchronisationsmechanismen eingesetzt werden, um konsistente Zustände sicherzustellen. Des Weiteren kommen auch Cache-Effekte hinzu. Laufen zwei Threads auf verschiedenen Kernen, so besitzt jeder seine eigene Sicht auf die Variablenwerte. Man muss nun dafür Sorge tragen, dass gemeinsam benutzte Daten, die aus Performance-Gründen in den Caches gehalten werden, immer synchron bleiben. Weiter ist es möglich, dass sich Threads gegenseitig in ihrem Fortkommen behindern oder sogar verklemmen.

1.2.3 Sicherer Umgang mit Nebenläufigkeit

Den verschiedenen Nachteilen versucht man durch die Einführung von Parallelisierungs- und Synchronisationskonzepten auf höherer Ebene entgegenzuwirken. Ziel ist es, dass Entwickler möglichst wenig mit *Low-Level*-Synchronisation und Thread-Koordination in Berührung kommen. Hierzu gibt es verschiedene Vorgehensweisen. So wird z. B. bei C/C++ mit OpenMP[1] die Steuerung der Parallelität deklarativ über `#pragma` im Code verankert. Der Compiler erzeugt aufgrund dieser Angaben parallel ablaufenden Code. Die Sprache Cilk erweitert C/C++ um neue Schlüsselworte, wie z. B. `cilk_for`[2].

Java geht hier den Weg über die Bereitstellung einer »*Concurrency*-Bibliothek«, die mit Java 5 eingeführt wurde und sukzessive erweitert wird. Nachdem zuerst Abstraktions- und Synchronisationskonzepte wie *Threadpools*, *Locks*, *Semaphore* und *Barrieren* angeboten wurden, sind mit Java 7 und Java 8 auch Parallelisierungsframeworks hinzugekommen. Nicht vergessen werden darf hier auch die Einführung Thread-sicherer Datenstrukturen, die unverzichtbar bei der Implementierung von Multithreaded-Anwendungen sind. Der Umgang mit diesen *High-Level*-Abstraktionen ist bequem und einfach. Nichtsdestotrotz gibt es auch hier Fallen, die man nur dann erkennt, wenn man die zugrunde liegenden *Low-Level*-Konzepte beherrscht. Deshalb werden im ersten Teil des Buches die Basiskonzepte ausführlich erklärt, auch wenn diese im direkten Praxiseinsatz immer mehr an Bedeutung verlieren.

1.3 Maße für die Parallelisierung

Neben der Schwierigkeit, korrekte nebenläufige Programme zu entwickeln, gibt es auch inhärente Grenzen für die Beschleunigung durch Parallelisierung. Eine wichtige Maßzahl für den Performance-Gewinn ist der *Speedup* (Beschleunigung bzw. Leistungssteigerung), der wie folgt definiert ist:

$$S = \frac{T_{seq}}{T_{par}}$$

Hierbei ist T_{seq} die Laufzeit mit einem Kern und T_{par} die Laufzeit mit mehreren.

1.3.1 Die Gesetze von Amdahl und Gustafson

Eine erste Näherung für den *Speedup* liefert das Gesetz von Amdahl [2]. Hier fasst man die Programmteile zusammen, die parallel ablaufen können.

[1] Siehe `http://www.openmp.org`.

[2] Siehe `http://www.cilkplus.org`.

Wenn P der prozentuale, parallelisierbare Anteil ist, dann entspricht $(1-P)$ dem sequenziellen, nicht parallelisierbaren. Hat man nun N Prozessoren bzw. Rechenkerne zur Verfügung, so ergibt sich der maximale *Speedup*

$$S(N) = \frac{\text{Sequenzielle Laufzeit}}{\text{Parallele Laufzeit}} = \frac{1}{\frac{P}{N} + (1-P)},$$

wobei hier implizit davon ausgegangen wird, dass die Parallelisierung einen konstanten, vernachlässigbaren, internen Verwaltungsaufwand verursacht. Durch Grenzwertbildung $N \to \infty$ ergibt sich dann der theoretisch maximal erreichbare *Speedup* beim Einsatz von unendlich vielen Kernen bzw. Prozessoren zu

$$\lim_{N\to\infty} S(N) = \lim_{N\to\infty} \frac{1}{\frac{P}{N} + (1-P)} = \frac{1}{(1-P)}.$$

An der Formel sieht man, dass der nicht parallelisierbare Anteil den *Speedup* begrenzt. Beträgt der parallelisierbare Anteil z.B. nur 50%, so kann nach dem Amdahl'schen Gesetz maximal nur eine Verdopplung der Ausführungsgeschwindigkeit erreicht werden (vgl. Abb. 1-1).

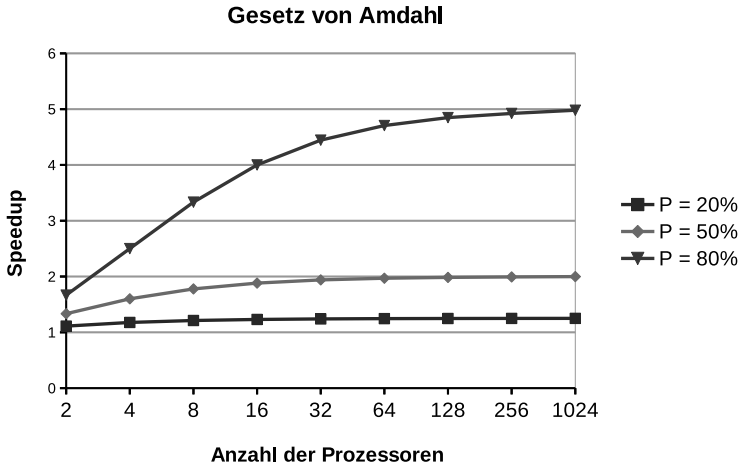

Abbildung 1-1: *Speedup* in Abhängigkeit von P und N

Man kann die Parallelisierung aber auch unter einem anderen Gesichtspunkt betrachten. Amdahl geht von einem fest vorgegebenen Programm bzw. einer fixen Problemgröße aus. Gustafson betrachtet dagegen eine variable Problemgröße in einem festen Zeitfenster [18]. Er macht die Annahme, dass sich die Vergrößerung des zu berechnenden Problems im Wesentlichen üblicherweise nur auf den parallelisierbaren Programmteil P auswirkt (man sagt, die Anwendung ist skalierbar). Unter diesem Aspekt er-

gibt sich ein *Speedup* von

$$S(N) = (1 - P) + N \cdot P$$

d.h., der Zuwachs ist hier proportional zu N.

Die unterschiedlichen Sichtweisen zwischen Amdahl und Gustafson sind in der Abbildung 1-2 verdeutlicht.

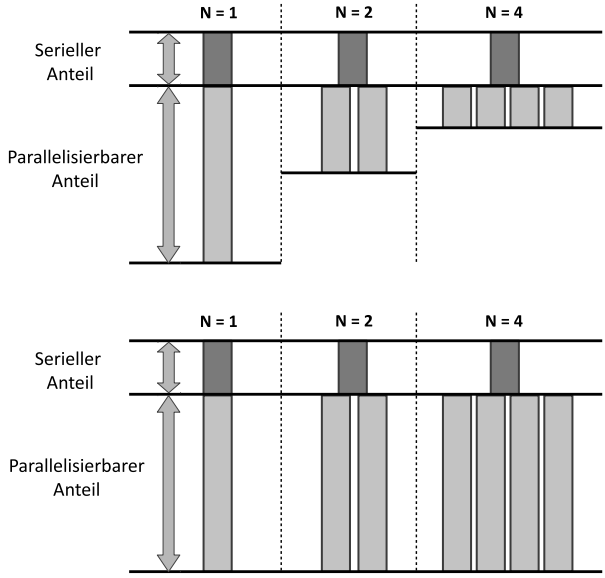

Abbildung 1-2: Amdahl (oben) versus Gustafson (unten)

1.3.2 Work-Span-Analyse

Eine weitere Methode, den Grad einer Parallelisierung zu beschreiben, ist die *Work-Span*-Analyse [10]. In dem zugrunde liegenden Modell werden die Abhängigkeiten der auszuführenden Aktivitäten in einem azyklischen Graphen dargestellt (vgl. Abb. 1-3). Eine Aktivität kann hier erst dann ausgeführt werden, wenn alle »Vorgänger« abgeschlossen sind.

Die von dem Algorithmus zu leistende Gesamtarbeit ist die Summe der auszuführenden Aktivitäten. Man bezeichnet die benötigte Zeit (*work*) hierfür mit T_1. Der sogenannte *span*, der mit T_∞ bezeichnet wird, entspricht dem kritischen Pfad, also dem längsten Weg von Aktivitäten, die nacheinander ausgeführt werden müssen[3].

[3] In der Literatur wird der *span* auch manchmal als *step complexity* oder *depth* bezeichnet.

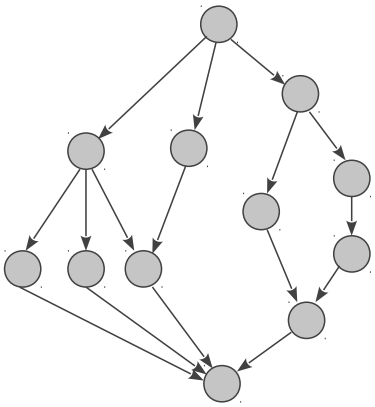

Abbildung 1-3: Azyklischer Aktivitätsgraph

Wenn wir uns den Aktivitätsgraphen in Abbildung 1-3 anschauen und annehmen, dass jede Aktivität eine Zeiteinheit dauert, so erhalten wir für den *work* $T_1 = 12$ und den *span* $T_\infty = 6$. Sei N wieder die Anzahl der Rechenkerne bzw. Prozessoren, dann erhält man als Speedup:

$$S(N) = \frac{T_1}{T_N} \leq N.$$

Der Speedup wächst linear mit der Anzahl der Prozessoren, vorausgesetzt dass die CPU immer voll ausgelastet ist (*greedy scheduling*). Der Speedup ist allerdings durch den *span* begrenzt, da der kritische Pfad sequenziell abgearbeitet werden muss:

$$S(N) = \frac{T_1}{T_N} \leq \frac{T_1}{T_\infty} = \frac{\mathit{work}}{\mathit{span}}.$$

In unserem Beispiel beträgt der maximal erreichbare Speedup $T1/T_\infty = 2$.

1.4 Parallelitätsmodelle

In der Literatur wird zwischen verschiedenen Modellen für die Parallelisierung unterschieden. Java unterstützt jedes dieser Modelle durch das Bereitstellen verschiedener Konzepte und APIs.

Zur Parallelisierung von Anwendungen gibt es grundsätzlich zwei Ansätze: Daten- und Task-Parallelität[4]. Bei der *Datenparallelität* wird ein Datenbestand geteilt und die Bearbeitung der Teilbereiche verschiedenen Threads zugeordnet. Hierbei führt jeder Thread dieselben Operationen aus.

[4]Die beiden Parallelisierungskonzepte werden ausführlich in [15] diskutiert.

Diese Art der Parallelisierung wird durch das Gesetz von Gustafson beschrieben und ist in der Regel gut skalierbar [53]. Mit dem ForkJoin-Framework und dem `Stream`-API stehen bei Java hierfür zwei leistungsfähige Möglichkeiten zur Verfügung (siehe Kapitel 13 und 14). Falls man diese Frameworks nicht einsetzen möchte, kann für eine explizite Umsetzung auf zahlreiche Synchronisationskonzepte zurückgegriffen werden (siehe Kapitel 11 und 12).

Bei der *Task-Parallelität*[5] wird die Anwendung in Funktionseinheiten zerlegt, die dann bezüglich ihrer Abhängigkeiten ausgeführt werden. Diese Art der Parallelisierung wird durch die *Work-Span*-Analyse beschrieben und kann bei Java mithilfe der `CompletableFuture`-Klasse oder je nachdem auch mit dem ForkJoin-Framework realisiert werden (siehe Kapitel 13 und 15).

Neben diesen beiden grundsätzlichen Ansätzen wird auch oft noch zwischen dem *Master-Slave*-, dem *Work-Pool*- und dem *Erzeuger-Verbraucher*- bzw. *Pipeline*-Programmiermuster unterschieden [32]. Das Unterscheidungsmerkmal ist hierbei die Art und Weise, wie die beteiligten Komponenten miteinander kommunizieren. Beim *Master-Slave*-Modell gibt es einen dedizierten Thread, der Aufgaben an andere verteilt und dann die Ergebnisse einsammelt. Bei Java kann dieses Modell mit dem `Future`-Konzept umgesetzt werden (siehe Abschnitt 6.2). Das *Work-Pool*-Modell entspricht dem `ExecutorService`, dem man Aufgaben zur Ausführung delegieren kann (siehe Abschnitt 6.1). Das bewährte *Erzeuger-Verbraucher*-Modell wird typischerweise durch `BlockingQueue`-Datenstrukturen realisiert und existiert in verschiedenen Varianten (siehe Abschnitt 10.3). In der Praxis findet man häufig Kombinationen der verschiedenen Modelle bzw. Muster.

[5] Genauer müsste man eigentlich »funktionale Dekomposition« (*functional decomposition*) sagen, da der Begriff Task-Parallelität oft auf alles Mögliche angewendet wird.