Kapitel 1

Grundsätzliches über Parallelrechner

In diesem Kapitel besprechen wir einleitend den Aufbau und die Leistungsbeurteilung von Parallelrechnern. Es schließt sich eine Diskussion von Programmiermodellen an.

1.1 Rechnertypen und Architekturen

Ein serieller Rechner besteht aus einer Zentraleinheit oder CPU (*central processing unit*), der ein Hauptspeicher zugeordnet ist und die mit Terminals, Druckern, Platten und anderen externen Speichermedien verbunden ist (Abbildung 1.1).

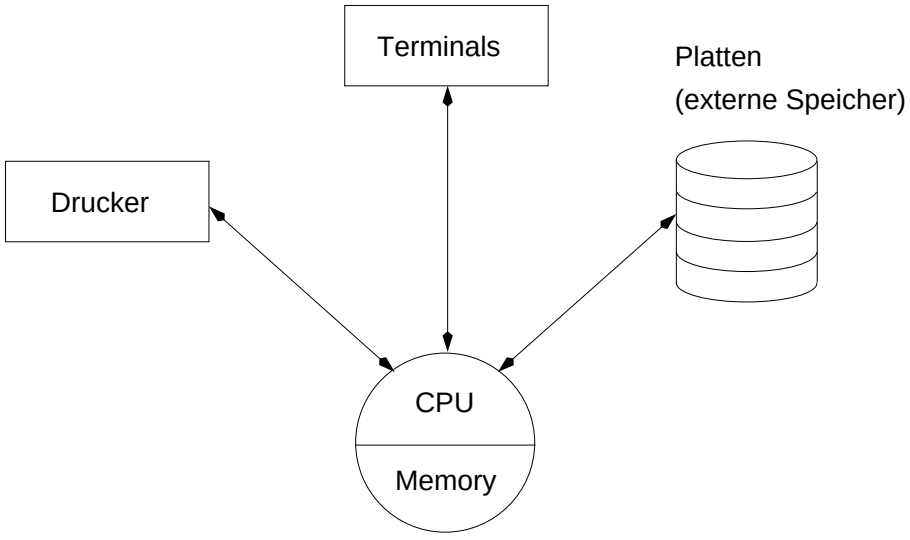

Abb. 1.1: Modell eines seriellen Rechners.

Parallelrechner hingegen bestehen aus mehreren Prozessoren, die die gleichen oder unterschiedlichen Befehle meist mit verschiedenen Daten gleichzeitig durchführen

können. Grundprinzip ist immer das Aufteilen des Rechenpensums in Teilaufgaben, die dann von mehreren Prozessoren gleichzeitig, also parallel, abgearbeitet werden. Hier gibt es verschiedene Konzepte, die nachfolgend kurz vorgestellt werden.

1.1.1 Verteilter und gemeinsamer Speicher

Ein Unterscheidungsmerkmal von Parallelrechnern ist die Art und Möglichkeit des Datenzugriffs seitens der einzelnen Prozessoren. Bei Rechnern mit verteiltem Speicher (*distributed memory*) hat jeder Prozessor P_i, $i = 1, \ldots, n$, einen eigenen lokalen Speicher M_i $i = 1, \ldots, n$, auf den nur er zugreifen kann (Abbildung 1.2).

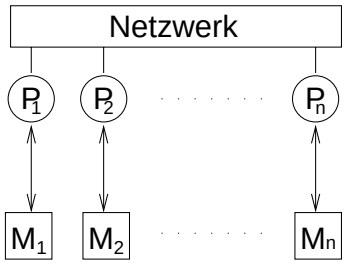

Abb. 1.2: Parallelrechner mit verteiltem Speicher.

Datenaustausch zwischen den einzelnen Prozessoren erfolgt durch explizites Versenden von Nachrichten (*message passing*) über ein Netzwerk. Zu dieser Klasse kann man auch ein Cluster von Einzelrechnern zählen, wie es z.B. in Anhang A beschrieben ist.

Im Gegensatz dazu ermöglichen Rechner mit gemeinsamem Speicher (*shared memory*) jedem Prozessor den Zugriff auf diesen gemeinsamen Speicher, wodurch auch der Datenaustausch zwischen den Prozessoren erfolgt (Abbildung 1.3).

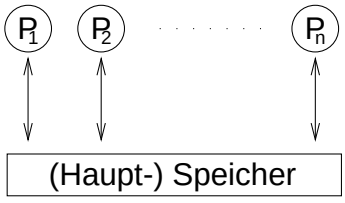

Abb. 1.3: Parallelrechner mit gemeinsamem Speicher.

Gewöhnlich ist die Programmierung bei Rechnern mit gemeinsamem Speicher einfacher als bei Rechnern mit verteiltem Speicher, da alle Prozessoren auf diesen gemeinsamen Speicher zugreifen können und das Versenden der Daten nicht vom Benutzer gesteuert werden muss. Solche Rechner besitzen oft nur eine kleine Anzahl (meist 2 bis 32) von Prozessoren, wodurch im praktischen Einsatz Schwierigkeiten beim

Datenzugriff vermieden werden. Diese Probleme bestehen bei Rechnern mit verteiltem Speicher nicht, da die meisten Daten nur zwischen Prozessor und lokalem Speicher wie bei einem sequentiellen Rechner transportiert werden. Dies ist aber für die praktische Programmierung ein wesentlicher Nachteil: Wenn Daten von anderen Prozessoren benötigt werden, sind diese durch spezielle Befehle zum Versenden über das Netzwerk zu transportieren. Vorteile beider Rechner werden durch eine Architektur kombiniert, bei der der Benutzer glaubt, einen Rechner mit gemeinsamem Speicher zu verwenden, so dass er sich nicht um das explizite Versenden der Daten kümmern muss. Dabei sind alle lokalen Speicher mit einem Suchprozessor verbunden. Ein solcher Rechner ist z.B. der KSR-1. Er wurde vor mehreren Jahren von der mittlerweile nicht mehr existierenden Firma Kendall Square Research gebaut.

1.1.2 Granularität

Ein zweites Unterscheidungsmerkmal ist die Granularität der Architektur des Parallelrechners. Man unterscheidet zwischen grob- und feingranularer Architektur. In einer grobgranularen Architektur (z.B. IBM SP2, Cray Y-MP oder PC-Cluster) kann jeder Prozessor ein relativ großes Rechenpensum erledigen, wobei er verhältnismäßig selten Daten mit anderen austauschen sollte. Bei feingranularer Architektur (z.B. Maspar) ist das Arbeitspensum eines Einzelprozessors eher klein, der Einzelprozessor weniger leistungsfähig und die Kommunikation häufiger. Erstere verwenden typischerweise das Konzept des *message passing*, bei letzteren ist eine sehr leistungsfähige Kommunikation nötig, was beispielsweise durch *virtual shared memory* realisiert wird.

1.1.3 Prozessortopologie

Ein drittes Unterscheidungsmerkmal ist die Anordnung der Prozessoren untereinander, die Prozessortopologie. Einige Beispiele sind:

- Prozessorring, wobei jeder Prozessor genau zwei Nachbarprozessoren hat (Abbildung 1.4).
- Hypercube mit 2^k Prozessoren, wobei jeder Prozessor mit k weiteren Prozessoren so verbunden ist, dass die Differenz der Prozessornummern benachbarter Prozessoren jeweils eine Potenz von zwei ist. Der 2^k-cube entsteht aus zwei 2^{k-1}-cubes dadurch, dass jeweils ein Knoten des ersten 2^{k-1}-cubes mit einem Knoten des zweiten 2^{k-1}-cubes so verbunden wird, dass sich diese Prozessornummern, jeweils um 2^{k-1} unterscheiden. (Abbildung 1.5)
- Binärbaum mit $2^p - 1$ Prozessoren, davon ein Wurzelknoten, $2^{p-1} - 2$ Prozessoren, die jeweils mit einem Vater und zwei Söhnen verbunden sind, und 2^{p-1} Blättern, die nur mit je einem Vater verbunden sind (Abbildung 1.6).

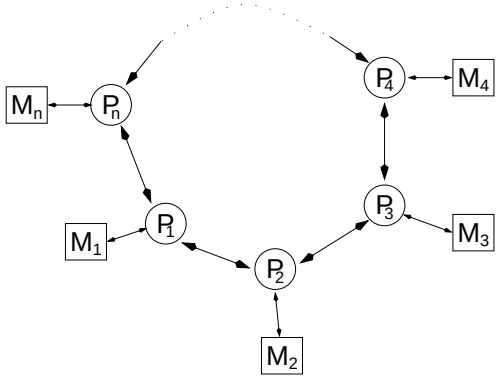

Abb. 1.4: Prozessorring für n Prozessoren mit verteiltem Speicher.

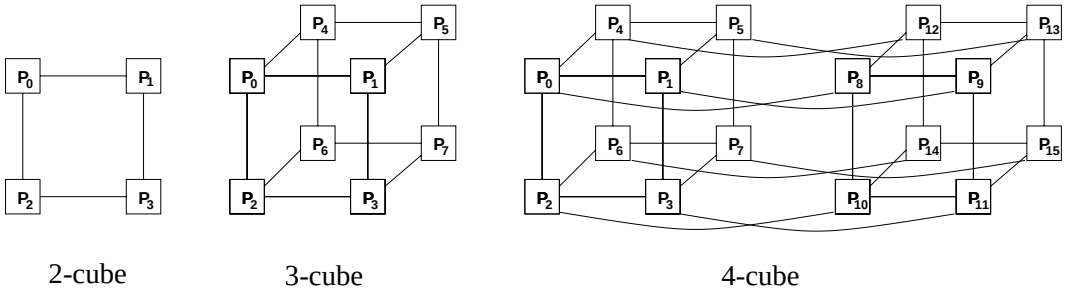

Abb. 1.5: Hypercube-Architektur für $k = 2, 3, 4$.

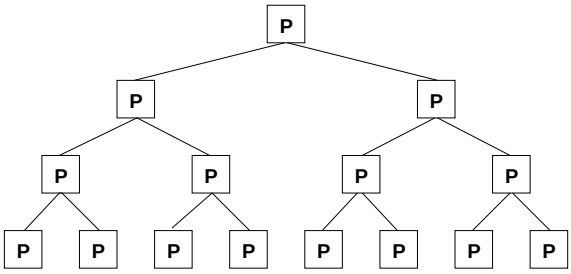

Abb. 1.6: Binärbaum ($p = 4$)

- Zweidimensionales Gitter mit meist k^2 Prozessoren, wobei außer den Prozessoren am Rand jeder vier Nachbarn hat (Abbildung 1.7).
- Torus, bestehend aus einem zweidimensionalen Gitter, bei dem die Randprozessoren so miteinander verbunden werden, dass in jeder der beiden Raumdimensionen Prozessorringe entstehen und jeder Prozessor vier Nachbarn hat (Abbildung 1.8).
- Dreidimensionales Gitter, bestehend aus mehreren Lagen von zweidimensionalen Gittern; außer den Randprozessoren sind alle Prozessoren mit sechs weiteren verbunden.

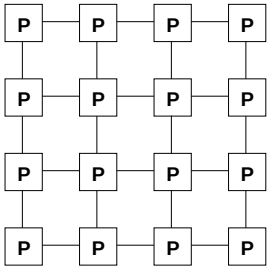

Abb. 1.7: 2D-Gitter ($k = 4$)

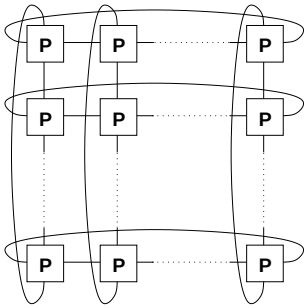

Abb. 1.8: Torus: Jeder Prozessor ist mit seinen vier unmittelbaren Nachbarn verbunden.

- Crossbar-Topologie: Jeder Knoten ist mit jedem anderen des Systems verbunden. Dies ist nur für kleine Prozessorzahlen sinnvoll, da die Anzahl der erforderlichen Verbindungen wesentlich schneller als die Prozessorzahl ansteigt (Abbildung 1.9).

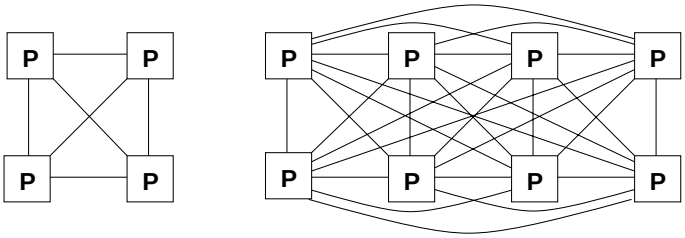

Abb. 1.9: Crossbar-Topologie für vier und acht Prozessoren.

Weitere, zum Teil komplexere Architekturen sind in der Praxis zu finden. Zum Beispiel besteht der am Rechenzentrum der Universität Karlsruhe betriebene Parallelrechner IBM RS/6000 SP2 aus einem mehrstufigen Netzwerk, aufgebaut aus so genannten Frames. Dabei bilden 4 bis 16 Prozessoren mit Crossbar-Topologie einen Frame. Ein Rechnersystem besteht aus mehreren Frames. Zur Kommunikation werden so genannte *high performance switches* eingesetzt. Eine Beispielkonfiguration ist in Abbildung 1.10 dargestellt.

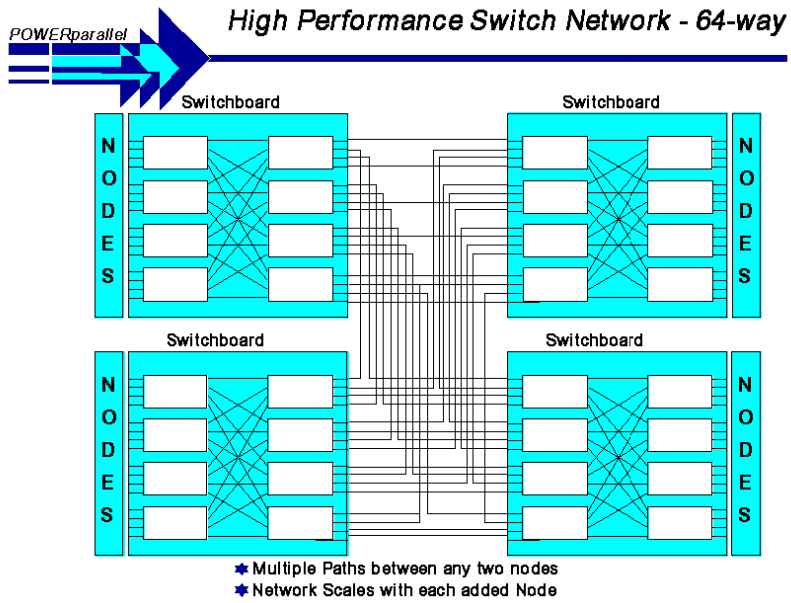

Abb. 1.10: Prozessorkonfiguration einer IBM SP2 mit 4 Frames zu je 16 Prozessoren und High Performance Switch.

1.1.4 SIMD- und MIMD-Rechner

Ein weiteres Unterscheidungsmerkmal von Parallelrechnern ist die Eigenständigkeit der Einzelprozessoren. SIMD-Rechner (*single instruction, multiple data*) ermöglichen nur die Durchführung der gleichen Operationen auf mehreren Prozessoren mit unterschiedlichen Daten. Dabei arbeitet ein zentraler Kontrollprozessor ein Programm ab und verteilt das Rechenpensum auf die anderen Prozessoren. In Abbildung 1.11 wird die Addition zweier Matrizen $\mathbf{A}+\mathbf{B}$ auf einem SIMD-Parallelrechner dargestellt, wobei jedem Prozessor genau ein Matrixelement zugeordnet wird. Falls weniger als n^2 Prozessoren vorhanden sind, können die Matrizen $\mathbf{A}$ und $\mathbf{B}$ in Blockkomponenten zerlegt werden. Auf den einzelnen Prozessoren wird dann eine Blockaddition durchgeführt. Bei MIMD-Rechnern (*multiple instruction, multiple data*) kann

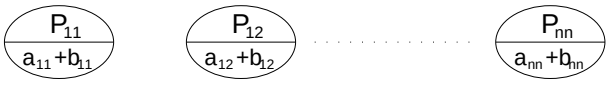

Abb. 1.11: Addition zweier Matrizen auf einem SIMD-Parallelrechner.

jeder Prozessor ein eigenes Programm mit eigenen Daten abarbeiten. Es können also gleichzeitig verschiedene Operationen mit unterschiedlichen Daten ausgeführt werden (Abbildung 1.12).

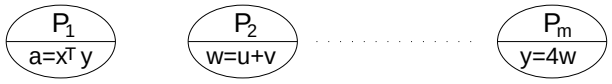

Abb. 1.12: Beispiele für Operationen auf einem MIMD-Parallelrechner

1.2 Leistungsbeurteilung von Parallelrechnern

Um Parallelrechner für ein vorgegebenes Problem effizient einzusetzen, sollten möglichst viele Programmteile zeitgleich abgearbeitet werden können. Man erhofft sich durch den Einsatz von Parallelrechnern entweder einen deutlichen Rechenzeitgewinn gegenüber sequentiellen Berechnungen oder die Möglichkeit der Bearbeitung von wesentlich größeren Problemen, die aus feineren Diskretisierungen oder komplexeren Problemen resultieren können und wegen des hohen Speicherbedarfs nicht sequentiell bearbeitet werden können.

1.2.1 Parallelisierungsgrad

Definition 1.1
Der Parallelisierungsgrad *eines Algorithmus ist die Anzahl der parallel ausführbaren Operationen.*

Beispielsweise hat die Addition zweier n-komponentiger Vektoren den Parallelisierungsgrad n, da die n Additionen unabhängig voneinander und somit zeitgleich ausgeführt werden können.

Dagegen hat die Berechnung der Summe

$$s = \sum_{i=1}^{n} s_i,$$

wobei s_i auf Prozessor P_i vorliegt, unter Verwendung der Fan-In-Methode nicht den vollen Parallelisierungsgrad. Die Vorgehensweise wird für $n = 8$ in Abbildung 1.13 graphisch dargestellt.

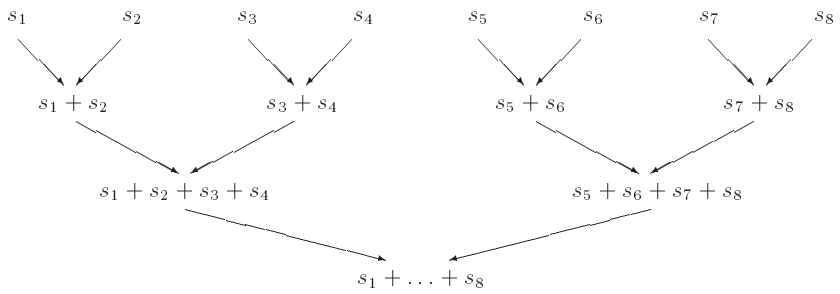

Abb. 1.13: Fan-In-Methode für 8 Prozessoren.

Für $n = 8 = 2^3$ erfolgt die Berechnung in 3 Stufen. In der ersten Stufe werden insgesamt vier Additionen, in der zweiten zwei und in der letzten eine durchgeführt. Für $n = 2^k$ werden $k = \log_2 n$ Stufen benötigt und jede Stufe hat einen anderen Parallelisierungsgrad. Der durchschnittliche Parallelisierungsgrad beträgt hier

$$\frac{2^k - 1}{k} = \frac{n-1}{\log_2 n}.$$

1.2.2 Speed-up und Effizienz

Zur Charakterisierung der Leistungsfähigkeit eines parallelen Programms wird meist der *Speed-up* verwendet.

Definition 1.2
Es sei T_s die Zeit, die man mit dem schnellsten bekannten seriellen Algorithmus zur Lösung eines gegebenen Problems auf einem Prozessor benötigt und T_P die Zeit, die man zur Lösung des gleichen Problems auf dem Parallelrechner mit P solchen Prozessoren benötigt. Der Speed-up *eines parallelen Algorithmus ist*

$$S = \frac{T_s}{T_P}. \tag{1.1}$$

In der Praxis findet man oft auch die Definition

$$S = \frac{T_1}{T_P}, \tag{1.2}$$

wobei T_1 die Rechenzeit auf einem Prozessor ist und der gleiche Algorithmus verwendet wird.

Bemerkung 1.1
Gewöhnlich gilt $T_1 \neq T_s$, meistens ist $T_1 \geq T_s$.

Eng verknüpft mit der Definition des Speed-up ist die Effizienz eines parallelen Programms. Sie ist ein Maß für die Auslastung der einzelnen Prozessoren.

Definition 1.3
Die Effizienz eines parallelen Algorithmus bei einer Berechnung mit P Prozessoren ist

$$e = \frac{S}{P},$$

wobei für S die Definition (1.2) zugrunde liegt.

Die Effizienz lässt sich folgendermaßen interpretieren: Ist e groß, d.h. nahe bei 1, so ist der Parallelrechner durch den verwendeten Algorithmus gut genutzt. Im Idealfall ist ein Algorithmus vollständig parallelisierbar. Dann hat man eine ideale Beschleunigung im Vergleich zur seriellen Berechnung, d.h. $S = P$ und das Problem kann

durch den Algorithmus in der Zeit $\frac{1}{P} \cdot T_1$ gelöst werden. In diesem Fall gilt $e = 1$. Im Allgemeinen ist aber $e < 1$, da in Anwendungen nicht 100% eines Algorithmus parallelisierbar sind und in der Regel Daten zwischen den Prozessoren auszutauschen sind, so dass durch die entsprechende Kommunikation zusätzlich Zeit benötigt wird.

Oft ist aufgrund der Problemgröße eine sequentielle Berechnung nicht mehr möglich, so dass T_s oder T_1 nicht vorliegt. In solchen Fällen macht der *inkrementelle Speed-up*

$$S_i(P) = \frac{\text{Laufzeit auf } \frac{P}{2} \text{ Prozessoren}}{\text{Laufzeit auf } P \text{ Prozessoren}}$$

eine entsprechende Aussage über die Qualität des parallelen Verfahrens, der im optimalen Fall den Wert 2 hat.

1.2.3 Das Gesetz von Amdahl

Wie sich der nicht parallelisierbare Anteil eines Programms auf die Gesamtrechenzeit auswirkt, besagt das Gesetz von Amdahl (Amdahl's law).

Satz 1.1 (Gesetz von Amdahl)
Sei α der prozentuale sequentielle Anteil des Algorithmus. Der restliche Anteil $1-\alpha$ sei mit dem Parallelisierungsgrad P ausführbar, wobei P die Anzahl der Prozessoren ist. Dann beträgt der Speed-up

$$S = \frac{1}{\alpha + (1-\alpha)/P}.$$

Beweis
Der Anteil $1-\alpha$ sei mit Parallelisierungsgrad P parallelisierbar. Die Anzahl der Operationen zur gesamten Problemlösung sei N und die Zeit für eine Operation sei τ. Die benötigte Zeit auf einem Parallelrechner mit P Prozessoren setzt sich nun zusammen aus dem nicht parallelisierbaren Anteil mit der Laufzeit $\alpha N\tau$ und der Zeit des parallelisierten Anteils:

$$T_P = \alpha N\tau + (1-\alpha)\frac{N\tau}{P}.$$

Für den Speed-up erhält man damit

$$S = \frac{T_1}{T_P} = \frac{N\tau}{\alpha N\tau + (1-\alpha)\frac{N\tau}{P}} = \frac{1}{\alpha + (1-\alpha)/P}\ .$$

□

Bemerkung 1.2
Wird ein Algorithmus zu 99 Prozent mit dem optimalen Parallelisierungsgrad P parallelisiert, so ist $\alpha = 0.01$. Dann ist mit 10 Prozessoren höchstens ein Speed-up $S = \frac{1}{0.109} \approx 9.17$ erreichbar. Mit 100 Prozessoren kann aber höchstens noch

$S = \frac{1}{0.0199} \approx 50.25$ *erzielt werden. Dabei ist die Zeit für eventuell erforderliche Kommunikation nicht berücksichtigt. Für große Prozessorenzahl wird eine Sättigung erreicht (Abbildung 1.14), denn es gilt*

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

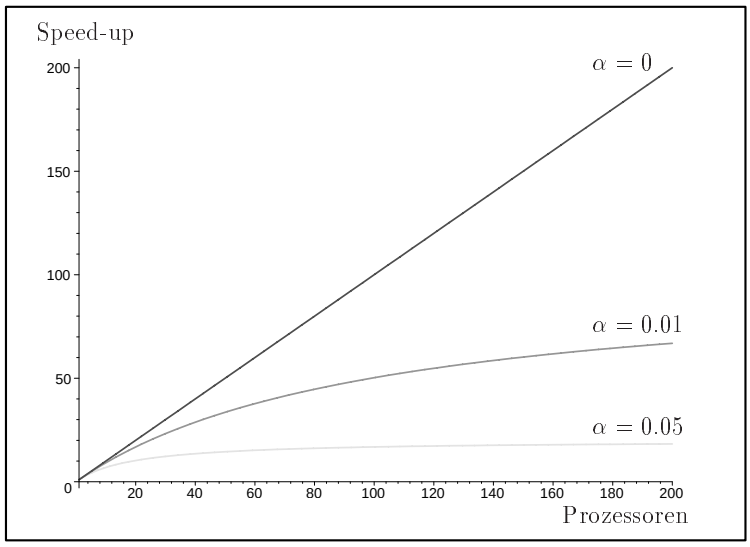

Abb. 1.14: Maximal erreichbarer Speed-up nach dem Gesetz von Amdahl.

1.3 Parallele Programmiermodelle

In diesem Abschnitt beschäftigen wir uns im Hinblick auf das parallele Programmieren mit so genannten Programmiermodellen, das sind Abstraktionen von real existierenden Rechnersystemen. Mit dem Begriff des Rechnersystems bezeichnen wir dabei die Gesamtheit der Hardware und der darauf zur Verfügung stehenden Systemsoftware wie Compilern und Laufzeitbibliotheken. Die Modelle sollen dabei einfach sein, um die Rechnersysteme in möglichst große Klassen unterteilen zu können, die realen Rechnersysteme gleichzeitig aber auch gut beschreiben.

Die nachfolgende Darstellung lehnt sich eng an das Vorgehen in [29] an.

Gründe für die Verwendung von Programmiermodellen sind:

- Sie bieten eine Abstraktion von der speziellen Hardware und ermöglichen somit parallele Algorithmen herstellerunabhängig aufzuschreiben.
- Sie erlauben eine Abschätzung der Laufzeit des parallelen Programms und somit einen Vergleich der Effizienz verschiedener paralleler Algorithmen.

- Sie gestatten asymptotische Aussagen über Algorithmen, die man auf Rechenanlagen nicht gewinnen kann.

Ein Programmiermodell beschreibt unter anderem:

- Die vom Programmierer nutzbaren Ebenen der Parallelität (z.B. Parallelität auf Instruktionsebene, Parallelität in Schleifen, Parallelität auf Prozessebene).
- Ob die Parallelität implizit oder explizit programmiert werden kann.
- Wie die parallel ablaufenden Prozesse erzeugt werden.
- Wie der Datenaustausch zwischen Prozessen erfolgt.

1.3.1 Ebenen der Parallelität

Bei Programmen unterscheidet man verschiedene Ebenen, auf denen Befehle parallele ausgeführt werden können. Die Unterteilung in diese Ebenen erfolgt anhand der so genannten *Granularität*. Der Begriff der Granularität beschreibt, wie viele Instruktionen parallel und unabhängig voneinander ausgeführt werden können. Von grobkörniger Granularität spricht man, wenn sehr viele Instruktionen parallel ausgeführt werden können, bevor Datenaustausch nötig ist. Ein Beispiel für grobkörnige Granularität ist die Parallelität auf Prozessebene. Bei feinkörniger Granularität können nur wenige Instruktionen parallel ausgeführt werden, bevor wieder Kommunikation nötig ist. Ein Beispiel für feinkörnige Granularität ist die Parallelität auf Instruktionsebene. Datenparallelität und parallele Schleifen weisen üblicherweise eine mittlere Granularität auf.

Parallelität auf Instruktionsebene

Bei der Abarbeitung eines Programms können mehrere Instruktionen, wenn diese voneinander unabhängig sind, parallel in einem Prozessor ausgeführt werden.

Das folgende Beispiel verdeutlicht diese Möglichkeit bei Prozessoren mit mehreren Ausführungseinheiten: Es soll der Ausdruck $c = a + b$ berechnet werden. Ein Prozessor mit nur einer Ausführungseinheit führt dazu die Instruktionen

```
(1) Lade a in Register1
(2) Lade b in in Register2
(3) Addiere Register1 und Register2 und lege Ergebnis in Register1 ab
(4) Schreibe Register1 nach c
```

aus. Ein Prozessor mit zwei Ausführungseinheiten berechnet hingegen in kürzerer Zeit:

(1) Lade a in Register1 Lade b in in Register2
(2) Addiere Register1 und Register2 und lege Ergebnis in Register1 ab
(3) Schreibe Register1 nach c

Die Verteilung der einzelnen Instruktionen auf die Berechnungseinheiten erfolgt zumeist durch einen in Hardware auf der CPU realisierten Instruktionsscheduler.

Bei VLIW-Prozessoren (Very-Long-Instruction-Word), wie etwa dem Itanium-Prozessor, werden in einem Instruktionswort mehrere einzelne Instruktionen übertragen, die parallel abgearbeitet werden. Die Zusammenstellung der einzelnen Instruktionen zu einem Instruktionswort erfolgt durch den Compiler. Er ist dafür verantwortlich, dass die einzelnen Instruktionen voneinander unabhängig ausgeführt werden können und nach Möglichkeit alle Ausführungseinheiten beschäftigt sind.

Parallele Schleifen

Schleifen sind ein wichtiger Bestandteil jeder Programmiersprache. Sie erlauben die wiederholte Ausführung von Anweisungen. Sind diese Anweisungen untereinander unabhängig, so können sie auch parallel von mehreren Prozessoren bearbeitet werden.

Beispielsweise kann die serielle Schleife

```
do i=1,n
  a(i) = b(i) + c(i)
enddo
```

bei der die einzelnen Anweisungen voneinander unabhängig zu bearbeiten sind, in der Programmiersprache High Performance Fortran (HPF) als forall-Schleife

```
!HPF$ forall (i = 1:n) a(i) = b(i) + c(i)
```

realisiert werden, die dann vom Compiler auf mehrere Prozessoren verteilt werden kann.

Datenparallelität

Sind auf unterschiedlichen Daten die gleichen Operationen durchzuführen, so kann man eine Parallelisierung durch gleichmäßiges Verteilen der Datenelemente auf die zur Verfügung stehenden Prozessoren erreichen.

Ein Beispiel aus der Programmiersprache Fortran90, bei der Datenparallelität genutzt wird, ist die Vektoranweisung

```
a(1:n) = b(1:n) + c(1:n)
```

bei der der Compiler die einzelnen Additionen und Zuweisungen von verschiedenen Prozessoren durchführen lassen kann. Ebenso kann die Matrixanweisung

```
A(1:n,1:n) = B(1:n,1:n) + C(1:n,1:n)
```

und Matrixmultiplikation mittels

```
A = matmul(B, C)
```

datenparallel durchgeführt werden.

Parallelität auf Prozessebene

Bei entsprechender Unterstützung durch das Betriebssystem in Form von so genanntem *Multitasking* können auf einem Rechner mehrere unterschiedliche oder gleiche Prozesse parallel ablaufen. Die Unterteilung der Berechnung in einzelne Teilprozesse ist durch den Benutzer festzulegen.

1.3.2 Implizite und explizite Parallelität

Programmiermodelle unterscheiden sich auch in der Art wie die Parallelität dargestellt wird.

Bei der impliziten Darstellung von Parallelität werden nur die notwendigen Berechnungen und ihre Abhängigkeiten untereinander, nicht aber die Berechnungsreihenfolge oder die Aufteilung der Berechnungen und Daten festgelegt. Die automatisch parallelisierenden Compiler folgen diesem Ansatz. Sie versuchen selbstständig ein vorgegebenes sequentielles Programm in ein paralleles Programm zu überführen. Dabei steht der Compiler vor der schwierigen Aufgabe, die Berechnungen und Daten so auf die Prozessoren zu verteilen, dass daraus eine möglichst gute Lastverteilung resultiert und gleichzeitig der notwendige Datenaustausch zwischen den Prozessoren niedrig bleibt.

Bei der expliziten Darstellung von Parallelität übernimmt der Programmierer die Aufteilung der Daten und Berechnungen auf die einzelnen Prozessoren. Ist Informationsaustausch zwischen den Prozessoren notwendig, so muss dieser explizit programmiert werden. Vertreter dieses Programmieransatzes sind beispielsweise die Message-Passing-Bibliotheken MPI (siehe auch Abschnitt A.3 auf Seite 212) und PVM.

1.3.3 Erzeugung von Prozessen

Die einzelnen Teilaufgaben von parallelen Programmen werden in parallel zueinander laufenden Prozessen realisiert. Programmiermodelle unterscheiden sich durch die Art wie diese Prozesse erzeugt werden.

Fork-Join-Konstrukt

Beim Fork-Join-Konstrukt handelt es sich um eine Möglichkeit Prozesse dynamische, d.h. während der Laufzeit zu erzeugen und auch wieder zu beenden. Ein bereits existierender Prozess erzeugt durch den Aufruf der Funktion `fork` einen so genannten Kindprozess, der eine Kopie des Erzeugerprozesses ist. Durch den gemeinsamen Aufruf der Funktion `join` werden die beiden Prozesse wieder zu einem zusammengeführt, der Kindprozess wird beendet, der Erzeugerprozess arbeitet weiter. Ruft zuerst der Erzeugerprozess die Funktion `join` auf, so wartet er solange bis auch der Kindprozess diese Funktion aufruft. Ruft zuerst der Kindprozess die Funktion `join` auf, so wird er sofort beendet. Da beide Prozesse, Kind- und Erzeugerprozess, die gleichen Daten und den gleichen Programmcode besitzen, müssen sie, wenn sie unterschiedliche Programmteile ausführen sollen, die Programmausführung von der Prozessnummer abhängig machen.

SPMD

Beim SPMD-Modell (Single Program Multiple Data) werden bei Programmstart eine feste Anzahl von Prozessen erzeugt. Diese Anzahl der Prozesse ist statisch, d.h. sie ändert sich über die gesamte Laufzeit des Programmes nicht. Alle Prozesse arbeiten das gleiche Programm ab. Sollen von den Prozessen unterschiedliche Programmteile ausgeführt werden, so muss die Ausführung dieser Programmteile von der Prozessnummer abhängig gemacht werden.

Eine Realisierung des SPMD-Modells findet man beispielsweise in der Programmbibliothek MPI. Zum Starten von zwei Prozessen, die beide das Programm `program` ausführen, verwendet man dort das Kommando

```
mpirun -np 2 program
```

(siehe auch Abschnitt A.3.4 auf Seite 219).

1.3.4 Datenaustausch

Ein wichtiger Aspekt bei der Parallelisierung ist, wie Daten zwischen den Prozessen ausgetauscht werden. Je nach zu Grunde liegender Hardware ergeben sich die beiden nachfolgenden Programmierkonzepte.

Rechner mit gemeinsamem Speicher

Bei Rechnern mit gemeinsamem Speicher (siehe auch Abschnitt 1.1.1 auf Seite 2) können Daten zwischen den Prozessen über den gemeinsamen Speicher ausgetauscht werden. Jeder Prozess besitzt einen Speicherbereich für private und einen Speicherbereich für gemeinsame Variablen. Der Datenaustausch zwischen Prozessen erfolgt

ausschließlich über die gemeinsamen Variablen. Private Variablen können von anderen Prozessen nicht zugegriffen werden. Will ein Prozess P_i einem Prozess P_j Daten übermitteln, so stellt er diese durch umkopieren in gemeinsame Variablen zur Verfügung. Prozess P_j kann sie dann zur weiteren Verarbeitung in seine privaten Variablen umkopieren.

Mit Hilfe von Synchronisationsoperationen muss dabei verhindert werden, dass Prozess P_j aus den gemeinsamen Variablen liest, bevor Prozess P_i die gewünschten Daten zur Verfügung gestellt hat. Ebenso muss das gleichzeitige Schreiben mehrerer Prozesse auf den gleichen gemeinsamen Variablen unterbunden werden, da sonst Inkonsistenzen auftreten können.

Eine solche notwendige Synchronisation zwischen den Prozessen P_i und P_j kann mit Hilfe eines so genannten binären *Semaphors* realisiert werden. Ein binäres Semaphor besteht aus einer Variable s, die nur die beiden Werte 1 ($\hat{=}$ ein Prozess schreibt auf die gemeinsamen Variablen) oder 0 ($\hat{=}$ kein Prozess schreibt auf die gemeinsamen Variablen) annehmen kann und den beiden unteilbaren Funktionen `signal(s)` und `wait(s)`. Die Funktion `signal(s)` setzt die Variable s auf den Wert 0. Die Funktion `wait(s)` wartet bis s den Wert 0 hat, setzt dann den Wert von s auf 1 und fährt mit der Befehlsausführung fort. Die Grundbelegung von s ist 0, also es schreibt kein anderer Prozess auf die gemeinsamen Variablen. Um das gleichzeitige Schreiben von mehreren Prozessen auf gemeinsame Variablen zu verhindern, muss nun jeder Prozess vor dem eigentlichen Schreiben die Funktion `wait(s)` und nach dem Schreiben die Funktion `signal(s)` ausführen.

Rechner mit verteiltem Speicher

Bei Rechnern mit verteiltem Speicher (siehe auch Abschnitt 1.1.1 auf Seite 2) kann der Datenaustausch nicht wie bei Rechnern mit gemeinsamem Speicher über einen allen Prozessen zugänglichen Speicherbereich erfolgen, sondern es sind explizite Kommunikationsanweisungen notwendig, um die Daten vom lokalen Speicherbereich eines Prozesses in den lokalen Speicherbereich des anderen Prozesses zu transferieren. Man spricht in diesem Zusammenhang auch von Nachrichtenaustausch (englisch: message passing). Man unterscheidet etwa die nachfolgenden Kommunikatonsarten.

Punkt-zu-Punkt-Kommunikation Bei dieser Kommunikationsart sind immer genau zwei Prozesse beteiligt. Der eine Prozess versendet eine Nachricht aus seinem lokalen Speicherbereich und der andere Prozess empfängt diese Nachricht und legt sie in seinem lokalen Speicherbereich ab (siehe dazu auch `MPI_Send` und `MPI_Recv` auf Seite 222).

Globale Kommunikation Bei der globalen Kommunikation sind immer alle Prozesse beteiligt. Einer dieser Prozesse übernimmt immer eine Sonderaufgabe, weil von

ihm die Datenverteilung ausgeht oder auf ihm die Dateneinsammlung stattfindet. Wir bezeichnen diesen Prozess im Folgenden als root-Prozess.

- **Rundruf (Broadcast)** Bei dieser Kommunikationsart versendet der root-Prozess eine Nachricht an alle Prozesse, so dass nach der Kommunikation alle Prozesse diese Nachricht in ihrem lokalen Speicher vorliegen haben (siehe dazu auch `MPI_Bcast` auf Seite 227).
- **Einsammeloperation (gather)** Bei dieser Kommunikationsart empfängt der root-Prozess von allen Prozessen eine Nachricht. Nach der Kommunikation liegt also im lokalen Speicher des root-Prozesses die Vereinigung aller von den Prozessen gesendeten Nachrichten vor (siehe dazu auch `MPI_Gather` auf Seite 228)
- **Verteileoperation (scatter)** Bei dieser Kommunikationsart werden Nachrichten vom root-Prozess gleichmäßig an alle Prozesse verteilt. Nach der Kommunikation hat jeder Prozess einen Teil der Nachrichten in seinem lokalen Speicher vorliegen (siehe dazu auch `MPI_Scatter` auf Seite 228).
- **Reduktionsoperation** Bei dieser Kommunikationsart wird von allen Prozessen gemeinsam eine Rechenoperation, etwa eine Addition, durchgeführt und das Ergebnis dieser Berechnung im lokalen Speicher des root-Prozesse abgelegt (siehe dazu auch `MPI_Reduce` auf Seite 229).