1

Von Megahertz zu Gigaflops

If you were plowing a field, what would you rather use? Two strong oxen or 1024 chickens?

Seymour Cray (1925–1996)

To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox.

W. Gropp, E. Lusk, A. Skjellum [57]

1.1 Ochsen oder Hühner?

Cluster Computing ist alter Wein in neuen Schläuchen. Der „alte Wein“ heißt *paralleles Rechnen auf Systemen mit verteiltem Speicher* und war jahrzehntelang ein eher randständiges Gebiet der Computerwissenschaft. Seit den 90er Jahren des letzten Jahrhunderts rückt das parallele Rechnen jedoch mehr und mehr in den Blickpunkt von Anwendern und Computerwissenschaftlern. Daran sind einerseits die „neuen Schläuche“ in Form preiswerter Hardware (PCs aus der Massenproduktion) und freier Software schuld, andererseits erfordern viele Anwendungen in Industrie und Forschung inzwischen Rechenleistungen, die mit keiner anderen Technologie erreicht werden können.

Das klassische Beispiel für eine Applikation, die den Einsatz von Parallelrechnern geradezu erzwingt, ist die *Wettervorhersage*, d. h. die numerische Simulation der Atmosphäre[1]. Dazu wird die Lufthülle diskretisiert und durch ein dreidimensionales Gitter repräsentiert. An jedem Punkt des Gitters werden atmosphärische Parameter wie Temperatur, Windgeschwindigkeit, Luftdruck usw. berechnet. Auch die Zeit wird diskretisiert: in einem Zeitschritt werden die Parameter an jedem Gitterpunkt in Abhängigkeit der Parameter der Nachbarpunkte neu berechnet. Für die Wetterentwicklung in Deutschland ist das Azoren-Hoch genauso wichtig wie das Kontinentalklima in Russland. Darum muss eine verlässliche Wettervorhersage ein großes Gebiet umfassen, am besten natürlich die ganze Erde. Andererseits sollte die Maschenweite des verwendeten Gitters möglichst klein sein, um wesentliche atmosphärischer Prozesse auflösen zu können. Beide Forderungen zusammen führen zu einer großen Anzahl von Gitterpunkten. Ein Gitter mit einer Maschenweite von 1 km,

[1] Das folgende Beispiel wurde leicht modifiziert aus [130] übernommen.

das die gesamte Erdatmosphäre bis in eine Höhe von 20 km darstellt, besteht aus ungefähr 10^{10} Gitterpunkten. Die zeitliche Auflösung muss der räumlichen angepasst sein und beträgt für eine Maschenweite von 1 km ungefähr zehn Sekunden. Für eine dreitägige Wettervorhersage bedeutet das, dass die Parameter an jedem der 10^{10} Gitterpunkte rund 26 000 mal aktualisiert werden müssen. Nehmen wir an, dass die Aktualisierung eines Gitterpunktes 100 Rechenoperationen erfordert, so ergibt das $2{,}6 \times 10^{16}$ Operationen für die gesamte Simulation. Ein aktueller PC vom freundlichen Computerhändler nebenan schafft ungefähr 10^9 (eine Milliarde!) Rechenoperationen pro Sekunde. Damit würde unsere Wettersimulation 300 Tage benötigen, wäre also um einen Faktor 100 langsamer als das wirkliche Wetter draußen vor der Tür.

Mit einem Rechner, der 10^{12} (eine Billion) Rechenoperationen pro Sekunde schafft, wäre unsere globale Drei-Tage-Wettervorhersage in weniger als acht Stunden machbar. Ein solcher Rechner hätte allerdings ein grundsätzliches Problem. Zum Rechnen braucht er Daten, und die können bei diesem Tempo nicht mehr schnell genug herangeschafft werden. Nehmen wir an, dass für jede Rechenoperation mindestens ein Datenelement aus dem Speicher zur CPU transportiert werden muss. In 10^{-12} Sekunden kann selbst das Licht nur 0,3 mm zurücklegen. Die Laufzeiten werden minimiert, wenn man den Speicher auf einer Kreisfläche mit Radius 0,3 mm anordnet, in deren Mitte man den Prozessor setzt. Auf dieser Kreisfläche müssen wir die Daten unserer Simulation unterbringen, pro Gitterpunkt also ca. 20 Zahlen für Temperatur, Windgeschwindigkeit etc., jede Zahl hat 32 Bit, macht für alle 10^{10} Gitterpunkte $6{,}4 \times 10^{12}$ Bit. Damit bleibt für jedes Bit im Speicher eine kreisförmige Fläche mit Radius 10^{-10} m. Das ist der Durchmesser eines Atoms.

Eine Speicherdichte von einem Bit pro Atom ist jenseits aller technischen Möglichkeiten. Trotzdem sind Rechner, die mehr als eine Billiarde Rechenoperationen pro Sekunde leisten, heute schon Realität. Diese Rechner sind allerdings massiv parallele Systeme, gebaut nach dem Prinzip: wenn die Daten nicht schnell genug zu einem Prozessor kommen können, müssen eben viele Prozessoren zu den Daten gehen. Die Grundidee ist einfach. Statt eines Prozessors mit 10^{12} Operationen pro Sekunde nehmen wir 1000 langsamere Prozessoren mit 10^9 Operationen pro Sekunde. Jeder Prozessor kümmert sich um einen kleinen Ausschnitt der Atmosphäre, genauer um ein Tausendstel oder 10^7 Gitterpunkte. Bedingt durch die niedrigere Geschwindigkeit der Prozessoren können deren Daten jetzt bis zu 300 mm vom Prozessor entfernt sein. Dank Arbeitsteilung müssen auf der größeren Speicherfläche auch weniger Daten untergebracht werden, nämlich $6{,}4 \times 10^9$ Bit oder 800 MByte. Das gelingt aber selbst mit dem preiswerten PC vom freundlichen Computerhändler um die Ecke.

Derartige Abschätzungen sind natürlich grob vereinfachend und können aus vielen Richtungen kritisiert werden. So könnte man das Speicherproblem dadurch entschärfen, dass man die Daten dreidimensional statt nur in der Fläche anordnet. Und neue Technologien wie die Quanten-Computer erlauben eines Tages vielleicht auch Speicherdichten von einem Bit pro Atom. In solchen Überlegungen steckt viel Spekulation. Was heute dagegen sicher ist und in näherer Zukunft richtig bleiben wird, ist dies: Rechenleistungen von mehr als ein paar Billionen Operationen pro Sekunde können nur von parallelen Systemen erbracht werden.

Skeptiker wie Seymour Cray hatten bei ihrer Kritik (Zitat auf Seite 3) immer das Problem vor Augen, massiv parallele Abläufe zu koordinieren. In der Tat sind Betrieb und Nutzung eines massiv parallelen Rechners mit einigem Mehraufwand verbunden, ein Mehraufwand, der sich zur Glanzzeit der Cray-Supercomputer nicht lohnte. Zu groß war damals der Unterschied zwischen den „Hühnern“ (billigen Prozessoren für die Parallelrechner) und den „Ochsen“ (hochgezüchteten Spezialprozessoren für die Supercomputer). Zwei Ochsen waren tatsächlich so stark wie 1000 Hühner aber einfacher zu dirigieren. Heute ist der Leistungsunterschied zwischen den preiswerten Prozessoren aus der Massenfertigung und teuren Spezialprozessoren dagegen so klein, dass die meisten Supercomputer mit denselben Prozessoren arbeiten wie der PC auf Ihrem Schreibtisch. Aus den Hühnern von damals sind Ochsen geworden.

1.2 Rechenleistung im Takt

Die Leistung eines Rechners lässt sich nicht so einfach angeben wie die Leistung einer Pumpe, was natürlich daran liegt, dass ein Rechner mehr kann als eine Pumpe. Im technisch-wissenschaftlichen Bereich ist es die *Fließkommaleistung*, die noch am ehesten einer Pumpleistung entspricht. Darunter versteht man die Anzahl der Fließkommaoperationen, die ein Rechner pro Sekunde ausführen kann. Angegeben wird diese Leistung in Fließkommaoperationen pro Sekunde (*Flops*), mit den üblichen Vorfaktoren M(ega) (10^6), G(iga) (10^9) und T(era) (10^{12}). Fließkommazahlen sind die Computer-Version der reellen Zahlen und entsprechen in der Programmiersprache C den Datentypen `float` und `double`. Es ist nicht eindeutig definiert, welche Fließkommaoperationen bei einer Leistungsangabe in MFlops berücksichtigt werden. Möglich sind neben den elementaren arithmetischen Operationen (Addition, Subtraktion, Multiplikation und Division) auch Zuweisungen, Vergleiche und Typ-Konversionen. Komplexere Funktionen wie die Berechnung der Quadratwurzel oder des Logarithmus zählen meist nicht dazu. Bei Prozessoren, die Fließkommaarithmetik hardwaremäßig unterstützen, gelten alle Operationen, die im Befehlssatz der FPU (*floating point unit*) enthalten sind, als Fließkommaoperationen.

Wenn eine Fließkommaoperation vom Prozessor mit einem einzigen Befehl erledigt werden kann, sollte die Fließkommaleistung dann nicht der *Taktfrequenz* des Prozessors entsprechen? Also „GFlops = GHz“? So einfach ist es leider nicht. Betrachten wir dazu ein einfaches Beispiel, die Summierung der Elemente eines Vektors:

```
#define N 1000000
unsigned long l;
double a[N], sum=0.0;
...
for (l=0; l<N; ++l)
  sum=sum+a[l];
```

Der Compiler macht aus diesem Code-Segment eine Liste von Befehlen für den Prozessor. Diese Befehlsliste befindet sich genau wie die Daten im Hauptspeicher

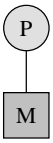

Abb. 1.1. Sequentieller Rechner mit Prozessor P und Speicher M.

des Rechners (Abb. 1.1). Der Prozessor selbst verfügt nur über wenige Speicherzellen, die sog. *Register*, und Rechenoperationen kann er nur auf diesen Registern ausführen. Zum Rechnen muss der Prozessor Befehl und Daten aus dem Speicher holen und in seinen Registern unterbringen. Diese vereinfachte Darstellung heißt das *von-Neumann-Modell* eines Rechners. Bei einer einzigen Iteration der Schleife oben durchläuft der Prozessor dann in etwa diese Schritte:

- Hole nächsten Befehl aus dem Speicher in das Befehlsregister.
- Interpretiere diesen Befehl.
- Lade erstes Argument (`sum`) aus dem Speicher in ein Register.
- Lade zweites Argument (`a[l]`) aus dem Speicher in ein anderes Register.
- Führe den Befehl aus und schreibe Ergebnis in ein drittes Register.
- Schreibe das Ergebnis (`sum`) zurück in den Speicher.

Sie sehen: selbst die Addition, die der Prozessor mit einem einzigen Befehl ausführen kann, benötigt mehrere Taktzyklen. Ist dann die Fließkommaleistung ein konstanter Bruchteil der Taktfrequenz? Auch falsch! Moderne Prozessoren sind wahre Wunderwerke der Technik. Sie haben für viele Aufgaben eigene Abteilungen, die unabhängig voneinander arbeiten können. So kann eine Abteilung einen gerade geladenen Befehl interpretieren, während eine andere Abteilung zur selben Zeit bereits den nächsten Befehl lädt. Das funktioniert genau wie die Produktion am Fließband, wo an jeder Station eine Teilarbeit erledigt wird. Zu jedem Zeitpunkt befinden sich viele unfertige Produkte auf dem Band, aber mit jedem Weiterrücken des Bandes fällt am Ende ein fertiges Produkt heraus. Bei den Prozessoren heißt das Fließband *Pipeline*, und wenn die Pipeline immer schön voll ist, wird bei jedem Taktzyklus ein kompletter Befehl wie `sum=sum+a[i]` fertig. In diesem Fall ist tatsächlich „MFlops = MHz“. Die *superskalaren* Prozessoren haben mehrere Pipelines, so dass in einem einzigen Taktzyklus jede dieser Pipelines ein fertiges Fließkommaresultat liefert. *Vektorprozessoren* verfügen über spezielle Instruktionen, um *eine* Fließkommaoperationen gleichzeitig auf *mehreren* ihrer Register durchführen zu können. Bei superskalaren oder vektoriellen Prozessoren wäre die Fließkommaleistung dann sogar ein Vielfaches der Taktfrequenz. Die MFlops-Werte in den Prospekten der Hardware-Hersteller stammen oft aus solchen Überlegungen. Diese sog. *peak performance* (siehe Tabelle 1.1) kann allerdings nur unter optimalen Bedingungen erreicht werden. In praktischen Anwendungen liegt die Fließkommaleistung zum Teil deutlich darunter.

1.3 Von-Neumann-Flaschenhals

Haupthindernis auf dem Weg zur peak-performance ist der Hauptspeicher. Die üblicherweise verwendete Speichertechnologie (DRAM [171], *dynamic random access memory*) erlaubt Zugriffszeiten von ungefähr 10 ns. Das entspricht einer Frequenz von 100 MHz, also nur einem Bruchteil der Taktfrequenz moderner Prozessoren. In Anlehnung an das von-Neumann-Modell heißt diese Diskrepanz auch *von-Neumann-Flaschenhals*. Eine treffende Bezeichnung, denn wenn der Prozessor jeden Befehl und jedes Datenelement aus dem Hauptspeicher holen müsste, würde sich sein Takt effektiv auf den Speichertakt reduzieren.

Um die Auswirkungen des von-Neumann-Flaschenhalses zu mindern, schaltet man zwischen Prozessor und Hauptspeicher einen Zwischenspeicher, den sog. *Cache*. Der Cache enthält eine Kopie von Teilen des Hauptspeichers. Er besteht im Gegensatz zum Hauptspeicher aus SRAM (*static random access memory*) und ermöglicht Zugriffsgeschwindigkeiten, die der Taktfrequenz der Prozessoren nahe kommen. Zugriffe auf Daten im Cache verlangsamen die Rechnung also nicht. Dadurch, dass SRAM sehr viel teurer als DRAM ist und der Cache heute meist im Chip des Prozessors integriert wird, ist die Größe des Caches beschränkt. Üblich sind Cachegrößen von weniger als einem MByte, bei Spitzen-CPUs wie Intels Itanium 2 auch schon mal bis zu 9 MByte. Der Anteil der *cache misses*, also der Zugriffe auf Befehle und Daten, die nicht im Cache liegen, wird zum bestimmenden Faktor für die effektive Fließkommaleistung.

Vom Effekt des Caches auf die Fließkommaleistung können Sie sich in einem kleinen Experiment selbst überzeugen. Auf der begleitenden Webseite [7] finden Sie das Programm `cachesize.c`, welches nichts weiter macht, als die Ausführungszeit der Schleife

```
for (l=0; l<N; ++l)
  sum=sum+a[l];
```

für verschiedene Werte von N zu messen, in MFlops umzurechnen und das Ergebnis auszugeben. Eine übliche Cache-Strategie ist es, stets die zuletzt benötigten Speicherelemente im Cache vorzuhalten. Wenn N also klein genug ist, sind alle Vektorelemente a_l spätestens nach dem ersten Durchlauf komplett im Cache. Jeder erneute Schleifendurchlauf sollte dann weitgehend ohne die teuren Speicherzugriffe auskommen. Ist N dagegen so groß, dass die Vektorelemente nicht alle in den Cache passen, sollten die nun erforderlichen Zugriffe auf den Hauptspeicher die Rechenleistung vermindern.

Abbildung 1.2 zeigt diesen Effekt deutlich. Gemessen wurde mit `cachesize.c` auf zwei Rechnern: einem PC mit AMD Athlon Prozessor (1,4 GHz) und einer Sun Workstation mit UltraSPARC II Prozessor (300 MHz). Der PC hat einen Cache von 256 KByte, bei der Sun Workstation ist der Cache 2 MByte groß. In beiden Fällen nimmt die effektive Rechenleistung genau bei diesen Werten drastisch ab, beim PC um den Faktor drei, bei der Workstation sogar um den Faktor vier. Die Abbildung zeigt darüberhinaus, dass auch der Compiler einen erheblichen Einfluss auf die Ausnutzung des Caches haben kann. Bei den oberen beiden Kurven wurde

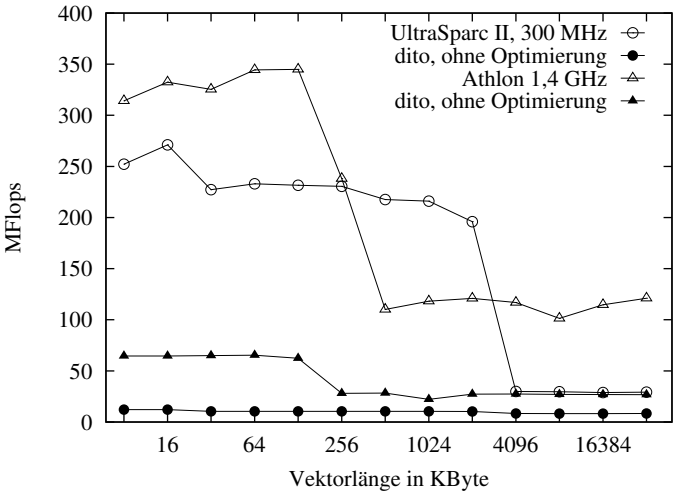

Abb. 1.2. Rechenleistung bei der Aufsummierung von Vektorelementen.

`cachesize.c` mit allerlei Optimierungs-Optionen des Compilers übersetzt, bei den unteren Kurven wurde dagegen völlig auf Compiler-Optimierungen verzichtet.

1.4 Benchmarks

In welchem Umfang sich Cachegröße und die Fähigkeiten des Compilers auf die Fließkommaleistung auswirken, hängt stark vom betrachteten Algorithmus und dessen Implementierung ab. Um trotzdem aussagefähige Leistungsmaße zu erhalten, werden gewöhnlich sog. *Benchmark-Tests* durchgeführt. Das sind mehr oder weniger standardisierte Programme, die unter kontrollierten Bedingungen übersetzt und ausgeführt werden. Deren Ausführungszeiten gelten dann als Maß für die Rechengeschwindigkeit. Für die Aussagefähigkeit dieser Methode ist es wichtig, dass die Benchmark-Programme in ihrer Struktur möglichst den „typischen“ Anwendungen entsprechen. Das kann man auf zweierlei Arten erreichen: entweder mittelt man über einen ganzen Satz tatsächlicher Anwendungen (Benchmark-Suite), oder man misst zentrale Bausteine, die in sehr vielen Anwendungen vorkommen (Kernel-Benchmark).

Die erste Strategie wird z. B. von der Standard Performance Evaluation Corporation (SPEC) verfolgt, einem Zusammenschluss diverser Hardware-Hersteller [154]. Ihr aktueller Benchmark SPEC CFP2000 zur Ermittlung der Fließkommaleistung besteht aus 14 Programmen (in FORTRAN 77, FORTRAN 90 und C), die realistischen Anwendungen aus Bereichen wie Zahlentheorie, Finite Elemente, Crash Simulation, numerische Strömungsmechanik, Bildverarbeitung usw. entsprechen. Der SPEC-Benchmark mittelt die Laufzeit über alle 14 Anwendungen. Das Resultat ist

allerdings kein MFlops-Wert, sondern die Leistung relativ zu einer Referenzmaschine.

Der berühmteste Vertreter der zweiten Kategorie ist der Linpack-Benchmark [95]. Linpack ist eigentlich eine Sammlung von FORTRAN-Unterprogrammen zur Lösung von linearen Gleichungssystemen. Linpack stützt sich dabei auf BLAS (*Basic Linear Algebra Subroutines*), einer Sammlung von Unterprogrammen für die elementaren arithmetischen Operationen der linearen Algebra, also das Matrix-Vektor-Produkt, das Skalarprodukt usw. [90]. Der Linpack-Benchmark misst im Wesentlichen die Performance dieser Routinen. Das ist naheliegend, denn im technisch-wissenschaftlichen Umfeld hantieren sehr viele Anwendungen mit Matrizen und Vektoren und verbringen einen großen Teil der Laufzeit mit BLAS-artigen Berechnungen.

Im Linpack-Benchmark werden die BLAS-Routinen verwendet, um ein lineares Gleichungssystem mit dem Gauß'schen Eliminiationsverfahren [136] zu lösen. Die Zahl der dazu erforderlichen Fließkommaoperationen ist genau bekannt, d. h., man kann aus der Laufzeit direkt die Leistung in MFlops ausrechnen. Den Linpack-Benchmark gibt es in drei Versionen, die sich in ihren Regeln unterscheiden:

- Beim 100×100 Linpack hat das Gleichungssystem 100 Unbekannte und der Quellcode darf nicht verändert werden. Nur automatische Optimierungen durch den Compiler sind erlaubt.
- Beim 1000×1000 Linpack hat das Gleichungssystem 1000 Unbekannte und der Quellcode darf beliebig verändert werden, allerdings muss sowohl der zugrundeliegende Algorithmus (Gauß'sches Eliminationsverfahren) als auch die Gesamtzahl der Fließkommaoperationen bewahrt bleiben.
- Beim *High-Performance Linpack* gelten die Regeln des 1000×1000 Linpacks, nur darf jetzt auch die Zahl der Unbekannten gewählt werden.

Tabelle 1.1. Resultate des 100×100 Linpack mit einfacher Genauigkeit.

CPU	Taktfrequenz	Peakperf.	Linpack
UltraSPARC I	140 MHz	280 MFlops	35 MFlops[1]
UltraSPARC II	200 MHz	400 MFlops	50 MFlops[1]
UltraSPARC II	400 MHz	800 MFlops	94 MFlops[1]
Pentium II	333 MHz	333 MFlops	85 MFlops[2]
Pentium III	800 MHz	800 MFlops	300 MFlops[2]
AMD Athlon	1400 MHz	2800 MFlops	635 MFlops[2]
Pentium M	1600 MHz	3200 MFlops	1100 MFlops[3]
Pentium 4	2400 MHz	4800 MFlops	820 MFlops[2]
			1320 MFlops[4]
Xeon	3060 MHz	6120 MFlops	1800 MFlops[4]

[1] Sun Workshop Compiler 5.0
[2] GCC 2.95.4 mit Option `-O3`
[3] ICC 8.1 mit Optionen `-O3 -xB`
[4] ICC 8.1 mit Optionen `-O3 -xN`

Der Linpack-Benchmark hat darüberhinaus den Vorteil, dass er frei verfügbar ist. Auf der begleitenden Webseite finden Sie die Datei `linpack.c`, eine C-Version des 100×100-Linpack, bei dem die Größe der Matrix allerdings per Kommandozeile modifiziert werden kann.

Tabelle 1.1 zeigt die Resultate dieses einfachen Linpack-Benchmarks für einige Prozessoren. Die MFlops-Werte steigen wie erwartet mit der Taktfrequenz an. Die genauen Zahlen hängen von einer ganzen Reihe weiterer Hardware-Parameter (Größe des Cache, Taktfrequenz des Speichers), den Fähigkeiten des Compilers und der Größe des verwendeten Gleichungssystems ab. Sie sollten sich nicht wundern, wenn Sie bei eigenen Messungen oder in der Literatur abweichende Werte finden. Für moderne Prozessoren liefert die Gleichung „GFlops = GHz“ zumindest die richtige Größenordnung der Fließkommaleistung. Das zeigt übrigens auch Jack Dongarras Liste [31] der Linpack-Resultate für mehr als 1000 Ein-Prozessor-Maschinen, vom kleinen Palm Pilot III (1,69 KFlops) bis hin zum Xeon mit 3,2 GHz (5,4 GFlops).

1.5 Amdahls Gesetz

Kann man mit 1000 Prozessoren, die jeweils ein 1 GFlops leisten, in den Bereich des Teraflop-Computings vorstoßen? Im Prinzip ja, aber ...

Massive Parallelität hat prinzipielle Beschränkungen, von den technischen Problemen ganz zu schweigen (die besprechen wir später). Die erste Beschränkung: nicht alle Teile eines Programms lassen sich parallelisieren. Sei σ die Zeit, die ein Programm mit nicht parallelisierbaren Anweisungen verbringt, und sei π die Ausführungszeit der parallelisierbaren Anweisungen. Dann ist die Laufzeit auf einem System mit p Prozessoren gegeben durch

$$T(p) = \sigma + \frac{\pi}{p}, \tag{1.1}$$

wobei wir idealerweise davon ausgegangen sind sind, dass sich der parallelisierbare Anteil gleichmäßig auf gleich schnelle Prozessoren verteilen lässt. Der *Speedup*

$$S(p) = \frac{T(1)}{T(p)} \tag{1.2}$$

gibt an, um wieviel schneller das Programm durch die Verwendung von p Prozessoren im Vergleich zur sequentiellen Ausführung wird. Im Idealfall erreicht man einen linearen Speedup, $S(p) = p$. Mit einem relativen Anteil

$$f = \frac{\sigma}{\sigma + \pi} \tag{1.3}$$

nicht parallelisierbarer Anweidungen wird daraus jedoch

$$S(p) = \frac{1}{f + (1-f)/p} \leq \frac{1}{f}. \tag{1.4}$$

Durch massive Parallelität können wir die Ausführungszeit des parallelisierbaren Anteils beliebig klein machen, der sequentielle Anteil bleibt davon aber unberührt. Bei einem Anteil von nur 1 % nicht parallelisierbarer Anweisungen ist der maximale Speedup gleich 100, egal wieviele Prozessoren wir einsetzen. Dieses Argument wurde 1967 von Gene Amdahl publiziert [2], und (1.4) heißt seitdem *Amdahls Gesetz*. Herr Amdahl und nach ihm viele andere zogen daraus den Schluss, dass sich massive Parallelität nicht lohne.

Der sequentielle Anteil ist noch nicht einmal die einzige Beschränkung paralleler Effizienz. In (1.1) sind wir davon ausgegangen, dass sich der parallelisierbare Anteil eines sequentiellen Programms völlig verlustlos auf p Prozessoren verteilen lässt. In der Regel müssen die Prozessoren ihre Arbeit aber irgendwie koordinieren. Das geschieht durch den Austausch von Daten, was erstens Zeit kostet und zweitens dazu führt, dass Prozessoren u. U. nicht weiterarbeiten können, weil sie auf das Zwischenergebnis eines anderen Prozessors warten. Eine weitere Quelle von Reibungsverlusten ist eine unausgewogene *Lastverteilung*: einige Prozessoren erledigen ihre Teilaufgaben u. U. schneller als andere und warten dann untätig bis zum Ende der gesamten Rechnung. Auch die Verwaltung der Teilaufgaben selbst kann Zeit kosten, die mit der eigentlichen Problembearbeitung nichts zu tun hat. Wir subsumieren alle diese Effekte in einer zusätzlichen, durch die Parallelisierung bedingten Laufzeit $T_c(p)$. Aus (1.1) wird so

$$T(p) = \sigma + \frac{\pi}{p} + T_c(p)\,. \tag{1.5}$$

Im Allgemeinen wird $T_c(p)$ monoton mit der Zahl der Prozessoren ansteigen, was jeder, der Erfahrung mit Teamarbeit hat, sofort einsieht. Nehmen wir der Einfachheit halber an, dass die Prozessoren immer gleichmäßig ausgelastet sind und $T_c(p)$ nur von der Zeit für die Kommunikation zwischen den Prozessoren bestimmt wird. Wir werden später noch sehen, dass das in vielen Fällen eine recht gute Annahme ist. In einem typischen Szenario paralleler Anwendungen (Abschnitt 9.1.3) kontrolliert ein Prozess (der Master) alle anderen (die Worker). Kommunikation findet nur zwischen Master und Worker statt, niemals zwischen Workern. Dann sollte $T_c(p)$ eine lineare Funktion von p sein:

$$T_c(p) = T_c(2)(p-1)\,. \tag{1.6}$$

Ein Prozessor allein führt keine Selbstgespräche, also ist $T_c(2)$ die minimale Zeit, die Kommunikation überhaupt kosten kann. Aus unserem Speedup wird jetzt

$$S(p) = \frac{1}{f - r + (1-f)/p + rp}\,, \tag{1.7}$$

wobei

$$r = \frac{T_c(2)}{T(1)} \tag{1.8}$$

das Verhältnis zwischen der minimalen Kommunikationszeit und der sequentiellen Rechenzeit angibt.

Durch die Berücksichtigung der Kommunikationskosten fällt der Speedup für große Werte von p jetzt sogar ab (Abb. 1.3), d. h., massive Parallelität kann kontraproduktiv sein. Der Speedup erreicht sein Maximum

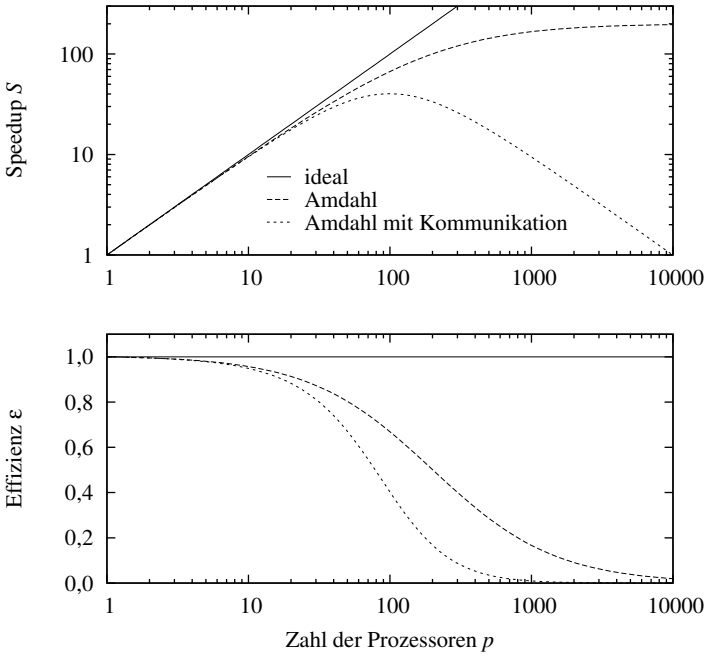

Abb. 1.3. Speedup und Effizienz paralleler Programme gemäß Amdahls Gesetz (mit $f = 0{,}005$) und bei zusätzlicher Berücksichtigung eines linearen Kommunikationsaufwandes (mit $r = 0{,}0001$).

$$S(p^*) = \frac{1}{f - r + 2\sqrt{(1-f)r}} \tag{1.9}$$

bei

$$p^* = \sqrt{\frac{1-f}{r}}\,. \tag{1.10}$$

Nur für $p \ll p^*$ ist $S(p) = p$, d. h. nur in diesem Bereich zahlt sich Parallelität aus.

Ein vom Speedup $S(p)$ abgeleitetes Maß für die Güte einer parallelen Anwendung ist die Effizienz

$$\varepsilon(p) = \frac{T(1)}{pT(p)} = \frac{S(p)}{p}\,. \tag{1.11}$$

Sie gibt den Anteil der Zeit an, die jeder Prozessor mit *nützlicher* Arbeit verbringt. Ein Wert von $\varepsilon(100) = 0{,}40$ wie im Beispiel der Abb. 1.3 bedeutet, dass jeder der 100 Prozesse 60 % der Zeit damit verbringt, auf Daten von anderen Prozesse zu warten oder Daten zu verschicken. Ideal ist natürlich eine Effizienz $\varepsilon(p) = 1$, aber sequentieller Anteil und Kommunikationsverluste sorgen für einen Abfall für größere Werte von p (siehe Abb. 1.3 unten).

Listing 1.1. Die Funktion `pintegral` in `pintegral.c` berechnet Partialsummen für die parallele numerische Integration.

```
/* berechnet Integral über f von a bis b mit Mittelpunktsregel an n
   Stützstellen. Liefert die k-te Partialsumme (0<=k<tasks) zurück. */
double pintegral(double (*f)(double), double a, double b,
                 long n, int k, int tasks) {
  double result=0.0;
  double h=(b-a)/n;
  long l;
  a=a+h/2;
  for (l=k; l<n; l+=tasks)
    result+=f(a+l*h);
  return h*result;
}
```

1.6 Granularität

Stellen Sie sich einen Maler vor, der für das Tapezieren eines Zimmers eine Stunde benötigt. Dass zwei Maler dieses Zimmer in einer halben Stunde schaffen, ist nicht unplausibel. Kein Mensch käme jedoch auf die Idee, dass 60 Maler ein Zimmer in einer Minute tapezieren könnten. Sie würden sich derart im Weg stehen oder um die wenigen Ressourcen (Tapeziertisch, Leiter) streiten, dass sie vermutlich sogar länger als eine Stunde bräuchten. Diese Alltagsversion des Amdahl'schen Gesetzes verdeutlicht den Schwachpunkt der zugrundeliegenden Prämisse, mit immer mehr Prozessoren ein gegebenes Problem immer schneller lösen zu wollen. Mit 60 Malern lässt sich ein einzelnes Zimmer nicht 60 mal so schnell tapezieren, wohl aber ein Hotel mit 60 Zimmern. Dazu muss man nur die Maler gleichmäßig auf die Zimmer des Hotels verteilen. Auf Parallelrechner übertragen heißt das, dass mit zunehmender Zahl von Prozessoren auch die Problemgröße wachsen sollte, um ein effizientes Arbeiten zu gewährleisten. Man sagt, dass Problem muss mit der Zahl der Prozessoren *skalieren*. Diese Erkenntnis erscheint banal, und doch setzte sie sich in der Informatik erst 1988 durch, 21 Jahre nach Amdahls Gesetz (Notiz 1.8)[2].

Betrachten wir als Beispiel die numerische Integration. Dabei wird das Integral einer Funktion f dadurch berechnet, dass man f an n Stützstellen x_i im Integrationsintervall $[a,b]$ auswertet. In der primitivsten Version (Mittelpunktsregel) wählt man äquidistante Stützstellen im Abstand $h = (b-a)/n$ und approximiert das Integral durch

$$\int_a^b f(x)\,\mathrm{d}x \approx h \cdot \sum_{i=0}^{n-1} f(x_i) \qquad \text{mit} \qquad x_i = a + \left(i + \frac{1}{2}\right) h\,. \tag{1.12}$$

Die Approximation ist natürlich umso genauer, je größer n ist.

Mit der Funktion `pintegral` (Listing 1.1) können wir die Integration parallel auf p Prozessoren durchführen lassen (Abb. 1.4). Dazu nummerieren wir die Prozessoren von 0 bis $p-1$ und lassen den k-ten Prozessor den Wert

[2] Am Ende eines jeden Kapitels finden Sie einige Notizen und Hinweise, die uns erwähnenswert erschienen, die aber im laufenden Text leider keinen Platz fanden.

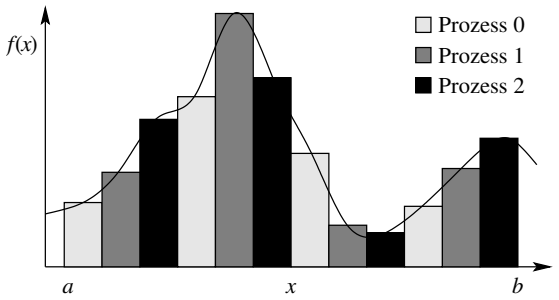

Abb. 1.4. Parallele numerische Integration mit drei Prozessoren.

```
mysum=pintegral(f, a, b, n, k, P);
```

ausrechnen. Danach müssen die Teilsummen `mysum` der einzelnen Prozessoren nur noch addiert werden. Wir werden in Kapitel 3 sehen, wie man das praktisch macht.

Der parallelisierbare Anteil diese Verfahrens (die Berechnung der Teilsummen) wächst linear mit n, während der sequentielle Anteil (Berechnung von h etc.) unabhängig ist von n. Der relative Anteil der nicht parallelisierbaren Anweisungen ist daher proportional zu $1/n$, nimmt also mit wachsender Problemgröße ab. Der Kommunikationsaufwand ist auch unabhängig von der Problemgröße, denn am Ende hat jeder Prozessor nur seine Teilsumme, d. h. eine einzige Zahl zu versenden. Also nimmt auch der relative Parallelisierungsverlust mit zunehmender Problemgröße ab. Solange also $p \ll n$ ist, können wir einen idealen Speedup $S(p) \approx p$ erwarten.

Parallele Algorithmen sollten so organisiert sein, dass der *relative* Anteil der inhärent sequentiellen Anweisungen und der durch die Parallelisierung erforderlichen Synchronisations- und Kommunikationszeiten mit wachsender Problemgröße abnimmt. Die inhärent sequentiellen Teile eines Algorithmus erfüllen diese Bedingung in den meisten Fällen automatisch. Synchronisation und Kommunikation skalierbar zu gestalten, ist in der Regel etwas schwieriger.

Betrachten wir unter diesem Gesichtspunkt das Eingangsbeispiel, die Klimasimulation. In der parallelen Variante hatten wir jedem Prozessor eine Teilmenge der Gitterpunkte zugeordnet. Um die erforderliche Kommunikation zwischen den Prozessoren möglichst gering zu halten, sollten die Gitterpunkte eines Prozessors alle benachbart sein. Dann kann jeder Prozessor fast alle Gitterpunkte seines Bereiches aktualisieren, ohne auf die Daten anderer Prozessoren angewiesen zu sein. Nur für die Neuberechnung der Punkte, die am Rand seines Gebietes liegen, muss ein Prozessor Daten mit anderen austauschen. Der Kommunikationsaufwand wächst daher wie die Oberfläche aneinander angrenzender Teilvolumina (Abb. 1.5). Bei einem d-dimensionalen Gitter mit Maschenweite Δ also wie $\Delta^{-(d-1)}$. Der Rechenaufwand wächst dagegen wie Δ^{-d}, d. h., der relative Anteil der Kommunikation skaliert wie Δ. Je feinmaschiger das Diskretisierungsgitter, umso weniger fällt der Datenaustausch der Prozessoren ins Gewicht. Dieser angenehme *Oberflächen-versus-Volumen-Effekt*

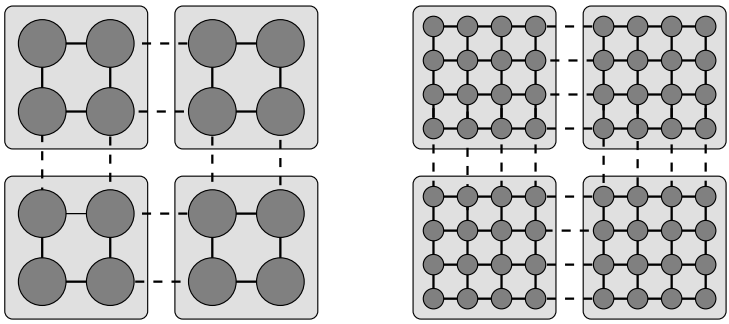

Abb. 1.5. Beim Übergang von einem groben (links) zu einem feinen Gitter (rechts) steigt die Rechenlast (Faktor vier) schneller als der Kommunikationsaufwand (Faktor zwei). Probleme, die sich auf Gittern mit lokalen Datenflüssen abbilden lassen, eignen sich deshalb hervorragend für Parallelrechner.

tritt bei allen Problemen auf, die sich auf Gitter mit lokalen Datenflüssen abbilden lassen, und das sind eine ganze Menge, sehr wichtiger Probleme (siehe Abschnitt 9.2.1).

Wann ist das Gitter groß genug für eine effiziente Behandlung auf einem Parallelrechner mit p Prozessoren? Offensichtlich dann, wenn ein Prozessor lange (nützlich) rechnen kann, bevor er auf einen Datenaustausch angewiesen ist. Man nennt die Anzahl der Rechenschritte, die von einem Prozessor zwischen zwei Kommunikationsvorgängen ausgeführt werden kann, *Granularität* oder *Körnigkeit*. Wünschenswert ist eine grobe Körnigkeit. Feingranulare Programme verbringen zu viel Zeit mit der Kommunikation und führen schnell auf den absteigenden Ast der Speedup-Kurve $S(p)$. Die Situation gleicht dem von-Neumann-Flaschenhals. Dieser führt zu Leistungseinbußen bei Zugriff auf den langsamen Hauptspeicher, eine feine Granularität führt zu Verlusten bei Zugriffen auf die Daten anderer Prozessoren.

Da die Granularität einen erheblichen Einfluss auf die Effizienz paralleler Programme hat, ist es vorteilhaft, wenn man sie leicht verändern und damit den Gegebenheiten anpassen kann. Bei der Klimamodellierung gelingt das durch die Wahl der Maschenweite des Gitters. Beim Lösen linearer Gleichungssysteme ist es die Anzahl der Unbekannten, die sich positiv auf die Granularität auswirkt. Deshalb darf bei der HPC-Variante des Linpack-Benchmarks die Zahl der Unbekannten beliebig gewählt werden. Die Details des parallelisierten Gauß'schen Eliminationsverfahrens sind kompliziert, aber es ist ganz offensichtlich unvernünftig, ein System mit nur 1000 Unbekannten auf einem Parallelrechner mit mehreren tausend Prozessoren lösen zu wollen. Mit 32768 Prozessoren wählt man z. B. ein Gleichungssystem mit rund einer Million Unbekannten. Damit erreicht man im HPC-Linpack dann 70 720 GFlops und den ersten Platz in der Liste der 500 schnellsten Computer der Welt (Stand: November 2004, siehe [166]).

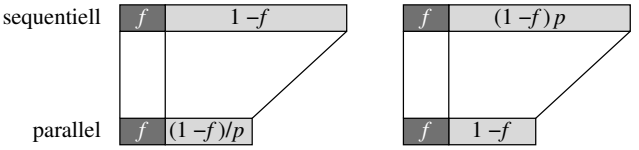

Abb. 1.6. Die Länge der Balken entspricht der Laufzeit, der Speedup ist das Verhältnis der Längen. Amdahls Sichtweise (links) bzw. skalierter Speedup nach Gustafson (rechts).

1.7 Parallele Leistungsmetriken

Amdahls Gesetz beruht auf dem Ansatz, ein Problem fester Größe durch Parallelisierung schneller zu lösen. Die Betrachtungen des letzten Abschnitts haben dagegen gezeigt, dass man mit zunehmender Parallelisierung eher größere Probleme bei gleichbleibender Laufzeit löst.

Ein einfacher Ansatz, der dies berücksichtigt, verwendet die parallele Laufzeit als Basisgröße und vergleicht diese mit der Laufzeit auf einem sequentiellen Rechner. Bei Amdahls Gesetz ist es genau umgekehrt (Abb. 1.6). Sei also $T(p)$ die Laufzeit auf einem parallelen Systems mit p Prozessoren. Ein Anteil f dieser Laufzeit werde dabei mit inhärent sequentiellem Code verbracht. Die entsprechende Laufzeit auf einem sequentiellen Rechner wäre daher $T(1) = fT(p) + p(1-f)T(p)$, und für den Speedup $S(p) = T(1)/T(p)$ ergibt sich

$$S(p) = f + p(1-f) = p + (1-p)f\,. \tag{1.13}$$

Gleichung (1.13) heißt *Gustafsons Gesetz* [60], der so definierte Speedup heißt *skalierter Speedup*. Der Unterschied zwischen unskaliertem (1.4) und skaliertem Speedup (1.13) ist offensichtlich. Bei gegebener Problemgröße (d. h. bei gegebener Laufzeit auf einem sequentiellen Rechner) können wir die Laufzeit des parallelisierbaren Anteils durch viele Prozessoren quasi auf Null reduzieren, behalten aber den unvermeidbaren sequentiellen Anteil. Bei gegebener skalierter Problemgröße (d. h. bei gegebener Laufzeit auf einem Parallelrechner) steigt die hypothetische Laufzeit auf einem sequentiellen Rechner dagegen linear mit p an. Die sequentielle Laufzeit ist dabei hypothetisch, weil das skalierte Problem durchaus so groß sein kann, dass es auf einem sequentiellen Rechner gar nicht mehr ausführbar ist, wie das Problem der globalen Wettervorhersage aus dem Eingangskapitel zeigt.

Gustafsons Gesetz vernachlässigt genau wie Amdahls Gesetz den Overhead, der durch die Parallelisierung ins Spiel kommt. Der skalierte Speedup ist daher als Leistungsmetrik nur bedingt aussagefähig. Besser geeignet ist die *Karp-Flatt-Metrik* [77]. Deren zentrale Idee ist es, den Speedup $S(p) = T(1)/T(p)$ für mehrere Werte von p experimentell zu bestimmen. Das nach f aufgelöste Amdahlsche Gesetz

$$f = \frac{1/S(p) - 1/p}{1 - 1/p} \tag{1.14}$$

erlaubt dann die Berechnung des *empirischen* sequentiellen Anteils f. Das ist die Karp-Flatt-Metrik. Wenn (wie in Amdahls Gesetz angenommen) die Verluste durch

Kommunikation und ungleiche Lastverteilung keine Rolle spielen, sollte f unabhängig von p sein. Ein paralleler Overhead führt dazu, dass $S(p)$ mit wachsendem p langsamer als von Amdahls Gesetz vorhergesagt zunimmt. Das empirische f wird in diesem Fall mit p zunehmen. Die Karp-Flatt-Metrik zeigt also an, ob die Effizienz eines parallelen Programms durch den inhärent sequentiellen Anteil oder durch die parallelen Reibungsverluste dominiert wird. Aus der genauen Abhängigkeit der Karp-Flatt-Metrik von der Anzahl der Prozessoren kann man weitergehende Schlüsse ziehen, z. B. ob die Effizienz von einer ungleichen Lastverteilung limitiert wird oder von den Verlusten, die durch die Kommunikation verursacht werden. In Abschnitt 7.3 werden wir die Karp-Flatt-Metrik am Beispiel der parallelen numerischen Integration diskutieren.

Im Allgemeinen wird die Effizienz (1.11) einer parallelen Anwendung mit wachsender Anzahl p von Prozessoren abnehmen, mit steigender Problemgröße n dagegen zunehmen. Die *Isoeffizienz-Funktion* [55] gibt an, wie stark die Problemgröße wachsen muss, um bei zunehmendem Parallelisierungsgrad die Effizienz konstant zu halten. Ausgangspunkt ist (1.5), wobei wir jetzt auch die Abhängigkeit aller Zeiten von der Problemgröße n berücksichtigen:

$$T(n,p) = \sigma(n) + \frac{\pi(n)}{p} + T_c(n,p)\,. \tag{1.15}$$

Die Effizienz $\varepsilon(n,p) = T(n,1)/(pT(n,p))$ lässt sich dann mit

$$T_0(n,p) = (p-1)\sigma(n) + pT_c(n,p) \tag{1.16}$$

als

$$\varepsilon(n,p) = \frac{\sigma(n)+\pi(n)}{p\sigma(n)+\pi(n)+pT_c(n,p)} = \frac{\sigma(n)+\pi(n)}{\sigma(n)+\pi(n)+T_0(n,p)} \tag{1.17}$$

schreiben. $T_0(n,p)$ ist die Summe aller Zeiten, die die Prozessoren mit Arbeit verbringen, die im sequentiellen Ablauf nicht vorhanden wäre. Der erste Term in $T_0(n,p)$ steht für die redundante Ausführung des sequentiellen Codes auf mehr als einem Prozessor bzw. die Zeit des Wartens, bis ein Prozessor diesen Code ausgeführt hat. Der zweite Term fasst die globalen Verluste durch Kommunikation und ungleiche Lastverteilung zusammen. Berücksichtigt man nun noch, dass

$$T(n,1) = \sigma(n) + \pi(n) \tag{1.18}$$

die sequentielle Laufzeit als Funktion von n angibt, so folgt aus (1.17)

$$T(n,1) = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}\,T_0(n,p)\,. \tag{1.19}$$

Konstante Effizienz $\varepsilon(n,p)$ bedeutet

$$T(n,1) = C\,T_0(n,p)\,, \tag{1.20}$$

d. h., die Problemgröße n muss (mindestens) so schnell wachsen, dass die sequentielle Laufzeit genauso schnell wächst wie die Gesamtlaufzeit des parallelen Overheads. Gleichung (1.20) ist die Isoeffizienz-Funktion.

Betrachten wir dazu das Beispiel der numerischen Wettervorhersage. Hier ist n die Anzahl der Gitterpunkte und $T(n,1) \propto n$. Jeder Prozessor verwaltet ein Teilgitter aus n/p Gitterpunkten, dessen Oberfläche $(n/p)^{2/3}$ Gitterpunkte enthält. Diese müssen bei jeder Iteration mit den Prozessoren ausgetauscht werden, die die sechs benachbarten Teilgitter verwalten. Das führt zu $T_0(n,p) \sim p\,(n/p)^{2/3}$, wobei wir angenommen haben, dass der inhärent sequentielle Teil σ von n unabhängig und damit für große n vernachlässigbar ist. Die Isoeffizienz-Relation ergibt in diesem Fall $n \sim p$. Um die Effizienz zu erhalten, sollte darum bei einer Verdoppelung der Prozessorzahl auch die Anzahl der Gitterpunkte verdoppelt werden.

1.8 Notizen

1.1 Wettervorhersage. Unsere Überlegungen zur numerischen Wettervorhersage sind nur eine Karikatur der Verfahren, mit denen der tägliche Wetterbericht erstellt wird. Der Deutsche Wetterdienst benutzt zum Beispiel mehrere Modelle. Beim globalen Modell beträgt die Maschenweite 60 km. Darin eingebettet ist das lokale Modell, das mit einer Maschenweite von 7 km Mitteleuropa abdeckt. Eine 48-stündige Wettervorhersage für das lokale Modell (325×325 Gitter in 35 vertikalen Schichten, Zeitauflösung 40 Sekunden) kann auf einer IBM RS/6000 SP mit 160 Prozessoren in weniger als einer Stunde erstellt werden [26].

1.2 Grand Challenges. Die hochauflösende, globale Wettersimulation ist nur eine von vielen großen Herausforderungen (*grand challenges*) die den Einsatz massiv paralleler Rechnersysteme erfordern. Weitere Beispiele sind die Proteinfaltung, der numerische Windkanal oder die Genom-Analyse. Eine etwas angestaubte, aber noch immer lesenwerte Darstellung einiger grand challenges finden Sie in [78]. Der Begriff „Grand Challenge“ wurde 1989 von K. G. Wilson geprägt [173]. In der Physik zählen die Quantenchromodynamik und die Simulation der Kollision schwarzer Löcher zu den großen Herausforderungen. Für letztere beträgt der Rechenaufwand etwa 10^{17} Fließkommaoperationen pro Simulation [5], d. h. mehr als drei Jahre auf einem 1 GFlops-Rechner, weniger als zwei Tage auf einem 1 TFlops-Rechner.

1.3 Google. Nicht nur die klassischen numerischen Probleme erfordern massiv parallele Rechner. Eine einzige Anfrage bei der Suchmaschine Google erfordert die Verarbeitung von mehreren 100 MByte an Daten und mehr als 10 Milliarden Rechenoperationen. In Spitzenzeiten wickelt Google tausende Anfragen pro Sekunde ab, jede davon in erstaunlich kurzer Zeit, wie wir alle wissen. Die dafür benötigte Rechenleistung bezieht Google keineswegs aus einem oder wenigen Supercomputern, sondern aus sehr vielen (mehr als 15 000) Standard-PCs, die in geografischen und funktionalen Clustern organisiert sind [6].

1.4 Das Moore'sche Gesetz. Im Jahre 1964 bemerkte der Halbleiter-Ingenieur Gordon Moore, dass sich die Packungsdichte der Schaltelemente auf Silizium-Chips jedes Jahr verdoppelt. Die Datenbasis, aus der Moore diesen Schluss zog, bestand aus nur sieben Datenpunkten, aber seine Hypothese, inzwischen als *Moore'sches Gesetz* bekannt, blieb auch in den folgenden Jahren gültig. Gordon Moore gründete im Jahre 1968 die Firma Intel, die maßgeblich dazu beitrug, dass seine Hypothese bis heute wahr blieb. Ende der siebziger Jahre verlangsamte sich der Fortschritt etwas, denn seitdem verdoppelt sich die Packungsdichte „nur noch" alle 18 Monate. Extrapoliert man die gegenwärtige Rate, so würde ca. im Jahr 2020 jedes Schaltelement aus nur noch einem einzigen Atom bestehen. Die Geschwindigkeit der Prozessoren zeigt eine vergleichbare exponentielle Wachstumsrate. Die Zugriffszeiten der Speicherbausteine halbieren sich dagegen nur alle sieben Jahre, was den von-Neumann-Flaschenhals immer enger werden lässt.

1.5 Linpack. Die Optimierungsfähigkeiten des Compilers haben einen erheblichen Einfluss auf die Ergebnisse. Für die Messungen auf Pentium- bzw. AMD-Systemen haben wir den `gcc`, Version 2.95 verwendet. Auf den Suns kam der Sun Workshop Compiler Version 5.0 zum Einsatz. Lässt man die Optimierungs-Optionen einfach weg, erhält man deutlich schlechtere Resultate, z. B. 187 MFlops statt 752 MFlops für den Athlon-Prozessor. Auf einer Sun unter Solaris gibt Ihnen das Programm `fpversion` übrigens nicht nur interessante Informationen über die Cachegröße, den CPU- und den Speichertakt, sondern auch die Compiler-Optionen für eine optimale Ausnutzung des Caches.

1.6 HPC Challenge. Neben dem Linpack gibt es eine ganze Reihe von weiteren Benchmarks die für numerischen Rechnungen interessant sind. Jack Dongarra, einer der „Väter" des Linpack-Benchmarks und der Top-500 Liste der schnellsten Computer der Welt [166], hat einige davon in einer *High Performance Computing* Benchmark-Suite zusammengefasst. Darin enthalten sind nebem dem Linpack z. B. auch Benchmarks, die explizit die Speicherzugriffszeit ausmessen oder die die Geschwindigkeit des Datentransfers in Verbindungsnetzwerken von Parallelrechnern ermitteln. Der *HPC Challenge* ist frei erhältlich, siehe [67].

1.7 Code-Effizienz. Effizienten Code zu schreiben, ist schon für sequentielle Anwendungen eine hohe Kunst. Falls Sie diese Kunst erlernen möchten, sollten Sie das Buch von Kevin Dowd und Charles R. Severance [32] studieren. Darin und im Web [8] finden Sie auch eine ausführliche Diskussion verschiedener Benchmarks.

1.8 Amdahls Gesetz. Ganze 21 Jahre lang wurden massiv parallele Systeme durch Amdahls Gesetz in eine unbedeutende Nische verbannt. Dann zeigten Gustafson, Montry und Brenner [61], dass man auf einem Parallelrechner mit 1024 Prozessoren für eine ganze Reihe von Anwendungen, wenn man die Probleme nur groß genug macht, einen Speedup von über 1000 erreicht. Dafür erhielten die Autoren den Gordon-Bell-Award 1987. Gustafson schreibt dazu [60]: „We feel that it is important for the computing research community to overcome the 'mental block' against massive parallelism imposed by a misuse of Amdahl's speedup formula ...".

1.9 Superlinearer Speedup. Ein superlinearer Speedup $S(p) > p$ bzw. eine Effizienz $\varepsilon(p) > 1$ sind theoretisch nicht möglich. Praktisch misst man so etwas gelegentlich doch. Ursache dafür ist meistens der Cache: wenn durch Parallelisierung die Datenmenge pro Prozessor kleiner wird (wie bei der Wettersimulation), passen immer größere Teile davon in den Cache, was die Ausführungszeit auf den einzelnen Prozessoren beschleunigt. Wenn dieser Effekt nicht durch die Kommunikationsverluste aufgefressen wird, erhält man einen superlinearen Speedup.