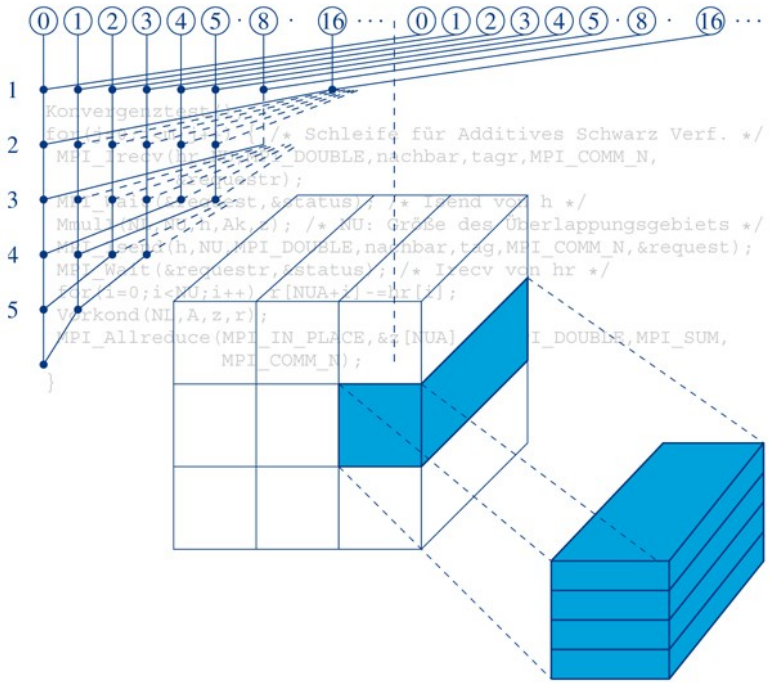
Josef Schüle

Paralleles Rechnen

Performancebetrachtungen zu Gleichungslösungen



Oldenbourg

Paralleles Rechnen

Performancebetrachtungen zu Gleichungslösern

von
Josef Schüle

Oldenbourg Verlag München

Dr. Josef Schüle ist Mitarbeiter am Rechenzentrum der TU Braunschweig und hat einen Lehrauftrag am Institut für Wissenschaftliches Rechnen – Paralleles Rechnen I+II und Hochleistungsrechnen auf Grafikkarten.

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Lektorat: Kathrin Mönch
Herstellung: Anna Grosser
Coverentwurf: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Grafik + Druck GmbH, München

ISBN 978-3-486-59851-3

Inhaltsverzeichnis

Meiner Frau

und all denen,
die mir Mut machten.

1 Einleitung

Paralleles Rechnen ist eine wissenschaftliche Disziplin im Spannungsfeld zwischen Informatik, Mathematik und den Ingenieur- und Naturwissenschaften. Die Entwicklung von Mehrkernprozessoren und deren Einzug in gewöhnliche Arbeitsplatzrechner wird dieser Dispziplin zukünftig zu vermehrter Aufmerksamkeit verhelfen, da die Taktrate von Rechnern seit einigen Jahren nicht mehr ansteigt, sondern nahezu stagniert. Diesem Sachverhalt trägt beispielsweise das BMBF in seiner Richtlinie zur Förderung von Forschungsvorhaben auf dem Gebiet „HPC-Software für skalierbare Parallelrechner“ Rechnung[1]. Der Stagnation bei der Prozessortaktrate versucht man durch Einbau mehrerer Prozessorkerne entgegenzuwirken. Aber was nutzen einzelnen Programmen mehrere langsamere Prozessorkerne, wenn nicht parallel auf diesen Kernen gerechnet werden kann? Erst wenn Teile des Programms gleichzeitig mehrere dieser Prozessoren benutzen, kann die Wartezeit bis zur Beendigung dieses Programms verkürzt werden. Das bedeutet aber, dass Parallelverarbeitung der einzige Weg ist, um zukünftig einer der Hauptanforderungen unserer Zeit gerecht zu werden: Immer schneller zu werden und neue Technologien konkurrenzfähig in kurzen Innovationszyklen zu entwickeln.

Im Unterschied zur Parallelverarbeitung, die über eine Vielzahl von Techniken verfügt, um Prozessabläufe in der IT durch gleichzeitige bzw. nebenläufige Bearbeitung von Prozessen zu beschleunigen, beschäftigt sich das Parallele Rechnen mit der Beschleunigung eines einzelnen Programms, genauer gesagt, eines Rechenprogramms. Dieser altmodische Begriff soll verdeutlichen, dass es sich dabei um Programme handelt, in denen mit Zahlen gerechnet wird und in denen mathematische Algorithmen zur Beschreibung physikalischer und naturwissenschaftlicher Vorgänge eine Rolle spielen. Entsprechend beschäftigt sich Paralleles Rechnen, wie ich es verstehe, mit numerischen Simulationen in den Ingenieurs- und den Naturwissenschaften. Und hier offenbart sich die Interdisziplinarität: Nur durch das Zusammenwirken von mathematischen Algorithmen mit Methoden der Informatik können realitätsnahe wissenschaftliche Ergebnisse aus numerischen Simulationen der Natur- und Ingenieurswissenschaften erzielt werden, die für Wirtschaft, Technik und den Erhalt unserer Lebensgrundlagen eine strukturell entscheidende Bedeutung besitzen.

Paralleles Rechnen, das neben theoretischen Untersuchungen und Experimenten längst zur dritten Säule des Erkenntniserwerbs herangereift ist, erfordert Kenntnisse zur Prozessortechnologie und Verbindungsnetzwerken, um zwei Bereiche aus der Informatik zu nennen, setzt algorithmische Kenntnisse aus der Mathematik voraus und beschreibt Vorgänge in den Ingenieurs- und Naturwissenschaften, deren Verständnis Voraussetzung dafür ist, geeignete Modellbeschreibungen und Algorithmen in Simulationsprogramme umzusetzen.

Die vorliegende Arbeit ist in diesem interdisziplinären Umfeld angesiedelt. Hier wird der Versuch unternommen, Besonderheiten moderner Parallelrechnerarchitekturen und Methoden für das Parallelrechnen auf diesen Architekturen zur erläutern und deren Performance kritisch zu

[1] BMBF Bekanntmachung vom 19.01.2010

untersuchen, um dann anhand eines zentralen mathematischen Verfahrens, der Lösung eines Gleichungssystems, die Effektivität verschiedener Algorithmen, Parallelisierungstechniken und deren Implementierung zu analysieren und zu vergleichen. Aufbauend auf diesen Performancebetrachtungen werden verschiedene Strategien im Design paralleler Programme bewertet. Dabei wird der Einsatz dieser Techniken und die entwickelten Designstrategien mit Hinweisen und kurzen Ausführungen zu verschiedenen wissenschaftlichen Projekten, wie Berechnungen von optischen Wellenleitern in der Elektrotechnik, Simulationen von Meeresströmungen, Simulationen zur Leitfähigkeitsberechnungen in der theoretischen Chemie, Teilchensimulationen zur Berechnung von Ionentriebwerken und zur Berechnung von Strömungen des Sonnenwindes um Planeten in der theoretischen Physik mit entsprechenden Anwendungsgebieten in den Ingenieur- und Naturwissenschaften verzahnt.

Im ersten Kapitel werden einige Grundbegriffe des Parallelen Rechnens erklärt und erläutert. Hier finden sich Diskussionen zum Speedup und zur Skalierbarkeit eines Programms neben Ausführungen zur Fließbandverarbeitung und zu Performancemessungen und zum Aufbau moderner Rechnerarchitekturen. Allerdings werden hier nur prinzipielle Architekturmerkmale ausgeführt, da dieser Bereich sehr kurzen Innovationszyklen unterliegt.

In den meisten Simulationsprogrammen spielt die Performance in der traditionellen Recheneinheit, der CPU, allerdings inzwischen eine untergeordnete Rolle. Vielmehr zeigten die Entwicklungen und Fortschritte in der Prozessortechnologie in den letzten Jahrzehnten eine kontinuierliche hohe Steigerungsrate, die nach der Mooreschen Gesetzmäßigkeit etwa alle zwei Jahre zu einer Verdoppelung der Performance führte, mit der die Entwicklungen in der Speichertechnologie nicht Schritt halten konnten.

Als Folge davon ist eine Neubewertung von Algorithmen notwendig: Nicht Rechenoperationen bestimmen die Laufzeit eines Programms, sondern die Zahl und die Art der Speicherzugriffe. Diese Aussage zieht sich gewissermaßen als Leitfaden durch die restlichen Kapitel. Um der zentralen Rolle von Speicherzugriffen gerecht zu werden, werden im ersten Kapitel verschiedene Speichertechnologien und Speicherhierarchien ausführlich behandelt und mit aktuellen Messungen von Speicherbandbreiten unterlegt. Auf der Basis dieser Messungen wird ein theoretisches Performancemodell um die Vorhersagemöglichkeit von Speicherzugriffszeiten erweitert.

Paralleles Rechnen mit gemeinsamen Variablen bietet sich auf Rechnern mit modernen Mehrkernprozessoren an, da diese Rechner einen gemeinsamen physikalischen Speicher besitzen. Dieser gemeinsame Speicher erlaubt es Threads, auf gemeinsame Adressen zuzugreifen.

Die wohl am häufigsten in numerischen Simulationen eingesetzte Programmiertechnik für das Parallele Rechnen mit gemeinsamen Variablen ist OpenMP, das in Kapitel 2 vorgestellt und anhand von Beispielen erläutert wird. Ein wichtiger Gesichtspunkt für die Effektivität von parallelen Programmen auf Rechnern mit gemeinsamem Speicher ist die Affinität einer Thread zu einem Prozessorkern und somit zu dem von ihr allokierten Speicher. Diesem Aspekt und ganz allgemein den Kosten für die wichtigsten Syntaxelemente von OpenMP wird in diesem Kapitel besondere Aufmerksamkeit gewidmet.

Ein klassischer Parallelrechner besteht konzeptionell aus einzelnen Rechenknoten, die mit einem Verbindungsnetzwerk miteinander verbunden sind. Dabei ist jeder dieser Rechenknoten ein vollständiges System, das insbesondere seinen eigenen Speicher besitzt, der für Prozesse auf diesem Rechenknoten privat zur Verfügung steht. Ein Datenaustausch zwischen zusam-

menwirkenden Prozessen auf einem derartigen Parallelrechner kann nur durch expliziten Informationsaustausch erfolgen, weswegen derartige Rechnersysteme als Message Passing Systeme bezeichnet werden, auf denen Message Passing Programmiertechniken zum Einsatz kommen. Die am häufigsten eingesetzte Programmiertechnik für das Parallele Rechnen in numerischen Simulationen ist das Message Passing Interface, MPI, dessen Einsatz in Kapitel 3 anhand zahlreicher Beispiele und Graphiken erläutert wird. Dabei liegt das Hauptaugenmerk auf dem praktischen Einsatz und den zugrundeliegenden Ideen und Konzepten und nicht auf einer umfassenden Beschreibung des Sprachstandards in den derzeit vorliegenden Versionen. Da sich hybrides Rechnen mit Threads und MPI Prozessen auf modernen Clustern mit Mehrkehrnprozessorknoten anbietet, wird kurz auf aktuelle Diskussionen bei der Weitereintwicklung zu MPI-3 eingegangen.

Ein konzeptionell unterschiedliches Programmiermodell liegt der Message Passing Umgebung Parallel Virtual Machine, PVM, zugrunde, auf dessen Aufbau und Funktionsweise an Beispielen zum Abschluss von Kapitel 3 kurz eingegangen wird. Obwohl PVM kaum noch weiterentwickelt wird, ist es nach wie vor für gewisse Anwendungsprobleme MPI konzeptionell überlegen, woraus eine nicht vernachlässigbare Verbreitung resultiert.

In den ersten drei Kapiteln werden Rechnerarchitekturen, Programmiertechniken und Performancegesichtspunkte im Hinblick auf deren Bedeutung für das Parallele Rechnen diskutiert, d.h. gewissermaßen überdecken diese Kapitel die informatischen Gesichtspunkte beim interdispziplinären Parallelen Rechnen. In den folgenden drei Kapitel werden diese Gesichtspunkte in mathematische Algorithmen übertragen und durch praktische Beispiele aus Forschungsarbeiten aus verschiedenen ingenieurs- und naturwissenschaftlichen Beispielen ergänzt. Dabei werden mathematische Algorithmen zur Lösung linearer Gleichungssysteme als Vehikel für Konzepte, Ideen und Techniken eingesetzt, die auf viele andere Algorithmen und Anwendungsfelder übertragbar sind.

In Kapitel 4 wird die algorithmische Umsetzung und Performance der Gaußelimination untersucht und ein Programmdesign zur Parallelisierung diskutiert. Ausgehend vom klassischen Verfahren, so wie es noch in den meisten Lehrbüchern zur Numerischen Mathematik zu finden ist, werden sinnvolle Speicherallokierungstechniken für mehrdimensionale Felder vorgestellt und die Performance des klassischen Algorithmus analysiert. Dabei wird auf die Ausführungen zu Speicherarchitekturen und -techniken aus Kapitel 1 zurückgegriffen. Auf dieser Grundlage werden Blockalgorithmen und ganz allgemein die Performance von Algorithmen der linearen Algebra in Abhängigkeit vom Verhältnis arithmetischer Operationen zu Speicherzugriffen klassifiziert. Diese Techniken werden mit ausführlichen Programmbeispielen schrittweise zu einem Gaußschen Eliminationsverfahren umgesetzt, das nahezu die Hardwareperformance eines einzelnen Prozessors erzielt, und somit die Effektivität des klassischen Verfahrens um ein Vielfaches übertrifft.

Das Herzstück dieses 4. Kapitels ist jedoch das in dieser Arbeit entwickelte Design paralleler Programme, das in eine Designschleife, bestehend aus Programmzerlegung und Analyse von Abhängigkeiten und des Kommunikationsaufkommens zwischen Prozessen, mit sich anschließender Zuweisung auf Prozessoren eingeteilt wird. Im weiteren Verlauf des Kapitels werden die Designschritte an Beispielen aus den Ingenieurs- und Naturwissenschaften erläutert. So werden mögliche Programmzerlegungen an Simulationen zur Meeresströmungen in der Ostsee diskutiert und die Bedeutung strukturierter Abhängigkeiten und regelmäßiger Kommunikationsmuster für die Programmeffektivität von Simulationen zur Leitfähigkeitsberechnung aufge-

zeichnet. Bei der Zuweisung auf Prozessoren werden Scheduling Strategien diskutiert, die bei funktionellen Zerlegungen Verwendung finden und exemplarisch bei der Parallelelisierung der Gaußelimination eingesetzt werden. Daneben werden Methoden zur Gebietsaufteilung vorgestellt, die bei datenparallelen Ansätzen und Gebietszerlegungen zum Einsatz kommen. In der abschließenden Fallstudie zu wechselwirkenden Teilchen, einem Beispiel aus der theoretischen Physik, mit dem die Strömungsverhältnisse bei Ionentriebwerken simuliert werden, werden Techniken der datenparallelen Zerlegung mit Scheduling Strategien kombiniert, um zu einem effektiven parallelen Verfahren zu gelangen.

Im 5. Kapitel wird der Einfluss unterschiedlicher Speicherungstechniken dünn besetzter Matrizen auf die Effektivität der Matrix–Vektor Multiplikation (MVM) und die Vorkonditionierung untersucht und mit zahlreichen Programmbeispielen veranschaulicht. Die MVM und die Vorkonditionierung sind die zentralen Schritte für die Performance iterativer Lösungsverfahren für lineare Gleichungssysteme, deren Parallelisierung exemplarisch für das konjugierte Gradientenverfahren diskutiert wird. Bei der Untersuchung der Effektivität der MVM werden der Reihe nach alle gängigen Speichertechniken für dünn besetzte Matrizen vorgestellt und die Effektivität mit Hilfe des in Kapitel 1 aufgestellten Performancemodells abgeschätzt. Nach diesen Abschätzungen werden die Ergebnisse der Modellbetrachtungen auf die Lösung des Vorkonditionierungsproblems übertragen, wobei hier exemplarisch die unvollständige LR–Zerlegung herausgegriffen wird. Insbesondere kommen dabei Färbetechniken zum Einsatz, die sich, wie schon im Designprozess in Kapitel 4 und bei dem Gauß–Seidel Verfahren, als eine der wichtigen Techniken für die Parallelisierung erweisen. Allerdings kann das reine konjugierte Gradientenverfahren, mit seinen Möglichkeiten der fein granularen Parallelisierung, kaum hoch skalierend parallel implementiert werden.

Zu hoch skalierenden parallelen Verfahren gelangt man mit den in Kapitel 6 beschriebenen Gebietszerlegungsmethoden, die die mathematischen Algorithmen für direkte Löser, die in Kapitel 4 beschrieben sind, und dem konjugierten Gradienten Verfahren, das in Kapitel 5 ausgeführt ist, kombinieren. Die Gebietszerlegungsmethoden, die sowohl mit überlappenden Teilgebieten als auch ohne Überlappung der Teilgebiete mit einem oder mehreren Diskretisierungsgittern durchgeführt werden können, eröffnen mit ihren Möglichkeiten zur grob granularen Parallelisierung den Einsatz massiver Parallelrechner, wie in diesem Kapitel algorithmisch ausgeführt wird.

In Kapitel 7 werden die algorithmischen Techniken aus den vorangegangenen Kapiteln aufgegriffen und auf den Einsatz von Graphical Processing Units (GPUs) übertragen, die neben Mehrkernprozessoren verstärkt als eigenständige Recheneinheiten oder als Beschleunigerkarten eingesetzt werden. Dieser Entwicklungstrend wird neben der immensen möglichen Performancesteigerung durch eine gleichzeitige Reduzierung der Betriebskosten beim Einsatz dieser Recheneinheiten getrieben, was bei steigenden Stromkosten zunehmend an Bedeutung gewinnt. Hier werden die Besonderheiten, SIMD Architektur, Threadgruppen und Kettenfäden und ihre Auswirkungen auf die Programierung an Beispielen erläutert, wobei die Algorithmen aus Kapitel 4 und 5 aufgegriffen und übertragen werden.

Die Interdisziplinarität im Parallelen Rechnen ist Gegenstand dieser Arbeit, die sich nicht darauf konzentriert, detailliertes Spezialwissen in einem der Fachgebiete darzulegen. Stattdessen steht mit theoretischen Modellberechnungen, dem Design paralleler Programme und dem Einsatz von Blockalgorithmen die Kombination der Mathematik und der Informatik im Vordergrund, die mit Beispielen aus wissenschaftlichen Projektarbeiten mit den Natur- und Ingenieurwissenschaften als dritte wissenschaftliche Disziplin verzahnt werden.

2 Modellbetrachtungen, Speicher- und Rechnerarchitekturen

... adding manpower to a late software project makes it later.

(Frederick P. Brooks, Jr. in „The Mythical Man–Month“)

2.1 Einleitung

In diesem Kapitel sollen einige Grundbegriffe des Parallelen Rechnens erklärt und erläutert werden. Die Beschleunigung eines Simulationsprogramms, eines der Ziele des Parallelen Rechnens, wirft natürlich Fragen auf: Wie kann dieses Ziel erreicht werden? Welche Beschleunigung (Speedup) ist dabei möglich? Kann diese mögliche Beschleunigung vorhergesagt werden und wenn ja, wie?

Paralleles Rechnen bedeutet natürlich nicht nur, ein Simulationsprogramm schneller zu beenden, sondern insbesondere auch, dass dieses Programm nach wie vor das richtige Ergebnis liefert. Deswegen wird auch die Thematik Fehler in Programmen im Vergleich zwischen parallelen und sequentiellen Programmen kurz angesprochen.

Natürlich hängt die Performance eines parallelen Programmms ganz entscheidend von der Performance auf einem einzelnen Prozessor ab, weswegen einige Hinweise zur Fließbandverarbeitung und Leistungsmessungen von Prozessoren ausgeführt werden und in einem kleinen Überblick aktuelle Rechnerarchitekturen und Auswirkungen dieser Architekturen auf unsere Arbeit diskutiert werden.

Ein besonderer Schwerpunkt dieses Kapitels liegt auf der Architektur des Hauptspeichers und von Speicherhierarchien. Seit Jahren führen die Innovationen in der Prozessortechnologie dazu, dass der Unterschied zwischen der Geschwindigkeit des Prozessors auf der einen Seite und des Speichers auf der anderen Seite immer weiter auseinander klafft. Deswegen ist für hochperformante Programme eine Kenntnis der Speicherarchitektur unabdinglich. Die hier erzielten Ergebnisse fließen direkt in ein erweitertes Performancemodell ein, auf das in den folgenden Kapiteln häufig zurückgegriffen wird.

Doch vorab noch eine Aufstellung notwendiger Grundbegriffe:

- **LAN**: local area network. Typischerweise ein Firmen- oder Campusnetzwerk auf Ethernetbasis mit Internetprotokoll.
- **Lock**: Zugriffsbeschränkung auf eine Ressource, die nur exklusiv benutzt werden kann.
- **Port**: Sowohl für Hard- als auch Software benutzte Bezeichnung für einen Kommunikationsendpunkt; gewissermaßen das Tor zwischen Netzwerk und Rechner.

- **Stack**: Speicherbereich, der lokale Variablen, Rücksprungadressen und Werte von Funktionsparametern enthält.
- Eine **Task** ist ein selbständiger Prozess. Mehrere Tasks können beispielsweise ein paralleles Programm bilden.
- Eine **Thread** ist eine nicht selbständige Untereinheit einer Task. Mehrere Threads können eine Task bilden und teilen sich gemeinsam die Betriebsmittel der Task wie z.B. das Datensegment und die Dateideskriptoren.

2.2 Speedup

Das englische Wort Speedup ist in diesem Zusammenhang so weit verbreitet und etabliert, dass ich es hier als Fachwort benutzen werde.

Wozu wollen wir uns der Mühe des Parallelen Rechnens unterziehen? Warum warten wir nicht einfach mit unserem Programm auf die nächste Prozessorgeneration und lassen es dann darauf schneller laufen? Ganz einfach – weil Zeit in Forschung und Wirtschaft knapp ist.

- Unser Rechenprogramm wird in einer **Wettbewerbssituation** eingesetzt. Sei dies nun in der Forschung, in der wir mit anderen Forschungseinrichtungen in der Welt konkurrieren oder sei dies im wirtschaftlichen Kontext, wo eine schnellere Markteinführung eines Produkts einen höheren Absatz sichert.
- Unser Rechenprogramm steht unter einem **Terminierungsdruck**. Sei es nun das Wetter, das heute für den Folgetag vorhergesagt werden soll, und nicht erst übermorgen, oder seien es Vorhersagen für die Börsenentwicklungen oder sei es eine akademische Arbeit oder ein Konferenzbericht, der zu einem gewissen Zeitpunkt abgegegeben werden muss.
- Unser Rechenprogramm soll eine der wichtigen Herausforderungen an die Wissenschaft lösen, d.h. das Problem stellt eine sog. **Grand Challenge** dar, wie etwa die Proteinfaltung, die von großem allgemeinen Interesse und wissenschaftlicher Bedeutung ist und nur durch neueste Techniken und massivstem Ressourceneinsatz gelöst werden kann.

Die obigen Situationen stellen Musterbeispiele für die Notwendigkeit dar, eine Berechnung in möglichst kurzer Zeit zu beenden. Ein weiterer, zunächst nicht so offensichtlicher Grund für den Einsatz eines Parallelrechners ist die Tatsache, dass meistens nur Rechner mit vielen Prozessoren genügend Hauptspeicher haben, um wirklich große Probleme überhaupt behandeln zu können. Diese Art von Problemen, bei der es eher um die Machbarkeit als solches als um die Verkürzung der Wartezeit auf die Beendigung des Programms geht, soll in diesem Kapitel nicht näher behandelt werden.

Also: Unsere Antriebsfeder ist die Wartezeit vom Beginn eines Programms bis zu dessen Beendigung. Diese Wartezeit hat ihre Ursache oft in der hohen Auslastung des Rechners, wenn lange Zeit darauf gewartet werden muss, bis uns die Prozessoren etwa in einem Batchsystem zur Verfügung gestellt werden oder wenn in einer Multi–User-Umgebung das Betriebssystem[1] zwischen verschiedenen Anforderungen hin und her wechselt. Diese Aspekte bleiben hier un-

[1] Man spricht dann von time sharing environment.

berücksichtigt; wir betrachten nur die Zeit zwischen dem Beginn eines Programms und dessen Beendigung. Wir wollen diese Zeit Laufzeit[2] nennen, um sie explizit von der Rechenzeit, auch CPU–Zeit genannt, zu unterscheiden. Letzteres ist die Zeit, die der Rechner, genauer gesagt die CPU, noch genauer, die beschäftigten Kerne der CPU, für das Programm arbeiten. Nur in wenigen Ausnahmefällen kann Paralleles Rechnen dazu führen, dass die CPU–Zeit reduziert wird. Schließlich müssen eine Vielzahl von Operationen ausgeführt werden, wobei deren Anzahl durch den Einsatz mehrerer Prozessoren in der Regel ja nicht verringert wird – im Gegenteil. Nein, es geht beim Parallelen Rechnen ausschließlich um die Laufzeit; die Zeit, die wir auf ein Ergebnis warten.

Es liegt auf der Hand, als Speedup das Verhältnis der Laufzeiten beim Einsatz eines Prozessors und p Prozessoren zu vergleichen:

$$S(p) = \frac{T_1}{T_p}\,. \tag{2.1}$$

Wenn wir diese Definition des Speedups auf unser Alltagsleben als Qualitätsmaßstab übertragen und Techniken mit großem Speedup bevorzugen, dann sollten z.B. zum Ausheben einer großen Grube viele Bauarbeiter mit einer Schaufel eingesetzt werden. Diese Technik besitzt sicherlich einen größeren Speedup als den Einsatz eines Baggers. Daran wird die Schwäche dieser Definition (2.1) deutlich. Sie wird etwas sinnvoller, wenn wir stattdessen setzen:

$$S(p) = \frac{\text{Bestes Programm für einen Prozessor}}{T_p}\,. \tag{2.2}$$

Gleichung (2.1) wird relativer Speedup bezeichnet und am häufigsten benutzt, da oft genug das „Beste Programm“ nicht bekannt oder nicht verfügbar ist, so dass der absolute Speedup (2.2) gar nicht zugänglich ist.

Idealerweise ist $S(p) = p$, d.h. bei 1000 Prozessoren ist das Programm 1000mal schneller fertig. Man spricht dann von idealem Speedup.

Aber nochmals zurück zu unseren Bauarbeitern und dem Bagger. Was sagt uns der in den meisten Veröffentlichungen vorzufindende relative Speedup? Meist nicht viel, wenn er besonders groß ist. Bei kleinen Werten offenbart er Probleme. Was uns wirklich interessiert, sind Wartezeiten und in möglichst kurzer Laufzeit ein Problem zu lösen, ohne dabei viele Prozessoren völlig ineffektiv zu benutzen. Die Effektivität für den Einsatz von Prozessoren ergibt sich dabei aus

$$E(p) = \frac{S(p)}{p}\,, \tag{2.3}$$

sprich, dem Speedup dividiert durch die Zahl der eingesetzten Prozessoren.

Ein erstes Modell zur Vorhersage des Speedups geht auf Gene Amdahl [2], geb. 1922 und Gründer der Amdahl Corporation, zurück. Es geht bei fester Problemgröße davon aus, dass ein Programm sich in nicht parallelisierbare Anteile, wie Initialisierungen, Speicherallokierungen, Ein-/Ausgabeoperation etc., und parallelisierbare Anteile zerlegen lässt, d.h.

$$T_1 = T_1(1-f) + T_1 \cdot f\,.$$

[2] Im Englischen wird meist wallclock time benutzt.

Dabei ist f der Anteil im Programm, der ideal parallelisierbar ist. Folglich ergibt sich für die parallele Rechenzeit ein konstanter unveränderlicher Anteil $T_1(1-f)$, während der parallele Anteil in diesem Ansatz ideal abnimmt. Für den relativen Speedup (2.1) ergibt sich daraus

$$S = \frac{T_1}{T_1 \cdot ((1-f) + f/p)} = \frac{p}{(1-f)p + f} \tag{2.4}$$

mit dem Grenzwert:

$$\lim_{p \to \infty} S(p) = \frac{1}{1-f}\,. \tag{2.5}$$

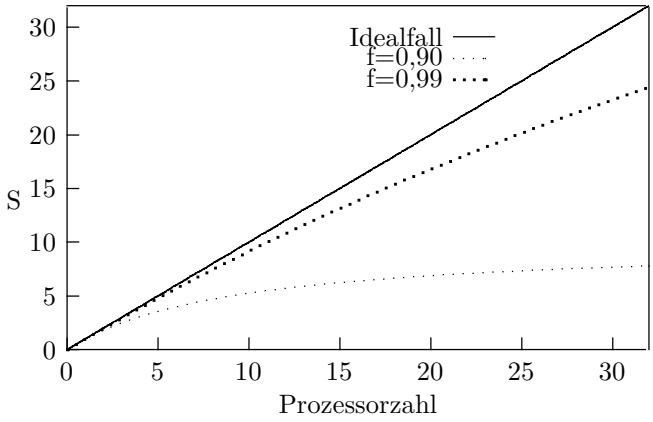


Abbildung 2.1: *Speedup nach dem Amdahlschen Gesetz* (2.4) *für verschiedene ideal parallelisierbare Anteile* f.

Obwohl Abb. 2.1 gerade einmal bis zur Prozessorzahl 32 reicht, was nun nicht gerade massiv parallel zu bezeichnen ist, sind die beiden Kurven für $f = 0,9$ und $f = 0,99$ weit vom Idealfall entfernt. Wie die Grenzwertbetrachtung (2.5) erkennen lässt, ist selbst bei unendlicher Prozessorzahl für $f = 0,99$ nur ein Speedup von 100 zu erwarten. Mehr ist nicht möglich.

Das Amdahlsche Gesetz gibt einen sehr pessimistischen Ausblick für das Parallele Rechnen, was seinen Hauptgrund darin hat, dass es von einer konstanten Problemgröße ausgeht. Seltener benutzt und hier nur kurz erwähnt sei die Karp–Flatt Metrik [77], die wie das Amdahlsche Gesetz ein ähnlich pessimistisches Verhalten für große Prozessorzahlen für Probleme fester Größe vorhersagt. Einen anderen Weg, der später (s. S. 9) diskutiert wird, gibt Gustafson vor. Neben der festen Problemgröße sind die Nichtberücksichtigung von heterogenen Prozessoren, die Veränderung des nicht parallelisierbaren Anteils mit der Prozessorzahl und Algorithmenwechsel bei parallelen Programmen weitere Schwachpunkte im Amdahlschen Gesetz, um nur zwei zu nennen. Nichts desto trotz wird dieses Gesetz häufig zitiert, denn effektive Parallelisierung bedeutet einen möglichst hohen parallelen Anteil f. Und das bedeutet, dass möglichst alle Teile eines Programms parallelisiert werden. Es nützt eben nichts, wenn nur 99% der Programmzeilen parallelisiert sind, wenn das Programm auf tausenden Prozessoren rechnen soll.

Daraus kann man ersehen, dass massiv parallele Programme viel Arbeit verlangen. Aus dem Amdahlschen Gesetz kann außerdem gefolgert werden, dass das Umschreiben eines existierenden sequentiellen Programms für einen Parallelrechner oft der falsche Weg ist. Stattdessen sollte im Vorfeld über Algorithmenwechsel und Neuprogrammierung nachgedacht werden.

Ausführlichere Diskussionen und eine Methode zur experimentellen Bestimmung von f finden sich in Ref. [126].

2.3 Skalierbarkeit

John Gustafsons [66] Ansatz zur Modelierung des Speedups paralleler Programm gleicht formal dem von Amdahl und unterscheidet sich zunächst nur in der Betrachtungsfolge. Hier ist die parallele Zeit für eine Problemgröße N Ausgangspunkt, d.h. es wird von der Zerlegung $T_p = ((1-f)+f)T_p$ in einen sequentiellen und einen ideal parallelen Teil ausgegangen. Folglich ergibt sich daraus die sequentielle Zeit zu $T_1 = T_p((1-f)+pf)$ und für den Speedup:

$$S_G(p,N) = \frac{(1-f+p\cdot f)T_p}{T_p} = 1 + f\cdot(p-1)\,. \tag{2.6}$$

Da $p \geq 1$ ist S_G stets positiv. Formel 2.6 wird Gustafsonsches Gesetz genannt. Man erkennt sofort, dass der Speedup nach Gustafson zu wesentlich optimistischeren Aussagen für das Parallele Rechnen führt. Insbesondere wächst S_G linear mit p ins Unendliche und erreicht nicht wie bei Amdahl bereits vorzeitig einen asymptotischen Wert.

Das Gustafsonsche Gesetz ist dabei nicht einfach nur ein Taschenspielertrick, sondern enthält in seiner Herleitung einen realistischen Hintergrund. Die Operationen in numerischen Algorithmen zeigen nämlich oft $\mathcal{O}(N^m)$, $m \geq 2$ Abhängigkeiten von der Problemgröße N, d.h., wird die Problemgröße für $m = 2$ verdoppelt, dann steigen die Operationen und meist damit auch die Rechenzeit für ein entsprechendes Programm auf das Vierfache an. Sequentielle Programmteile wie Initialisierungen etc. wachsen dagegen meist weniger stark mit der Problemgröße N. Hier gilt oft $\mathcal{O}(N^q)$ mit $q \leq 1$. Diese Tatsache wird bei Amdahl vernachlässigt und ist bei Gustafson implizit enthalten.

Angenommen, wir hätten ein Programm mit $\mathcal{O}(N^2)$ Operationen im ideal parallelisierten Rechenteil, der auf 4 Prozessoren 1 Std. benötigt, und zusätzlich einen sequentiellen Intialisierungsteil mit $\mathcal{O}(N^1)$, der 0,1 Std. dauert, d.h. $f(N) \approx 0,91$. Dieses Programm benötigt auf einem Prozessor

$$T_1(N) = 0,1 + 4\cdot 1 \text{ Stunden}$$

mit einem Speedup von

$$S_G(p,N) \approx 1 + 0,91(p-1)\,.$$

Verdoppeln wir nun N, so wird sich die Rechenzeit des parallelisierten Teils um einen Faktor 4 und die Initialisierung um einen Faktor 2 verlängern, so dass die Rechnung auf 4 Prozessoren nun $4,2$ Stunden benötigt. Sequentiell würde dieses Programm stattdessen $T_1(2N) = 0,2 + 4\cdot 4 = 16,2$ Stunden dauern. Um $S_G(2N)$ vorherzusagen, muss nun auch f neu bestimmt werden. Wir erhalten $f = 4/4,2 \approx 0,95$, d.h. der Speedup fällt für das größere Problem höher aus.