eXamen.press

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Thomas Rauber · Gudula Rünger

Parallele Programmierung

3. Auflage

Springer

Thomas Rauber
Universität Bayreuth
Fakultät für Mathematik, Physik
und Informatik
Bayreuth
Deutschland

Gudula Rünger
Technische Universität Chemnitz
Fakultät für Informatik
Chemnitz
Deutschland

ISSN 1614-5216
ISBN 978-3-642-13603-0
ISBN 978-3-642-13604-7 (eBook)
DOI 10.1007/978-3-642-13604-7

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

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer DE ist Teil der Fachverlagsgruppe Springer Science+Business Media
www.springer.de

Vorwort

Das Anliegen dieses Buches ist es, dem Leser detaillierte Kenntnisse der Parallelverarbeitung zu vermitteln und ihn insbesondere mit dem heutigen Stand der Techniken der parallelen Programmierung vertraut zu machen. Das vorliegende Buch ist die dritte Auflage des im Jahr 2000 erstmals erschienenen Buches *Parallele und Verteilte Programmierung*; die zweite Auflage stammt aus dem Jahr 2007. Seit dem Erscheinen der ersten Auflage hat die technologische Entwicklung u. a. durch die weite Verbreitung von Clustersystemen und die Einführung von Multicore-Prozessoren dazu geführt, dass die Techniken der parallelen Programmierung enorm an Wichtigkeit zugenommen haben. Dies gilt nicht nur für die bisherigen Hauptanwendungsgebiete im Bereich wissenschaftlich-technischer Berechnungen. Die parallele Programmierung spielt auch für die effiziente Nutzung typischer Desktop-Rechner eine große Rolle, so dass sich parallele Programmiertechniken in alle Bereiche der Softwareentwicklung ausbreiten. Ein neuer Trend ist auch die Auslagerung von Berechnungen auf Graphics Processing Units (GPUs), die mehrere Hundert Prozessorkerne umfassen können und somit ein großes Rechenpotential zur Verfügung stellen. Durch die durchgängig zur Verfügung stehende parallele Hardware werden in Zukunft Standard-Softwareprodukte auf Konzepten der parallelen Programmierung basieren, um die parallelen Hardwareressourcen auch ausnutzen zu können. Dadurch ergibt sich ein enormer Bedarf an Softwareentwicklern mit parallelen Programmierkenntnissen. Entsprechend fordern Prozessorhersteller, die parallele Programmierung als obligatorische Komponente in die Curricula der Informatik aufzunehmen.

Die vorliegende dritte Auflage trägt den neuen Entwicklungen dadurch Rechnung, dass die für die Programmierung von Multicore-Prozessoren erforderlichen Programmiertechniken einen breiten Raum einnehmen. Insbesondere wurde ein Kapitel zur Programmierung von GPUs neu hinzugefügt. Die anderen Kapitel wurden entsprechend der technologischen Entwicklung überarbeitet. Die Überarbeitung betrifft insbesondere das Kapitel über die Architektur paralleler Plattformen, das verstärkt die Architektur von Multicore-Prozessoren behandelt.

Das Buch ist thematisch in drei Hauptteile gegliedert, die alle Bereiche der Parallelverarbeitung beginnend mit der Architektur paralleler Plattformen bis hin zur Realisierung paralleler Anwendungsalgorithmen behandeln. Breiten Raum nimmt die eigentliche parallele Programmierung ein. Im ersten Teil geben wir einen kurzen Überblick über die Architektur paralleler Systeme, wobei wir uns vor allem auf wichtige prinzipielle Eigenschaften wie Cache- und Speicherorganisation oder Verbindungsnetzwerke einschließlich der Routing- und Switching-Techniken konzentrieren, aber auch Hardwaretechnologien wie Hyperthreading oder Speicherkonsistenzmechanismen behandeln.

Im zweiten Teil stellen wir Programmier- und Kostenmodelle sowie Methoden zur Formulierung paralleler Programme vor und beschreiben derzeit aktuelle portable Programmierumgebungen wie MPI, Pthreads, Java-Threads und OpenMP. Neu aufgenommen wurde der Bereich der GPU-Programmierung mit CUDA und OpenCL einschließlich der Beschreibung aktueller GPU-Architekturen. Ausführliche Programmbeispiele begleiten die Darstellung der Programmierkonzepte und dienen zur Demonstration der Unterschiede zwischen den dargestellten Programmierumgebungen.

Im dritten Teil wenden wir die dargestellten Programmiertechniken auf Algorithmen aus dem wissenschaftlich-technischen Bereich an. Wir konzentrieren uns dabei auf grundlegende Verfahren zur Behandlung linearer Gleichungssysteme, die für eine praktische Realisierung vieler Simulationsalgorithmen eine große Rolle spielen. Der Schwerpunkt der Darstellung liegt dabei nicht auf den mathematischen Eigenschaften der Lösungsverfahren, sondern auf der Untersuchung ihrer algorithmischen Struktur und den daraus resultierenden Parallelisierungsmöglichkeiten. Zu jedem Algorithmus geben wir z. T. mehrere, repräsentativ ausgewählte Parallelisierungsvarianten an, die sich im zugrunde liegenden Programmiermodell und der verwendeten Parallelisierungsstrategie unterscheiden. Eine Webseite mit begleitendem Material ist unter ai2.inf.uni-bayreuth.de/pp_buch_3A eingerichtet. Dort werden u. a. weitere Materialien zum Inhalt des Buches sowie Informationen zu neueren Entwicklungen zur Verfügung gestellt.

Bei der Erstellung des Manuskripts haben wir vielfältige Hilfestellung erfahren, und wir möchten an dieser Stelle all denen danken, die am Zustandekommen dieses Buches beteiligt waren. Für zahlreiche Anregungen und Verbesserungsvorschläge danken wir Jörg Dümmler, Sergei Gorlatch, Reiner Haupt, Hilmar Hennings, Klaus Hering, Michael Hofmann, Christoph Keßler, Raphael Kunis, Jens Lang, Paul Molitor, John O'Donnell, Robert Reilein, Carsten Scholtes, Michael Schwind und Reinhard Wilhelm. Kerstin Beier, Erika Brandt, Daniela Funke, Monika Glaser, Ekkehard Petzold, Michael Stach, Luise Steinbach und Michael Walter danken wir für die Mitarbeit an der LaTeX-Erstellung des Manuskriptes. Viele weitere Personen haben zum Gelingen dieses Buches durch zahlreiche Hinweise beigetragen; auch ihnen sei hiermit gedankt. Nicht zuletzt gilt unser Dank dem Springer-Verlag für die effiziente und angenehme Zusammenarbeit.

Bayreuth und Chemnitz, Mai 2012

Thomas Rauber
Gudula Rünger

Inhaltsverzeichnis

Kapitel 1
Einleitung

1.1 Motivation

Ein seit längerem zu beobachtender Trend ist die ständig steigende Nachfrage nach immer höherer Rechenleistung. Dies gilt insbesondere für Anwendungen aus dem Bereich der Simulation naturwissenschaftlicher Phänomene. Solche Anwendungen sind z. B. die Wettervorhersage, Windkanal- und Fahrsimulationen von Automobilen, das Design von Medikamenten oder computergraphische Anwendungen aus der Film-, Spiel- und Werbeindustrie. Je nach Einsatzgebiet ist die Computersimulation entweder die wesentliche Grundlage für die Errechnung des Ergebnisses oder sie ersetzt bzw. ergänzt physikalische Versuchsanordnungen. Ein typisches Beispiel für den ersten Fall ist die Wettersimulation, bei der es um die Vorhersage des Wetterverhaltens in den jeweils nächsten Tagen geht. Eine solche Vorhersage kann nur mit Hilfe von Simulationen erreicht werden. Ein weiteres Beispiel sind Havariefälle in Kraftwerken, die in der Realität naheliegenderweise nur schwer oder mit gravierenden Folgen nachgespielt werden könnten. Der Grund für den Einsatz von Computersimulationen im zweiten Fall ist zum einen, dass die Realität durch den Einsatz eines Computers genauer nachgebildet werden kann, als dies mit einer typischen Versuchsanordnung möglich wäre, zum anderen können durch den Einsatz eines Computers vergleichbare Resultate kostengünstiger erzielt werden. So hat eine Computersimulation im Gegensatz zur klassischen Windkanalsimulation, in der das zu testende Fahrzeug in einen Windkanal gestellt und einem Windstrom ausgesetzt wird, den Vorteil, dass die relative Bewegung des Fahrzeuges zur Fahrbahn in die Simulation mit einbezogen werden kann, d. h. die Computersimulation kann prinzipiell zu realitätsnäheren Ergebnissen führen als die Windkanalsimulation. Crashtests von Autos sind ein offensichtliches Beispiel für ein Einsatzgebiet, in dem Computersimulationen in der Regel kostengünstiger sind als reale Tests.

Alle erwähnten Computersimulationen haben einen sehr hohen Berechnungsaufwand, und die Durchführung der Simulationen kann daher durch eine zu geringe Rechenleistung der verwendeten Computer eingeschränkt werden. Wenn eine hö-

T. Rauber, G. Rünger, *Parallele Programmierung*, eXamen.press,
DOI 10.1007/978-3-642-13604-7_1,

here Rechenleistung zur Verfügung steht, kann diese zum einen zur schnelleren Berechnung einer Aufgabenstellung verwendet werden, zum anderen können aber auch größere Aufgabenstellungen, die zu genaueren Resultaten führen, in ähnlicher Rechenzeit bearbeitet werden. Der Einsatz der Parallelverarbeitung bietet die Möglichkeit, eine wesentlich höhere Rechenleistung zu nutzen, als sie sequentielle Rechner bereitstellen, indem mehrere Prozessoren oder Verarbeitungseinheiten gemeinsam eine Aufgabe bearbeiten. Dabei können speziell für die Parallelverarbeitung entworfene Parallelrechner, aber auch über ein Netzwerk miteinander verbundene Rechner verwendet werden. Seit Einführung von Multicore-Prozessoren stehen auch innerhalb eines Prozessorchips mehrere unabhängige Prozessorkerne zur Verfügung. Darüber hinaus bieten GPUs eine Vielzahl von spezialisierten Prozessorkernen, die für parallele Berechnungen genutzt werden können. Parallelverarbeitung ist jedoch nur möglich, wenn der für eine Simulation abzuarbeitende Algorithmus dafür geeignet ist, d. h. wenn er sich in Teilberechnungen zerlegen lässt, die unabhängig voneinander parallel ausgeführt werden können. Viele Simulationsalgorithmen aus dem wissenschaftlich-technischen Bereich erfüllen diese Voraussetzung.

Für die Nutzung der Parallelverarbeitung ist es notwendig, dass der Algorithmus für eine parallele Abarbeitung vorbereitet wird, indem er in einer parallelen Programmiersprache formuliert oder durch Einsatz von Programmierumgebungen mit zusätzlichen Direktiven oder Anweisungen versehen wird, die die parallele Abarbeitung steuern. Die dabei anzuwendenden Techniken und dafür zur Verfügung stehende Programmierumgebungen werden in diesem Buch vorgestellt. Die Erstellung eines effizienten parallelen Programms verursacht für den Anwendungsprogrammierer je nach Algorithmus z. T. einen recht großen Aufwand, der aber im Erfolgsfall ein Programm ergibt, das auf einer geeigneten Plattform um ein Vielfaches schneller abgearbeitet werden kann als das zugehörige sequentielle Programm. Durch den Einsatz portabler Programmierumgebungen ist das parallele Programm auf einer Vielzahl unterschiedlicher Plattformen ausführbar. Aufgrund dieser Vorteile wird die Parallelverarbeitung in vielen Bereichen erfolgreich eingesetzt.

Ein weiterer Grund, sich mit der parallelen Programmierung zu beschäftigen, besteht darin, dass die Parallelverarbeitung auch für sequentielle Rechner eine zunehmend wichtigere Rolle spielt, da nur durch parallele Technologien eine weitere Leistungssteigerung erreicht werden kann. Dies liegt auch daran, dass die Taktrate von Prozessoren, durch die ihre Verarbeitungsgeschwindigkeit bestimmt wird, nicht beliebig gesteigert werden kann. Die Gründe dafür liegen zum einen in der mit einer Erhöhung der Taktrate verbundenen erhöhten Leistungsaufnahme und Wärmeentwicklung. Zum anderen wirkt die Endlichkeit der Übertragungsgeschwindigkeit der Signale als limitierender Faktor und gewinnt mit zunehmender Taktrate an Einfluss, was mit folgender Beispielrechnung verdeutlicht werden kann: Ein mit einer Taktrate von 3 GHz arbeitender Prozessor hat entsprechend eine Zykluszeit, also eine Dauer eines Taktes, von etwa 0,33 ns. In dieser Zeit kann ein Signal eine Entfernung von $0{,}33 \cdot 10^{-9}\,\mathrm{s} \cdot 0{,}3 \cdot 10^{9}\,\mathrm{m/s} \approx 10\,\mathrm{cm}$ zurücklegen, wobei als Obergrenze der Übertragungsgeschwindigkeit die Lichtgeschwindigkeit im Vakuum ($0{,}3 \cdot 10^{9}\,\mathrm{m/s}$)

angenommen wird. Bei einer Verzehnfachung der Taktrate würde die Zykluszeit entsprechend zehnmal kleiner und die Signale könnten in einem Zyklus also gerade noch 1 cm zurücklegen, womit die Größenordnung der Ausdehnung eines Prozessorchips, die aktuell zwischen 200 und 400 mm^2 liegt, erreicht wäre. Der Transfer von Signalen zwischen zwei beliebigen Positionen auf dem Prozessorchip könnten also nicht innerhalb eines Taktes durchgeführt werden. Die maximal nutzbare Taktrate wird damit von den Signallaufzeiten mitbestimmt.

Die Leistungssteigerung der Prozessoren ist in der Vergangenheit jedoch nicht allein auf eine Steigerung der Taktrate zurückzuführen gewesen, sondern auch durch architektonische Verbesserungen der Prozessoren erzielt worden, die zum großen Teil auf dem Einsatz interner Parallelverarbeitung beruhen. Aber auch diesen architektonischen Verbesserungen sind Grenzen gesetzt, die im Wesentlichen darin begründet sind, dass der Prozessor einen sequentiellen Befehlsstrom bearbeitet, der von einem Übersetzer aus einem sequentiellen Programm erzeugt wird und in der Regel viele Abhängigkeiten zwischen den abzuarbeitenden Instruktionen enthält. Dadurch bleibt der effektive Einsatz parallel arbeitender Funktionseinheiten innerhalb eines Prozessors begrenzt, obwohl die Fortschritte in der VLSI-Technologie eine Integration vieler Funktionseinheiten erlauben würden. Dies führte zur Entwicklung der Multicore-Prozessoren, die mehrere Prozessorkerne (engl. *execution cores*) auf einem Prozessorchip integrieren. Jeder Prozessorkern ist eine unabhängige Verarbeitungseinheit, die von einem separaten Befehlsstrom gesteuert wird. Zur effizienten Ausnutzung der Prozessorkerne eines Multicore-Prozessors ist es daher erforderlich, dass mehrere Berechnungsströme verfügbar sind, die den Prozessorkernen zugeordnet werden können und die beim Zugriff auf gemeinsame Daten auch koordiniert werden müssen. Die Entwicklung hin zu Multicore-Prozessoren ist bei GPUs (*Graphics Processing Unit*) besonders ausgeprägt; eine einzelne GPU kann je nach Ausführung mehrere Hundert Prozessorkerne enthalten.

Zur Bereitstellung eines Berechnungsstroms für jeden Prozessorkern können im Prinzip zwei unterschiedliche Ansätze verfolgt werden. Zum einen kann versucht werden, die Übersetzerbautechniken so zu verbessern, dass der Übersetzer aus einem sequentiellen Befehlsstrom mehrere unabhängige Berechnungsströme erzeugt, die dann gleichzeitig verschiedenen Verarbeitungseinheiten zugeordnet werden. Dieser Ansatz wird seit vielen Jahren verfolgt, die Komplexität der Problemstellung hat aber bisher eine für eine breite Klasse von Anwendungen zufriedenstellende Lösung verhindert. Ein anderer Ansatz besteht darin, dem Übersetzer bereits mehrere Befehlsströme für die Übersetzung zur Verfügung zu stellen, so dass dieser sich auf die eigentliche Übersetzung konzentrieren kann. Dies kann durch Anwendung von Techniken der parallelen Programmierung erreicht werden, indem der Programmierer ein paralleles Programm bereitstellt. Dieser Ansatz ist am vielversprechendsten, bewirkt aber, dass für die effiziente Nutzung typischer Desktop-Rechner mit Multicore-Prozessoren Programmiertechniken der Parallelverarbeitung eingesetzt werden müssen.

Das vorliegende Buches soll dem Leser die wichtigsten Programmiertechniken für alle Einsatzgebiete der parallelen Programmierung vermitteln. Bevor wir einen detaillierten Überblick über den Inhalt dieses Buches geben, möchten wir im folgenden Abschnitt grundlegende Begriffe der Parallelverarbeitung einführen. Diese werden dann in den späteren Kapiteln näher präzisiert.

1.2 Begriffe der Parallelverarbeitung

Eines der wichtigsten Ziele der Parallelverarbeitung ist es, Aufgaben in einer kürzeren Ausführungszeit zu erledigen, als dies durch eine Ausführung auf sequentiellen Rechnerplattformen möglich wäre. Die durch den Einsatz paralleler Rechnertechnologie erhaltene erhöhte Rechenleistung wird häufig auch dazu genutzt, komplexere Aufgabenstellungen zu bearbeiten, die zu besseren oder genaueren Lösungen führen, als sie durch den Einsatz einer sequentiellen Rechnerplattform in vertretbarer Zeit möglich wären. Andere Ziele der Parallelverarbeitung sind das Erreichen von Ausfallsicherheit durch Replikation von Berechnungen oder die Erfüllung größerer Speicheranforderungen.

Die Grundidee zur Erreichung einer kürzeren Ausführungszeit besteht darin, die Ausführungszeit eines Programms dadurch zu reduzieren, dass mehrere Berechnungsströme erzeugt werden, die gleichzeitig, also parallel, ausgeführt werden können und durch koordinierte Zusammenarbeit die gewünschte Aufgabe erledigen. Zur Erzeugung der Berechnungsströme wird die auszuführende Aufgabe in *Teilaufgaben* zerlegt. Zur Benennung solcher Teilaufgaben haben sich die Begriffe *Prozesse*, *Threads* oder *Tasks* herausgebildet, die für unterschiedliche Programmiermodelle und -umgebungen jedoch geringfügig unterschiedliche Bedeutungen haben können. Zur tatsächlichen parallelen Abarbeitung werden die Teilaufgaben auf physikalische Berechnungseinheiten abgebildet, was auch als *Mapping* bezeichnet wird. Dies kann statisch zur Übersetzungszeit oder dynamisch zur Laufzeit des Programms stattfinden. Zur Vereinfachung der Darstellung werden wir im Folgenden unabhängige physikalische Berechnungseinheiten als Prozessoren bezeichnen und meinen damit sowohl Prozessoren als auch Prozessorkerne eines Multicore-Prozessors.

Typischerweise sind die erzeugten Teilaufgaben nicht vollkommen unabhängig voneinander, sondern können durch *Daten- und Kontrollabhängigkeiten* gekoppelt sein, so dass bestimmte Teilaufgaben nicht ausgeführt werden können, bevor andere Teilaufgaben benötigte Daten oder Informationen für den nachfolgenden Kontrollfluss bereitgestellt haben. Eine der wichtigsten Aufgaben der parallelen Programmierung ist es, eine korrekte Abarbeitung der parallelen Teilaufgaben durch geeignete *Synchronisation* und notwendigen *Informationsaustausch* zwischen den Berechnungsströmen sicherzustellen. Parallele Programmiermodelle und -umgebungen stellen hierzu eine Vielzahl unterschiedlicher Methoden und Mechanismen zur Verfügung. Eine grobe Unterteilung solcher Mechanismen

kann anhand der Unterscheidung in Programmiermodelle mit *gemeinsamem* oder *verteiltem Adressraum* erfolgen, die sich eng an die Hardwaretechnologie der Speicherorganisation anlehnt.

Bei einem *gemeinsamen Adressraum* wird dieser über gemeinsam zugreifbare Variablen zum Informationsaustausch genutzt. Diese einfache Art des Informationsaustausches wird durch vielfältige Mechanismen zur Synchronisation der meist als Threads bezeichneten Berechnungsströme ergänzt, die den konkurrierenden Datenzugriff durch mehrere Threads koordinieren.

Bei einem *verteilten Adressraum* sind die Daten eines parallelen Programms in privaten Adressbereichen abgelegt, auf die nur der entsprechende, meist als Prozess bezeichnete Berechnungsstrom Zugriff hat. Ein Informationsaustausch kann durch explizite Kommunikationsanweisungen erfolgen, mit denen ein Prozess Daten seines privaten Adressbereichs an einen anderen Prozess senden kann. Zur Koordination der parallelen Berechnungsströme eignet sich eine Synchronisation in Form einer *Barrier-Synchronisation*. Diese bewirkt, dass alle beteiligten Prozesse aufeinander warten und kein Prozess eine nach der Synchronisation stehende Anweisung ausführt, bevor nicht die restlichen Prozesse den Synchronisationspunkt erreicht haben.

Zur Bewertung der Ausführungszeit paralleler Programme werden verschiedene Kostenmaße verwendet. Die *parallele Laufzeit* eines Programms setzt sich aus der Rechenzeit der einzelnen Berechnungsströme und der Zeit für den erforderlichen Informationsaustausch oder benötigte Synchronisationen zusammen. Zur Erreichung einer geringen parallelen Laufzeit sollte eine möglichst gleichmäßige Verteilung der Rechenlast auf die Prozessoren angestrebt werden (*Load balancing*), so dass ein *Lastgleichgewicht* entsteht. Ein Vermeiden langer Wartezeiten und möglichst wenig Informationsaustausch sind insbesondere für parallele Programme mit verteiltem Adressraum wichtig zur Erzielung einer geringen parallelen Laufzeit. Für einen gemeinsamen Adressraum sollten entsprechend Wartezeiten an Synchronisationspunkten minimiert werden.

Die Zuordnung von Teilaufgaben an Prozessoren sollte so gestaltet werden, dass Teilaufgaben, die häufig Informationen austauschen müssen, dem gleichen Prozessor zugeordnet werden. Ein gleichzeitiges Erreichen eines optimalen Lastgleichgewichts und eine Minimierung des Informationsaustausches ist oft schwierig, da eine Reduzierung des Informationsaustausches zu einem Lastungleichgewicht führen kann, während eine gleichmäßige Verteilung der Arbeit größere Abhängigkeiten zwischen den Berechnungsströmen und damit mehr Informationsaustausch verursachen kann. Zur Bewertung der resultierenden Berechnungszeit eines parallelen Programms im Verhältnis zur Berechnungszeit eines entsprechenden sequentiellen Programms werden Kostenmaße wie *Speedup* und *Effizienz* verwendet.

Das Erreichen einer Gleichverteilung der Last hängt eng mit der Zerlegung in Teilaufgaben zusammen. Die Zerlegung legt den Grad der Parallelität sowie die *Granularität*, d. h. die durchschnittliche Größe der Teilaufgaben (z. B. gemessen als Anzahl der Instruktionen) fest. Um eine hohe Flexibilität bei der Zuteilung von Teilaufgaben an Prozessoren sicherzustellen und eine gleichmäßige Lastverteilung

zu ermöglichen, ist ein möglichst hoher Grad an Parallelität günstig. Zur Reduktion des Verwaltungsaufwandes für die Abarbeitung der Teilaufgaben durch die einzelnen Prozessoren ist es dagegen erstrebenswert, mit möglichst wenigen Teilaufgaben entsprechend grober Granularität zu arbeiten, d. h. auch hier muss ein Kompromiss zwischen entgegengesetzten Zielstellungen gefunden werden. Die Abarbeitung der erzeugten Teilaufgaben unterliegt den geschilderten Einschränkungen, die die erreichbare Anzahl von parallel, also gleichzeitig auf verschiedenen Prozessoren ausführbaren Teilaufgaben bestimmen. In diesem Zusammenhang spricht man auch vom erreichbaren *Grad der Parallelität* oder dem *potentiellen Parallelitätsgrad* einer Anwendung. Der Entscheidungsvorgang, in welcher Reihenfolge die Teilaufgaben (unter Berücksichtigung der Abhängigkeiten) parallel abgearbeitet werden, wird *Scheduling* genannt. Es werden statische, d. h. zur Übersetzungszeit arbeitende, oder dynamische, d. h. während des Programmlaufes arbeitende, Schedulingalgorithmen auf verschiedenen Parallelisierungsebenen genutzt. Schedulingverfahren und -algorithmen werden in der Parallelverarbeitung in sehr unterschiedlicher Form benötigt. Dies reicht von Thread-Scheduling in vielen Modellen des gemeinsamen Adressraums bis zum Scheduling von Teilaufgaben mit Abhängigkeiten auf Programmebene in der Programmierung für verteilten Adressraum.

Die Granularität und Anzahl der Teilaufgaben wird also wesentlich von den für die betrachtete Anwendung durchzuführenden Berechnungen und den Abhängigkeiten zwischen diesen Berechnungen bestimmt. Die genaue Abbildung der Teilaufgaben auf die Prozessoren hängt zusätzlich von der Architektur des verwendeten Parallelrechners und von der verwendeten Programmiersprache oder Programmierumgebung ab. Dieses Zusammenspiel der parallelen Eigenschaften des zu bearbeitetenden Anwendungsproblems, der Architektur des Parallelrechners und der Programmierumgebung ist grundlegend für die parallele Programmierung. Wir werden dem Rechnung tragen, indem wir in den einzelnen Kapiteln zunächst auf die unterschiedlichen Typen von Parallelrechnern und parallelen Plattformen eingehen, einen Überblick über parallele Programmierumgebungen geben und abschließend Charakteristika wichtiger Anwendungsalgorithmen aus dem Bereich des wissenschaftlichen Rechnens behandeln. Wir stellen die Inhalte der einzelnen Kapitel nun noch etwas genauer vor.

1.3 Überblick über den Inhalt des Buches

Das nächste Kap. 2 gibt einen Überblick über die Architektur paralleler Plattformen und behandelt deren Ausprägungen hinsichtlich der Kontrollmechanismen, der Speicherorganisation und des Verbindungsnetzwerkes. Bei der Speicherorganisation wird im Wesentlichen unterschieden zwischen Rechnern mit verteiltem Speicher, bei denen der Speicher in Form lokaler Speicher für die einzelnen Prozessoren vorliegt, und Rechnern mit gemeinsamem Speicher, bei denen alle Prozessoren den gleichen globalen Speicher gemeinsam nutzen. Diese Unterschei-

dung ist die Grundlage für die später vorgestellten Programmierumgebungen. Zu Plattformen mit gemeinsamem Speicher gehören auch Desktop-Rechner, die mit Multicore-Prozessoren ausgestattet sind. Clustersysteme, die aus mehreren Multicore-Prozessoren bestehen, deren Prozessorkerne jeweils auf einen gemeinsamen Speicher zugreifen, während Prozessorkerne unterschiedlicher Prozessoren Informationen und Daten über ein Verbindungsnetzwerk austauschen müssen, sind Hybridmodelle, die sich durch Speicherhierarchien und Caches verschiedener Stufen auszeichnen. In Kap. 2 werden die entsprechenden Eigenschaften von Verbindungsnetzwerken, Routing- und Switching-Strategien sowie Caches und Speicherhierarchien mit den zugehörigen Kohärenz- und Konsistenzmodellen behandelt. Diese Abschnitte können bei einer stärkeren Konzentration auf die parallele Programmierung übersprungen werden, ohne dass das Verständnis der späteren Kapitel beeinträchtigt wäre. Verbindungsnetzwerke und deren Routing- und Switchingstrategien sind ein erster Ansatzpunkt für Kostenmodelle für den Rechenzeitbedarf paralleler Programme, an den das spätere Kapitel zu Kostenmodellen anknüpft.

Die Kap. 3 und 4 befassen sich mit verschiedenen parallelen Programmiermodellen. Das Kap. 3 stellt parallele Programmiermodelle und -paradigmen vor und beschreibt die auf verschiedenen Programmebenen verfügbare Parallelität sowie die Möglichkeiten ihrer Ausnutzung in parallelen Programmierumgebungen. Insbesondere werden die für einen gemeinsamen oder verteilten Adressraum benötigten Koordinations-, Synchronisations- und Kommunikationsoperationen allgemein vorgestellt. Kapitel 4 führt grundlegende Definitionen zur Bewertung paralleler Programme ein und beschreibt, wie Kostenmodelle dazu verwendet werden können, einen quantitative Abschätzung der Laufzeit paralleler Programme für einen gegebenen Parallelrechner zu erhalten. Dadurch ist die Grundlage für eine statische Planung der Abarbeitung paralleler Programme gegeben.

Die Kap. 5 bis 7 stellen parallele Programmierumgebungen für Parallelrechner mit verteiltem Adressraum, gemeinsamem Adressraum und für GPUs vor. In Kap. 5 werden portable Programmierumgebungen für einen verteilten Adressraum beschrieben, die oft in Form von Message-Passing-Bibliotheken für Plattformen mit verteiltem Speicher eingesetzt werden. Für die derzeit als De-facto-Standard anzusehende MPI-Bibliothek (Message Passing Interface) werden die zur Verfügung gestellten Funktionen detailliert vorgestellt. Die zum Entwurf paralleler MPI-Programme notwendigen Techniken werden an Beispielprogrammen besprochen. Kapitel 6 beschreibt Programmierumgebungen für einen gemeinsamen Adressraum und gibt einen Überblick über die Pthreads-Bibliothek, die von vielen UNIX-ähnlichen Betriebssystemen unterstützt wird, und über OpenMP, das als Standard vor allem für Programme des wissenschaftlichen Rechnens genutzt wird. Dieses Kapitel geht auch auf sprachbasierte Threadansätze wie Java-Threads oder Unified Parallel C (UPC) ein. Kapitel 7 enthält eine Einführung in die Programmierung von GPUs für die Durchführung von Berechnungen nicht-graphikorientierter Programme z. B. aus dem wissenschaftlich-technischen Bereich. Dabei gehen wir insbesondere auf das CUDA-System (Compute Unified Device Architecture) von

NVIDIA für die Programmierung von NVIDIA GPUs ein und geben einen Ausblick auf die herstellerunabhängige Programmbibliothek OpenCL.

In Kap. 8 werden größere Anwendungsbeispiele behandelt. Diese stammen aus dem Bereich der direkten und iterativen Verfahren zur Lösung linearer Gleichungssysteme. Kapitel 8 beschreibt die algorithmischen Eigenschaften für jedes Verfahren und gibt mehrere Möglichkeiten einer parallelen Implementierung insbesondere für einen verteilten Adressraum an. Um dem Leser die Erstellung der zugehörigen parallelen Programme zu erleichtern, geben wir Programmfragmente an, die die relevanten Details zur Steuerung der parallelen Abarbeitung enthalten und die relativ einfach zu kompletten Programmen ausgebaut werden können. Für parallele Plattformen mit verteiltem Adressraum werden die erforderlichen Kommunikationsoperationen in MPI ausgedrückt.

Die Kapitel dieses Buches sind im Wesentlichen jeweils in sich selbst abgeschlossen, so dass sie genutzt werden können, um Vorlesungen im Bereich der Parallelverarbeitung mit verschiedenen Schwerpunkten modular aufzubauen. Ein Basiskurs zur Parallelen Programmierung sollte Teile des Kap. 2, insbesondere Abschn. 2.1 bis 2.6, Kap. 3, Abschn. 4.2, 4.3 und 4.5 sowie je nach Schwerpunkt Kap. 5 zu MPI und Kap. 6 etwa mit Pthreads oder OpenMP enthalten.

Eine spezielle Vorlesung zur Multicore-Programmierung könnte aus Abschn. 2.1 bis 2.6 sowie Abschn. 2.7 zu Caches und Abschn. 2.8 zur Architektur von Multicore-Prozessoren aufgebaut sein und danach Kap. 3 und Kap. 4 sowie Kap. 6 enthalten. Für eine Vorlesung über paralleles wissenschaftliches Rechnen wäre eine Kurzfassung aus Abschn. 2.1 bis 2.4, Kap. 3 mit Schwerpunkt auf Abschn. 3.7, Abschn. 4.2 und 4.4, Kap. 5 und Kap. 8 zu empfehlen. Das Kap. 7 zur Programmierung von GPUs könnte je nach Ausrichtung in alle oben genannten Vorlesungen aufgenommen werden.

Unter ai2.inf.uni-bayreuth.de/pp_buch_3A ist eine Webseite mit begleitendem Material zu diesem Buch eingerichtet. Dort werden u. a. weitere Materialien zum Inhalt des Buches sowie Informationen zu neueren Entwicklungen zur Verfügung gestellt.

Kapitel 2
Architektur paralleler Plattformen

Wie in der Einleitung bereits angerissen wurde, hängen die Möglichkeiten einer parallelen Abarbeitung stark von den Gegebenheiten der benutzten Hardware ab. Wir wollen in diesem Kapitel daher den prinzipiellen Aufbau paralleler Plattformen vorstellen, auf die die auf Programmebene gegebene Parallelität abgebildet werden kann, um eine tatsächlich gleichzeitige Abarbeitung verschiedener Programmteile zu erreichen. In den Abschn. 2.1 und 2.2 beginnen wir mit einer kurzen Darstellung der innerhalb eines Prozessors oder Prozessorkerns zur Verfügung stehenden Möglichkeiten einer parallelen Verarbeitung. Hierbei wird deutlich, dass schon bei einzelnen Prozessorkernen eine Ausnutzung der verfügbaren Parallelität (auf Instruktionsebene) zu einer erheblichen Leistungssteigerung führen kann. Die weiteren Abschnitte des Kapitels sind Hardwarekomponenten von Parallelrechnern gewidmet. In den Abschn. 2.3 und 2.4 gehen wir auf die Kontroll- und Speicherorganisation paralleler Plattformen ein, indem wir zum einen die Flynnsche Klassifikation einführen und zum anderen Rechner mit verteiltem und Rechner mit gemeinsamem Speicher einander gegenüberstellen.

Eine weitere wichtige Komponente paralleler Hardware sind Verbindungsnetzwerke, die Prozessoren und Speicher bzw. verschiedene Prozessoren physikalisch miteinander verbinden. Verbindungsnetzwerke spielen auch bei Multicore-Prozessoren eine große Rolle, und zwar zur Verbindung der Prozessorkerne untereinander sowie mit den Caches des Prozessorchips. Statische und dynamische Verbindungsnetzwerke und deren Bewertung anhand verschiedener Kriterien wie Durchmesser, Bisektionsbandbreite, Konnektivität und Einbettbarkeit anderer Netzwerke werden in Abschn. 2.5 eingeführt. Zum Verschicken von Daten zwischen zwei Prozessoren wird das Verbindungsnetzwerk genutzt, wozu meist mehrere Pfade im Verbindungsnetzwerk zur Verfügung stehen. In Abschn. 2.6 beschreiben wir Routingtechniken zur Auswahl eines solchen Pfades durch das Netzwerk sowie Switchingverfahren, die die Übertragung der Nachricht über einen vorgegebenen Pfad regeln. In Abschn. 2.7 werden Speicherhierarchien sequentieller und paralleler Plattformen betrachtet. Wir gehen insbesondere auf die bei parallelen Plattformen auftretenden Cachekohärenz- und Speicherkonsistenzprobleme ein. In

T. Rauber, G. Rünger, *Parallele Programmierung*, eXamen.press,
DOI 10.1007/978-3-642-13604-7_2,

Abschn. 2.8 werden Prozessortechnologien wie simultanes Multithreading oder Multicore-Prozessoren zur Realisierung prozessorinterner Parallelverarbeitung auf Thread- oder Prozessebene vorgestellt. Abschließend enthält Abschn. 2.9 als Beispiel für die Architektur eines aktuellen Parallelrechners eine kurze Beschreibung der Architektur der IBM Blue Gene/Q Systeme.

2.1 Überblick über die Prozessorentwicklung

Bei der Prozessorentwicklung sind bestimmte Trends zu beobachten, die die Basis für Prognosen über die weitere voraussichtliche Entwicklung bilden. Ein wesentlicher Punkt ist die Performance-Entwicklung der Prozessoren, die von verschiedenen technologischen Faktoren beeinflusst wird. Ein wichtiger Faktor ist die Taktrate der Prozessoren, die die Zykluszeit des Prozessors und damit die Zeit für das Ausführen von Instruktionen bestimmt. Es ist zu beobachten, dass die Taktrate typischer Mikroprozessoren, wie sie z. B. in Desktop-Rechnern eingesetzt werden, zwischen 1987 und 2003 durchschnittlich um ca. 40 % pro Jahr gestiegen ist [76]. Seit 2003 ist die Taktrate dann ungefähr gleich geblieben und es sind in der nahen Zukunft auch keine signifikanten Steigerungen zu erwarten [73, 106]. Der Grund für diese Entwicklung liegt darin, dass mit einer Steigerung der Taktrate auch ein erhöhter Stromverbrauch einhergeht, der aufgrund von Leckströmen vor allem zu einer Erhöhung der Wärmeentwicklung führt, die wiederum einen erhöhten Aufwand für die Prozessorkühlung erforderlich macht. Mit der derzeitigen Luftkühlungstechnologie können jedoch ohne einen sehr großen Aufwand aktuell nur Prozessoren gekühlt werden, deren Taktrate ca. 3,3 GHz nicht wesentlich übersteigt.

Ein weiterer Einflussfaktor für die Performance-Entwicklung der Prozessoren ist die Anzahl der Transistoren eines Prozessorchips, die ein ungefähres Maß für die Komplexität des Schaltkreises ist und die pro Jahr um etwa 60 % bis 80 % wächst. Dadurch wird ständig mehr Platz für Register, Caches und Funktionseinheiten zur Verfügung gestellt. Diese von der Prozessorfertigungstechnik getragene, seit über 40 Jahren gültige empirische Beobachtung wird auch als Gesetz von Moore (engl. *Moore's law*) bezeichnet. Ein typischer Prozessor aus dem Jahr 2012 besteht aus ca. 1 bis 3 Milliarden Transistoren. Beispielsweise enthält ein Intel Core i7 Sandy Bridge Quadcore Prozessor ca. 995 Millionen Transistoren, ein Intel Core i7 Ivy Bridge-HE-4 Quadcore Prozessor ca. 1,4 Milliarden Transistoren und ein Intel Xeon Westmere-EX 10-Core Prozessor ca. 2,6 Milliarden Transistoren.

Zur Leistungsbewertung von Prozessoren können Benchmarks verwendet werden, die meist eine Sammlung von Programmen aus verschiedenen Anwendungsbereichen sind und deren Ausführung repräsentativ für die Nutzung eines Rechnersystems sein soll. Häufig verwendet werden die SPEC-Benchmarks (*System Performance and Evaluation Cooperative*), die zur Messung der Integer- bzw. Floating-Point-Performance eines Rechners dienen [83, 139, 170], vgl. auch