Math. Semesterber. (1993) 40:93–104

Buchbesprechungen

N. Carriero, D. Gelernter: How to Write Parallel Programs – A First Course. MIT Press, Cambridge MA, London 1990. 232 S., DM 89,–.

Die Fortschritte in der Halbleitertechnologie haben in den letzten Jahrzehnten mit der Miniaturisierung logischer Schaltkreise die Rechnerleistungen um Größenordnungen gesteigert. Da hier mit der rein sequentiellen Verarbeitung bereits physikalische Grenzen erreicht wurden, stieg das Interesse in die Parallelverarbeitung zur weiteren Leistungssteigerung. Die Industrie ist heute in der Lage, zu recht guten Preis/Leistungs-Verhältnissen leistungsfähige Parallelrechner zur Verfügung zu stellen. Auch vernetzte Arbeitsplatzrechner bieten sich schon zur Parallelverarbeitung an. Das Problem besteht dabei nicht mehr darin, Parallelrechner zu bauen, sondern diese zu programmieren. Die Möglichkeiten zur automatischen Parallelisierung sequentieller Programme sind nur sehr begrenzt vorhanden, so daß zur effizienten Nutzung der vorhandenen Parallelrechner nur die explizite Parallelprogrammierung in Frage kommt. Auch ist für viele Anwendungen die Erstellung eines parallelen Programmes natürlicher als die Erstellung eines sequentiellen Programmes. Genau in diesen Bereich – also *wie* parallele Programme geschrieben werden können – zielt das vorliegende Buch. Dabei beschränkt es sich auf das MIMD-Modell (multiple instruction stream – multiple data stream) mit asynchroner Kommunikation entsprechend Flynn's Taxonomie für parallele Rechnerarchitekturen.

Nach Entfernen des blauen Schutzumschlags hält man ein mit schlichtem schwarzen Einband ausgestattetes Buch in der Hand. Mit seinen gut zweihundert Seiten wirkt es recht schlank und verspricht eine leicht verdauliche Lektüre zu werden, da nach einem ersten Durchblättern keinerlei Formalismen zu entdecken sind.

In der Einführung wird zunächst das Was, Warum und Wie der Parallelverarbeitung diskutiert. Parallelverarbeitung ist nur ein Beipsiel für Koordination. Koordinations-Theorie ist nicht beschränkt auf Software-Komponenten. Sie beschäftigt sich z.B. auch mit Mensch/Maschine-Koordination (sogenannte Turingware) bis hin zu Mensch/Mensch-Koordination (Soziologie). In einem philosophischen Ausflug gehen die Autoren hier soweit, daß für diese unterschiedlichen Koordinations-Ebenen gleichartige Koordinations-Medien gefunden werden können/sollten.

Eine parallele Programmiersprache besteht dann grob gesehen aus einer *Berechnungs-Sprache* (computation language) und einer *Koordinierungs-Sprache* (coordination language). In der Koordinations-Sprache wird die Koordinierung sequentieller Anweisungen, die in der Berechnungs-Sprache programmiert werden, programmiert.

Wie werden nun parallele Programme geschrieben? Als zentraler Rahmen werden dazu drei Basis-Modelle zum Erstellen paralleler Programme vorgestellt. Diese sind *Resultat-Parallelität* (result parallelism), *Aufgaben-Parallelität* (agenda parallelism) sowie *Spezialisten-Parallelität* (specialist parallelism). Resultat-Parallelität orientiert sich an der Form des gewünschten Resultats: das Programm wird entsprechend der Struktur des Ergebnisses geplant. Aufgaben-Parallelität orientiert sich an einer Liste zu erledigender Aufgaben: das Programm wird entsprechend der Struktur der Aufgaben geplant. Spezialisten-Parallelität orientiert sich an einer Gruppe von Prozessen, die jeweils spezielle Probleme lösen sollen: das Programm wird entsprechend der Struktur eines logischen Netzwerks, das diese Spezialisten untereinander verbindet, geplant. Zu diesem Programmier-Modellen gibt es jeweils passende Programmier-Methoden, mit deren Hilfe geeignete Programme erstellt werden können:

Resultat-Parallelität	↔	Aktive Datenstrukturen
Aufgaben-Parallelität	↔	Verteilte Datenstrukturen
Spezialisten-Parallelität	⇢	Nachrichten-Austausch

In aktiven Datenstrukturen (live data structures) sind die nebenläufigen Prozesse in Datenobjekte eingeschlossen: jeder Prozeß berechnet genau ein Datum. Bei verteilten Datenstrukturen (distributed data structures) sind Prozesse und Datenobjekte autonome Teile der Programmstruktur: Kommunikation erfolgt über gemeinsame Datenobjekte. Beim Nachrichten-Austausch (message passing) sind die Datenobjekte in Prozessen eingeschlossen: Kommunikation erfolgt durch Austausch von Nachrichten.

Um ein paralleles Programm zu schreiben, muß zunächst ein passendes Modell ausgewählt werden, das gut zum zu lösenden Problem paßt. Dann wird mit der entsprechenden Methode ein Programm geschrieben. Falls das resultierende Programm bzgl. der Laufzeit-Effizienz nicht den Ansprüchen genügt, kann es methodisch in ein effizienteres Programm transformiert werden. Dabei wird dann von einer eher natürlichen Lösungs-Methode für das Problem zu einer effizienteren Methode übergegangen. Solche Transformationen werden sich kaum vollautomatisch in imperativen Programmiersprachen durchführen lassen. Um optimale Effizienz zu erreichen, ist das Talent der Programmierer gefordert, das durch geeignete Werkzeuge unterstützt werden sollte.

Zunächst werden die einzelnen Modelle recht anschaulich anhand eines Hausbaus beschrieben, was dem Leser einen ersten Zugang zu den Methoden bietet. Dann werden die Transformationen in einer DNA-Datenbank, zu Matrix-Manipulationen sowie zu einem Prozeß-Netzwerk, das Daten von Sensoren in Realzeit verarbeitet, vorgeführt. Das letzte Kapitel widmet sich dann einigen *klassischen* Problemen der Parallelprogrammierung – z.B. dem Problem der dinierenden Philosophen. Zu allen diskutierten Problemen werden Lösungen in C-Linda angegeben. C-Linda ist eine Kombination aus der Berechnungs-Sprache C und der Koordinations-Sprache Linda, die von den Autoren an der Yale Universität entwickelt wurde. Kommunikation unter Prozessen in C-Linda erfolgt über das Einfügen, Entfernen und Lesen von Tupeln in einem globalen, gemeinsamen Tupel-Raum (tuple space). Sie wird deshalb auch *generative Kommunikation* genannt. Ein Tupel ist eine endliche, geordnete Folge von Werten. Die möglichen Typen dieser Werte werden durch die Berechnungs-Sprache bestimmt. Der Zugriff auf die Tupel erfolgt assoziativ.

Zur Lösung der Probleme stellt sich das Finden einer geeigneten Granularität (also das Verhältnis von Kommunikation zu Berechnung) zum Erreichen effizienter Lösungen als wesentliche Aufgabe dar. Die Aufgaben-Parallelität erweist sich als das überlegene Modell, das dann jeweils das Ende bei den Transformationen markiert.

Das Meister/Arbeiter-Modell (master/worker model) ist ein Beispiel für Aufgaben-Parallelität, das eine sehr einfache Last-Verteilung unter aktuell verfügbaren Prozessoren erlaubt. In einem Meister/Arbeiter-Programm initialisiert ein Meister-Prozeß die Berechnung, indem Aufgaben im Tupel-Raum abgelegt werden und eine Menge von identischen Arbeiter-Prozessen gestartet wird. Jeder Arbeiter holt sich in einer Schleife Aufgaben, bearbeitet diese und legt die Ergebnisse wieder im Tupel-Raum ab. Der Meister kann die Ergebnisse dann einsammeln und das Programm beenden. Ein paralleles Programm endet, sobald *alle* Prozesse, die zu diesem Programm gehören, terminieren. Ein Meister/Arbeiter-Programm arbeitet unabhängig von der zur Verfügung stehenden Anzahl Arbeiter (mindestens jedoch einer), so daß diese Anzahl leicht an die zur Verfügung stehende Anzahl Prozessoren angepaßt werden kann. Aufgaben-Parallelität und insbesondere das Meister/Arbeiter-Modell ist sehr leicht in Linda programmierbar.

Da der Grund für den Schritt von der rein sequentiellen zur parallelen Programmierung meistens eine Erhöhung der Laufzeit-Effizienz der Programme ist, wird auch ausführlich auf die Leistungs-Eigenschaften und -Messung sowie auf die dabei entstehenden Probleme eingegangen. Aber parallele Programmierung ist natürlich nicht nur zur Effizienz-Steigerung interessant. Die meisten realen Systeme haben eine inherent parallele Natur, so daß es nur natürlich ist, deren Modelle auch parallel zu programmieren. Es wäre unnatürlich, solche Programme in sequentielle Abläufe zu zwingen, die nicht inhärent vorhanden sind.

Glücklicherweise wurden in das offensichtlich mit LaTeX erstellt Buch auch einige Graphiken aufgenommen, die den Text leichter verständlich machen. Da LaTeX das Erstellen von Graphiken nicht besonders gut unterstützt, wurden diese durch andere Programme erzeugt. Dies hat leider zur Folge, daß die Graphiken stilistisch nicht zum Text passen und durch die breiten Linien und schraffierten Flächen häufig wie Fremdkörper wirken. Das ist wahrscheinlich ein generelles Problem für LaTeX-Dokumente. Das Buch enthält erfreulich wenige Tippfehler. Ein Text, der sich mit konkreten Programm-Beispielen beschäftigt, hat natürlich gegenüber formalen, theoretischen Texten den Vorteil, daß die präsentierten Beispiele getestet werden können (dagegen können formale Arbeiten *bewiesen* werden).

Die Beispiele werden in einzelnen Schritten durch Verfeinerung erarbeitet, was dem Leser einen anschaulichen Zugang zu den vorgestellten Lösungen bietet, die dann jeweils abschließend als vollständige C-Linda-Programme angegeben werden. Zu den ausführlich beschriebenen Übungen sind keine Lösungen angegeben, so daß sich dieses Lehrbuch gut als Grundlage zu einer Vorlesung im Hauptstudium eignen dürfte, in der der Stoff durch praktische Übungen vertieft werden soll. Kenntnisse in Programmiersprachen

wie Pascal oder C sind notwendig. Weiterhin sollte der Ratschlag der Autoren beachtet werden, zusätzlich zu diesem Buch auch Literatur zu parallelen Architekturen und Algorithmen, sowie anderen parallelen Programmiersprachen und Systemen, für eine Lehrveranstaltung zur Parallelverarbeitung zu verwenden. Die Übungsaufgaben werden häufig *objekt-orientiert* präsentiert. Bei der Implementation in C-Linda dürfte sich dann das Fehlen von Mechanismen zur Informations-Einkapselung im Tupel-Raum als Nachteil erweisen. Linda-Kombinationen mit Berechnungs-Sprachen von höherem Niveau als C werden solche Probleme sicherlich beseitigen (sofern sie denn verfügbar sind).

Lesern, die Einsichten in die praktischen Probleme und in Lösungsmöglichkeiten der Programmierung paralleler Anwendungen erwerben wollen, wird dieses Buch einen guten Startpunkt liefern.

Wilhelm Hasselbring, Universität GH Essen, Fachbereich Mathematik und Informatik, Schützenbahn 70, W-4300 Essen 1