Think ~~pink~~ parallel

Grundbegriffe der Parallelität

Parallele Datenverarbeitung
Parallele Datenverarbeitung ist Datenverarbeitung, die auf gleichzeitige Berechnung von Ergebnissen auf mehreren Prozessoren/Rechnern (im Folgenden als Einheit bezeichnet) abzielt, die zur Lösung eines Problems erforderlich sind. Als Einheit kann sowohl ein Prozessor als auch ein Rechner bezeichnet werden, das Prinzip der Parallelisierung ist übertragbar.

Grad der Parallelität
Der Grad der Parallelität einer parallelen Berechnung ist die Anzahl der Prozessoren, die an der Berechnung beteiligt sind.

Durchsatz
Durchsatz quantifiziert die Menge von Problemen, die in einer Zeiteinheit von einem Rechner gelöst werden kann.

Latenz
Latenz ist die Ausführungszeit der Berechnung eines speziellen Problems.

Speedup (engl. für *Beschleunigung*) ist ein Begriff aus der Informatik und beschreibt mathematisch den Zusammenhang zwischen der Abarbeitungsgeschwindigkeit auf einer Einheit im Verhältnis zu mehreren Einheiten. Gegeben sei ein Problem, das von einer Einheit P nicht schneller als in der Zeit T gelöst werden kann.
n Einheiten vom Typ P kann dasselbe Problem nicht schneller als in der Zeit T/n lösen, mit anderen Worten:
Bei n Prozessoren ist der Speedup höchstens n.

Pipelining
Pipelining ist eine Form von paralleler Datenverarbeitung, bei der die Daten auf einem einfach gerichteten Datenpfad während der Berechnung von einem

Prozessor/Rechner zum nächsten weitergereicht werden.

Skalierbarkeit
Ein Algorithmus ist skalierbar, wenn sein Parallelitätsgrad mindestens linear mit der Problemgröße wächst.

Modularität
Ein Algorithmus ist modular, wenn er bei wachsender Problemgröße durch den wiederholten oder parallelen Einsatz desselben Algorithmus für die originale Problemgröße implementiert werden kann.

Multitasking/Multithreading
Die Nebenläufigkeit von mehreren unabhängigen Prozessen bezeichnet man als Multitasking; Nebenläufigkeit innerhalb eines Prozesses als Multithreading.

Kosten einer parallelen Berechnung
Die Kosten einer parallelen Berechnung ist das Produkt aus Parallelitätsgrad und Latenz der Berechnung.
Die Kosten eines parallelen Algorithmus für ein gegebenes Problem können nie kleiner sein als die des besten sequentiellen Algorithmus. Cpar >=Cseq

Optimaler paralleler Algorithmus
Ein Algorithmus mit dem Parallelitätsgrad p heißt optimal,
wenn für seine Kosten gilt: Cpar <= k *Cseq
wobei k eine Konstante ist, die nicht von p oder von der Problemgröße n abhängt.

Amdahls Gesetz
Wenn f der Anteil einer Berechnung ist, der nicht parallelisiert werden kann, dann gilt für den Speedup, der durch Verwendung von p Einheiten erreicht werden kann

$$S \leq \frac{1}{f + (1 - f) / p}$$

Warum macht Parallelverarbeitung Sinn?

In den vergangenen Jahrzehnten konnte dieser Hunger nach Performance durch beeindruckende Verbesserungen im Bereich der Rechnerarchitektur gestillt werden: Etwa alle 5 Jahre wuchs die Anzahl der Instruktionen pro Zeiteinheit etwa um den Faktor 10. Aufgrund von physikalischen Randbedingungen (wie z.B. der Lichtgeschwindigkeit) lässt sich die Taktfrequenz eines Mikroprozessors nicht mehr beliebig steigern. Dieser schlechten Nachricht stehen aber zwei gute gegenüber: Aufgrund von Massenfabrikation und standardisierten Herstellungsverfahren werden Hardwarekomponenten zunehmend billiger und kleiner.
Was liegt also näher, als viele leistungsfähige Einheiten zusammenzuschließen und sie alle gemeinsam an einer Aufgabe arbeiten zu lassen?

Aufgrund des stetig wachsenden Datenvolumen und der Ausreizung der technischen

Gegebenheiten bekommt der Aspekt einer geschickten Verarbeitungslogik künftig (wieder) eine größere Bedeutung. Heute verteilt man die Verarbeitung auf mehrere Rechner (RAC). Dies bildet die Grundlage für Grid Computing.

Die derzeitigen Haupteinsatzgebiete für Parallelverarbeitung sind dort wo Massendaten ausgewertet werden. Dies umfasst dabei die Bereiche Biologie, Chemie, Geologie, Luft- und Raumfahrt, Medizin, Wettervorhersage, Klimaforschung, Militär und Physik. Die Bereiche kennzeichnen sich dadurch, dass es sich um sehr komplexe Systeme bzw. Teilsysteme handelt, die in weitreichendem Maße miteinander verknüpft sind. So haben Veränderungen in dem einen Teilsystem meist mehr oder minder starke Auswirkungen auf benachbarte oder angeschlossene Systeme.
Durch den Einsatz verteilter Datenverarbeitung wird es immer leichter möglich, viele solcher Konsequenzen zu berücksichtigen oder sogar zu prognostizieren, wodurch bereits weit im Vorfeld etwaige Gegenmaßnahmen getroffen werden könnten. Dies gilt z.B. bei Simulationen zum Klimawandel, der Vorhersagen von Erdbeben oder Vulkanausbrüchen sowie in der Medizin bei der Simulation neuer Wirkstoffe auf den Organismus. Solche Simulationen sind logischerweise, ganz unabhängig von der Rechenleistung, nur so genau, wie es die programmierten Parameter bzw. Modelle zur Berechnung zulassen.

Beispiel:
SETI@home (Search for extraterrestrial intelligence at home, englisch für „Suche nach außerirdischer Intelligenz zu Hause“) ist ein Verteiltes-Rechnen-Projekt der Universität Berkeley, das sich mit der Suche nach außerirdischem intelligenten Leben befasst.

SETI@Home hat stattdessen das Radioteleskop des auf der Karibikinsel Puerto Rico gelegenen Arecibo-Observatoriums, das zu astronomischen Beobachtungen dient, mit einem zusätzlichen Empfänger ausgerüstet und zeichnet so Radiosignale auf, während das Teleskop andere wissenschaftliche Beobachtungen macht. SETI@Home erhält also eine große Menge an Radiodaten, ohne eigene Teleskopzeit zu belegen. Zur Auswertung der riesigen Datenmengen wird ebenfalls nur wenig eigene Hardware benötigt, die Rechenlast wird stattdessen an die PCs der weltweiten SETI@Home-Gemeinde ausgelagert. Bei diesem Beispiel kann die Verarbeitung sehr gut aufgeteilt werden. Denn man geht u.a. wie folgt vor:

Es werden hauptsächlich drei Tests mit den Daten durchgeführt:

- Suche nach Gaußschen Anstiegen und Fällen der Übertragungsleistung, die möglicherweise auf eine Radioquelle hindeuten könnten.
- Suche nach Pulsen, die eine schmalbandige Digital-artige Transmission sein könnten
- Suche nach Tripeln, also drei Pulsen nacheinander

Oft hängt das Ergebnis jedoch von der Zusammensetzung der anderen Daten ab. Ein einfaches Beispiel ist die Sortierung von Daten.

Arten der Parallelverarbeitung

Systemgesteuerte automatische Parallelisierung
Parallele Programmierung kann zum Beispiel explizit dadurch geschehen, dass der Programmierer Programmteile in separaten Prozessen oder Threads ausführen lässt, oder es geschieht automatisch, so dass kausal unabhängige (nebenläufige) Anweisungsfolgen "nebeneinander" ausgeführt werden. Diese automatische Parallelisierung kann durch den Compiler vorgenommen werden, wenn als Zielplattform ein Parallelrechner zur Verfügung steht, aber auch einige moderne CPUs können solche Unabhängigkeiten erkennen und die Anweisungen so auf verschiedene Teile des Prozessors verteilen, dass sie gleichzeitig ausgeführt werden (Pipeline-Architektur). Die systemgesteuerte automatische Parallelisierung findet heute größere Verbreitung.

Parallelisierung durch parallele Programmierung
Um die Leistungsfähigkeit einer Parallelverarbeitung richtig ausnutzen zu können, muss die Programmierung genau auf die Verteilung zugeschnitten werden. Prinzipiell handelt es sich dabei um ein logistisches Problem. Es gilt die knappen Ressourcen – Rechenzeit, Speicherzugriffe, Datenbusse – effizient auszunutzen. Stets sollte der sequentielle Programm-Overhead minimal sein (Amdahlsches Gesetz).
Parallele Programmierung ist ein Programmierparadigma. Es umfasst zum einen Methoden, ein Computerprogramm in einzelne Teilstücke aufzuteilen, die nebenläufig ausgeführt werden können, zum anderen Methoden, nebenläufige Programmabschnitte zu synchronisieren. Dies steht im Gegensatz zur klassischen, sequentiellen (oder seriellen) Programmierung. Ein Vorteil der parallelen Programmierung ist neben möglicher Effizienzsteigerung (bspw. bei Nutzung mehrerer Prozessorkerne) die Möglichkeit, das typische Alltagsphänomen Nebenläufigkeit direkt in Programmen abzubilden, was zu einfacherem, verständlicherem Quelltext führen kann. Ein Nachteil ist, dass das Laufzeitverhalten paralleler Algorithmen schwieriger nachvollziehbar sein kann als das eines äquivalenten sequentiellen Algorithmus.

Viele bekannte sequentielle Algorithmen sind parallelisierbar, so z. B. einige bekannte Sortieralgorithmen wie Bubblesort oder Quicksort. Es gehört jedoch zu den offenen Fragen der theoretischen Informatik ob *alle* Algorithmen parallelisierbar sind. Jeder parallele Algorithmus ist sequentiell ausführbar, umgekehrt ist es offen.

Je größer die Anzahl der parallel arbeitenden Einheiten ist, desto schneller steigt der Kommunikationsaufwand (egal welchen Typs) überproportional an. Der Zusammenhang ist nicht linear. Das Prinzip der Parallelität ist übertragbar.

So gesehen muss das Optimum zwischen Anzahl der parallel arbeitenden Einheiten und Kommunikationsaufwand und Programmierung gefunden werden. Auch in der Art der Vernetzung der Einheiten liegt ein Optimierungspotential.

Welche Architektur ist sinnvoll?

Die algorithmisch interessanteste Klasse liegt zweifelsohne vor, wenn eine konstante

Anzahl von asynchron arbeitenden Einheiten mit lokalem Speicher verschaltet ist. Das Gegenstück dass sich mehrere Einheiten einen gemeinsamen Speicher teilen, wird hier nicht betrachtet.

Probleme der Parallelisierung

Die typischen Probleme beim Entwurf eines parallelen Algorithmus für diese Konfiguration lassen sich wie folgt charakterisieren:

- **Topologie**
  Abhängig von der Problemstruktur muss ein Kommunikationsgraph für die Vernetzung der Einheiten gewählt werden.
- **Mapping**
  Die Anzahl der Einheiten deckt sich i.d.R. nicht mit der Anzahl durchzuführenden Paralleloperationen. Prozesse die hohen Kommunikationsbedarf miteinander haben sollten in derselben Einheit oder eng beieinander liegenden Einheiten abgearbeitet werden.
- **Lastverteilung**
  Durch das Mapping wird bereits eine statische Lastverteilung definiert, die zu Beginn der Rechnung eine Unter- oder Überbeschäftigung der Einheiten verhindert. Je nach gewähltem Lösungsansatz kann es jedoch erforderlich sein, während der Rechnung einen dynamischen Lastausgleich durchzuführen. Hierzu ist parallel zur Rechnung fortwährend ein Maß für die pro Einheit anliegende unerledigte Arbeit auszuwerten und ggf. durch Austausch von Teilaufgaben die ‘‘Schieflage’’ zu korrigieren.
- **Terminierung**
  Zum Schluss soll noch das Problem der Terminierung angesprochen werden, welches im sequentiellen Fall trivialerweise durch eine Meldung des einzigen Prozessors mitgeteilt werden kann. In einem asynchron arbeitenden Multiprozessorsystem liegt folgende Situation vor: Es gibt aktive und passive Prozesse. Aktive Prozesse haben im Augenblick noch mehrere unerledigte Tasks in ihrer Work-Queue, können also nach einer Weile, wenn die Arbeit erledigt ist, spontan passiv werden. Passive Prozesse sind zurzeit arbeitslos, können aber durch den Empfang einer Nachricht wieder mit Arbeit versorgt werden und wechseln dann in den Zustand aktiv.
  Die entscheidende Frage lautet: Wann sind alle Prozesse passiv?

  Zur Lösung dieser Aufgabe wird der Hamiltonkreis genutzt, der als Teilgraph in der gewählten Topologie enthalten sein sollte. Ein Hamiltonkreis ist ein geschlossener Pfad in einem Graphen, der jeden Knoten genau einmal enthält, ein Knoten ist der Master. Sobald der Master zum ersten Mal passiv wird, schickt er ein grünes Token auf die Reise. Längs des Hamiltonkreises wird das Token von passiven Prozessen weitergereicht. Bei Erhalt des grünen Tokens beim Master färbt dieser es rot und schickt es erneut in die Runde. Ein rotes Token wird von einem passiven Prozeß rot weitergereicht, falls er seit dem letzten Tokendurchgang keine neue Arbeit erhalten hat, andernfalls wird es grün weitergereicht. Erhält der Master ein rotes Token, so folgert er, dass alle Prozesse passiv sind.
  Und wenn sie nicht gestorben sind, dann leben sie noch heute!

Ein sehr guter Kompromiss kann mit folgenden räumlichen Strukturen erzielt werden:

- Cube
- Cube Connected Cycle
- Butterfly

Übersicht über die gängigen Topologien

	Knoten	Durchmesser	Schnittbreite	Grad	konstante Kantenlänge
Array	p	p-1	1	2	Ja
Ring	p	p/2	2	2	Ja
Mesh	$p=k^2$	2k-2	k	4	Ja
Torus	$p=k^2$	k-1	2k	4	Ja
3-D-Mesh	$p=k^3$	3k-3	k^2	6	Ja
Baum	p-1	2(log p - 1)	1	3	Nein
Hypercube	p	log p	p/2	log p	Nein
CCC	$p=2^{k*}k$	2k	2^{k-1}	3	Nein

Welche technische Grenzen/Nebeneffekte bei der Parallelverarbeitung gibt es?

- Parallelität kann u.U. zu Effekten führen, die die Effizienz verringern (oder sogar die Ergebnisse verfälschen).

- Häufig wird ein geringer Parallelitätsgrad mit optimalem Speedup, massive Parallelität aber nur mit einem geringeren Speedup belohnt. Der Sättigungspunkt ist eine Funktion der Problemgröße n.

Parallelisierungsstrategie Divide & Conquer

Von den Parallelisierungsstrategien soll eine als Beispiel kurz vorgestellt werden:

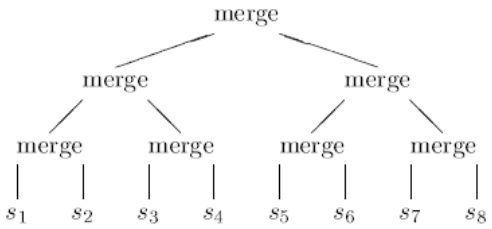
- Divide & Conquer

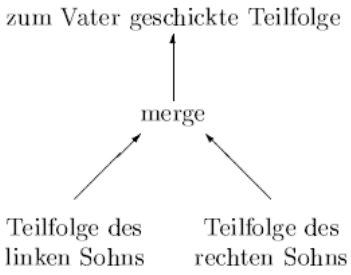
Dabei wird ein Problem in Teilprobleme zerlegt, welche dieselbe Form haben wie das ursprüngliche Problem. Es wird mit Hilfe der Rekursion diese Zerlegung solange durchgeführt, bis keine weitere Teilung mehr möglich ist. Die so vereinfachten Teilaufgaben werden dann bearbeitet und die Ergebnisse kombiniert und wieder zu einer größeren
Teilaufgabe vereint, solange bis die Ebene des Ausgangsproblems erreicht ist und somit alle Teilergebnisse zu einem Gesamtergebnis zusammengeführt wurden.

Diese Methode kann z.B. für globale Operationen auf Listen wie die Sortierung der Liste (Mergesort od. Quicksort), das Auffinden eines max./min. Wertes oder das Suchen in Listen angewandt werden. Werden bei jeder Zerlegung genau zwei Teilprobleme generiert, so entsteht beim rekursiven Divide-and-Conquer Ansatz ein binärer Baum, der bei der Zerlegung nach unten verfolgt werden und bei der Zusammenführung der Ergebnisse nach oben.

Beispiel für einen einfachen parallelisierbaren Algorithmus:
Mergesort:

```
funktion mergesort(liste);
  falls (Größe von liste <= 1) dann antworte liste
  sonst
     halbiere die liste in linkeListe, rechteListe
     linkeListe = mergesort(linkeListe)          ← parallel
     rechteListe = mergesort(rechteListe)        ← parallel
     antworte merge(linkeListe, rechteListe)
```





Leider ist die Laufzeit nicht optimal!

Referenzen:

Parallele Algorithmen, Oliver Vornberger, Fachbereich Mathematik/Informatik, Universität Osnabrück

Implementierung Massiv Paralleler Systeme, Manfred Schimmler, Lars Wienbrandt, Sven Koschnicke, Universität Kiel

Script zur Vorlesung Parallele Algorithmen, Prof. Dr. Burkhard Monien, Jürgen Schulze, Sven Grothlags, Universität Paderborn