Vorlesung Parallele Algorithmen

`http://www.mpi-sb.mpg.de/~sanders/courses/paralg/paralg.html`

Ort: Di u. Do, 11–13, SR 015, Geb. 45. 13.5. und 3.6. sind Feiertag.

Erste Vorlesung: 13.4.99

Dozenten:

- Dr. Peter Sanders, Geb. 46, Raum 315, Tel. 0681 9325 115, `sanders@mpi-sb.mpg.de`; Priv.-Doz.
- Dr. Jop Sibeyn, Geb. 46, Raum 318, Tel. 0681 9325 118, `jopsi@mpi-sb.mpg.de`

Vorkenntnisse: Vordiplom, Grundkenntnisse über sequentielle Algorithmen

Sprechstunde: Immer

Literatur

- PRAM Algorithmen: J. JáJá, „An Introduction to Parallel Algorithms“, Addison Wesley, 1992. Leider vergriffen → Handapparat
- Algorithmen für Verbindungsnetzwerke: F. Thomson Leighton, „Introduction to Parallel Algorithms and Architectures“, Morgan Kaufmann, 1992.
- Einfach und praxisnah: V. Kumar, A. Grama, A. Gupta und G. Karypis, „Introduction to Parallel Computing. Design and Analysis of Algorithms“, Benjamin/Cummings, 1994.
- Probabilistische Algorithmen: R. Motwani und P. Raghavan, „Randomized Algorithms“, Cambridge University Press, 1995.
- Zum selber Programmieren: P. Sanders und T. Worsch, „Parallele Programmierung mit MPI – ein Praktikum“, ISBN 3-931216-76-4, Logos Verlag Berlin, 1997.

Überblick

Teil I: ≈Peter Sanders

- Modelle der Parallelverarbeitung (heute)
- Schaltkreise
- Einfache parallele Algorithmen und das Prinzip der Nachbarschaftskommunikation. 1D versus 2D versus 3D Gitter. Was ist physikalisch realisierbar?
- Konkrete Verbindungsnetzwerke
- Kollektive Operationen
- Lastbalancierung I
- Routing auf Butterfly-Netzwerken
- Lineare Algebra ($\geq$ 2 Vorlesungen)
- Lastbalancierung II ($\geq$ 2 Vorlesungen)
- Einfache Sortieralgorithmen
- Parallele Datenstrukturen

Teil II: ≈Jop Sibeyn

- Routen und Sortieren auf Gittern
- PRAM-Simulation
- Sortieren & Co. auf. PRAMs
- Graphenalgorithmen auf PRAMs

Vermischtes/Optionales

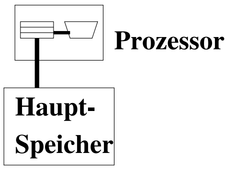
Anwendungen, Externspeicheralgorithmen, untere Schranken …

1 Berechnungsmodelle

[JáJá Kapitel I.]

1.1 Ein (das?) sequentielles Modell

Random **A**ccess **M**achine [von Neumann 45]



- N Speicherzellen (jedenfalls genug)
- Feste Anzahl Taktzyklen für einen Befehl (Speicherzugriff, Berechnung, (bedingter) Sprung) Operationen arbeiten mit Speicherworten a $\mathcal{O}(\log N)$ Bit.

\+ Einfach

\+ Turingmachinen, Termersetzungssysteme, ... nicht so gut geeignet

1.2 Algorithmenanalyse:

- Zyklen zählen: $T(I)$, für gegebene Probleminstanz I.
- Worst case in Abhängigkeit von Problemgröße:

$$T(n) = \max_{|I|=n} T(I)$$

  Beispiel: Heapsort hat Ausführungszeit $\mathcal{O}(n \log n)$ wenn n die Anzahl Elemente ist. Ist klar, daß n ein sinnvolles Maß für die Problemgröße ist?
- $\mathcal{O}(\cdot)$ plättet lästige Konstanten
- Die Ausführungszeit kann von mehreren Parametern abhängen: Beispiel: Eine effiziente Variante von Dijkstra's Algorithmus für kürzeste Wege benötigt Zeit $\mathcal{O}(m + n \log n)$ wenn n die Anzahl Knoten und m die Anzahl Kanten ist. (Es muss immer klar sein, welche Parameter was bedeuten.) [Cormen et al. Abschnitt 25.2]

- Average case:

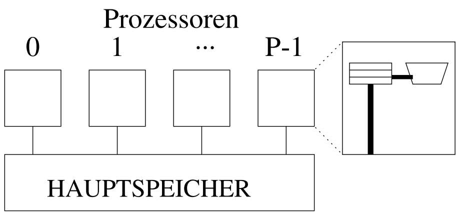
$$T_{\text{avg}}(n) = \frac{\sum_{|I|=n} T(I)}{|\{I : |I| = n\}|}$$

  Beispiel: Quicksort hat average case Ausführungszeit $\mathcal{O}(n \log n)$
- Probabilistische (randomisierte) Algorithmen: $T(n)$ (worst case) ist eine Zufallsvariable. Wir interessieren uns z.B. für deren Erwartungswert (später mehr). Nicht mit average case verwechseln. Beispiel: Quicksort mit *zufälliger* Pivotwahl hat erwarteten worst case Aufwand $\mathbf{E}[T(n)] = \mathcal{O}(n \log n)$ [Cormen et al. Abschnitt 8.3].
- Sekundärziel: Speicherplatz

1.3 Ein einfaches paralleles Modell: PRAMs

Idee: RAM so wenig wie möglich verändern.

- p Prozessoren (**P**rozessor**E**lemente); durchnumeriert von 1 bis p (oder, wenn das eleganter ist, von 0 bis $p-1$). Diese Nummer ist dem Prozessor bekannt.
- In jedem Schritt kann jeder Prozessor einen (fast) beliebigen Befehl ausführen. → Alle Prozessoren arbeiten synchron.
- Alle Prozessoren teilen sich einen gemeinsamen Speicher ohne sich zu behindern (siehe unten)



Zugriffskonflikte?

EREW: Exclusive Read Exclusive Write.
Gleichzeitige Zugriffe sind verboten

CREW: Concurrent Read Exclusive Write.
Gleichzeitiges lesen OK. Beispiel: Einer schreibt, andere lesen = „Broadcast“

CRCW: Concurrent Read Concurrent Write.
Chaos droht:

- **common:** Alle Schreiber müssen sich einig sein. Beispiel: OR in konstanter Zeit (AND?)
- **arbitrary:** Irgendeiner setzt sich durch
- **priority:** Schreiber mit kleinster Nummer setzt sich durch
- **combine:** Alle Werte werden kombiniert. Zum Beispiel Summe. → SBPRAM

Wir verwenden meist common oder arbitrary.
Beispiel: Maximum, [JáJá Algorithmus 2.8].

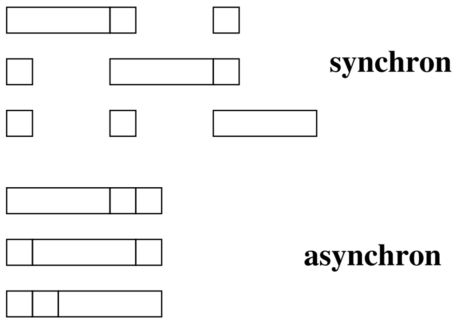
1.4 Formulierung paralleler Algorithmen

- Pascal-ähnlicher Pseudocode
- Explizit parallele Schleifen [JáJá S. 72]
- **S**ingle **P**rogram **M**ultiple **D**ata Prinzip. Der Prozessorindex wird genutzt um die Symmetrie zu brechen. Beispiel: Nochmal Maximum

code PE index as pair (i, j) where $i < j \leq n$
local boolean B
global variable maxValue
$B := (A[i] \geq A[j])$
M[i] := 1
if $\neg B$ **then** M[i] := false
if $M[i]$ **then** maxValue := $A[i]$

Synchron versus asynchron

PE index is j
for $i = 1$ **to** p **do** (a)synchronously
  if $i = j$ **then**
    for $k = 1$ **to** l **do** foo
  else
    foo



pl foos versus $p + l - 1$ foos.

1.5 Analyse paralleler Algorithmen

Im Prinzip nur ein zusätzlicher Parameter: p.
Finde Ausführungszeit $T(I, p)$.
Problem: Interpretation.

Work: $W = pT(p)$ ist ein Kostenmaß. (z.B. Maximum: $W = \Theta(n^2)$)

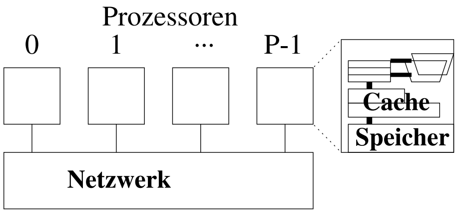
Speedup: $S = T_{\text{seq}}/T(p)$ Beschleunigung. Benutze *besten bekannten* sequentiellen Algorithmus. Relative Beschleunigung $S_{\text{rel}} = T(1)/T(p)$ ist i.allg. was anderes! (z.B. Maximum: $S = \Theta(n)$, $S_{\text{rel}} = \Theta(n^2)$)

Effizienz: $E = S/p$. Ziel: $E \approx 1$ oder wenigstens $E = \Theta(1)$. (Sinnvolles Kostenmaß?) „Superlineare Beschleunigung“: $E > 1$. (möglich?). Beispiel Maximum: $E = \Theta(1/n)$.

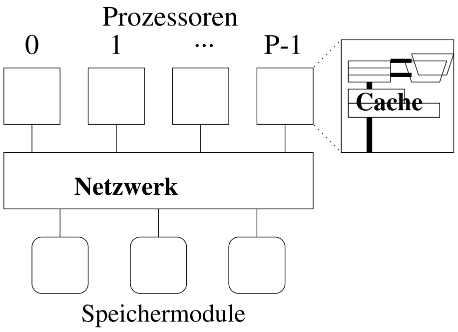
Sehr detaillierte Diskussion in
[Kumar et al. 94, Kapitel 4]

1.6 PRAM vs. reale Parallelrechner

Distributed Memory



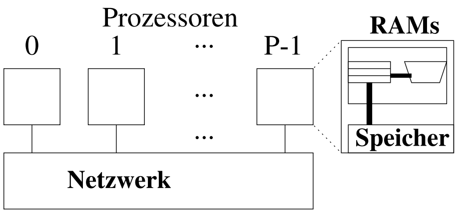
(Symmetric) Shared Memory



Probleme

- Contention (Stau) für gleiche Speichermodule. Beispiel: Der $\Theta(1)$ PRAM Algorithmus für globales OR wird zu $\Theta(p)$. Selbst für EREW-PRAM-Algorithmen kann es Probleme geben.
- Lokaler/Cache-Speicher ist (viel) schneller zugreifbar als globaler Speicher
- Das Netzwerk wird mit zunehmendem p komplizierter und die Verzögerungen werden größer.
- Contention im Netzwerk
- Es interessiert der maximale lokale Speicherverbrauch und weniger die Summe der lokalen Speicherverbräuche

1.7 Modelle mit Verbindungsnetzwerken

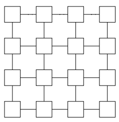


- Prozessoren sind RAMs
- asynchrone Programmabarbeitung
- Interaktion durch Nachrichtenaustausch

Entscheidend ist das Kostenmodell für den Nachrichtenaustausch

1.8 Explizites „Store-and-Forward"

- Wir kennen die Struktur des Verbindungsgraphen $(V = \{1, \ldots, P\}, E \subseteq V \times V)$. Varianten:
  - $V = \{1, \ldots, P\} \cup R$ mit zusätzlichen „dummen" Routerknoten (ggf. mit Pufferspeicher).
  - Busse $\rightarrow$ Hyperedges
- Zu jeder Zeiteinheit kann jede Kante maximal k' Datenpakete konstanter Länge transportieren (meist $k' = 1$)
- In einer k-Port-Maschine kann jeder Knoten k Pakete gleichzeitig senden oder empfangen. $k = 1$ nennt sich *single-ported*. (Variante: ungerichteter Graph, gleichzeitiges Senden/Empfangen über eine Kante OK)



Diskussion

+ einfach formuliert
– low level ⇒ „messy algorithms“
– *Hardwarerouter* erlauben schnelle Komm. wann immer ein Kommunikationspfad gefunden wird.

1.9 Vollständige Verknüpfung

- $E = V \times V$, single ported
- $T_{\text{comm}}(m) = T_{\text{start}} + mT_{\text{byte}}$.

+ Realistische Behandlung von Nachrichtenlängen
+ Viele Verbindungsnetze approximieren vollständige Verknüpfung ⇒ sinnvolle Abstraktion
+ Ergebnis für Maschinen mit Hardwarerouter anwendbar, wenn zusätzlich nachgewiesen wird, daß es keine überlasteten Kanten gibt
+ Durch „künstliches“ Vergrößern von T_{start}, T_{byte} auch an „schwächliche“ Netzwerke anpassbar
+ Asynchrones Modell
– Etwas Händewedeln beim Erklären des Zusammenhangs mit realen Netzwerken

1.10 BSP

Bulk **S**ynchronous **P**rocessing
[McColl LNCS Band 1000, S. 46]

Maschine wir duch drei Parameter beschrieben: p, l und g.

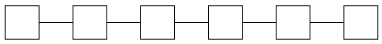
l**:** Startup overhead für einen *kollektiven* Nachrichtenaustausch – an dem alle PEs beteiligt sind

g**:** gap$\approx \frac{\text{Rechengeschwindigkeit}}{\text{Kommunikationsbandbreite}}$

Formaler: Eine h-Relation ist eine kollektiver Nachrichtenaustausch, bei dem jedes PE bis zu h Pakete sendet oder empfängt (beliebige Adressaten). Der Zeitaufwand ist

$$l + hg \ .$$

Diskussion



- g und l können eine Funktion von p sein, wenn man eine ganze Klasse von Verbindungsnetzwerken beschreibt (z.B. lineare Gitter).

+ Abstrakt und einfach
+ An viele Architekturen anpassbar
+ Wenig „Händewedeln“
– Nur global synchronisierte Kommunikation
– Nachrichtenlängen nicht berücksichtigt → $BSP*$
– Einfache und schwierige Kommunikationsmuster nicht unterscheidbar. (z.B. Nachbarschaftskommunikation)

Erweiterte BSP Modelle

h-Relationen sind nur eines von vielen gutuntersuchten Kommunikationsmustern. Wir können zusätzliche Parameter für weitere architekturunabhängige Operationen einführen: (mehr dazu später)

- Broadcast (einer an alle), Minimum, Maximum, Summe, ...
- Nachbarschaftskommunikation auf *logischen* (warum?) Gittern (oder anderen Graphen)
- ...