

Notizen zu Algorithmen II

Jens Ochsenmeier

13. Februar 2018

Inhaltsverzeichnis

- 1 Parallele Algorithmen • 5
 - 1.1 Einleitung • 5
 - 1.2 Nachrichtengekoppelte Parallelrechner • 6

1

Parallele Algorithmen

1.1 Einleitung

Indem man Parallelverarbeitung verwendet, kann man sowohl Ressourcen einsparen als auch Ressourcenrestriktionen brechen:

- **Zeitersparnis:** Arbeiten p Computer an einem Problem, so sind sie bis zu p mal so schnell.
- **Kommunikationsersparnis:** Fallen Daten verteilt an, so kann man sie auch verteilt (vor)verarbeiten.
- **Energieersparnis:** Zwei Prozessoren mit halber Taktfrequenz brauchen weniger Energie als ein voll getakteter Prozessor.
- **Speicherbeschränkung:** Mehr Prozessoren haben mehr Hauptspeicher, mehr Cache,...

Es gibt sehr viele Modelle der Parallelverarbeitung, wir werden hier allerdings nur zwei Standardmodelle diskutieren:

- **Prozessornetze mit Nachrichtenkopplung.**

Prozessoren sind hier "normale CPUs" mit lokalen Speicher. Gemeinsam mit seinem lokalen Speicher wird eine lokale CPU auch *processing element* (PE) genannt. Der Datenaustausch passiert über ein Netzwerk in Form von Nachrichten zwischen zwei Prozessoren.

- **Parallele Registermaschinen mit Speicherkopplung.**

Auch hier wird mit normalen CPUs gearbeitet, allerdings haben diese keinen lokalen Speicher, sondern sind über ein Netzwerk mit Speichermodulen verbunden, auf welche alle CPUs zugreifen können. Der Datenaustausch erfolgt über das Netzwerk zwischen einer CPU und einem Speicher.

Auf nachrichtengekoppelte Rechner werden wir genauer eingehen.

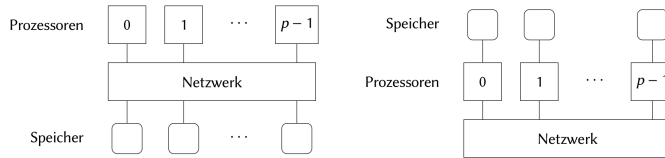


Abbildung 1.1. Vergleich zwischen parallelen Registermaschinen mit *Speicherkopplung* (links) und einem Prozessornetzwerk mit *Nachrichtenkopplung* (rechts).

Nachrichtenkopplung vs. Speicherkopplung

Neben den oben erläuterten Vorteilen von Parallelverarbeitung haben beide Modelle auch Nachteile:

- **Nachrichtenkopplung:**
 - Zwei Prozessoren werden zum Datentransport benötigt. Das passt dem Empfänger nicht immer.
 - Parallelismus muss explizit programmiert werden.
- **Speicherkopplung:**
 - *Skalierbarkeit:* Ist eine große Anzahl an Prozessoren sinnvoll?
 - *Kostenmaß* bei Speicherzugriffskonflikten?

Als eine gute Strategie hat es sich erwiesen, den Entwurf für einen verteilten Speicher durchzuführen, da dieser einen viel breiteren Bereich abdecken kann. Die Implementierung erfolgt dann gegebenenfalls für einen gemeinsamen Speicher.

1.2 Nachrichtengekoppelte Parallelrechner

Modell

- **Netzwerk:** Vollständig verknüpftes Punkt-zu-Punkt-Netzwerk
 - voll-duplex
 - Nachrichten überholen sich nicht

- **Prozessoren:** können jeweils maximal gleichzeitig
 - eine Nachricht an einen beliebigen Empfänger senden ($\text{send}(\text{msg}, \text{to})$)
 - eine Nachricht von einem beliebigen Sender empfangen ($\text{rmsg} := \text{recv}(\text{from})$)
 - oder beides gleichzeitig ($\text{rmsg} := \text{sendRecv}(\text{msg}, \text{to}, \text{from})$)

Als *Kostenmodell* für das Senden oder Empfangen von l Bytes verwenden wir

$$T_{\text{comm}}(l) = T_{\text{start}} + l \cdot T_{\text{byte}},$$

wobei in der Praxis meist $T_{\text{byte}} \ll T_{\text{start}}$. Ignoriert wird hier unter anderem der “Abstand” zwischen Sender und Empfänger.

Als *Programmiermodell* verwenden wir **SPMD** (*single program multiple data*). Alle PEs führen hier dasselbe Programm aus, unterschieden wird lediglich durch “Ränge” der PEs (paarweise verschiedene PE-Nummern).

Parallele Reduktion

Im Folgenden gehen wir über die grundlegenden Werkzeuge, die wir benötigen, um parallele Programme analysieren zu können.

Definition 1.2.1 (Reduktion). Sei \otimes eine binäre, assoziative Operation auf einer Menge M .

Für $x = (x_0, \dots, x_{p-1}) \in M$ definieren wir

$$R_{\otimes}(x) = \bigotimes_{i < p} x_i = x_0 \otimes \dots \otimes x_{p-1}.$$

Nun gilt folgender Satz:

Satz 1.2.2. Wenn \otimes eine binäre, assoziative Operation ist und p Elemente x_0, \dots, x_{p-1} auf p PEs verteilt sind, dann kann man $\bigotimes_{i < p} x_i$ in Zeit $O(\log p)$ auf PE 0 berechnen.

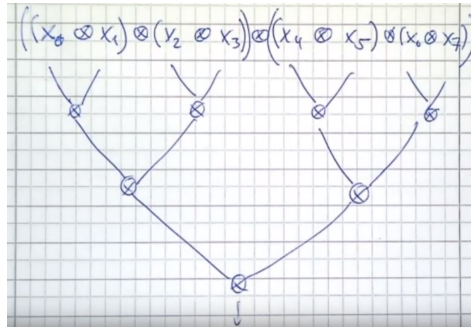


Abbildung 1.2. Idee hinter obigem Satz.

Sequenziell wäre die optimale Laufzeit $O(n)$ gewesen. Auf $p = n$ PEs hätte man nach obigem Satz eine Laufzeit von $O(\log n)$ erreicht, also eine Beschleunigung von $O\left(\frac{n}{\log n}\right)$. "Ideal" wäre eine Beschleunigung von $O(p) = O(n)$ gewesen.

Wie kann man die Beschleunigung noch weiter verbessern?

Parallele Reduktion mit $p < n$

Verwenden wir nun $p < n$ viele PEs, um eine Reduktion auf n Elementen durchzuführen, so erhält jedes PE n/p Datenelemente. Die PEs berechnen zuerst die Reduktion der lokalen Elemente, und anschließend wird auf diesen Reduktionen eine parallele Reduktion durchgeführt. Laufzeit hierfür ist $O(n/p) + O(\log p)$ und die Beschleunigung somit

$$\frac{O(n)}{O(n/p + \log p)}.$$

Ist $p \in O(n/\log n)$, so ist die Beschleunigung $O(p)$.

Man kann also durch Verringerung der Prozessorzahl p die Beschleunigung in die Nähe von p bringen. Dieses Prinzip nennt man **Brent's Prinzip**.

Analyse paralleler Programme

Wir interessieren uns in erster Linie für

- Laufzeit,
- Beschleunigung und
- verrichtete Arbeit.

Es ist

- $T_{\text{par}}(I, p)$ die parallele Laufzeit der Probleminstanz I , bearbeitet mit p PEs,
- $T_{\text{seq}}(I)$ die sequenzielle Laufzeit der Probleminstanz I mit dem "besten bekannten Algorithmus".

Im Allgemeinen ist $T_{\text{seq}}(I) < T_{\text{par}}(I, 1)$. Man erhält aber leicht Gleichheit, indem man den parallelen Algorithmus einfach den sequenziellen ausführen lässt, falls nur ein PE zur Verfügung steht.

Definition 1.2.3 (Speedup). Wir definieren den **Speedup** als

$$S(I, p) = \frac{T_{\text{seq}}(I)}{T_{\text{par}}(I, p)}.$$

Vergrößert:

$$S(n, p) = \inf \{S(I, p) : n = |I|\}.$$

Zur Berechnung des Speedups können wir praktischerweise folgende Spezialfälle benutzen, die für Instanzen I, I' gleicher Größe n gelten:

- $T_{\text{par}}(I, p) = T_{\text{par}}(I', p) = T_{\text{par}}(n, p),$
- $T_{\text{seq}}(I) = T_{\text{seq}}(I') = T_{\text{seq}}(n)$