Notizen zu Algorithmen II

Jens Ochsenmeier

13. Februar 2018

Inhaltsverzeichnis

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:

- **Prozessornetzwerke 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.

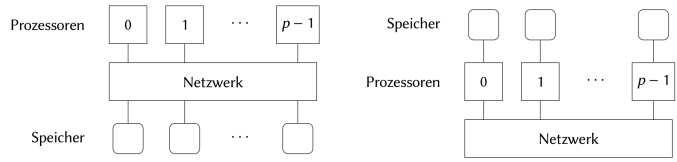

Abbildung 1.1. Vergleich zwischen *parallelen Registermaschinen mit Speicherkopplung* (links) und einem *Prozessornetzwerk mit Nachrichtenkopplung* (rechts).

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 (send(smsg,to))
  - eine Nachricht von einem beliebigen Sender empfangen (rmsg := recv(from))
  - *oder* beides gleichzeitig (rmsg := sendRecv(smsg, to, 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 $x = (x_0, \ldots, x_{p-1}) \in M$ definieren wir

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

Nun gilt folgender Satz:

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

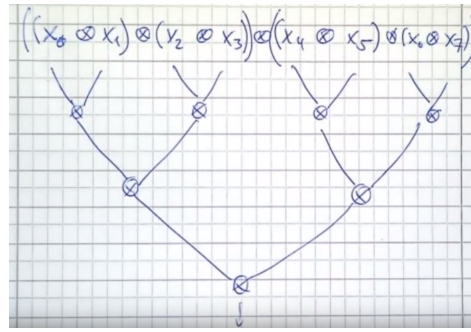

Abbildung 1.2. Idee hinter obigem Satz.

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

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

Vergröbert für alle Probleminstanzen der Größe n:

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

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

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

Simuliert man p Prozessoren durch einen Prozessor, so sehen wir, dass ein sequenzieller Algorithmus nie langsamer als $O(p \cdot T_{\text{par}}(I, p))$ sein kann, weswegen immer $\frac{T_{\text{seq}}(I)}{T_{\text{par}}(I,p)} \in O(p)$ ist und $S(n, p) \in O(p)$ ebenfalls.

Definition 1.2.4 (Effizienz). Wir definieren die **Effizienz** eines parallelen Algorithmus als

$$E(n, p) := \frac{S(n, p)}{p}.$$

Es ist $S(n, p) \in O(p)$ und daher $E(n, p) \in O(1)$. Tatsächlich ist sogar $E(n, p) > 1$ möglich (sogenannter *superlinearer Speedup*).

Es ist

$$E(n, p) \in \frac{T_{\text{par}}(n, p)}{p \cdot T_{\text{seq}}(n)}.$$

Definition 1.2.5 (Arbeit). Wir definieren **Arbeit** als

$$W(n, p) := p \cdot T_{\text{par}}(n, p).$$

Mit obiger Definition gilt

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n,p) \cdot p} = \frac{T_{\text{seq}}(n)}{W(n,p)}.$$

Wir können nun die parallele Reduktion mit $p < n$ von vorhin genauer analysieren:

- Es werden zuerst $\frac{n}{p}$ Elemente sequenziell und anschließend p partielle Summen reduziert.
- **Laufzeit**: $T_{\text{par}}(n,p) = O(n/p) + O(\log p)$
- **Beschleunigung**: $S(n,p) = \frac{O(n)}{O(n/p+\log p)}$
- **Effizienz**: $E(n,p) = \frac{O(n)}{O(n+p\log p)}$

Wir betrachten noch die zwei Sonderfälle:

- $p = n$:
  - $S(n,n) \in O\left(\frac{n}{\log n}\right)$
  - $E(n,n) \in O\left(\frac{1}{\log n}\right)$
- $p \in O\left(\frac{n}{\log n}\right)$:
  - $S(n,p) = O(p)$
  - $E(n,p) = O(1)$

Wir erkennen hier die größere Effizienz für kleinere p (nach Brents Prinzip).

Parallele Präfixsummen

Wir verwenden $\otimes$ wie oben definiert. Wir definieren nun

$$P_\otimes(x) = y = (y_0, \ldots, y_{p-1})$$
$$\text{mit } y_i = \bigotimes_{k \le i} x_k.$$

Es ist also $y_0 = x_0$ und $y_{i+1} = y_i \otimes x_{i+1}$.

Präfixsummen — Hyperwürfel-Algorithmus

Wir machen es uns hier einfach und legen fest, dass $p = n = 2^d$ (für $d \in \mathbb{N}$) und $\otimes$ kommutativ.
Jede PE erhält nun eine "Koordinate" $0 \le i \le 2^d - 1$. Diese wird als Bitvektor

$$i = (i_{d-1} \cdots i_0) \quad \text{mit} \quad i_j \in \{0,1\}$$

repräsentiert. Hier ist i_k das k-te Bit von rechts in i.
Wir erlauben Kommunikation zwischen zwei PE i und i' nur, wenn die Hammingdistanz ihrer Bitvektoren 1 ist, sie sich also nur in einer Stelle unterscheiden. Wir erhalten so einen *Hyperwürfel* aus PEs.
Wir berechnen nun Präfixsummen auf einem solchen Hyperwürfel:

```
PREFIXSUM(x, ⊗)
y := x  // auf PE i liegt x_i
s := x  // für Summe von Elementen in Unterwürfel
for k := 0 to d − 1 do
  s' := SENDRECV(s, i ⊕ 2^k, i ⊕ 2^k)
  s := s ⊗ s'
  if i_k ≡ 1 then y := y ⊗ s'
// auf PE i liegt y_i = x_0 ⊗ ··· ⊗ x_i
```

Wir erhalten eine Laufzeit

$$T_{\text{prefix}} \in O((T_{\text{start}} + l \cdot T_{\text{byte}}) \cdot \log p).$$

Diese Laufzeit ist nicht optimal. Wie man sie optimieren kann wird in der Vorlesung "Parallele Algorithmen" näher erläutert.

Paralleles Sortieren

Hier gibt es zwei verschiedene Aufgabenvarianten:

1. Alle n Elemente liegen zu Beginn auf PE 0. Deswegen muss jedes Element von PE 0 mindestens einmal angefasst werden. Die Laufzeit ist daher in $\Omega(n)$.
2. Je n/p Elemente liegen zu Beginn auf PE i. Dieser Fall ist wesentlich interessanter.

Zunächst betrachten wir den einfachen Fall $p = n$ (Prozessor i hat Eingabeelement x_i). Wir behalten die Grundidee von Quicksort bei:

- Wir wählen ein Element pv als Pivot.
- Elemente werden umverteilt:
  - kleiner als pv: auf Prozessoren mit kleineren Rängen
  - größer als pv: auf Prozessoren mit großen Rängen
- Parallele Rekursion.