Föreläsning 15: Parallella algoritmer

Parallella algoritmer

Modellen är följande:
Istället för en enda processor tänker vi oss ett godtyckligt (i princip) antal processorer.
Vi tillåter i analysen att antalet processorer beror på n.
Tiden för en parallell algoritm definieras som för en vanlig algoritm.
Kostnaden definieras som tiden $\times$ antalet processorer.
Arbetet är antalet operationer som totalt utförs.

Ex: Matrismultiplikation. Vi har 2 st. $n \times n$-matriser som skall multipliceras. Med normal metod tar det tid $O(n^3)$.

Antag att vi har n^2 processorer som räknar ut alla matriselementen **samtidigt**. Tiden det tar då är $O(n)$.

Parallell MergeSort

```
ParaMergeSort(v[i..j])
(1)   if i < j
(2)       m ← ⌊(i+j)/2⌋
(3)       ParaMergeSort(v[i..m])
(4)       ParaMergeSort(v[m+1..j])
(5)       v[i..j] = Merge(v[i..m], v[m+1..j])
```

De båda anropen på raderna (3) och (4) utförs parallellt, d.v.s. på var sin processor.

Tid:	$n + n/2 + n/4 + \cdots + 1 \approx 2n$
Processorer:	$n/2$
Arbete:	$\Theta(n \log n)$
Kostnad:	$\Theta(n^2)$

Jämför sekventiell MergeSort där körtiden är $\Theta(n \log n)$.

Hur kraftfull är parallellitet?

Varje algoritm som tar tid $t(n)$ och använder $p(n)$ processorer kan simuleras på en enda processor i tid $t(n)p(n)$.
Oavgörbara problem förblir därför oavgörbara också om ett stort antal processorer tillåts.
Så länge antalet processorer är polynomiskt i n hjälper inte heller parallellitet på de **NP**-fullständiga problemen.

Klassen NC

NC består av problemen som kan lösas i
polylogaritmisk (d.v.s. $O(\log^k n)$ för något k) tid med polynomiskt många processorer.
För ett problem i **NC** är arbetet polynomiskt, så $\mathbf{NC} \subseteq \mathbf{P}$.
Ex. på problem i **NC**: Sortering och matrismultiplikation
Man tror att $\mathbf{NC} \neq \mathbf{P}$. Det finns nämligen många problem i **P** som ingen **NC**-algoritm är känd för, t.ex. maximalt flöde i graf och GCD.

Praktiska problem med parallella algoritmer

Kommunikation Processorerna arbetar i regel med samma globala data. Vad händer om flera processorer samtidigt vill läsa eller skriva på samma minnesposition?

Resursfördelning Hur gör man så att fördelningen av t.ex. diskåtkomster blir rättvis?

Synkronisering De flesta parallella algoritmer förutsätter att alla processorer arbetar i fas. Hur garanteras detta?

Komplexitetsklasser

Vi har redan sett definitioner av klasserna **P** och **NP**. Nedan är några andra definitioner av klasser.

LogTime: Består av problem som kan avgöras i tid $O(\log^k(n))$.

LogSpace: Består av problem som kan avgöras med användande av minne av storlek $O(\log^k(n))$.

PSpace: Består av problem som kan avgöras med användande av minne av storlek $O(n^k)$.

ExpTime: Består av problem som kan avgöras i tid $O(2^{n^k})$.

ExpSpace: Består av problem som kan avgöras med användande av minne av storlek $O(2^{n^k})$.

Time(f(n)): Består av problem som kan lösas i tid $\leq f(n)$.

Space(f(n)): Består av problem som kan lösas med minne $\leq f(n)$.

Vi har följande samband:

LogTime $\subseteq$ **LogSpace** $\subseteq$ **P** $\subseteq$ **NP** $\subseteq$ **PSpace** $\subseteq$ **ExpTime** $\subseteq$ **ExpSpace**.

Som förklaring till några av dessa relationer kan nämnas att det alltid gäller att om $L \in$ **Time(f(n)** så gäller $L \in$ **Space(f(n)**. Förklaringen till detta är att en algoritm som arbetar i k tidssteg inte kan besöka eller modifiera mer än k minnesceller.

Att **NP** $\subseteq$ **PSpace** kan förklaras på följande sätt: Om $L \in$ **NP** så finns det en icke-deterministisk Turingmaskin och en accepterande beräkning av *polynomiell* längd. Det finns oftast ett exponentiellt antal möjliga beräkningar att testa. Men en vanlig Turingmaskin kan gå igenom dessa beräkningar i tur och ordning. De kräver var och en högst polynomiellt minnesutrymme och minnet kan raderas och användas för nästa beräkning. Även om tiden blir exponentiell blir minnesåtgången polynomiell. Alltså $L \in$ **PSpace**.