

## Work-Span 1: Model

(21/12/2016)

- ① PRAM, Optellen.
- ② CREW, OR
- ③ Work & Span
- ④ Scheduling
- ⑤ Analyse: Som
- ⑥ Conclusies

Op GPU is grootschalig parallellisme bereikbaar.  
In C# is het handig te gebruiken dmv  
Parallel.For en Invoke.

Het Work-Span model helpt je, goeie algoritmen te bedenken en te analyseren om dit parallellisme efficient te gebruiken.

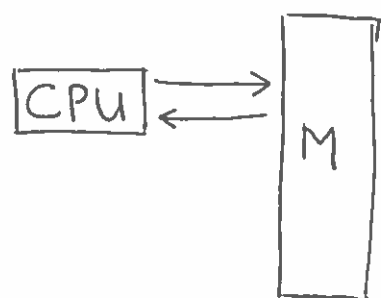
Nodig (uit bv Data Structuren): Master Theorem

Beperking van WS-Model: geheugen bandwidth wordt geen rekening mee gehouden

## ① De PRAM

Voor ontwerp en analyse van sequentiële algoritmen gaan we uit van het RAM model: Random Access Machine.

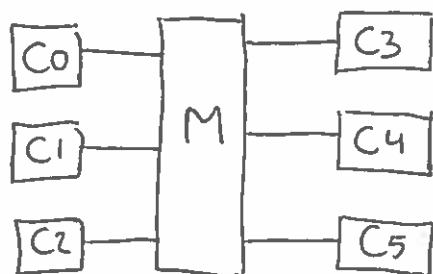
Alle locaties in het geheugen kunnen in  $O(1)$  tijd worden gelezen / geschreven: Random Access.



```
s = 0  
for(i=0; i<n; i++)  
    s = s + Ai
```

Sommeren (berekenen van  $\sum_{j=0}^n A_j$ ) kan in  $\Theta(n)$  tijd.

Te analyseren aan algoritmen: tijd en geheugen.



Het PRAM model, Parallel Random Access Machine, veronderstelt ook één algemeen toegankelijk geheugen, maar daarnaast meerdere processors.

Kan ik alle optelstappen parallel doen?

```
s = 0  
parallel.for (i=0..n)  
    s += Ai
```

Er bestaat (in C# en OpenCL) wel zoiets als atomic-addition, maar de diverse tasks zullen

op de toegang tot  $s$  worden gesequentialiseerd zodat dit toch lineaire tijd gaat kosten

Zoem Hoe kun je snel optellen?

Stap 1 Voor even  $i$

$$A_i += A_{i+1}$$

Stap 2 Voor viervoud  $i$

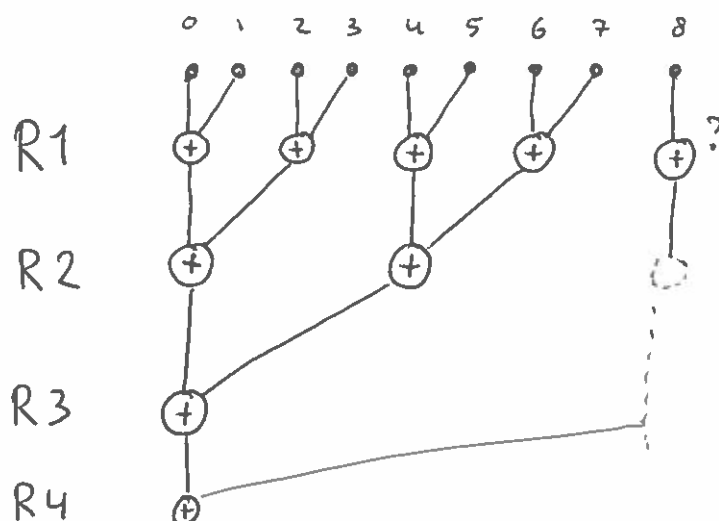
$$A_i += A_{i+2}$$

Stap 3 Voor achttvoud  $i$

$$A_i += A_{i+4}$$

Stap 4 Voor zestenvoud  $i$

$$A_i += A_{i+8}$$



Om stap 1 in  $1*$  te kunnen heb ik  $n/2$ , dus  $\Theta(n)$  cores nodig. Om de berekening af te maken heb ik de apparatuur  $\Theta(\lg n)$  tijd nodig.

Dus: de Hardware Kosten, gedefinieerd als  $p * t_p$  is hier  $\Theta(n \lg n)$ .

Het lijkt alsof optellen wel sneller kan met parallelisme (tijd  $\Theta(\lg n)$  ipv  $\Theta(n)$ ) maar tegen hogere kosten!

Zoem: ① Kan het sneller dan  $\Theta(\lg n)$   
② Kan het met minder Hardware Kosten?

Ad ①, Sublogaritmisch?

Optellen is een binaire operator

Een parallelle executie van binaire operaties kan, na  $i$  rondes, alleen waarden produceren die van hoogstens  $2^i$  inputs afhangen!

Dus hoewel je ook tegelijk doet, de som van  $n$  getallen kun je niet in minder dan  $\lg n$  rondes produceren!

Bewijsje met inductie.

$i=0$ : Na 0 rondes heb je alleen de invoeren, elke invoer is alleen "afhankelijk van" zichzelf dus 1 invoer ofwel  $2^0$ .

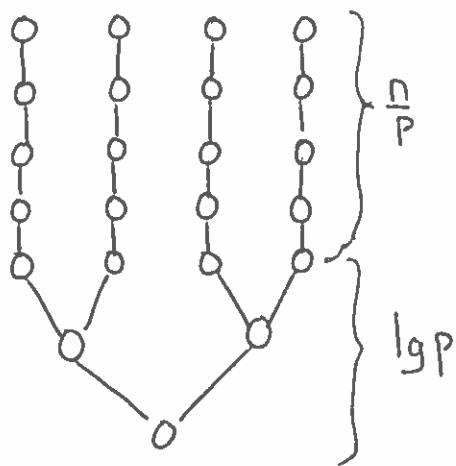
$i+1$ : In ronde  $i+1$  kun je alleen twee dingen combineren uit ronde  $i$ , dus (IH) die van hoogstens  $2^i$  inputs afhangen. Het resultaat hangt van hoogstens  $2^i + 2^i$  dus  $2^{i+1}$  invoeren af.  $\square$

Conclusie: Optellen kan niet sublogaritmisch

In feite geldt dit voor alles wat je met binaire operaties moet uitrekenen. Daarom geldt  $\Theta(\lg^c n)$  (polylogaritmisch) als een soort hoogst bereikbaar doel bij parallel rekenen.

Suggestie voor ②:

Probeer dit twee-fasen systeem.



- Als er  $p$  cores zijn, verdeel  $A$  in  $p$  groepjes van  $n/p$
- Fase 1: core telt zijn groepje sequentieel op
- Fase 2: Samenwerken met subtotalen volgens tree-systeem.

Dit kost  $t_p = \underset{\textcircled{1}}{n/p} + \underset{\textcircled{2}}{\lg p}$ .

Zolang  $p < \frac{n}{\lg n}$  is term ① dominant

Met  $p = \frac{n}{\lg n}$  is  $t_p = \Theta(\lg n)$ ,  
en Hardware Kosten =  $\Theta(n)$

Goed nieuws: het kan tegen dezelfde kosten als sequentieel ( $HK = \Theta(n)$ ) maar dan snel nl. in  $\log$  tijd!

Bedenkelyk: Moet je voor elk aantal cores een andere berekening definiëren?

NEE, daarvoor in de plaats komt de work-span theorie!

## ② CREW, Disjunctie

Gegeven  $n$  bits  $b_0 \dots b_{n-1}$   
Bereken  $C = b_0 \vee b_1 \vee \dots \vee b_{n-1}$

Disjunctie kost sequentieel

$\Theta(n)$  bit operaties. Het is een binaire operatie

dus je verwacht dat  
het parallel  $\Omega(\lg n)$  kost.

R0  $C = F$

R1  $\forall i: \text{if}(b_i)$

R2  $\uparrow$   
Per  $i$  een  $\{C = T\}$   
core

ALS de writes op  $C$  na elkaar worden afgewikkeld, het Exclusive Write model, zal dit nog  $\Theta(n)$  tijd kosten (of: zoveel als aantal Trues).

Het Concurrent Write idee veronderstelt dat meerdere writes tegelijk kunnen (waarna een van de geschreven waarden blijft zitten). In het CW model kost dit programma  $\Theta(1)$  tijd. Er wordt een T in  $C$  geschreven als er een of meerdere  $b_i$  gelden dus de disjunctie wordt berekend.

Dus: Disjunctie kan in constante tijd  
(Superparallel) op een CW-PRAM.

## Search

$i$	1	2	3	4	5	6	7
$p_i$	2	3	5	7	13	17	19

Gegeven een gesorteerde

rij  $p$  en een  $x$ , vind (unieke)  $a$  zdd  $p_a < x \leq p_{a+1}$

Neem aan dat je een core hebt voor elke  $i$  en laat de cores parallel deze code uitvoeren.

R.1: if ( $p_i < x$   
R2         $\wedge x \leq p_{i+1}$ )  
R3        {  $a = x$  }

Precies 1 core gaat in  $a$  schrijven dus het kan op EW.

Wel moeten we aannemen dat alle cores tegelijk  $x$  kunnen lezen! Als dat lezen van  $x$  wordt gesequentialiseerd (Exclusive Read) kost dit alsnog lineaire tijd.

Als lezen door meerdere cores tegelijk kan (Concurrent Read) kan het totaal ook in constante tijd.

Conclusie Op een CREW-PRAM kan Search in  $\Theta(1)$  tijd met  $\Theta(n)$  cores.

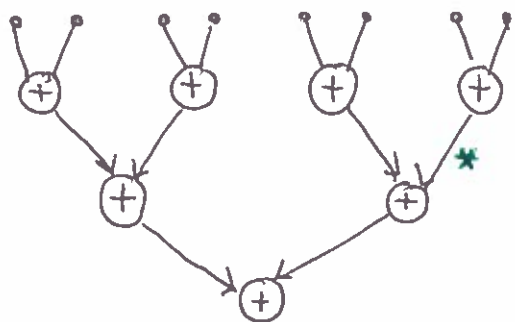
Het meest bestudeerde model is CREW-PRAM

(Binary Search is minder werk  $\Theta(\lg n)$  stappen, maar die kunnen alleen strikt na elkaar!)

### ③ Work, Span, Scheduling

Je wilt niet voor elk aantal cores een andere optimale berekening. Definieer een algoritme met 2 parameters (ipv de ene: tyd voor een sequentieel) die je vervolgens met Scheduling op elk aantal cores kunt afbeelden.

Zie berekening als netwerk (graaf) van uit te voeren stappen, waarbij precedentierelaties als pijl worden weergegeven:



Optellen van 8 inputs

Binair zoeken in 7

\* Afhankelyk : tweede optelling kan pas als resultaat van eerste bekend

\$ Afhankelyk : pas nadat x met het midden is vergeleken weet je wat de volgende vergelyking is.

#### Definitie

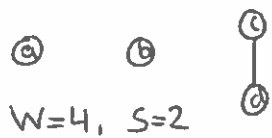
work = aantal knopen in de berekeningsgraaf

span = aantal knopen op langste pad.



Als  $w$  en  $s$  bekend zijn, kun je de tijd op  $p$  cores schatten als  $T_p \approx \frac{w}{p} + s$

of  $\max(\frac{w}{p}, s)$ ,  
wat binnen een factor 2 hetzelfde is.



Vb.  $T_1$ , 1 core doet alles  
na elkaar dus  $T_1 = w = 4$

$T_\infty = 2$ , als er ongelimiteerd  
cores zijn is  $s$  de tijd!

$T_3 = 2$  : Doe  $a$   $b$   $c$  in  $R1$   
en  $d$  in  $R2$ .

Voor  $T_2$  lijkt het of je handig moet zijn

Zoem: Wat is  $T_2$ ?

Als je begint met  $ab$  in  $R1$ , kun je alleen  
 $c$  doen in  $R2$  en heb je 3 rondes nodig. (B)  
Als je begint met  $ac$  kun je daarna  $bd$   
doen en is 2 genoeg.

Voor het exacte minimum aantal rondes  
moet je de graaf goed analyseren. vooruit  
kijken en complex plannen!

#### ④ Brent: Scheduling

Brent bewees dat "Greedy Scheduling" altijd 2-Optimaal is. Stel  $T_p^*$  is de tijd van 't snelste schedule

Feit 1:  $T_p^* \geq \frac{w}{p}$       want zelfs 't beste kan hoogstens  $p$  tegelijk.

Feit 2:  $T_p^* \geq s$       want de rekenstappen op een pad moeten hoe dan ook na elkaar.

Noem een stap "ready" als zijn voorgangers in vorige ronden zijn geweest.

Een Greedy Scheduler pakt in elke ronde zoveel mogelijk ready stappen, maar denkt niet na over welke! Schedule (B) is dus Greedy!

Zoveel mogelijk wil zeggen:

- Als er  $p$  of meer ready stappen zijn, dan stop je er  $p$  in de ronde, elke core bezet
- of • Er zijn minder dan  $p$  ready stappen, dan doe je alle ready stappen in de ronde (maar niet alle cores zijn bezet!)

De eerste situatie noemen we een VOLLE ronde en komt hoogstens  $w/p$  keer voor omdat er maar  $w$  stappen zijn.

De tweede situatie noemen we een LEGE ronde en komt hoogstens  $s$  keer voor omdat het langste pad in de resterende graaf 1 korter wordt.

De lengte van het Greedy Schedule is dus

$$T_p \leq \frac{w}{p} + s$$

En nu geldt:

$$T_p \leq w/p + s$$

net bewezen

$$\leq T_p^* + T_p^*$$

Feit 1, Feit 2

$$= 2 * T_p^*$$

Dus een Greedy Schedule heeft een lengte die hoogstens tweemaal het optimale is.  
Ofwel asymptotisch optimaal  $\Theta(T_p^*)$ .

Omdat asymptotisch  $\frac{w}{p} + s$  en  $\max(w/p, s)$  hetzelfde zijn (verschillen hooguit een factor 2)

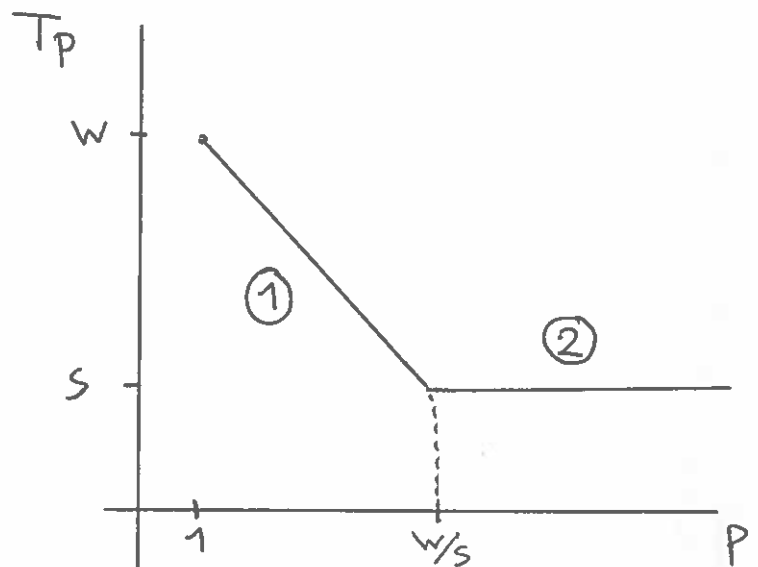
kunnen we stellen dat

$$T_p = \max(w/p, s)$$

- ① Zolang  $p \leq \frac{w}{s}$ , is de eerste dominant. De rekentyd kan met meer cores verlaagd worden.

Work-bound fase

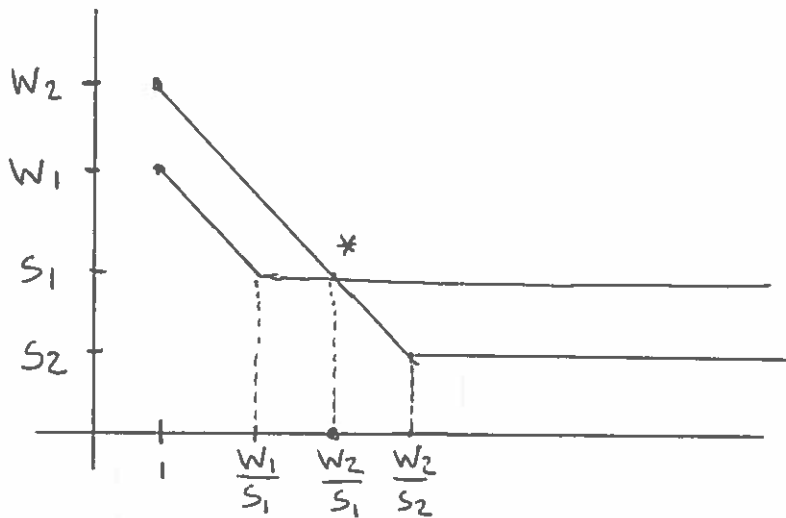
- ② Als je  $p \geq \frac{w}{s}$  cores neemt, daalt de tijd niet verder: Span-bound fase



Voor veel algoritmen zit je vaak, zelfs met het megaparallelisme van de GPU, nog in de Work-bound fase.

Bij het vergelijken van algoritmen is natuurlijk een lage w beter dan een hoge w en een lage s beter dan een hoge s.

Wat moet je doen als je ene algoritme  $A_1$  een betere work heeft:  $w_1 < w_2$  en je andere,  $A_2$ , een betere span:  $s_1 > s_2$



Je kunt het break-even punt uitrekenen waar  $A_2$  sneller wordt dan  $A_1$ , \*, in termen van  $t_p$ , en dat is doorgaans heel hoog!

I.i.g. na de

Work-bound fase van Algoritme 1.

Conclusie: Een lage span is theoretisch leuk, maar pas ervoor op je span te verlagen ten koste van een hogere work!

## ⑤ Analyse: Som

Het berekenen van Work en Span van een programma/ algoritme gaat net als het berekenen van de tijd van een algoritme zoals dat by DataStructuren werd geleerd.

- Instructies bekijken/tellen.
- Loop: kosten van body keer aantal slagen
- Recursie: Betrekking opstellen en M.T.

Voor de analyse van parallelle algoritmen komen er een paar vrij logische truukjes by:

- Je moet het twee keer doen, namelijk voor Work en voor Span.
- Work, het totale aantal reken stappen, is precies wat je by sequentiële algoritmen (dus by DS) al uitrekende, dus dat gaat op de "oude" manier.
- Voor de Span moet je dingen die parallel worden uitgevoerd, maar 1\* tellen!

Voorbeeld: Alle getallen in een  $n \times n$  matrix  $A[i, j]$  op tellen, met subtotalen per kolom.

1.  $\text{for } (i, 0, n)$
2.      $\{ s_i = 0$
3.          $\text{for } (j=0; j < n; j++) \quad s_i += A_{ij} \}$
4.      $t = 0$
5.      $\text{for } (i=0; i < n; i++) \quad t += s_i$

Work: - Loop op r2/3 kost  $\Theta(n)$  stappen  
- Volgens r1 wordt dat  $n \times$  gedaan  
dus r1-3 kost  $\Theta(n^2)$   
- r4/5 kost  $\Theta(n)$   
- Totaal  $\Theta(n^2)$  werk.

Het berekenen van  $W$  houdt eigenlijk geen rekening met parallelisme.

Span - Loop in r3 is sequentieel dus kost  $n$  stappen na elkaar.  
- Met r2 erby nog steeds  $\Theta(n)$  voor body.  
- De loop in r1 is een parallelle dus voor het hele stuk r1-3 hebben we nog steeds  $\Theta(n)$  als langste pad van stappen  
- r4/5 geeft ook weer  $\Theta(n)$  stappen na elkaar  
- Totale Span is  $\Theta(n) + \Theta(n) = \Theta(n)$ .

Het voorbeeld heeft dus     Work      $\Theta(n^2)$   
                                  en     Span      $\Theta(n)$

By het ontwerpen van parallelle algoritmen kun je vaak succesvol gebruik maken van Recursie.

Je wilt een methode  $\text{Som}(A, p, q)$  die de som van getallen uit  $A$  oplevert (inc  $p$ , exc  $q$ ).

Doe alsof je al weet hoe je op z'n slimst  $n/2$  getallen kunt optellen; hoe kun je dan  $n$  getallen optellen?

### Aanpak 1 : Compressie

Combineer je inputs twee aan twee parallel, om te reduceren tot een kleinere invoer.

$\text{Som}(A, p, q)$

1.  $\text{pfor}(i, 0, q-p/2)$
2.  $B[i] = A[p+2*i] + A[p+2*i+1]$
3.  $\text{return Som}(B, 0, q-p/2)$

- ① Mogelijkheid van oneven aantal is hier genegeerd
- ② Kleine invoer (1 of 2 getallen): niet in recursie!
- ③ Vaak is er ook een threshold om wel in recursie te gaan maar niet parallel te werken.  
( if  $(q-p < 1200)$  return SeqSum .... )

Analyse van Werk voor  $n$  getallen:

- $r_2$  is  $\Theta(1)$  werk.
- $r_1$  zegt dit  $n/2$  keer doen dus  $\Theta(n/2)$
- Dan  $r_3$  een aanroep op  $n/2$  inputs!

Noem het werk  $W_c$  (C van Compressie)

dan vind je

$$W_c(n) = \Theta(n/2) + W_c(n/2)$$

Dit is op te lossen met de Master Theorem  
en geeft  $W_c = \Theta(n)$ .

Span: Noem de Span  $S_c(n)$ .

De compressie in  $r_1/2$  kost maar constante Span omdat alle paren onafhankelijk worden opgeteld. Dus geldt

$$S_c(n) = \Theta(1) + S_c(n/2)$$

met oplossing (uit MT)

$$S_c(n) = \Theta(\lg n).$$

Conclusie: Met de Compressie-methode kun je  $n$  getallen optellen in lineair Work en logaritmische Span.



Aanpak 2: Split Verdeel je invoer in twee kleinere delen die je apart recursief oplost.

```
Som (A, p, q)
1  if (q = p+1) return Ap
2  m = (p+q) / 2
3  Invoke ( s = Som (A, p, m)
4           t = Som (A, m, q) )
5  return (s+t)
```

Het ziet er heel anders uit dan Compressie!  
Want ik heb nu twee recursieve aanroepen op halve input grootte.

Werk voor Split-aanpak:  $W_s(n)$

Regels 1, 2, 5 kosten  $\Theta(1)$

Regels 3 en 4 elk  $W_s(n/2)$

Dus  $W_s(n) = \Theta(1) + 2 \cdot W_s(n/2)$

met oplossing  $W_s(n) = \Theta(n)$

Span  $S_s(n)$

Regels 3 en 4 gaan parallel dus tellen 1\* :

RB is  $S_s(n) = \Theta(1) + S_s(n/2)$

met oplossing  $S_s(n) = \Theta(\lg n)$

Conclusie: Met een Split-aanpak kan ik ook  $n$  getallen optellen in lineair Work en logaritmische Span.

## ⑥ Conclusie en vooruitblik

- Work/Span model: Kijk naar  $W$ : totale aantal rekenstappen en  $S$ : langste sliert (die door afhankelijkheid noodzakelijk na elkaar komen)
- Negeer geheugen issues zoals caching:  
PRAM model stelt uniforme kosten voor elke geheugentoegang.
- Tijd voor uitvoering op  $p$  cores:  $T_p = \Theta(\frac{W}{p} + S)$   
(Vgl Amdahl's  $T_p = \left(\frac{\alpha}{p} + (1-\alpha)\right) \cdot T_1$  )
- Analyse van  $W$  en  $S$ : ① klassieke technieken uit DataStructuren, inc Master Thm!  
② Voor de Span reken je parallelle takken maar 1x.
- Vaak succes met recursief ontwerp:  
Stel dat je het voor input  $n/2$  al kunt.  
Compressie: Verdicht je hele input tot  $n/2$  elementen.  
Split: Verwerk de twee helften van je input apart en combineer.
- Volgende week: Prefix Sum.  
Niet alleen som van alles maar ook alle deel-sommen opleveren.

in :	3	8	2	1	2	5	7
out :	3	11	13	14	16	21	28