

(Banská Bystrica 12 apr 2011)

H7 Parallel Sorteren

- ① Parallelle Complexiteit
- ② Flash & Merge Sort
- ③ Parallel Mergen
- ④ Wh2go Q?

- Vergelyk Scheduling Theorem $T_p \approx \frac{T_1}{p} + T_\infty$ met Amdahl.
- Grafiek $T_p = \frac{1}{p}$

① Parallelle Complexiteit

Stel dat je een taak hebt die serieel in T tijd kan.
Mooie wens. Parallel in $O(1)$ tijd met T (of meer!) processors.
We spreken van Superparallisme als $O(1)$ tijd haalbaar is.
Vb Search, Insert in gesorteerd array. (Lijst kan ook).

Aan voorbeeld van optellen zagen we. Voor vrijwel alle zinvolle taken is $\Omega(\lg n)$ tijd nodig: probeer daarom polylog.

Algoritme wordt gekarakteriseerd door span en work.

Doel van algoritme-ontwerp span is $O(\lg^c n)$: polylog.

Algoritme geeft een werk-overhead van $\frac{\text{work}}{T}$.

We noemen algo efficient als tijd en overh. polylog:

$$\text{span} = O(\lg^c n) \quad \text{work} = O(T \cdot \lg^d n)$$

We noemen het: optimaal als tijd polylog, zonder overhead:

$$\text{span} = O(\lg^c n) \quad \text{work} = O(T)$$

①

Voorbeelden van vorige keer.

Algo	Seq Tijd	span	work	overh	Klassificatie
Sommenen	n	$\lg n$	n	$O(1)$	Optimaal
Search	$\lg n$	$O(1)$	n	$n/\lg n$	Superpar, niet optimaal!
Insert	n	$O(1)$	n	$O(1)$	Optimaal.
Prefix Sum	n	$O(\lg n)$	n	$O(1)$	Optimaal.
FlashSort	$n \lg n$	$\lg n$	n^2	$n/\lg n$	"Te duur!"
C MergeSort	$n \lg n$	n	$n \lg n$	$O(1)$	"Te traag!"

Vandaag kijken we naar sorteren.

Input: array met n getallen (of andere geordende types)

Output: array met dezelfde getallen, oplopend ($i < j \Rightarrow A_i \leq A_j$)

Sequentiele tijd. $\Theta(n \lg n)$, we weten dat 't niet beter kan.

Algoritmen die dit halen: Heap, Merge, Quick Sort.

Werkning van MergeSort(1..n)

MergeSort (linker helft)

MergeSort (rechter helft)

Merge (links, rechts, tot een geheel)

kost $\Omega(n)$ werk
sequentieel.

② Flash en Klassieke Merge Sort

Het is niet zo moeilijk om snel te werken met een overkill aan processors. Neem n^2 processors p_{ij}

Flash Sort

1 Stap voor p_{ij} : $C_{ij} = 1$ als $A_i \leq A_j$
0 anders.

2 Optellen: $L_j = \sum_i C_{ij}$: het aantal getallen $\leq A_j$
dus: RANG van A_j !

3 Verplaatsen p_{ij} : $b = A_j$; $A_{L_j} = b$

Hier kost het Vergelyken en Optellen $2(n^2)$ werk.

span $\lg n$

work n^2

overh $\frac{n}{\lg n}$: Te veel werk!

Kan het sneller? MAX/MIN vinden wel!
Maar brekke in + middelen niet.

Analyse van Merge Sort: Twee Rec. aan roepen gaan parallel!

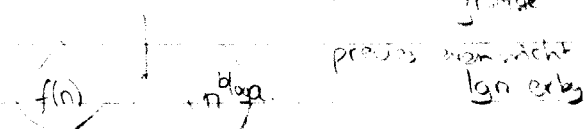
$$\text{span}(n) = \text{span}(n/2) + n \rightarrow \text{span}(n) = O(n)$$

$$\text{work}(n) = 2 \cdot \text{work}(n/2) + n \rightarrow \text{work}(n) = O(n \lg n)$$

$$\text{overh} = c$$

Master Theorem polynomiële doorlopen tijds

Het schiet niet erg op!

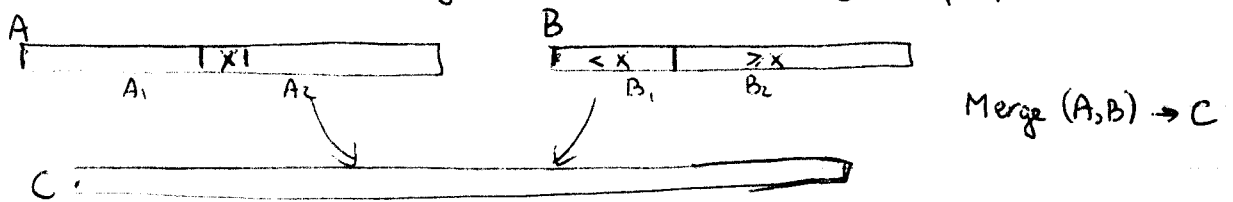


Het algoritme doet weinig anders dan mergen, want de twee Recursieve aanroepen worden ook helemaal in Merge's omgezet.

③ Parallel Mergen

Kunnen we sneller mergen?

A en B zijn oplopend



Je zou van voren en van achteren tegelijk kunnen mergen (EVEN NADENKEN!)

Helaas win je daar mee maar een constante factor 2.

We willen wel een idee om het mergen in taken te verdelen, maar dan zo, dat het recursief toepasbaar is.

Stel dat ik weet, waar het middelste element, x , van A, past in B: dwz x verdeelt A in A_1 A_2 , B in B_1 B_2 .

Dan kan ik separaat A_1 met B_1 mergen en A_2 met B_2 .

Waarheen? Beginstuk $A_1 \& B_1$: 1^e plaats in C

Beginstuk $A_2 \& B_2$: NIET "HALVERWEGE".

x komt op pos $\#A_1 + \#B_1 + 1$

Het lastige is, dat weliswaar A gelijk verdeeld is over A_1 en A_2 , maar B is ongebalanceerd verdeeld.

Dus, bv $(100, 100) \rightarrow (50, 5) \& (50, 95)$

Daarom pas ik bovenstaande truuks toe, waarbij ik altijd begin, het midden te pakken uit de grootste array!

Dus: $(50, 5) \rightarrow (25, \dots) \& (25, \dots)$

$(50, 95) \rightarrow (\dots, 47) \& (\dots, 48)$

Het totaal aantal te mergen elementen daalt nu met een factor $\frac{3}{4}$ of beter.

By het verdelen van 2 segmenten met ^{by elkaar} n elementen, krijgen de linker en rechter

merge-opdracht, (samen n elementen)

ieder hoogstens $\frac{3}{4}n$ elementen.

Bewijs. Je verdeelt het grootste segment exact; dat is minstens de helft, dus elke deelopdracht krijgt minstens een kwart. \square

begin
positie lengte
 ↓ ↙

```

PMerge (A, p1, n1, B, p2, n2, C, p3)
  if (n1 == 0) { Copy B ; return; } // Niks te mergen
  if (n2 == 0) { Copy A ; return; } // als een stuk leeg
  if (n1 >= n2)
  {
    q1 = p1 +  $\frac{n_1}{2}$  // Midden positie A
    x = A[q1]
    q2 = Search(x, B, p2, n2)
    q3 = p3 + (q1 - p1) + (q2 - p2) // Nieuwe pos. X
    tegelyk: PMerge(A, p1, (q1 - p1), B, p2, (q2 - p2), C, p3)
             & PMerge(A, q1 + 1, n1 - q1 - 1, B, q2, n2 - q2,
                       C, q3 + 1)
  }
  else
  { Het zelfde, maar dan rollen van A en B verwisseld }
  
```

Het verdelen van de merge in twee merges kost nu

- $\lg n$ tyd en werk als je binair zoekt.

Dit levert voor de complete merge: $(\lg n)^2$ tyd en n werk

Voor de complete merge sort: Werk = $n \lg n$
 span = $(\lg n)^3$.

- Wat gebeurt er mer Flash Search? Die heeft span = 1 werk = n .
 De complete merge kost dan $(\lg n)$ tyd en $n \lg n$ werk.
 Merge Sort komt totaal op $(\lg n)^2$ tyd en $n(\lg n)^2$ werk.

Je hebt een extra factor $\lg n$ werk (overhead) om $\lg n$ tyd te besparen.

	span	work	overhead	
PMerge - Bin	$(\lg n)^3$	$n \lg n$	1	Optimaal
PMerge - FI	$(\lg n)^2$	$n(\lg n)^2$	$\lg n$	Efficient
PQuick	$(\lg n)^2$	$n \lg n$	1	Optimaal.

④ What happened to Good Old QuickSort?

Het is in 2011, 50 jaar geleden dat Hoare QuickSort heeft uitgevonden. De structuur lijkt een beetje op MergeSort, maar ipv "lineair samenvoeg NA rec. L en R" heb je "lineair uiteenrafel VOOR rec L en R".

Om een segment, zeg van $A[p..q]$ te sorteren:

- Kies random 1 element als pivot
- Vergelijk elk element met de pivot, en plaats kleinere vooraan en grotere achteraan, pivot ertussen.
- Sorteer recursief Linker- en Rechterdeel.

Waarom is QuickSort zo leuk?

- ① Snelst bekende algo (sequentieel), in situ
- ② "Boeiende" (sic) analyse omdat L en R niet perse even groot zijn.

Na veel rekenen blijkt dat niets uit te maken: $O(n \lg n)$

De Partitie-stap gaat doorgaans sequentieel. Als je de twee deelaanroepen daarna parallel doet, krijg je tyd:
 $\text{span}(n) = O(n) + \text{span}(n/2)$

en dat geeft lineaire tyd!

Alleen aan het eind, als er al veel stukjes zijn, loopt het parallellisme lekker op.

QuickSort is optimaal te parallelliseren tot een span van $(\lg n)^2$ (!) door de partitie parallel te doen. Het is dan niet meer in-situ.

Partitie met $n/\lg n$ parallellisme

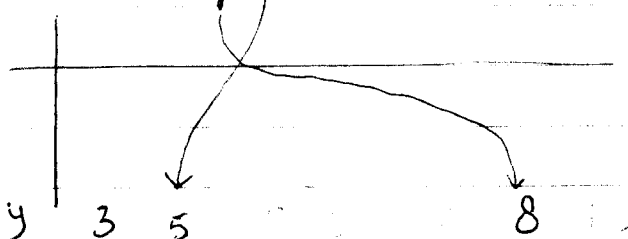
- ① Vergelyk elke x_i met pivot, (7)
 zet $k_i = 1$ als kleiner en
 $g_i = 1$ als groter.

x	3	8	5	2	7	10	5
k	1		1	1			1
g		1				1	

- ② Doe een Prefix-Sum op k en een Suffix Sum op g

k:	1	1	2	3	3	3	4
g:	2	2	1	1	1	1	0

- ③ Verplaats een "kleine x " naar $A[p + k_i - 1]$, grote naar $A[q - k_i + 1]$



- ④ Vul gat op met pivot

Met parallele Prefix Sum kost dit span $O(\lg n)$
 en werk $O(n)$.

Als je dit op $\lg n$ recursie niveaus doet krijg je
 Totale span $(\lg n)^2$
 werk $n \lg n$.

Beperkt parallelle Partition

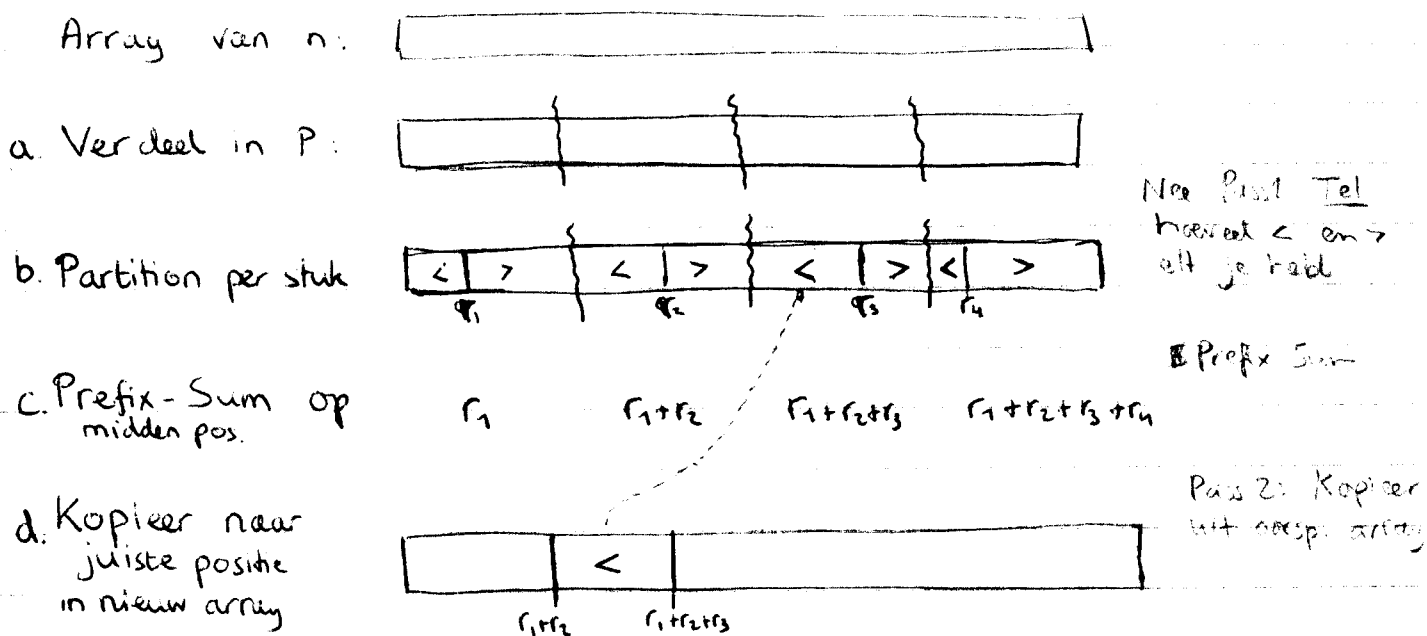
Meestal is je aantal processors p veel kleiner dan n of zelfs $n/\lg n$.

Bv nVidia kaart heeft 128 cores, voor miljoen getallen is $n/\lg n = 50000$.

Uitvoering van Tsigas:

Als je P cores beschikbaar hebt om een stuk van n te partitioneren, verdeelt je array dan eerst in P stukken van n/p lang.

Iedere processor doet een "normale" partitie van 1 stuk.



a is een "impliciet" stap, b en d kosten n werk in n/p tyd.
c kost P werk in $\lg P$ tyd.

Dus afgezien van de $\lg P$ tyd voor het berekenen van de Prefix Sum, is het lineaire werk nu perfect verdeeld over n/p voor elk van de P processors.