

Work-Span 2 : Prefix Sum

- ① Inleiding
- ② Split aanpak
- ③ Compressie aanpak
- ④ Analyse
- ⑤ Toepassing: Compaction
- ⑥ Toepassing: Addition
- ⑦ Conclusies

Snuffel eens op Wikipedia onder PrefixSum en de verwijzingen daarin!

① Inleiding

Gegeven een rij A van n getallen (waarden).
Het gaat (weer) om optellen, maar je moet niet alleen het totaal van alles opleveren maar ook de som van elke prefix:

Vb input:	1	2	3	4	5	6	7	8	9
Verkoop per week	3	4	4	4	4	3	5	4	5

output:									
verkoop t/m week	3	7	11	15	19	22	27	31	36

- Vaak wordt de uitvoer in dezelfde array berekend.
- Je kunt 't ook een "cumulatief" noemen.
- Variant: Prefix Product / Conjunctie / Compositie Suffix, 0- / 1-based array.
- Sequentieel uitrekenen kan heel simpel in één sweep van links naar rechts met exact $n-1$ optellingen, die in strikte volgorde plaatsvinden!
- Het is ook het pattern "Scan" uit Structured Parallel Programming

```
for (i=2; i<n; i++)  
    A[i] += A[i-1]  
Seq. Prefix Sum
```

② Split aanpak

We hebben weer succes met een recursieve aanpak: dus neem aan dat je een PS op $n/2$ elementen "al kunt".

Wat krijg ik als ik dat gewoon eens doe voor de linker- en rechterhelft van de data?

in :	3	4	4	4	4		3	5	4	5
(doel :	3	7	11	15	19		22	27	31	36)
na PS-L en PS-R :	3	7	11	15	19		3	8	12	17

Dit lijkt op hoe het moet worden, maar de outputs Rechts moeten nog wel even allemaal met 19 worden verhoogd.

Die 19 is het laatste getal van Links en is inmiddels al uitgerekend.

Prefix Sum (A, p, q)

if (q == p) return ;

(if (q == p+1) { $A_q += A_p$; return ; })

$m = (q+p) / 2$

Invoke (PrefixSum (A, p, m)

PrefixSum (A, m+1, q))

for (i, m+1, q) $A_i += A_m$

p en q zijn
inclusieve
grenzen van
het segment

Deze is wel nodig,
maar sla 'm op
hoorcollege even over

Dit is wel snel en dat tegen niet te hoge maar toch wel enige kosten.

Je kunt (op Werk College) de Work en Span uitrekenen en dan blijkt (alles asymptotisch)

$$\text{Span}(n) = \lg n$$

$$\text{Work}(n) = n \lg n.$$

Een betere Span kunnen we niet verwachten dus $S = \lg n$ is heel goed.

De (parallelliserings-) overhead van een algoritme is het Work gedeeld door de kosten van het (beste) sequentiele algoritme.

Omdat we hier $n \lg n$ werk steken in iets dat sequentieel lineair kan, is er sprake van logaritmische overhead.

Definitie Een parallel algoritme is

- Efficient als de Span polylogaritmisch en de Overhead polylogaritmisch is
- Optimaal als de Span polylogaritmisch en de Overhead constant is.

Terug naar bovenste zin:

snel \rightarrow polylog span (hier $(\lg n)^1$)

enige kosten \rightarrow er is niet-constante overhead $\lg n$

niet te hoog \rightarrow overhead is polylog.

③ Compressie - aanpak

Maak combinaties van paren en bereken daarvan PS.

in	3	4	4	4	4	3	5	4	5
(doel	3	7	11	15	19	22	27	31	36)
paren	7		8		7		9		
hiervan de PS	7		15		22		31		

De gewenste antwoorden van de even posities zijn al bekend!

Daaraan dus niets meer doen, alleen de oneven posities moeten nog worden ingevuld.

Daar gewoon de invoerwaarde nog optellen bij de PS-uitvoer links ervan.

De oplossing draait nog steeds rondom "de linkerbuur erby tellen" ($A_i += A_{i-1}$) maar ipv een sequentie, gaat het nu in twee groepen (even i , oneven i) met daartussen een PS over de even posities:

① parallel, even i : $A_i += A_{i-1}$

② PrefixSum op even posities

③ parallel, oneven i : $A_i += A_{i-1}$

(Ik vind 'm briljant in z'n eenvoud en elegantie.)

Iets over het programmeren.

Hoe maak je de PS over "even posities"?

Je kunt de paar-sommen uit stap ① in een aparte array B zetten. Maar dan moet je die (in stap ③) ook weer terug zetten:

$$\textcircled{1} \quad \text{pfor}(i \dots) \quad B_i = A_{2i} + A_{2i-1}$$

$$\textcircled{2} \quad \text{PrefixSum}(B \dots)$$

$$\textcircled{3} \quad \text{pfor}(i \dots) \quad A_{2i} = B_i \\ A_{2i+1} += A_{2i}$$

Mooier en efficiënter: geef aan PrefixSum een h ("hop") parameter mee die de actie beperkt tot de "deelry" van veelvouden van h:

PrefixSum(A, h)

if (--) Randvoorwaarden, base-case, thresholds.

$$\textcircled{1} \quad \text{pfor}(i \text{ is evenveelvoud van } h) \\ A_i += A_{i-h}$$

$$\textcircled{2} \quad \text{PrefixSum}(A, 2 \times h)$$

$$\textcircled{3} \quad \text{pfor}(i \text{ is oneven veelvoud van } h) \\ A_i += A_{i-h}$$

④ Analyse

Het werkt omdat optelling associatief is.
De sequentiële berekening voor plek 4 is:

$$((A_1 + A_2) + A_3) + A_4.$$

De compressie-recursieve aanpak doet

$$(A_1 + A_2) + (A_3 + A_4)$$

en dat is 't zelfde.

Associatief zegt dat je haakjes mag verzetten:

$$(a+b)+c = a+(b+c)$$

Commutatief zegt dat je volgorde mag veranderen: $a+b = b+a$.

Optellen is wel commutatief, maar voor de Parallele PrefixSum is dat niet relevant, want het is niet nodig.

Functie-compositie is associatief:

$$(f \circ g) \circ h = f \circ (g \circ h)$$

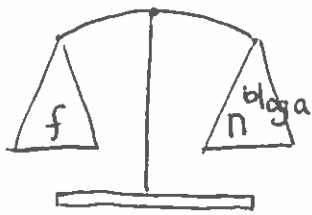
maar niet commutatief $f \circ g = g \circ f$.

Vanwege de associativiteit werkt de parallelle PS voor functiecompositie WEL.

Nou ff de complexiteit uitrekenen:

Work Het algoritme doet $n-1$ optellingen,
èn 1x een PrefixSum van halve omvang.
Betrekking: $W(n) = n + W(n/2)$

Master Thm erby, $a=1$, $b=2$: ${}^b\log a = 0$



$f(n) = n^1$ dus poly-meer dan n^0

De poly-weegschaal slaat
door naar f ,
uitkomst $W(n) = n^1 = n$.

Span De vele optellingen worden in 2 groepen
gedaan, 1 voor en 1 na de sub-PS.

Betrekking: $S(n) = 1 + S(n/2) + 1$

Weer is $a=1$ en $b=2$ dus ${}^b\log a = 0$.

Maar f is nu constant dus n^0 .

De weegschaal is in evenwicht dus extra \lg .

Uitkomst $S(n) = \lg n$.

Er is constante (dwz: "geen") overhead en toch
logaritmische span.

Dit algoritme is een optimale parallellisering
van de PrefixSum-berekening!

⑤ Toepassing: Array Compaction

Gegeven een array A met n elementen. Je wilt een kortere array B maken met alleen die elementen die aan een bepaalde eigenschap voldoen.

Voorbeeld hier: even-selectie

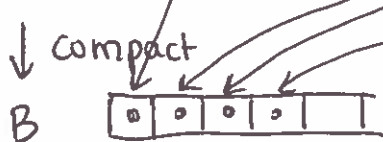
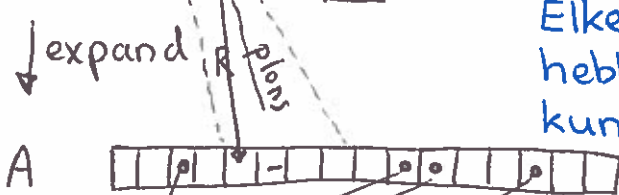
in A	11	28	01	88	32	41	41	07	53	82	64	82
uit B	28	88	32	82	64	82						

Je weet niet van tevoren hoeveel ruimte nodig is!

Serieuze toepassing:

C is een ry Caveman-states (in k stappen bereikbaar)

In een parallelle expansie maak je een 4×20 lange A die alleen de bereikbare, nieuwe states bevat. Voordat je met de volgende ronde begint, moet je A weer compacten.



Elke state kan vier opvolgers hebben (LRUD), maar sommige kunnen misschien niet (code plons) of zijn al eerder geweest

Dit is maar een voorbeeld: in Open CL kun je dit simpeler oplossen en heb je geen Compaction nodig! Wel by Path Tracing: shadow rays moet je elimineren en dan Compacten

Compaction kan sequentieel weer heel simpel met een lineair proces:

```
t=0;  
for (i=0; i<A.len; i++)  
    if (even(Ai)) Bt++ = Ai
```

Maar helaas wel weer erg inherent sequentieel.

Met PrefixSum kun je, per element, snel uitrekenen waar hij moet komen

```
Compact(A, B, even)  
  pfor (i, 0, A.len)  
    Ki = (even(Ai) ? 1 : 0)
```

```
PrefixSum(K --)
```

```
// Voor even Ai geldt nu:  
// het is de (Ki)e vanaf het begin!
```

```
pfor (i, 0, A.len)  
    if (even(Ai)) BKi = Ai
```

Compaction kan dus mbv PrefixSum,
constante extra Span, (testen & verplaatsen)
lineair veel extra Work (testen & verplaatsen)
lineair veel extra geheugen. (K)

⑤ Toepassing: Addition

Voor processorfabrikanten is het een hele kunst om de ADD instructie in 1 clock cycle voor elkaar te krijgen. Het gaat om het binair optellen van twee k -bits binaire getallen.

A is bv 13

B is bv 25

$k-1$	$k-2$	2	1	0
0	0	1	1	0
0	1	1	0	0

geeft S

1	0	0	1	1	0
---	---	---	---	---	---

De k kolommen zijn genummerd $k-1$ t/m 0 en een 1 in kolom i "representeert" waarde 2^i .

In kolom i kan een "carry" ontstaan (nl als het totaal daar meer dan 1 is) en die carry wordt meegeteld in kolom $i+1$.

Per kolom werkt het zo:

- tel het aantal enen in $\{A_i, B_i, C_{i-1}\}$
- S_i is de pariteit van dit aantal
- C_i is 1 als dit aantal ≥ 2 is, anders 0.

Optellen is weer redelijk makkelijk te doen, maar als ik hiervan een binair circuit maak (netwerk van transistortjes) heeft dat lineaire diepte!

Dus een 64-bits optelling in 0,33 ns (3GHz kloktik) gaat zo niet lukken!

Carry Prediction

Door Carry Prediction is de Add uit te drukken in een binair circuit van logaritmische diepte. In lg k tijd berekent elke kolom, welke carry hy geeft aan de volgende kolom.

Carry kolom functie.

Hoe hangt de carry C_i die kolom i geeft, af van de vorige carry C_{i-1} ?

- Als kolom i twee enen bevat ($A_i = B_i = 1$) dan geeft hy zeker een 1, dus de carry functie is de constante functie $E(C_{i-1})$ ($E(0) = E(1) = 1$)
- Als kolom i twee nullen bevat is de carry zeker een 0 dus $C_i = N(C_{i-1})$ (waar $N(0) = N(1) = 0$)
- Als de kolom een 0 en een 1 bevat, is de carry C_i gelijk aan C_{i-1} , een Copy-kolom $C_i = I(C_{i-1})$, $I(0) = 0$ $I(1) = 1$.

Het voorbeeld

A	0	0	0	1	1	0	1
B	0	0	1	1	0	0	1

geeft dus deze

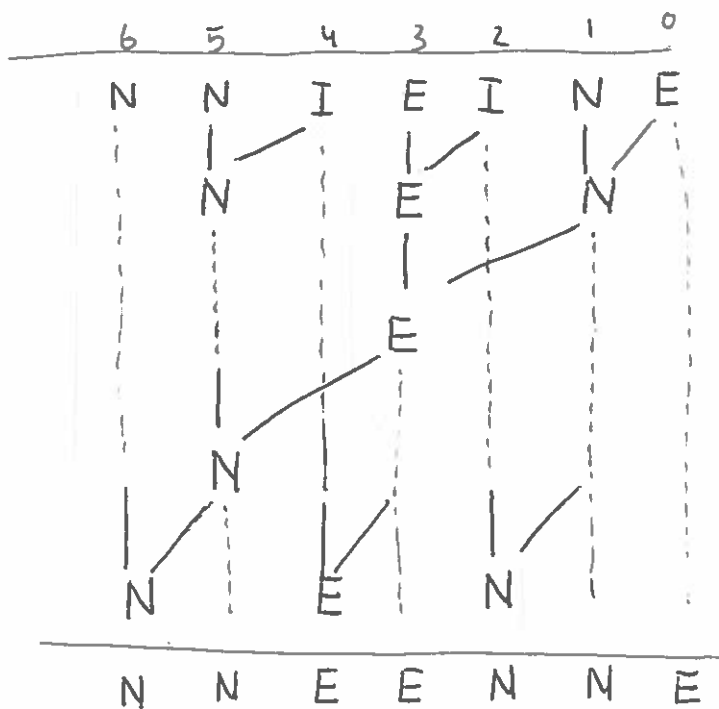
	6	5	4	3	2	1	0	
reeks carry-functies	N	N	N	I	E	I	N	E

Omdat kolom 0 de carry 0 "krygt", kun je nu de carry C_i van kolom i naar $i+1$ uit drukken als $C_i = f_i \circ f_{i-1} \dots \dots \circ f_1 \circ f_0 (0)$

Elke kolomfunctie kun je met 2 bits coderen en je kunt de functies samenstellen volgens deze tabel.

De suffix-composities zijn met het PS-algoritme te berekenen.

$f \circ g$	N	I	E
N	N	N	N
I	N	I	E
E	E	E	E



Hier is de k vrij klein (7) en het verschil tussen $k-1$ en 2 lgt valt niet zo op. Maar voor $k=64$ vervang je 64-diep door 12-diep!

De uitkomst kan aan de rechterkant een paar I's hebben; die kolommen geven dan carry 0.

⑦ Conclusies

Doel by paralleliseren is

- lage span, liefst polylogaritmisch
- weinig work, liefst niet meer dan het sequentieel kost, als dat niet lukt: polylog overhead.

Als dit lukt is het algoritme efficient of zelfs optimaal.

Prefix Sum:

- Leuke case
- Handige toepassingen: Compaction

Omdat een optelling heel snel is, is Prefix Sum vrijwel altijd "Memory Bound". Ondanks alle slimmigheid is het sequentiële algoritme op een PC nauwelijks te verslaan!