je to alternativní model k paralelnímu Turingovu stroji

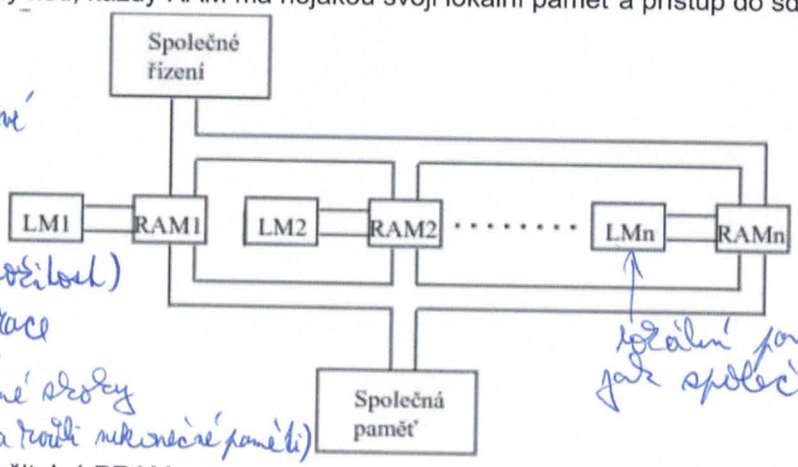
skládá se z p procesorů RAM

# 56. Model PRAM, suma prefixů a její aplikace.

Model je rozhraní, které odděluje aplikaci (high-level) od architektury (low-level). Model poskytuje aplikaci operace prostřednictvím nějakého rozhraní a po architektuře požaduje implementaci těchto operací. Model PRAM (Parallel Random Access Machine) se vyznačuje následujícím:

- synchronní model paralelního výpočtu, ve kterém několik procesorů komunikuje pomocí sdílené paměti
- sdílená paměť se skládá z neomezeného počtu registrů
- všechny procesory (model RAM) jsou řízeny jedním společným programem a vykonávají tedy stejný kód, každý RAM má nějakou svoji lokální paměť a přístup do sdílené paměti

Procesor
- chceme, aby jednotlivé operace procesoru měly jednotkovou cenu (konstantní čas. složitost)
- aditivní (logické) operace
- multiplikativní operace
- podmíněné a nepodmíněné skoky
- adresování (není složka režie nekonečné paměti)




lokální paměť, menší latence než společná paměť ⇒ NUMA

Každý problém řešitelný PRAMem s p procesory v t krocích je také řešitelný s p' <= p procesory v O(t p/p') krocích. Abychom takového chování dosáhli, rozdělíme původních p procesorů do p' skupin rovnoměrně a každý z p' procesorů pak simuluje chování své skupiny procesorů.

- při simulaci musí provést nejprve pro všechny virt. proc. fáze čtení, pak lokální operace a až pak pro všechny nejvýše jeden zápis do SM

**Souběžný přístup ke sdílené paměti** může být modelem omezen, rozlišujeme čtyři různé architektury přístupu:

- **EREW** – exclusive read, exclusive write
- ERCW – exclusive read, concurrent write – nepoužívá se, nemá opodstatnění (souběžný zápis je náročnější zajistit než souběžné čtení)
- **CREW** – concurrent read, exclusive write
- **CRCW** – concurrent read, concurrent write – technicky obtížně realizovatelný

U architektur CRCW rozlišujeme ještě architektury dle způsobu řešení zápisových konfliktů:

- COMMON – všechny zapisované hodnoty musí být shodné
- ARBITRARY – zapisované hodnoty mohou být různé, zapíše se libovolná z nich
- PRIORITY – procesory mají pevné priority, zapíše se hodnota zapisovaná procesorem s nejvyšší prioritou

Procesory mohou během kroku provádět různé operace a mohou používat svůj index.

Definice PRAMu

PRAM je synchronní model paralelního výpočtu, ve kterém procesory komunikují sdílenou pamětí. Skládá se z p procesorů, které jsou RAM. Výpočet probíhá po krocích synchronně: ① čtení sdílené paměti ② lokální operace ③ zápis do sdílené paměti

Zavádíme relaci ≥ s interpretací $A \geq B$ znamená, že algoritmus, který běží na architektuře B, běží beze změn i na architektuře A. Platí hierarchie PRIORITY ≥ ARBITRARY ≥ COMMON ≥ CREW PRAM ≥ EREW PRAM.

Příklad algoritmus BROADCAST – chceme rozšířit jednu hodnotu mezi všechny procesory. Na CRCW a CREW triviální – každý procesor si načte hodnotu ze sdílené paměti v konstantním čase. Na EREW je třeba simulovat současné čtení, např. tak, že procesor P1 pošle hodnotu P2, v druhém kroce P1 pošle hodnotu P3, P2 pošle hodnotu P4, ve třetím kroce P1 pošle hodnotu P5, P2 pošle hodnotu P6, P3 pošle hodnotu P7, P4 pošle hodnotu P8, …

# Suma prefixů

Suma prefixů je operace, jejímž vstupem je binární asociativní operátor ⊕ a uspořádáná posloupnost $[a_0, a_1, \ldots, a_n]$ a výstupem je posloupnost $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_n)]$. Například pro operátor sčítání a posloupnost [1, 2, 3, 4] je výstupem sumy prefixů [1, 3, 6, 10]. Mnoho problémů jde řešit pomocí sumy prefixů (viz např. parallel splitting u algoritmu select). Suma prefixů je v knihovně MPI implementována ve funkci MPI_scan. Operátorem může být libovolný asociativní operátor (tzn. $(a \oplus b) \oplus c = a \oplus (b \oplus c)$), např. sčítání, násobení, minimum, maximum, and, or, xor, konkatenace řetězce, násobení matic.

Rozlišujeme 3 typy operací, všechny svým způsobem počítají sumu prefixů:

- scan – suma prefixů přesně jak byla definována výše (tzv. inclusive scan)
- prescan – suma prefixů, ale na i-tou pozici není započítán i-tý prvek. Výstupem je tedy posloupnost $[I, a_0, (a_0 \oplus a_1), \ldots (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$, kde I je neutrální prvek operace.
- reduce – výstupem je poslední hodnota operace scan, tzn. výsledek operace ⊕ aplikované na celou posloupnost

neutrální prvek

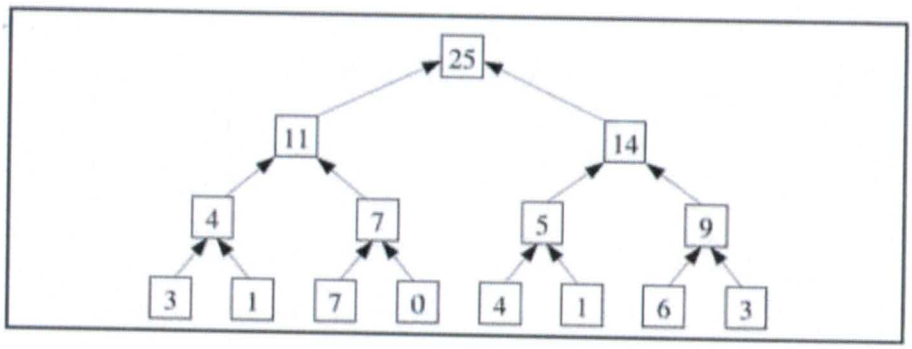
SCAN, ALLSUMS

I, a0, a0⊕a1, ....., a0⊕a1⊕ ...⊕an-2, a0⊕a1⊕ ...⊕an-1

PRESCAN REDUCE

poslední prvek

Operaci reduce můžeme efektivně paralelně spočítat pomocí stromu procesorů:




(a) Executing a +-reduce on a tree.

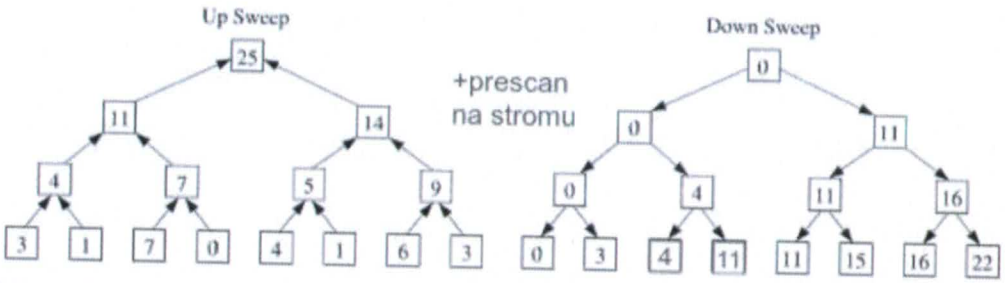
**for** $d$ **from** 0 **to** $(\lg n) - 1$
  **in parallel for** $i$ **from** 0 **to** $n - 1$ **by** $2^{d+1}$
    $a[i + 2^{d+1} - 1] \leftarrow a[i + 2^d - 1] + a[i + 2^{d+1} - 1]$

| Step | Vector in Memory | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | [ **3** | **1** | **7** | **0** | **4** | **1** | **6** | **3** ] |
| 1 | [ 3 | **4** | 7 | **7** | 4 | **5** | 6 | **9** ] |
| 2 | [ 3 | 4 | 7 | **11** | 4 | 5 | 6 | **14** ] |
| 3 | [ 3 | 4 | 7 | 11 | 4 | 5 | 6 | **25** ] |

Strom má výšku log n, časová složitost je tedy O(log n). Počet procesorů je ovšem lineární, tzn. cena je O(n log n), což není optimální (triviální sekvenční algoritmus má lineární časovou složitost). Lepší cenu dosáhneme tím, že procesorů bude méně než je počet prvků, tzn. každý procesor bude zpracovávat několik čísel. Nejprve provede každý procesor operaci reduce pomocí sekvenčního optimálního algoritmu na svoji podsekvenci, a pak probíhá reduce po stromě stejně jako výše. Pro N procesorů dostáváme složitost O(n/N + log N). Pokud je log N < n/N, pak je časová složitost O(n/N) a tedy cena O(n), což je optimální.

Operace reduce se také nazývá upsweep – posouváme hodnoty nahoru stromem. Pro výpočet pre-scan se nejprve spočítá operace reduce a potom se provede operace downsweep, která funguje následovně:

- kořenu se přiřadí neutrální prvek operace
- provádí se log n kroků, počínaje kořenem, směrem k listům (po úrovním). Uzel dá svému pravému synovi (*svoji hodnotu* $\oplus$ *hodnotu levého syna*) a levému synovi dá svoji hodnotu.



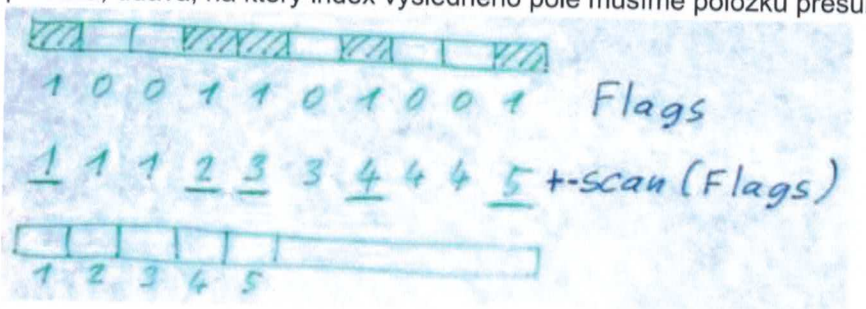

*(Ukázka Down Sweep je demonstrována na obrázku. Vstupní konfigurací byl strom ukázaný vlevo.)*

Pokud bychom chtěli inclusive scan, namísto exclusive scan, odstraníme první prvek (neutrální prvek) a na konec přidáme hodnotu reduce. Časová složitost downsweep je stejná jako u upsweep, opět provádíme operace na stromě po úrovních – O(n/N), cena je tedy O(n), což je optimální.

## Aplikace sumy prefixů

### Packing problem

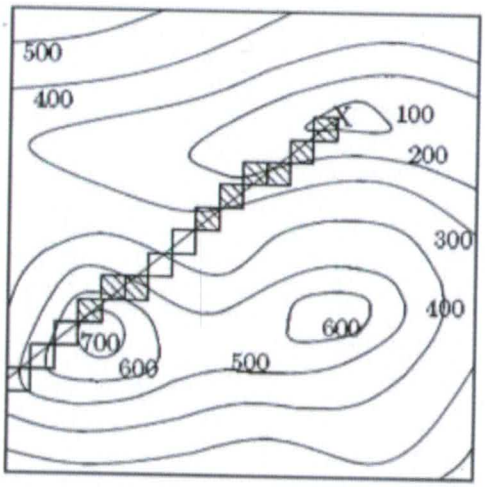
Máme k dispozici k položek rozmístěných v poli o velikosti n, k < n, tzn. v poli jsou prázdná místa. Chceme prvky smrsknout do prvních k pozic. Přiřadíme hodnotu 1 prvkům pole, které mají položku, 0 ostatním. Následně spočítáme +-scan, jehož výsledek pro prvek, na kterém je položka, udává, na který index výsledného pole musíme položku přesunout:

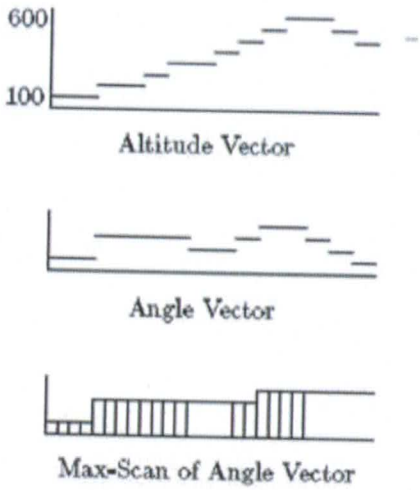



### Viditelnost

Je dána matice terénu ve formě matice nadmořských výšek a pozorovací bod X (místo pozorovatele). Chceme určit, které body jsou viditelné z místa X. Princip řešení je následující:

- určíme vektor výšek bodů podél pozorovacího paprsku z místa X
- převedeme vektor výšek na vektor úhlů
- spočítáme max-prescan. Pro zjištění viditelnosti srovnáme každý úhel s maximem – pokud je úhel větší než dosavadní maximum (spočtené v prescanu), pak je bod viditelný:



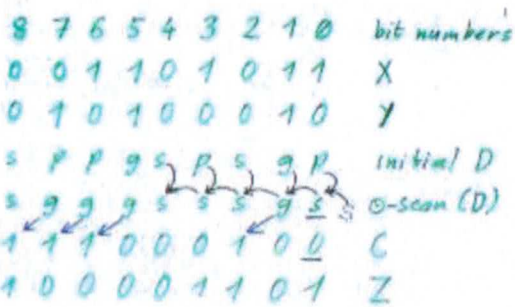

## Paralelní CLA (Carry-Look-Ahead) sčítačka

Principem je určit pole D, kde hodnoty jsou propagate (p – propagujeme přenos), stop (s – zastavíme přenos), generate (g – vygenerujeme přenos). Výpočet je jednoduchý, pro každý bit:

- pokud jsou oba bity 1, pak nastavíme generate
- pokud jsou oba bity 0, pak nastavíme stop
- jinak nastavíme propagate

Následně zadefinujeme nad p, s, g operaci a spočítáme její scan, abychom určili, jaké budou carry bity:

| $\odot$ | $s$ | $p$ | $g$ |
|---|---|---|---|
| $s$ | $s$ | $s$ | $s$ |
| $p$ | $s$ | $p$ | $g$ |
| $g$ | $g$ | $g$ | $g$ |




## Radix sort

Bitový radix sort (radix = 2) funguje následovně. V každém kroku se bere v úvahu 1 bit klíče a pomocí operace split se prvky s nulovým bitem přemístí na začátek pole a prvky s jedničkovým bitem na konec. Operaci split můžeme paralelizovat pomocí $\oplus$ –*prescanu* a to tak, že pomocí operací scan/prescan určíme správnou pozici prvku a v konstantním čase přemístíme.

$A(n) = O(n/N \cdot \log n + \log n \cdot \log N)$

## Quicksort

V rámci paralelizace funkce quicksort se využívá upravená suma prefixů, tzv. segmentovaný scan. Kromě klasických vstupů ještě přidáváme uspořádanou posloupnost příznaků (jedničky a

nuly), která rozděluje vstupní posloupnost na segmenty. V každém segmentu je suma prefixů počítána zvlášť:

- Příklad :
  - a = [5 1 3 4 3 9 2 6]
  - f = [1 0 1 0 0 0 1 0]
  - Segmented +_SCAN = [5 6 3 7 10 19 2 8]
  - segmented max_SCAN = [5 5 3 4 4 9 2 6]

# Quicksort

- Jeden z prvků se vybere jako pivot (medián, náhodně, první), prvky se rozdělí do 3 skupin (menší, rovné, větší než pivot) a pro každou skupinu se rekurzivně volá quicksort
- Použije se segmentovaný scan a každá skupina bude ve svém vlastním segmentu
- Algoritmus
  - (1) zkontroluj, zda už prvky nejsou seřazené. Každý procesor se podívá, zda předchozí procesor má menší, nebo stejnou hodnotu. S výsledky se provede and-reduce
  - (2) v každém segmentu najdi pivot a předej jej ostatním procesorům v segmentu. Vybírá-li se jako pivot 1. prvek, lze použít segmented-copy-scan, kde binární operátor copy vrací 1. ze svých 2 parametrů:
    - a ← copy(a, b)
    - To má za následek rozšíření pivota v celém segmentu (lze také pivota vybírat jinak)
  - (3) v každém segmentu porovnej prvky s pivotem a rozděl segment na 3 části (=, <, >). Po rozdělení se použije modifikovaný split z radix-sortu.
  - (4) v rámci každého segmentu vlož dodatečné příznaky, které rozdělí segment na 3 segmenty. Každý procesor se podívá na předchozí prvek a pozná, zda je na začátku segmentu.
  - (5) jdi na krok (1)

*Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele Fifinas.*