# PRAM

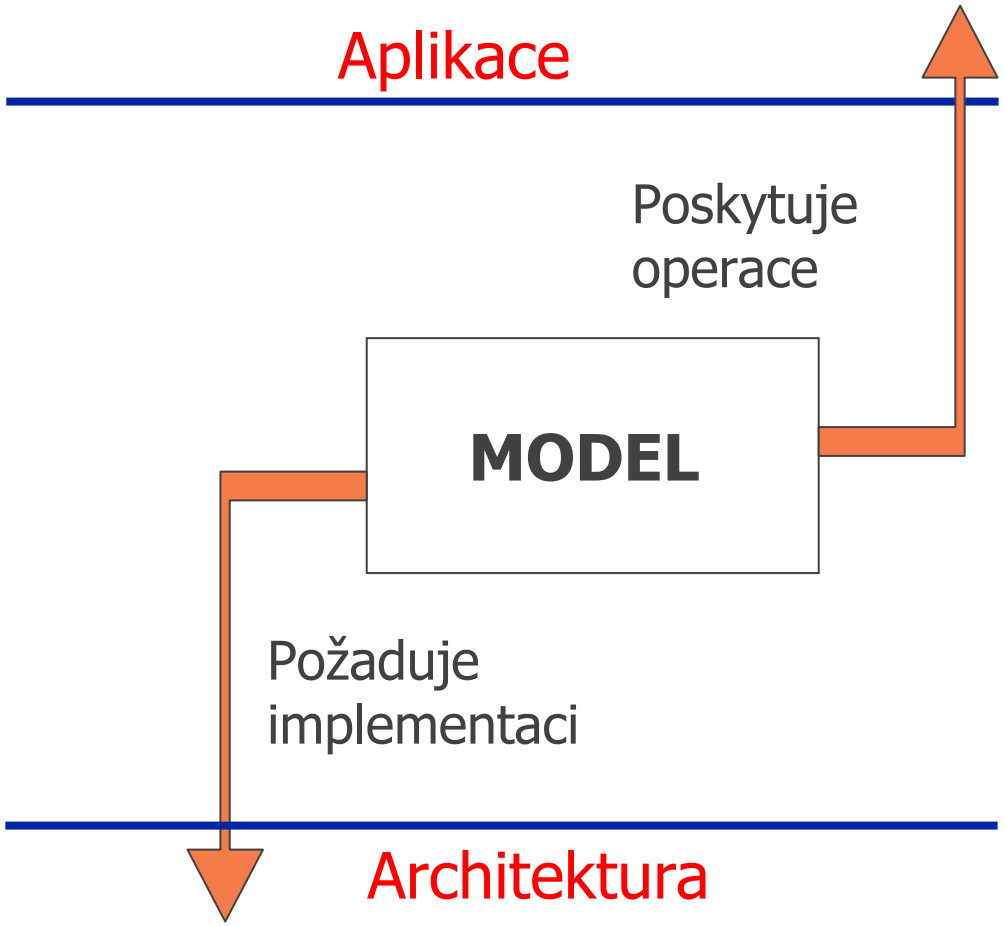
Petr Hanáček

Upd 2005, Upd 2007

Upd 2008/9

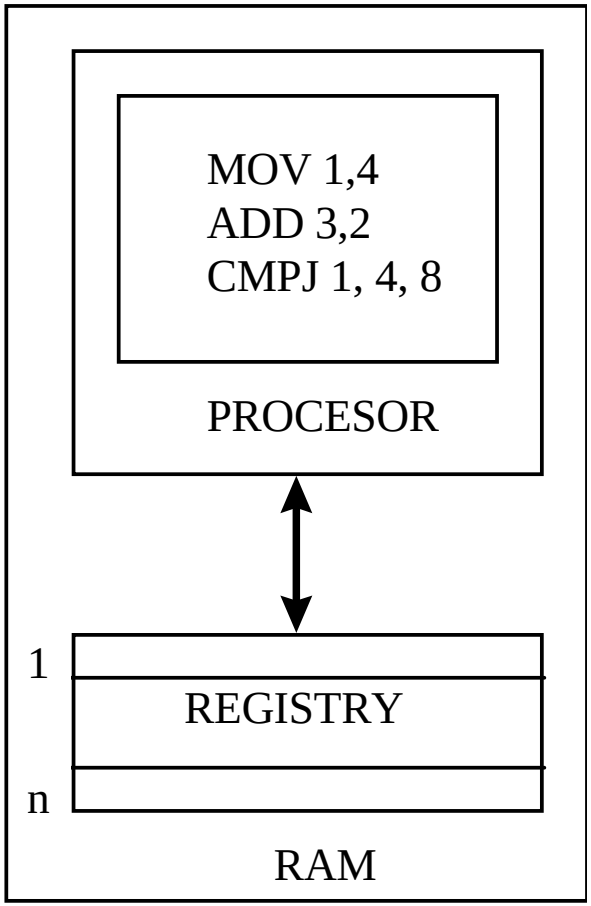
# Model

Model je rozhraní, které odděluje aplikaci (high-level) od architektury (low level)



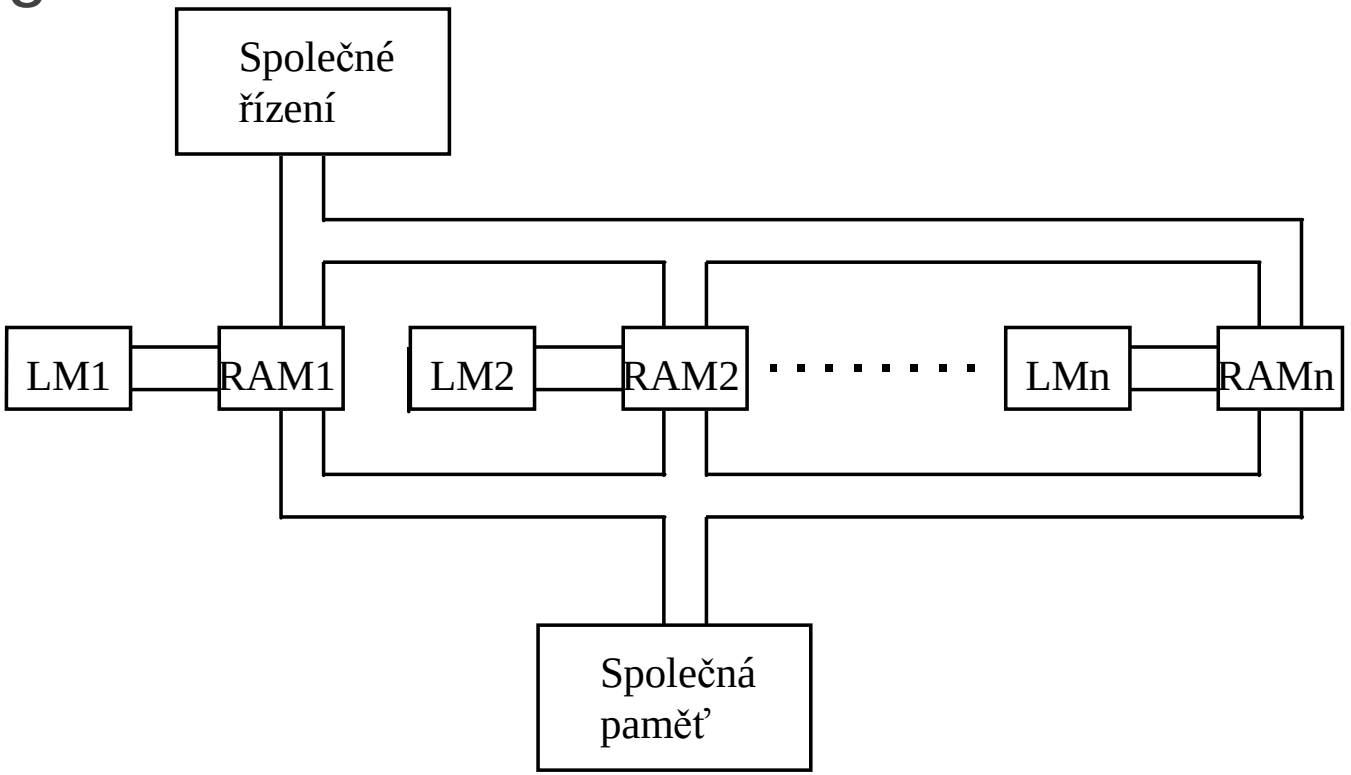
# PRAM

- Parallel Random Access Machine
  - *Synchronní* model paralelního výpočtu, procesory komunikují *sdílenou pamětí*
  - Skládá se z p procesorů RAM
- Procesor
  - Aditivní (logické) operace
  - [Multiplikativní operace]
  - Podmíněné skoky na základě porovnání
  - *[A*dresování*]*
- Paměť (sada registrů)
  - Neomezený počet
  - *[N*eomezená délka slova*]* - není příliš vhodná, už se nepoužívá



# PRAM

- Je to alternativní model k paralelnímu Turingovu stroji
- Všechny RAMy jsou řízeny jedním společným programem



- Definice
  - PRAM je synchronní model paralelního výpočtu, ve kterém procesory komunikují sdílenou pamětí. Skládá se z $p$ procesorů P1...Pn, jež jsou RAM.
  - Výpočet probíhá po krocích synchronně. Krok:
    - čtení sdílené paměti
    - lokální operace
    - zápis do sdílené paměti
  - Procesory mohou během kroku provádět různé operace a mohou používat svůj index (unikátní číslo procesoru).
- Teorém
  - Každý problém, řešitelné PRAMem s $p$ procesory v $t$ krocích je také řešitelné $p' \leq p$ procesory v $O(t\ p/p')$ krocích.
- Důkaz
  - Původních $p$ procesorů je rozděleno do $p'$ skupin o velikosti max $\lceil p/p' \rceil$. Každý z $p'$ procesorů simulujícího stroje se stará o jednu skupinu. Pro simulaci jednoho kroku původního stroje každý z $p'$ procesorů simuluje čtecí fázi procesorů své skupiny, pak lokální fázi a na konec zápisovou fázi.

# Omezení přístupu ke sdílené paměti

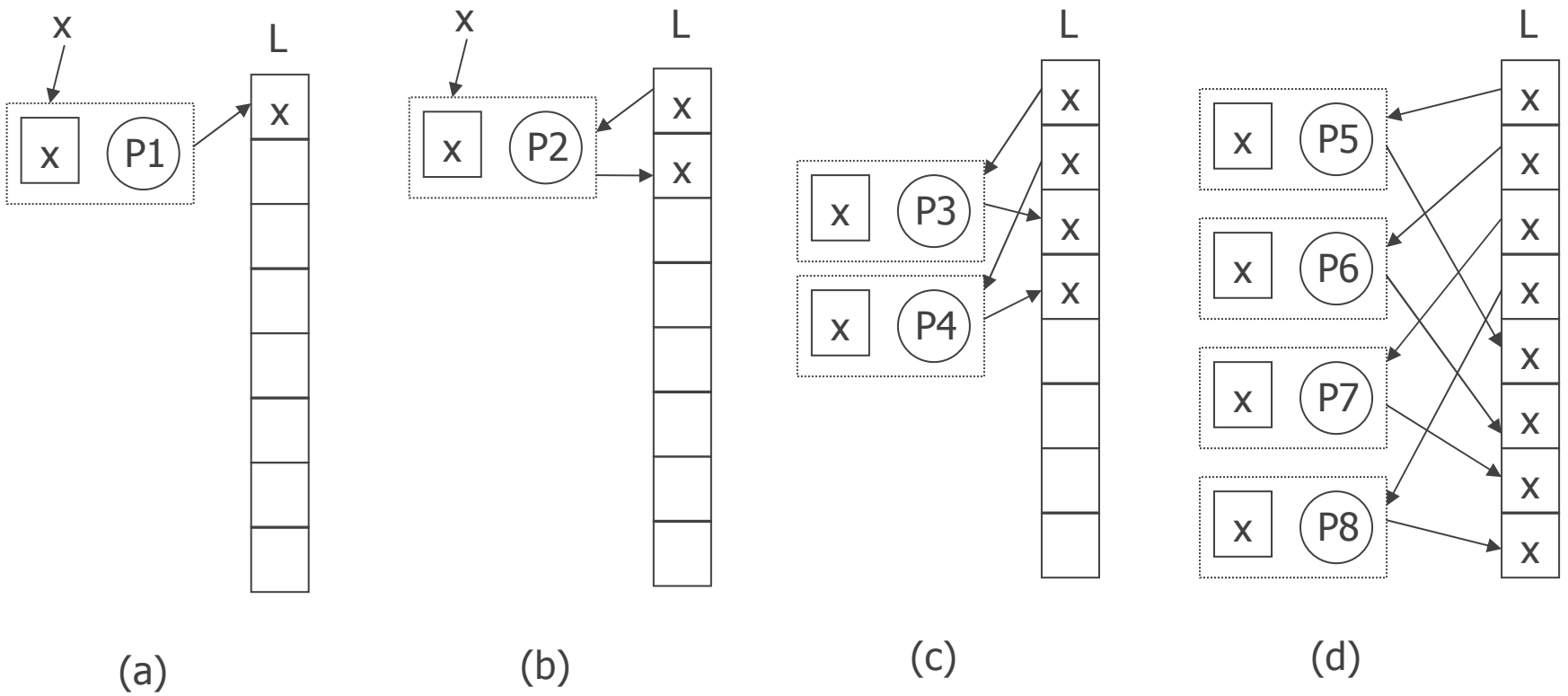
- Je dovoleno současné čtení ?
- Je dovoleno současné zapisování ?
- Čtyři různé architektury přístupu k paměti
  - » EREW - exclusive read, exclusive write
  - » ERCW - exclusive read, concurrent write - nemá opodstatnění, nepoužívá se
  - » CREW - concurrent read, exclusive write
  - » CRCW - concurrent read, concurrent write - splňují pouze některé architektury, technicky obtížněji realizovatelný
- U architektury CRCW je třeba specifikovat 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
  - – Relace A ≥ B - algoritmus, který běží na architektuře B, běží beze změn i na A. ( A je stejně tolerantní nebo tolerantnější k zápisovým konfliktům)
- Pak platí:
  - – PRIORITY ≥ ARBITRARY ≥ COMMON ≥ CREW PRAM ≥ EREW PRAM

# DALŠÍ ALGORITMY

# Broadcast

- Hodnota, uložená v paměti, má být rozšířena mezi N procesory
  - pro CREW a CRCW PRAM je triviální řešení v konstantním čase
  - pro EREW je třeba simulovat současné čtení
- Funkce
  - P1 přečte D a  zpřístupní jej P2.
  - P1 a P2 jej zpřístupní paralelně P3 a P4.
  - P1, P2, P3 a P4 jej zpřístupní paralelně P5, P6, P7 a P8..
  - …

```
Algoritmus
D - hodnota, která se má rozšířit mezi N procesory
A[1..N] - pole ve sdílené paměti o délce N
procedure BROADCAST(D, N, A)
(1) A[1] = D;
(2) for i = 0 to (log N-1) do
        for j = 2^(i+1) to 2^(i+2)-1 do in parallel
                A[j] = A[j-2^i]
        endfor
   endfor
```




(a) (b) (c) (d)

# Analýza

| | |
|---|---|
| SEQ | t(n) = O(n) |
| EREW | t(n) = O(log n) |
| CREW, CRCW | t(n) = O(c) |

# Suma prefixů

- All-prefix-sums, allsums, scan
- Jeden ze základních kamenů stavby paralelních algoritmů
- Definice
  - Suma prefixů je operace, jejímž vstupem je binární asociativní operátor $\oplus$ a uspořádaná posloupnost prvků

    $[a_0, a_1, ..., a_{n-1}]$

  a která vrací posloupnost

  $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$

- Např. jestliže je operátor $\oplus$ sčítání a vstupní posloupnost

  [3 1 7 0 4 1 6 3]
- pak výsledek sumy prefixů je

  [3 4 11 11 15 16 22 25].

# Suma prefixů

- Některá použití:
  - Vyhodnocování polynomů
  - Sčítání binárních čísel v hardware
  - Lexikální porovnávání řetězců
  - Lexikální analýza
  - Implementace radix-sortu, quick-sortu
  - Rušení označených prvků v poli
  - Vyhledávání regulárních výrazů (grep)
  - Implementace některých stromových operací
  - Označování komponent ve dvourozměrných obrázcích
- Některé jiné názvy
  - V knihovně MPI:  MPI_scan
  - V programu MATLAB:  y=cumsum(x)

# Lze použít jakýkoli asociativní operátor $\oplus$

Asociativita:

$(a \oplus b) \oplus c = a \oplus (b \oplus c)$

| Sum (+)<br>Product (*)<br>Max<br>Min |
| --- |
| Vstup: čísla |

| AND “$\wedge$”<br>OR “$\vee$”<br>XOR |
| --- |
| Vstup: Bity (Boolean) |

| Konkatenace |
| --- |
| Vstup: Řetězce |

| MatMul |
| --- |
| Vstup: Matice |

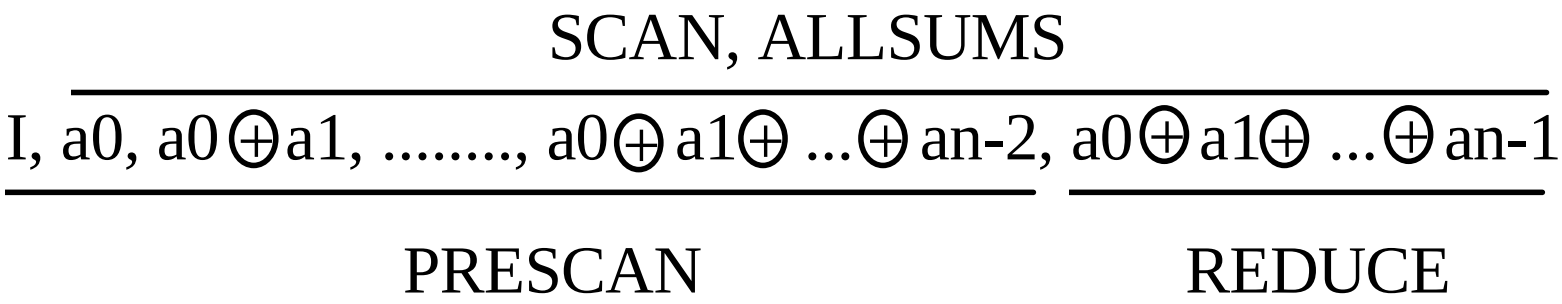
# Sekvenční řešení

```
Sekvenční algoritmus
procedure allsums (Out, In)
i=0
sum = In[0]
while i<length do
        i = i+1
        sum = sum + In[i]
        Out[i] = sum
endwhile
```

- Časová složitost je t(n) = O(n)

# Scan, prescan, reduce

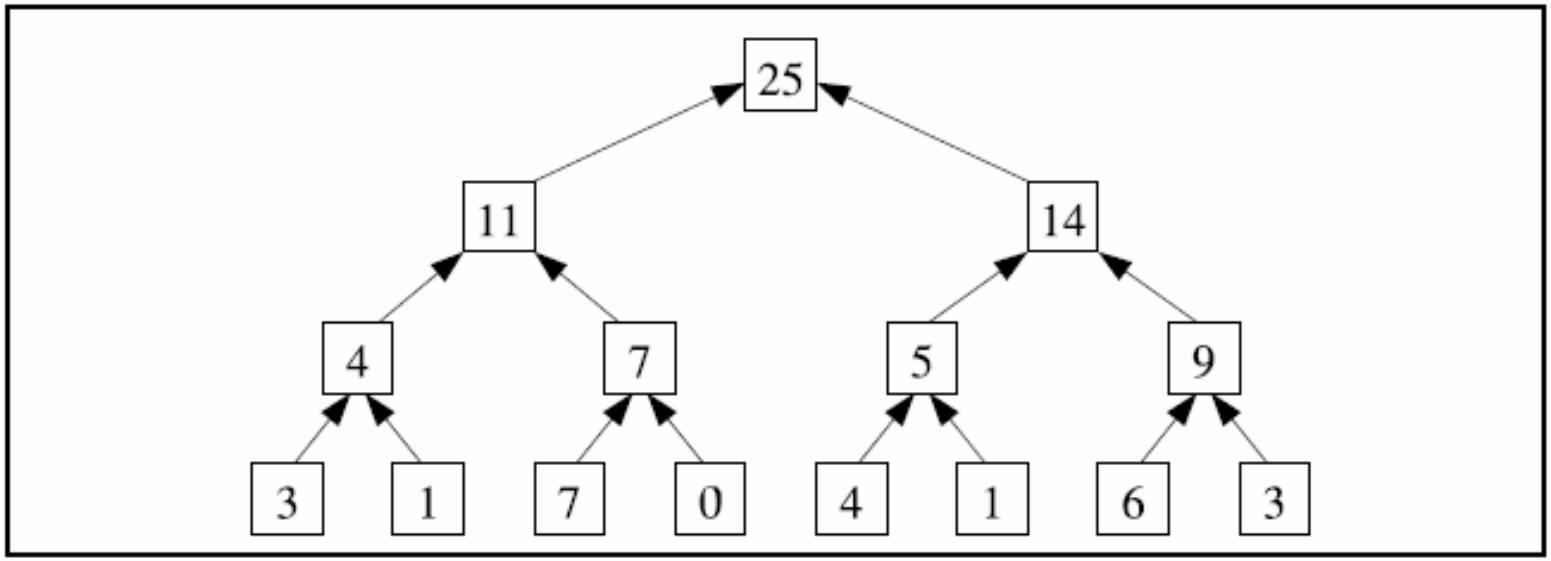
- Definice
  - Operace scan je suma prefixů
- Definice
  - Operace prescan má jako vstup binární asociativní operátor $\oplus$, neutrální prvek I a vektor $[a_0, ..., a_{n-1}]$ a vrací vektor $[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$
- Definice
  - Operace reduce má stejný vstup jak scan a vrací hodnotu $a_0 \oplus a_1 \oplus ... \oplus a_{n-1}$



# Paralelní suma prefixů - Reduce

- Reduce
  - Reduce může být spočtena pomocí stromu procesorů, za předpokladu, že $\oplus$ je asociativní (nemusí být komutativní)

```
Algoritmus
for j = 0 to log n - 1 do
  for i = 0 to n - 1 step 2^(j+1) do in parallel
    a[i+2^(j+1)-1] = a[i + 2^j -1] ⊕ a[i + 2^(j+1) - 1]
  end for
end for
```




(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** ] |

(b) Executing a +-reduce on a PRAM.

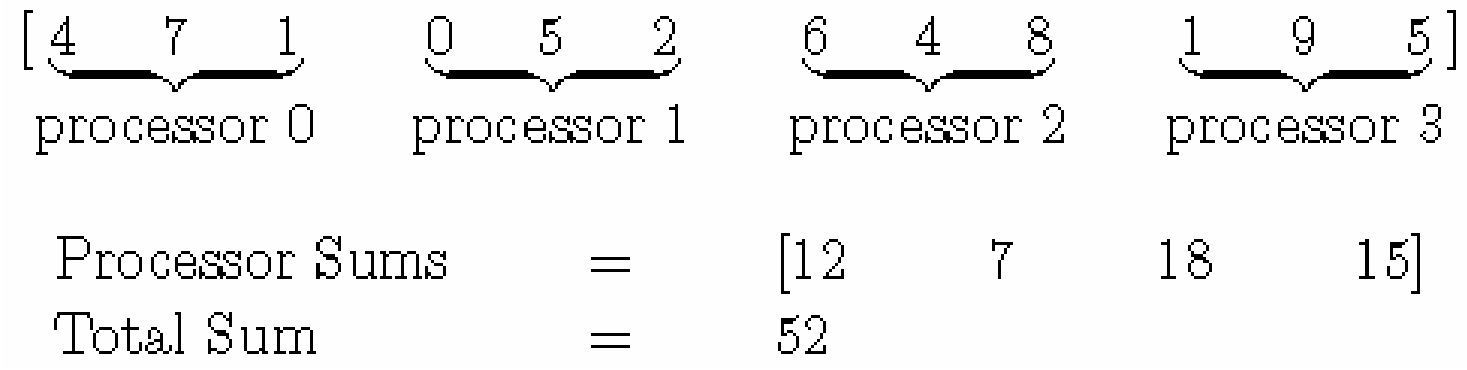
# Reduce

- Diskuse
  - strom má výšku log n, pro každý pár prvků je nutný 1 procesor
  - t(n) = O(log n) c(n) = O(n.log n)
  - p(n) = n/2 $\rightarrow$ což není optimální
- máme-li méně procesorů N<n, každý procesor provede sekvenční reduce pro svůj kousek posloupnosti o délce n/N a výsledky se zpracují pomocí reduce

# +-reduce pro n > p

- Algoritmus

```
for each processor i do in parallel
        sum[i] = a[(n/N).i]
        for j = 1 to n/N do
                sum[i] = sum[i] + a[(n/N).i + j]
result = +-reduce(sum)
```

- 

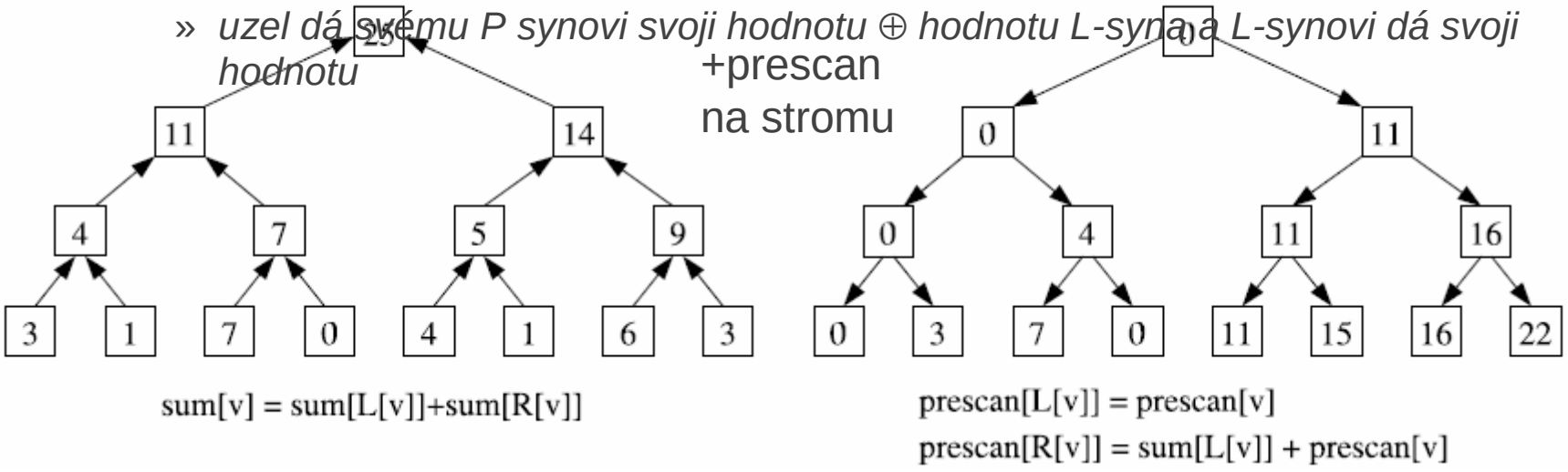

  Pak
  - t(n) = ⌈n/N⌉ + ⌈log N⌉ = O(n/N + log N)
  - je-li log N < n/N, pak

    t(n) = O(n/N) a

    c(n) = O(n/N).N = O(n) → což je optimální

# Prescan a scan

- Uvedeme prescan, scan se získá posunem doleva, přidáním reduce
- Algoritmus
  - (i) UpSweep *algoritmus totožný s reduce, ale každý uzel si pamatuje mezisoučet*
  - (ii) DownSweep
    - » *kořenu se přiřadí neutrální prvek I*
    - » *nyní se provádí log n kroků (každá úroveň jednou), počínaje kořenem, směrem k listům a v každém kroku procesory v té úrovni pracují paralelně:*
    - » *uzel dá svému P synovi svoji hodnotu ⊕ hodnotu L-syna a L-synovi dá svoji hodnotu*



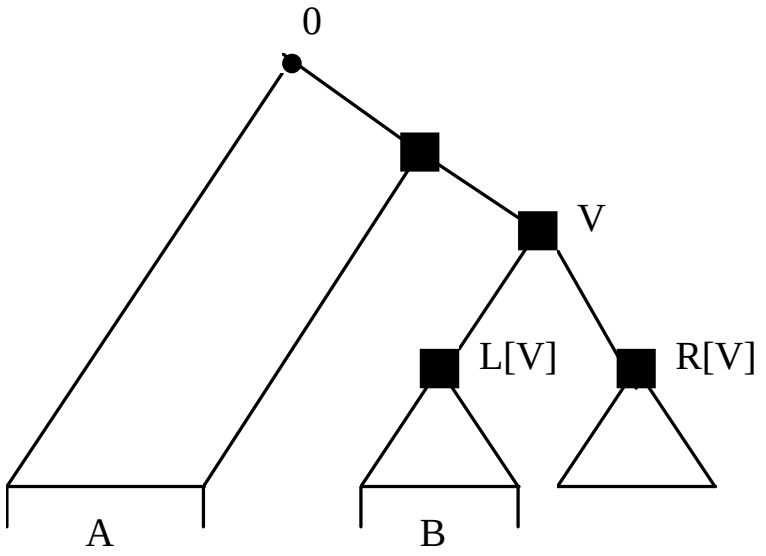
# +-prescan na architektuře PRAM

```
Procedure down-sweep (a)
a[n-1] = 0
for d = (log n) - 1 downto 0 do
  for i = 0 to n-1 step 2^(d+1) do in parallel
    t = a[i + 2^d - 1]
    a[i + 2^d - 1] = a[i + 2^(d+1) - 1]      //left child = parent
    a[i + 2^(d+1) - 1] = t + a[i + 2^(d+1) - 1]//right child = parent+right
  end for
end for
```

| | Step | Vector in Memory | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | [ **3** | **1** | **7** | **0** | **4** | **1** | **6** | **3** ] |
| up | 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** ] |
| clear | 4 | [ 3 | 4 | 7 | 11 | 4 | 5 | 6 | **0** ] |
| down | 5 | [ 3 | 4 | 7 | **0** | 4 | 5 | 6 | **11** ] |
| | 6 | [ 3 | **0** | 7 | **4** | 4 | **11** | 6 | **16** ] |
| | 7 | [ **0** | **3** | **4** | **11** | **11** | **15** | **16** | **22** ] |

- Teorém: po dokončení downsweep obsahuje každý uzel sumu hodnot všech listů, jež ho předcházejí
- Důkaz indukcí:
  - kořen nepředcházejí žádné listy, takže jeho správná hodnota je neutrální prvek
  - L-syn je předcházen týmiž uzly, jako samotný uzel (A), za předpokladu, že otec má správnou hodnotu, stačí ji předat L-synovi

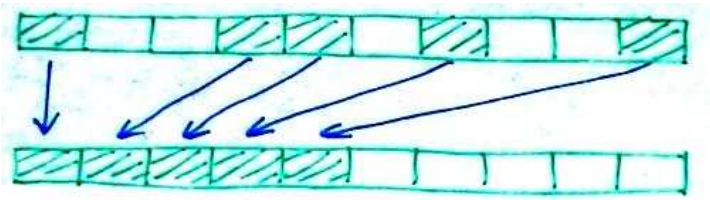
- pravý syn je předcházen listy A, B, proto jeho hodnota je hodnota otce ⊕ hodnota levého syna



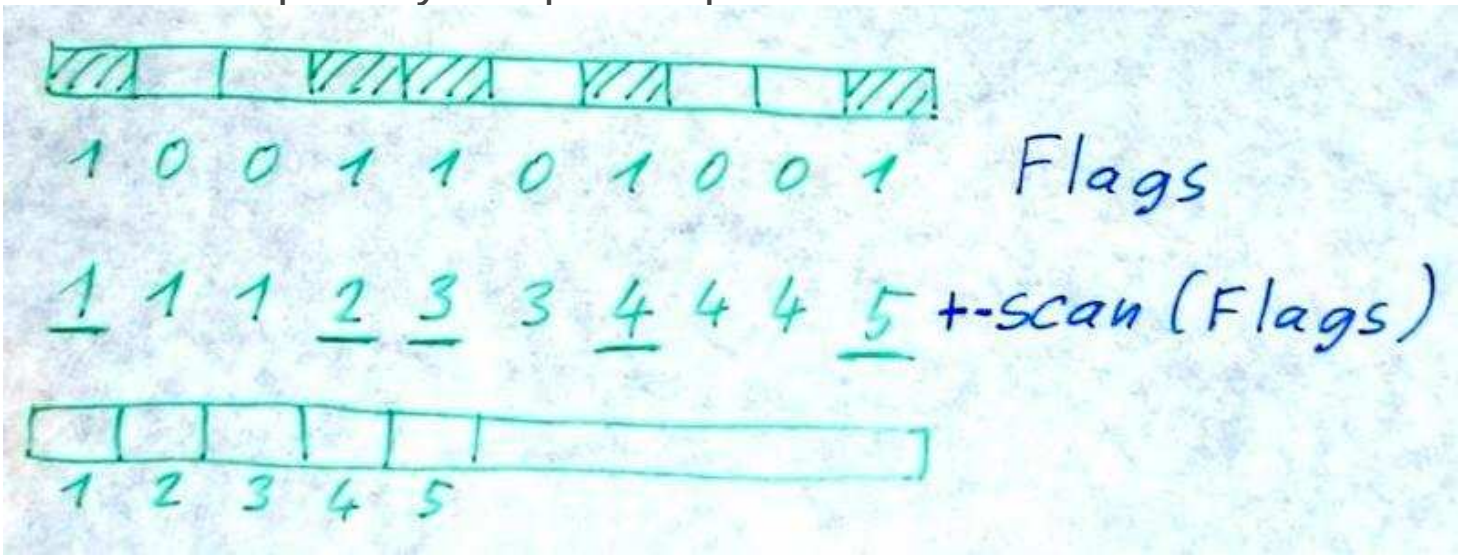
- Analýza
- Složitost je stejná, jako u reduce
- Zlepšení ceny:
  - t(n) = O(n/N)
  - c(n) = O(n), za předpokladu, že log N < n/N $\rightarrow$ což je optimální

# Packing problem

- Máme k vstupních položek, rozmístěných v poli o n pozicích, k<n
- Cílem je vytvořit výstupní pole, kde položky zaujímají prvních k pozic

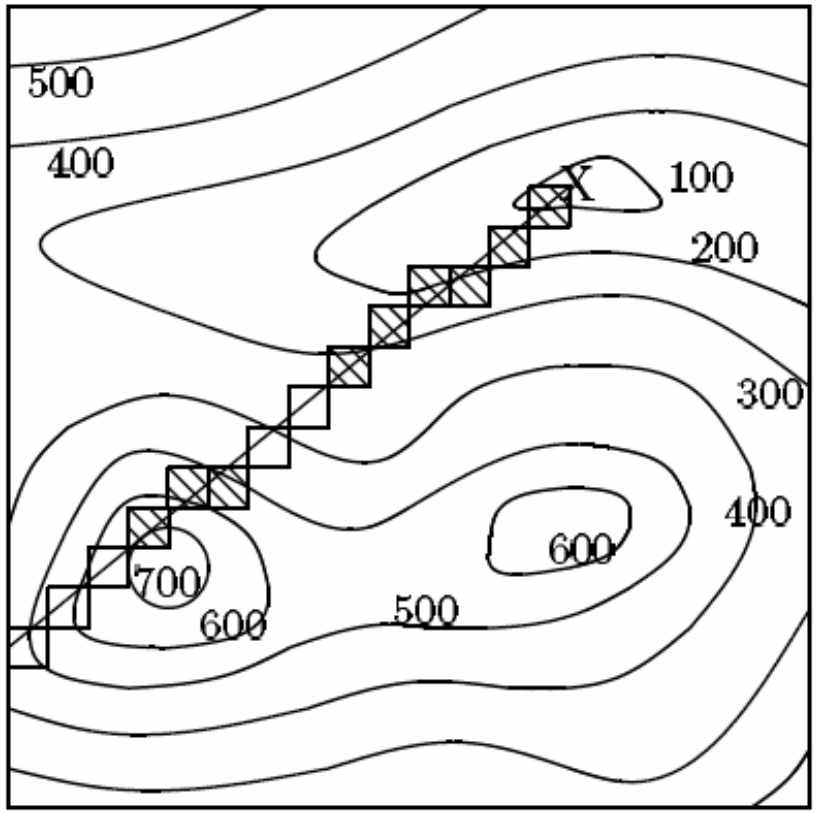


- Algoritmus
  - Spočteme pole binárních příznaků, 1-položka existuje, 0-neexistuje
  - Spočteme +-scan tohoto pole
  - Přesuneme položky na správné pozice

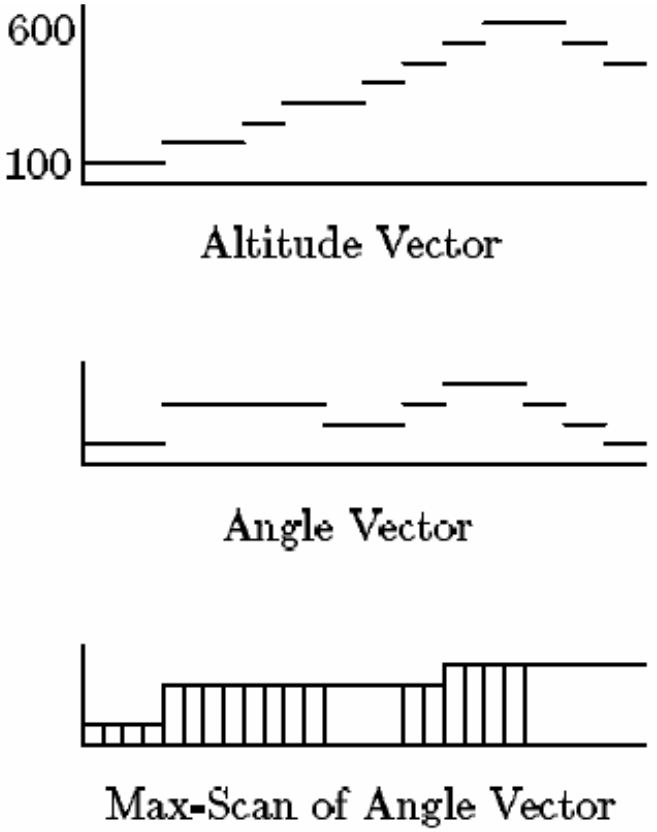


# Viditelnost

- Problém
  - Je dána matice terénu ve formě matice nadmořských výšek a pozorovací bod X(místo pozorovatele), zjistěte, které body podél paprsku vycházejícího z místa X jsou viditelné
- Řešení
  - Bod je viditelný, pokud žádný bod mezi pozorovatelem a jím nemá větší vertikální úhel.
  - (i) vytvoří se vektor výšek bodů podél pozorovacího paprsku
  - (ii) vektor výšek se přepočítá na vektor úhlů
  - (iii) pomocí max_prescan se spočte vektor maximálních úhlů pro zjištění viditelnosti bodu stačí určit jeho úhel a porovnat s maximem.
- Analýza
  - t(n) = O(n/N + log N) na EREW PRAM




Altitude Map

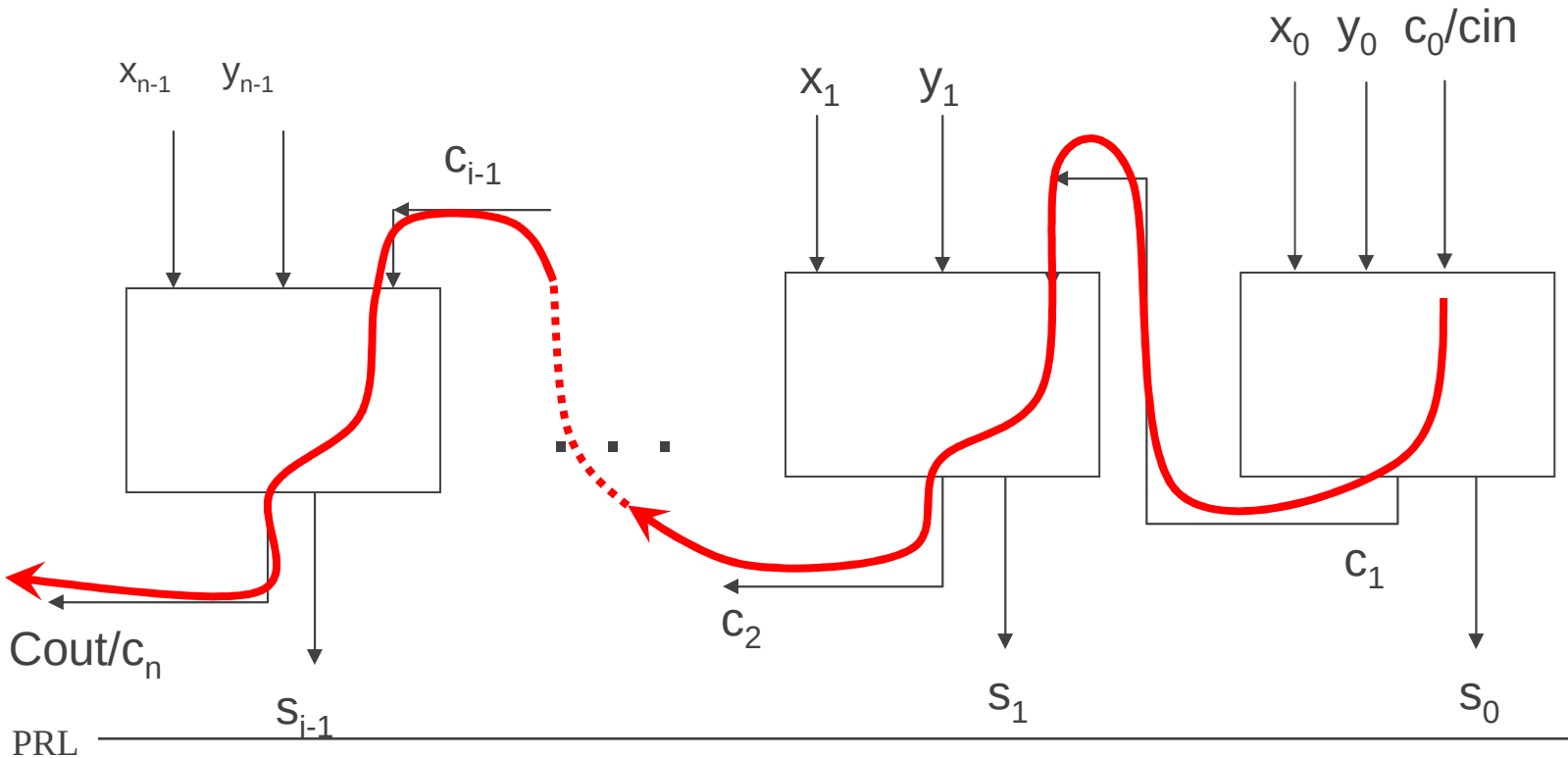



Ray Vectors

```
procedure LINE_OF_SIGHT
for each index i do in parallel
        angle[i] = arctan ((alt[i] - alt[0])/i)
max_prev_angle = max_prescan(angle)
for each index i do in parallel
        if (angle[i] > max_prev_angle[i]) then result[i] = visible
                else result[i] = unvisible endif
endfor
```

# Carry Look Ahead Parallel Binary Adder

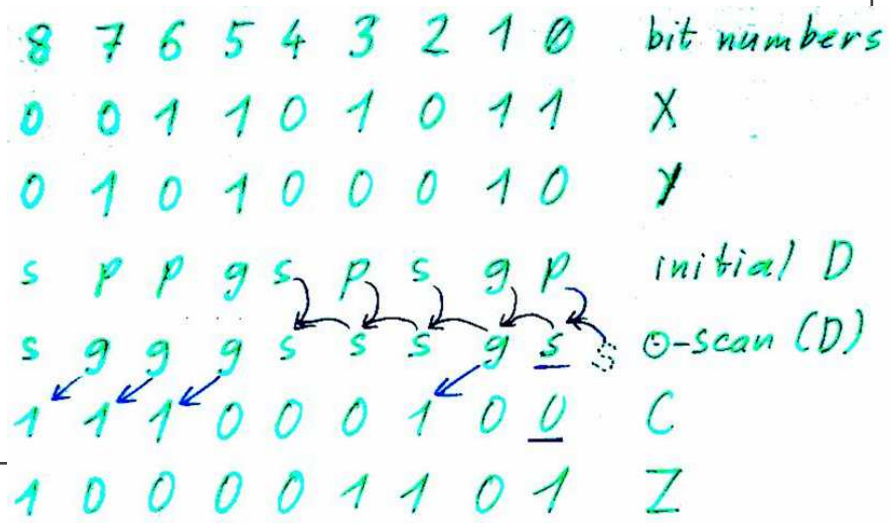
- Vstup: Dvě n-bitová binární čísla $X = x_{n-1}..x_0$ a $Y = y_{n-1}..y_0$
- Cíl: Spočítat $Z = z_{n-1}..z_0 = X + Y$ pomocí log n kroků
- Musíme předvypočítat všechny bity přenosu $c_{n-1}..c_0$ abychom mohli přímo spočítat $z_i = x_i + y_i + c_i$ $I=0..n-1$



# Carry Look Ahead Parallel Binary Adder

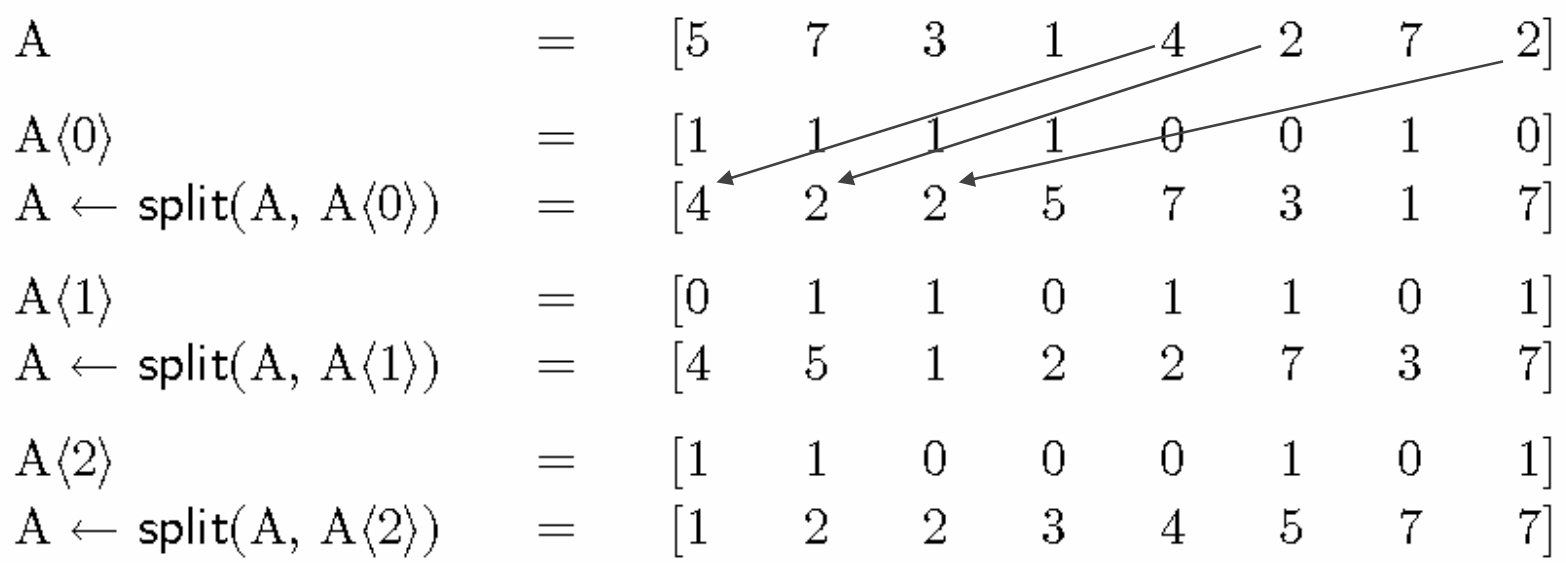
- Algoritmus:
  - Spočteme pole D = $d_{n-1}…d_0$ kde $d_i \in$ {propagate, stop, generate}
  - **for i=0 to n-1 do in parallel**
    **if ($x_i$=1) and ($y_i$=1) then $d_i$ = g**
    **else if ($x_i$=0) and ($y_i$=0) then $d_i$ = s**
    **else $d_i$ = p**
    **endfor**
  - Spočteme ⊙-scan pole D a tím dostaneme všechny bity přenosu v logaritmickém čase

| ⊙ | $s$ | $p$ | $g$ |
|---|---|---|---|
| $s$ | $s$ | $s$ | $s$ |
| $p$ | $s$ | $p$ | $g$ |
| $g$ | $g$ | $g$ | $g$ |



# Radix sort

- Bitový radix sort (radix = 2)
  - V každém kroku se bere v úvahu 1 bit klíče a pomocí operace split se prvky s nulovým bitem přemístí na začátek řazeného pole řazených čísel a s jedničkovým bitem na konec




```
procedure SPLIT_RADIX_SORT(A, number_of_bits)
for i = 0 to (number_of_bits - 1) do
        A = split(A, A<i>)
```

# Radix sort - Operace split

- Jak udělat split? - sekvenční složitost je O(n)
- Idea
  - pro každý prvek určíme správnou pozici a v konstantním čase přemístíme (EREW)
- Postup
    - pro prvky s nulovým bitem se jejich pozice získá provedením ⊕ - prescan na invertované pole bitů
    - pro prvky s jedničkovým bitem provedu ⊕ scan na reverzované pole bitů (tj. od konce) a výsledek se odečte od n.
- Analýza
  - split má stejnou složitost jako scan
- Radix sort:
  - $t(n) = O(n/N + \log N).O(\log n) = O(n/N.\log n + \log n \,.\, \log N)$

# Operace split

```
procedure split(A, Flags)
    I-down = +-prescan(not(Flags))
    I-up = n - +-scan(reverse-order(Flags))
    for i=0 to n-1 do in parallel
        if (Flags[i]) Index[i] = I-up[i]
        else Index[i] = I-down[i] endif
    endfor
    result = permute (A, Index)
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | = | [ 5 | 7 | 3 | 1 | 4 | 2 | 7 | 2 ] |
| Flags | = | [ 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 ] |
| I-down | = | [ 0 | 0 | 0 | 0 | **0** | **1** | 2 | **2** ] |
| I-up | = | [ **3** | **4** | **5** | **6** | 7 | 7 | **7** | 8 ] |
| Index | = | [ 3 | 4 | 5 | 6 | 0 | 1 | 7 | 2 ] |
| permute(A, Index) | = | [ 4 | 2 | 2 | 5 | 7 | 3 | 1 | 7 ] |

# Segmentovaný scan

- Definice
  - Suma prefixů je operace, jejímž vstupem je binární asociativní operátor $\oplus$, uspořádaná posloupnost prvků

    $[a_0, a_1, ..., a_{n-1}]$

    uspořádaná posloupnost příznaků

    $[f_0, f_1, ..., f_{n-1}]$

    a která vrací posloupnost

    $[s_0, s_1, ..., s_{n-1}]$

    kde v posloupnosti s jsou sumy prefixů přes jednotlivé segmenty, kde hranice segmentu je dána hodnotou 1 v poli příznaků f
- 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)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| klíč | 6,4 | 9,2 | 3,4 | 1,6 | 8,7 | 4,1 | 9,2 | 3,4 |
| flags | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Pivots | 6,4 | 6,4 | 6,4 | 6,4 | 6,4 | 6,4 | 6,4 | 6,4 |
| F | == | > | < | < | > | < | > | < |
| split | 3,4 | 1,6 | 4,1 | 3,4 | 6,4 | 9,2 | 8,7 | 9,2 |
| flags | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| pivots | 3,4 | 3,4 | 3,4 | 3,4 | 6,4 | 9,2 | 9,2 | 9,2 |

- Analýza
  - Každá iterace má konstantní počet operací scan
  - Při vhodné volbě pivotů skončí algoritmus po O(log n) krocích, takže složitost je:
  - $t(n) = O(n/N + \log N).O(\log n) = O(n/N.\log n + \log N.\log n)$
  - $c(n) = O(n.\log N + N.\log n.\log N)$

    $\rightarrow$ pro dostatečně malé N optimální

# KONEC