Complexitatea algoritmilor paraleli

Ciprian Dobre
ciprian.dobre@cs.pub.ro

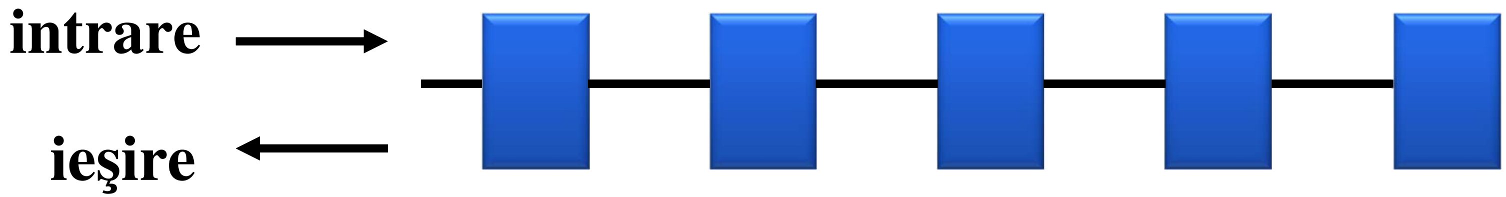
Complexitatea algoritmilor paraleli

- Cât de mult paralelism există într-o aplicație?
  “Depinde”

- Performanța – funcție de:
  - **Timp de execuție**
  - Memorie ocupată
  - Număr de procesoare
  - **Scalabilitate**
  - Eficiență
  - Fiabilitate
  - Toleranță la defecte
  - Portabilitate
  - Costuri

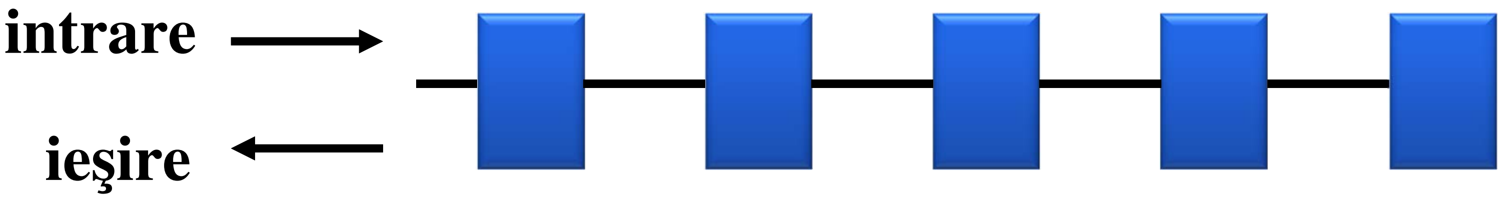
- Exemplu: sortarea pe un vector de procesoare

Sortarea pe un vector de procesoare



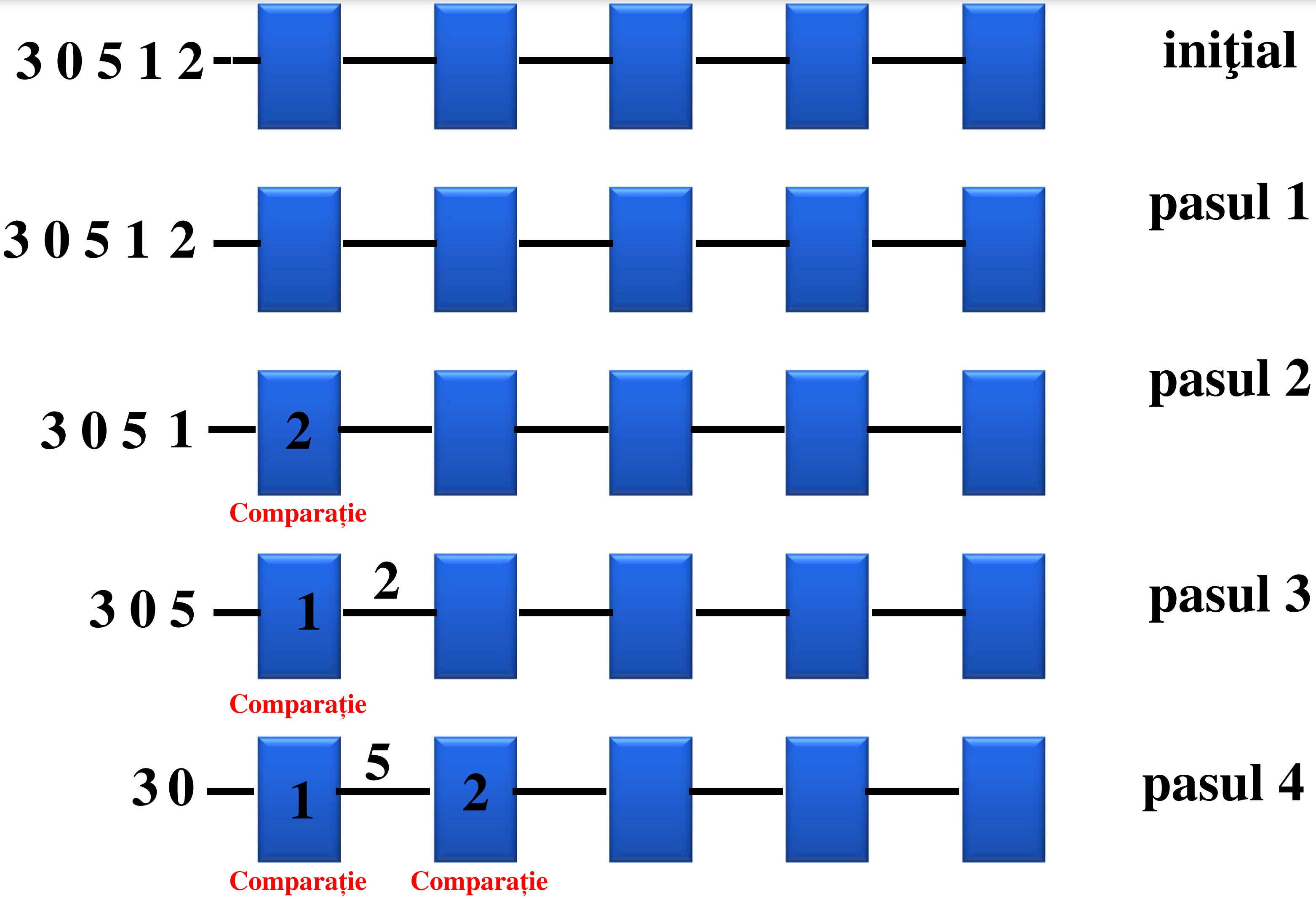
- Fiecare procesor are o memorie locală și propriul său program
- Funcționare **sistolică**
  - **Un ceas global comandă execuția simultană a operațiilor**
  - **Fiecare procesor realizează la fiecare pas:**
    - **Recepția unor date de la vecini**
    - **Inspecția memoriei locale**
    - **Execuția calculelor specificate de program**
    - **Modificarea memoriei locale**
    - **Transmiterea unor date către vecini**

Sortarea pe un vector de procesoare



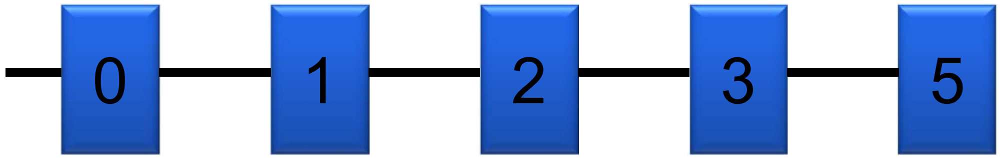
- Vector de N procesoare
- 2 faze
- În fiecare pas al **primei faze**, fiecare procesor realizează următoarele operaţii:
  1) acceptă o valoare de la vecinul din stânga;
  2) compară valoarea cu cea memorată local;
  3) transmite valoarea mai mare vecinului din dreapta;
  4) memorează local valoarea mai mică.

Pașii algoritmului de sortare



Pașii algoritmului de sortare

După pasul 9:



Pentru **N** valori, **faza 1** durează **2N-1** pași

Faza a doua

- Valorile ordonate sunt scoase prin celula din stânga
- Variante:
  1. fiecare procesor începe să transmită spre stânga, imediat ce numerele sunt sortate;
  2. cunoscând poziţia sa din vector şi numărând valorile inspectate, fiecare procesor calculează momentul când începe să transmită spre stânga;
  3. procesorul din dreapta începe să transmită spre stânga imediat ce primeşte o valoare; toate celelalte încep să transmită spre stânga imediat ce primesc o valoare din dreapta;
  4. fiecare procesor începe să transmită spre stânga imediat ce nu mai primeşte o valoare dinspre stânga.

- Complexitate:
  - metoda 3 → **4N-3** pași
  - metoda 4 → **4N-3** pași

- Variantele 1 și 2 impun cunoașterea
  - Pozitiei procesoarelor in vector
  - Contorizarea valorilor

Măsuri de performanţă

- **T : Timpul total** necesar execuţiei
- **P : Numărul de procesoare utilizate**
  - În algoritmul prezentat, $P = N$ și $T = O(N)$.
  - **N** : numărul de valori
- **S : Accelerația** *(speedup)*

$$S = \frac{G}{T}$$

  - **G** : timpul de execuţie al celui mai rapid algoritm secvenţial (în cazul nostru: *quicksort*)
  - În algoritmul precedent : $S = \dfrac{O(N \log N)}{O(N)} = O(\log N)$

Măsuri de performanţă

- Accelerația liniară $S = O(P)$
  - Exemplu ???

Ideal ar fi ca algoritmul paralel să se execute de P ori mai rapid decât cel mai bun algoritm secvențial

  - Cea mai bună valoare a accelerației ce se poate obține
  - Justificare:

    Dacă un program se poate executa în T pași pe P procesoare, putem oricând să rulăm respectivul algoritm în mod secvențial în TP pași

$G \leftarrow TP \quad \Leftrightarrow \quad S \leftarrow P$

Măsuri de performanţă (2)

- Se poate obține mereu accelerație liniară ?
  - Topologia sistemului impune uneori restricții de timp (ex: liniară $O(N)$), arbore $O(\log N)$)
  - Secțiuni secvențiale – Legea lui Amdahl
    - Presupunând că procentul **f** din totalul calculelor trebuie să se desfăşoare secvenţial

$$T = fG + (1 - f)\frac{G}{P} \Rightarrow$$

$$S = \frac{1}{f + \frac{(1 - f)}{P}} \Rightarrow$$

$$S \leq \frac{1}{f}$$

    - Un program paralel nu va rula mai repede decât suma porțiunilor sale secvențiale, indiferent de numărul de procesoare pe care se execută

$$T \geq f \cdot G$$

Măsuri de performanţă (3)

- *"… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."*

  Amdhal, 1967

- *"… speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size."*

  Gustafson, 1988

Măsuri de performanţă (4)

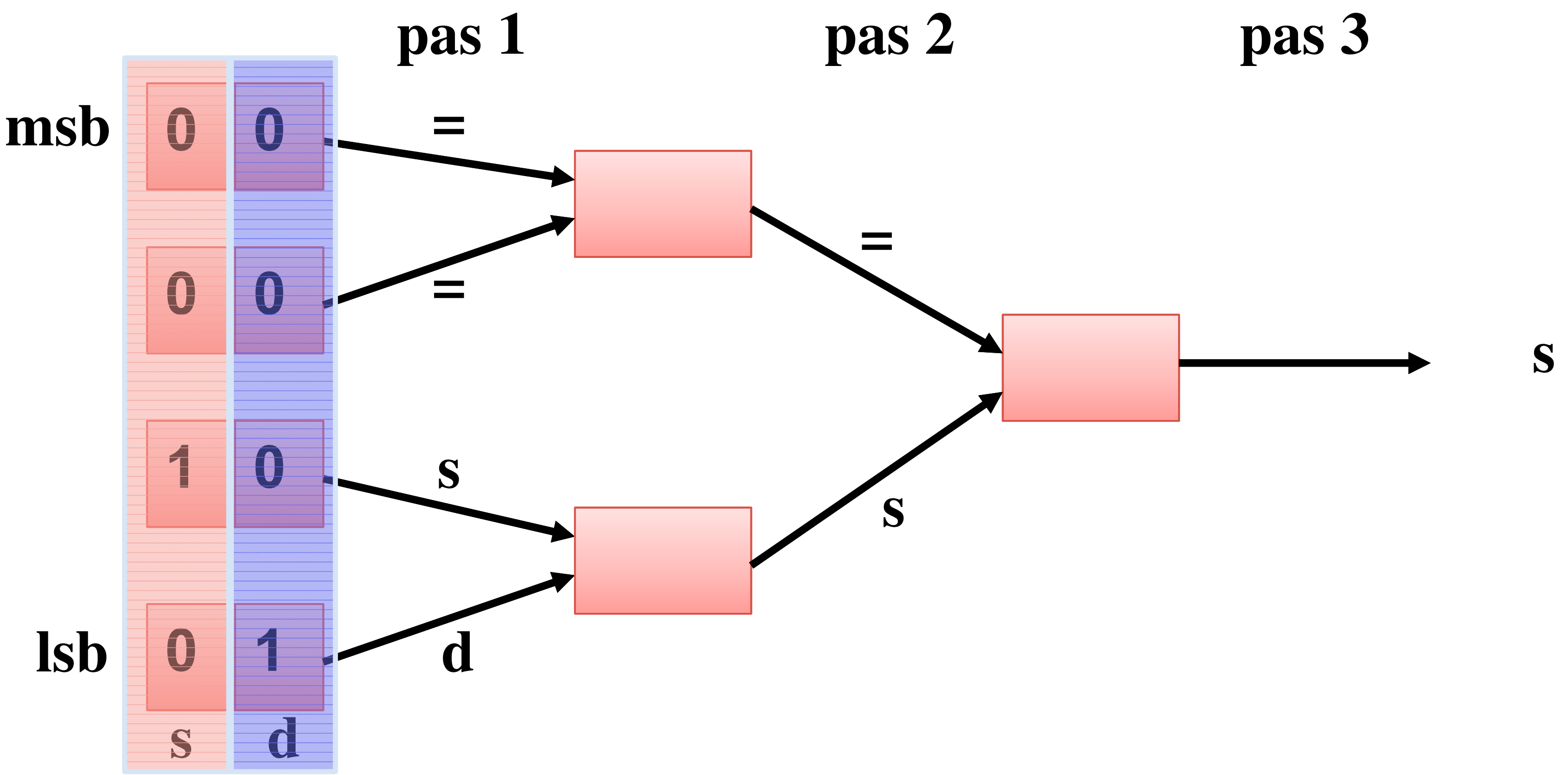
- **Costul** $C = T \cdot P$
  - *în exemplu :* $C = O(N^2)$
  - Caracterizează ineficiența datorată nefolosirii complete a procesoarelor
- **Eficienţa** $E = \frac{G}{C}$
  - $E = \frac{G}{C} = \frac{G}{TP} = \frac{S}{P}$
  - *în exemplu :* $E = O\left(\frac{\log N}{N}\right)$
- **Scalabilitatea**
  - măsură a *accelerării* date de adăugarea mai multor procesoare

Calculul detaliat al complexităţii

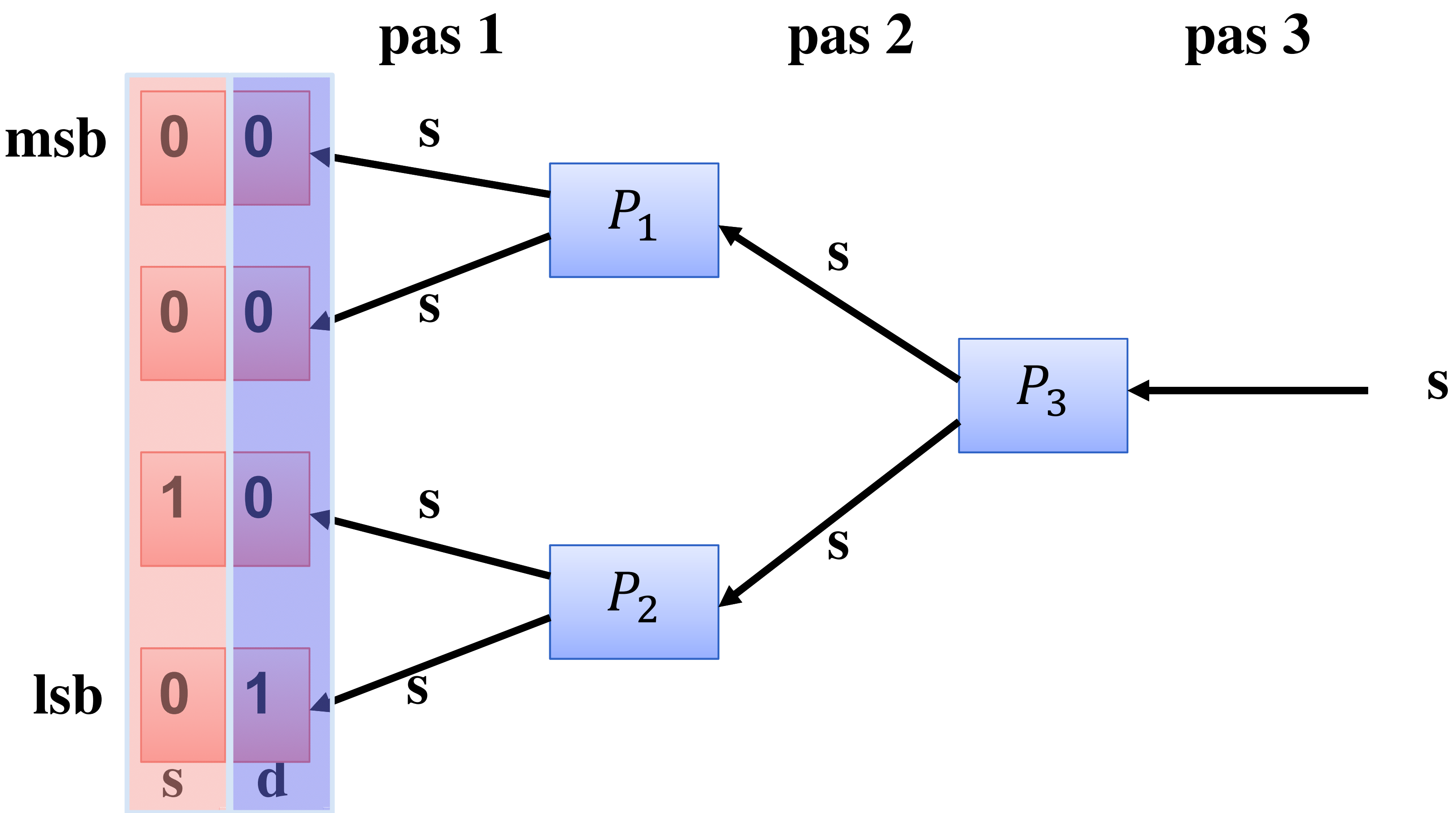
- Paralelizarea se face pentru atingerea unui timp de execuție cât mai redus
- Cum putem ști dacă un algoritm paralel este optim sau nu?
- Răspunsul depinde de **modelul de evaluare adoptat**
- În algoritmul anterior pasul algoritmului reprezenta o **operație** asupra unor **cuvinte**

Calculul detaliat al complexităţii

- Trecere la modelul "bit"
  - Fiecare procesor operează pe 1 bit
  - Operația principală – compararea a două numere "s" și "d"
  - Topologie arborescentă



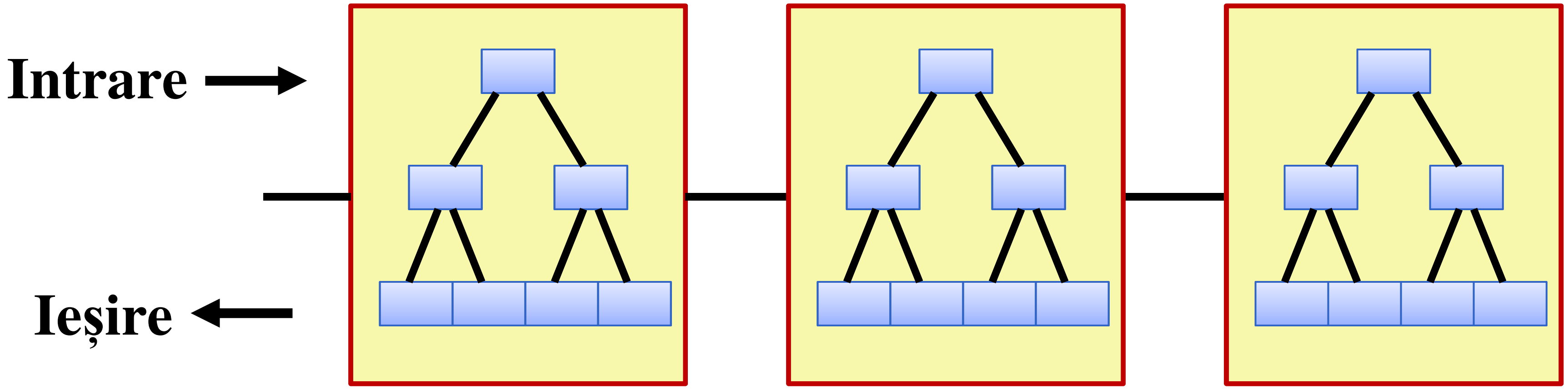
Calculul detaliat al complexităţii



- k : numărul de biți
- comparaţia terminată in 2 * log k paşi
  - $log\, k$ $pași\ pentru\ transmisie\ și$ $log\, k$ *pași pentru recepție*
- 2k - 1 procesoare

Calculul detaliat al complexităţii

- Algoritmul de sortare devine:



- Rețea de **(2k-1)*N** procesoare
- **Test**: Câți pași sunt necesari în această abordare pentru faza 1?
- **(2N-1)*2*logK** pași pentru faza 1

Abordare pipeline

- Algoritm **mai bun** de comparare
- Tablou liniar de **k** procesoare pe bit
- În timp ce un procesor compară o pereche de biți a două numere succesive, procesorul de sub el lucrează cu biții mai puțin semnificativi ai perechii de numere anterioare
- La fiecare pas, un procesor primește la intrare un bit, iar de la procesorul de deasupra o informație asupra comparației făcute de el la pasul anterior:
  - **s** – dacă numărul de la intrare este mai mare
  - **d** – dacă numărul memorat este mai mare
  - **=** - dacă cele două numere sunt egale
- Dacă primește **s**, procesorul transmite bitul de la intrare la ieșire și transmite **s** în jos
- Dacă primește **d**, procesorul transmite bitul memorat la ieșire și transmite **d** în jos
- Dacă primește **=**, transmite bitul mai mare la ieșire, memorează bitul mai mic și transmite în jos **s**, **d**, sau **=** în mod corespunzător

Abordare pipeline

```
    3 0 5 1 2  <- iniţial
-----------------------------------------------------------------------
    0 0 1 0 0 [ ]       <-msb            0 [0] -> 0 1 0
   / / / / /                            /   =        /
  1 0 0 0 1   [ ]                      1 0 [0] -> 0 1
 / / / / /                            /     s      /
1 0 1 1 0     [ ]       <-lsb        1 0 1 [1] -> 0
              iniţial                      după pasul 4
-----------------------------------------------------------------------
      0 0 1 0 [0]                          [0] -> 0 0 1 0
     / / / /                                =        / /
    1 0 0 0 1 [ ]                        1 [0] -> 0 0 1
   / / / /                                  =      / /
  1 0 1 1 0   [ ]                      1 0 [1] -> 1 0
              după pasul 1                 după pasul 5
-----------------------------------------------------------------------
        0 0 1 [0] -> 0                     [0] ->    0 0 1 0
       / / /   =                                     / / /
      1 0 0 0 [1]                          [0] -> 1 0 0 1
     / / /                                  s      / / /
    1 0 1 1 0 [ ]                        1 [0] -> 1 1 0
              după pasul 2                 după pasul 6
-----------------------------------------------------------------------
          0 0 [0] -> 1 0                   [0] ->      0 0 1 0
         / /   s                                     / / / /
        1 0 0 [0] -> 1                     [0] ->    1 0 0 1
      / /      d                                   / / / /
      1 0 1 1 [0]                          [0] -> 1 1 1 0
              după pasul 3                 după pasul 7
-----------------------------------------------------------------------
rezultat ->                                 0     3 1 5 2
```

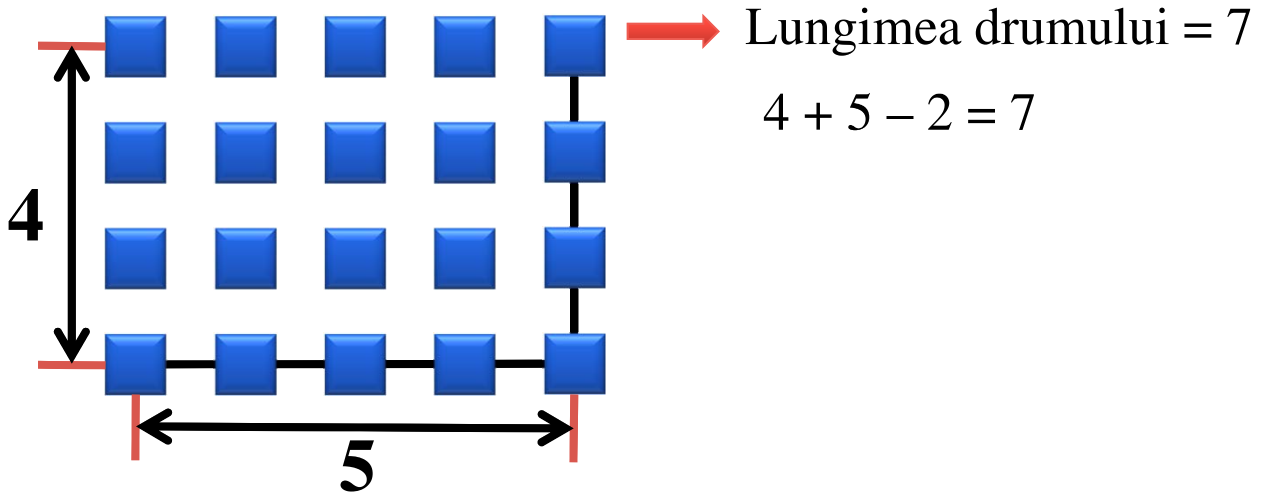
Complexitate:

(k-1)+(2N-1) paşi faza 1

Limite inferioare

- **Lărgimea benzii** pentru introducerea datelor
  - *ex: N numere se introduc în N pași*

- **Diametrul reţelei** – distanța maximă între oricare două noduri *(numărul de muchii)*
  - *ex: pentru un grid de k*N noduri, diametrul este N + k -2*



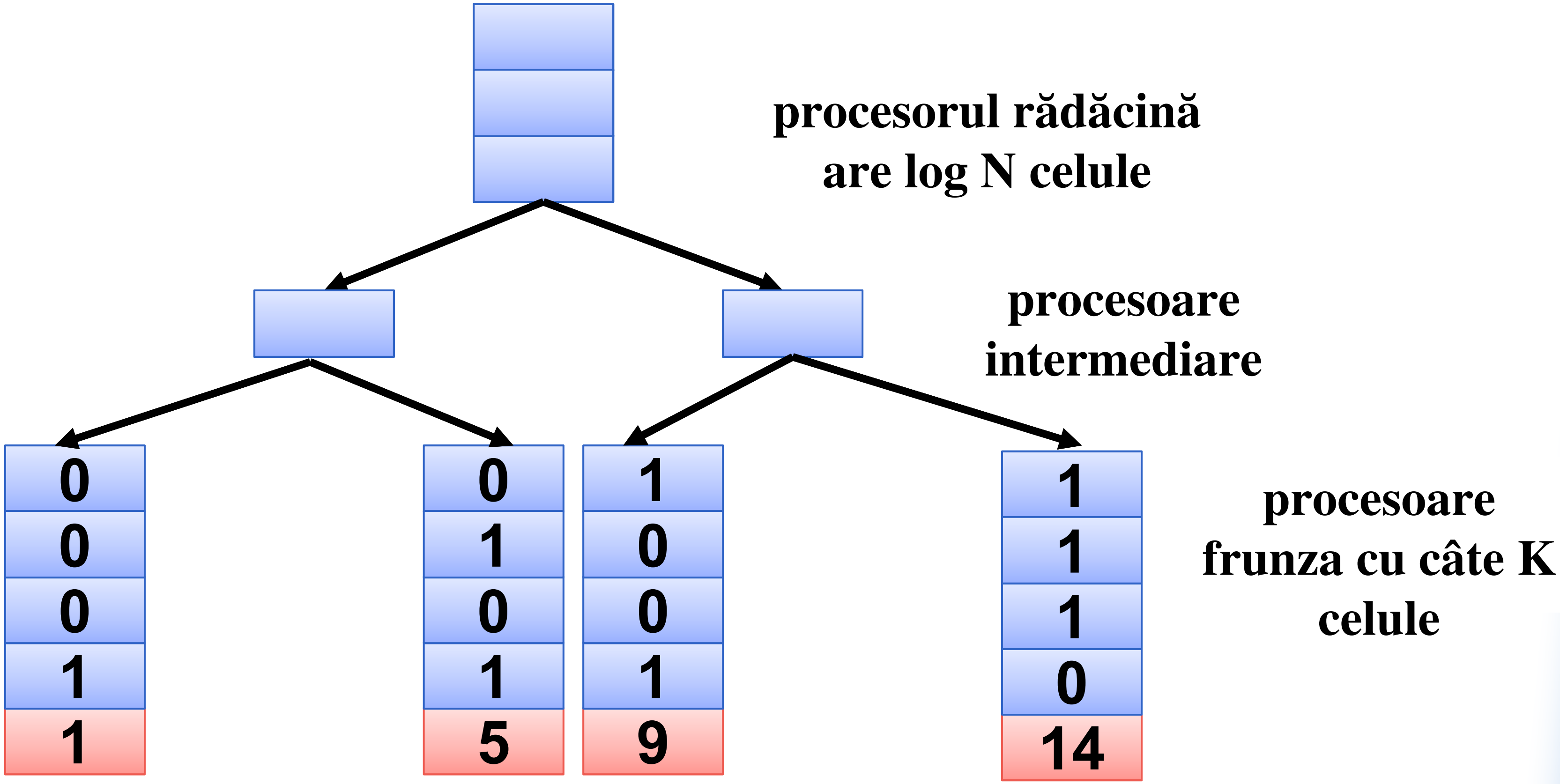
Limite inferioare

- **Lărgimea benzii** pentru introducerea datelor
  - *ex: N numere se introduc în N pași*

- **Diametrul reţelei** – distanța maximă între oricare două noduri *(numărul de muchii)*
  - *ex: pentru un grid de k * N noduri, diametrul este N + k - 2*

- **Lărgimea bisecţiei** rețelei – numărul minim de legături care trebuie îndepărtate pentru a împărți rețeaua în două subrețele separate, având același număr de noduri
  - interschimbarea datelor
  - *ex:* $\frac{kN}{2}$ *biți "trec" prin bisecția de lărgime k în minim* $\frac{N}{2}$ *pași*

Limite inferioare



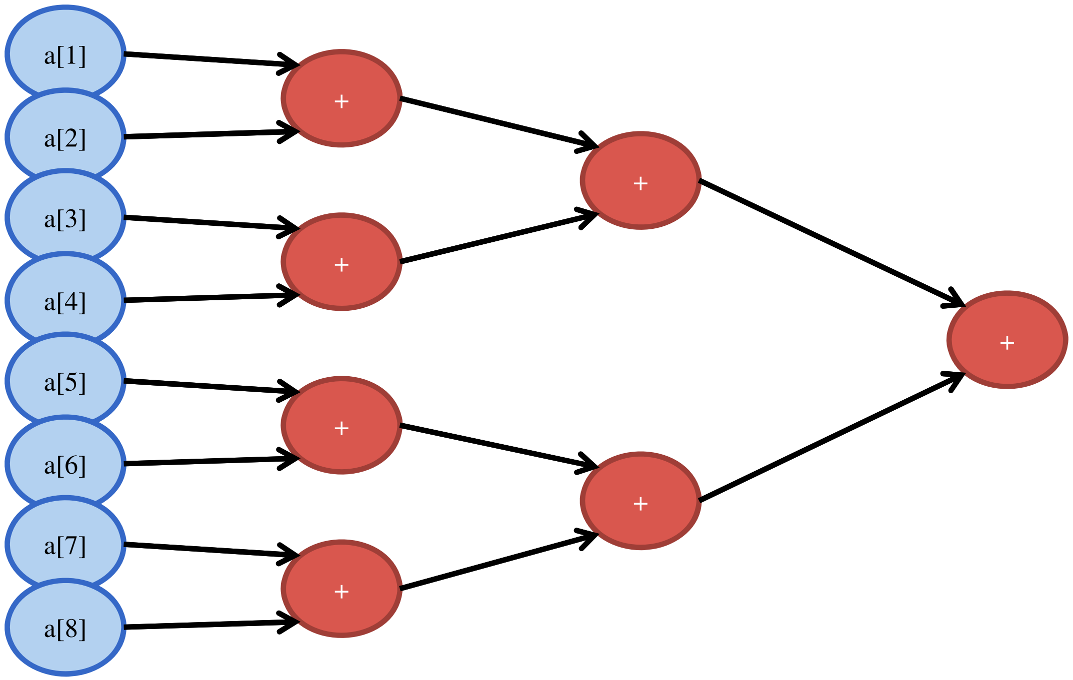
- Diametrul reţelei este $2\log N + 2K - 2$
- Lărgimea benzii de intrare a datelor este $k \cdot N$
- Lărgimea bisecţiei este 1, dacă log N = 1 şi 2 în celelalte cazuri
- Sortare in $O(KN)$ pași
- dar, pentru $K = 1$ există algoritmi în $O(\log N)$

Modele generale

- Analiza complexității algoritmilor presupune folosirea unui **model formal**:
  - particulare
    - *ex: SIMD cu memorie locală*
  - abstracte:
    - Modele orientate spre categorii de mașini paralele:
      - ✓ Mașini cu memorie locală
      - ✓ Mașini cu memorie modulară
      - ✓ Mașini PRAM
    - Modele orientate spre algoritmi:
      - ✓ Modelul grafurilor orientate aciclice *(work depth)*

Modelul grafurilor orientate aciclice

- Model de graf *(work depth)*
  - fiecare intrare este reprezentată ca un nod fără arce de intrare
  - fiecare operație este reprezentată ca un nod ale cărui arce de intrare provin de la nodurile care reprezintă operanzii
  - o ieşire este reprezentată ca un nod fără arce de ieşire.



Modelul grafurilor orientate aciclice (2)

- Modelul este independent de orice arhitectură şi orice număr concret de procesoare.
- Pune în evidenţă:
  - **lucrul** efectuat de algoritm (numărul total de operaţii)
  - **adâncimea** (lungimea celui mai lung lanţ de dependenţe secvenţiale din algoritm).
- **paralelismul algoritmului**

$$\Rightarrow \frac{lucru}{adâncime}$$

- însumarea a **n** numere
  - lucrul = **n - 1**
  - adâncimea = **log n**

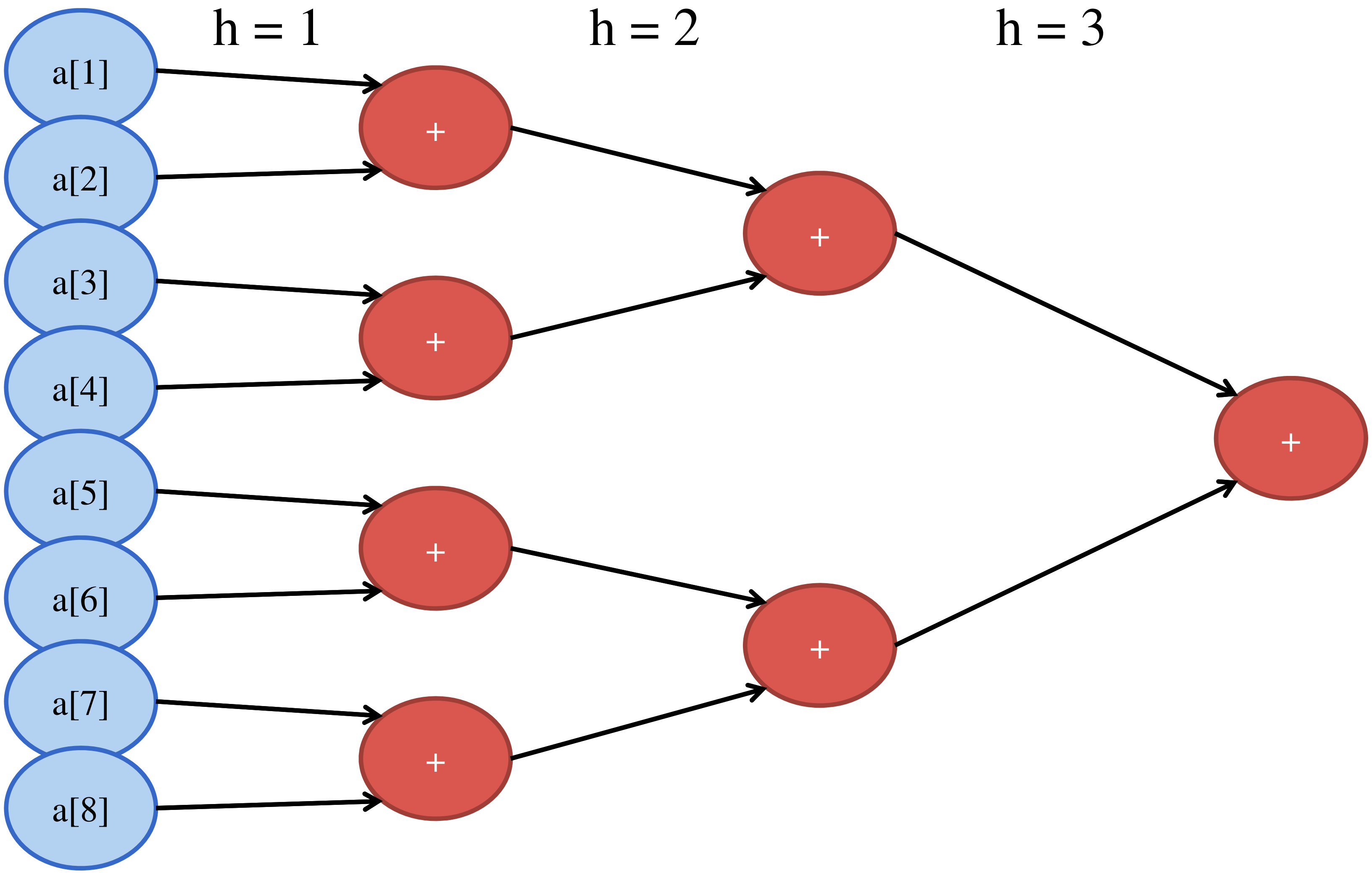
Calculul complexităţii

```
var a, b: array [1:n] of real; /* presupunem n = 2^k */
var s: real;
fa i := 1 to n do in parallel
   b[i] := a[i]
af;
fa h := 1 to log n do
    fa i := 1 to n/2^h do in parallel
        b[i] := b[2*i-1] + b[2*i]
    af;
af;
s := b[1];
```

$T(n) = 1 + \log n + 1 = O(\log n)$ *(timpul total necesar execuției)*

$W(n) = n + \sum_{h=1}^{\log n} \frac{n}{2^h} + 1 = O(n)$ *(numărul total de operații)*

Exemplu



Principiul de planificare pentru PRAM

- Teorema lui Brent
  - Pentru un algoritm care rulează în $T(n)$ unităţi de timp, executând $W(n)$ operaţii, se poate obţine o adaptare a algoritmului care să ruleze pe $\boldsymbol{p}$ procesoare PRAM în cel mult $inf\left(\frac{W(n)}{p}\right) + T(n)$ unităţi de timp
- Justificare
  - Fie $\boldsymbol{W_i(n)}$ numărul de operaţii executate în unitatea **i** de timp.
  - Pentru fiecare **i**, $1 \leq i \leq T(n)$, $\boldsymbol{W_i(n)}$ operaţii se execută în $\frac{\boldsymbol{W_i(n)}}{\boldsymbol{p}}$ paşi paraleli, pe cele $\boldsymbol{p}$ procesoare.
  - numărul de pași paraleli:

$$\sum_i sup\left(\frac{W_i(n)}{p}\right) \leq \sum_i\left(inf\left(\frac{W_i(n)}{p}\right) + 1\right) \leq inf\left(\frac{W(n)}{p}\right) + T(n)$$

- Exemplu
  - algoritm de însumare executat de $\boldsymbol{p}$ procesoare PRAM
  - $p = 2^q \leq n = 2^k$

Principiul de planificare pentru PRAM (2)

```
var A, B: array [1:n] of real;    /* n = 2^k si p = 2^q */
var S: real;
var l: int := n / p;
fa s := 1 to p do in parallel
   fa j := 1 to l -> B[l*(s-1)+j] := A[l*(s-1)+j] af;
   fa h := 1 to log n ->
      if k-h-q >= 0 ->
         fa j := 2^(k-h-q)(s-1) + 1 to 2^(k-h-q) s ->
            B[j] := B[2*j-1] + B[2*j]
         af
      [] k-h-q < 0 ->
         if s ← 2^(k-h) -> B[s] := B[2*s-1] + B[2*s] fi
      fi
   af
   if s = 1 -> S := B[1] fi
af;
```

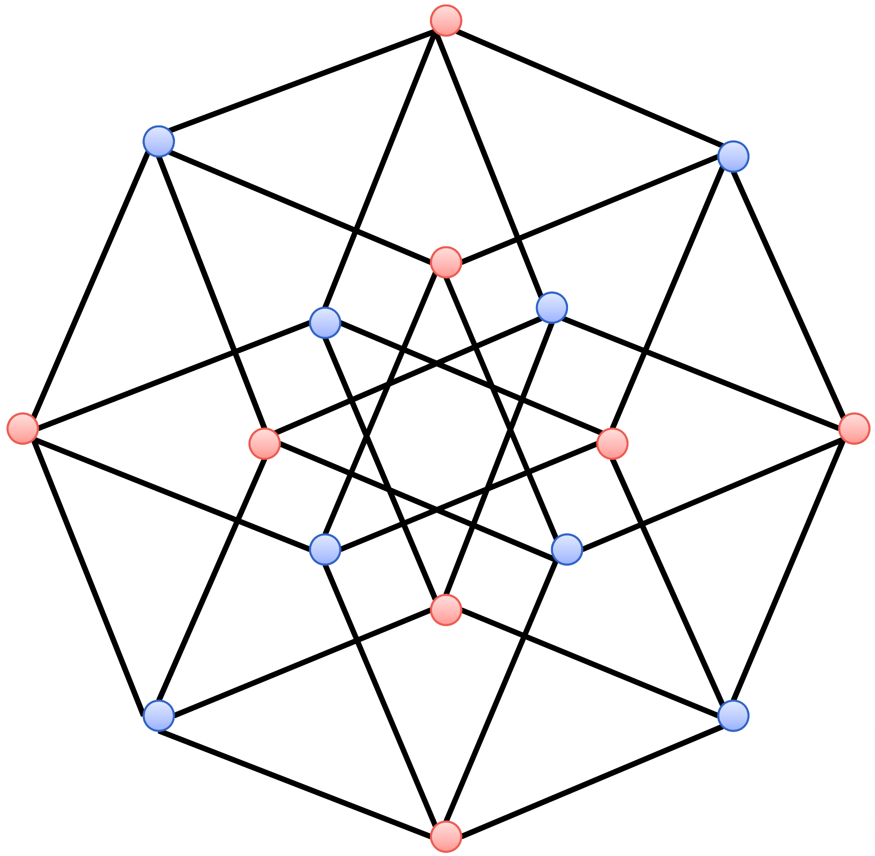
Cerințe pentru algoritmi PRAM eficienți

- **P < n** (dimensiunea problemei), P adaptiv (dependent de dimensiunea problemei)
  - Ex: P(n) funcție subliniară de *n* – rădăcina pătrată de n
- **Tp(n)** mic, adaptiv, invers proporţional cu P
- **C** minim

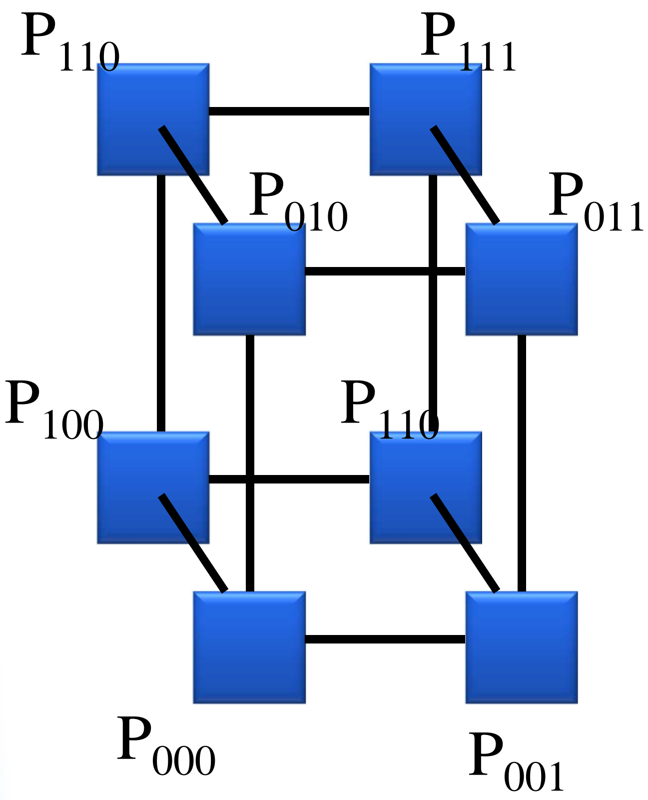
Implementare pentru reţele de procesoare

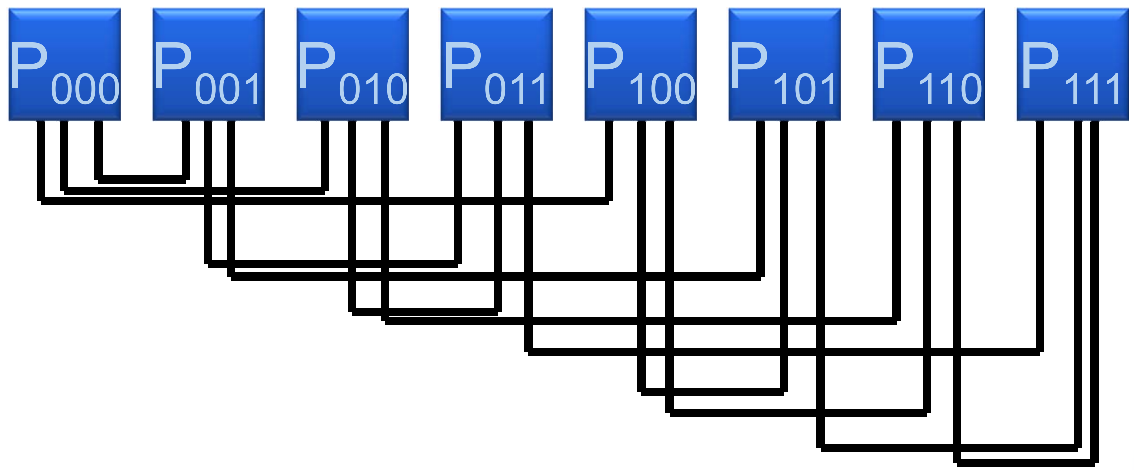
Topologie **hipercub**:

- $p = 2^d$ procesoare, indexate de la 0 la p-1
- reprezentarea binară a lui **i**, $0 \leq i \leq p - 1$, de forma $i_{d-1}i_{d-2}\ldots i_0$
- două procesoare sunt conectate dacă indicii lor diferă doar într-o singură pozitie a reprezentărilor lor binare
- structură recursivă



Implementare pentru reţele de procesoare





Implementare pentru reţele de procesoare (2)

- Algoritmul de însumare:
  - fiecare element *A[i]* al tabloului cu **n** elemente este memorat într-un nod *P[i]* al hipercubului cu **n** noduri $(n = 2^d)$
  - rezultatul acumulat în *A[0]*

- Paşii:
  - A[0] := A[0] + A[4]
  - A[1] := A[1] + A[5]
  - A[2] := A[2] + A[6]
  - A[3] := A[3] + A[7]

A[0] := A[0] + A[4] + A[2] + A[6]

A[1] := A[1] + A[5] + A[3] + A[7]

A[0] := A[0] + A[4] + A[2] + A[6] + A[1] + A[5] + A[3] + A[7]

Implementare pentru reţele de procesoare (2)

```
var A: array [0:n-1] of real;          /* n = 2^d */
fa i := 0 to n-1 do in parallel
   fa l := d-1 to 0 ->
        if i ≤ 2^l - 1 ->
                A[i] := A[i] + A[i(l)]
        fi
   af
af
```

Observație: i(l) denotă indexul obţinut din **i** prin complementarea bitului **l**

Complexitatea: O(log n)

Sumar

- Complexitatea algoritmilor paraleli
  - Măsuri de performanță
  - Calculul detaliat al complexității
  - Limite inferioare
- Sortarea pe un vector de procesoare
- Modele generice
  - Modelul grafurilor orientate aciclice
  - Principiul de planificare pentru PRAM

Întrebări?