Aplicații ale paralelismului de date

Ciprian Dobre
ciprian.dobre@cs.pub.ro

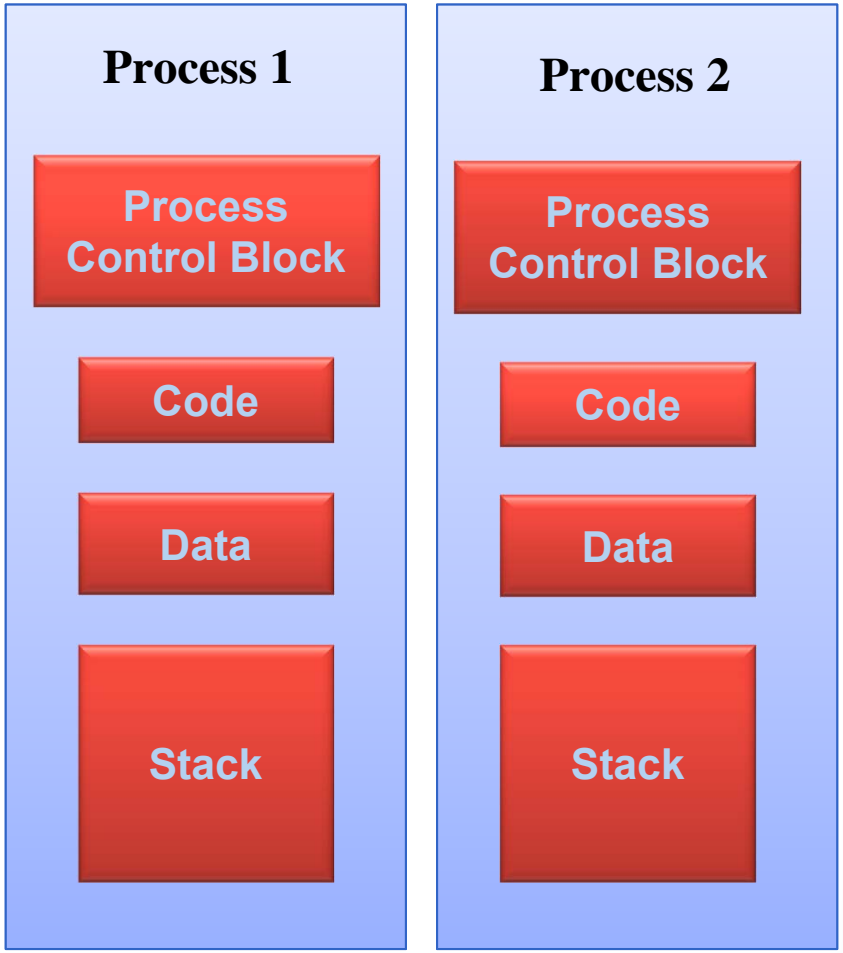
Programarea concurentă în Java

Procese și fire de execuție

- Partajare date în Java

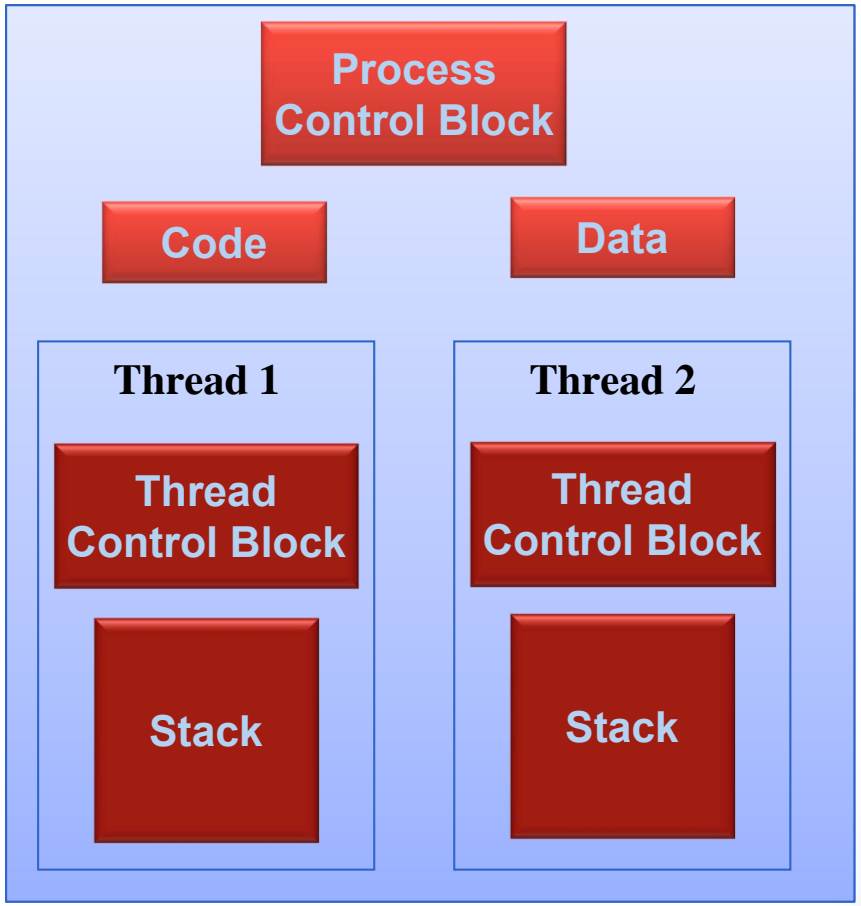
  → Fire de execuție

- Proces
  - Instanță a unui program în execuție
  - Pentru un proces, SO alocă:
    - Un spațiu în memorie (codul programului, zona de date, stiva)
    - Controlul anumitor resurse (fișiere, dispozitive I/O, …)

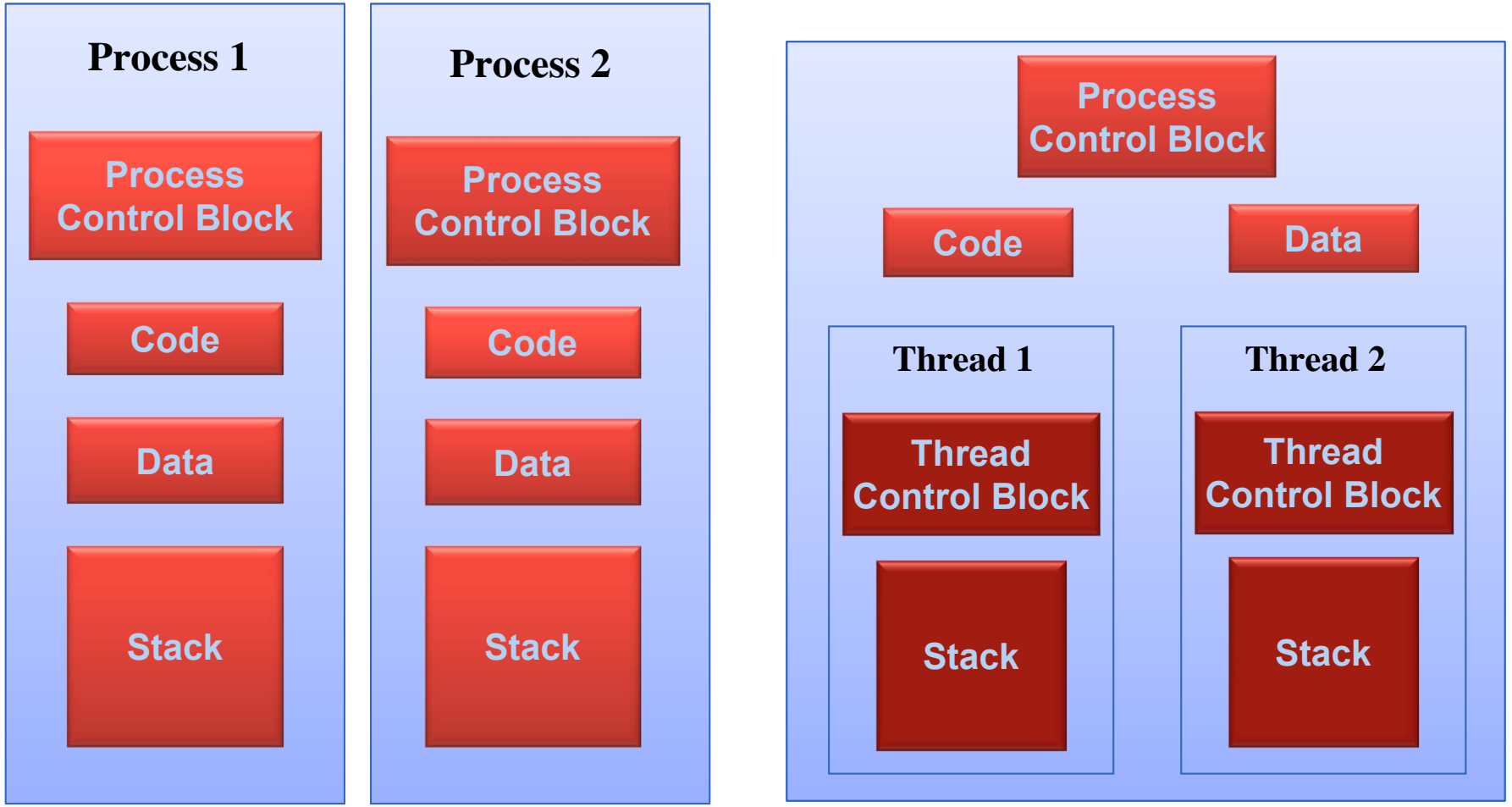


Fire de execuție

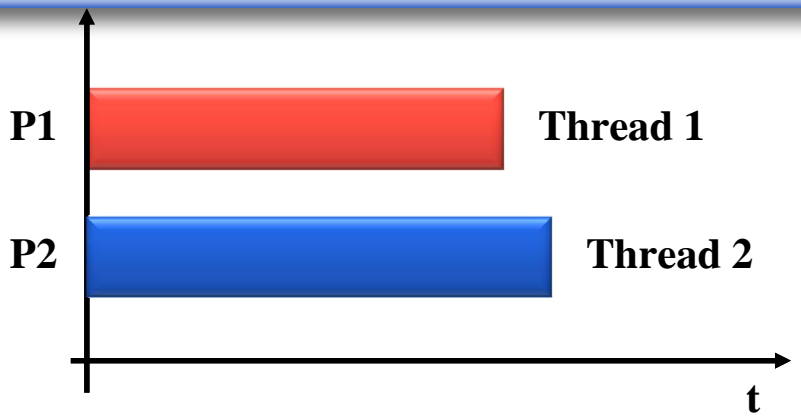
- Un proces poate avea mai multe fire de execuție
- Fir de execuție (thread):
  - Flux de control secvențial în cadrul unui proces
- Firele de execuție
  - împart același spațiu de adrese;
  - au în comun: zonele de cod și date din memorie, resursele procesului
  - zona de stivă – reprezintă starea curentă a thread-ului și este proprie thread-ului



Multi-proces vs. multi-thread

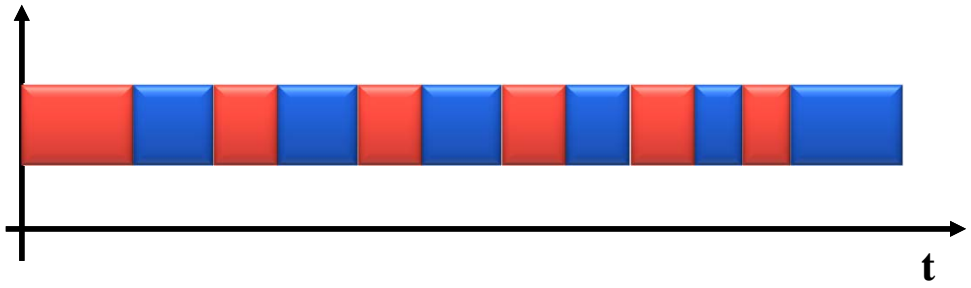


Execuția thread-urilor



Sisteme multiprocesor:

→ execuție în paralel



Sisteme uniprocesor:

→ planificarea este realizată de către sistemul de operare

→ politici: divizarea timpului (*time sharing*), priorități

Avantaje ale utilizării firelor de execuție

- Îmbunătățirea productivității
  - În sistemelor multiprocesor: utilizarea simultană a procesoarelor
  - În general: utilizarea simultană a diverselor resurse
    - Ex: în timp ce se așteaptă la un dispozitiv I/O, se pot executa alte operații cu procesorul
- Îmbunătățirea timpului de răspuns al interfețelor grafice
- Avantaj față de procese: crearea unui nou fir de execuție este mai “ieftină”
- Structurarea mai bună a programelor – mai multe unități de execuție

Programare concurentă în Java

- Limbajul oferă suport pentru fire de execuție
- Avantaj: programarea este independentă de platformă
- Începând cu versiunea 1.5: set variat de clase utilitare pentru programarea concurentă
- Pentru crearea firelor de execuție:
  - Clasa `java.lang.Thread`
  - Interfața `Runnable`
- Metode specifice clasei `Thread`:
  - `start(), sleep(), getPrority(), setPriority()`
- Metode specifice clasei `Object`:
  - `wait(), notify()`

Exemplu: Crearea unui fir de execuție

```
1  Class MyThread extends Thread {
2    public void run() {
3      ...
4      sleep(100);
5      ...
6    }
7  }
8  ...
9  MyThread mt = new MyThread();
10 ...
11 mt.start();
```

```
1  Class MyModule extends Module implements Runnable {
2    public void run() {
3      ...
4      Thread t = Thread.currentThread();
5      t.sleep(100);
6      ...
7    }
8  }
9  ...
10 MyModule mm = new MyModule();
11 Thread mmt = new Thread(mm);
12 ...
13 mmt.start();
```

Controlul execuției thread-urilor

- De către JVM:
  - Prin intermediul priorităților: între `Thread.MIN_PRIORITY (1)` și `Thread.MAX_PRIORITY(10)`
- De către sistemul de operare:
  - Fără divizarea timpului: thread-ul cedează controlul prin metode ca `yield()` sau `wait()`
    - Se poate întampla ca un thread să preia controlul și să se execute fără întrerupere, împiedicând execuția altora
  - Cu divizarea timpului: fiecare thread se execută pentru o cuantă de timp, apoi se trece la execuția altuia etc.

Stările posibile ale unui thread

- Creat: obiectul a fost creat cu operația `new()`; se poate apela metoda `start()`
- Gata de execuție: a fost apelată metoda `start()`, firul poate fi executat
- Suspendat: a fost apelat `sleep()` sau `wait()`
- Terminat: metoda `run()` a fost terminată

Exemplu: Programare concurentă

```
1  class TestMem {
2    private int x = 1;
3    private long y = 2;
4
5    public void modifică() {
6      x = 10;
7      y = 20;
8    }
9
10   public void afișează() {
11     System.out.println("x= " + x);
12     System.out.println("y= " + y);
13   }
14 }
```

Scenariu:

- instanțiem un obiect `x` de tipul `TestMem`
- pornim două fire de execuție
- într-un fir de execuție apelăm `x.modifică()`, iar în celălalt apelăm `x.afișează()`

Întrebare:

Putem ști exact ce se va afișa pe ecran?

Exemplu

```
1  Class FirNeSincronizat extends Thread {
2   FirNeSincronizat(String nume) {
3      super(nume);
4    }
5    void metoda() {
6      System.out.println(getName() + " a=" + a + " b=" + b);
7      a++;
8      try {
9        sleep((int)Math.random() * 1000));
10     } catch (InterruptedException e) {}
11     b++;
12   }
13 public void run() {
14   public void run() {
15     for (int i = 0; i < 3; i++) {
16         metoda();
17     }
18     System.out.println("GATA!");
19   }
20  }
21 }
```

Scenariu de execuție

```
1   class TestSincronizare extends Thread {
2     public static void main (String args[]) {
3       FirNeSincronizat f1 = new FirNeSincronizat("1");
4       FirNeSincronizat f2 = new FirNeSincronizat("2");
5
6       f1.start();
7       f2.start();
8
9       System.out.println("GATA main!");
10    }
11  }
```

Rezultat

```
GATA main!
2 a = 0 b = 0
1 a = 0 b = 0
1 a = 2 b = 1
2 a = 3 b = 2
1 a = 4 b = 3
2 a = 5 b = 4
GATA!
GATA!
```

Sincronizarea firelor de execuție

- **Două situații:**
  - Concurență
  - Cooperare
- **Sincronizarea**
  - asigură ***excluderea mutuală*** – un singur fir poate executa la un moment dat o metodă (secvență de cod) sincronizată: **secțiune critică**
  - Folosește mecanismul de **zăvor** :
    - Fiecare obiect are asociat câte un **zăvor**
    - **synchronized(o)** asigură intrarea în secțiunea critică
    - se poate asocia unei **metode** sau unei **secvențe** de cod
    - pe parcursul execuției secvențelor de cod sincronizate, zăvorul este „închis”

Exemplu: Sincronizare pentru accesul concurent la o resursă (1)

```
1  Class FirSincronizat extends Thread {
2    static int a = 0, b = 0;
3    static Object o = new Object();
4
5    FirSincronizat (String nume) {
6      super(nume);
7    }
8
9    void metodă() {
10     System.out.println(getName() + " a=" + a + " b=" + b);
11     a++;
12     try {
13       sleep((int)Math.random() * 1000));
14     } catch (InterruptedException e) {}
15     b++;
16   }
17
```

Exemplu: Sincronizare pentru accesul concurent la o resursă (2)

```
1     public void run() {
2       for (int i = 0; i < 3; i++) {
3         synchronized(o) {
4           metodă();
5         }
6       }
7       System.out.println("GATA!");
8     }
9   }
10  class TestSincronizare extends Thread {
11    public static void main (String args[]) {
12      FirSincronizat f1 = new FirSincronizat("1");
13      FirSincronizat f2 = new FirSincronizat("2");
14      f1.start(); f2.start();
15      System.out.println("GATA main!");
16    }
17  }
```

Exemplu: sincronizare cu zăvoare

```
1  public class SyncExample {
2      public static class Thingie {
3          private Date lastAccess;
4          public synchronized void setLastAccess(Date date){
5              this.lastAccess = date;
6          }
7      }
8
9      public static class MyThread extends Thread {
10         private Thingie thingie;
11
12         public MyThread(Thingie thingie) {
13             this.thingie = thingie;
14         }
15
16         public void run() {
17             thingie.setLastAccess(new Date());
18         }
19     }
20
21     public static void main() {
22         Thingie thingie1 = new Thingie(),
23         thingie2 = new Thingie();
24         new MyThread(thingie1).start();
25         new MyThread(thingie2).start();
26     }
27 }
```

Metode:

`synchronized`:

lock pe **obiect**

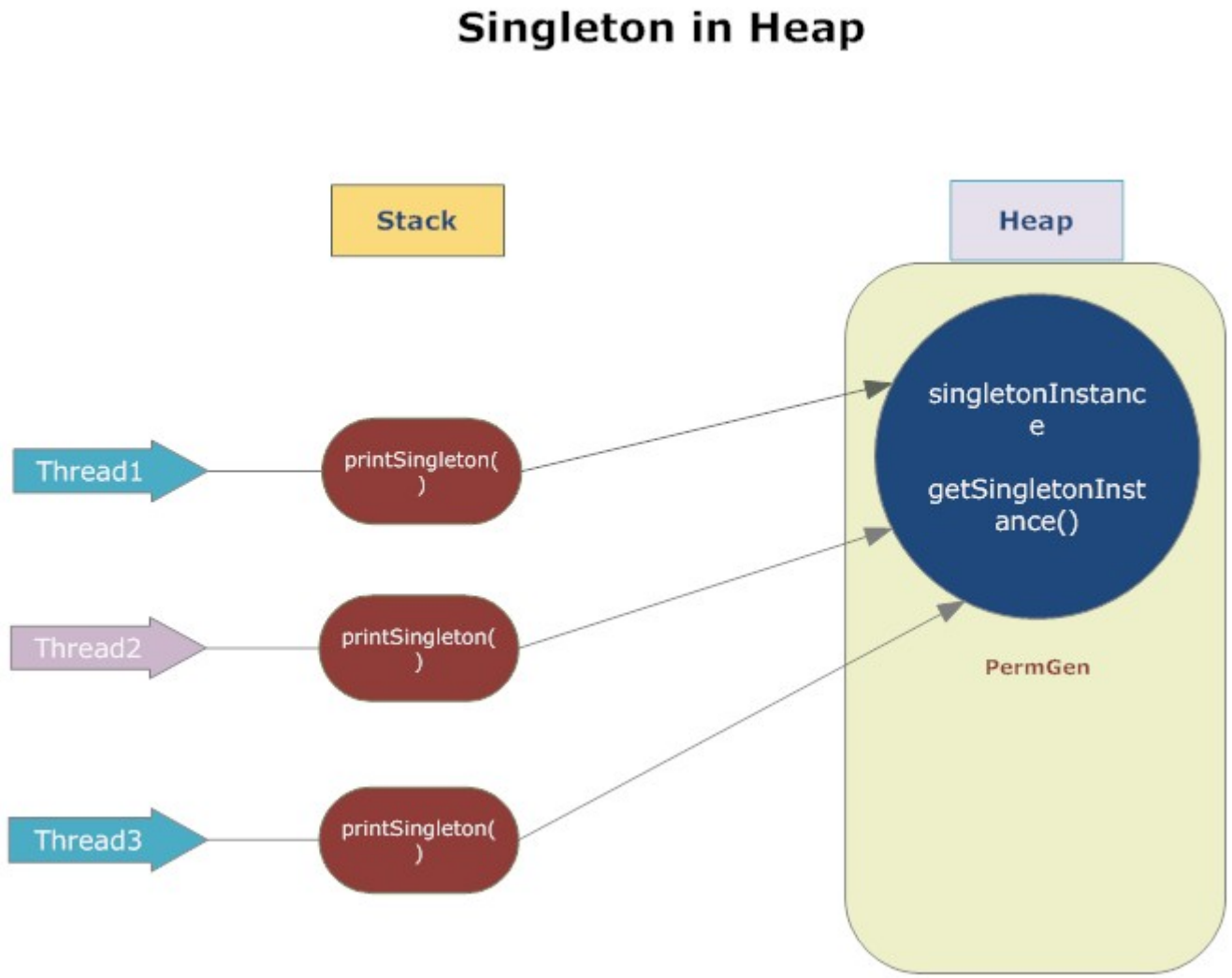
`static synchronized`:

lock pe **clasă**

Exemplul 2 (lazy instantiation)

```
Class Singleton {
   private static Singleton uniqueInstance = null;
   private Singleton( ) { .. } // private constructor
   public static Singleton getInstance( ) {
        if (uniqueInstance == null)
                uniqueInstance = new Singleton();
                // call constructor
        return uniqueInstance;
   }
}
```

Singleton: Double check locking



Singleton: Double check locking

- Când avem mai multe thread-uri, avem nevoie de synchronized:

```
1  public static synchronized SingletonExample getSingletonInstance() {
2          if (null == singletonInstance) {
3              singletonInstance = new SingletonExample();
4          }
5          return singletonInstance;
6      }
```

- De fiecare dată când apelăm însă metoda apare un overhead suplimentar
- În realitate însă nu avem nevoie să forțăm decât verificarea la:

```
1  singletonInstance = new SingletonExample();
```

- Modul acesta de verificare se numește *Double check locking*

Singleton: Double check locking

```
01 public static volatile SingletonExample getSingletonInstance() {
02         if (null == singletonInstance) {
03             synchronized (SingletonExample.class){
04                     if (null == singletonInstance) {
05             singletonInstance = new SingletonExample();
06             }
07         }
08         }
09         return singletonInstance;
10     }
```

- sau….

```
01 public class SingletonExample {
02     private static final SingletonExample singletonInstance = new SingletonExample;
03
04     // SingletonExample prevents any other class from instantiating
05     private SingletonExample() {
06     }
07
08     // Providing Global point of access
09     public static SingletonExample getSingletonInstance() {
10
11         return singletonInstance;
12     }
13
14     public void printSingleton(){
15         System.out.println('Inside print Singleton');
16     }
17 }
```

Metoda `wait()`

- Permite manevrarea zăvorului asociat cu un obiect
- La apelul metodei `wait()` pentru un obiect ***m*** de către un fir de execuție ***t***:
  - se deblochează zăvorul asociat cu ***m*** și ***t*** este adăugat la un set de thread-uri blocate, *wait set*-ul lui ***m***
  - dacă ***t*** nu deține zăvorul pentru ***m***: `IllegalMonitorStateException`
  - ***t*** va continua execuția doar când va fi scos din *wait set*-ul lui ***m***, prin:
    - o operație **notify() / notifyAll()**
    - expirarea timpului de așteptare
    - o acțiune dependentă de implementare

Notificări

- Metode: `notify()`, `notifyAll()`
- La o notificare apelată din thread-ul ***t*** pentru obiectul ***m***:
  - `notify()`: un thread ***u*** din *wait set*-ul lui ***m*** este scos și repus în execuție
  - `notifyAll()`: toate thread-urile sunt scoase din *wait set*-ul lui ***m*** – dar ***numai unul*** va putea obține zăvorul
  - daca ***t*** nu deține zăvorul pentru ***m***: `IllegalMonitorStateException`

Exemplu – sincronizare pentru colaborare (1)

```
1  Class Producător extends Thread {
2    private ZonăTampon tampon;
3    private int număr; // ID-ul producătorului
4
5    public Producător(ZonăTampon z, int număr) {
6      tampon = z;
7      this.număr = număr;
8    }
9
10   public void run() {
11     for (int i = 0; i < 10; i++) {
12       tampon.aTransmis(i);
13       System.out.println("Producător " +
14                          număr + " a transmis: " + i);
15       try {
16         sleep((int)Math.random() * 100));
17       } catch (InterruptedException e) {}
18     }
19   }
20 }
```

Exemplu – sincronizare pentru colaborare (2)

```
1  class Consumator extends Thread {
2    private ZonăTampon tampon;
3    private int număr; // ID-ul consumatorului
4
5    public Consumator(ZonăTampon z, int număr) {
6      tampon = z;
7      this.număr = număr;
8    }
9
10   public void run() {
11     int valoare = 0;
12     for (int i = 0; i < 10; i++) {
13       valoare = tampon.aPreluat();
14       System.out.println("Consumator " +
15                          număr + " a preluat " +
16                          valoare);
17     }
18   }
19 }
```

Exemplu – sincronizare pentru colaborare (3)

```
1  class ZonăTampon {
2    private int valoare; // valoarea curentă din tampon
3    private boolean disponibil = false; //existența unei valori pentru Consum
4    public synchronized int aPreluat() {
5      if (!disponibil) {
6        try {
7          wait();
8        } catch (InterruptedException e) {}
9      }
10     disponibil = false;
11     notify();
12     return valoare;
13   }
14   public synchronized void aTransmis(int valoare) {
15     if (disponibil) {
16       try {
17         wait();
18       } catch (InterruptedException e) {}
19     }
20     this.valoare = valoare;
21     disponibil = true;
22     notify();
23   }
24 }
```

Suportul pentru concurență în JDK 5.0

- Pachet nou: `java.util.concurrent`
- Îmbunătățiri:
  - Schimbări la nivelul mașinii virtuale: exploatarea noilor instrucțiuni disponibile la procesoarele moderne
  - Clase utilitare de bază:
    - Lock-uri
    - Variabile atomice
  - Clase de nivel înalt:
    - Semafoare
    - Bariere
    - Pool-uri de fire de execuție

Clase utile pentru sincronizare în JDK 1.5

- `Semaphore`
- `Mutex`
- `CyclicBarrier`
  - barieră reutilizabilă
  - are ca argument un contor care arată numărul de fire din grup
- `CountDownLatch`
  - similar cu bariera, are un contor, dar decrementarea contorului este separată de așteptarea ajungerii la zero
  - decrementarea semnifică terminarea unor operații
- `Exchanger`
  - *rendez-vous* cu schimb de valori în ambele sensuri între threaduri

Facilități pentru sincronizare de nivel scăzut

- `Lock`
  - generalizare *lock monitor* cu așteptări contorizate, întreruptibile, teste etc.
- `ReentrantLock`
- `Conditions`
  - permit mai multe condiții per lock
- `ReadWriteLock`
  - exploatarea faptului că, la un moment dat, un singur fir modifică datele comune și ceilalți doar citesc
- Variabile atomice:
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicReference`
  - permit execuția atomică *read-modify-write*

Exemplu: Utilizare semafoare in Java 1.5

```
1  final  private Semaphore s = new Semaphore(1, true);
2
3  s.acquireUninterruptibly();
4
5  try {
6    balance = balance + 10; //protected value
7  } finally {
8    s.release(); //return semaphore token
9  }
```

Referințe

- [Ath02] Irina Athanasiu

  Java ca limbaj pentru programarea distribuită, MatrixRom, 2002
- [JLS05] - *The Java Language Specification*

  http://java.sun.com/docs/books/jls/
- [J2SE05N] - *J2SE 5.0 in a Nutshell*

  http://java.sun.com/developer/technicalArticles/releases/j2se15/
- Threads: Basic Theory and Libraries

  http://www.cs.cf.ac.uk/Dave/C/node29.html

Aplicații ale paralelismului de date

Paralelism de date

- Procesoarele rulează *simultan* aceleași taskuri asupra unui set distribuit de date
- *Embarrassingly parallel*
- Aplicații:
  - Vectori
  - Matrice
  - Liste

Aplicații folosind paralelismul de date
Calculul sumelor prefix

Problemă:

Se dă tabloul **a[1:n]**, se cere **s[1:n]**, unde:

$$s[i] = \sum_{k=1}^{i} a[k]$$

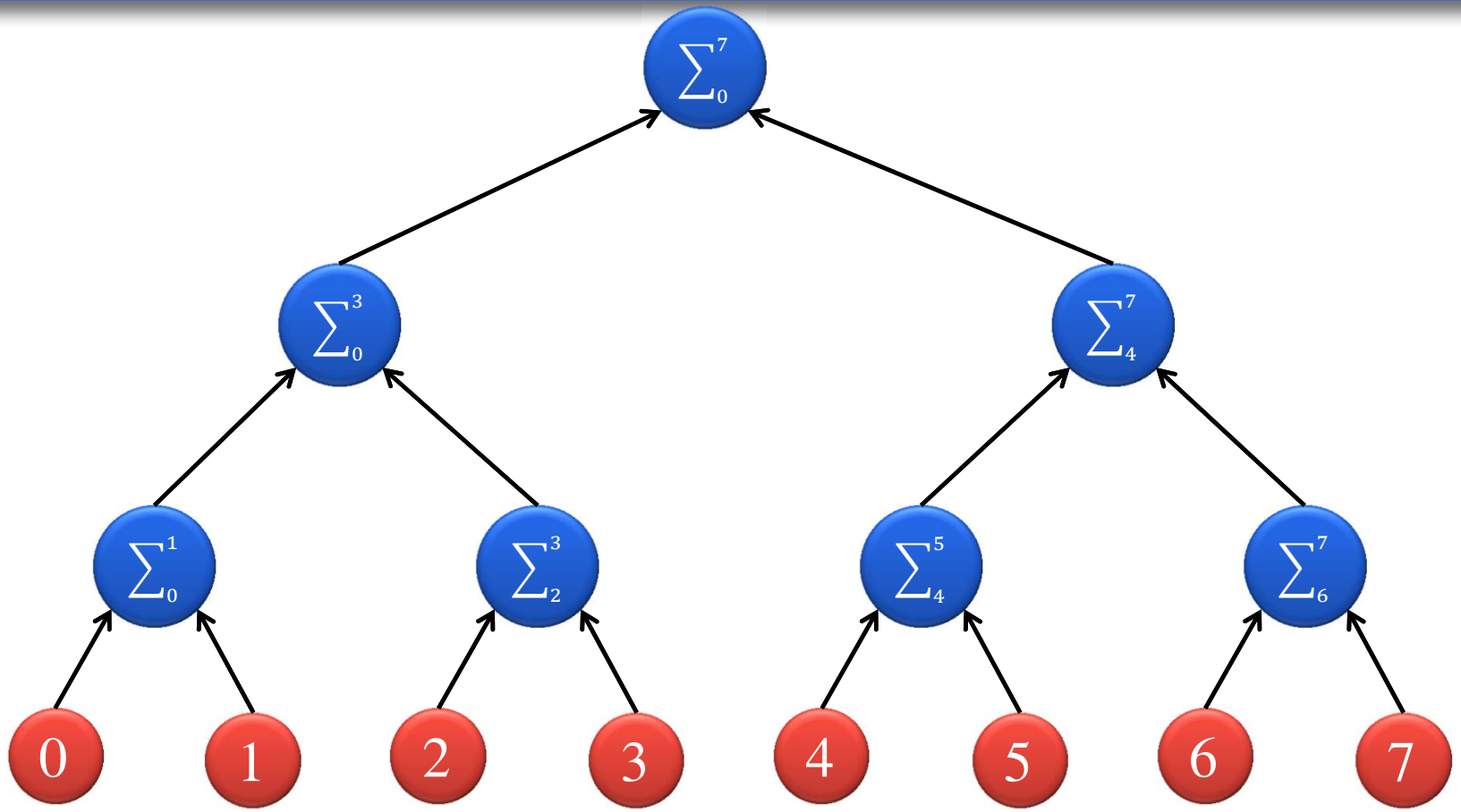
- Algoritm secvențial:

```
s[1] := a[1];
fa i := 2 to n ->
   s[i] := a[i] + s[i-1]
af
```

- Algoritm paralel:

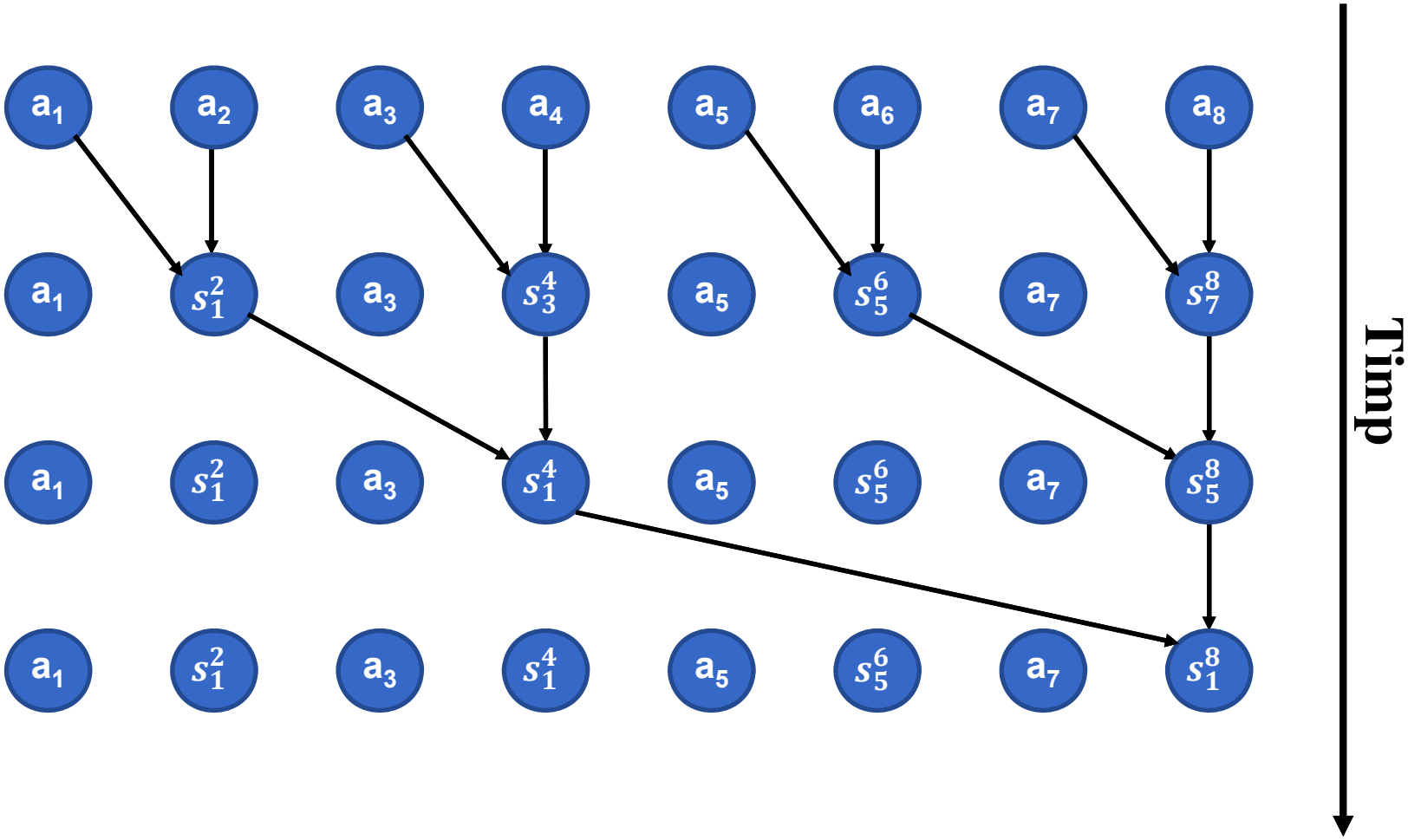
  - derivat din algoritmul ***sumei elementelor unui vector***

Suma elementelor unui vector



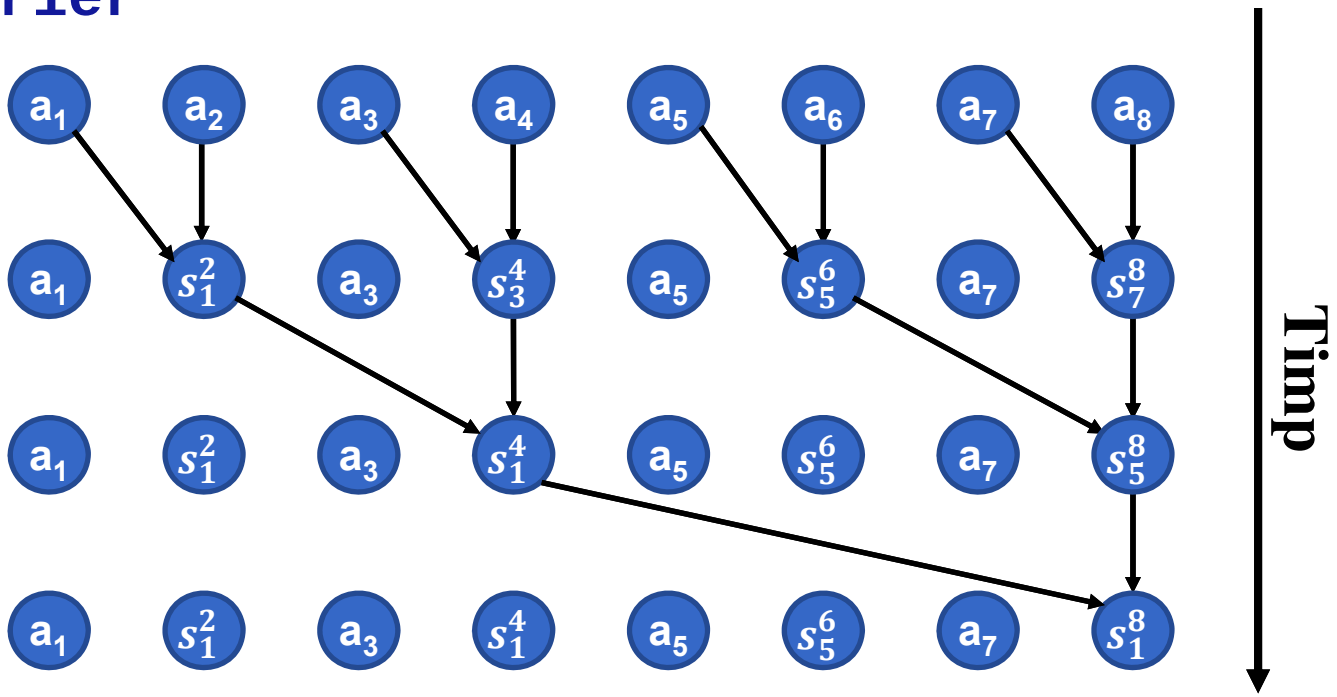
$\sup\left(\frac{n}{2}\right)$ procesoare, $\sup(\log_2 n)$ pași

Suma elementelor unui vector

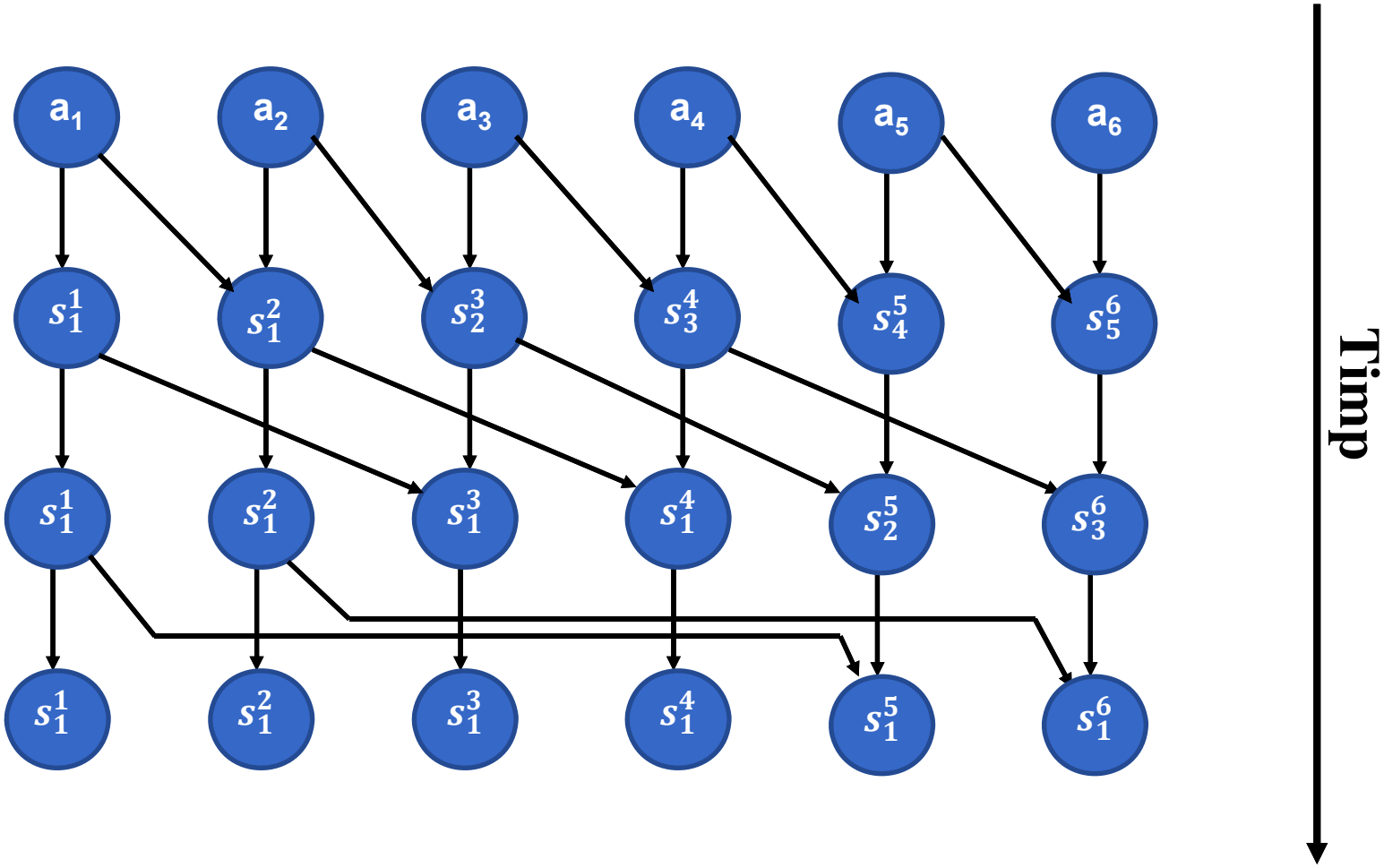


Suma elementelor unui vector

```
var a: array [1:n] of int;
co suma (k:1..n)::
      fa j := 1 to sup(log2 n) ->
            if k mod 2^j = 0 ->
                  a[k] := a[k-2^j-1] + a[k]
            fi
            barrier
      af
oc
```



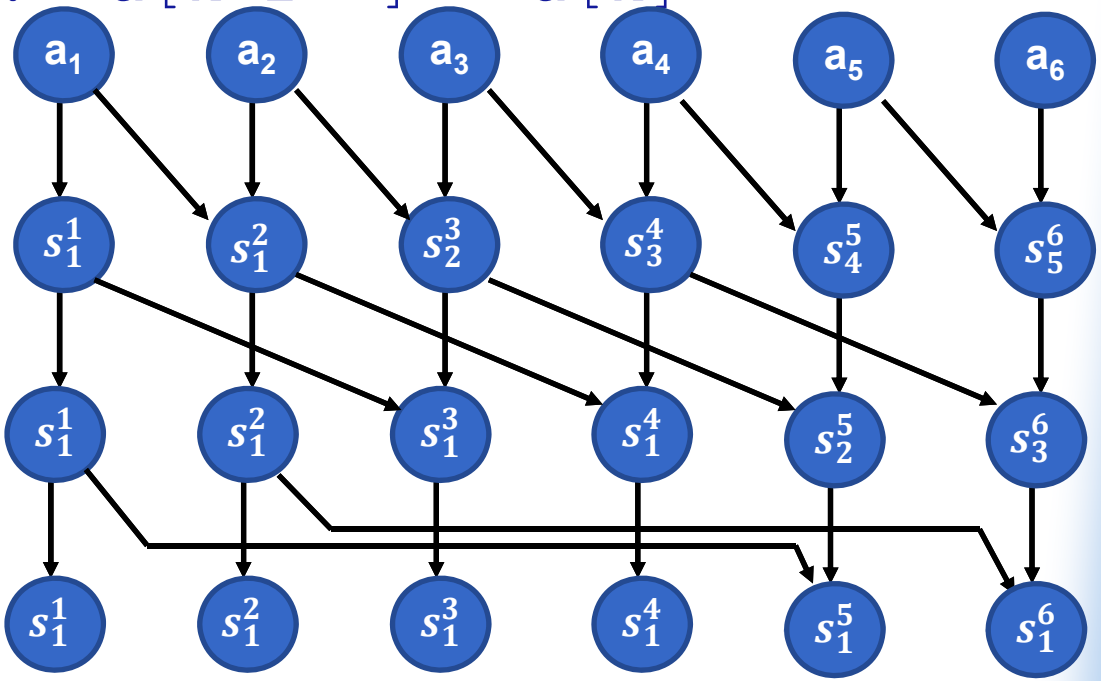
Sume prefix (varianta 1)



Sume prefix – varianta 1

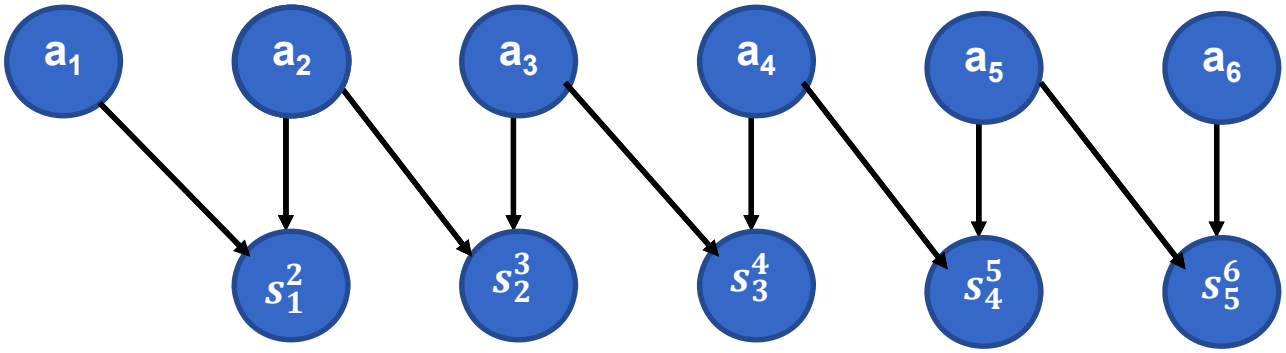
```
var a: array [1:n] of int;
co suma(k:1..n)::
      fa j := 1 to sup(log2 n) ->
            if k - 2^(j-1) >= 1 ->
                  a[k] := a[k-2^(j-1)] + a[k]
            fi
            barrier
      af
oc
```

Eroare de sincronizare...



Sume prefix – varianta 1 – probleme

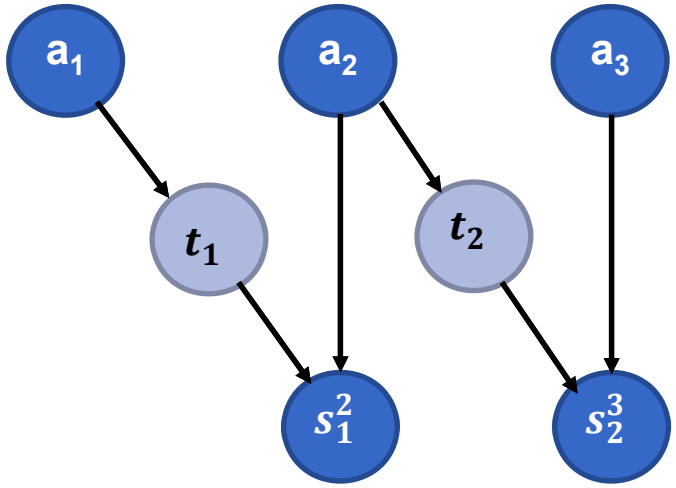
- Presupunem că procesorul numărul 3 este mai lent decât restul.
  - Suprascriere a locației de memorie



$$s_2^3 = a_3 + s_1^2$$

Sume prefix – varianta 2

```
var a, temp: array [1:n] of int;
co suma(k:1..n)::
     fa j := 1 to sup(log2 n) ->
          temp[k] := a[k];
          barrier
          if k - 2^(j-1) >= 1 ->
                    a[k] := temp[k-2^(j-1)] + a[k]
          fi
          barrier
     af
oc
```



Sume prefix – varianta 3

```
var a, temp: array [1:n] of int;
co suma(k:1..n)::
   var d := 1;
   do d < n ->
      temp[k] := a[k];
      barrier
      if k - d >= 1 -> a[k] := temp[k-d] + a[k]
      fi
      barrier
      d := 2 * d
   od
oc
```

Notație SIMD

```
do steps i to j in parallel
        step i
        ...
        step j
od

fa i := j to k do in parallel
        operaţiile lui Pi
af

fa i := r, s, ...t do in paralel
        operaţiile lui Pi
af

fa i in S do in paralel
        operaţiile lui Pi
af
```

Sume prefix – varianta 1 SIMD

```
var a: array [1:n] of int;
fa k := 1 to n do in parallel
    (Procesor P_k)
    var temp: int; /* locală P_k */
    var d := 1;
    do d < n ->
      if k - d >= 1 -> temp := a[k-d];
                       a[k] := temp + a[k]
      fi
      d := 2 * d
    od
af
```

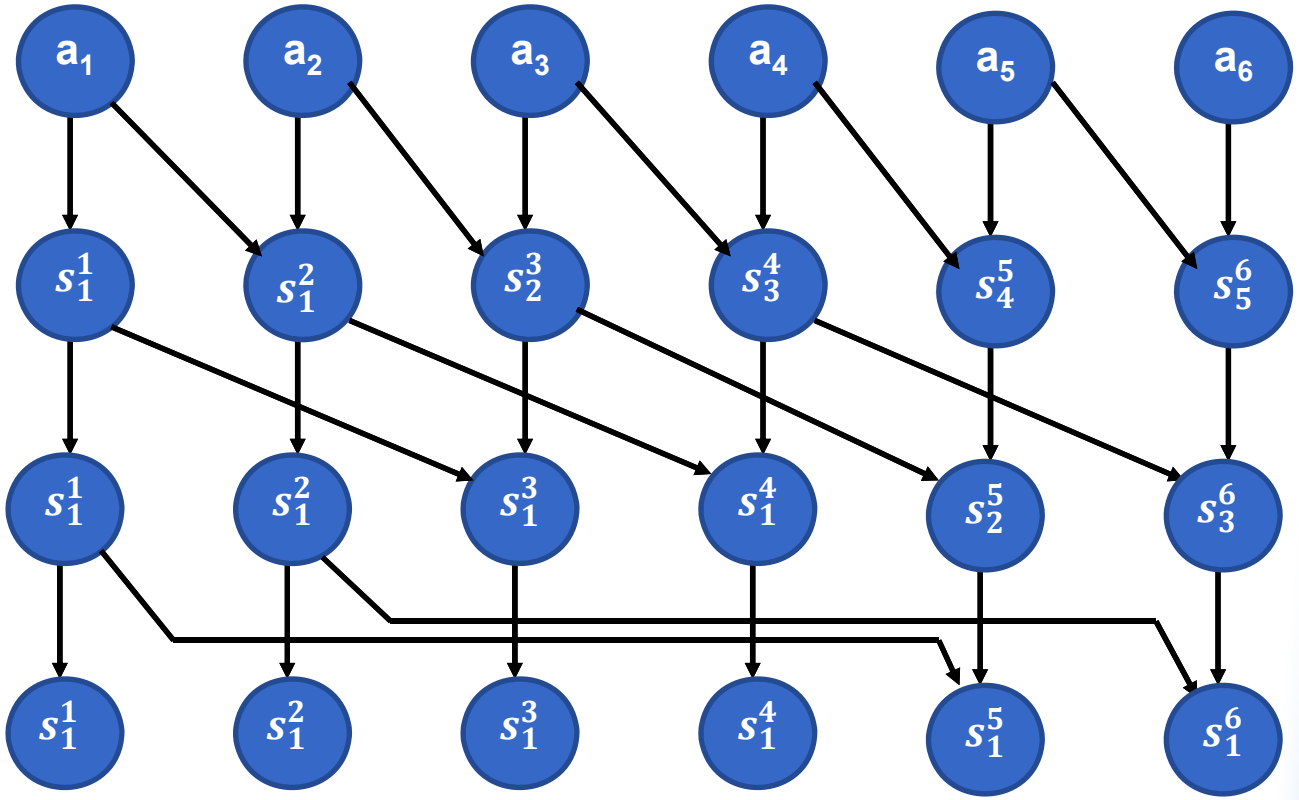
Sume prefix – varianta 1 SIMD

- Se observă că, la anumiți pași, unele procesoare nu operează.

Pas 1: ***P1***

Pas 2: ***P1***, ***P2***

Pas 3: ***P1***, ***P2***, ***P3***, ***P4***



Sume prefix – varianta 2 SIMD

- Varianta pune în evidență procesoarele care lucrează.

```
var A: array [1:N] of real;
fa j := 0 to log N - 1 do
  fa k := 2^j+1 to N do in parallel
      (Procesor P_k)
           var temp: real;
           /* temp locala procesorului k */
           1. temp := A[k-2^j];
           2. A[k] := temp + A[k]
  af
af
```

Operații cu vectori – broadcast

Pas 1: (Procesorul P_1)

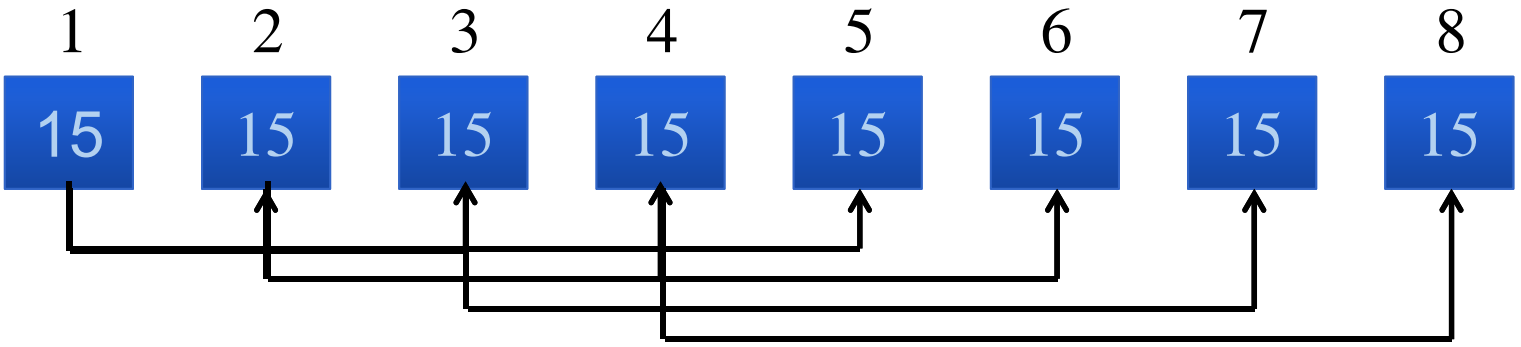
```
   var t: real;/* t locala P1 */
   t := D;
   A[1] := t;
```

Pas 2:

```
   fa i = 0 to (log N -1) do
       fa j = 2^i+1 to 2^i+1 do in parallel
            /* (Procesor Pj) */
            var t: real; /* t locală Pj */
            t := A[j-2^i]
            A[j] := t;
       af
   af
```

Operații cu vectori – broadcast

- Ex: dorim să propagăm valoarea *15* la toate procesoarele.



Operații cu matrice - produs

```
var a, b, c: array [1:n, 1:n] of
  real;

co Prod(i:1..n, j:1..n)::
      var sum : real := 0;
      fa k := 1 to n ->
         sum := sum + a[i,k] * b[k,j]
      af
      c[i,j] := sum
oc
```

Operații cu matrice - grilă

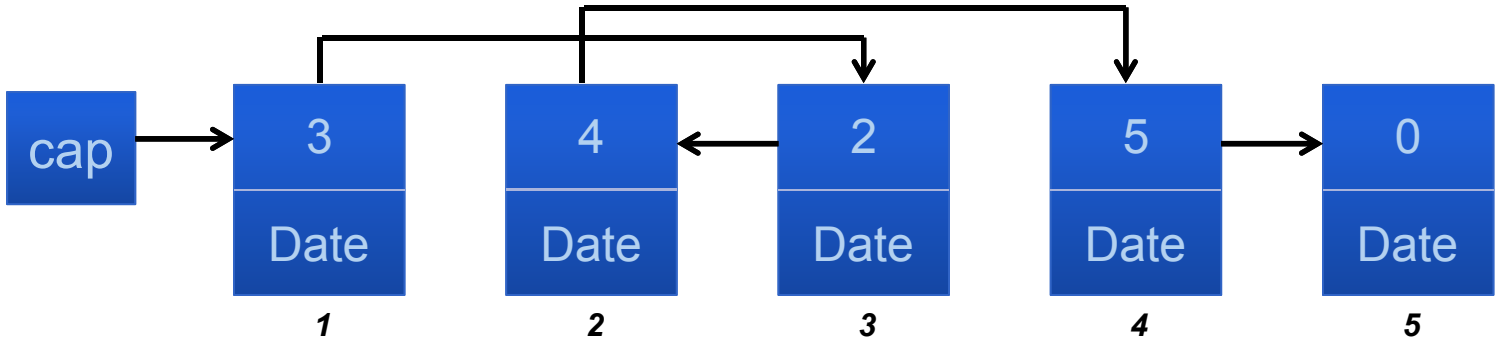
- Soluție în două dimensiuni pentru Laplace:
$$\Delta^2(\Phi) = 0$$
- Grila `[0:n+1, 0:n+1]`: matrice de puncte
- Diferențe finite: la fiecare iterație – un punct interior ca media aritmetică a vecinilor
- Metoda staționară: soluția converge când noile valori diferă de vechile printr-un ε

Operații cu matrice - grilă

```
var grila, noua: array [0:n+1, 0:n+1] of
  real;
var converge: boolean := false;
co CalculGrilă(i:1..n, j:1..n)::
     do not converge ->
       noua[i,j] := (grila[i-1,j]+
                     grila[i+1,j]+
                     grila[i,j-1]+
                     grila[i,j+1])/4;
       barrier
       test_convergență;
       barrier
       grilă[i,j] := nouă[i,j];
       barrier
     od
oc
```

Operații cu liste

- Listă *n* elemente: `data[1:n]`
- Legăturile între elemente: `leg[1:n]`
- Capul listei: `cap`
- Elementul `i` face parte din listă ⇔ fie
  - `cap = i`
  - există un element `j`, între `1` și `n`, a.î. `leg[j]=i`

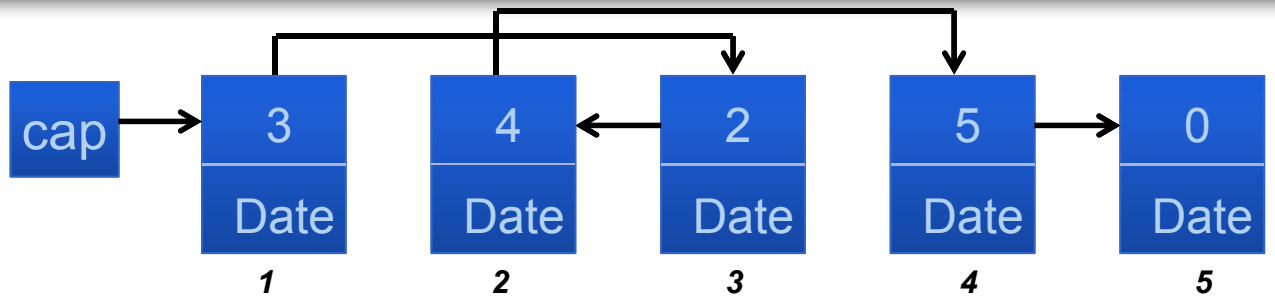


- Problemă: vrem ca fiecare procesor să afle capătul listei

Operații cu liste

```
var leg, end: array [1:n] of int;
co Află (i:1..n)::
     var nou: int; d:int:=1;
     end[i] := leg[i];
     barrier
     do d<n ->
          nou := 0;
          if end[i]<>0 and end[end[i]]<>0 ->
               nou:=end[end[i]] fi
          barrier
          if nou<>0 -> end[i]:=nou fi
          barrier
          d := 2*d
     od
oc
```

Operații cu liste



Procesor	End[i]	End[i]	End[i]	End[i]
1	3	2	5	5
2	4	5	5	5
3	2	4	5	5
4	5	5	5	5
5	0	0	0	0
	init	*d = 1* *d < 5*	*d = 2* *d < 5*	*d = 4* *d < 5*

```
var leg, end: array [1:n] of int;
co Află (i:1..n)::
   var nou: int; d:int:=1;
   end[i] := leg[i];
   barrier
   do d < n ->
      nou := 0;
      if end[i]<>0 and end[end[i]]<>0 ->
         nou := end[end[i]]
      fi
      barrier
      if nou <> 0 ->
         end[i] := nou
      fi
      barrier
      d := 2 * d
   od
oc
```

Varianta SIMD

```
var leg, end: array [1:n] of int;
fa i:=1 to n do in parallel
  end[i] := leg[i];
  do end[i] <> 0 and end[end[i]] <> 0 ->
     end[i] := end[end[i]];
  od
af
```

Sumar

- Suportul pentru concurență în Java
- Aplicații ale paralelismului de date
  - Sume prefix
  - Notații SIMD
  - Operații cu liste

Întrebări?