

Algoritmi Paraleli și Distribuiți

Ciprian Dobre
ciprian.dobre@cs.pub.ro

Reguli, punctaje, ...



Despre curs (1)

Materiale
Forum
Teme

<http://curs.cs.pub.ro>

	Luni		Marti		Miercuri		Joi		Vineri	
08-10							331CC	Dragos Comaneci		
10-12					331CC	Mihai Carabas	333CC	Dragos Comaneci		
12-14					333CC	Eliana Tirsa				
14-16	332CC	Eliana Tirsa					332CC	Gabriel Gutu		
16-18							334CC	Alecsandru Patrascu		
18-20							334CC	Alecsandru Patrascu		

Despre curs (2)

- Nota:

$$10 = 5 \text{ (Laborator)} + 1.5 \text{ (Curs)} + 0.5 \text{ (Supl)} + 4 \text{ (Examen)}$$

Laborator+Curs ≥ 3 Examen ≥ 2

Laborator: 4 teme(4)+activitate(1)

Curs: lucrări de curs (1) + teste la laborator (0.5)

Examen: scris (4) + problemă suplimentară (0.5)

Întrebări?

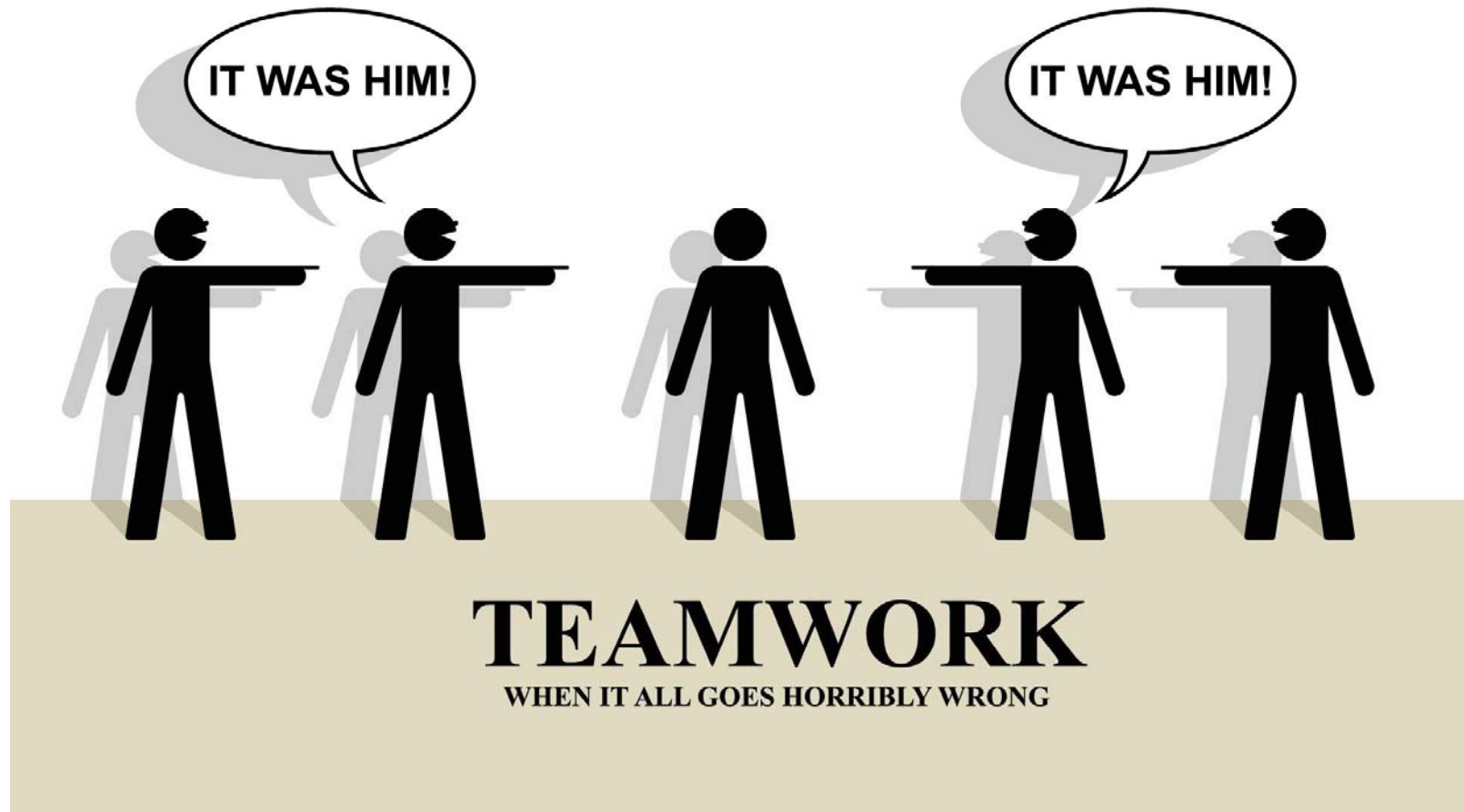
Despre curs (3)

- Interactivitate – dialog
- Open-office: ????, EG403
- Regulament afișat pe pagina cursului
- Punctajele obținute pe parcurs sau examen se pot pastra pentru în an universitar (nu acumulare)
- Temele se pot trimite doar pe parcursul primului semestru universitar
- Mărirea de notă nu se poate face fără re-susținerea examenului

Vineri, 10-12 AM



Interactivitate - teamwork



Despre mine

Absolvent al Facultatii de
Automatica si Calculatoare



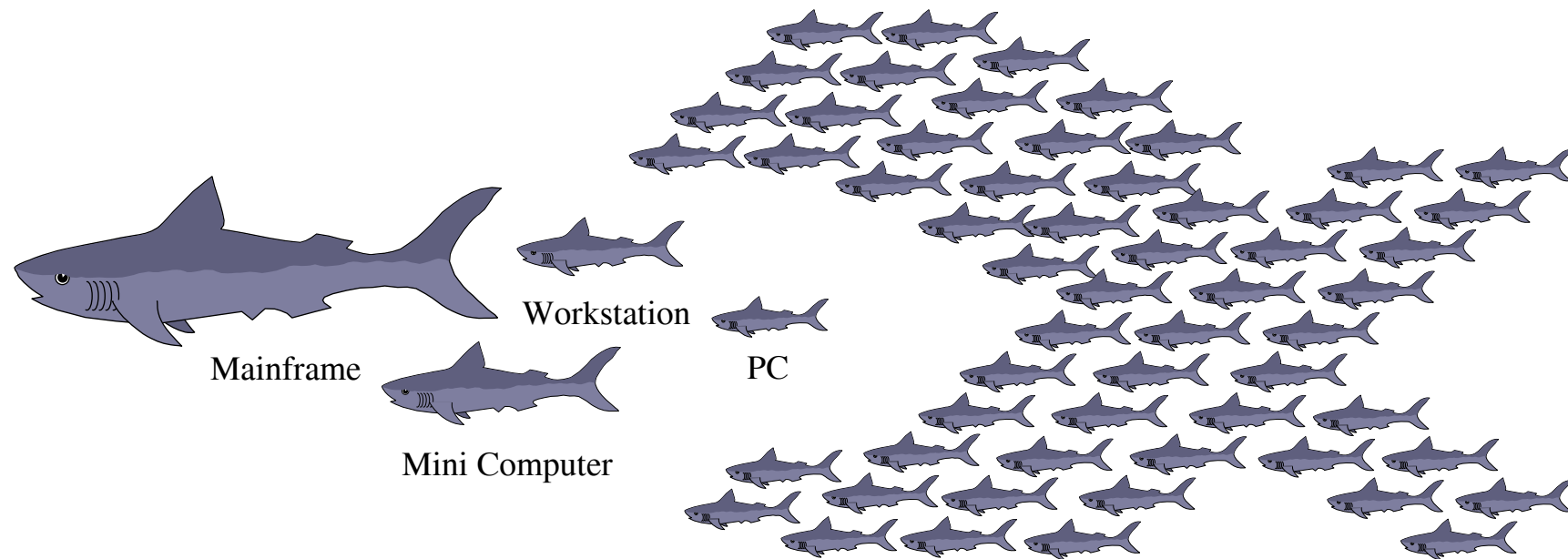
<http://cipsm.hpc.pub.ro>

E-mail: ciprian.dobre@cs.pub.ro

Facebook: <http://www.facebook.com/ciprian.dobre>



Despre curs...



Bibliografie

G.R.Andrews
Concurrent Programming. Principles and Practice
The Benjamin/Cummings Publishing Company, Inc., 1991

G.R.Andrews
Foundations of Multithreaded, Parallel, and Distributed Programming
Addison Wesley, Inc., 2000

F. Thomson Leighton
Introduction to Parallel Algorithms and Architectures: Arrays. Trees. Hypercubes.
Morgan Kaufmann Publishers, San Mateo, California, 1992

A.G. Akl
Parallel Computation. Models and Methods
Prentice Hall 1997

M.J. Quinn
Parallel Computing. Theory and Practice
McGraw-Hill, 1994

Claudia Leopold
Parallel and Distributed Computing
John Wiley & Sons, 2001

A.Y.H. Zomaya
Parallel and Distributed Computing
McGraw-Hill, 1996

Gerard Tel
Introduction to Distributed Algorithms
Cambridge University, 1994

Robert W. Sebesta
Concepts of Programming Languages (Second Edition)
The Benjamin Cummings, 1993

S.A. Williams
Programming Models for Parallel Systems
John Wiley & Sons, 1990

K.M. Chandy, J.Misra
Parallel program design.
A foundation Addison-Wesley Publishing Company, 1988

M. Ben Ari
Principles of concurrent and distributed programming
Prentice Hall, N.Y. 1990

H.Ball
Programming distributed systems
Prentice Hall, NY 1990

C.A.R. Hoare
Communicating Sequential Processes
Comm. of the ACM, 1978

S.G. Akl
The Design and Analysis of Parallel Algorithms
Prentice Hall, 1989

G.R. Andrews, R.A. Olsson
The SR Programming Language. Concurrency in Practice
The Benjamin/Cummings Publishing Company, Inc., 1993

Ian Foster
Designing and Building Parallel Programs
Addison-Wesley Publishing Company, 1995

A.S. Tanenbaum
Structured Computer Organization (Fourth Edition)
Prentice Hall, 1999

Curs

1.	Introducere - Limbaje de descriere a algoritmilor paraleli. Concurența și sincronizare. Atomicitate. Bariere.	3 ore
2.	Paralelism de date - Calcule prefix. Prelucrări de liste și matrice.	3 ore
3.	Complexitatea calculului paralel - Măsuri de performanță. Calculul complexității. Proprietăți ale modelului de evaluare. Modelul Work-depth.	3 ore
4.	Dezvoltarea algoritmilor folosind variabile partajate	2 ore
5.	Dezvoltarea aplicațiilor pentru modele PRAM - Căutarea paralelă. Selecția paralelă.	3 ore
6.	Comunicarea prin mesaje - Biblioteci pentru programare distribuită. MPI.	3 ore
7.	Complexitatea calculului distribuit - Modelul Foster. Modelul LogP.	2 ore

Curs (2)

8.	Ceasuri logice și ordonarea evenimentelor - Vectori de timp (vector timestamps).	3 ore
9.	Algoritmi undă – Descriere și proprietăți. Algoritmii inel, arbore, ecou. Algoritmul fazelor. Algoritmul lui Finn.	3 ore
10.	Stabilirea topologiei	2 ore
11.	Terminarea programelor distribuite	3 ore
12.	Algoritmi pentru sisteme tolerante la defecte	3 ore
13.	Alegerea liderului	3 ore
14.	Algoritmi pentru excludere mutuală	3 ore

Obiectiv

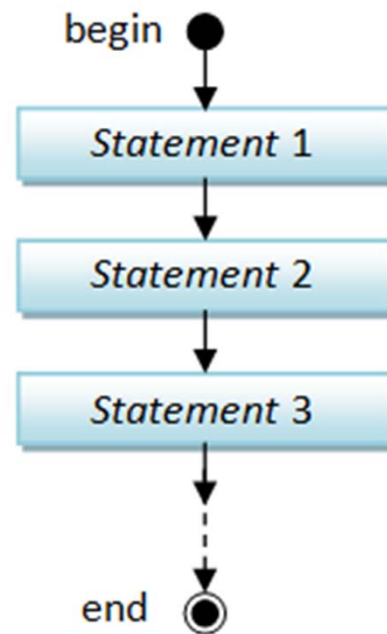
Acumularea competentelor necesare pentru rezolvarea problemelor prin solutii paralele sau distribuite

- **Calcul paralel** = impartirea unei aplicatii in task-uri executate simultan
- **Calcul distribuit** = impartirea unei aplicatii in task-uri executate in sisteme diferite (cu resurse diferite)

APD difera de algoritmi secventiali

- Au la baza **concepte** diferite
 - ♦ *Communicating Sequential Processes (Hoare)*
 - Concurenta
 - Atomicitate
 - Sincronizare
- Folosesc **modele** de programare care asigura comunicarea intre procese prin
 - ♦ Date partajate
 - ♦ Comunicare de mesaje
- ➔ Algoritmi paraleli si distributi NU sunt simple extensii sau versiuni ale celor secventiali
- ➔ Sunt folosite **abordari** diferite

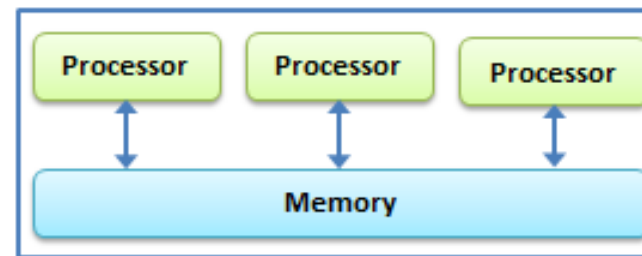
APD difera de algoritmii secventiali



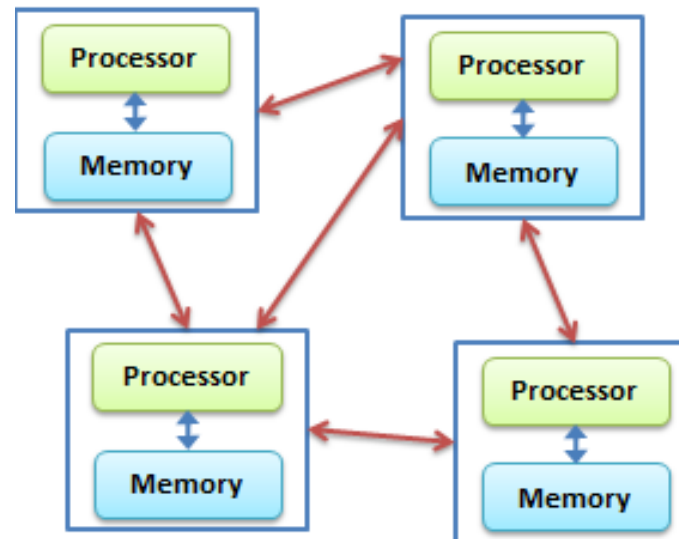
Sequential



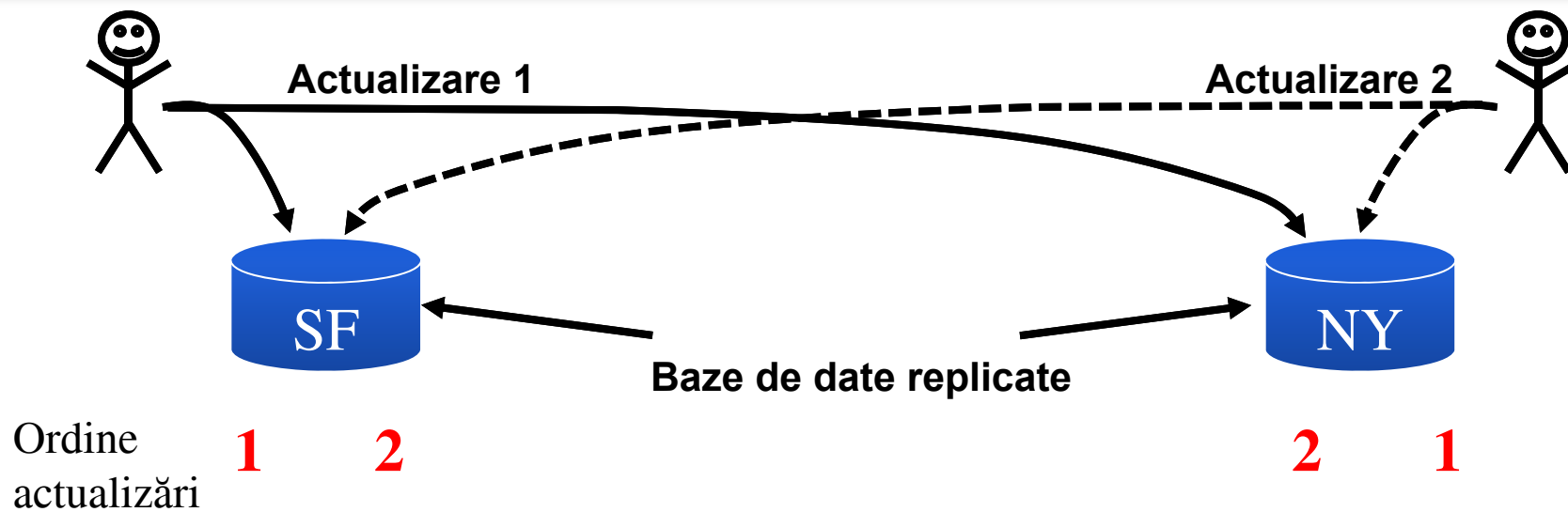
Parallel Computing



Distributed Computing



Exemplu



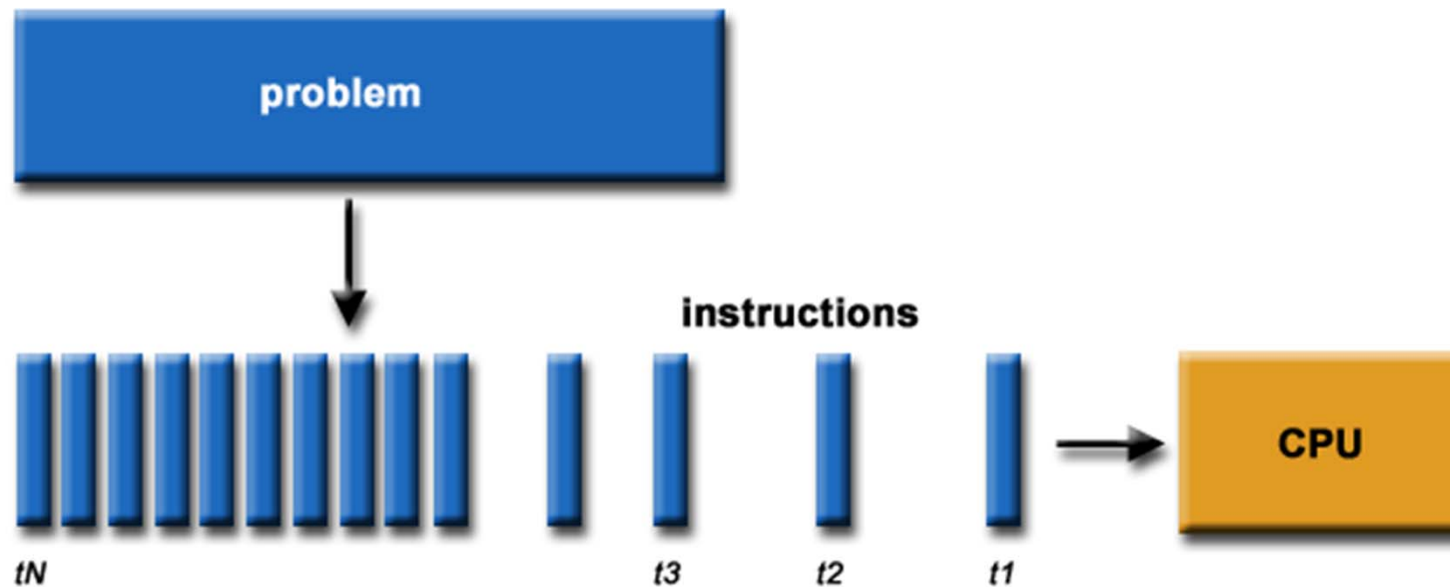
- ♦ Două conturi replicate în New York(NY) și San Francisco(SF)
- ♦ Două actualizări în același timp:
 - Soldul curent: \$1,000
 - Actualizare1: Adaugă \$100 la SF;
 - Actualizare2: Adaugă dobânda de 1% la NY
 - Whoops, stări inconsistente!

La terminare veti cunoaste

- Conceptele de baza
- Modelele de programare
- Metode de proiectare a solutiilor paralele si distribuite
- Modalitati de implementare a solutiilor folosind limbaje de programare / biblioteci
 - ♦ Java concurrent
 - ♦ MPI
- Metode de imbunatatire a performantei solutiilor folosind modele de complexitate

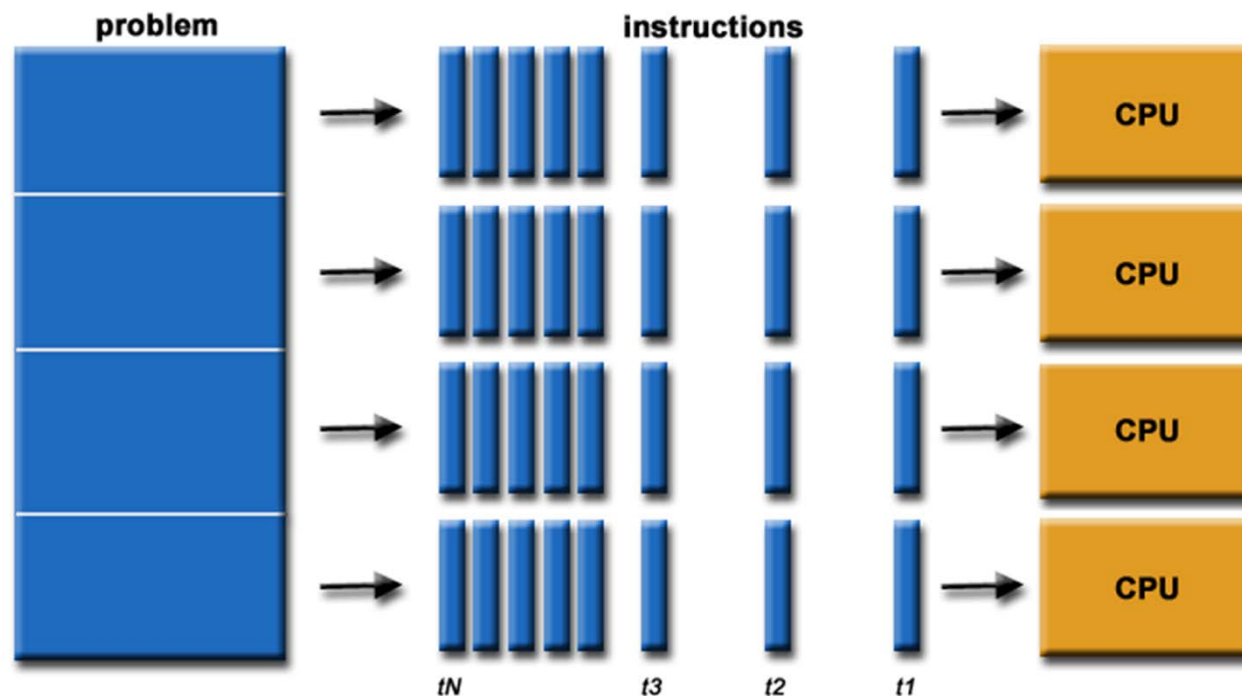
Ce este calculul paralel?

- abordarea *serială*:



Ce este calculul paralel? (2)

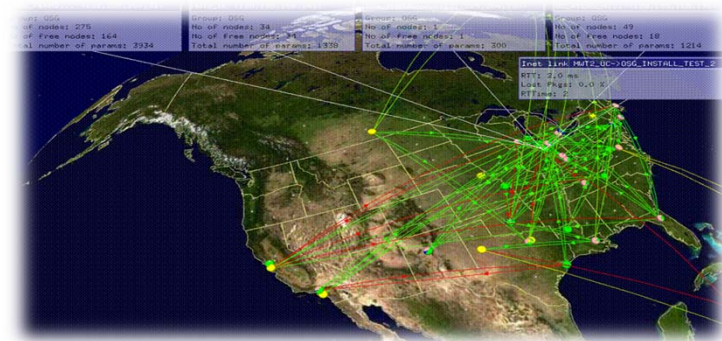
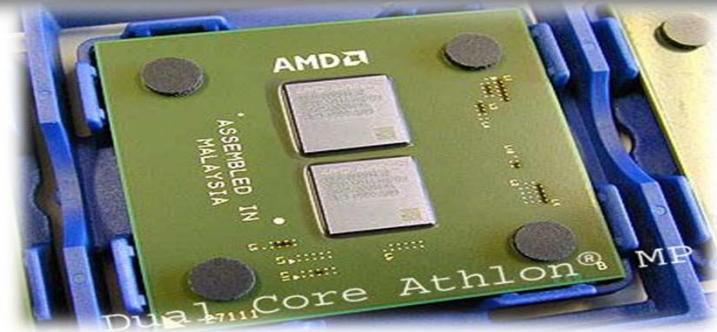
- abordarea *paralelă*:



→ (*simplificat*): folosirea simultană a mai multor **resurse de calcul** pentru rezolvarea unei **probleme computaționale**

Resurse de calcul

- Un singur computer cu mai multe CPU
 - ♦ *DualCore, QuadCore, HPC*
- Mai multe computere conectate într-o rețea
 - ♦ *Clustere*
- Combinație între cele două
 - ♦ *Grid-uri, Sisteme Cloud*



Problemă computațională

- Caracteristici:
 - ♦ Poate fi divizată în **părți discrete** ce pot fi rezolvate simultan
 - ♦ Poate executa mai multe **instrucțiuni concomitent**
 - ♦ Poate fi rezolvată în **mai puțin timp** cu resurse multiple decât cu o singură resursă

De ce calcul paralel? (2)

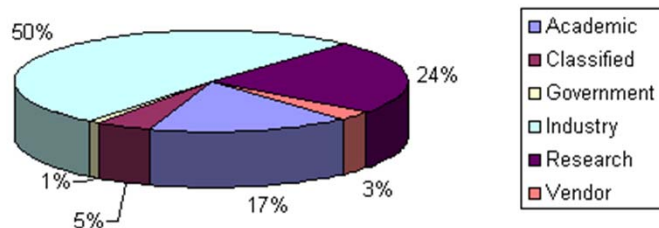
- Timp mai puțin (*wall clock time*)
- Probleme de dimensiuni mai mari
- Concurență (*procesări simultane*)
- Folosirea resurselor nelocale
- Reducerea costurilor
- Depășirea constrângerilor de memorie

De ce calcul paralel? (3)

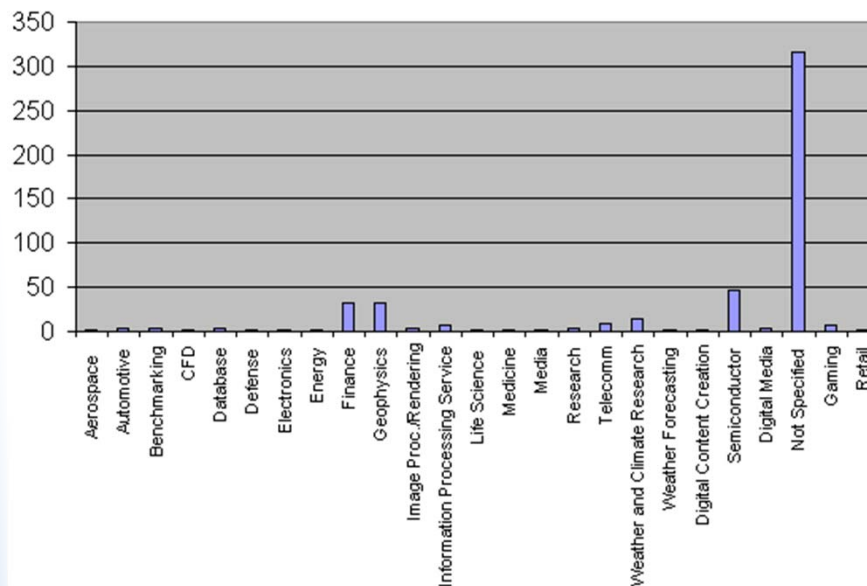
- Limitări ale arhitecturilor seriale:
 - ♦ Viteze de transmisie – dependențe de hardware
 - ♦ Limitări de miniaturizare – chiar și în *molecular computing*!
 - ♦ Limitări economice – costuri ridicate pentru a realiza *un* procesor mai rapid (*ex: Intel, IBM Cell*)

Cine și ce?

Who's Doing Parallel Computing?



What Are They Using it For?



“Great Challenge Problems”:

- Fizica nucleară
- Climă, meteo
- Biologie – genomul uman
- Geologie – activitate seismică
- Electronică – circuite
- Medicină – imagistică

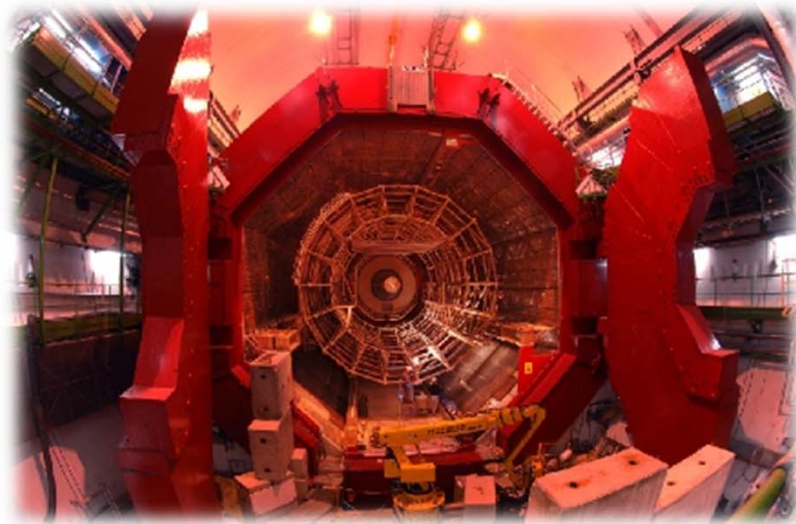
Aplicații comerciale:

- Baze de date paralele – data minning
- Motoare de căutare
- Collaborative work
- Realitate virtuala (gaming), grafica
- Networked video
- Aviație - modelare

Cine și ce? (2)

Experimentul **ALICE** la CERN:

- Unul din cele 4 experimente LHC, dedicat fizicii ionilor grei
- Volum de date:
 - ♦ 1 luna de experimente Pb-Pb ~ **1 Pbyte**
 - ♦ 11 luni de experimente p-p ~ **1 Pbyte**
- Simulare:
 - ♦ 1 eveniment Pb-Pb ~24 ore
- Reconstrucție de date, filtrare, analiză, calibrare



Paralel vs. Distribuit

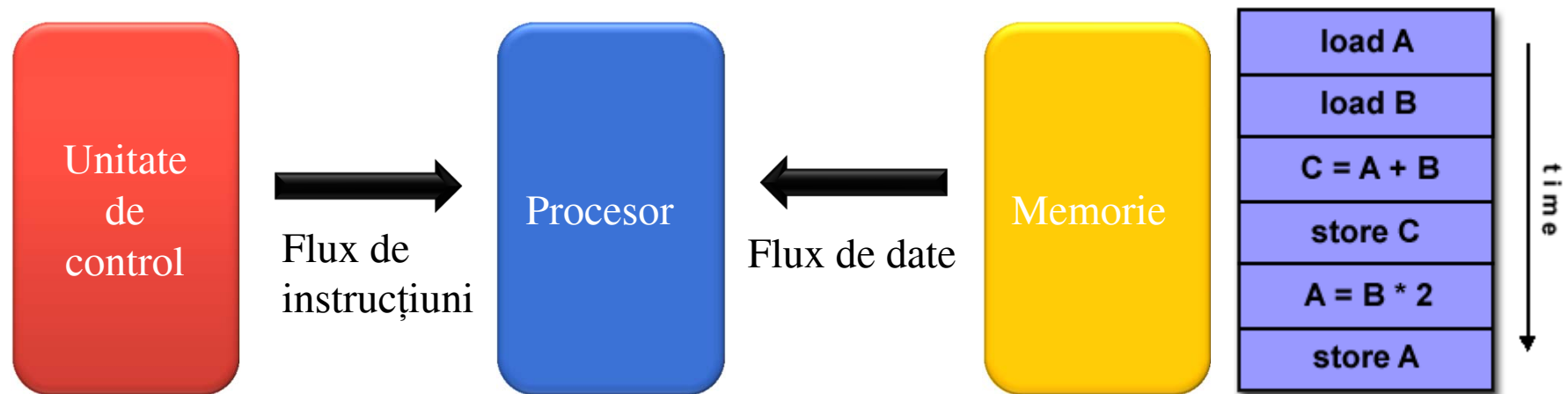
- **Calcul paralel**
 - ♦ împărțirea unei aplicații în task-uri executate simultan
- **Calcul distribuit**
 - ♦ împărțirea unei aplicații în task-uri executate în sisteme diferite (cu resurse diferite)
- **Convergență paralel ⇔ distribuit**
 - ♦ Folosesc din ce în ce mai mult aceleași arhitecturi
 - Sisteme distribuite folosite în calcul paralel
 - Calculatoare paralele folosite ca servere de mare performanță
 - ♦ Au zone de aplicații comune
 - ♦ Problemele de cercetare se întrepătrund și sunt abordate în comun
 - ♦ Se folosește termenul comun de “calcul de înalta performanță” (HPC – High Performance Computing)

Arhitecturi Paralele

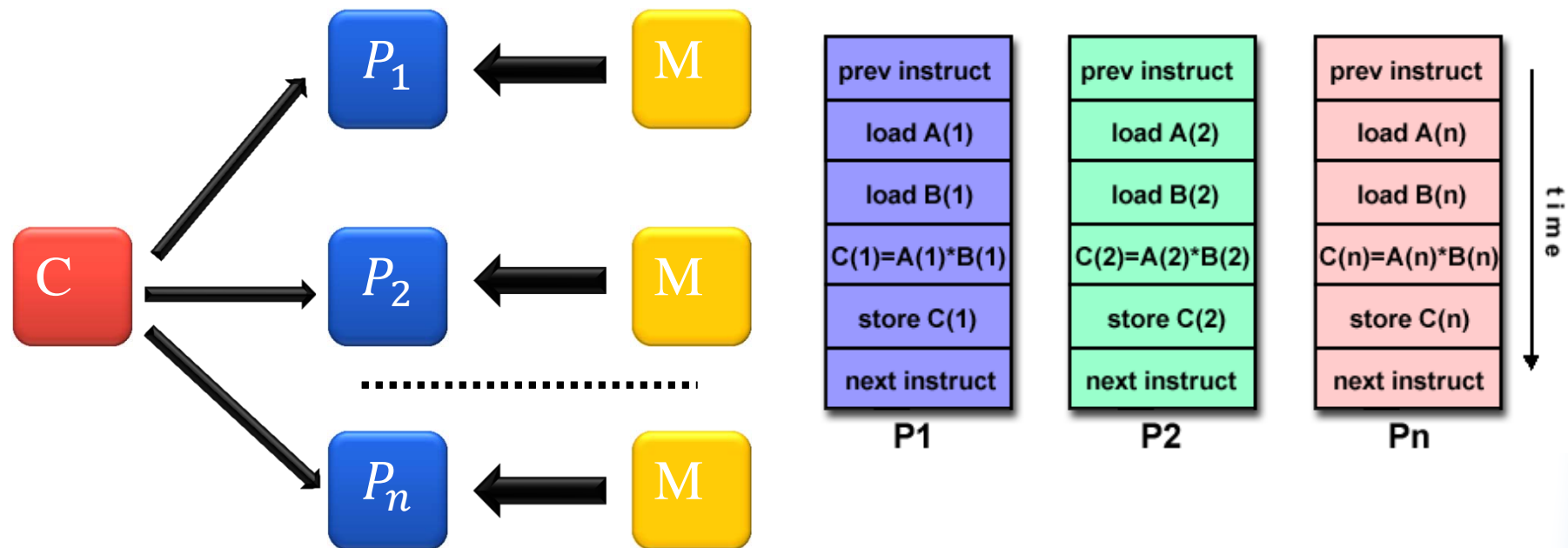
- Taxonomia Flynn (1986)
 - **SISD** – Single Instruction Stream, Single Data Stream
 - **SIMD** – Single Instruction Stream, Multiple Data Stream
 - **MISD** – Multiple Instruction Stream, Single Data Stream
 - **MIMD** – Multiple Instruction Stream, Multiple Data Stream
- Importanța pentru implementarea algoritmilor paraleli

SISD

Model clasic *von Neumann*



SIMD



- Implementat ca
 - ♦ Sisteme cu memorie partajată - *Shared Memory (PRAM)*
 - ♦ Multiprocesoare interconectate - *Interconnected Multiprocessors*

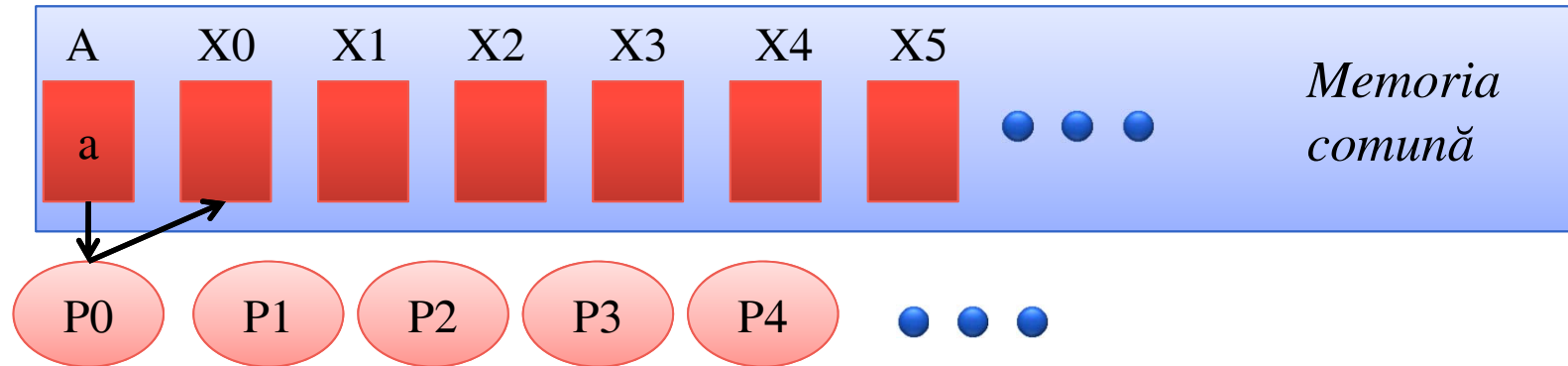
SIMD (2)

- **Shared memory** (Parallel Random Access Machine - PRAM)
 - ♦ **EREW** - Exclusive Read Exclusive Write
 - ♦ **CREW** - Concurrent Read Exclusive Write
 - ♦ **ERCW**
 - ♦ **CRCW**
- Influențează performanța
 - ♦ Exemplu: citirea valorii unei variabile partajate
 - ♦ Un pas in CREW, CRCW
 - ♦ $\log N$ pași in EREW, ERCW

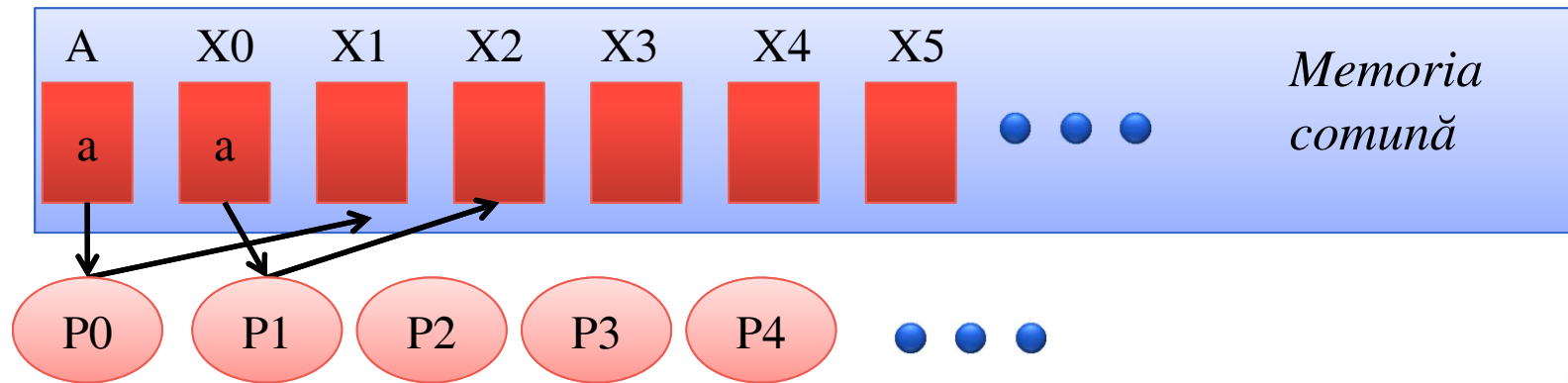
SIMD (3)

- Valoarea este în variabila A în memoria comună
- Folosește $X0, X1, \dots$ din memoria comună
 - ♦ Procesul $P0$ citește valoarea și o scrie în $X0$
 - ♦ Procesul $P1$ citește $X0$ și o scrie în $X1$
 - ♦ Procesele $P2, P3$ citesc $X0, X1$ și scriu în $X2, X3$
 - ♦ numărul de copii se dublează la fiecare pas
 - ♦ pentru N procesoare operația se termină după $\log N$ pași

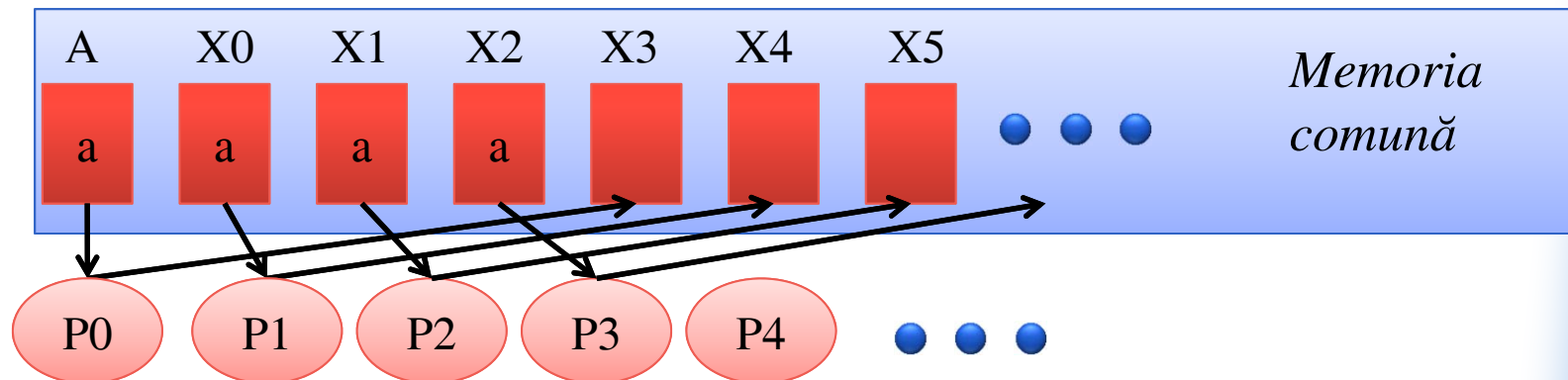
Pas 1



Pas 2



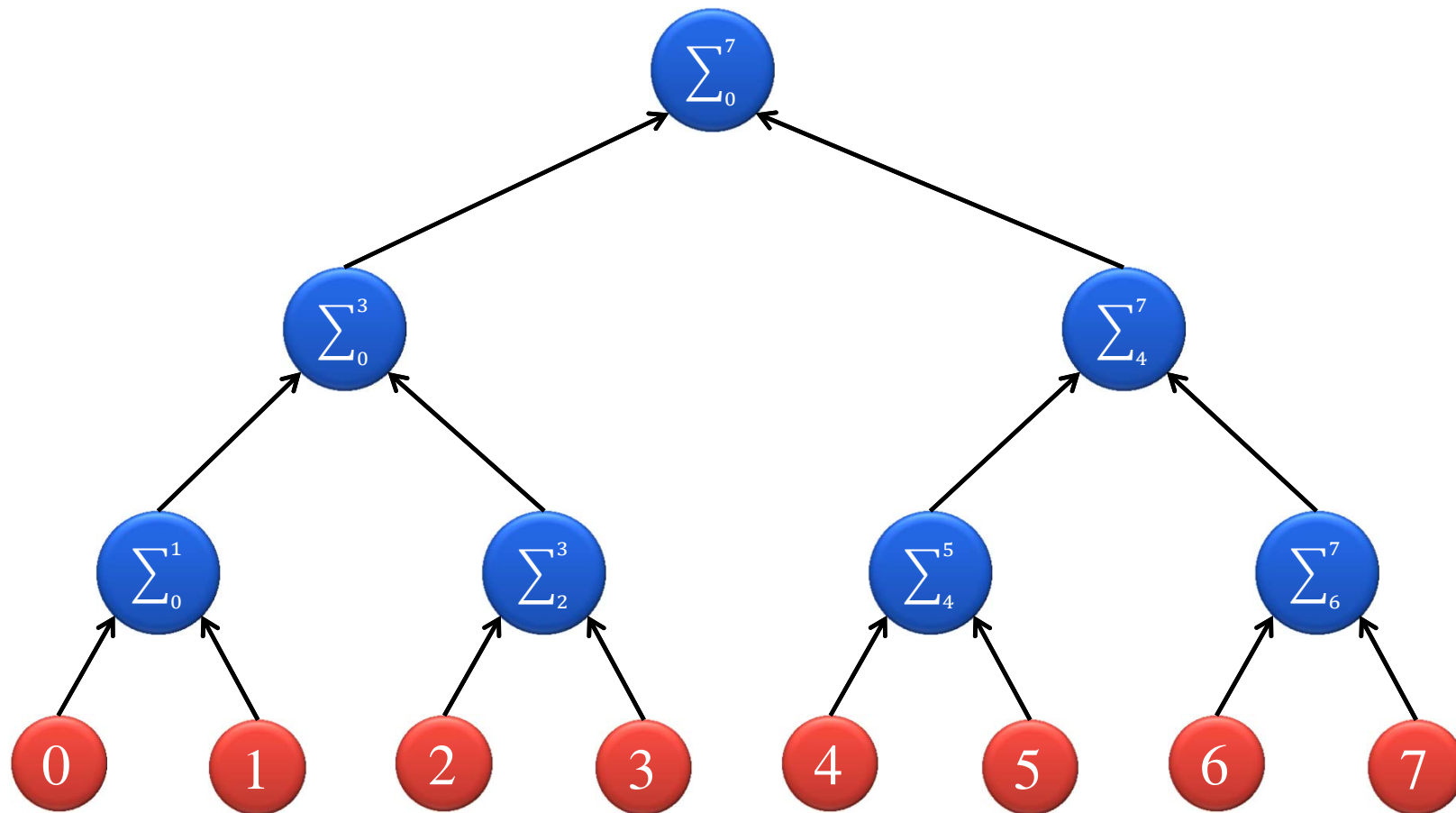
Pas 3



SIMD (4)

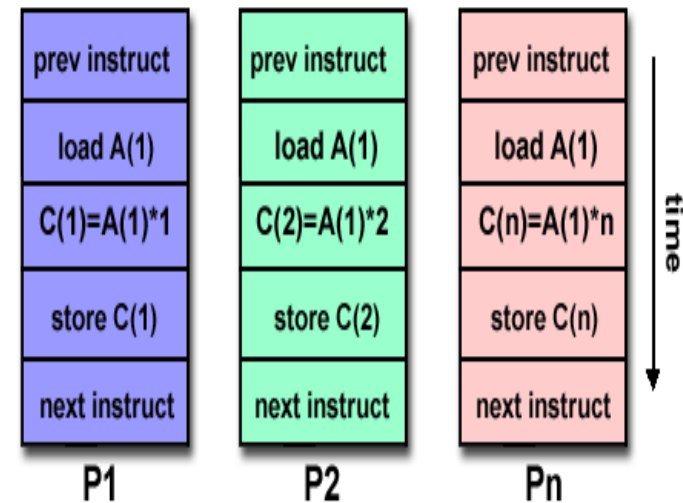
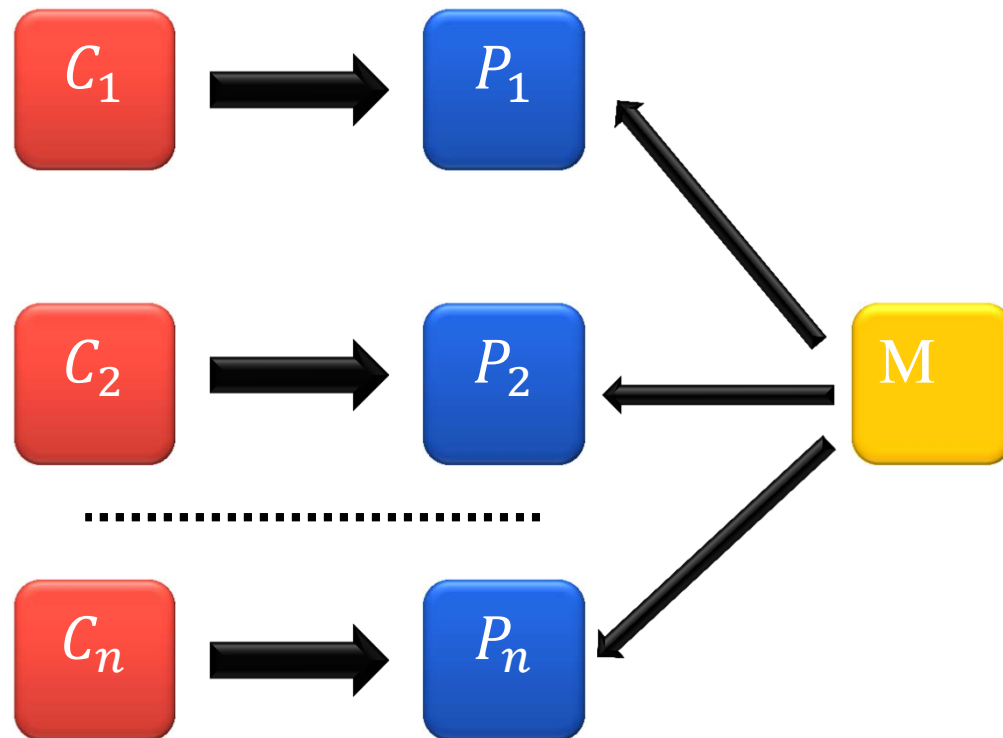
- **Rețele de interconectare**
- Topologii:
 - ♦ tablou
 - ♦ arbore
 - ♦ cub
 - ♦ hipercub
- Configurația depinde de:
 - ♦ aplicație
 - ♦ performanțele dorite
 - ♦ numărul procesoarelor disponibile
- Exemple: IBM 9000, Cray C90, Fujitsu VP

Arbore



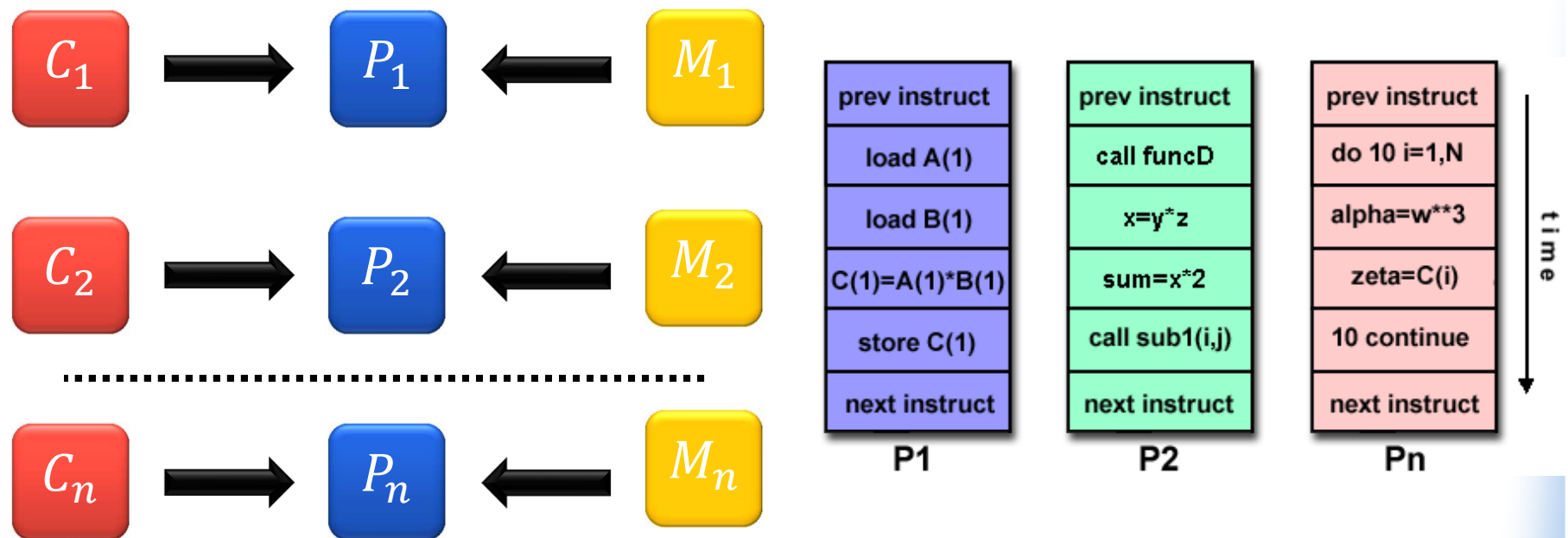
MISD

- Fără relevanță practică



MIMD

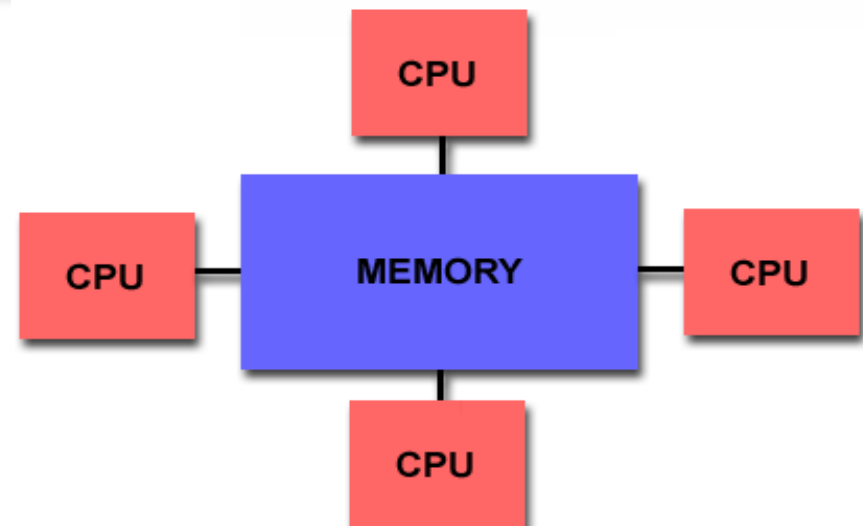
- Implementată ca Multi-calculatoare (memorie distribuită) sau Multi-procesoare (memorie partajată)



MIMD (2)

- **"Shared Memory"**

- ♦ Uniform Memory Access (UMA)
- ♦ Non-Uniform Memory Access (NUMA)
- ♦ Cache coherent Non-Uniform Memory Access (ccNUMA)



Avantaje:

- Spațiu de adrese global – ușurința în programare
- Partajare rapidă a datelor între procese datorită proximității memorie – CPU

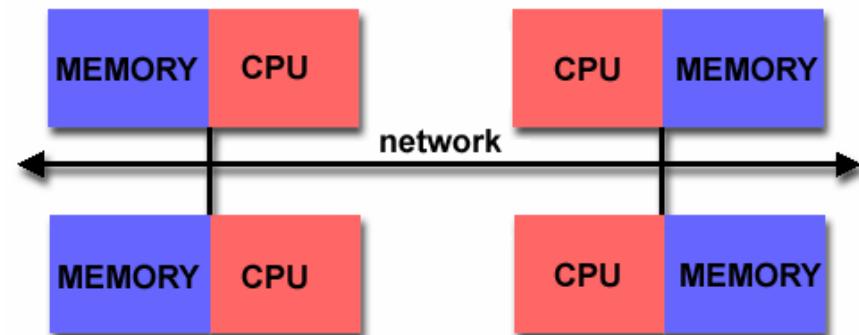
Dezavantaje:

- Lipsă scalabilității între memorie și CPU
- Sincronizarea în responsabilitatea programatorului
- Scump

MIMD (3)

- **"Multi-Computer"**

- ♦ Massively Parallel Processors (MPP)
- ♦ Network Of Workstations (NOW)



Avantaje:

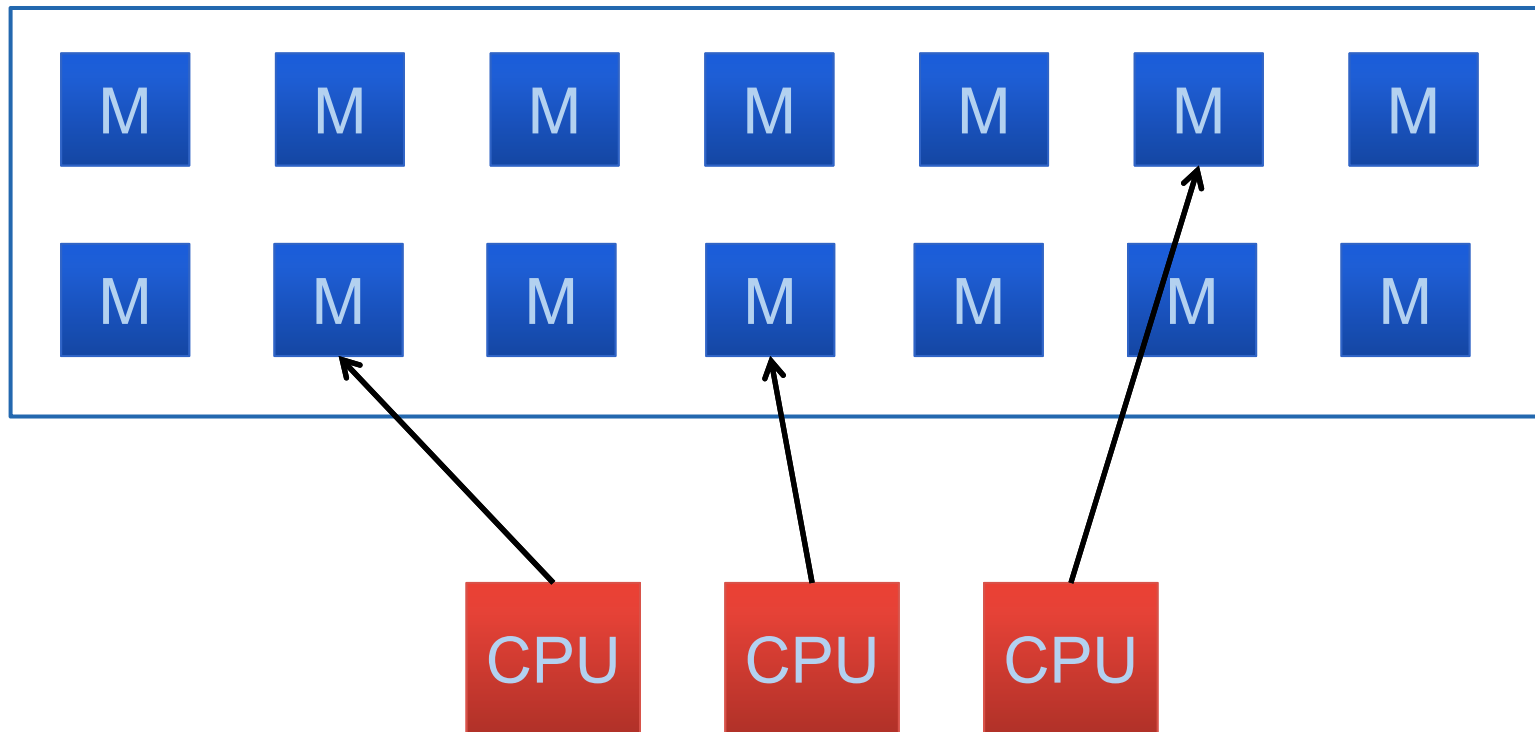
- Scalabilitate memorie – CPU
- Acces rapid la memorie
- Costuri reduse: procesoare + networking

Dezavantaje:

- Responsabilitatea programatorului pentru comunicația inter-procesoare
- Mapare dificilă a structurilor de date globale

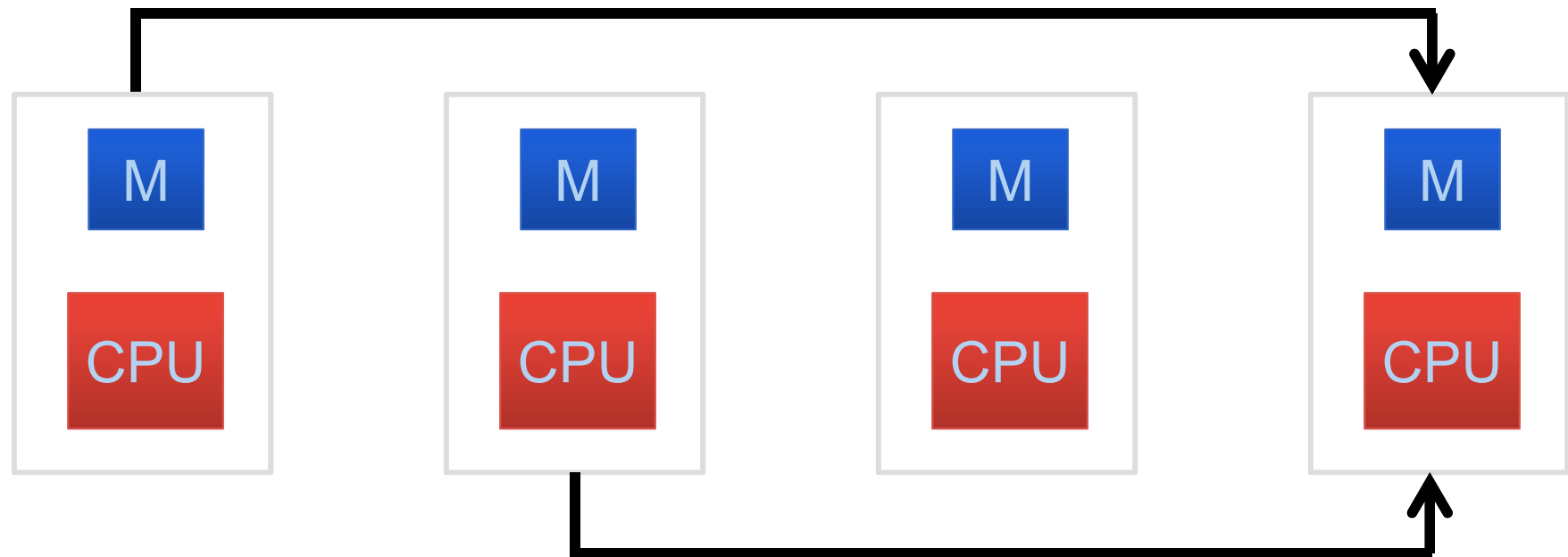
Metode de programare

- Date partajate (*Shared data*)



Metode de Programare (2)

- Transmitere de mesaje (*Message passing*)



Un model de programare

- Un program paralel / distribuit = colecție de procese paralele comunicante -
Communicating Sequential Processes
 - ◆ Bazat pe modelul CSP al lui Hoare
 - ◆ Folosit în multe limbaje și biblioteci paralele / distribuite
 - ◆ Adaptat pentru *message passing* și *shared data*

Tipuri de bază

boolean	bool
întreg	int
real	real
caracter	char
șir caractere	string

Declarația variabilelor

```
var id1: tip1:= val1, ... , idn: tipn:= valn;
```

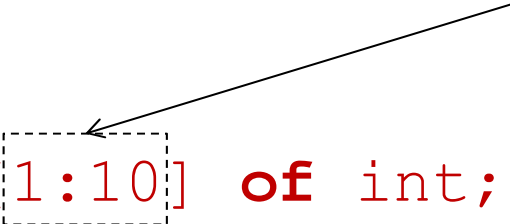
Definițiile de constante

```
const id1 = val1, ... , idn = valn;
```

Tipuri de bază (2)

Tablou

Elemente contigue ce pot fi procesate în paralel



```
var vector: array [1:10] of int;  
    matrice: array [1:n,1:m] of real;
```

Constructor

```
var forks: array [1:5] of bool := ([5>false]);
```

Tipuri de bază (3)

Înregistrare

```
type student = rec (nume : string[20];  
    vârsta: int;  
    clase: array [1:5] of string[15]);
```

```
var queue: rec (front: int :=1;  
    rear: int :=1;  
    size: int :=0;  
    contents: array [1:n] of int);
```

Instrucțiuni

Atribuirea

$x := e$

Interschimbarea

$x1 := x2$

Comanda cu gardă

$B \rightarrow S$

S se execută doar dacă **B** este evaluat la valoarea *true*

Instrucțiunea compusă [S] este o secvență de instrucțiuni.

Selecția (if)

if $B1 \rightarrow S1$

$[] B2 \rightarrow S2$

\dots

$[] Bn \rightarrow Sn$

fi

Instrucțiuni (2)

Iterația (do)

```
do B1 → S1  
[] B2 → S2  
...  
[] Bn → Sn  
od
```

Ciclul cu contor (fa)

```
fa cuantificatori → instrucțiuni af  
variabilă := val_init to val_finală st B
```

Exemple:

```
fa i:=1 to n, j:=i+1 to n → m[i,j]::m[j,i] af  
fa i:=1 to n, j:=i+1 to n st a[i]>a[j] → a[i]::a[j]  
af
```

Proceduri

```
procedure p(f1: t1; ...; fn: tn) returns r: tr;  
    declarații  
    instrucțiuni  
end;
```

Exemplu: calculul factorialului.

```
procedure fact(i : int) returns f: int;  
    if i < 0 → f := -1  
    [] i = 0 or i=1 → f:=1  
    [] i > 1 → f := i * fact(i-1)  
    fi  
end;
```

Execuție concurentă

```
co S1 || S2 || ... || Sn oc
```

Ex.1:

```
x:=0; y:=0;
```

```
co x:=x+1 || y:=y+1 oc
```

```
z:=x+y;
```

Ex. 2:

```
co j:=1 to n  $\rightarrow$  a[j]:=0 oc
```

Execuție concurentă

```
co S1 || S2 || ... || Sn oc
```

Ex.1:

```
x:=0; y:=0;
```

```
co x:=x+1 || y:=y+1 oc
```

```
z:=x+y;
```

Ex. 2:

```
co j:=1 to n  $\rightarrow$  a[j]:=0 oc
```


Execuție concurentă (2)

Ex. 3: produs de matrici

```
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
```

Acțiuni atomice

- Acțiuni **indivizibile** care examinează sau modifică starea sistemului / starea programului
- Orice **stare intermediară** din implementarea acțiunii atomice **nu trebuie să fie vizibilă** celorlalte procese
 - ♦ Ex: instrucțiuni mașină de *load* sau *store*
- **Read/write – acțiuni atomice**, fiecare proces are propriul set de regiștri, stările intermediare evaluării unei expresii complexe sunt stocate în regiștrii proprii procesului
- Execuția unui program concurent constă în **întrețeserea** secvențelor de acțiuni atomice executate de fiecare proces
- Istorie (**trace**): $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$

Acțiuni Atomice (2)

- Variabilele pot fi împărțite în:
 - ♦ *R*: Variabile care sunt citite, dar nu și modificate
 - ♦ *W*: Variabile care sunt modificate (pot fi și citite)
- Două părți ale unui program sunt independente dacă mulțimea *R* a fiecărei părți este disjunctă față de mulțimile *R* și *W* ale celeilalte părți
- *Referință critică*: o variabilă dintr-o expresie care este modificată de un alt proces

Acțiuni Atomice (3)

```
y := 0; z := 0;  
co x := y + z || y := 1 || z := 2 oc;
```

- La sfârșit, x poate avea oricare valoare dintre: 0,1,2,3.
- **Corecție:** în cursul evaluării unei expresii, variabilele nu trebuie modificate de alte procese
└─> evaluarea atomică
- Majoritatea declarațiilor din programele concurente nu îndeplinesc această condiție!

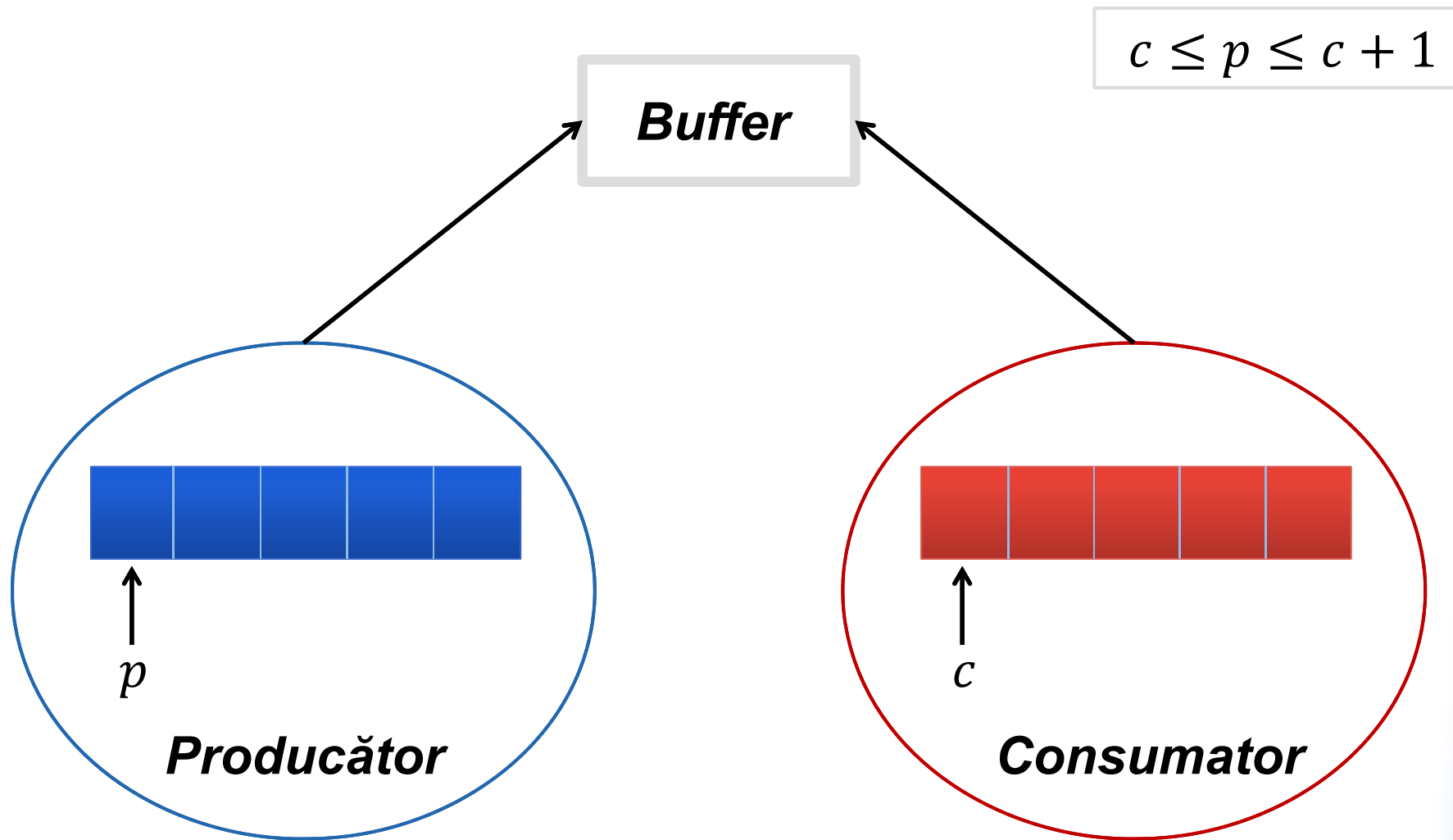
At Most Once

- $x := e$ satisface proprietatea dacă fie:
 - ♦ e conține cel mult o referință critică și x nu e citit de alte procese
 - ♦ e nu conține referințe critice (deci x poate fi citit de alte procese)
- Dacă nu e îndeplinită această condiție, folosim mecanisme de *sincronizare* pentru a asigura atomicitatea.

Sincronizare

- Soluția:
 - ♦ Acțiuni atomice: \langle , \rangle
 - ♦ Sincronizare folosind await: $\langle \text{await } B \rightarrow S \rangle$
- Doar sincronizare condiționată: $\langle \text{await } B \rangle$
 - ♦ spin loop / busy waiting !

Exemplu: Producător - Consumator



Exemplu: Producător-Consumator

```
var buf: int, p: int :=0, c:int :=0;
```

```
co Producer::
```

```
    var a: array[1:n] of int;
```

```
    do p < n → <await p = c>;
```

```
        buf := a[p + 1];
```

```
        p := p + 1
```

```
    od
```

```
oc
```

```
co Consumer::
```

```
    var b: array [1:n] of int;
```

```
    do c < n → <await p > c>;
```

```
        b[c + 1] := buf;
```

```
        c := c + 1
```

```
    od
```

```
oc
```


Sincronizarea cu barieră

- Paralelism de date, algoritmi iterativi:

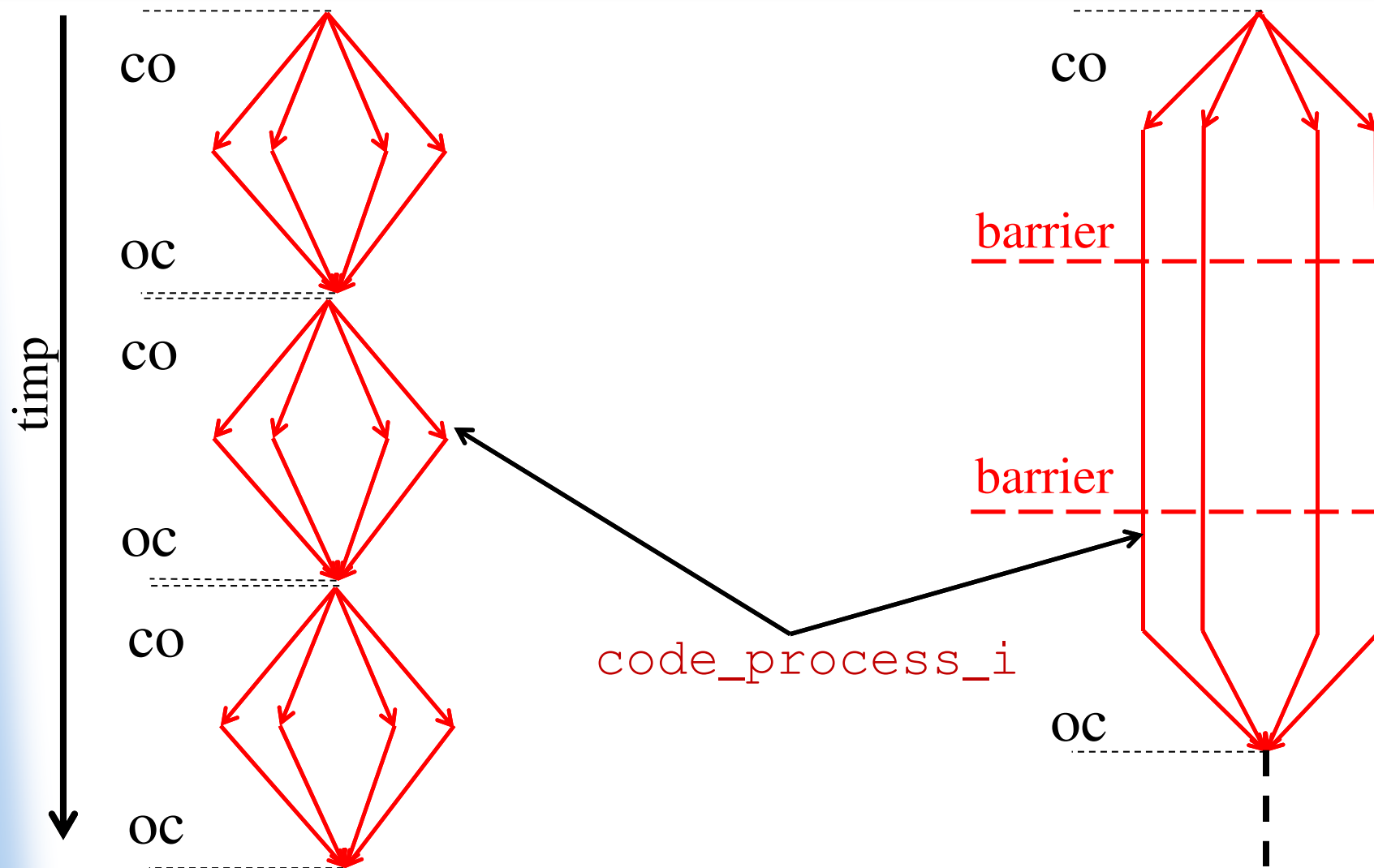
```
do true →  
  co i := 1 to n → code_process_i oc  
od
```

Soluție ineficientă!

```
co Process(i: 1..n)::  
  do true →  
    code_process_i  
    barrier  
  od  
oc
```

- Toate procesele se sincronizează la sfârșitul fiecarui ciclu
- Util când fiecare iterație depinde în întregime de rezultatele iterației precedente

Sincronizarea cu barieră



Exemplu: calcul grilă

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

        barrier
        test_convergentă;
        barrier
        grilă[i,j] := nouă[i,j];
        barrier
    od
oc
```

Sumar

- Problematica calculului paralel și distribuit
- Arhitecturi paralele
- Metode de programare
- Limbaje de descriere a algoritmilor paraleli
- Concurență și sincronizare
- Atomicitate
- Bariere

Întrebări?