

# Curs 1

Programare Paralela si Distribuita

# Continutul cursului

(realizat si pe baza pe <http://grid.cs.gsu.edu/~tcpp/curriculum/>)

## Teoretic

- Notiuni introductive: arhitecturi, concurenta, paralelism
- Etape in dezvoltarea programelor paralele
- Evaluarea performantei programelor paralele
- Modele de programare paralela
  - Diferenta intre cele bazate pe memorie partajata si memorie distribuita
- *Patterns*
  - Pt programare paralela
  - Pt programare distribuita

## Practic

- Java threads (low level API)
- C++ (>=C++11) threads
- High-level API: pachete Java-> java.util.concurrent packages.
- Java streams
- OpenMP (C++)
- CUDA (C++)
- MPI –Message Passing Interface
  - exemplificari C, C++

# Bibliografie

- Ian Foster. Designing and Building Parallel Programs, Addison-Wesley 1995.
- Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders, Addison A Pattern Language for Parallel Programming. Wesley Software Patterns Series, 2004.
- Michael McCool, Arch Robinson, James Reinders, Structured Parallel Programming: Patterns for Efficient Computation,” Morgan Kaufmann,, 2012 .
- D. Culler, J. Pal Singh, A. Gupta. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann. 1998.
- Grama, A. Gupta, G. Karypis, V. Kumar. Introduction to Parallel Computing, Addison Wesley, 2003.
- D. Grigoras. Calculul Paralel. De la sisteme la programarea aplicatiilor. Computer Libris Agora, 2000.
- V. Niculescu. Calcul Paralel. Proiectare si dezvoltare formală a programelor paralele. Presa Univ. Clujana, 2006.
- B. Wilkinson, M. Allen, Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice Hall, 2002
- A. Williams. C++ Concurrency in Action PRACTICAL MULTITHREADING. Manning Publisher.2012.
- Tutoriale Java: <http://docs.oracle.com/javase/tutorial/essential/concurrency/further.html>
- C++11 <http://en.cppreference.com/w/>
- OpenMP: <http://openmp.org/>
- MPI: <http://www.mpi-forum.org/>

# *Evaluare*

- Laborator
  - (30%) **NL**->Programe/proiecte
  - (5%) **NS**-> Exercitii in timpul laboratorului (“in class”)

## In sesiune

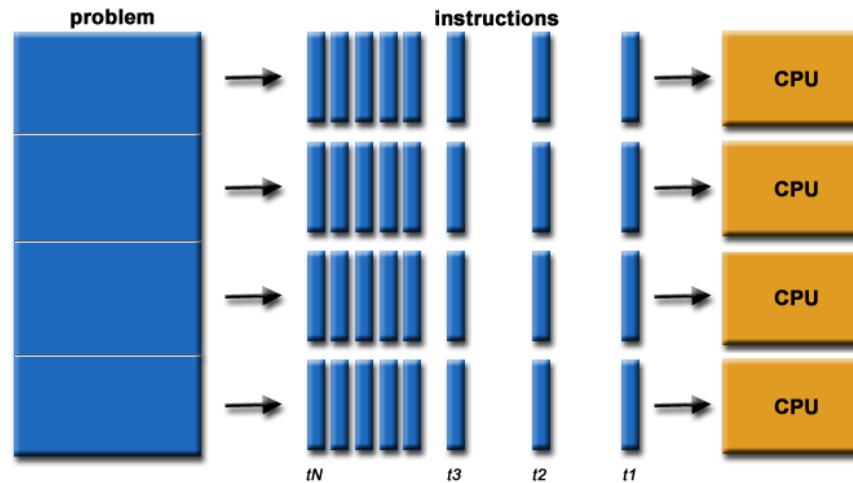
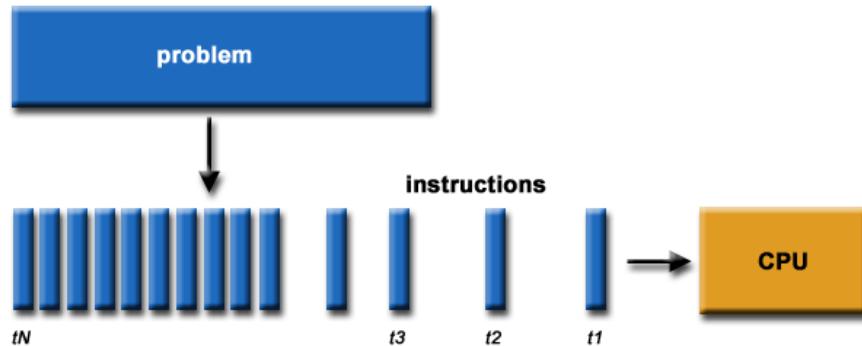
- (25%) **NT** -> Test practic
- (40%) **NE** -> Examen teoretic **NE>=4.5**

# Procesare Paralela

- Un *calculator paralel* este un calculator (sistem) care foloseste multiple elemente de procesare simultana intr-o maniera cooperativa pentru a rezolva o problema computationala.
- Procesarea Paralela include tehnici si tehnologii care fac posibil calculul in paralel
  - Hardware, retele, SO, biblioteci, limbaje, compilatoare, algoritmi ...
- PERFORMANCE
  - *Parallelism is very much about performance!*

# Calcul Serial vs. Paralel

(images from **Introduction to Parallel Computing Blaise Barney** )

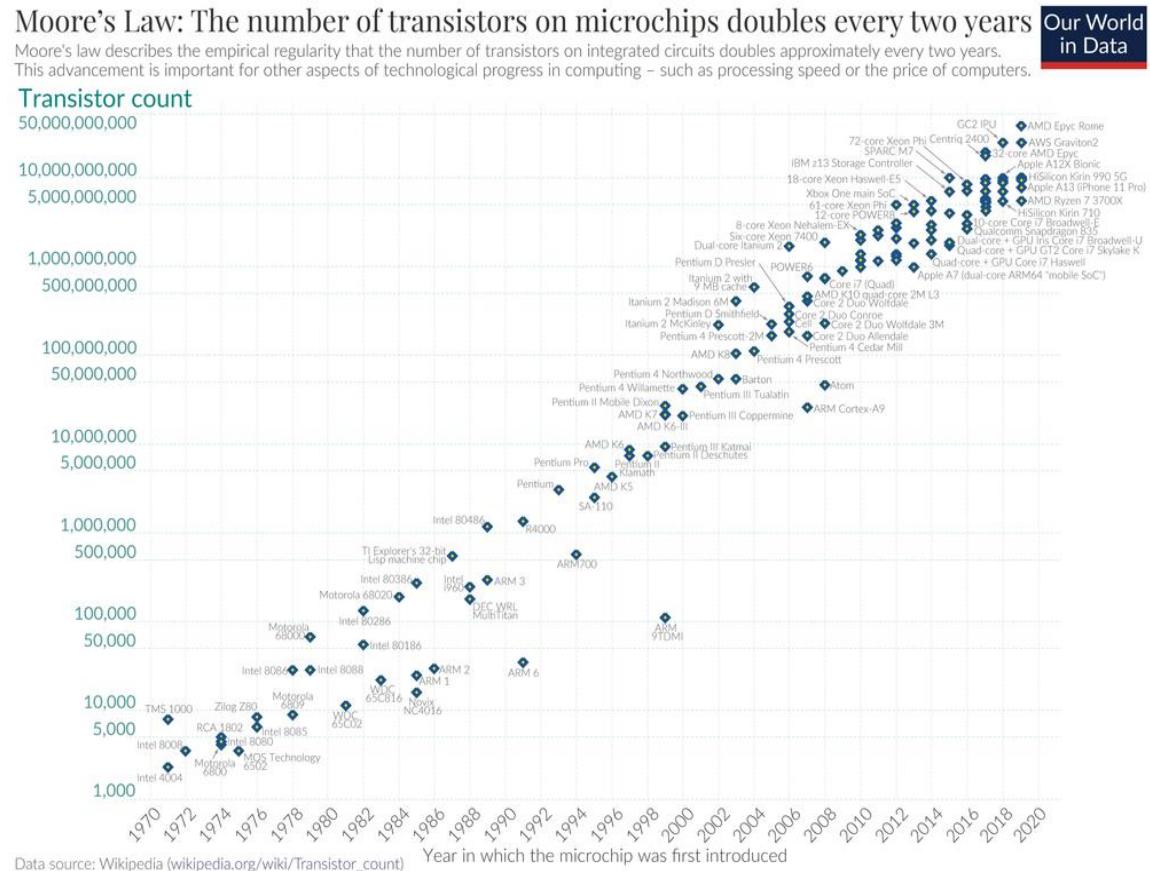


***“It would appear that we have reached the limits  
of what it is possible to achieve with computer technology,  
although one should be careful with such statements,  
as they tend to sound pretty silly in 5 years. “***

(John von Neumann, 1949)

# Limite ale programarii seriale

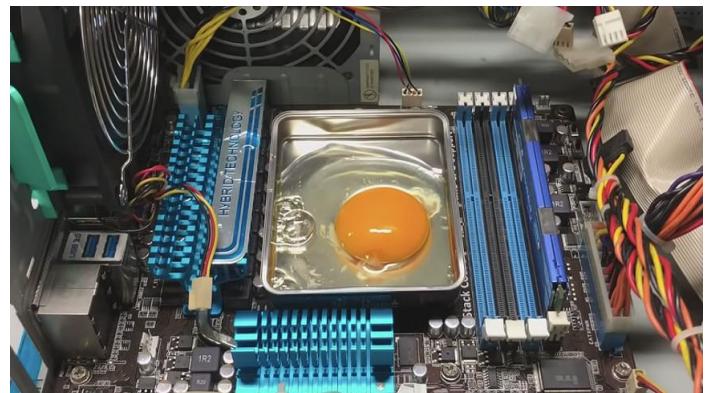
- Viteza de transmisie –
  - Viteza luminii (30 cm/nanosecond),
  - limita de transmisie pe fir de cupru (9 cm/nanosecond).
- Limitarea miniaturizarii – numar de tranzistori pe chip.
  - **Legea lui Moore:** numărul de tranzistori care pot fi plasati un circuit integrat (per square inch chip) se dubleaza la fiecare 2 ani.
  - Impune costuri mari.
- Limitari economice



# Istoric

- Cresterea performantei procesor prin cresterea frecventei ceasului CPU (CPU clock frequency)
  - Riding Moore's law
- Probleme : incalzirea puternica a chipurilor!
  - Frecventa ceas mai mare  $\Rightarrow$  consum electric mai mare

○ *Frying an egg on a computer*



- Solutie – adugare mai multor core-uri pt a ajunge la performanta dorita
  - Se pastreaza frecventa de ceas la fel sau chiar micsorare
  - nu creste consumul.

# Niveluri de paralelism

## 1. paralelism la nivel de job:

- intre joburi;
- intre faze ale joburilor;

## 2. paralelism la nivel de program:

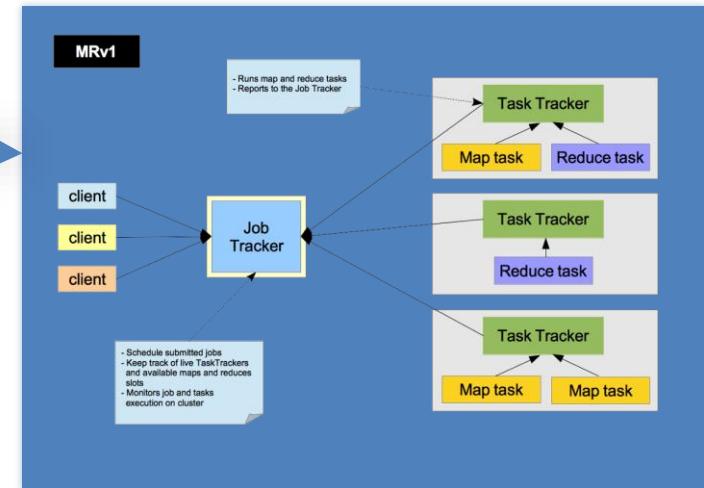
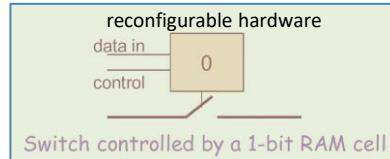
- intre părți ale programului;
- în anumite cicluri;

## 3. paralelism la nivel de instrucții:

- *Instruction-level parallelism (ILP)*

## 4. paralelism la nivel aritmetic și la nivel de bit:

- intre elemente ale unei operații vectoriale;
- intre circuitele logice aritmetice.

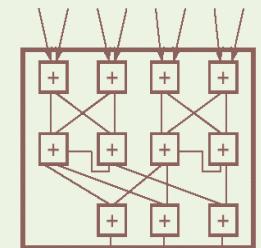


ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.

## Bit-Level Parallelism: Popcount

Returns the number of 1 bits in the value of x.

```
popcount(int v) {  
    int i, sum;  
  
    for (i=0; i<8; i++)  
        sum += (v>>i)&1;  
  
    return sum;  
}
```



# *Hardware parallelism*

- Arhitecturile curente se bazeaza tot mai mult pe paralelism la nivel hardware pentru a imbunatati performanta

Metrici performanta procesor=>

- ✓ *number of instruction issues per machine cycle (IPC)*
  - ✓ *number of instructions per second = IPC x clock rate (cycles per sec given in Hertz)*
  - ✓ *floating point operations per second*
- (DAR performanta depinde si de software si de configuratia intregii masini – !!! memorie ([memory hierarchy](#)))

Modalitati de imbunatatire:

- Multiple execution units
- Pipelined instructions
- Multi-core

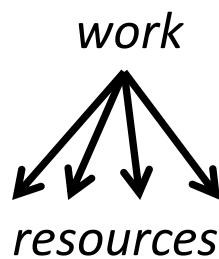
# Paralelism <-> Concurinta

- Consideram mai multe taskuri care trebuie executate pe un calculator
- Taskurile se considera a fi ***pur paralele*** daca:
  - Se pot executa in acelasi timp (*parallel execution*)
- Dependente -> executie concurrenta:
  - Un task are nevoie de rezultatele altora;
  - Un task trebuie sa se execute dupa ce o anumita conditie e indeplinita
  - Mai multe taskuri incearca sa foloseasca aceeasi resursa
    - => Forme de sincronizare trebuie folosite pentru a satisface conditiile/dependentele
- Concurinta este fundamentala in *computer science*
  - Sisteme de operare, baze de date, networking, ...

# Paralelism vs. Concurinta

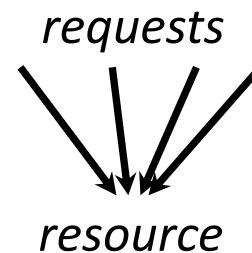
## Paralelism:

Se folosesc mai multe resurse pentru a rezolva o problema mai rapid



## Concurinta:

Gestiunea corecta si eficienta a accesului la resurse comune



Obs:

- Se pot folosi *threaduri* sau procese in ambele cazuri
  - Daca un calcul paralel necesita acces la resurse comune atunci este nevoie sa se gestioneze corect concurinta
- ⇒ Paralelismul poate implica concurinta  
⇒ Concurinta nu exista fara paralelizare

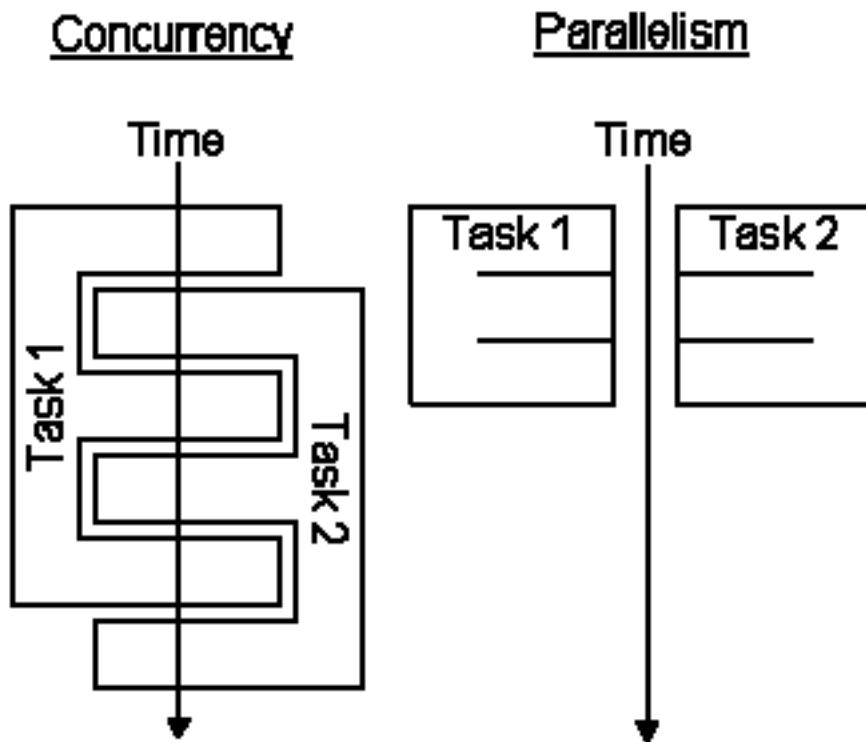
# Concurrenta si Paralelism

- Concurrent vs. paralel
- Executie Paralela:
  - Taskurile se executa efectiv in acelasi timp;
  - Este necesara existenta de multiple resurse de calcul
- **Paralelism = concurrenta + hardware “paralel”**

# Paralelism

- Există mai multe niveluri de *software parallelism* :
  - Procese, threads, routine, instrucțiuni, ...
- Trebuie să fie suportate de resursele hardware
  - Procesoare, nuclee(cores), ... (execuția instrucțiunilor)
  - Memorii, DMA, retele , ... (operări asociate)

# o abordare simplista - (*debatable*)



<http://www.java-programming.info/tutorial/pdf/java/11-Java-Multithreaded-Programming.pdf>

# De ce sa folosim programare paralela?

- Motive primare:
  - Timp de calcul mai rapid (*response time*)
  - Rezolvarea problemelor ‘mari’ de calcul (in timp rezonabil de calcul)
- Motive secundare:
  - Folosirea efectiva a resurselor de calcul
  - Costuri reduse
  - Reducerea constrangerilor asociate memoriei
  - Limitarile masinilor seriale
- **Paralelism = concurenta + hardware ‘paralel’+ performanta**

- **Rezolvarea problemelor dificile, mari:**
  - "Grand Challenge" ([en.wikipedia.org/wiki/Grand\\_Challenge](http://en.wikipedia.org/wiki/Grand_Challenge)) problems requiring PetaFLOPS and PetaBytes of computing resources.
  - Web search engines/databases processing millions of transactions per second
- **Folosirea resurselor non-locale:**
  - SETI@home ([setiathome.berkeley.edu](http://setiathome.berkeley.edu)) uses over 330,000 computers for a compute power over 528 TeraFLOPS (as of August 04, 2008)
  - Folding@home ([folding.stanford.edu](http://folding.stanford.edu)) uses over 340,000 computers for a compute power of 4.2 PetaFLOPS (as of November 4, 2008)

# Grand Challenge Problems

- The solution or simulation of fundamental problems in science and engineering, with a strong scientific and economic impact, known as Grand Challenge Problems (GCPs), have been the driving force for Parallel Computing.
- Typically, GCPs simulate phenomena that cannot be measured by experimentation:
  - ✓ I Global Climate modeling
  - ✓ I Biology: genomics; protein folding, drug design
  - ✓ I Astrophysical modeling
  - ✓ I Computational Chemistry
  - ✓ I Earthquake and structural modeling
  - ✓ I Computational fluid dynamics (airplane design)
  - ✓ I Crash simulation
  - ✓ I Financial and economic modeling

# New Data-Intensive Applications

- Currently, large volumes of data are produced and their processing and analysis also require high performance computing.
- Some examples:
  - ✓ I Data mining
  - ✓ I Web search
  - ✓ I networked video
  - ✓ I Video games and virtual reality
  - ✓ I Computer aided medical diagnosis
  - ✓ I ...
- Similarly, data is collected and stored at enormous speeds(GByte/hour).
- Some examples:
  - ✓ I sensor data streams
  - ✓ I telescope scanning the skies
  - ✓ I micro-arrays generating gene expression data

# Directii in procesarea paralela

- Arhitecturi paralele
  - Necesitati Hardware
  - Computer system design
- Sisteme de operare (Paralelism/concurrenta)
- Gestionarea aspectelor sistem pentru un calculator paralel
- Programare paralela
  - Biblioteci (low-level, high-level)
  - Limbaje
  - Medii de dezvoltare
  - Software
- Algoritmi Paraleli
- Evaluarea performantei programelor paralele
- Testarea vs. asigurarea corectitudinii
- *Parallel tools:*
  - Performanta, analize, vizualizare, ...

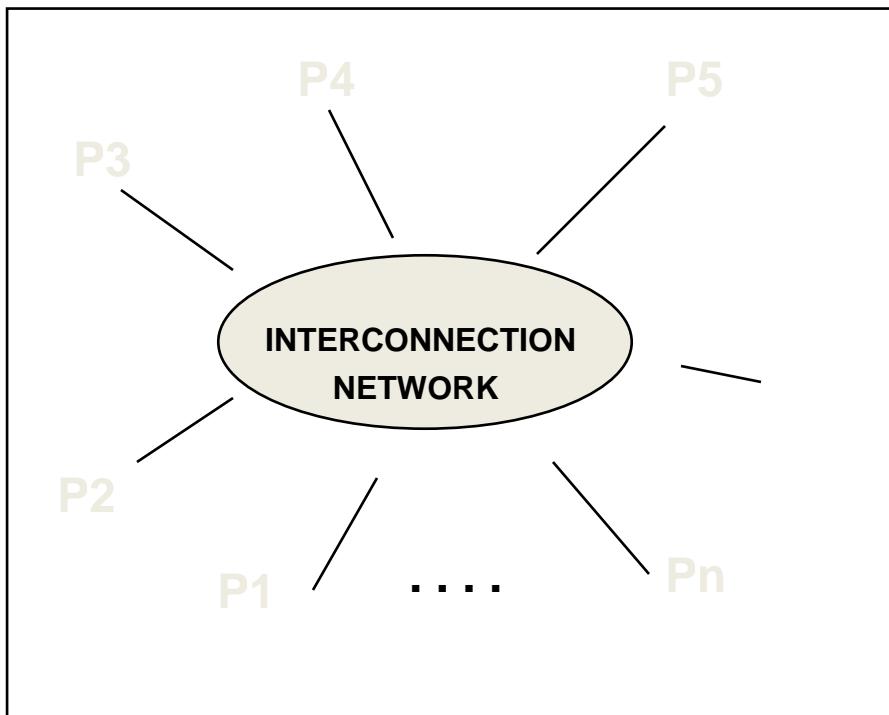
# De ce sa studiem programare paralela?

- Arhitecturi de calcul
  - Inovatiile conduc la noi modele de programare
- Convergenta tehnologica
  - “killer micro” este peste tot
  - Laptop-urile si supercomutere sunt fundamental similare
  - Trend-urile actuale conduc la convergenta abordarilor diverse
- Trendurile tehnologice fac calculul paralel inevitabil
  - Multi-core processors!
  - Acum orice sistem de calcul este paralel
- **Intelegerea principiilor fundamentale !!!**
  - Programare, comunicatii, memorie, ...
  - Performanta
- **“Parallelism is the future of computing” - Blaise Barney**
  - M. Andrews, J. S. Walicki. “Concurrency and parallelism—future of computing” in Proceeding of ACM '85 Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective. pp.224-231.

# Inevitabilitatea Procesarii Paralele

- Cerintele pt aplicatii
  - Necesitatea uriasa de cicluri de calcul
- Trenduri tehnologice
  - Procesare si memorie
- Trenduri Architecturale
- Factori economici
- Treduri actuale:
  - *Today's microprocessors have multiprocessor support*
  - *Servers and workstations available as multiprocessors*
  - *Tomorrow's microprocessors are multiprocessors*
  - *Multi-core is here to stay and #cores/processor is growing*
  - *Accelerators (GPUs, gaming systems)*

# Programare paralela vs. Programare distribuita



# TIPURI DE MULTIPROCESARE **PARALLEL**      **DISTRIBUTED**

## ASPECTE TEHNICE

### • **PARALLEL COMPUTERS** (- IN MOD UZUAL ) LUCREAZA BAZAT PE

- **CUPLARE STRANSA,**
- in general bazate pe **SINCRONICITATE,**
- **CU UN SISTEM DE COMUNICATIE FOARTE RAPID SI FIABIL**
- Spatiu unic de adresare (intr-o masura mare)

### • **DISTRIBUTED COMPUTERS**

- **MAI INDEPENDENTE,**
- **COMUNICATIE MAI PUTIN FREVENTA SI mai putin RAPIDA (ASINCRONA)**
- **COOPERARE LIMITATA**
- **NU EXISTA CEAS GLOBAL**
- “Independent failures”

## SCOPURI

### • **PARALLEL COMPUTERS** COOPEREAZA PENTRU A REZOLVA MAI EFICIENT PROBLEME DIFICILE

### • **DISTRIBUTED COMPUTERS** AU SCOPURI INDIVIDUALE SI ACTIVITATI PRIVATE. DOAR UNEORI INTERCOMUNICAREA ESTE NECESARA

**PARALLEL COMPUTERS:** COOPERARE IN SENS “POZITIV”

**DISTRIBUTED COMPUTERS:** COOPERARE IN SENS “NEGATIV” --  
DOAR ATUNCI CAND ESTE NECESARA

In general ...

## Aplicatii paralele

Suntem interesati sa rezolvam problemele *mai rapid* in paralel

## Aplicatii distribuite

Suntem interesati sa rezolvam anumite probleme specifice :

- COMMUNICATION SERVICES  
ROUTING  
BROADCASTING
- MAINTENANCE OF CONTROL STRUCTURE  
TOPOLOGY UPDATE  
LEADER ELECTION
- RESOURCE CONTROL ACTIVITIES  
LOAD BALANCING  
MANAGING GLOBAL DIRECTORIES

# Sistemele Distribuite

-pot fi folosite pentru -

- Aplicatii distribuite implicit
  - BD Distribuite, rezervari bilete avion/etc. sistem bancar
- Informatii partajate intre useri
- Partajare resurse
- Raport cost / performanta mai bun pt aplicatii paralele
  - Pot fi folosite eficient pt. aplicatii cu granularitate mare(*coarse-grained*) si/sau pt aplicatii paralele de tip *embarrassingly parallel applications*
- Fiabilitate (*Reliability*)
- Scalabilitate
  - Cuplare slaba (Loosely coupled connection) ; hot plug-in
- Flexibilitate
  - Reconfigurare sistem pentru a intruni cerintele

# Performanta/Scalabilitate

Spre deosebire de sistemele paralele cele distribuite implica:

- mediu mai putin rapid de transfer al datelor (retea mai putin rapida)
- Heterogenitate

Solutii:

- Procesare *batch* a mesajelor:
  - Se evita interventia SO pt fiecare transfer de mesaj.
- Cache data
  - Se evita repetarea transferului aceleiasi date
- Evitarea entitatilor si a algoritmilor centralizati
  - Evitarea saturarii retelei
- Realizare operatii “post” la nivelul clientului
  - Evitarea traficului intens intre clienti si servere
- ....

# Securitate

- Nu există doar un singur punct de control
- Probleme:
  - Mesaje, furate, modificate, copiate, ...
    - Solutie : folosire Criptografie
  - Failures
    - Fault Tolerance solutions

Un punct de vedere...

# Parallel v.s. Distributed Systems

(from M. FUKUDA CSS434 System Models)

	<b>Parallel Systems</b>	<b>Distributed Systems</b>
<b>Memory</b>	<b>Tightly coupled shared</b> memory UMA, NUMA	<b>Distributed</b> memory Message passing, RPC, and/or used of distributed shared memory
<b>Control</b>	<b>Global clock</b> control SIMD, MIMD	<b>No global clock</b> control Synchronization algorithms needed
<b>Processor interconnection</b>	Order of <b>Tbps</b> Bus, mesh, tree, mesh of tree, and hypercube (-related) network	Order of <b>Gbps</b> Ethernet(bus), token ring and SCI (ring), myrinet(switching network)
<b>Main focus</b>	<b>Performance</b> Ex. - Scientific computing	Performance( <b>cost and scalability</b> ) <b>Reliability/availability</b> <b>Information/resource sharing</b>

# Curs 2

## Programare Paralela si Distribuita

- Arhitecturi paralele
- Clasificarea sistemelor paralele
- *Cache Consistency*
- Top 500 Benchmarking

# Clasificarea sistemelor paralele -criterii-

## *Resurse*

- numărul de procesoare și puterea procesorului individual;
- Tipul procesoarelor – omogene- heterogene
- Dimensiunea memoriei

## *Accesul la date, comunicatie si sincronizare*

- complexitatea rețelei de conectare și flexibilitatea sistemului
- distribuția controlului sistemului,
  - dacă multimea de procesoare este condusa de catre un procesor sau
  - dacă fiecare procesor are propriul său controller;
- Modalitatea de comunicare (de transmitere a datelor);
- Primitive de cooperare (abstractizari)

## *Performanta si scalabilitate*

- Ce performanta se poate obtine?
- Ce scalabilitate permite?

# Clasificarea Flynn

[Michael J. Flynn în 1966](#)

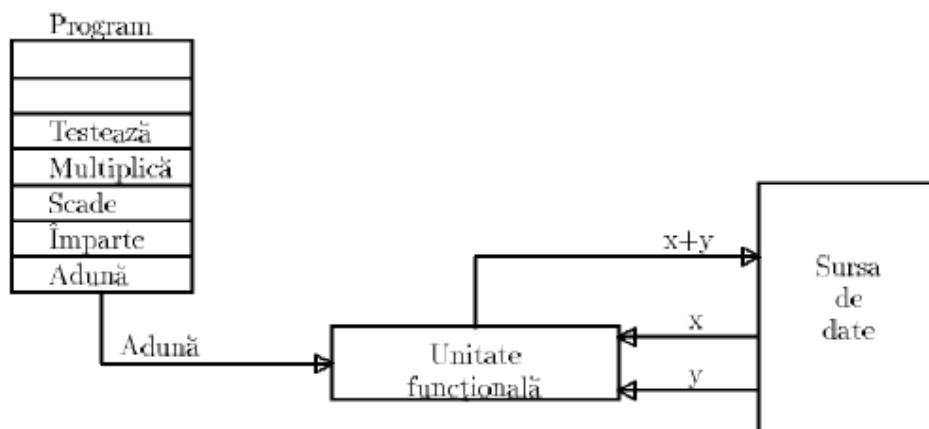
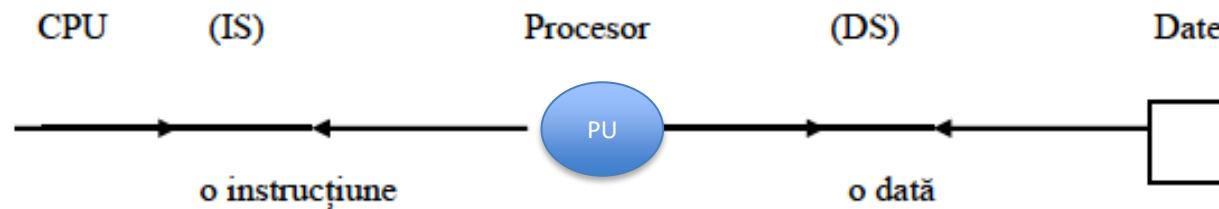
- SISD: sistem cu un singur flux de instrucțiuni și un singur flux de date;
- SIMD: sistem cu un singur flux de instrucțiuni și mai multe fluxuri de date;
- MISD: sistem cu mai multe fluxuri de instrucțiuni și un singur flux de date;
- MIMD: cu mai multe fluxuri de instrucțiuni și mai multe fluxuri de date.

(imagini urm. preluate din ELENA NECHITA, CERASELA CRIȘAN, MIHAI TALMACIU, ALGORITMI PARALELI SI DISTRIBUȚI)

# SISD(Single instruction stream, single data stream)

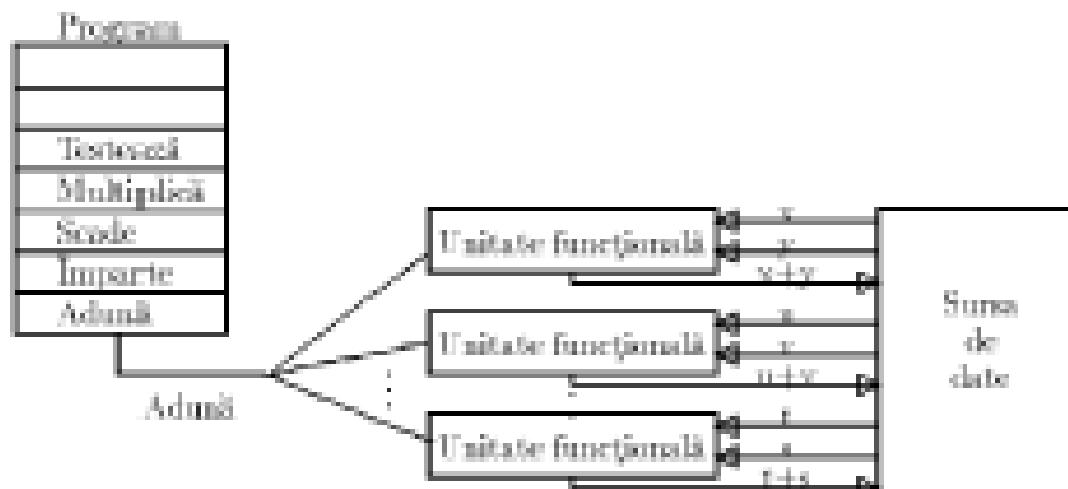
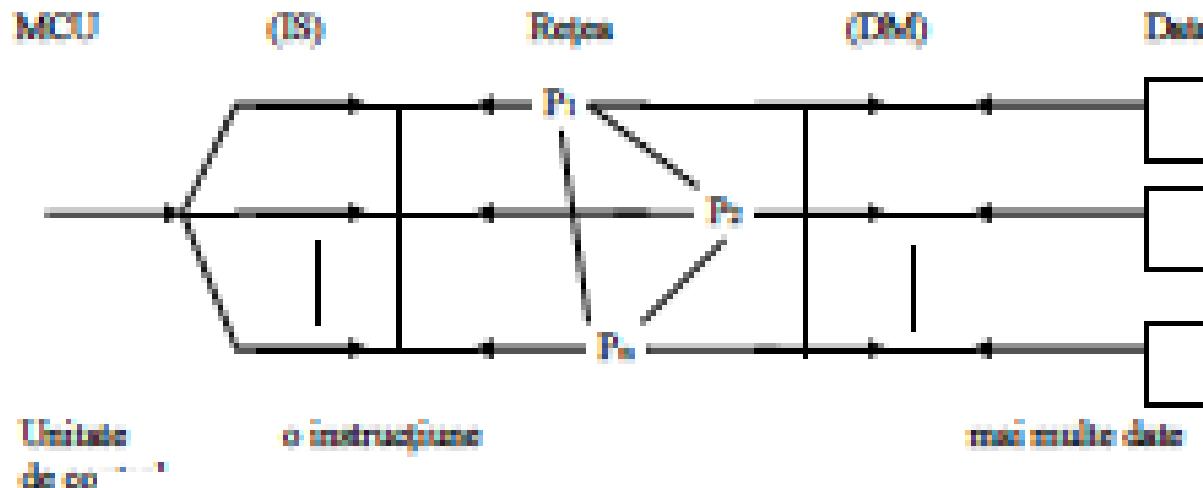
Flux de instrucțiuni singular, flux de date singular (SISD)-

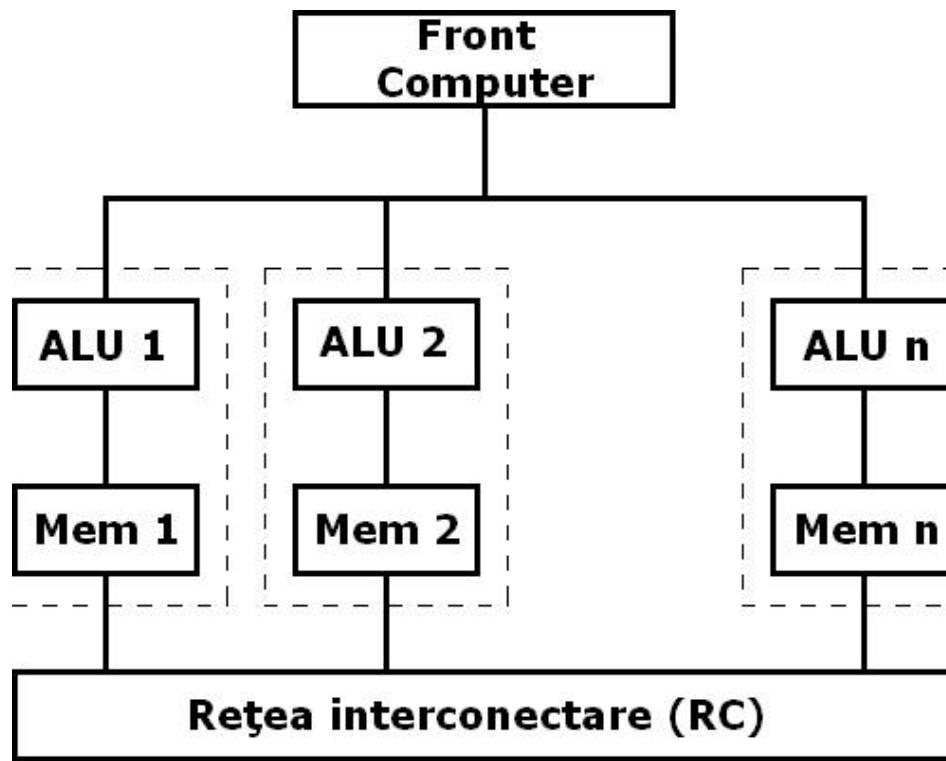
- microprocesoarele clasice cu arhitecturi von Neumann
- Funcționare ciclică: preluare instr., stocare rez. în mem. , etc.



# SIMD (Single instruction stream, multiple data stream)

Flux de instrucțiuni singular, flux de date multiplu





# Executie conditională în SIMD Processors

```
if (B == 0)
    C = A;
else
    C = A/B;
```

(a)

A	5
B	0
C	0

Processor 0

A	4
B	2
C	0

Processor 1

A	1
B	1
C	0

Processor 2

A	0
B	0
C	0

Processor 3

Initial values

A	5
B	0
C	5

Processor 0

Idle	
A	4
B	2
C	0

Processor 1

Idle	
A	1
B	1
C	0

Processor 2

A	0
B	0
C	0

Processor 3

Step 1

Idle	
A	5
B	0
C	5

Processor 0

A	4
B	2
C	2

Processor 1

A	1
B	1
C	1

Processor 2

Idle	
A	0
B	0

Processor 3

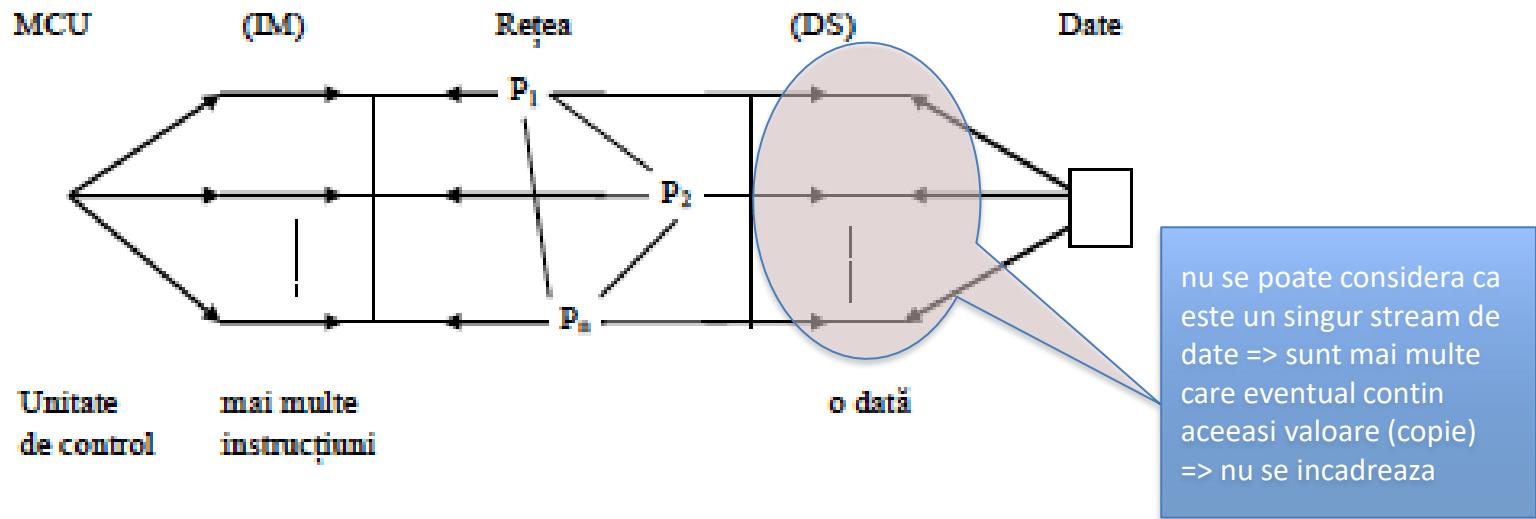
Step 2

(b)

# MISD (multiple instruction stream, single data stream)

Flux de instrucțiuni multiplu, flux de date singular

- multime vida !!!



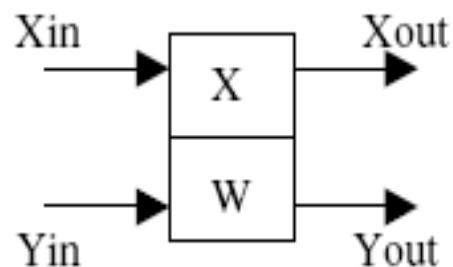
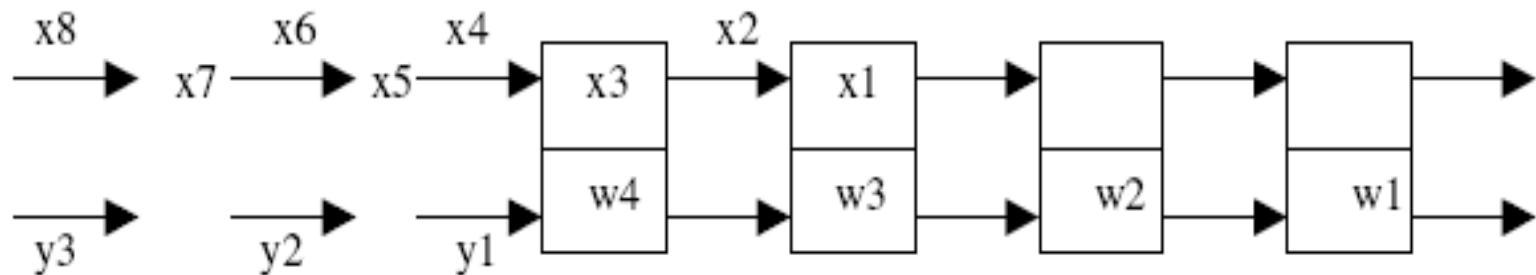
? ~ procesoare pipeline:

intr-un **procesor pipeline** exista un singur flux (stream) de date dar aceasta trece prin transformari succesive (mai multe instructiuni) iar paralelismul este realizat prin execuția simultană a diferitelor etape de calcul asupra unor date diferite (secvența de date care intra succesiv pe streamul de date)

## Exemplu – retea liniara (pipeline)

*Exemplu:* se consideră un sistem simplu pentru calcularea conoluțiilor liniare, utilizând o rețea liniară de elemente de prelucrare:

$$y(i) = w1*x(i)+w2*x(i+1)+w3*x(i+2)+w4*x(i+3)$$



$$\begin{aligned} X_{out} &= X \\ X &= X_{in} \\ Y_{out} &= Y_{in} + W \cdot X_{in} \end{aligned}$$

*Rețea liniară pentru calcularea conoluțiilor liniar.*

# Arhitectura sistolica

Orchestrate data flow for high throughput with less memory access

**Different from pipelining**

Nonlinear array structure, multidirection data flow, each PE may have (small) local instruction and data memory

**Different from SIMD**

Each PE may do something different

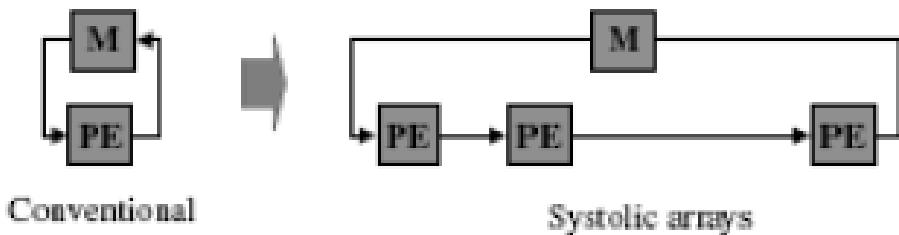
**Initial motivation**

VLSI enables inexpensive special-purpose chips

Represent algorithms directly by chips connected in regular pattern

## Systolic Architectures

Very-large-scale  
integration



Replace a processing element(PE) with an array of PE's  
without increasing I/O bandwidth

# Exemplu: matrix-vector multiplication

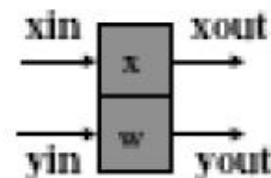
$$y_i = \sum_{j=1}^n a_{ij}x_j, i = 1, \dots, n$$



Recursive algorithm

```
for i = 1 to n  
    y(i,0) = 0  
    for j = 0 to n  
        y(i,0) = y(i,0) + a(i,j) * x(j,0)
```

Use the following PE

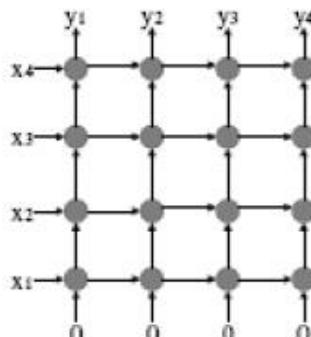


$$xout = x$$

$$x = xin$$

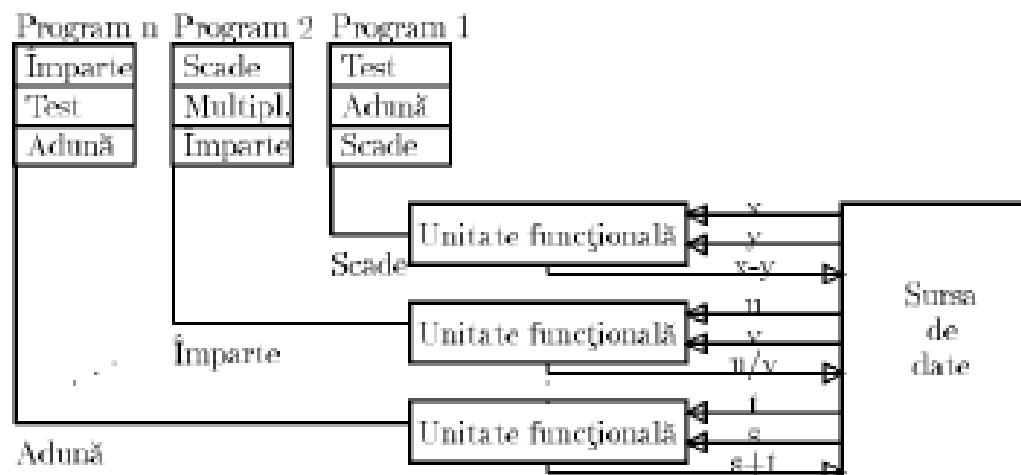
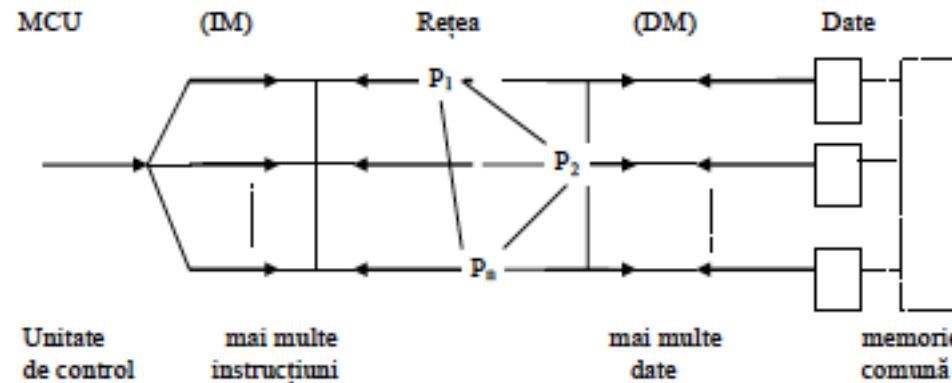
$$yout = yin + w * xin$$

## Systolic Array Representation of Matrix Multiplication



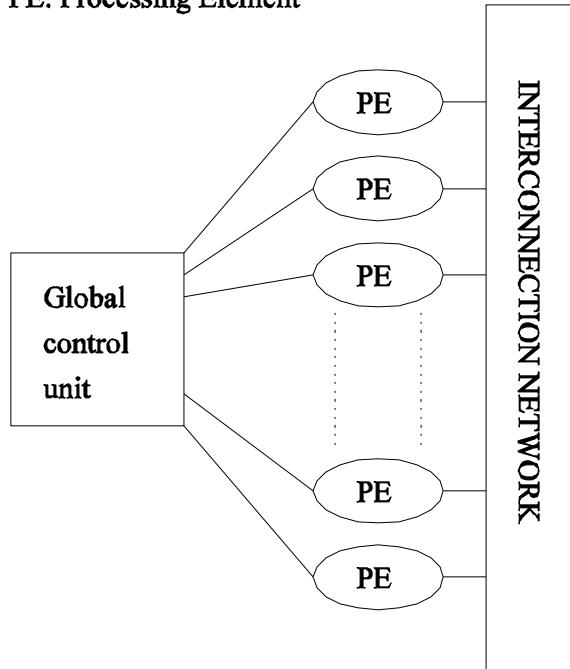
# MIMD (multiple instruction stream, multiple data stream)

Flux de instrucțiuni multiplu, flux de date multiplu

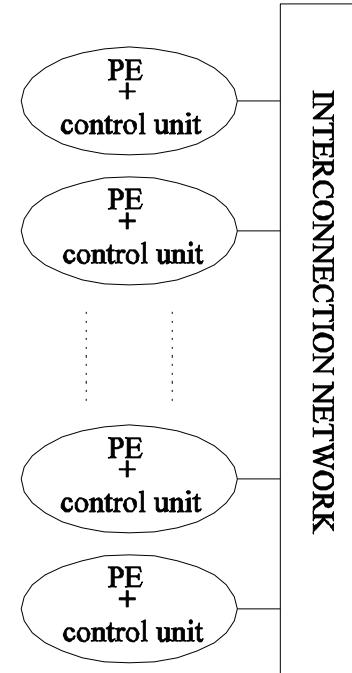


# SIMD versus MIMD

PE: Processing Element

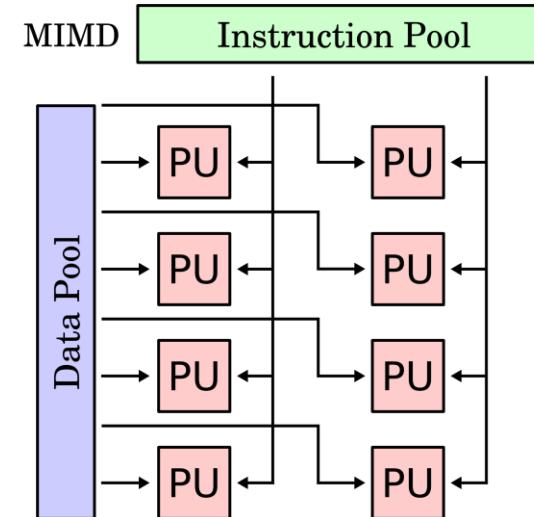
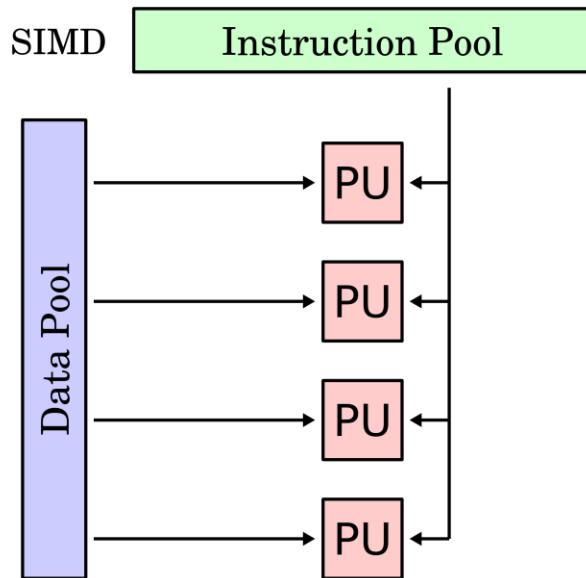
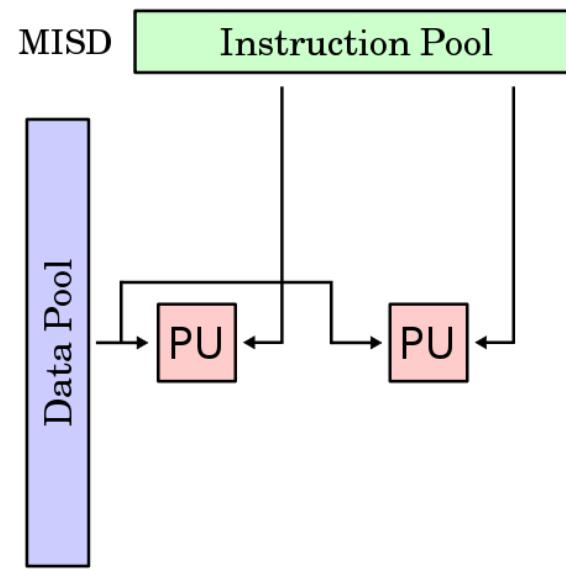
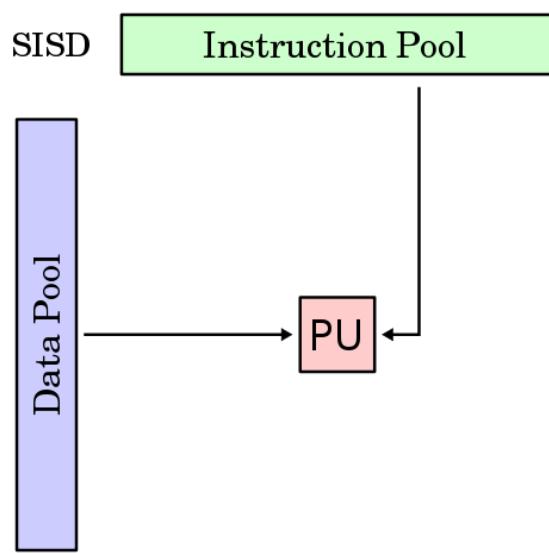


(a)



(b)

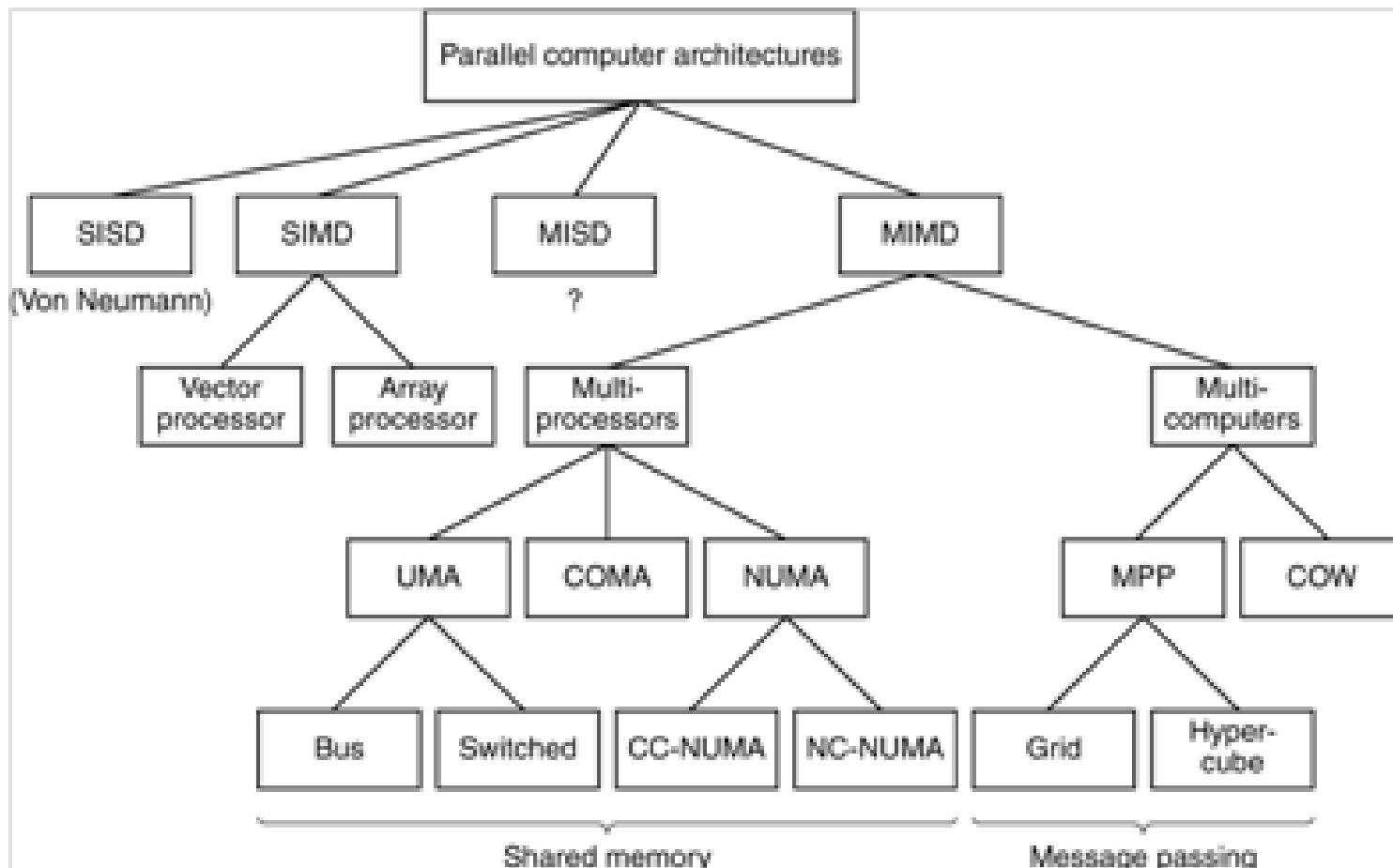
# Sumar -scheme Comparative – clasificare Flynn



# Paralelizare la nivel hardware – istoric

- Etapa 1 (1950s): executie secventiala a instructiunilor
- Etapa 2 (1960s): *sequential instruction issue*
  - Executie Pipeline,
  - *Instruction Level Parallelism* (ILP)
- Etapa 3 (1970s): procesoare vectoriale
  - Unitati aritmetice care fol. Pipeline
  - Registrii, sisteme de memorie paralele *multi-bank*
- Etapa 4 (1980s): SIMD si SMPs
- Etapa 5 (1990s): MPPs si clustere
  - *Communicating sequential processors*
- Etapa 6 (>2000): many-cores, multi-cores, acceleratori, heterogenous clusters

# Vedere generala



# MIMD

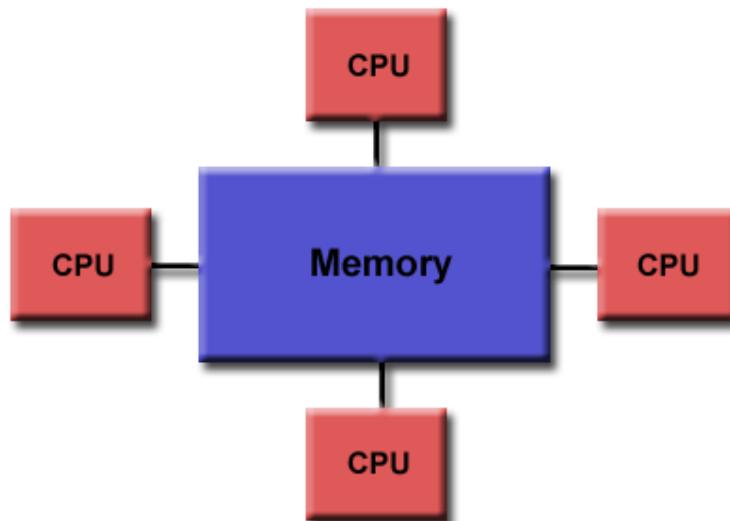
- Clasificare in functie de tipul de memorie
  - partajata
  - distribuita
  - hibrida

# Memorie partajata/ Shared Memory

- Toate procesoarele pot accesa intreaga memorie -> un singur spatiu de memorie (*global address space.*)
- Shared memory=> 2 clase mari: **UMA** and **NUMA**.

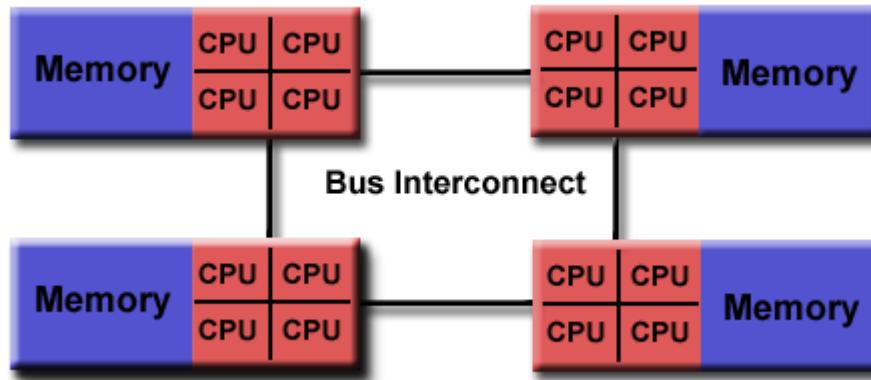
# Shared Memory (UMA)

- **Uniform Memory Access (UMA):**
- Acelasi timp de acces la memorie
- **CC-UMA - Cache Coherent UMA.** (daca un procesor modifica o locatie de memorie toate celelalte “stiu” despre aceasta modificare.  
Cache coherency se obtine la nivel hardware.



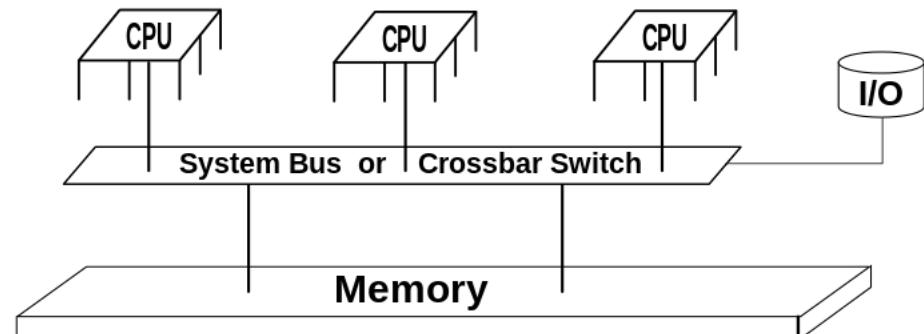
# Non-Uniform Memory Access (NUMA):

- Se obtine deseori prin unirea a 2 sau mai multe arhitecturi UMA
- Nu e acelasi timp de acces la orice locatie de memorie
- **Poate fi si varianta CC-NUMA - Cache Coherent NUMA**
  - ex. HP's Superdome, SUN15K, IBMp690



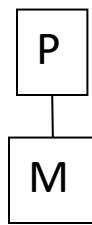
# ***SMP Symmetric multiprocessor computer***

- acces similar la toate procesoarele dar si la I/O devices, USB ports ,hard disks,...
- o singura memorie comună
- un sistem de operare
- controlul procesoarelor – egal (similar)
- distributia threadurilor – echilibrata+echidistanta
- exemplu simplu: 2 procesoare Intel Xeon-E5 processors ->aceeasi motherboard
- Ex. - servere

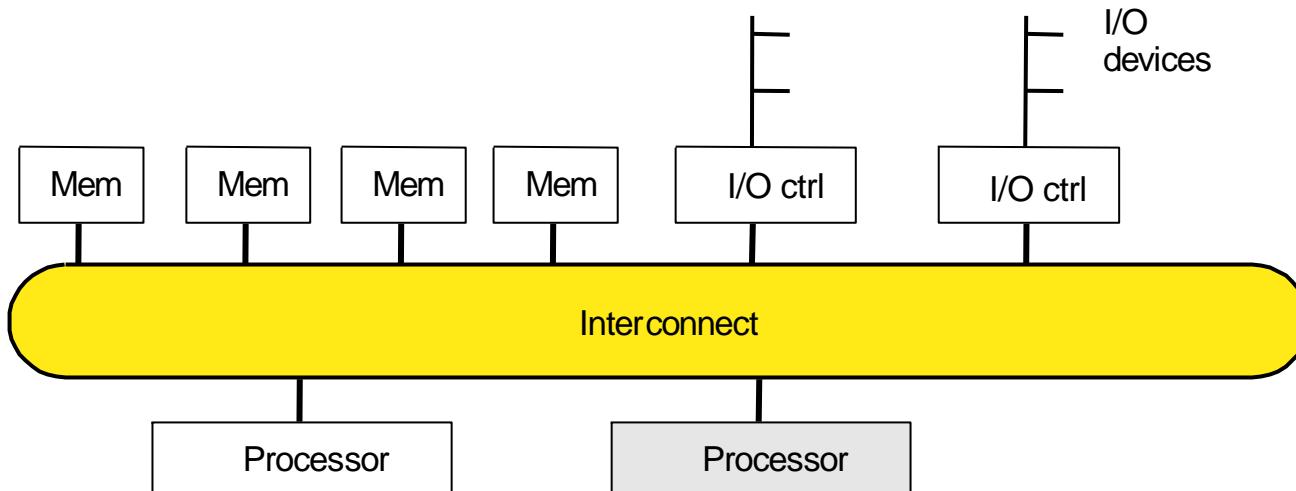
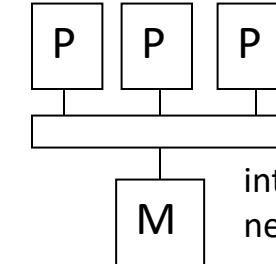
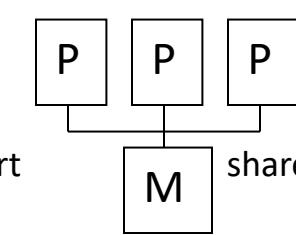
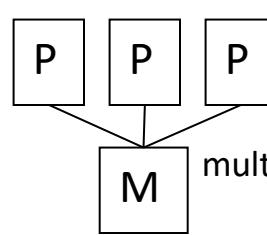


# Shared Memory Multiprocessors (SMP) - overview

Single processor

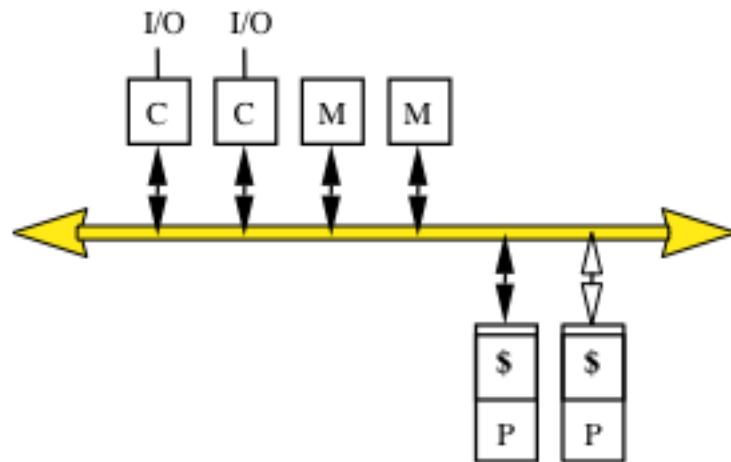


Multiple processors

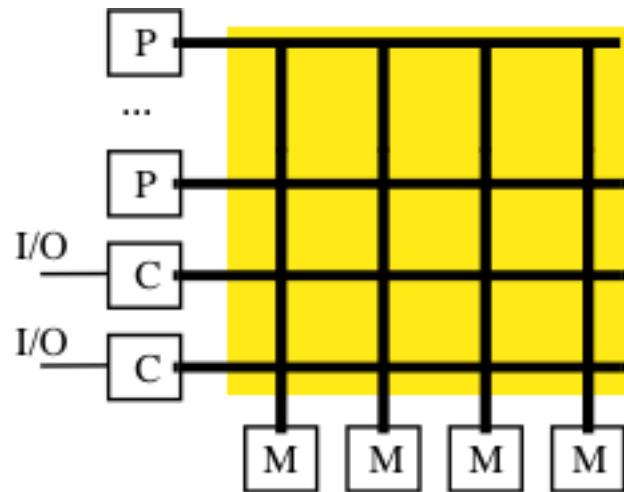


# Bus-based SMP(Symmetric Multi-Processor)

- *Uniform Memory Access (UMA)*
- Pot avea module multiple de memorie



# Crossbar SMP



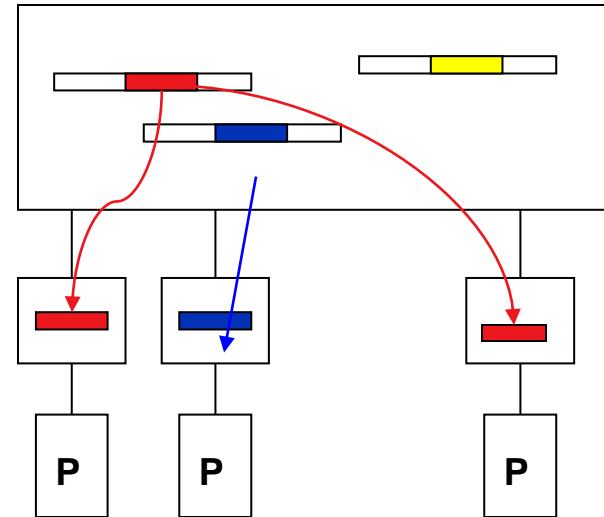
# Parametrii de performanta corespunzatori accesului la memorie

- Latenta = timpul in care o data ajunge sa fie disponibila la procesor dupa ce s-a initiat cererea.
- Largimea de banda (*Bandwidth*) = rata de transfer a datelor din memorie catre procesor
  - store reg → mem
  - load reg ← mem

# *Caching in sistemele cu memorie partajata*

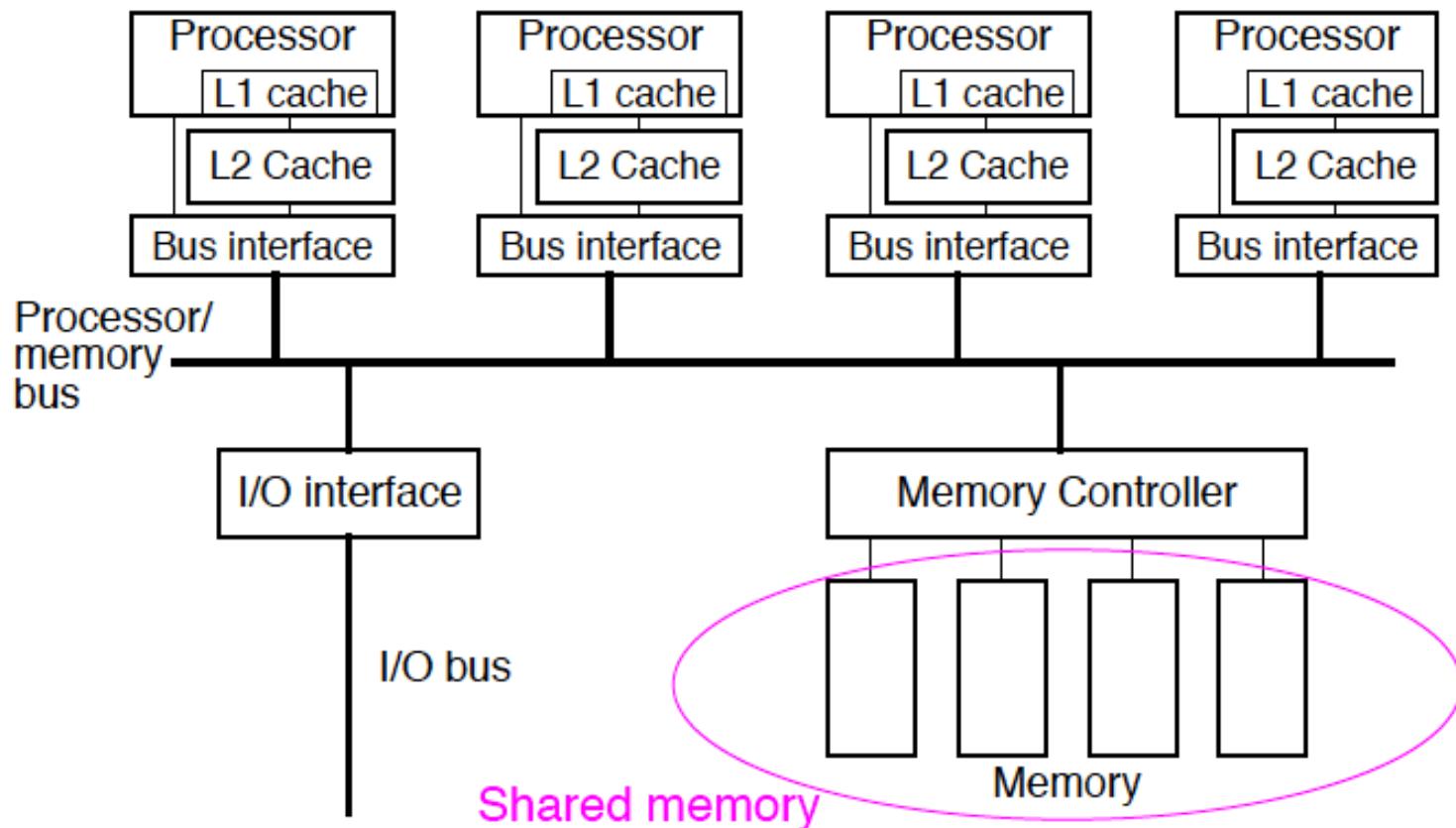
- Folosirea memorilor cache intr-un system de tip SPM introduce probleme legate de ***cache coherency***:

– Cum se garanteaza faptul ca atunci cand o data este modificata, aceasta modificare este reflectata in celelalte memorii cache si in main memory?

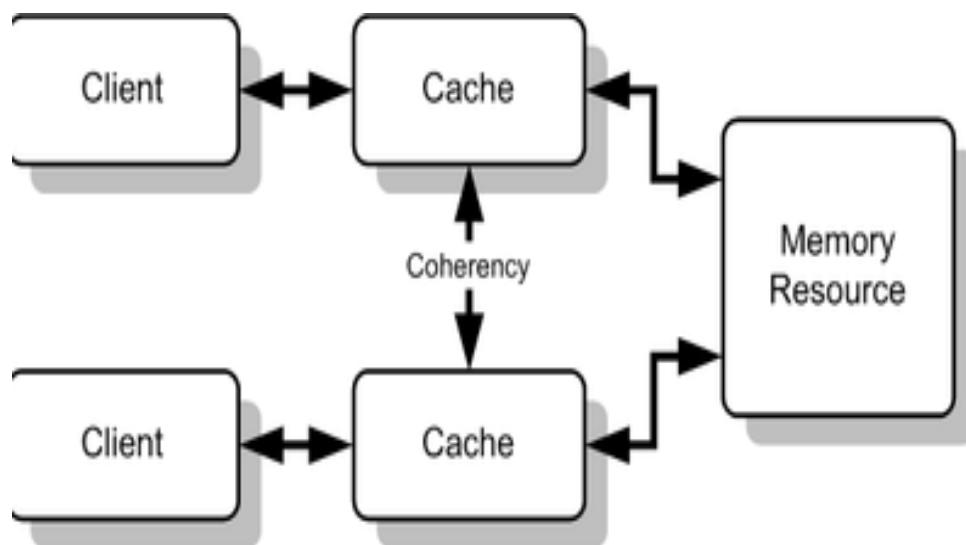


- *coherency* diminueaza scalabilitatea
  - shared memory systems=> maximum 60 CPUs (2016).

# Niveluri de caching



# *Cache coherence*



# *Cache Coherency <-> SMP*

- Memoriile cache sunt foarte importante in SMP pentru asigurarea performantei
  - Reduce timpul mediu de acces la date
  - Reduce cerinta pentru largime de banda- *bandwidth*- plasate pe interconexiuni partajate
- Probleme coresp. *processor caches*
  - Copiile unei variabile pot fi prezente in cache-uri multiple;
  - o scriere de catre un procesor poate sa nu fie vizibila altor procesoare
    - acestea vor avea valori vechi in propriile cache-uri

⇒ *Cache coherence problem*
- Solutii:
  - organizare ierarhica a memoriei;
  - Detectare si actiuni de actualizare.

# Motivatii pentru asigurarea consistentei memoriei

- Coerenta implica faptul ca scrierile la o locatie devin vizibile tuturor procesoarelor in aceeasi ordine .
- cum se stabileste ordinea dintre o citire si o scriere?
  - Sincronizare (*event based*)
  - Implementarea unui protocol hardware pentru *cache coherency*.
  - Protocolul se poate baza pe un model de consistenta a memoriei.

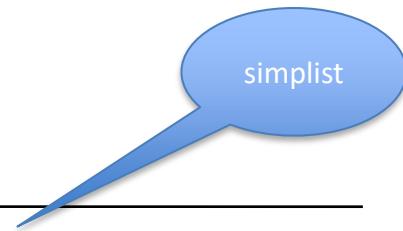
$P_1$                            $P_2$

---

```
/* Assume initial value of A and flag is 0 */

A = 1;
flag = 1;

while (flag == 0); /* spin idly */
print A;
```



simplist

# Asigurarea consistentei memoriei

- Specificare de constrangeri legate de ordinea in care operatiile cu memoria pot sa se execute.
- Implicatii exista atat pentru programator cat si pentru proiectantul de sistem:
  - programatorul le foloseste pentru a asigura corectitudinea ;
  - proiectantul de sistem le poate folosi pentru a constrange gradul de reordonare a instructiunilor al compilatorului sau al hardware-ului.
- Contract intre programator si sistem.

## *(Consistenta secventiala) Sequential Consistency*

- Ordine totala prin intreteserea accesurilor de la diferite procesoare
  - *program order*
  - Operatiile cu memoria ale tuturor procesoarelor par sa inceapa, sa se execute si sa se termine 'atomic' ca si cum ar fi doar o singura memorie (no cache).

*"A multiprocessor is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*  
[Lamport, 1979]

# Shared Memory

## Avantaje:

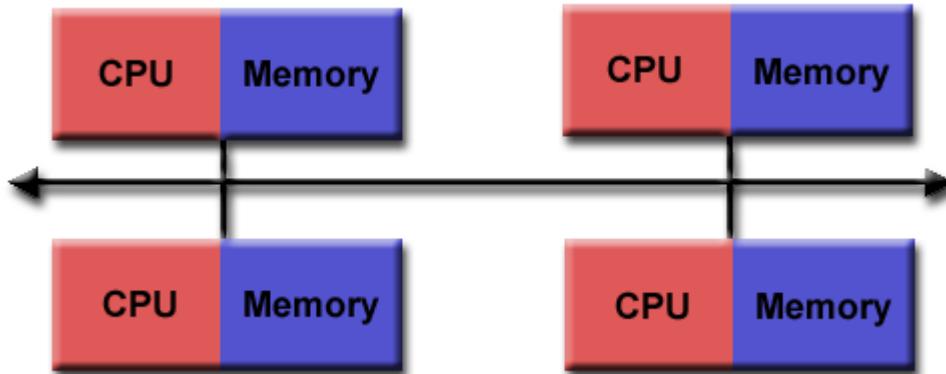
- *Global address space*
- *Partajare date rapida si uniforma*

## Dezavantaje:

- Lipsa scalabilitatii
- Sincronizare in sarcina programatorului
- Costuri mari

# Arhitecturi cu Memorie Distribuită/*Distributed Memory*

- Retea de interconectivitate /***communication network***
- Procesoare cu memorie locală ***local memory***.



# Arhitecturi cu memorie distribuita

## **Avantaje:**

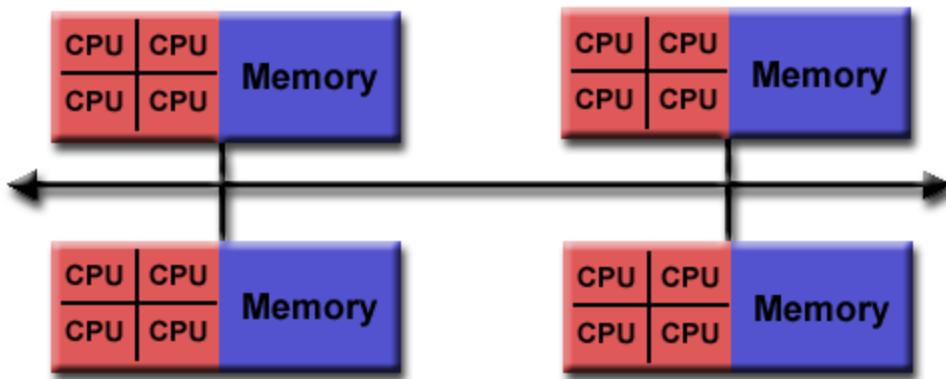
- Memorie scalabila – odata cu cresterea nr de procesoare
- Cost redus – retele

## **Dezavantaje:**

- Responsibilitatea programatorului sa rezolve comunicatiile.
- Dificil de a mapa structuri de date mari pe mem. distribuita.
- Acces Ne-uniform la memorie

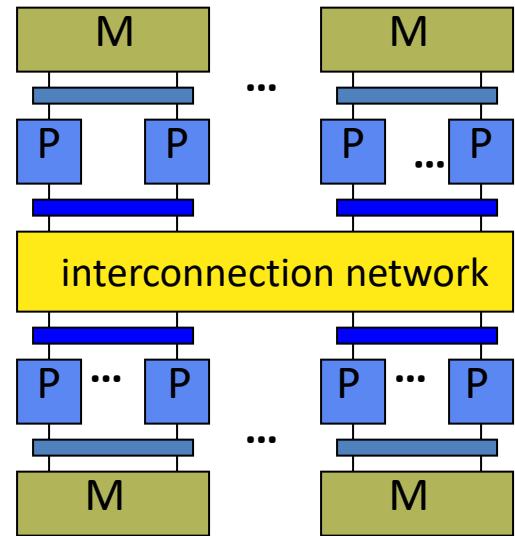
# Hybrid Distributed-Shared Memory

- Retea de SMP-uri



# SMP Cluster

- Clustering
  - Noduri integrate
- Motivare
  - Partajare resurse
  - Se reduc costurile de retea
  - Se reduc cerintele pt largimea de banda (*bandwidth*)
  - Se reduce latenta globala
  - Creste performanta per node
  - Scalabil



# MPP(Massively Parallel Processor)

- Fiecare nod este un sistem independent care are local:
  - Memorie fizica
  - Spatiu de adresare
  - Disc local si conexiuni la retea
  - Sistem de operare
- *MPP (massively parallel processing) is the coordinated processing of a program by multiple processors that work on different parts of the program, with **each processor using its own operating system and memory**. Typically, MPP processors communicate using some messaging interface. In some implementations, up to 200 or more processors can work on the same application. An "interconnect" arrangement of data paths allows messages to be sent between processors. Typically, the setup for MPP is more complicated, requiring thought about how to partition a common database among and how to assign work among the processors. An MPP system is also known as a "loosely coupled" or "shared nothing" system.*
- *An MPP system is considered better than a symmetrically parallel system ( SMP ) for applications that allow a number of databases to be searched in parallel. These include decision support system and data warehouse applications.*

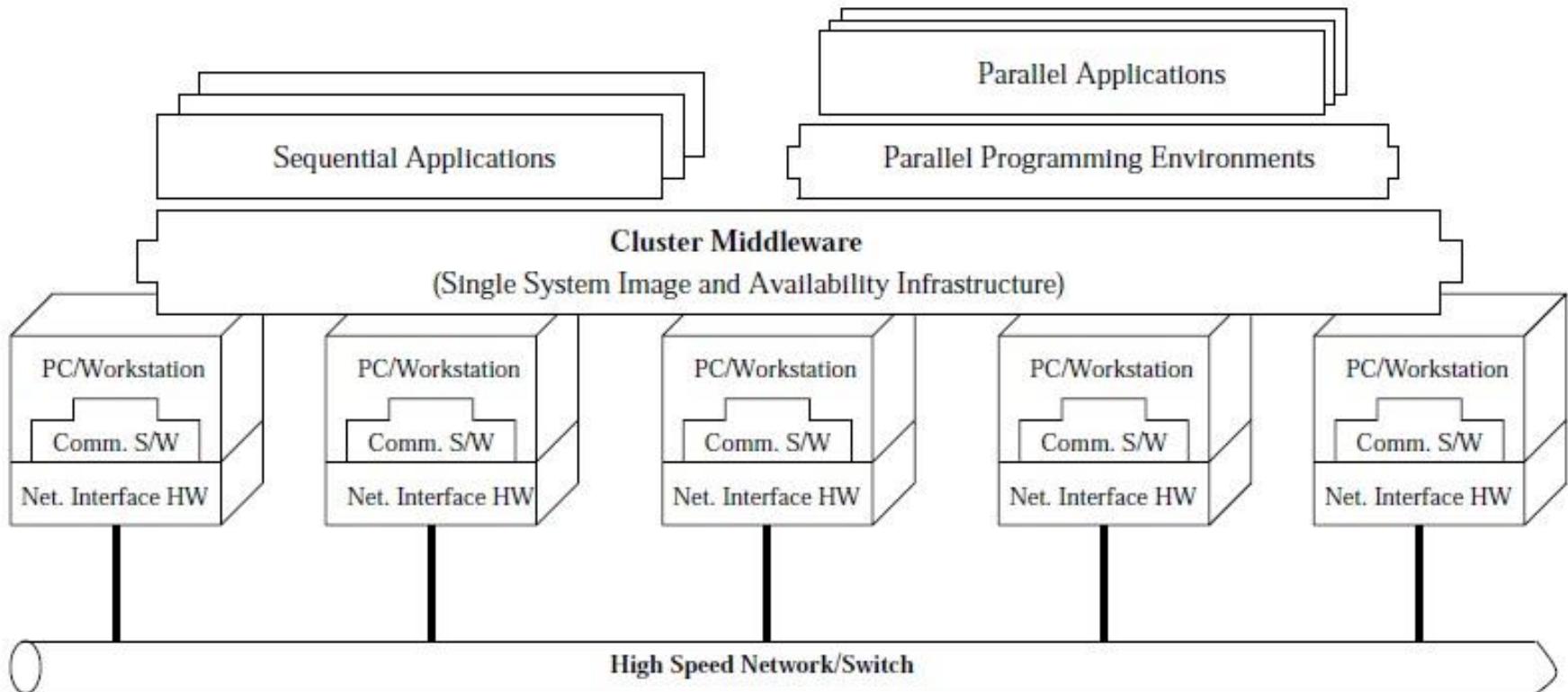
	<b>Symmetric Multi Processor</b>	<b>Massive Parallel Processor</b>
1	SMP stands for Symmetric Multi processor	MPP stands for Massive parallel Processing
2	In SMP every processor share a single copy of the operating system (OS)	In MPP each processor use its own operating system (OS) and memory.
3	SMP supports shared Architecture.	MPP supports shared nothing Architecture
4	SMP is the primary parallel architecture employed in servers	MPP is the coordinated processing of a single task by multiple processors,
5	SMP architecture is a tightly coupled multiprocessor system	In MPP each processor works on a different part of the task.
6	In SMP resources like bus, memory and an I/O system are common	Each processor has its own set of disks
7	SMP processor share whole work between them	Each node is responsible for processing only the rows on its own disk
8	No Separate buffer pool or lock tables, All is shared	Each node maintains its own set of lock tables and buffer pool increasing usability of in memory feature
9	SMP grows by buying a bigger System	Scalability is easy by just adding racks , from few TB's to 6 peta byte
10	SMP usually faces resource contention	MPP is solution to resource contention
11	To create Distributed Architecture complex design is required and can only achieve partially.	MPP is designed to be Distributed Architecture
12	In – Memory feature provided by software is totally depending on amount of RAM and load.	Data is Horizontally Partitioned with huge compression up to 40 x explores in-memory in best manners
13	In SMP every CPU have its own cache either its dual or quad core but rest all resources are shared	MPP processors communicate between each other using some form of Messaging interface

# COMA

- Cache-Only Memory Architecture
- Each memory module acts as a huge cache memory in which each block has a tag with the address and the state.
- Increases the chances of data being available locally because the hardware transparently replicates the data and migrates it to the memory module of the node that is currently accessing it.

# COW

- Cluster of Workstations



# Scalabilitatea sistemelor de calcul

- Cat de mult se poate mări sistemul?
  - unități de procesare,
  - unități de memorie
- Cate procesoare se pot adăuga fără a se diminua caracteristicile generale ale acestuia (viteză de comunicare, viteză de accesare memorie, etc.)
- Măsuri de eficiență (*performance metrics*)

SMP grows by buying  
a bigger system.



MPP grows by adding  
to the existing system.



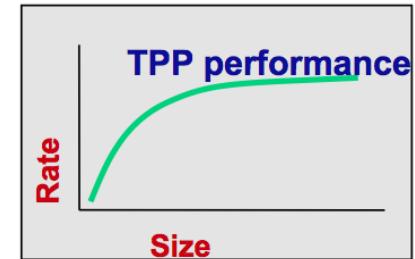
# Performanta

- Problema: daca un procesor este evaluat la nivel k MFLOPS si sunt p procesoare, este performanta totala de ordin  $k \cdot p$  MFLOPS?
- Mai concret: daca un calcul necesita 100 sec. pe un procesor se va putea face in 10 sec. pe 10 procesoare?
- Cauze care pot afecta performanta
  - Fiecare proc. –unitate independenta
  - Interactiunea lor poate fi complexa
  - *Overhead* ...
- *Need to understand performance space*

# Top 500 Benchmarking

<https://www.top500.org/project/linpack/>

- Cele mai puternice 500 calculatoare din lume
- High-performance computing (HPC)
  - Rmax : *maximal performance Linpack benchmark*
    - Sistem dens liniar de ecuatii ( $Ax = b$ )
- Informatii date
  - Rpeak : *theoretical peak performance*
  - Nmax : dimensiunea problemei necesara pt a se atinge Rmax
  - $N^{1/2}$  : dimensiunea problemei necesara pt a se atinge  $1/2$  of Rmax
  - Producator si tipul calculatorului
  - Detalii legate de instalare (location, an,...)
- Actualizare de 2 ori pe an



# UBB CLUSTER – IBM Intelligent Cluster

<http://hpc.cs.ubbcluj.ro/>

- Hybrid architecture
  - HPC system +
  - private cloud



# HPC – IBM NextScale

- Rpeak 62 Tflops, Rmax 40 Tflops
- 68 noduri NX360 M5, din care
  - 12 nodes with 2 GPU Nvidia K40X,
  - 6 nodes with Intel Phi
- 2 processors E5-2660 v3 with 10Cores per node
- 128 GB RAM per node, 2 HDD SATA de 500 Gb / node
- Subscription rate 1:1 between nodes based on Switch: IB Mellanox SX6512 with 216 ports
- Storage NetApp E5660, 120 HDD SAS cu 600 Gb/Hdd => total 72Tb
  - IBM GPFS 4.x -parallel file system
- IBM TS3100 Tape library for data archivation
- Operating systems on each node : RedHat Linux 6 with subscription
- Management Software: IBM Platform HPC 4.2

# Private Cloud – IBM Flex System

- 10 virtualization servers Flex System x240
  - 128 Gb RAM / server
  - Procesoare 2 x Intel Xeon E5-2640 v2 / server
  - 2 x SSD SATA 240 Gb / server
- 1 management server
- Software for private cloud: IBM cloud manager with OpenStack 4.2
- Software for monitoring and management: IBM Flex System Manager software stack
- Virtualization software: Vmware vSphere Enterprise 5.1

## SUMARIZARE

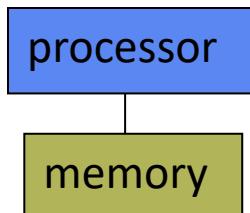
Vedere actuală asupra tipurilor de arhitecturi paralele

# Parallel Architecture Types

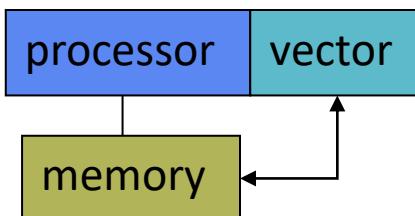
imagini preluate de la course pres. Introduction to Parallel Computing CIS 410/510, Univ. of Oregon

- Uniprocessor

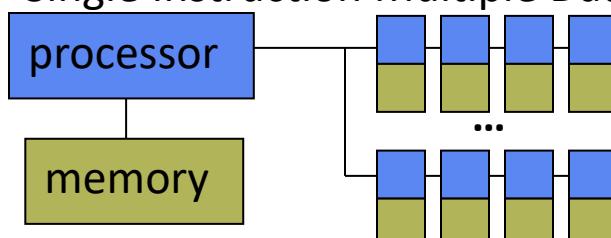
- Scalar processor



- Vector processor



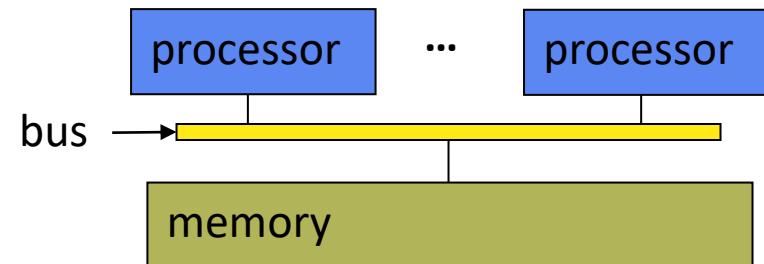
- Single Instruction Multiple Data



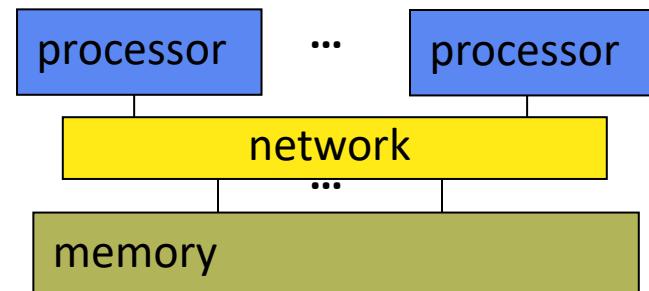
- Shared Memory

- Multiprocessor (SMP)

- Shared memory address space
  - Bus-based memory system

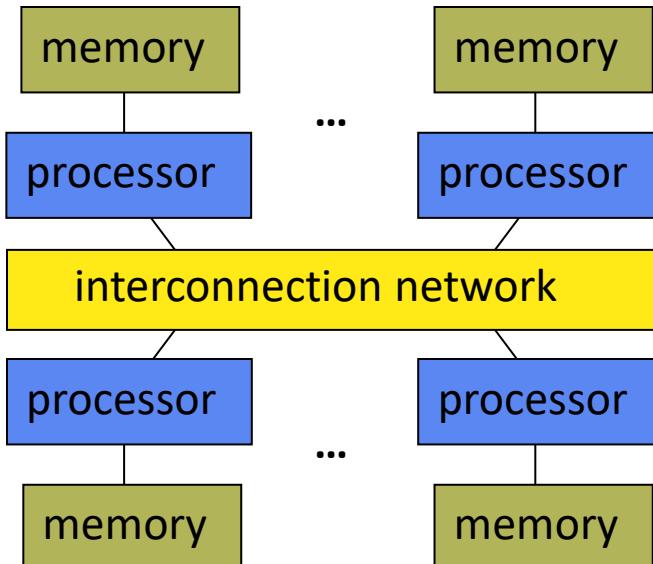


- Interconnection network

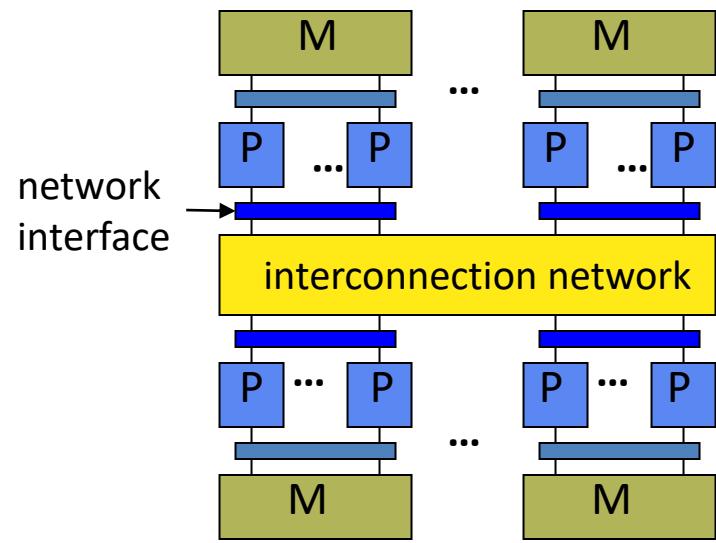


# Parallel Architecture Types (2)

- Distributed Memory Multiprocessor
  - Message passing between nodes
- Cluster of SMPs
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes



- Massively Parallel Processor (MPP)
  - many, many processors
  - fast interconnection

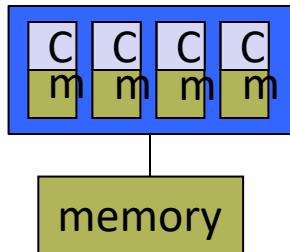


- Can also be regarded as MPP if processor number is large and the communication is fast

# Parallel Architecture Types (3)

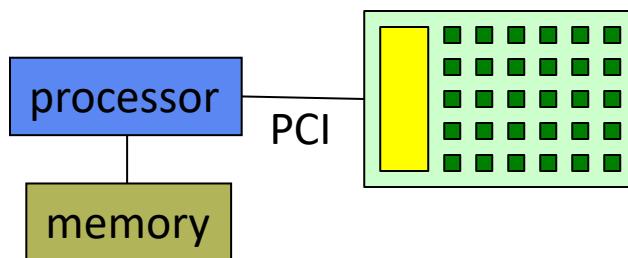
## □ Multicore

- Multicore processor

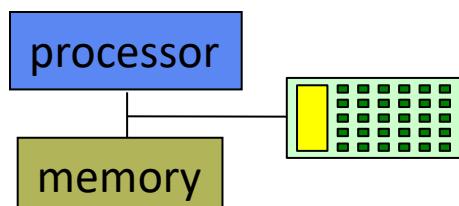


cores can be hardware multithreaded (hyperthread)

- GPU accelerator

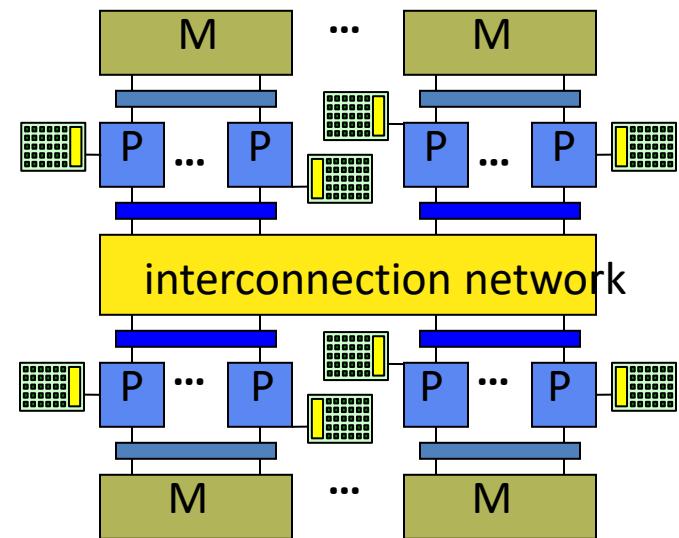


- “Fused” processor accelerator



- Multicore SMP+GPU Cluster

- Shared memory addressing within SMP node
- Message passing between SMP nodes
- GPU accelerators attached



# Curs 3

## Programare Paralela si Distribuita

Paralelism la nivel de instrucțiune

Paralelism implicit versus paralelism explicit

Modele de calcul versus sisteme

Procese versus Fire de executie

Race-condition - sectiuni critice

## PARALLELISM LA NIVEL DE INSTRUCTIUNE

# Parallelism in the CPU

(ref: [https://www.tutorialspoint.com/cuda/cuda\\_tutorial.pdf](https://www.tutorialspoint.com/cuda/cuda_tutorial.pdf) )

- Following are the five essential steps required for an instruction to finish:
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Instruction execute (Ex)
  - Memory access (Mem)
  - Register write-back (WB)
- This is a basic five-stage RISC architecture.
- There are multiple ways to achieve parallelism in the CPU.
  - one is ILP (Instruction Level Parallelism), also known as pipelining.

# Instruction Level Parallelism

- The following figure will help you understand how *Instruction Level Parallelism* works:
- Using instruction pipelining, the instruction throughput has increased. Now, we can process many instructions in one-clock cycle. But for ILP, the resources of a chip would have been sitting idle.**

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

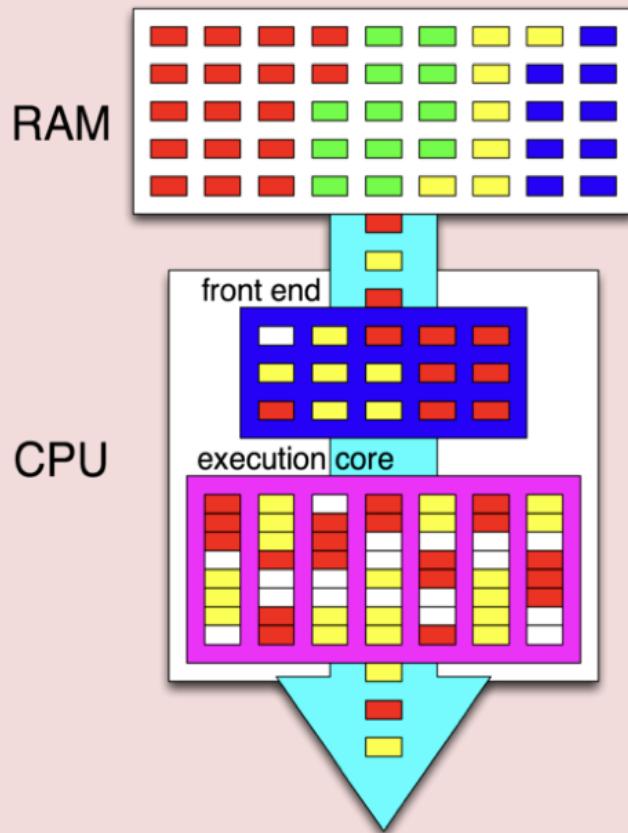
# ILP in a superscalar architecture

- The primary difference between a **superscalar** and a **pipelined** processor is (a superscalar processor is also pipelined) that the former uses multiple execution units (on the same chip) to achieve ILP whereas the latter divides the EU in multiple phases to do that.
  - This means that in superscalar, ***several instructions can simultaneously be in the same stage of the execution cycle.*** This is not possible in a simple pipelined chip.
  - Superscalar microprocessors can execute two or more instructions at the same time. They typically have at least 2 ALUs.
- **Superscalar processors can dispatch multiple instructions in the same clock cycle.**
  - This means that multiple instructions can be started in the same clock cycle.
  - In a pipelined architecture at any clock cycle, only one instruction is dispatched.
  - This is not the case with superscalars. But we have only one instruction counter (in-flight, multiple instructions are tracked). This is still just one process !

# Hyper-threading

- HT is just a technology to utilize a processor core better.
  - Many times, a processor core is utilizing only a fraction of its resources to execute instructions.
- What HT does is that it takes a few more CPU registers, and executes more instructions on the part of the core that is sitting idle.
- Thus, one core now appears as two core.
  - It is to be considered that they are not completely independent.
- If both the ‘cores’ need to access the CPU resource, one of them ends up waiting.
  - That is the reason why we cannot replace a dual-core CPU with a hyper-threaded, single core CPU.
  - A dual core CPU will have truly independent, out-of-order cores, each with its own resources.
- HT is Intel’s implementation of SMT (Simultaneous Multithreading).
- SPARC has a different implementation of SMT, with identical goals.

# SMT(Simultaneous Multithreading)



- The pink box represents a single CPU core.
- The RAM contains instructions of 4 different programs, indicated by different colors.
- The CPU implements the SMT, using a technology similar to hyper-threading.

=> it is able to run instructions of two different programs (red and yellow) simultaneously.
- White boxes represent pipeline stalls.

# Vector processors

- a **vector processor** is a (CPU) that implements an instruction set where its instructions are designed to operate efficiently and effectively on one-dimensional arrays
- a **scalar processor** is a CPU whose instructions operate on single data items only,

BUT

- some of these scalar processors have additional single instruction, multiple data (SIMD) or SWAR Arithmetic Units.

# scalar processors with (SIMD)

- **Pure (fixed) SIMD** - also known as "Packed SIMD" or SIMD within a Register (SWAR)
  - Examples: Intel x86's MMX, SSE and AVX instructions, AMD's 3DNow! extensions, ARM NEON, Sparc's VIS extension, PowerPC's AltiVec and MIPS' MSA
- **Predicated SIMD** -associative processing
  - - examples: ARM SVE2 and AVX-512

# illustration

- For (...i<n...)  $C[i] = A[i] + B[i]$ 
  - In every iteration there will be 2 load, 1 store and 1 add instruction(simple view)
- With SIMD instruction set less than n number of CPU instructions - since each instruction will process multiple elements.
  - with SIMD instruction set that has **128 bit register**, processing on 4 integers at a time is possible:
    - A single load instruction to get 4 values of A into a 128 bit SIMD register
    - A single load instruction to get 4 values of B into a 128 bit SIMD register
    - A single add instruction to add corresponding values in both register.

## PARALLELISM LA NIVEL DE PROGRAM

# Paralelism implicit SAU explicit

- Implicit Parallelism

Programatorul nu specifica explicit paralelismul,  
lăsa compilatorul și sistemul de suport al executiei (*run-time support system*)  
sa paralelizeze automat.

- Explicit Parallelism

Programatorul specifica explicit paralelismul în codul sursă prin  
construcții speciale de limbaj, sau prin directive complexe sau prin  
apeluri de biblioteci.

# Modele de programare paralele Implicite

## **Implicit Parallelism: Parallelizing Compilers**

- Automatic parallelization of sequential programs
  - Dependency Analysis
  - Data dependency
  - Control dependency

Se poate obtine paralelizare dar nu completa si impune analize foarte dificile!

Exemplificare simplă:

*JIT(Just In Time) compilation can choose SSE2 vector CPU instructions when it detects that the CPU supports them*

# Modele de programare paralela Explicită

Cele mai folosite:

- Shared-variable model
- Message-passing model
- Data-parallel model

# Analiza generală a caracteristicilor

Main Features	Data-Parallel	Message-Passing	Shared-Variable
Control flow (threading)	Single	Multiple	Multiple
Synchrony	Loosely synchronous	Asynchronous	Asynchronous
Address space	Single	Multiple	Multiple
Interaction	Implicit	Explicit	Explicit
Data allocation	Implicit or semiexplicit	Explicit	Implicit or semiexplicit

# **Legatura intre Modele de Programare si Arhitecturi**

Exemplificare pe problema concreta:

Suma de numere

Exemplu de aplicatie paralela: calcularea sumei

$$\sum_{i=0}^{n-1} f(A[i])$$

Solutia generala:

$n/p$  operatii   $p$  procesoare (procese).

Se disting doua seturi de date:

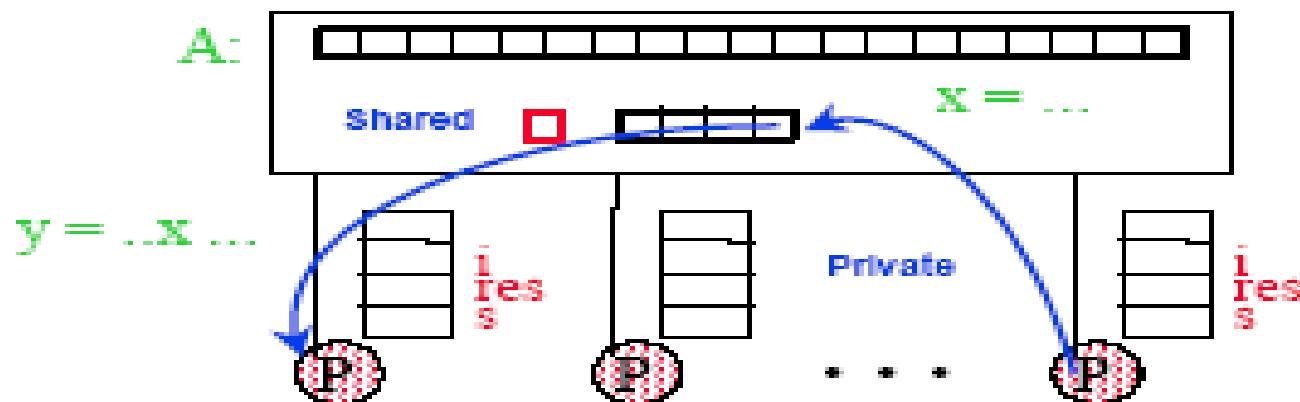
- partajate: valorile  $A[i]$  si suma finala;
- private: evalurile individuale de functii si sumele partiale

## 1) Model de programare: spatiu partajat de adrese.

**Programul** = colectie de fire de executie, fiecare avand un set de variabile private, iar impreuna partajeaza un alt set de variabile.

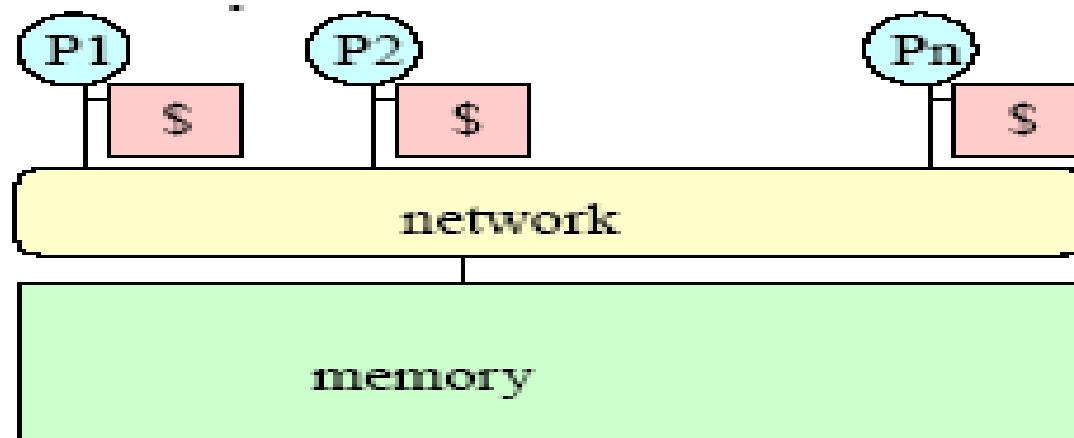
**Comunicatia** dintre firele de executie: **prin citirea/scrierea variabilelor partajate**.

**Coordonarea** firelor de executie prin operatii de **sincronizare**: indicatori (flags), lacate (locks), semafoare, monitoare.



Masina paralela corespunzatoare modelului 1: masina cu memorie partajata (sistemele multiprosesor, sistemele multiprosesor simetrice -- SMP-Symmetric Multiprocessors).

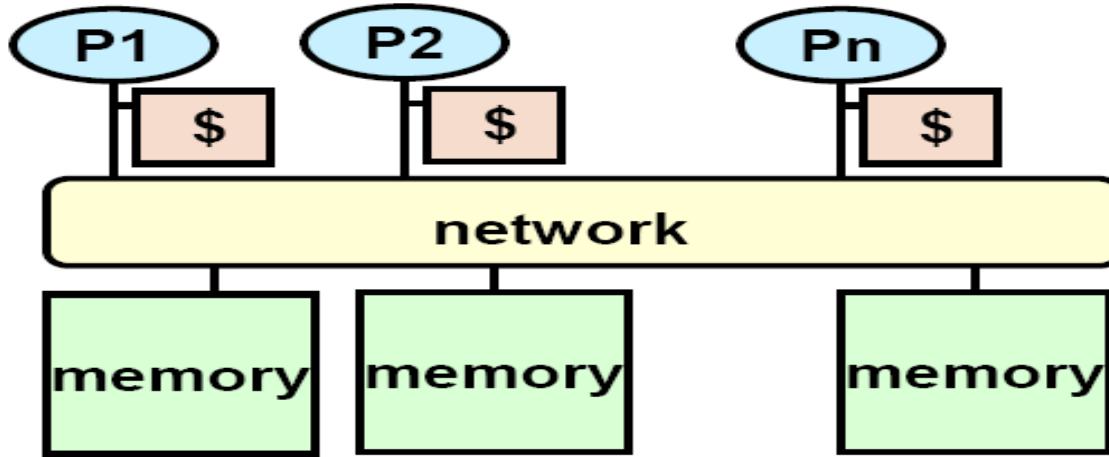
Exemple: sisteme de la Sun, DEC, Intel (Millennium), SGI Origin.



Variante ale acestui model:

a) masina cu memorie partajata distribuita (logic partajata, dar fizic distribuita).

Exemplu: SGI Origin (scalabila la cateva sute de procesoare).



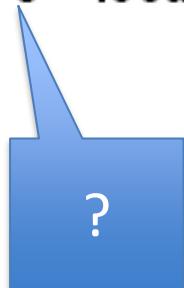
b) masina cu spatiu partajat de adrese (memoriile cache inlocuite cu memorii locale). Exemplu: Cray T3E.

*single address space*

O posibila solutie pentru rezolvarea problemei:

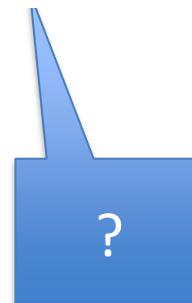
### Thread 1

```
[s = 0 initially]  
local_s1= 0  
for i = 0, n/2-1  
    local_s1 = local_s1 + f(A[i])  
s = s + local_s1
```



### Thread 2

```
[s = 0 initially]  
local_s2 = 0  
for i = n/2, n-1  
    local_s2= local_s2 + f(A[i])  
s = s +local_s2
```



Este necesara sincronizarea threadurilor pentru accesul la variabilele partajate !

Exemplu: prin excludere mutuală, folosind operația de blocare(lock):

### Thread 1

```
lock  
load s  
s = s+local_s1  
store s  
unlock
```

### Thread 2

```
lock  
load s  
s = s+local_s2  
store s  
unlock
```

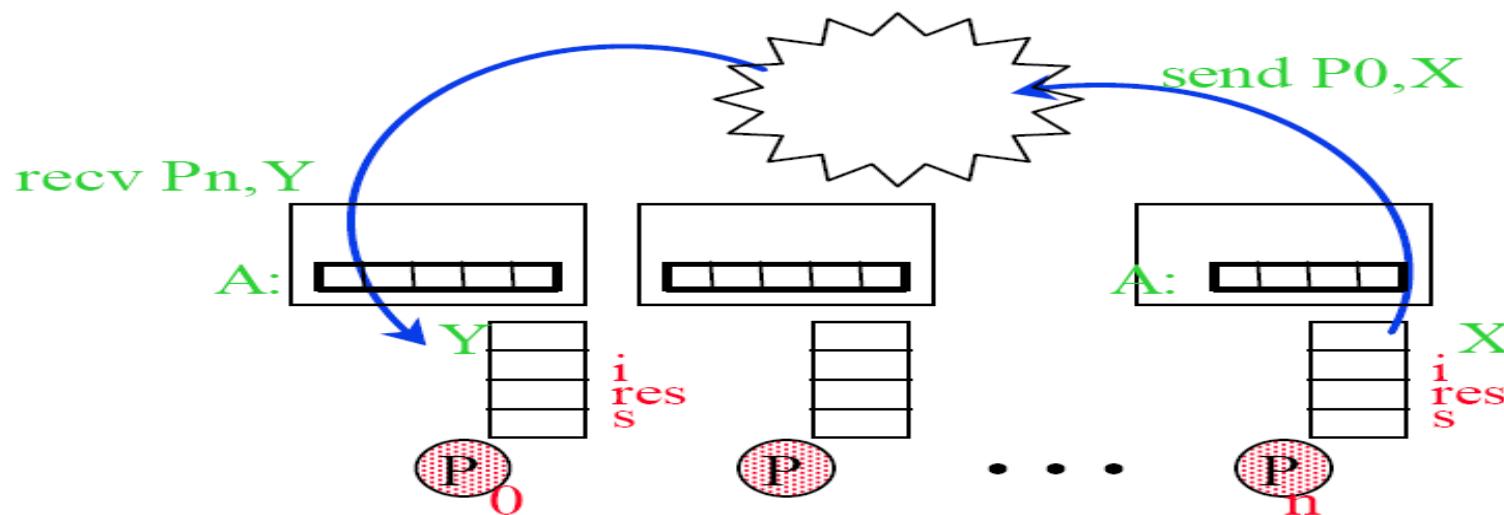
## 2) Modelul de programare: transfer de mesaje.

**Programul** = colectie de procese, fiecare cu thread de control si spatiu local de adrese, variabile locale, variabile statice, blocuri comune.

**Comunicatia** dintre procese: prin transfer explicit de date (perechi de operatii corespunzatoare **send** si **receive** la procesele sursa si respectiv destinatie).

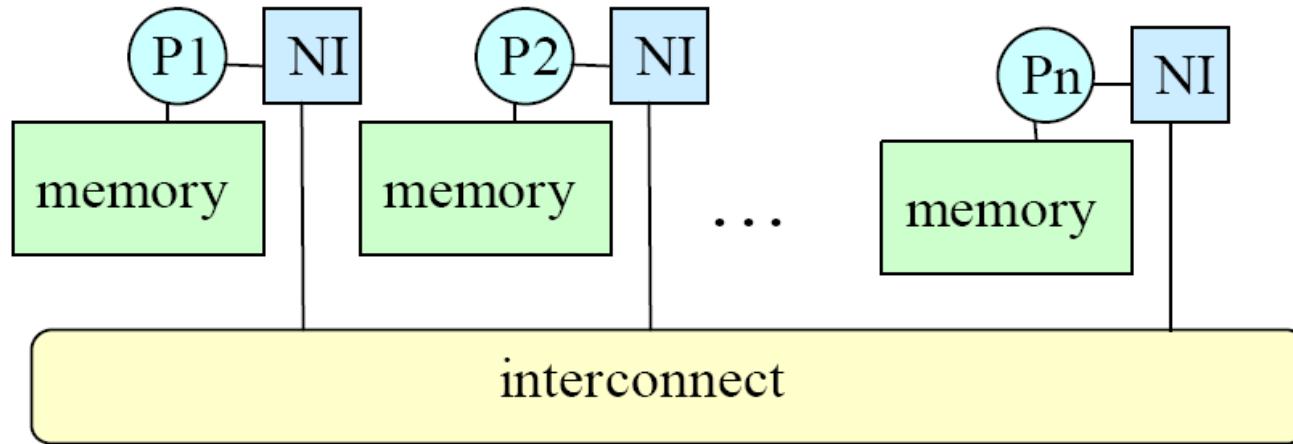
**Coordonarea**: transfer de mesaje

Datele partajate din punct de vedere logic sunt partitionate intre toate procesele.  
=> asemanare cu programarea distribuita!



Exista biblioteci standard (exemplu: MPI si PVM).

Masina corespunzatoare modelului 2  
sistem cu memorie distribuita (multicalculator):



Exemple: Cray T3E (poate fi incadrat si in aceasta categorie), IBM SP2, NOW, Millenium.

O posibila solutie a problemei in cadrul modelului in transfer de mesaje simplificare => suma se calculeaza :  $s = f(A[1]) + f(A[2])$  :  
(consideram operatii: send si receive blocante !!!)

**Procesor 1**

**xlocal = f(A[1])**

**send xlocal, proc2**

**receive xremote, proc2**

**s = xlocal + xremote**

**Procesor 2**

**xlocal = f(A[2])**

**send xlocal, proc1**

**receive xremote, proc1**

**s = xlocal + xremote**

sau:

**Procesor 1**

**xlocal = f(A[1])**

**send xlocal, proc2**

**receive xremote, proc2**

**s = xlocal + xremote**

**Procesor 2**

**xlocal = f(A[2])**

**receive xremote, proc1**

**send xlocal, proc1**

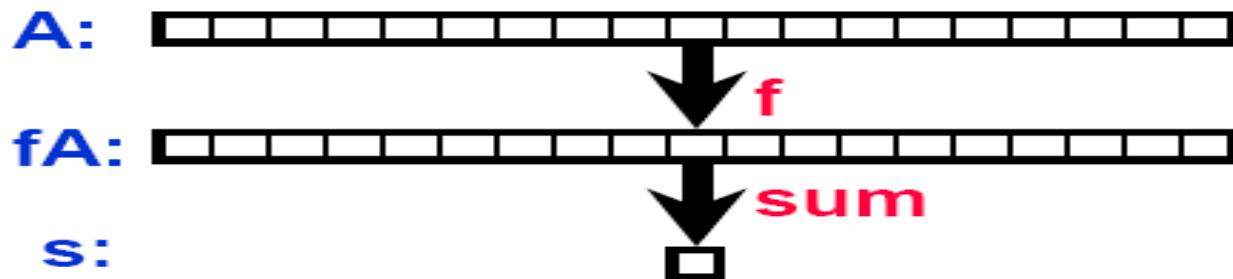
**s = xlocal + xremote**



Deadlock?

### 3) Modelul de programare: paralelism al datelor.

**Program:** Thread singular, secential de control care controleaza un set de operatii paralele aplicate intregii structuri de date, sau numai unui singur subset.

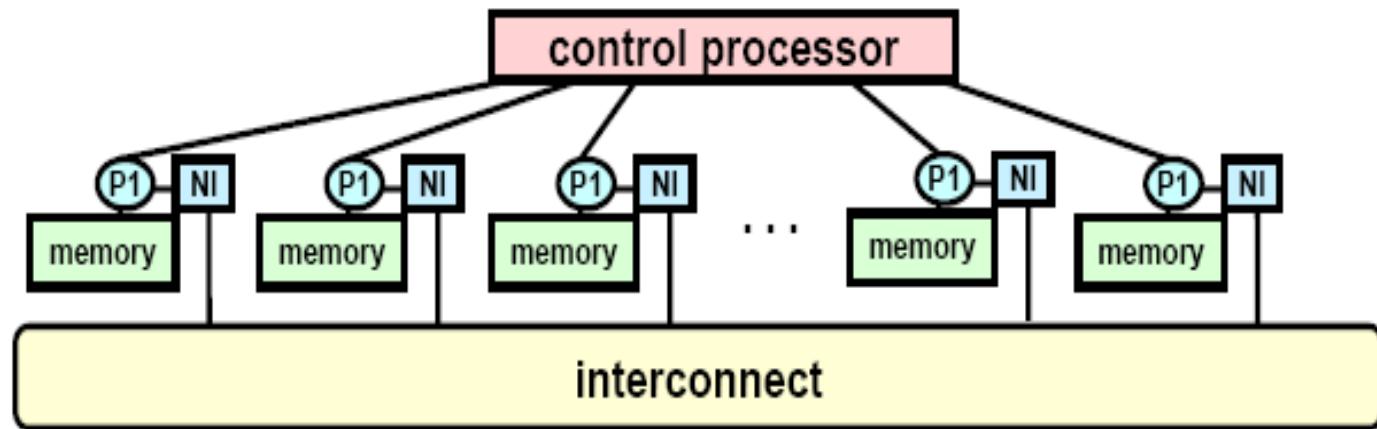


map + reduce

**Comunicatia:** implicita, in modul de deplasare a datelor.

Eficienta numai pentru anumite probleme (exemplu: prelucrari de tablouri)!

Masina corespunzatoare modelului 3: sistem SIMD - Single Instruction Multiple Data (numar mare de procesoare elementare comandate de un singur procesor de control, executand aceeasi instructiune, posibil anumite procesoare inactive la anumite momente de timp - la executia anumitor instructiuni).



Exemple: CM2, MASPAR, sistemele sistolice VLSI

Varianta: masina vectoriala (un singur procesor cu unitati functionale multiple, toate efectuand aceeasi operatie in acelasi moment de timp).

~ GPU

#### **4) Modelul hibrid**

=> **cluster de SMP-uri** sau CLUMP (mai multe SMP-uri conectate intr-o retea).

Fiecare SMP: sistem cu memorie partajata!

Comunicatia intre SMP-uri: prin transfer de mesaje.

Exemple: Millennium, IBM SPx, ASCI Red (Intel).

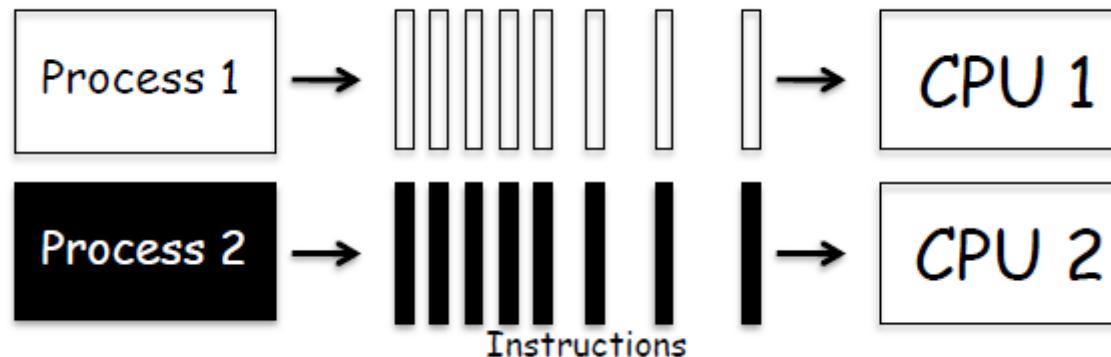
#### **Model de programare:**

- se poate utiliza transfer de mesaje, chiar in interiorul SMP-urilor!
- Varianta hibrida!

## Procese versus Fire de executie

# *Multiprocessing -> Paralelism*

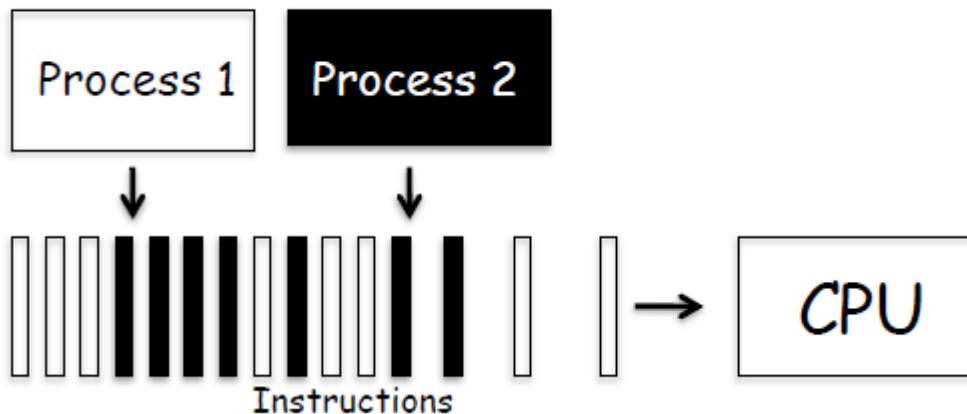
- *Multicore processors*



- Daca se folosesc mai multe unitati de procesare  
=>Procesele se pot executa in acelasi timp

# Multitasking-> Concurenta

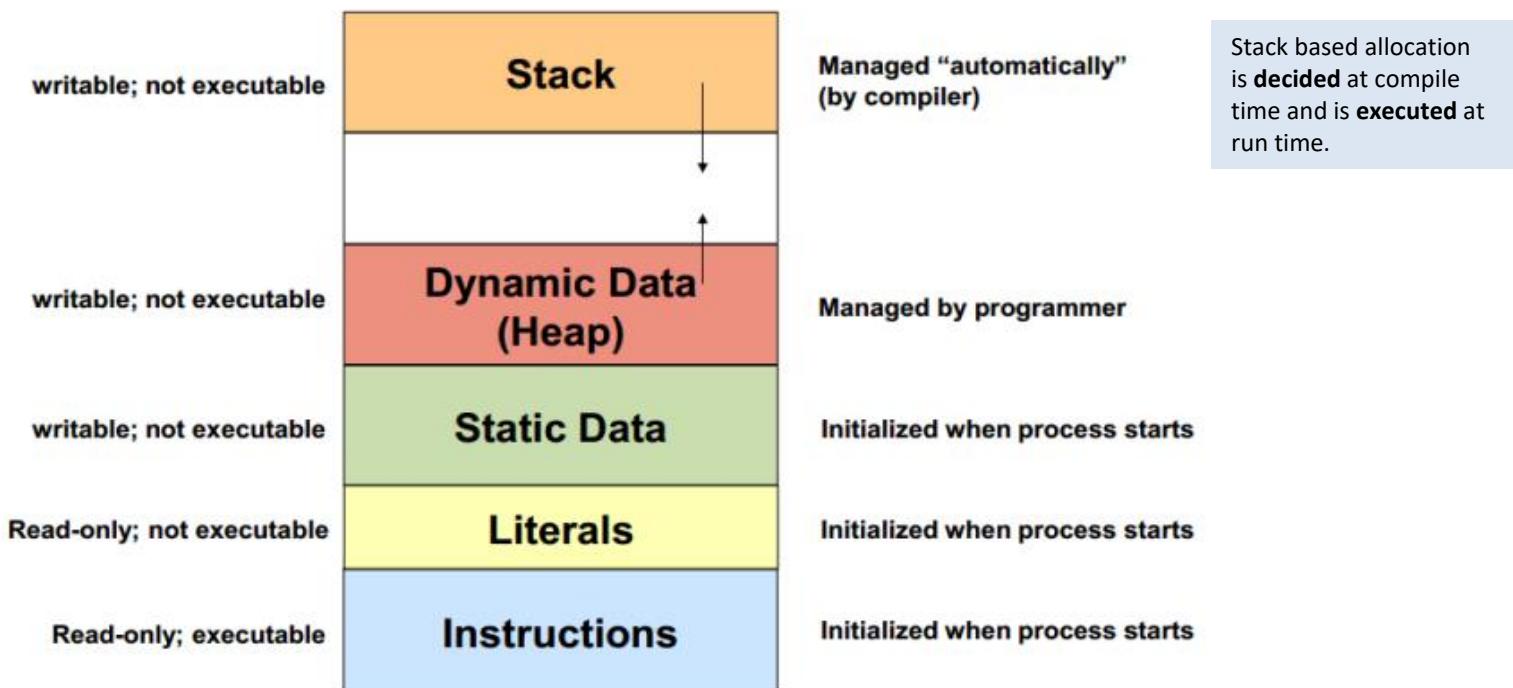
- Sistemul de operare schimba executia intre diferite taskuri
- Resursa comună = CPU



- *Interleaving*
  - sunt mai multe taskuri active, dar doar unul se executa la un moment dat
- *Multitasking*:
  - SO ruleaza executii intretesute

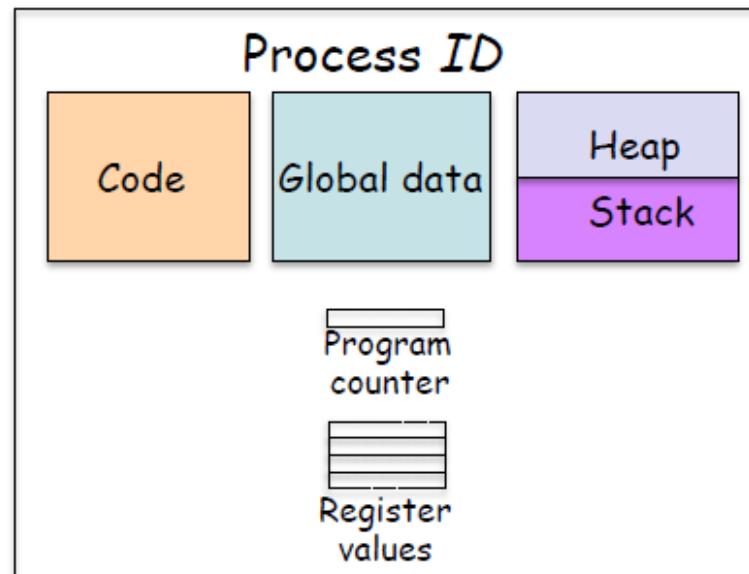
# Procese

- Un program (secvential) = un set de instructiuni  
(in paradigma programarii imperative)
- Un proces = o instanta a unui program care se executa



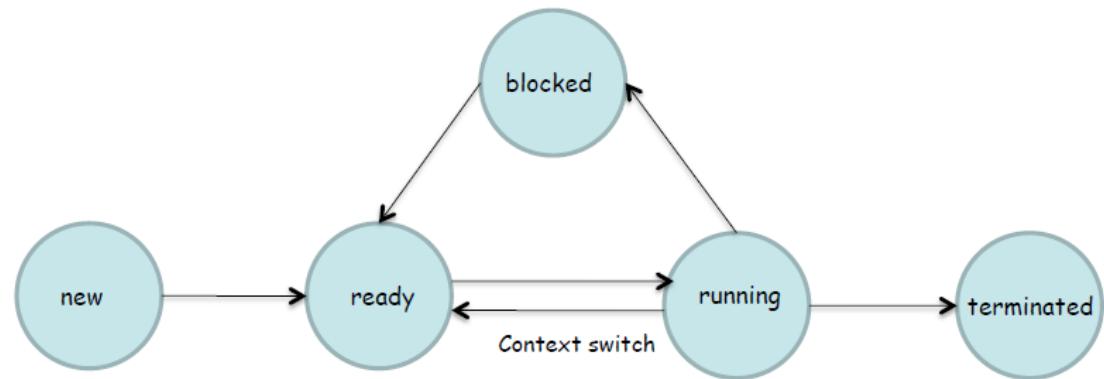
# Procese in sisteme de operare

- Structura unui proces
  - Identifier de proces (ID)
  - Starea procesului (activitatea curentă)
  - Contextul procesului (valori registrii, *program counter*)
  - Memorie (codul program, date globale/statice, stiva si heap)



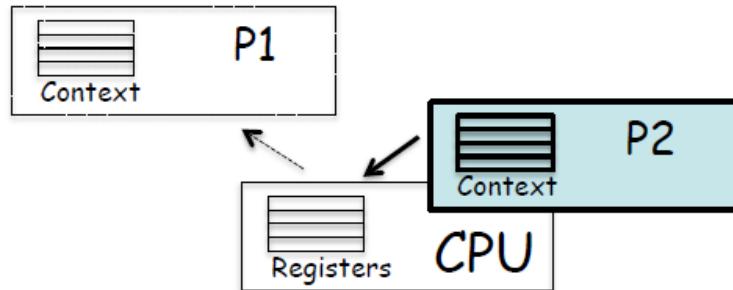
# *Scheduler*

- Un program care controleaza executia proceselor
  - Seteaza starile procesului
    - new
    - running
    - blocked (nu poate fi selectat pt executie; este nevoie de un event extern pt a iesi din aceasta stare)
    - ready
    - terminated



# *Context switch*

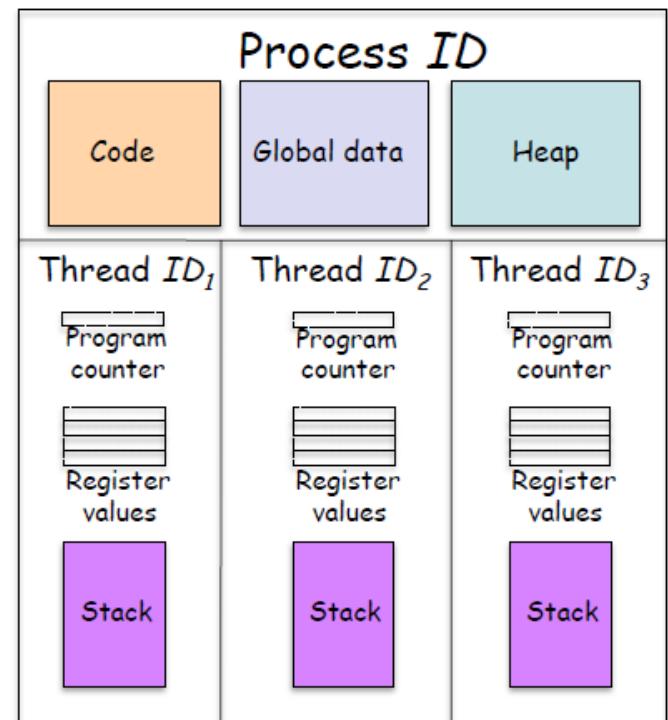
- Atunci cand *scheduler*-ul schimba procesul executat de o unitate de procesare



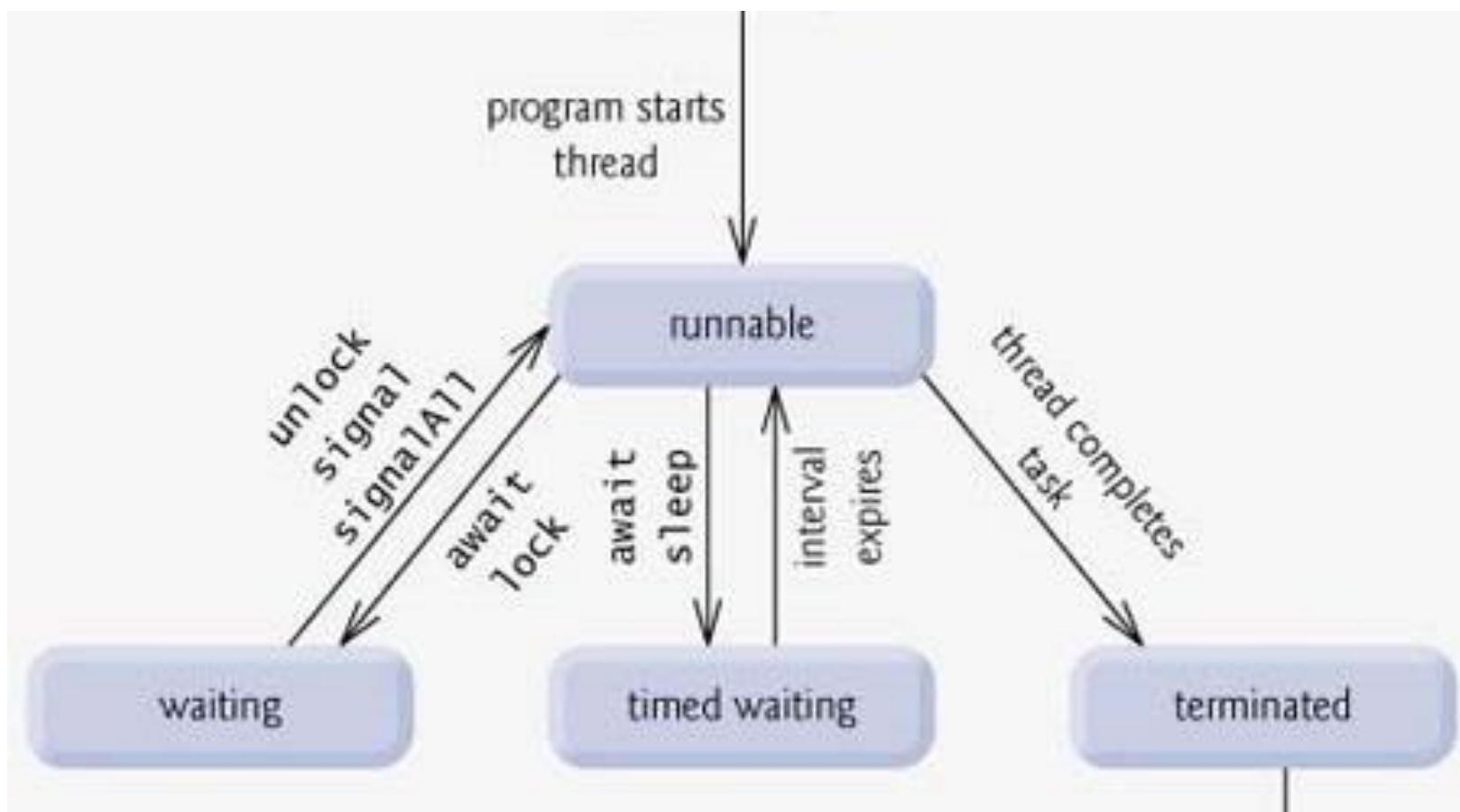
- Actiuni asociate:
  - $P1.state := \text{ready}$
  - Se salveaza valoarea registrilor in memorie dar si contextul lui P1
  - Se foloseste contextul lui P2 pt a seta registrii
  - $P2.state := \text{running}$
  - in plus -> switching memory address space:
    - include: memory addresses, page tables, kernel resources, caches

# *Threads*

- Un thread este o parte a unui proces al sistemului de operare
- Componente private fiecarui thread:
  - Identificator
  - Stare
  - Context
  - Memorie (doar stiva)
- Componente partajate cu alte thread-uri
  - Cod program
  - Date globale
  - Heap



# Threads



# Preemptive multitasking

- A **preemptive multitasking operating system** permits preemption of tasks.
- A **cooperative multitasking operating system** - processes or tasks must be explicitly programmed to yield when they do not need system resources.
- Preemptive multitasking involves the use of **an interrupt mechanism** which suspends the currently executing process and invokes a scheduler to determine which process should execute next.
  - Therefore, all processes will get some amount of CPU time at any given time.
- In preemptive multitasking, the operating system kernel can also initiate a context switch to satisfy the scheduling policy's priority constraint, thus preempting the active task.

- ***Non Preemptive threading model:*** Once a thread is started it cannot be stopped or the control cannot be transferred to other threads until the thread has completed its task.
- ***Preemptive Threading Model:*** The runtime is allowed to step in and hand control from one thread to another at any time. Higher priority threads are given precedence over Lower priority threads.

# CPU use

- Processes could:
  - wait for input or output (called "**I/O bound**"),
  - utilize the CPU ("CPU bound").
- In early systems, processes would often "**poll**", or "**busywait**" while waiting for requested input (such as disk, keyboard or network input).
  - During this time, the process was not performing useful work, but still maintained complete control of the CPU.
- With the advent of interrupts and preemptive multitasking, these I/O bound processes could be "**blocked**", or **put on hold**, pending the arrival of the necessary data, allowing other processes to utilize the CPU.
- As the arrival of the requested data would generate an interrupt, blocked processes could be guaranteed a timely return to execution.

# Multitasking advantages

- Although multitasking techniques were originally developed to allow multiple users to share a single machine, multitasking is useful regardless of the number of users.
- Multitasking makes it possible for a single user to run multiple applications at the same time, or to run "background" processes while retaining control of the computer.
- ***Preemptive multitasking allows the computer system to more reliably guarantee each process a regular "slice" of operating time.***
  - It also allows the system to rapidly deal with important external events like incoming data, which might require the immediate attention of one or another process.

# race condition

- **race condition (race hazard)**
- *a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes*

= o conditie care poate sa apară într-un sistem (electronic, software,...) în care **comportamentul este dependent de ordinea în care apar anumite evenimente (necontrolate strict).**

-Daca exista una sau mai multe posibilitati de rezultat (comportament) nedorit => EROARE .

= doua sau mai multe operatii sunt executate in acelasi timp dar natura sistemului impune sequentializarea lor. Daca nu este posibila orice sesequentializare atunci pot apare erori.

- Termenul *race condition* a aparut inainte de 1955,  
(e.g. David A. Huffman's doctoral thesis "The synthesis of sequential switching circuits", 1954)

# critical race condition vs. non-critical race condition

- *critical race condition* = atunci cand ordinea in care se modifica variabilele interne determina starea finala a sistemului
- *non-critical race condition* = atunci cand ordinea in care se modifica variabilele interne NU are impact asupra starii finale a sistemului

T1:

```
update (){  
    a=a+1;  
}
```

T2:

```
update (){  
    b=a*2;  
}
```

# Data race

- A data race occurs when two threads access the same variable concurrently, and at least one of the accesses is a write.
- o situatie in care un thread executa o operatie prin care se incearca sa se accesize o locatie de memorie care este in acelasi timp accesata pentru scriere de catre alt thread.

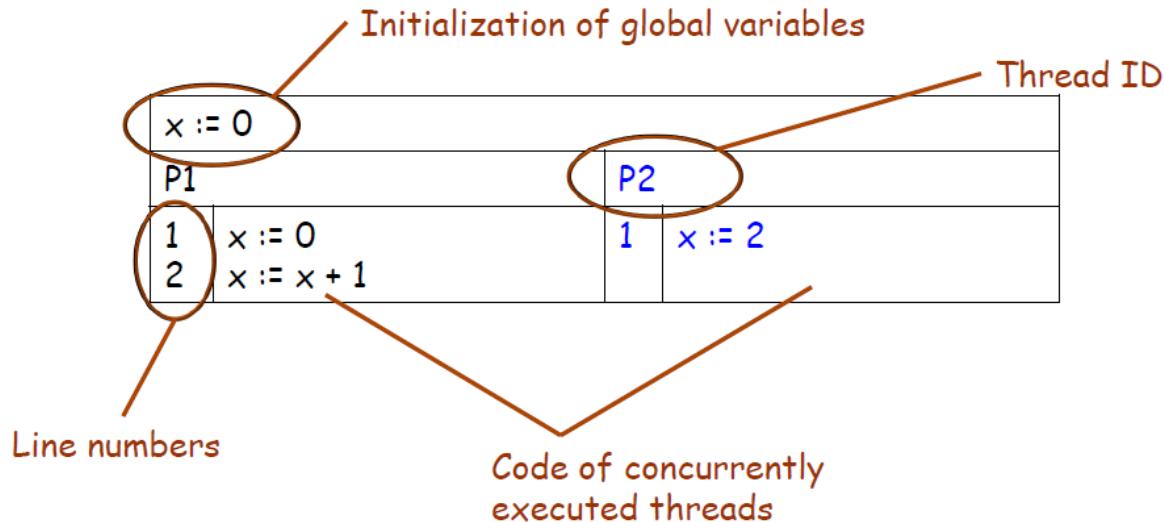
# Deterministic versus non-deterministic computation

- Intr-o executie **determinista** ordinea de executie a operatiilor este complet determinata de specificatii/program
  - ex. program secential
- Intr-o executie **NEdeterminista** ordinea de executie a operatiilor NU este complet determinata de specificatii/program –
  - ex. program parallel

Images from

# Concurrenta cu Threads

Un program care la executie conduce la un proces care contine mai multe threaduri



# Variante de executie

- Secvențe de executie

Instruction executed with Thread ID and line number

Variable values after execution of the code on the line

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

P2	1	x := 2	x = 2
P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1

P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

P1	1	x := 0	x = 0
P1	2	x := x + 1	x = 1
P2	1	x := 2	x = 2

# Instructiuni atomice

- <instr> este atomica daca executia sa nu poate fi “interleaved” cu cea a altelui instructiuni inainte de terminarea ei.
- Niveluri de atomicitate

Ex:  $x := x + 1$

Executie:

temp := x	LOAD REG, x
temp := temp + 1	ADD REG, #1
x := temp	STORE REG, x

# Variante de executie

- exemplul anterior

$x := 0$			
P1		P2	
1	$x := 0$	1	
2	$\text{temp} := x$		
3	$\text{temp} := \text{temp} + 1$		
4	$x := \text{temp}$		

- o executie "interleaving"

P1	1	$x := 0$	$x = 0$
P1	2	$\text{temp} := x$	$x = 0, \text{temp} = 0$
P2	1	$x := 2$	$x = 2, \text{temp} = 0$
P1	3	$\text{temp} := \text{temp} + 1$	$x = 2, \text{temp} = 1$
P1	4	$x := \text{temp}$	$x = 1, \text{temp} = 1$

## Exemplul -2

Două fire de execuție decrementează variabila V până la 0

```
while (v>0)  
    v--;
```

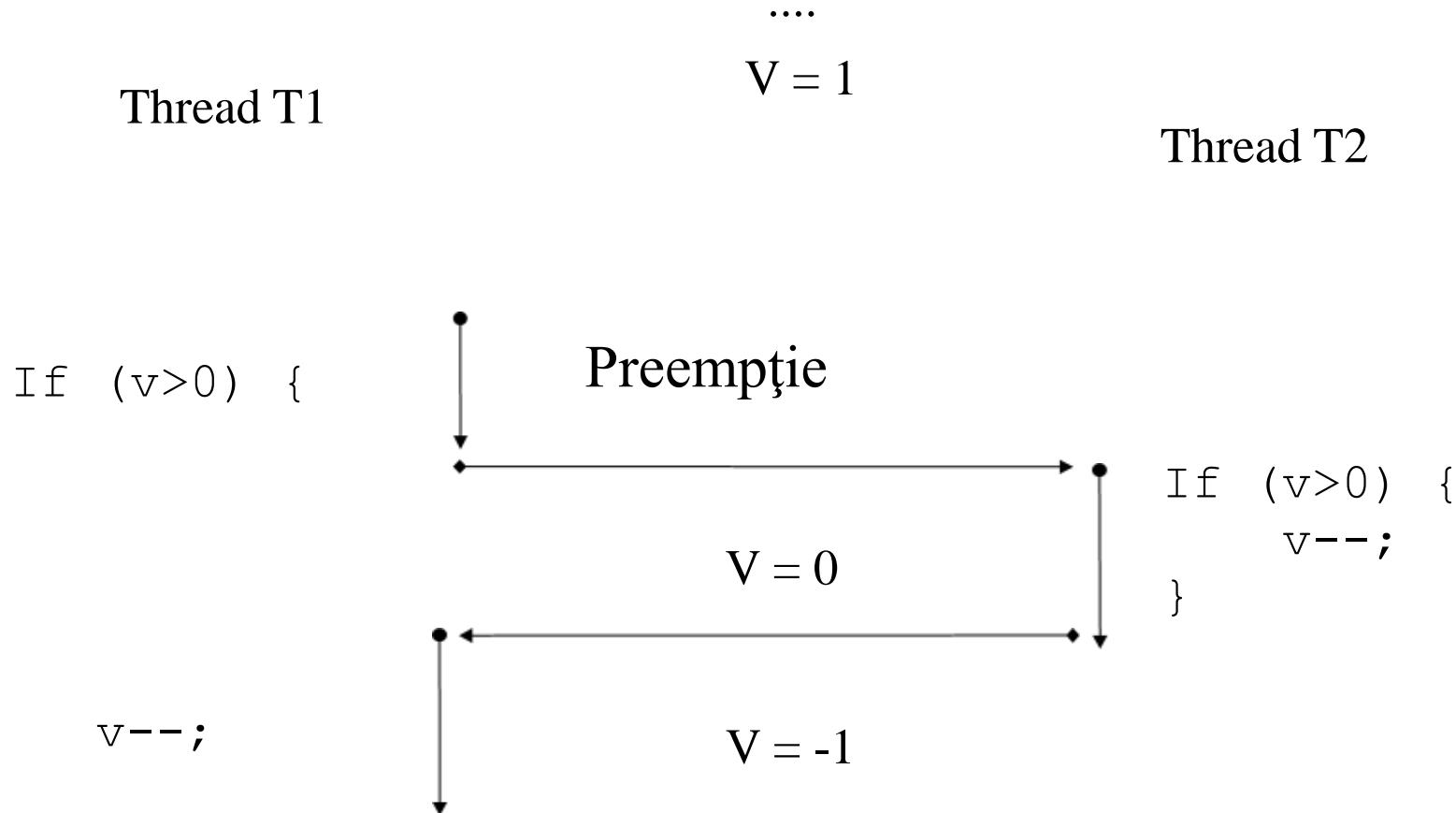
Thread T1

```
while (v>0)  
    v--;
```

Thread T2

Ce valoare va avea variabila V după terminarea execuției celor două fire de execuție?

## Exemplul 2- Varianta de executie



# Race condition & Critical section

- Procese / threaduri independente => executie simpla fara probleme
- Daca exista interactiune (ex. Accesarea si modificarea acelasi variabile) => pot apare probleme
- ***Nondeterministic interleaving***
- Daca rezultatul depinde de interleaving => *race condition*
- Se incearca sa se foloseasca **aceeasi resursa** si ordinea in care este folosita este importanta!
- Pot fi erori extrem de greu de depistat!!!

# Race Conditions & Critical Sections

- A **Critical Section** is a code segment that accesses shared variables.  
Data-race may occur inside a critical section

```
public class Counter {  
    protected long count = 0;  
    public void add(long value){  
        this.count = this.count + value;  
    }  
}
```

Daca un obiect de tip Counter este folosit de 2 sau mai multe threaduri!

=> Nu e *thread-safe*!

- Metoda add() este un exemplu de sectiune critica care conduce la race conditions.

# Counter -> Detaliere la nivel de registrii

- Codul nu este executat ca si o instructiune atomica:

get this.count from memory into register  
add value to register  
write register to memory

- Exemplu de intretesere

this.count = 0;

A: reads this.count into a register (0)

B: reads this.count into a register (0)

B: adds value 2 to register

B: writes register value (2) back to memory. this.count now equals 2

A: adds value 3 to register

A: writes register value (3) back to memory. this.count now equals 3

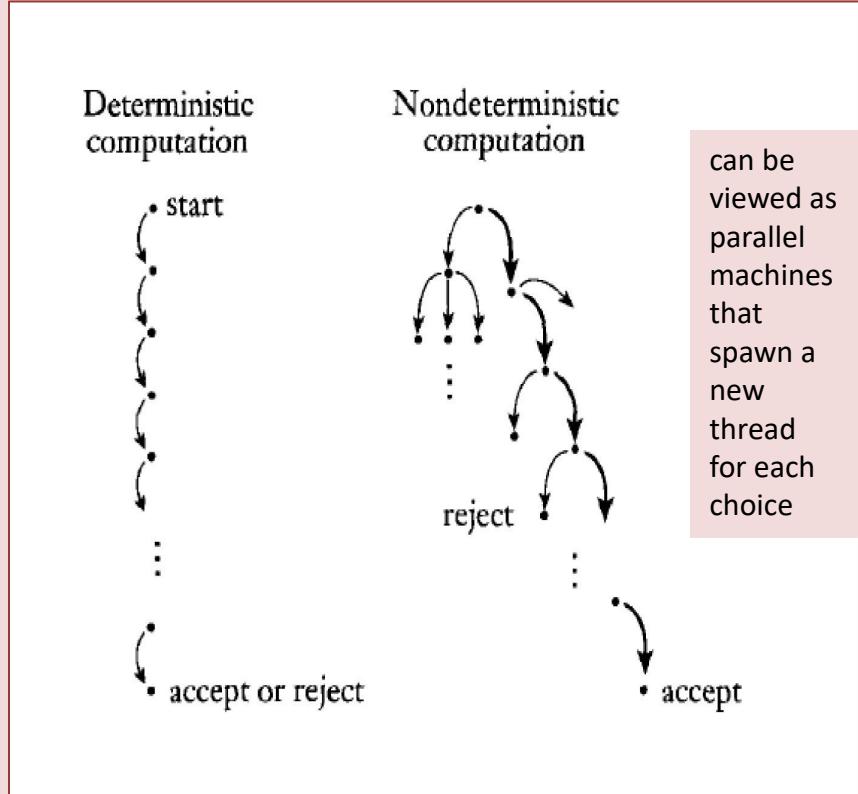
# Solutii

Necesar -> Accesul la date în secțiune critică făcut într-un mod ordonat și atomic, astfel încât rezultatele să fie predictibile

- Soluții:
  - atomicizarea zonei critice
  - dezactivarea preempției în zona critică
  - sevențializarea accesului la zona critică

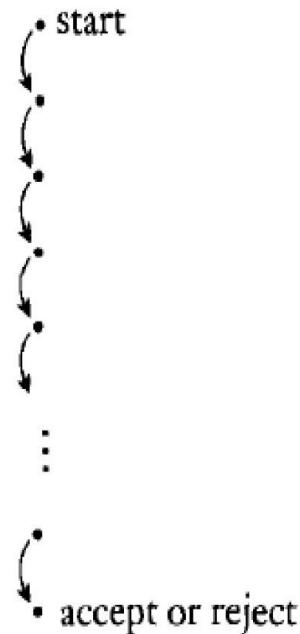
# Deterministic versus non-deterministic computation

- Determinism (Computer Science)
- *A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.*
- Nondeterminism (Computer Science)
- *A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.*

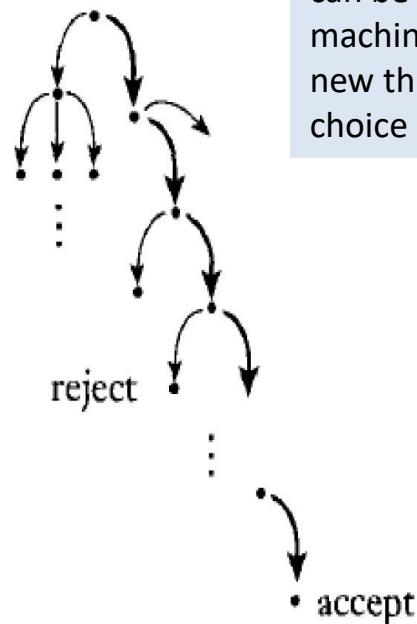


# Deterministic versus non-deterministic computation

Deterministic  
computation



Nondeterministic  
computation

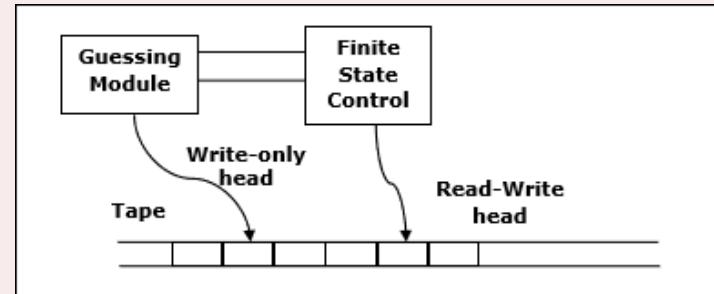
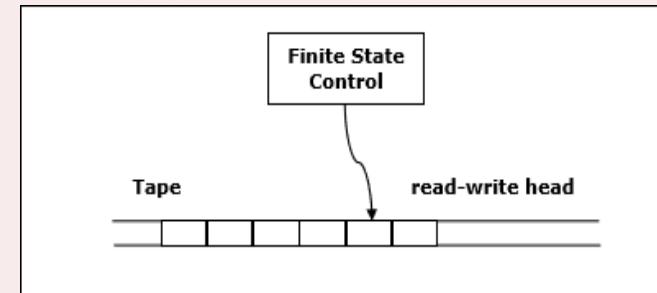


can be viewed as parallel  
machines that spawn a  
new thread for each  
choice

# Turing machines

[https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/design\\_and\\_analysis\\_of\\_algorithms\\_deterministic\\_vs\\_nondeterministic\\_computations.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_deterministic_vs_nondeterministic_computations.htm)

- **Deterministic Turing Machine** - consists of a finite state control, a read-write head and a two-way tape with infinite sequence.
- A program for a deterministic Turing machine specifies the following information –
  - A finite set of tape symbols (input symbols and a blank symbol)
  - A finite set of states
  - A transition function
- In algorithmic analysis,
  - if a problem is solvable in polynomial time by a deterministic one tape Turing machine, the problem belongs to P class.
- **Nondeterministic Turing Machine** -one additional module known as the **guessing module**, which is associated with one write-only head.
  - If the problem is solvable in polynomial time by a non-deterministic Turing machine, the problem belongs to NP class.



# Curs 4

Programare Paralela si Distribuita

Message Passing Interface - MPI

# MPI: Message Passing Interface

- MPI -documentation
  - <http://mpi-forum.org>
- Tutoriale:
  - <https://computing.llnl.gov/tutorials/mpi/>
  - ...

# MPI

- **specificatie de biblioteca(API) pentru programare paralela bazata pe transmitere de mesaje;**
- **propusa ca standard de producatori si utilizatori;**
- **gandita sa ofere performanta mare pe masini paralele dar si pe clustere;**

# Istoric

- Apr 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia=> Preliminary draft proposal
- Nov 1992: Minneapolis. MPI draft proposal (MPI1) from ORNL presented.
- Nov 1993: Supercomputing 93 conference - draft MPI standard presented.
- May 1994: Final version of MPI-1.0 released
- MPI-1.1 (Jun 1995)
- MPI-1.2 (Jul 1997)
- MPI-1.3 (May 2008).
- 1998: MPI-2 picked up where the first MPI specification left off, and addressed topics which went far beyond the MPI-1 specification.
- MPI-2.1 (Sep 2008)
- MPI-2.2 (Sep 2009)
- Sep 2012: The MPI-3.0 standard approved.
- MPI-3.1 (Jun 2015)
- MPI-4

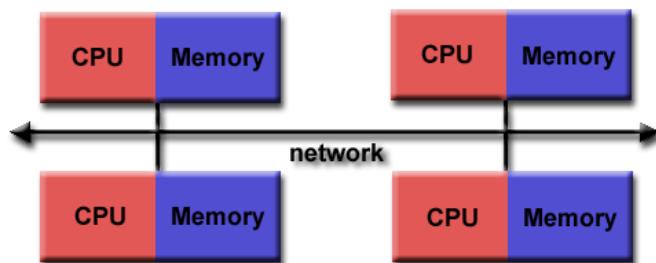
# Implementari

**Exemple:**

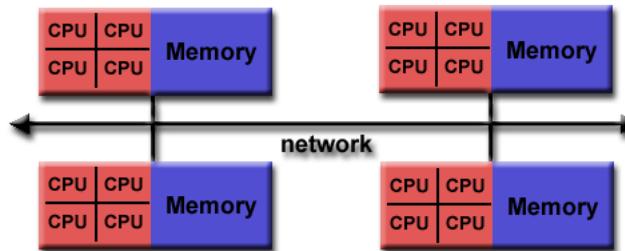
- **MPICH –**
- **Open MPI –**
- **IBM MPI –**
- **IntelMPI (not free)**
- **Links:**  
<http://www.dcs.ed.ac.uk/home/trollius/www.osc.edu/mpi/>

# Modelul de programare

Initial doar pt DM



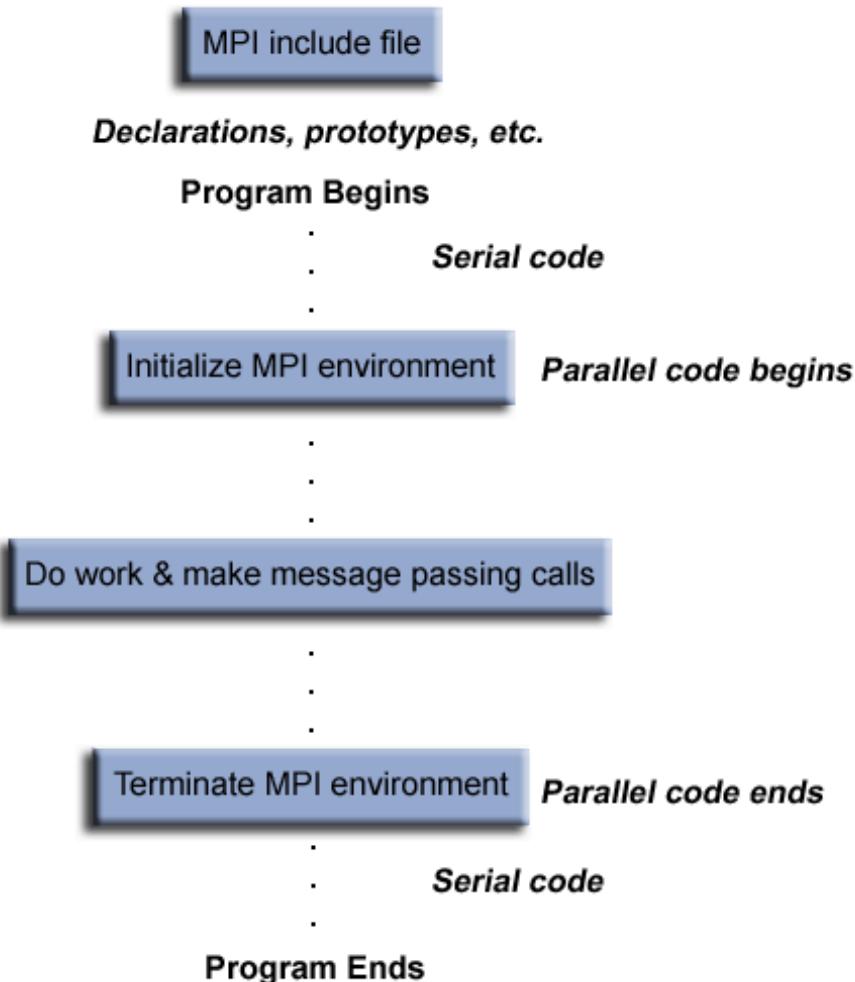
Ulterior si pt SM



Platforme suportate

- Distributed Memory
- Shared Memory
- Hybrid

# Structura program MPI



# Hello World in MPI

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv)
{
    int namelen, myid, numprocs;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf( "Process %d / %d : Hello world\n", myid, numprocs);

    MPI_Finalize();
    return 0;
}
```

compilare  
\$ mpicc hello.c -o hello

executie  
\$ mpirun -np 4 hello

Process 0 / 4 : Hello world  
Process 2 / 4 : Hello world  
Process 1 / 4 : Hello world  
Process 3 / 4 : Hello world

# Formatul functiilor MPI

rc = MPI\_Xxxxx(parameter, ... )

Exemplu:

rc=MPI\_Bsend( &buf, count, type, dest, tag, comm)

Cod de eroare: Intors ca "rc". MPI\_SUCCESS pentru succes

# Comunicatori si grupuri

- MPI foloseste obiecte numite comunicatori si grupuri pentru a defini ce colectii de procese pot comunica intre ele. Cele mai multe functii MPI necesita specificarea unui comunicator ca argument.
- Pentru simplitate exista comunicatorul predefinit care include toate procesele MPI numit MPI\_COMM\_WORLD.

# Rangul unui proces

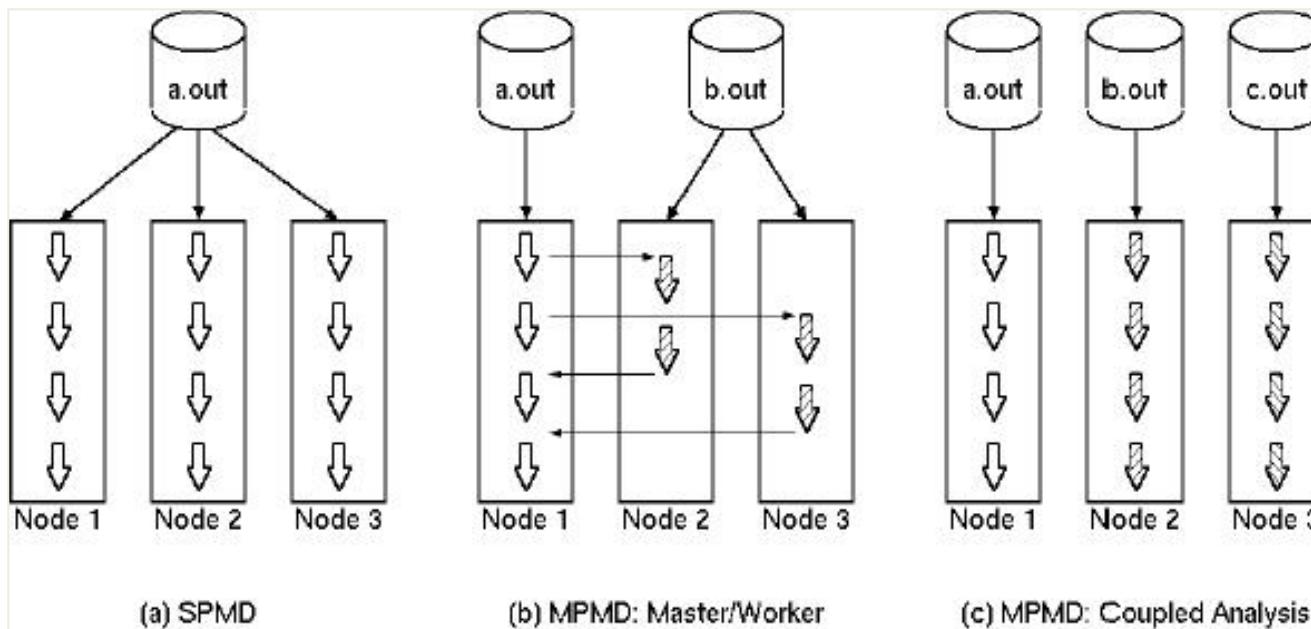
- **Intr-un comunicator, fiecare proces are un identificator unic, rang. El are o valoare intreaga, unica in sistem, atribuita la initializarea mediului.**
- **Utilizat pentru a specifica sursa si destinatia mesajelor.**
- **De asemenea se foloseste pentru a controla executia programului:**  
**e.g: daca rank=0 fa ceva / daca rank=1 fa altceva , etc.**

# SPMD/MPMD

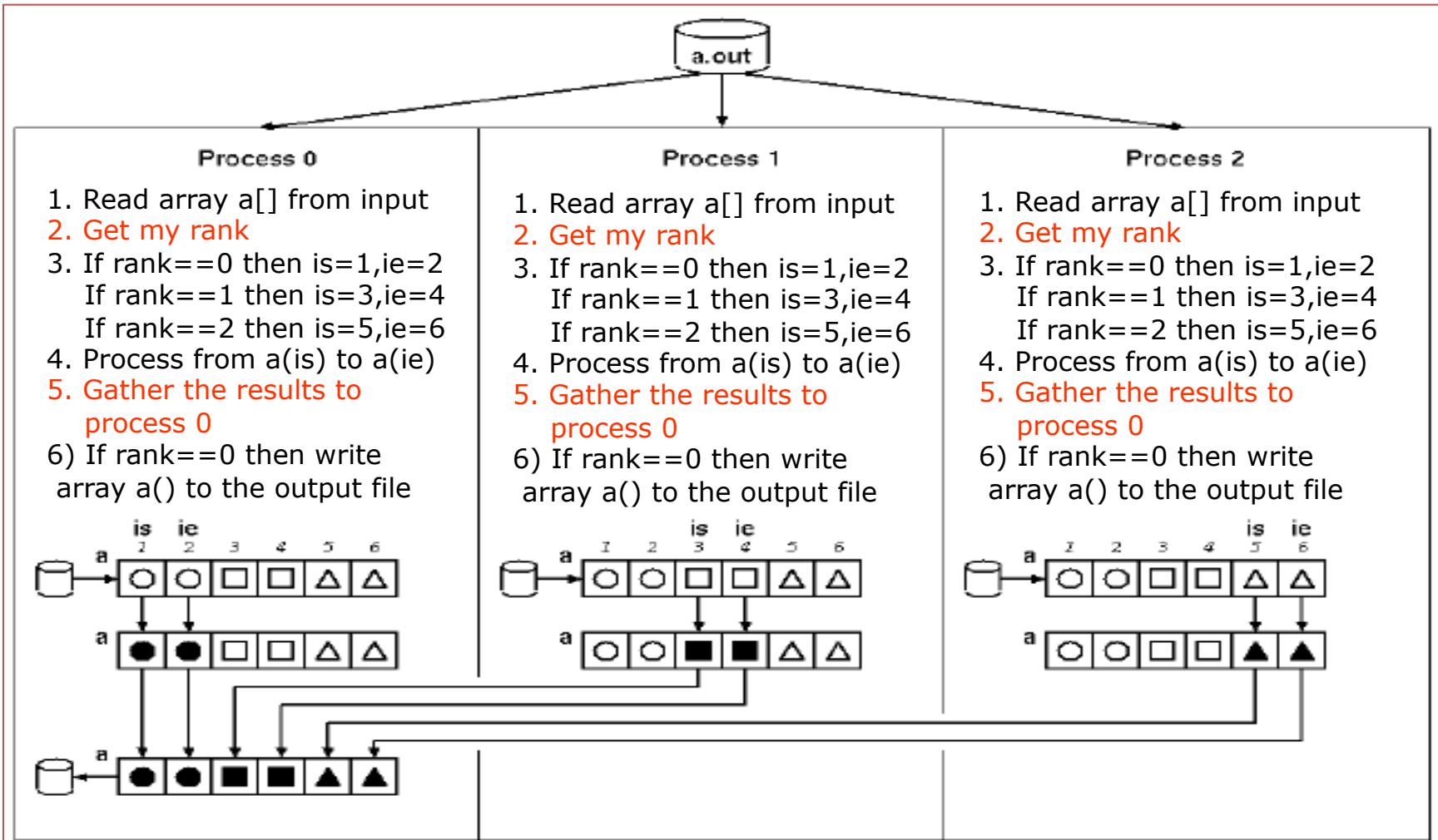
## Modele de calcul paralel in sisteme cu memorie distribuita

SPMD (Single Program Multiple Data) (Fig a)

MPMD (Multiple Program Multiple Data) (Fig b,c)



# Modelul SPMD – Single Program Multiple Data



# MPI. Clase de functii

- *Functii de management mediu*
- *Functii de comunicatie punct-la-punct*
- *Operatii colective*
- *Grupuri de procese/Comunicatori*
- *Topologii (virtuale) de procese*

# Functii de management mediu

- ***initializare, terminare, interogare mediu***

- ***MPI\_Init – initializare mediu***

MPI\_Init (&argc,&argv)  
MPI\_INIT (ierr)

- ***MPI\_Comm\_size – determina numarul de procese din grupul asociat unui com.***

MPI\_Comm\_size (comm,&size)  
MPI\_COMM\_SIZE (comm,size,ierr)

- ***MPI\_Comm\_rank – determina rangul procesului apelant in cadrul unui com.***

MPI\_Comm\_rank (comm,&rank)  
MPI\_COMM\_RANK (comm,rank,ierr)

- ***MPI\_Abort – opreste toate procesele asociate unui comunicator***

MPI\_Abort (comm,errorcode)  
MPI\_ABORT (comm,errorcode,ierr)

- ***MPI\_Finalize -finalizare mediu MPI***

MPI\_Finalize ()  
MPI\_FINALIZE (ierr)

# Exemplu

- **initializare, terminare, interogare mediu**

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[] )
{
int numtasks, rank, rc;
rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
    printf ("Error starting MPI program. Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
}
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);
    ***** do some work *****/
MPI_Finalize();
}
```

# Executie

```
$ mpirun -np 4 test
```

- se creeaza 4 instante de process si fiecare executa acelasi program executabil
- Intre *init* si *finalize* se pot apela functii MPI
  - context in care interactiune dintre procese este posibila prin functiile MPI

# Comunicatie punct-la-punct

Transferul de mesaje intre 2 taskuri MPI distincte intr-un anumit sens.

- *Tipuri de operatii punct-la-punct*

Exista diferite semantici pentru operatiile de *send/receive* :

- Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send
- o rutina *send* poate fi utilizata cu orice alt tip de rutina *receive*
- rutine MPI asociate (*wait,probe*)

# Comunicatie punct-la-punct-

## *Operatii blocante vs ne-blocante*

**send** are 4 moduri de comunicare:

**Standard**

**Buffered**

**Synchronous**

**Ready**

Fiecare poate fi **blocking** or **non-blocking**

**receive** are 2 moduri de comunicare

**blocking**

**non-blocking**

### *Operatii blocante*

O operatie de *send blocanta* va “returna”(se va finaliza) doar atunci cand zona de date folosita pentru trimitere poate fi reutilizata, fara sa afecteze datele primite de destinatar.

### *Operatii ne-blocante*

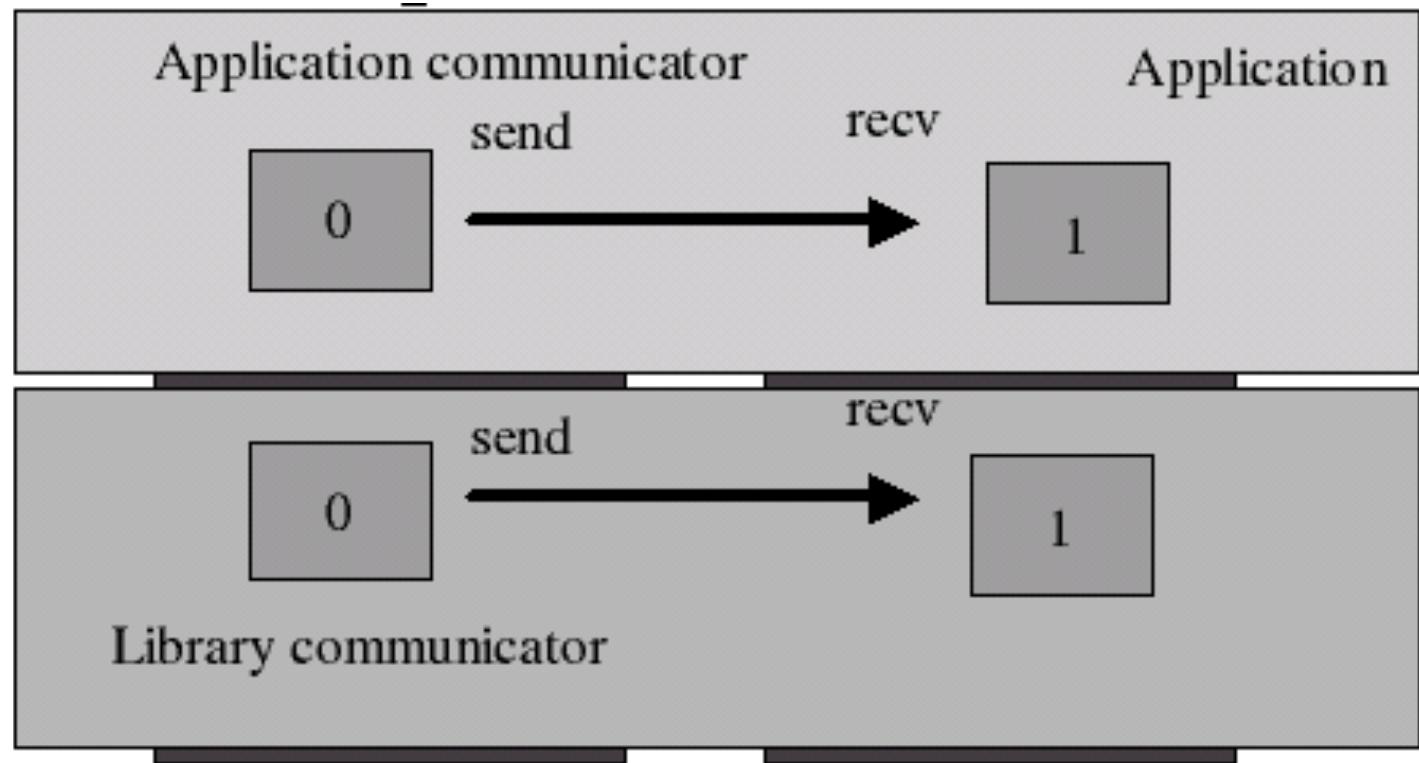
Returneaza controlul imediat, notifica libraria care se va ocupa de transfer. Exista functii speciale de asteptare/interrogare a statusului transferului.

## Comunicatie punct-la-punct

### ***Operatii blocante vs ne-blocante***

Blocking send	<code>MPI_Send(buffer,count,type,dest,tag,comm)</code>
Blocking receive	<code>MPI_Recv(buffer,count,type,source,tag,comm, status)</code>
Blocking Probe	<code>MPI_Probe (source,tag,comm,&amp;status)</code>
Non-blocking send	<code>MPI_Isend(buffer,count,type,dest,tag,comm, request)</code>
Non-blocking receive	<code>MPI_Irecv(buffer,count,type,source,tag,comm, request)</code>
Wait	<code>MPI_Wait (&amp;request,&amp;status)</code>
Test	<code>MPI_Test (&amp;request,&amp;flag,&amp;status)</code>
Non-blocking probe	<code>MPI_Iprobe (source,tag,comm,&amp;flag,&amp;status)</code>

# Comunicatii send-recv



## Determinism si nedeterminism

- Modele de programare paralela bazate pe transmitere de mesaje sunt implicit nedeterministe: ordinea in care mesajele transmise de la doua procese A si B la al treilea C, nu este definita.
  - Este responsabilitatea programatorul de a asigura o executie determinista, daca aceasta se cere.
- In modelul bazat pe transmitere pe canale de comunicatie, determinismul este garantat prin definirea de canale separate pentru comunicatii diferite, si prin asigurarea faptului ca fiecare canal are doar un singur „scriitor” si un singur „cititor”.

# Determinism in MPI

- Pentru obtinerea determinismului in MPI, sistemul trebuie sa adauge anumite informatii datelor pe care programul trebuie sa le trimita. Aceste informatii aditionale formeaza un asa numit "plic" al mesajului.
- In MPI acesta contine urmatoarele informatii:
  - un comunicator.
  - rangul procesului transmitator
  - rangul procesului receptor
  - **un tag (marcaj)**— este un intreg specificat de catre programator, pentru a se putea face distinctie intre mesaje receptionate de la acelasi proces transmitator.
- Comunicatorul stabileste grupul de procese in care se face transmiterea.

# MPI Basic (Blocking) Send

**MPI\_SEND (start, count, datatype, dest, tag, comm)**

- mesajul descris de **(start, count, datatype)**
- **dest** - id process destinatie

## MPI Basic (Blocking) Recv

### **MPI\_Recv (&buf,count,datatype,source,tag,comm,&status)**

- MPI permite omiterea specificarii procesului de la care trebuie sa se primeasca mesajul, caz in care se va folosi constanta predefinita: MPI\_ANY\_SOURCE. (pt send- procesul destinatie trebuie precizat intotdeauna exact.)
  - Marcajul – tagul – mesajului poate fi inlocuit de MPI\_ANY\_TAG, daca se considera ca lipsa lui nu poate duce la ambiguitate.
- Ultimul parametru al functiei MPI\_Recv, **status**, returneaza informatii despre datele care au fost receptionate in fapt. Reprezinta o referinta la o inregistrare cu doua campuri: unul pentru sursa si unul pentru tag. Astfel daca sursa a fost MPI\_ANY\_SOURCE, in status se poate gasi rangul procesului care a trimis de fapt mesajul respectiv.

# MPI Data Types

- **MPI\_CHAR** signed char
- **MPI\_SHORT** signed short int
- **MPI\_INT** signed int
- **MPI\_LONG** signed long int
- **MPI\_LONG\_LONG\_INT**
- **MPI\_LONG\_LONG** signed long long int
- **MPI\_SIGNED\_CHAR** signed char
- **MPI\_UNSIGNED\_CHAR** unsigned char
- **MPI\_UNSIGNED\_SHORT** unsigned short int
- **MPI\_UNSIGNED** unsigned int
- **MPI\_UNSIGNED\_LONG** unsigned long int
- **MPI\_UNSIGNED\_LONG\_LONG** unsigned long long int
- **MPI\_FLOAT** float
- **MPI\_DOUBLE** double
- **MPI\_LONG\_DOUBLE** long double
- ...

# Exemplu operatii blocante

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[]) {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    dest = source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &Stat);
}
}
```

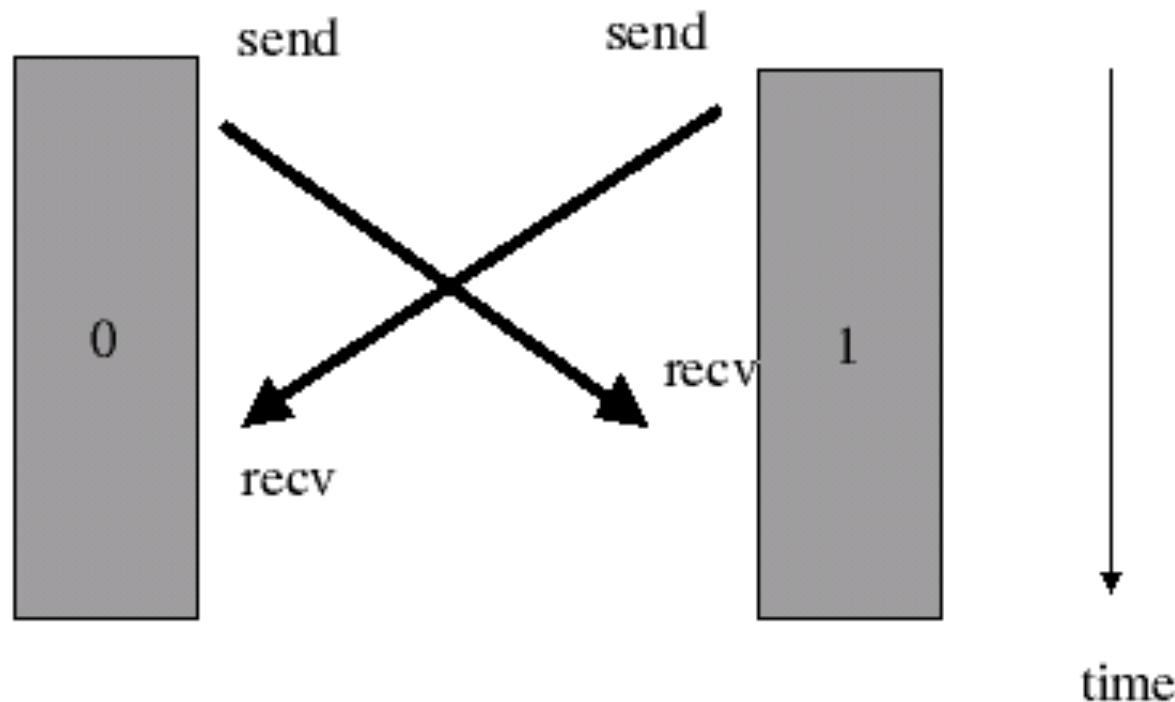
## Exemplu operatii blocante (cont)

```
else if (rank == 1){  
    dest = source = 0;  
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag,  
                  MPI_COMM_WORLD, &Stat);  
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag,  
                  MPI_COMM_WORLD);  
}  
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);  
printf("Task %d: Received %d char(s) from task %d with tag %d  
      \n", rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);  
MPI_Finalize();  
}
```

# MPI deadlocks

- Scenariu:
  - Presupunem ca avem doua procese in cadrul carora comunicatia se face dupa urmatorul protocol
    - Primul proces trimit date catre cel de-al doilea si asteapta raspunsuri de la acesta.
    - Cel de-al doilea proces trimit date catre primul si apoi asteapta raspunsul de la acesta.
  - Daca bufferele sistem nu sunt suficiente se poate ajunge la deadlock. Orice comunicatie care se bazeaza pe bufferele sistem este nesigura din punct de vedere al deadlock-ului.
  - In orice tip de comunicatie care include cicluri pot apare deadlock-uri.

# Deadlock



# **OPERATII COLECTIVE**

# Operatii colective

- *Operatiile colective implica toate procesele din cadrul unui comunicator. Toate procesele sunt membre ale comunicatorului initial, predefinit MPI\_COMM\_WORLD.*

*Tipuri de operatii colective:*

- Sincronizare: procesele asteapta toti membrii grupului sa ajunga in punctul de jonctiune.
- Transfer de date - broadcast, scatter/gather, all to all.
- Calcule colective (reductions) – un membru al grupului colecteaza datele de la toti ceilalti membrii si realizeaza o operatie asupra acestora (min, max, adunare, inmultire, etc.)

**Observatie:**

Toate operatiile colective sunt blocante

## Operatii colective

### ***MPI\_Barrier***

MPI\_Barrier (comm)

MPI\_BARRIER (comm,ierr)

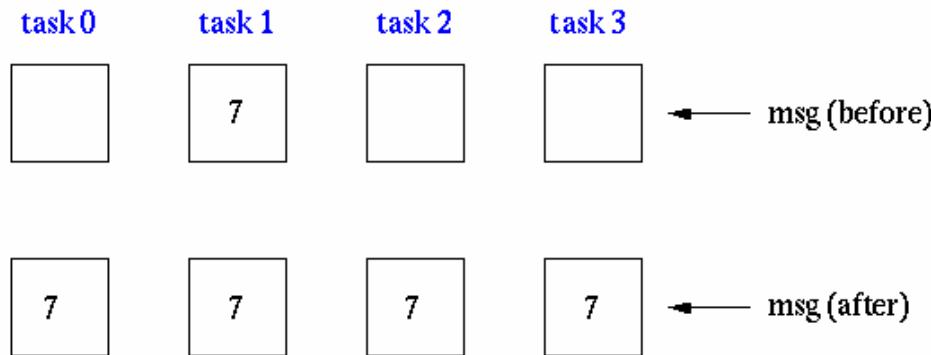
*Fiecare task se va bloca in acest apel pana ce toti membri din grup au ajuns in acest punct*

# Operatii colective

## MPI\_Bcast

Broadcasts a message to all other processes of that group

```
count = 1;  
source = 1;           broadcast originates in task 1  
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

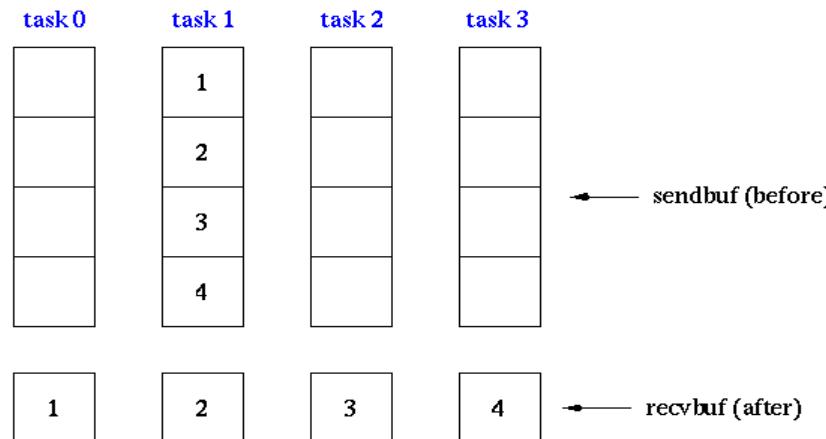


# Operatii colective

## MPI\_Scatter

Sends data from one task to all other tasks in a group

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           task 1 contains the message to be scattered  
MPI_Scatter(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```

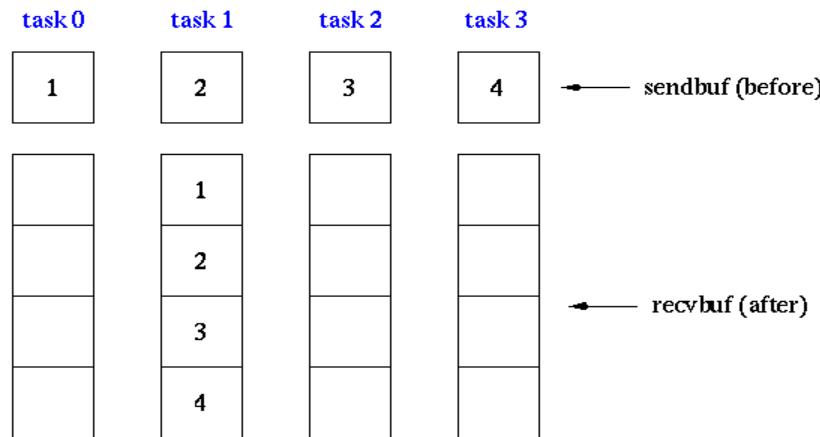


# Operatii colective

## MPI\_Gather

Gathers together values from a group of processes

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;           messages will be gathered in task 1  
MPI_Gather(sendbuf, sendcnt, MPI_INT,  
            recvbuf, recvcnt, MPI_INT,  
            src, MPI_COMM_WORLD);
```

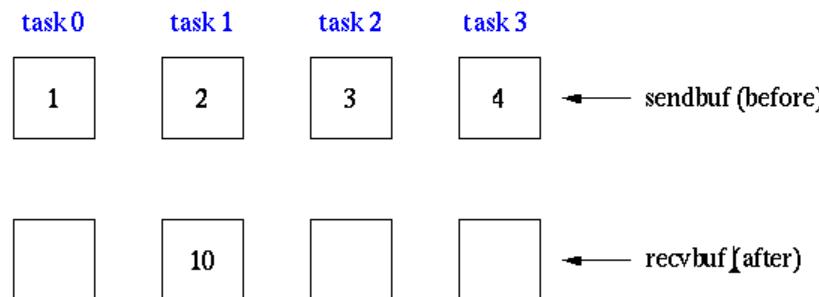


# Operatii colective

## MPI\_Reduce

Perform and associate reduction operation across all tasks in the group and place the result in one task

```
count = 1;  
dest = 1;           result will be placed in task 1  
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,  
           dest, MPI_COMM_WORLD);
```



# Curs 5

## Programare Paralela si Distribuita

Concurrenta

Deadlock, Starvation, Livelock

Semafoare, Mutex, Monitoare, Variabile Conditionale

# Forme de interacțiune între procese/threaduri

## 1. comunicarea între procese distincte

-transmiterea de informații între procese/threaduri

Exemplu: comunicare punct la punct (*sender – receiver*)

## 2. sincronizarea astfel încât procesele să aștepte informațiile de care au

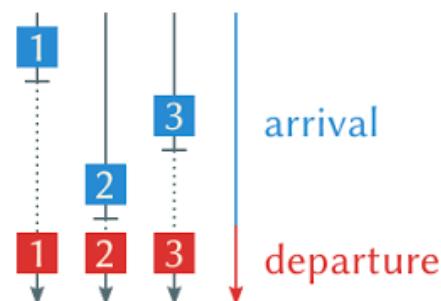
nevoie și nu sunt produse încă de alte procese/thread-uri

- restricții asupra evoluției în timp a unui proces/thread

Exemple:

-excludere mutuală

-bariera de sincronizare



# Correctness

There are two kinds of correctness criteria:

- Partial Correctness
  - If the preconditions hold and the program terminates, then the postconditions will hold.
- Total Correctness
  - If the preconditions hold, then the program will terminate and the postconditions will hold.

For concurrent programs that are supposed to terminate, we prefix these definitions with "**For all possible interleaved execution sequences.**"

- For programs that are not supposed to terminate, we have to write correctness criteria in terms of
    - properties that must *always hold* (**safety properties**) and
    - properties that must *eventually hold* (**liveness properties**).
- Both are important: a program that does nothing is safe !

# *Safety – Fairness - Liveness*

- ***Safety***
  - "nothing bad ever happens"
  - *a program never terminates with a wrong answer*
- ***Fairness***
  - presupune o rezolvare corecta a nedeterminismului in executie
  - Weak fairness
    - daca o actiune este in mod continuu accesibila (*continuously enabled*) (stare-ready) atunci trebuie sa fie executata infinit de des (*infinitely often*).
    - Exemplu => "If a thread continually makes a request it will eventually be granted"
  - Strong fairness
    - daca o actiune este infinit de des accesibilila (*infinitely often enabled*) dar nu obligatoriu in mod continuu atunci trebuie sa fie executata infinit de des (*infinitely often*).
    - exemplu => "If a thread makes a request infinitely often it will eventually be granted."
- ***Liveness***
  - "something good eventually happens" (pana la urma se progreseaza )
  - *a program eventually terminates (pana la urma programul se termina)*

# Forme de sincronizare

- excluderea mutuală: se evită utilizarea simultană de către mai multe procese a unei resurse critice.

O resursă este critică dacă poate utilizarea ei de către mai multe threaduri/procese poate conduce la *race-condition(data race)*. Prin urmare poate fi utilizată doar de către singur process/thread la un moment dat.

*Sectiune critica* – segmentul de cod în care se foloseste o resursă critică.

  - *arbitrare*: se evită accesul simultan din partea mai multor procese/threaduri la aceeași locație de memorie – alegere arbitrara.
  - *blocare*: se realizează o secvențializare a accesului, impunând așteptarea până când procesul/threadul care a obținut accesul și-a încheiat activitatea asupra locației de memorie.
- sincronizarea pe condiție: se amână execuția unui proces până când o anumită condiție devine adevărată;

# Reliability

- What if a thread is interrupted, is suspended, or crashes inside its critical section?
  - In the middle of the critical section, the system may be in an inconsistent state
    - a thread is holding a lock and if it dies no other thread waiting on that lock can proceed!
  - Critical sections must be treated as transactions and must always be allowed to finish.
- **Developers must ensure critical regions are very short and always terminate.**

# *Deadlock – Starvation - Livelock*

- ***Deadlock***
  - situatia in care un grup de procese/threaduri se blocheaza la infinit pentru ca fiecare proces asteapta dupa o resursa care este retinuta de alt proces care la randul lui asteapta dupa alta resursa.
- ***Starvation***
  - Daca unui thread nu i se aloca timp de executie *CPU time* pentru ca alte threaduri folosesc CPU (e.g. min priority)
  - Thread este "*starved to death*" pentru ca alte threaduri au acces la CPU in locul lui.
  - Situatia corecta "*fairness*" toate threadurile au sanse egale la folosire CPU.
- ***Livelock***
  - Situatia in care un grup de procese/threaduri nu progreseaza datorita faptului ca isi cedeaza reciproc executia

# Exemplu - Deadlock

```
public class TreeNode {  
  
    TreeNode parent = null;  
    List<TreeNode> children = new ArrayList<TreeNode>();  
  
    public synchronized void addChild(TreeNode child){  
        if(! this.children.contains(child)) {  
            this.children.add(child);  
            child.setParentOnly(this);  
        }  
    }  
  
    public synchronized void addChildOnly(TreeNode child){  
        if(!this.children.contains(child)){  
            this.children.add(child);  
        }  
    }  
  
    public synchronized void setParent(TreeNode parent){  
        this.parent = parent;  
        parent.addChildOnly(this);  
    }  
  
    public synchronized void setParentOnly(TreeNode parent){  
        this.parent = parent;  
    }  
}
```

Cum/cand ar putea  
apare deadlock?

In ce situatii?

# Mecanisme de sincronizare

- Semafoare
- Variabile conditionale
- Monitoare

Ref.: **Bertrand Meyer. Sebastian Nanz.** *Concepts of Concurrent Computation*

# Semafoare

- Primitiva de sincronizare de nivel inalt (nu cel mai inalt)
- Foarte mult folosita
- Implementarea necesita operatii atomice
- Inventata de E.W. Dijkstra in 1965

Definitie

Semafor (general)=> s este caracterizat de

- O variabila ->  $count = v(s)$  (valoarea semaforului)
- 2 operatii P(s)/down si V(s)/up:

# Operatii atomice

- Operatiile atomice sunt operatii care sunt efectuate ca o singură unitate – indivizibil. Nu se pot efectua parțial – ori complet ori deloc.
- Atomicitatea operațiilor poate fi asigurată la nivel hardware sau software.

Sincronizare de nivel jos (e.g. hardware)

- Operatii CAS –compare and swap
- Atomic variables

# Operatiile semafoarelor

- Gestiunea semafoarelor: prin 2 operații indivizibile
  - P(s) – este apelată de către procese care doresc să acceseze o regiune critică pt a obține acces.
    - Efect: - incercarea obtinerii accesului procesului apelant la secțiunea critică si decrementarea valorii.
    - dacă  $v(s) \leq 0$  , procesul ce dorește execuția sectiunii critice așteaptă
  - V(s)
    - Efect : incrementarea valorii semaforului.
    - se apelează la sfârșitul secțiunii critice și semnifică eliberarea acesteia pt. alte procese.

- Succesiune instrucțiuni:

P(s)

regiune critică

V(s)

Restul procesului

# Semafor

- Cerinte de atomicitate:
  - Testarea valorii contorului (valorii semaforului)
  - Incrementare/decrementarea valorii lui
- Un semafor general se numeste si semafor de numarare  
*(Counting semaphore)*
- Valoarea unui semafor = valoarea *count*

```
class SEMAPHORE feature
  count : INTEGER
  down
    do
      await count > 0
      count := count - 1
    end
  up
    do
      count := count + 1
    end
  end
```

# Semafor Binar

- Valoarea semaforului poate lua doar valorile 0 si 1  
Valoarea => poate fi de tip boolean

```
b : BOOLEAN
down
  do
    await b
    b := false
  end
up
  do
    b := true
  end
```

# Starvation-free

- Daca semaforul se foloseste fara a se mentine o evidenta a proceselor care asteapta intrarea in sectiunea critica nu se poate asigura *starvation-free*
- Pentru a se evita aceasta problema, procesele (referinte catre ele) blocate sunt tinute intr-o **colectie** care are urmatoarele operatii:
  - add(P)
  - Remove (P)
  - is\_empty

# Weak Semaphore

- Un semafor ‘slab’ se poate defini ca o pereche  $\{v(s), c(s)\}$  unde:
  - $v(s)$  este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese.
  - $c(s)$  o **multime de așteptare** la semafor - conține referințe la procesele care așteaptă la semaforul  $s$ .

+

Operatiile  $P(s)/down$  și  $V(s)/up$

# Strong Semaphore

- Un semafor ‘puternic’ se poate defini ca o pereche  $\{v(s), c(s)\}$  unde:
  - $v(s)$  este valoarea semaforului- un nr. întreg a cărui valoare poate varia pe durata execuției diferitelor procese.
  - $c(s)$  o **coadă de aşteptare** la semafor - conține referințe la procesele care aşteaptă la semaforul  $s$  (**FIFO**).

+

## Operatiile P/down și V/up

# Schita de implementare

```
count : INTEGER
blocked: CONTAINER
down
do
  if count > 0 then
    count := count - 1
  else
    blocked.add(P)      -- P is the current process
    P.state := blocked -- block process P
  end
end
up
do
  if blocked.is_empty then
    count := count + 1
  else
    Q := blocked.remove -- select some process Q
    Q.state := ready     -- unblock process Q
  end
end
```

# Analiza

- Invariant:

*count >= 0*

*count = k + #up - #down*

- $k \geq 0$ : valoarea initiala a semaforului
- count: valoarea curenta a semaforului
- #down: nr. de op. down terminate
- #up: nr. de op. up terminate

- Demonstratie

## Apel down:

- if  $count > 0 \Rightarrow \#down$  este incrementat si  $count$  decrementat
- if  $count \leq 0 \Rightarrow down$  nu se termina si  $count$  nu se modifica.

## Apel up:

- if  $blocked (is\_empty) \Rightarrow \#up$  si  $count$  sunt incrementate;
- if  $blocked (not is\_empty) \Rightarrow \#up$  and  $\#down$  sunt incrementate si  $count$  nu se modifica.

- *Starvation*
  - este posibila pt semafoarele de tip *weak semaphores*:  
Pentru ca procesul de selectie este de tip random

# Semafoare Binare

- Count ia doar 2 valori
  - 0->false
  - 1 ->true

=> excludere mutuală

- Mutex -> un semafor binar

# Simulare semafor general prin semafoare binare

```
mutex.count := 1 -- binary semaphore  
delay.count := 1 -- binary semaphore  
count := k
```

- mutex protejeaza citirea si modificarea var count

```
general_down  
do  
    delay.down  
    mutex.down  
    count := count - 1  
    if count > 0 then  
        delay.up  
    end  
    mutex.up  
end
```

```
general_up  
do  
    mutex.down  
    count := count + 1  
    if count = 1 then  
        delay.up  
    end  
    mutex.up  
end
```

Primele k-1 procese  
nu asteapta;  
Urmatoarele DA.

# Varianta de bariera de sincronizare folosind semafoare pentru 2 procese

2 semafoare

<pre>s1.count := 0 s2.count := 0</pre>			
P1		P2	
1	code before the barrier	1	code before the barrier
2	s1.up	2	s2.up
3	s2.down	3	s1.down
4	code after the barrier	4	code after the barrier

s1 furnizeaza bariera pentru P2,  
s2 furnizeaza bariera pentru P1

# Java

## `java.util.concurrent.Semaphore` package

- Constructors:
  - `Semaphore(int k)`, weak semaphore
  - `Semaphore(int k, boolean b)`, strong semaphore if `b=true`
- Operations:
  - `acquire()`, (down)→ throws `InterruptedException`
  - `release()`, (up)

# Dezavantaje - semafoare

- Nu se poate determina utilizarea corecta a unui semafor doar din bucată de cod în care apare; întreg programul trebuie analizat.
- Dacă se poziționează incorect o operatie P sau V atunci se compromite corectitudinea.
- Este usor să se introducă *deadlocks* în program.
- => o variantă mai structurată de nivel mai înalt => Monitor

# Monitor

- Un monitor poate fi considerat un tip abstract de date (poate fi implementat ca și o clasa) care constă din:
  - un set permanent de variabile ce reprezintă resursa critică,
  - un set de proceduri ce reprezintă operații asupra variabilelor și
  - un corp (secvență de instrucțiuni).
    - Corpul este apelat la lansarea ‘programului’ și produce valori inițiale pentru variabilele-monitor (cod de initializare).
    - Apoi monitorul este accesat numai prin procedurile sale.
- codul de inițializare este executat înaintea oricărui conflict asupra datelor ;
- **numai una dintre procedurile monitorului poate fi executată la un moment dat;**
- Monitorul creează o coadă de așteptare a proceselor care fac referire la anumite variabile comune.

# Monitor

- Excluderea mutuală este realizată prin faptul că la **un moment dat poate fi executată doar o singură procedură a monitorului!**
- **Sincronizarea pe condiție este posibila în cadrul unui monitor și se poate realiza prin mijloace definite explicit de către programator prin variabile de tip condiție și două operații:**
  - **signal** (notify)
  - **wait**.
- Dacă un proces care a apelat o procedură de monitor găsește **condiția falsă**, execută operația **wait** (punere în aşteptare a procesului într-un sir asociat condiției și eliberează monitorul).
- în cazul în care alt proces care execută o procedură a aceluiași monitor găsește/seteaza **condiția adevărată**, execută o operație **signal**
  - procesul continuă dacă sirul de aşteptare este vid, altfel este pus în aşteptare și se va executa un alt proces extras din sirul de aşteptare al condiției.

# Monitor – object oriented view

Monitor class :

- toate atributele sunt private
- rutinele/metodele sale se executa prin excludere mutuala;

Instantiere clasa Monitor = monitor

- Attribute <->*shared variables*, (thread-urile le acceseaza doar via monitor)
- Corpurile rutinelor corespund sectiunilor critice – doar o rutina este activa in interiorul monitorului la orice moment).

# Schita Implementare

```
monitor class MONITOR_NAME
```

```
  feature
```

```
    -- attribute declarations
```

```
    a1 : TYPE1
```

```
    ...
```

```
    -- routine declarations
```

```
    r1 (arg1, ..., argk) do ... end
```

```
    ...
```

```
  invariant
```

```
    -- monitor invariant
```

```
end
```

# Implementare folosind un semafor binar: strong semaphore (monitor lock - lacat)

`entry` : SEMAPHORE

$r$  ( $\text{arg}_1, \dots, \text{arg}_k$ )

`do`

`entry.down`

$\text{body}_r$

`entry.up`

`end`

Initializare  $v(\text{entry}) = 1$

# Variabile conditionale in monitoare

## Variabile conditionale

- O abstractizare care permite sincronizarea conditională;
- Variabile conditionale sunt asociate cu lacatul unui monitor (monitor lock);
- Permit threadurilor să aștepte în interiorul unei secțiuni critice eliberând lacatul monitorului.

# Variabile conditionale -> sincronizare conditională

Monitoarele pot oferi variabile conditionale.

**O variabila conditională constă dintr-o coadă de blocare și 3 operații atomice:**

- **wait** eliberează lacatul monitorului, blochează threadul care se executa și îl adaugă în coadă
- **signal** – dacă coada este empty nu are efect;
  - altfel deblochează un thread din coada
- **is\_empty** returnează ->true, dacă coada este empty,
  - > false, altfel.
- Operațiile **wait** și **signal** pot fi apelate doar din corpul unei rutine a monitorului (=> acces sincronizat).

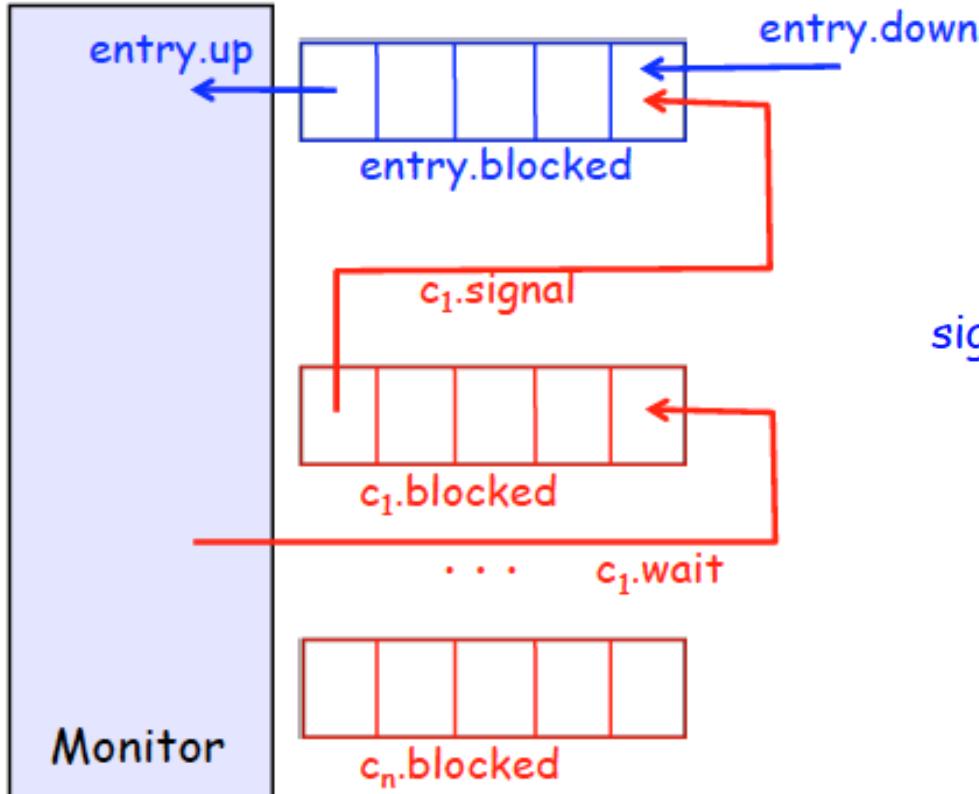
# Schita Implementare pentru variabila conditionala (generală)

```
class CONDITION_VARIABLE
feature
    blocked: QUEUE
    wait
        do
            entry.up          -- release the lock on the monitor
            blocked.add(P)   -- P is the current process
            P.state := blocked -- block process P
        end
    signal deferred end    -- behavior depends on signaling discipline
    is_empty: BOOLEAN
        do
            result := blocked.is_empty
        end
    end
```

# Disciplina de semnalizare (*signal*)

- Atunci cand un proces executa un semnal/*signal* pe o conditie el se executa inca in interiorul monitorului;
- Doar un proces se poate executa in interiorul monitorului => un proces neblocat nu poate intra in monitor imediat
- Doua solutii:
  1. Procesul de semnalizare (**P**) continua si procesul notificat (**Q**) este mutat la intrarea monitorului;
  2. Procesul care semnalizeaza (**P**) lasa monitorul si procesul semnalizat (**Q**) continua.

# Signal & Continue

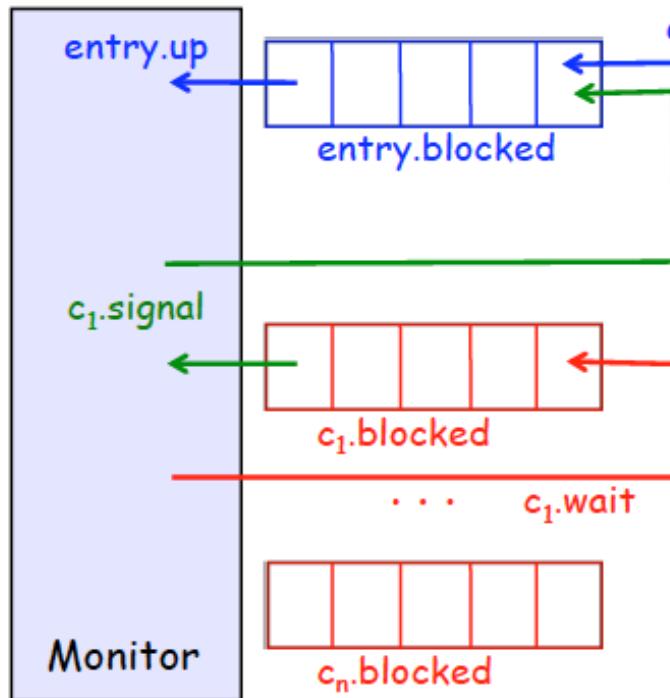


Pentru fiecare  
conditie => o  
coada

```
signal
do
    if not blocked.is_empty then
        Q := blocked.remove
        entry.blocked.add(Q)
    end
end
```

Q se introduce in  
coada semaforului

# Signal & wait



```
signal  
do  
  if not blocked.is_empty then  
    entry.blocked.add(P) -- P is the current process  
    Q := blocked.remove  
    Q.state := ready      -- unblock process Q  
    P.state := blocked    -- block process P  
  end  
end
```

- 'Signal and Continue', -> **signal** este doar un "hint" ca o conditie ar putea fi adevarata – dar alte threaduri ar putea intra si seta conditia la false
- => important ca verificarea conditiei sa se face in **while** nu cu **if !!!**
- Pt. 'Signal and Continue' este si operatia **signal\_all**

```
while not blocked.is_empty do signal end
```

# Alte discipline

- Urgent Signal and Continue: caz special pt ‘Signal and Continue’ prin care thread-ului deblocat prin **signal** i se da o prioritate mai mare in **entry.blocked** (trece in fata)
- Signal and Urgent Wait: caz special pt ‘Signal and Wait’, prin care thread-ului care a semnalizat i se da o prioritate mai mare in **entry.blocked** (trece in fata)

# Monitor in Java

- Fiecare obiect din Java are un monitor care poate fi blocat sau deblocat in blocurile sincronizate:

```
Object lock = new Object();
synchronized (lock) {
    // critical section
}
:
synchronized type m(args) {
    // body
}
• echivalent
type m(args) {
    synchronized (this) {
        // body
    }
}
```

# Monitor in Java

Prin metodele **synchronized** monitoarele pot fi emulate

- nu e monitor original
- variabilele conditionale nu sunt explicit disponibile , dar metodele
  - wait()
  - notify() // signal
  - notifyAll() // signal\_all

pot fi apelate din orice cod **synchronized**

- Disciplina = ‘ Signal and Continue’
- Java "monitors" nu sunt starvation-free – **notify()** deblocheaza un proces arbitrar.

# Avantaje ale folosirii monitoarelor

- Abordare structurata
  - Implica mai putine probleme pt programator pentru a implementa excluderea mutuală;
- *Separation of concerns:*
  - *mutual exclusion for free,*
  - *condition synchronization -> condition variables*

# Probleme

- *trade-off* -> suport pt programator si performanta
- •Disciplinele de semnalizare – sursa de confuzie;
  - *Signal and Continue* – conditia se poate schimba inainte ca procesul semnalizat sa intre in monitor
- *Nested monitor calls*:
  - Doua monitoare M1 si M2  
Rutina r1 din M1 apeleaza rutina r2 din monitorul M2.  
Daca r2 contine o operatie wait atunci excluderea mutuala trebuie relaxata si pentru M1 dar si pentru M2, ori doar pentru M2?

# Variabile conditionale (CV)

## –in general nu doar in interiorul monitoarelor-

- O abstractizare care permite sincronizarea conditională;  
Operatii: **wait**; **signal** ; [broadcast]
- O variabilă conditională **C** este asociată cu
  - o variabilă de tip **Lock – m**
  - o coadă
- Thread **t** apel **wait** =>
  - suspendă **t** și îl adaugă în coadă lui **C** + deblochează **m** (op atomica)
- Atunci cand **t** își reia executia **m** se blochează
- Thread **v** apel **signal** =>
  - se verifică dacă este vreun thread care așteaptă, dacă da alege un thread și activează și deblochează **m**

Legatura cu monitor:

- Variabilele conditionale pot fi asociate cu lacatul unui monitor (monitor lock);
- Permit threadurilor să aștepte în interiorul unei secțiuni critice eliberând lacatul monitorului.

# CV implementare orientativa (Lock implementat ca si un semafor binar initializat cu 1)

```
class CV {  
    Semaphore s, x;  
    Lock m;  
    int waiters = 0;  
public CV(Lock m) {  
    // Constructor  
    this.m = m;  
    s = new Semaphore();  
    s.count = 0;    s.limit = 1;  
    x = new Semaphore();  
    x.count = 1;    x.limit = 1;  
}  
  
// x protejeaza accesul la variabila 'waiters'
```

```
    public void Wait() {  
        // Pre-condition: this thread holds "m"  
        //=> Wait se poate apela doar dintr-un cod  
        // sincronizat (blocat ) cu "m"  
        x.P(); {  
            waiters++; }  
        x.V();  
        m.Release();  
    }  
    s.P();  
    m.Acquire();  
}  
  
public void Signal() {  
    x.P(); {  
        if (waiters > 0)  
        {    waiters--;    s.V();    }  
        x.V();  
    }  
}
```

# Java and C++

## Java

### Interface Condition

#### Methods

##### [await\(\)](#)

The current thread suspends its execution until it is signalled or interrupted.

##### [await\(long time, TimeUnit unit\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.

##### [awaitNanos\(long nanosTimeout\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.

##### [awaitUninterruptibly\(\)](#)

The current thread suspends its execution until it is signalled (cannot be interrupted).

##### [await\(long time, TimeUnit unit\)](#)

The current thread suspends its execution until it is signalled, interrupted, or the specified deadline elapses.

##### [signal\(\)](#)

This method wakes a thread waiting on this condition.

##### [signalAll\(\)](#)

This method wakes all threads waiting on this condition.

## C++

### [std::condition\\_variable](#)

[\(constructor\)](#) `condition_variable(...);`

`condition_variable(const condition_variabl  
e&) = delete;`

#### Notification

[notify\\_one](#) notifies one waiting thread  
([public member function](#))

[notify\\_all](#) notifies all waiting threads  
([public member function](#))

#### Waiting

[wait](#) template< class [Predicate](#) >  
`void wait( std::unique\_lock<std::mutex>& lock, P  
redicate pred );`

template< class Rep, class Period, class [Predicate](#) >  
`bool wait_for( std::unique\_lock<std::mutex>& lock,  
const std::chrono::duration<Rep,Period>& rel_time,  
Predicate pred);`

template< class Clock, class Duration >  
`wait_until( std::unique\_lock<std::mutex>& lock,  
const std::chrono::time\_point<Clock,  
Duration>& timeout_time );`

# C++ example

```
#include <condition_variable>
#include <iostream>
#include <thread>

std::mutex a_mutex;
std::condition_variable condVar;
bool dataReady = false;

void waitingForWork(){
    std::cout << "Waiting\n";
    std::unique_lock<std::mutex> lck(a_mutex);
    condVar.wait( lck, []{ return dataReady; } );
    std::cout << "Running\n";
}
```

```
void setDataReady(){
{
    std::lock_guard<std::mutex> lck(a_mutex);
    dataReady = true;
    std::cout << "Data prepared\n";
    condVar.notify_one();
}
}

int main(){
    std::thread t1(waitingForWork);

    std::thread t2(setDataReady);

    t1.join(); t2.join();
}
```

# Curs 6

## Programare Paralela si Distribuita

Thread Safety

Forme de sincronizare - Java

# *Thread Safety si Shared Resources*

- Codul care poate fi apelat simultan de mai multe threaduri și produce întotdeauna rezultatul dorit/asteptat se numește ***thread safe***.
- Dacă o bucată de cod este *thread safe* atunci nu conține *critical race conditions*.
- În multithreading *Race condition* apare atunci când mai multe threaduri actualizează resurse partajate.
  - care pot fi acestea acestea....?

## *Thread Control Escape Rule*

- Daca o resursa este creata, folosita si eliminata in interiorul controlului aceluiasi thread atunci folosirea acelei resurse este *thread safe*.

# Variabile Locale

- Sunt stocate pe stiva de executie a fiecarui thread.
- Prin urmare nu sunt niciodata partajate.
  - => *thread safe*.

```
public void someMethod(){  
    long threadSafeInt = 0;  
    threadSafeInt++;  
}
```

# Referinte Locale

- Referintele nu sunt partajate (orice obiect este accesibil printr-o referinta).
- Obiectul referit este partajat (*shared heap*).
- Daca un obiect creat local nu se foloseste decat local in metoda care il creeaza atunci este *thread safe*.
- Daca un obiect creat local este transferat altor metode dar nu este transferat altor threaduri atunci este *thread safe*.

Cum se asigura ca nu va fi transferat altor threaduri???

Ex:

```
public void someMethod(){  
    LocalObject localObject = new LocalObject();  
    localObject.callMethod();  
    method2(localObject);  
}  
public void method2(LocalObject localObject){  
    localObject.setValue("value");  
}
```

# Thread-safe class

- ***Thread-safe class***
  - daca comportamentul instantelor sale este corect chiar daca sunt accesate din threaduri multiple - indiferent de executia intreatesuta a lor(interleaving)  
fara sa fie nevoie de sincronizari aditionale sau alte conditii impuse codului apelant.
  - sincronizarile sunt encapsulate in interior si astfel clientii clasei nu trebuie sa foloseasca altele speciale.
- Similar ***Thread-safe code***

# Exemplu: not thread safe

```
public class NotThreadSafe{  
  
    StringBuilder builder =  
        new StringBuilder();  
  
    public void add(String text){  
        this.builder.append(text);  
    }  
  
    public static void main(String[]a){  
  
        NotThreadSafe sharedInstance =  
            new NotThreadSafe();  
  
        new Thread(new  
            MyRunnable(sharedInstance)).start();  
        new Thread(new  
            MyRunnable(sharedInstance)).start();  
  
    }  
}
```

```
public class MyRunnable implements  
    Runnable{  
    NotThreadSafe instance = null;  
  
    public MyRunnable(NotThreadSafe  
        instance){  
        this.instance = instance;  
    }  
  
    public void run(){  
        this.instance.add("text LUNG");  
    }  
}
```

## *Thread-Safe shared variables*

- Daca mai multe thread-uri folosesc o variabila mutabila(modificabila) fara sa foloseasca sincronizari codul ***nu este safe***.
- Solutii:
  - eliminarea partajarii valorii variabilei intre threaduri
  - transformarea variabile in variabila\_imutabila (var imutable sunt thread-safe)
  - sincronizarea accesului la starea variabilei

# Forme de sincronizare Java

# Excludere mutuala

- Fiecare obiect din Java are un *lock/mutex* care poate fi blocat sau deblocat in blocurile sincronizate:
- *Bloc sincronizat*

```
Object critical_object = new Object();
synchronized (critical_object) {
    // critical section
}
```

:> sau *metoda* (obiectul blocat este “this”)

```
synchronized type metoda(args) {
    // body
}
```

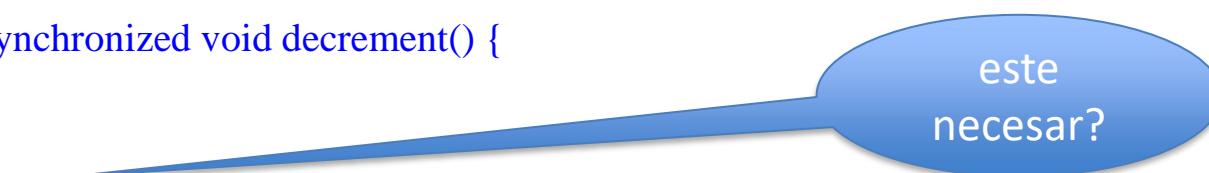
- echivalent

```
type metoda(args) {
    synchronized (this) {
        // body
    }
}
```



# Exemplu

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public void increment() {  
        synchronized (this) {  
            c++;  
        }  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}  
.... SynchronizedCounter co=..new SynchronizedCounter ();  
  
T1 = new MyThread(co).... run(){ co.increment(); }  
T2 = new MyThread(co) .....run(){ co.decrement(); }  
T3 = new MyThread(co) .....run(){ co.value(); }
```



este  
necesar?

# Monitor in Java

Prin metodele **synchronized** monitoarele pot fi emulate

- nu e monitor original
- variabilele conditionale nu sunt explicit disponibile, dar metodele
  - `wait()`
  - `notify() // signal`
  - `notifyAll() // signal_all`

pot fi apelate din orice cod **synchronized**

**≈ variabila conditională implicită**

- Disciplina = ‘ Signal and Continue’
- nu este starvation-free – `notify()` deblocheaza un proces arbitrar.

# Synchronized Static Methods

```
Class Counter{  
    static int count;  
    int x;  
    public synchronized int getX(){ return x; }  
  
    public static synchronized void add(int value){  
        count += value;  
    }  
    public static synchronized void decrease(int value){  
        count -= value;  
    }  
}
```

-> blocare pe *class object of the class* => **Counter.class**

- Ce se intampla daca sunt mai multe metode statice sincronizate ?

# fine-grained synchronization

```
public class CounterC1C2 {  
  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```

- Ce se intampla daca lock1 sau lock2 se modifica?

- Ce se intampla daca sunt metode de tip instanta sincronizate dar si metode statice sincronizate?

# Counter => fine-grained synchronization

```
public class Counter {  
    private long c = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc() {  
        synchronized(lock1) {  
            c++;  
        }  
    }  
    public void dec() {  
        synchronized(lock2) {  
            c--;  
        }  
    }  
}
```

• Este corect?

• Ce probleme exista?

# Nonblocking Counter – Atomic variables

```
public class NonblockingCounter {  
    private AtomicInteger value;  
  
    public int getValue() {  
        return value.get();  
    }  
  
    public int increment() {  
        int v;  
        do {  
            v = value.get();  
        } while (!value.compareAndSet(v, v + 1));  
        return v + 1;  
    }  
}
```

# Operatii atomice

- Operații cu întregi:
  - incrementare, decrementare, adunare, scădere
  - **compare-and-swap (CAS)** operatii

CAS – instructiune *atomica* care compara continutul memoriei cu o valoare data si doar daca aceastea sunt egale modifica continutul locatiei de memorie cu noua valoare data.

*The value of CAS is that it is **implemented in hardware** and is extremely lightweight (on most processors).*

## Compare and swap (CAS)

<http://www.ibm.com/developerworks/library/j-jtp11234/>

- Intel...The first processors that supported concurrency provided atomic test-and-set operations, which generally operated on a single bit.
- The most common approach taken by current processors, including Intel and Sparc processors, is to implement a primitive called *compare-and-swap*, or CAS.
- On Intel processors, compare-and-swap is implemented by the cmpxchg family of instructions.
- PowerPC processors have a pair of instructions called "load and reserve" and "store conditional" that accomplish the same goal; similar for MIPS, except the first is called "load linked."

# CAS operation

- A CAS operation includes three operands
  - a memory location (V),
  - the expected old value (A), and
  - a new value (B).
- The processor will atomically update the location to the new value( $V==B$ ) if the value that is there matches the expected old value ( $V==A$ ), otherwise it will do nothing.
- In either case, it returns the value that was at that location prior to the CAS instruction.

# CAS synchronization

- The natural way to use CAS for synchronization is to read a value A from an address V, perform a multistep computation to derive a new value B, and then use CAS to change the value of V from A to B.
  - The CAS succeeds if the value at V has not been changed in the meantime.
- Instructions like CAS allow an algorithm to execute a read-modify-write sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation.

# java.util.concurrent.atomic

## Class Summary

Class	Description
<code>AtomicBoolean</code>	A <code>boolean</code> value that may be updated atomically.
<code>AtomicInteger</code>	An <code>int</code> value that may be updated atomically.
<code>AtomicIntegerArray</code>	An <code>int</code> array in which elements may be updated atomically.
<code>AtomicIntegerFieldUpdater&lt;T&gt;</code>	A reflection-based utility that enables atomic updates to designated <code>volatile int</code> fields of designated classes.
<code>AtomicLong</code>	A <code>long</code> value that may be updated atomically.
<code>AtomicLongArray</code>	A <code>long</code> array in which elements may be updated atomically.
<code>AtomicLongFieldUpdater&lt;T&gt;</code>	A reflection-based utility that enables atomic updates to designated <code>volatile long</code> fields of designated classes.
<code>AtomicMarkableReference&lt;V&gt;</code>	An <code>AtomicMarkableReference</code> maintains an object reference along with a mark bit, that can be updated atomically.
<code>AtomicReference&lt;V&gt;</code>	An object reference that may be updated atomically.
<code>AtomicReferenceArray&lt;E&gt;</code>	An array of object references in which elements may be updated atomically.
<code>AtomicReferenceFieldUpdater&lt;T,V&gt;</code>	A reflection-based utility that enables atomic updates to designated <code>volatile</code> reference fields of designated classes.
<code>AtomicStampedReference&lt;V&gt;</code>	An <code>AtomicStampedReference</code> maintains an object reference along with an integer "stamp", that can be updated atomically.

A small toolkit of classes that support lock-free  
thread-safe programming on single variables.

# AtomicInteger

java.lang.Object

java.lang.Number

java.util.concurrent.atomic.AtomicInteger

int	<b>addAndGet(int delta)</b>
boolean	<b>compareAndSet(int expect, int update)</b>
int	<b>decrementAndGet()</b>
double	<b>doubleValue()</b>
float	<b>floatValue()</b>
int	<b>get()</b>
int	<b>getAndAdd(int delta)</b>
int	<b>getAndDecrement()</b>
int	<b>getAndIncrement()</b>
int	<b>getAndSet(int newValue)</b>
int	<b>incrementAndGet()</b>
int	<b>intValue()</b>
void	<b>lazySet(int newValue)</b>
long	<b>longValue()</b>
void	<b>set(int newValue)</b>
<b>String</b>	<b>toString()</b>
boolean	<b>weakCompareAndSet(int expect, int update)</b>

# Exemplu

```
class Sequencer {  
  
    private final AtomicLong sequenceNumber = new AtomicLong(0);  
  
    public long next() {  
        return sequenceNumber.getAndIncrement();  
    }  
}
```

# *Thread Signaling*

- Permite transmiterea de semnale/mesaje de la unul thread la altul.
- Un thread poate astepta un semnal de la altul.
- Asteptare conditionata

# *Signaling via Shared Objects*

- Setarea unei variabile partajate- *comunicare prin variabile partajate.*

```
public class MySignal{  
  
    protected boolean hasDataToProcess = false;  
  
    public synchronized boolean hasDataToProcess(){  
        return this.hasDataToProcess;  
    }  
  
    public synchronized  
    void setHasDataToProcess(boolean hasData){  
        this.hasDataToProcess = hasData;  
    }  
}
```

# Busy Wait - INCORECT

- Thread B asteapta ca data sa devina disponibila pentru a o procesa.

- => asteapta un semnal de la threadul A

=> `hasDataToProcess()` to return `true`.

- **Busy waiting NU implica o utilizare eficienta a CPU (cu exceptia situatiei in care timpul mediu de asteptare este foarte mic).**

- este de dorit ca asteptarea sa fie inactiva (fara folosire procesor) –

- poate sa produca blocaj!

```
protected MySignal sharedSignal = ...
```

```
...
```

```
while( ! sharedSignal.hasDataToProcess()){
    //do nothing... busy waiting ;
}
```

# Operatii

- `wait()`
- `notify() // signal`
- `notifyAll() // signal_all`

## Exemplul 1 → Producator- Consumator / Buffer de dimensiune = 1

```
public class Producer extends Thread {  
  
    ... ITER  
  
    private Location loc;  
  
    private int number; //id  
  
    public Producer(Location c, int number) {  
        loc = c;  
        this.number = number;  
    }  
  
    public void run() {  
        for (int i = 0; i < ITER; i++) {  
            loc.put(i);  
        }  
    }  
}
```

```
public class Consumer extends Thread {  
  
    ... ITER  
  
    private Location loc;  
  
    private int number; //id  
  
    public Consumer(Location c, int number) {  
        loc = c;  
        this.number = number;  
    }  
  
    public void run() {  
        int value = 0;  
        for (int i = 0; i < ITER; i++) {  
            value = loc.get();  
        }  
    }  
}
```

```

public class Location {
    private int contents;          // shared data : didactic
    private boolean available = false;

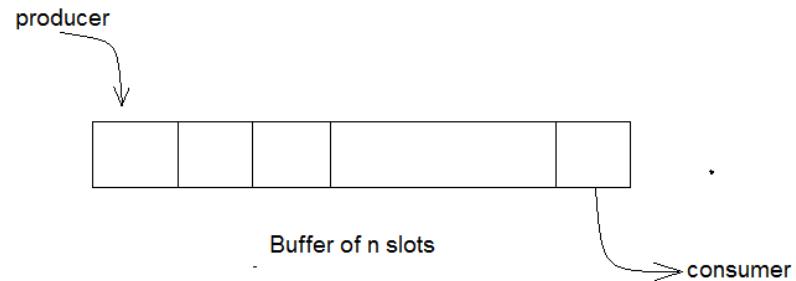
/* Method used by the consumer to access the shared data */
public synchronized int get() {
    while (available == false) {
        try {
            this.wait();           // Consumer enters a wait state until notified by the Producer
        } catch (InterruptedException e) { }
    }
    available = false;
    this.notifyAll();             // Consumer notifies Producers that it can store new contents
    return contents;
}

/* Method used by the consumer to store the shared data */
public synchronized void put (int value) {
    while (available == true) {
        try {
            this.wait();           // Producer who wants to store contents enters
                                // a wait state until notified by the Consumer
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    this.notifyAll();             // Producer notifies Consumer to come out
                                // of the wait state and consume the contents
}
}

```

## Exemplul 2: BlockingQueue : buffer size >1

```
class BlockingQueue {  
    int n = 0;  
    Queue data = ...;  
  
    public synchronized Object remove() {  
        // wait until there is something to read  
        while (n==0)  
            this.wait();  
  
        n--;  
        // return data element from queue  
    }  
  
    public synchronized void write(Object o) {  
        n++;  
        // add data to queue (considere that there is unlimited space)  
  
        notifyAll();  
    }  
}
```



## Missed Signals- Starvation

- Apelurile metodelor `notify()` si `notifyAll()` nu se salveaza in cazul in care nici un thread nu asteapta atunci cand sunt apelate.
- Astfel semnalul `notify` se poate pierde.
- Acest lucru poate conduce la situatii in care un thread asteapta needefinit, pentru ca mesajul corespunzator de notificare s-a pierdut anterior.

- Propunere:
  - Evitarea problemei prin salvarea semnalelor in interiorul clasei care le trimit.
- =>analiza!

```

public class MyWaitNotify2{

    MonitorObject myMonitorObject = new MonitorObject();
    boolean wasSignalled = false;

    public void doWait(){
        synchronized(myMonitorObject){
            if(!wasSignalled){
                try{
                    myMonitorObject.wait();
                } catch(InterruptedException e){...}
            }
            //clear signal and continue running.
            wasSignalled = false;
        }
    }

    public void doNotify(){
        synchronized(myMonitorObject){
            wasSignalled = true;
            myMonitorObject.notify();
        }
    }
}

```

# Lock

Oracle docs:

## public interface Lock

- Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.
- **They allow more flexible structuring**, may have quite different properties, and may support **multiple associated Condition objects**.

Modifier and Type	Method and Description
void	<a href="#">lock()</a> Acquires the lock.
void	<a href="#">lockInterruptibly()</a> Acquires the lock unless the current thread is <a href="#">interrupted</a> .
<a href="#">Condition</a>	<a href="#">newCondition()</a> Returns a new <a href="#">Condition</a> instance that is bound to this Lock instance.
boolean	<a href="#">tryLock()</a> Acquires the lock only if it is free at the time of invocation.
boolean	<a href="#">tryLock</a> (long time, <a href="#">TimeUnit</a> unit)Acquires the lock if it is free within the given waiting time and the current thread has not been <a href="#">interrupted</a> .
void	<a href="#">unlock()</a> Releases the lock.

## Lock (java.util.concurrent.locks.Lock)

```
public class Counter{  
  
    private int count = 0;  
  
    public int inc(){  
        synchronized(this){  
            return ++count;  
        }  
    }  
}
```

```
public class Counter{  
    private  
    Lock lock = new ReentrantLock();  
    private int count = 0;  
  
    public int inc(){  
        lock.lock();  
        try{  
            int newCount = ++count; }  
        finally{  
            lock.unlock(); }  
        return newCount;  
    }  
}
```

# Metode ale interfetei Lock

lock()

lockInterruptibly()

tryLock()

tryLock(long timeout, TimeUnit timeUnit)

unlock()

The `lockInterruptibly()` method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

# Diferente Lock vs synchronized

- Nu se poate trimite un parametru la intrarea într-un bloc synchronized => nu se poate preciza o valoare timp corespunzătoare unui interval maxim de așteptare-> timeout.
- Un bloc synchronized trebuie să fie complet continut în interiorul unei metode
  - lock() și unlock() pot fi apelate în metode separate.

# Lock Reentrance

- Blocurile sincronizate in Java au proprietatea de a permite ‘reintrarea’ (*reentrant Lock*).
- Daca un thread intra intr-un bloc sincronizat si blocheaza astfel monitorul obiectului corespunzator, atunci threadul poate intra in alt cod sincronizat prin monitorul aceluiasi obiect.

```
public class Reentrant{
    public synchronized outer(){
        inner();
    }
    public synchronized inner(){
        //do something
    }
}
```

# Conditions in Java

- `java.util.concurrent.locks`
- Interface Condition
- Avantaj fata de “`wait-notify`” din monitorul definit pentru `Object`
- Imparte metodele (`wait`, `notify` , `notifyAll`) in obiecte distincte pentru diferite conditii
  - permite mai multe *wait-sets per object.*

# Exemplu – Prod-Cons FIFO Buffer

```
class BoundedBuffer {  
    static final MAX = 100;  
    final Lock lock = new ReentrantLock();  
    final Condition notFull = lock.newCondition();  
    final Condition notEmpty = lock.newCondition();  
  
    final Object[] items = new Object[MAX];  
    int putptr, takeptr, count;  
  
    public void put(Object x) throws InterruptedException  
    {  
        lock.lock();  
        try {  
            while (count == items.length)  
                notFull.await();  
            items[putptr] = x;  
            if (++putptr == items.length) putptr = 0;  
            ++count;  
            notEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}  
  
public Object take() throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == 0)  
            notEmpty.await();  
        Object x = items[takeptr];  
        if (++takeptr == items.length) takeptr = 0;  
        --count;  
        notFull.signal();  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```

# Semaphore

(java.util.concurrent.Semaphore)

- Semafor binar (=> excludere mutuală)

```
Semaphore semaphore = new Semaphore(1);
```

```
//critical section  
semaphore.acquire();  
...  
semaphore.release();
```

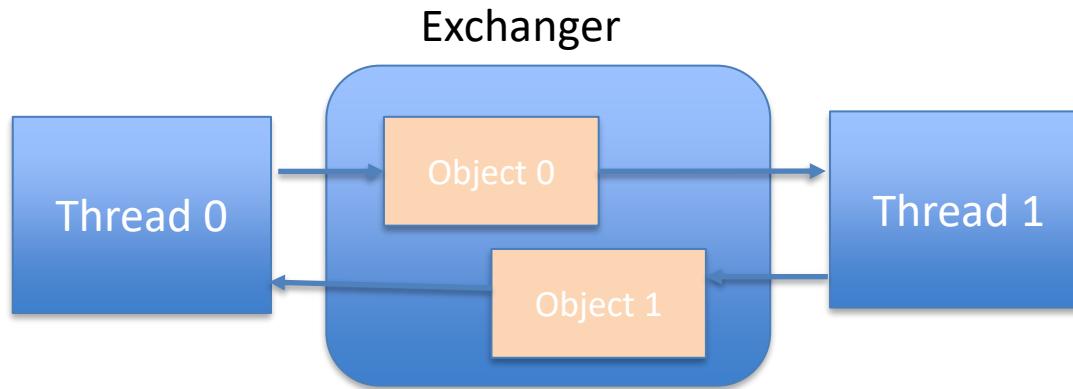
- Fair/Strong Semaphore

```
Semaphore semaphore = new Semaphore(1, true);
```

# Conceptul de întâlnire (Rendez-vous)

- Conceptul de întâlnire (rendez-vous) a fost introdus initial în limbajul Ada pentru a facilita comunicarea între două task-uri.
  - a) Procesul B este gata să transmită informațiile, dar procesul A nu le-a cerut încă. În acest caz, procesul B rămâne în aşteptare până când procesul A îl cere.
  - b) Procesul B este gata să transmită informațiile cerute, iar procesul A cere aceste date. În acest caz, se realizează un *rendez-vous*, cele două procese lucrează sincron până când își termină schimbul, după care fiecare își continuă activitatea independent.
  - c) Procesul A a lansat o cerere, dar procesul B nu este în măsură să-i furnizeze informațiile solicitate. În acest caz, A rămâne în aşteptare până la întâlnirea cu B.

# Java Exchanger <-> Rendez-Vous



Jacob Jenkov Tutorial

Thread-0 exchanged Object0 for Object1

Thread-1 exchanged Object1 for Object0

# Exchanger (java doc)

public V exchange(V x) throws InterruptedException

- Waits for another thread to arrive at this exchange point (unless the current thread is interrupted), and then transfers the given object to it, receiving its object in return.
- If another thread is already waiting at the exchange point then it is resumed for thread scheduling purposes and receives the object passed in by the current thread.
  - The current thread returns immediately, receiving the object passed to the exchange by that other thread.

If no other thread is already waiting at the exchange then the current thread is disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread enters the exchange; or
- Some other thread interrupts the current thread.

If the current thread:

- has its interrupted status set on entry to this method; or
- is interrupted while waiting for the exchange,
  - then InterruptedException is thrown and the current thread's interrupted status is cleared.

```
Exchanger exchanger = new Exchanger();
```

```
ExchangerRunnable exchangerRunnable1 = new ExchangerRunnable(exchanger, "A");
```

```
ExchangerRunnable exchangerRunnable2 = new ExchangerRunnable(exchanger, "B");
```

```
new Thread(exchangerRunnable1).start();
new Thread(exchangerRunnable2).start();
```

```
public class ExchangerRunnable implements Runnable{
    Exchanger exchanger = null;
    Object object = null;

    public ExchangerRunnable(Exchanger exchanger, Object object) {
        this.exchanger = exchanger;
        this.object = object;
    }

    public void run() {
        try {
            Object previous = this.object;

            Object received =
                this.exchanger.exchange (this.object);
            this.object = received;

            System.out.println(
                Thread.currentThread().getName() +
                " exchanged " + previous + " for " + this.object
            );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Class SynchronousQueue (->Rendez-vous)

## Java doc

[java.lang.Object](#)  
[java.util.AbstractCollection<E>](#)  
[java.util.AbstractQueue<E>](#)  
java.util.concurrent.SynchronousQueue<E>

A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. *A synchronous queue does not have any internal capacity, not even a capacity of one.* You cannot peek at a synchronous queue because an element is only present when you try to remove it; *you cannot insert an element (using any method) unless another thread is trying to remove it*; you cannot iterate as there is nothing to iterate. The *head* of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements  
[<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html>]

- boolean [\*\*offer\(E e\)\*\*](#)
  - Inserts the specified element into this queue, if another thread is waiting to receive it.
- void [\*\*put\(E o\)\*\*](#)
  - Adds the specified element to this queue, waiting if necessary for another thread to receive it.
- [\*\*E poll\(\)\*\*](#)
  - Retrieves and removes the head of this queue, if another thread is currently making an element available.

# ReadWriteLock

- Read Access      -> daca nici un alt thread nu scrie si nici nu cere acces pt scriere.
- Write Access      -> daca nici alt un thread nici nu scrie nici nu citeste.

public interface **ReadWriteLock**

- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
- The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.
- The write lock is exclusive.

# Exemplu

```
public class TSArrayList<E>
{
    private final ReadWriteLock readWriteLock =
        new ReentrantReadWriteLock();

    private final Lock readLock = readWriteLock.readLock();
    private final Lock writeLock = readWriteLock.writeLock();

    private final List<E> list = new ArrayList<>();

    public static void main(String[] args)
    {
        TSArrayList<String> threadSafeArrayList =
            new TSArrayList<>();
        threadSafeArrayList.set("1");
        threadSafeArrayList.set("2");
        threadSafeArrayList.set("3");

        System.out.println(threadSafeArrayList.get(1));
    }
}
```

```
public void set(E o)
{
    writeLock.lock();
    try
    {
        list.add(o);
    }
    finally
    {
        writeLock.unlock();
    }
}

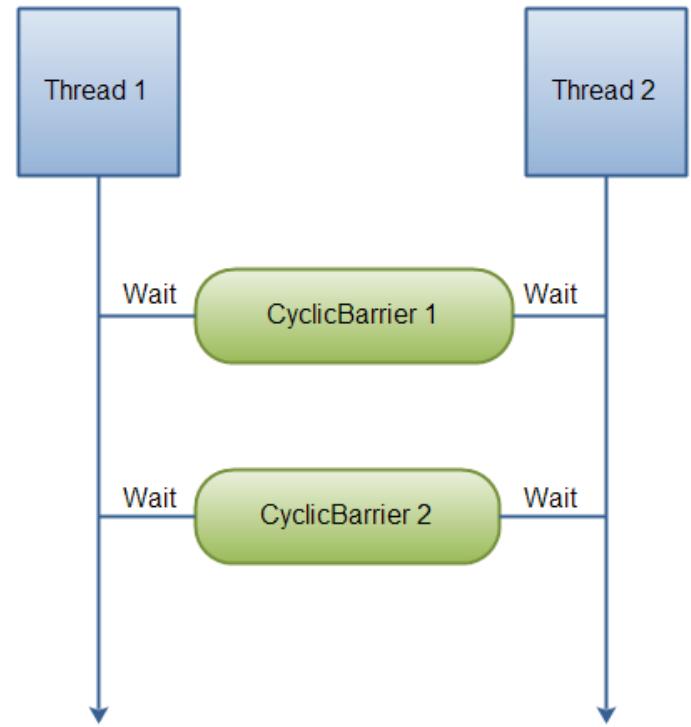
public E get(int i)
{
    readLock.lock();
    try
    {
        return list.get(i);
    }
    finally
    {
        readLock.unlock();
    }
}
```

# Bariera de sincronizare

```
CyclicBarrier barrier = new CyclicBarrier(2);  
// 2 = no_of_threads_to_wait_for  
  
barrier.await();  
  
barrier.await(10, TimeUnit.SECONDS);
```

## Bariera de sincronizare:

- Bariera secentiala – pt implementare se fol. in general 2 variabile – {no\_threads(0..n), state(stop/pass)}
- Bariera ierarhica (tree-barrier)



Jacob Jenkov Tutorial

## java.util.concurrent>

### java.util.concurrent.CyclicBarrier (java documentation)

- A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other.
- The barrier is called cyclic because it can be re-used after the waiting threads are released.
- A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.
  - [CyclicBarrier\(int parties\)](#)
  - [CyclicBarrier\(int parties, Runnable barrierAction\)](#)

# Curs 7

Taskuri

Future-Promise

Executie asincrona

Executori

Apeluri asincrone

Future

Promise

# Istoric

- Termenul *promise* a fost propus de catre Daniel P. Friedman si David Wise in 1976;
- ~ aceeasi perioada Peter Hibbard l-a denumit *eventual*;
- conceptul *future* a fost introdus in 1977 intr-un articol scris de catre Henry Baker si Carl Hewitt .
- *Future* si *promise* isi au originea in programarea functionala si paradigmile conexe (progr. logica)
- Scop: **decuplarea unei valori (*a future*) de ceea ce o calculeaza**
  - Permite calcul flexibil si paralelizabil
- Folosirea in programarea Paralela si distribuita a aparut ulterior mai intai pentru
  - reducerea latentei de comunicatie (*round trips*).  
apoi
  - in programele asincrone.

# Promise pipelining

Barbara Liskov and Liuba Shrira in 1988

Mark S. Miller, Dean Tribble and Rob Jellinghaus 1989

Conventional RPC

```
t3 := ( x.a() ).c( y.b() )
```

Echivalent cu

```
t1 := x.a();
```

```
t2 := y.b();
```

```
t3 := t1.c(t2); //executie dupa ce t1 si t2 se termina
```

Daca folosim apel *remote* atunci este nevoie de 3 round-trip.

(a,b,c se executa remote)

Folosind futures

("Dataflow" with Promises)

```
t3 := (x <- a()) <- c(y <- b())
```

Echivalent cu

```
t1 := x <- a()
```

```
t2 := y <- b()
```

```
t3 := t1 <- c(t2)
```

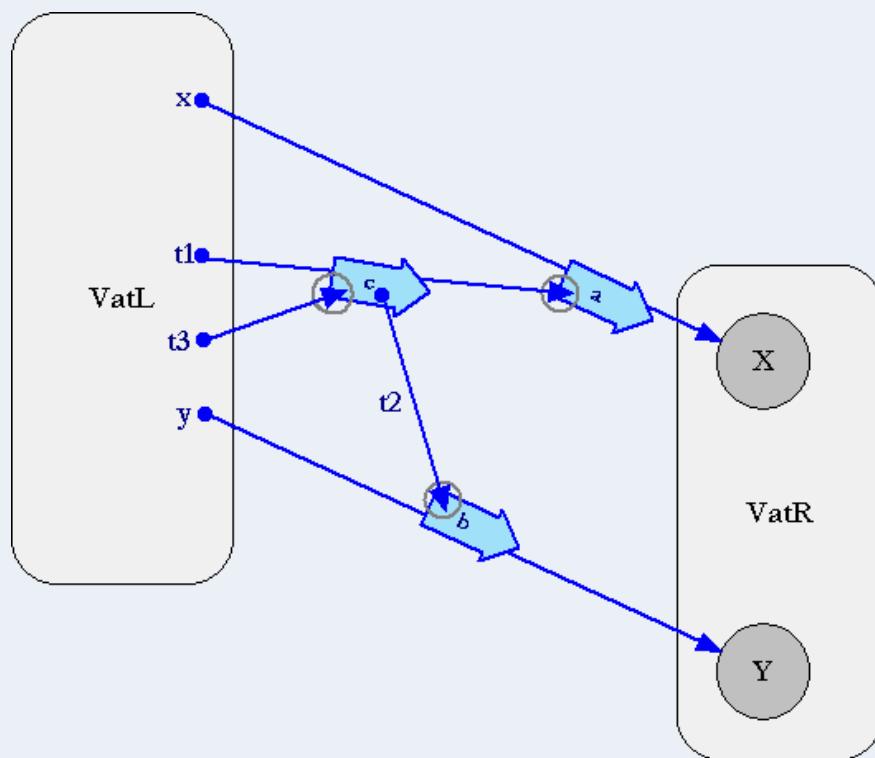
Daca x, y, t1, si t2 sunt localizate pe aceeasi masina remote atunci se poate rezolva in 1 round-trip.

- O cerere trimisa si un raspuns necesar!

# Promise Pipelining

From <http://www.erights.org/elib/distrib/pipeline.html>

The key is that later messages can be sent specifying promises for the results of earlier messages as recipients or arguments, despite the fact that these promises are unresolved at the time these messages are sent. This allows to stream out all three messages at the same time, possibly in the same packet.



# Evaluare

- *call by future*
  - *non-deterministic*: valoarea se va calcula candva intre momentul crearii variabilei *future* si momentul cand aceasta se va folosi
  - *eager evaluation*: imediat ce *future* a fost creata
  - *lazy evaluation*, doar atunci cand e folosita
  - Odata ce valoarea a fost atribuita nu se mai recalculeaza atunci cand se refoloseste.
- *lazy future* : calculul valorii incepe prima oara cand aceasta este ceruta (folosita)
  - e.g. in C++11
    - Politica de apel `std::launch::deferred` -> La apelul `std::async`.

- **Future and Promise**
  - the two sides of an asynchronous operation:
  - **consumer/caller** vs. **producer/implementor**
  - a **caller** of an asynchronous task will get a **Future** as a handle to the computation's result
  - **Future** handles the computation's result
    - e.g. call `get()`
  - The **implementor** must return a **Future**
    - it is responsible for completing that future as soon as the computation is done.

# Blocking vs non-blocking semantic

- Accesare sincrona->
  - De exemplu la transmiterea unui mesaj (se asteapta pana la primirea mesajului)
- Asincron – nu se blocheaza...(doar se verifica...)
- Accesare sincrona-> posibilitati:
  - Accesul blocheaza threadul curent /procesul pana cand se calculeaza valoarea (eventual timeout).
  - Accesul sincronizat produce o eroare (aruncare exceptie)
  - Se poate obtine fie succes daca valoarea este deja calculata sau se transmite eroare daca nu este inca calculata -> poate introduce race conditions.
- in C++11, un thread care are nevoie de valoarea unei *future* se poate bloca pana cand se calculeaza (wait() ori get() ). Eventual timeout.
  - Daca future a aparut prin apelul de tip std::async atunci un apel wait cu blocare poate produce invocare sincrona a functiei care calculeaza rezultatul.

# C++11

- `future`
  - `promise`
  - `async`
  - `packaged_task`

# Future

- (1) *future from packaged\_task*
- (2) *future from async()*
- (3) *future from promise*

# packaged\_task

- `std::packaged_task` object = wraps a callable object

callable object:

- *can be wrapped in a `std::function` object,*
- *passed to a `std::thread` as the thread function,*
- *passed to another function that requires a callable object,*
- *invoked directly.*

# async

- `async`
  - Executa o functia `f` asynchron  
• posibil in alt thread si
  - returneaza un obiect `std::future` care va contine rezultatul

# async

- Depinde de implementare daca `std::async` porneste un nou thread sau daca taskul se va executa sincron atunci cand se cere valoarea pt future.
  - `std::launch::deferred` - se amana pana cand se apeleaza fie `wait()` fie `get()` si se va rula in threadul curent (care poate sa nu fie cel care a apelat `async`)
  - `std::launch::async` - se ruleaza in thread separat (lazy evaluation)

Constant	Explanation
<code>std::launch::async</code>	a new thread is launched to execute the task asynchronously
<code>std::launch::deferred</code>	the task is executed on the calling thread the first time its result is requested (lazy evaluation)

# std::promise

- Furnizeaza un mecanism de a stoca o valoare sau o exceptie care va fi apoi obtinuta asincron via un obiect [std::future](#) care a fost creat prin obiectul [promise](#).
- Actiuni:
  - *make ready*: se stocheaza rezultatul in ‘shared state’.
    - Deblocheaza threadurile care asteapta actualizarea unui obiect future asociat cu ‘shared state’.
  - *release*: se elibereaza referinta la ‘shared state’.
  - *abandon*: shared state = *ready* +
    - exception of type [std::future\\_error](#) with error code [std::future\\_errc::broken\\_promis](#)

# std::promise

**promise** furnizeaza un obiect **future**.

- Se furnizeaza si un mecanism de transfer de informatie intre threaduri
  - T1-> wait()
  - T2-> set\_value() => future ready.
- d.p.d.v al threadului care asteapta nu e important de unde a aparut informatia.

# Exemple

//(1) future from a packaged\_task

```
std::packaged_task<int()> task( []() { return 7; } ); // wrap the function
```

```
std::future<int> f1 = task.get_future(); // get a future
```

```
std::thread(std::move(task)).detach(); // launch on a thread
```

//(2) future from an async()

```
std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });
```

// (3) future from a promise

```
std::promise<int> p;
```

```
std::future<int> f3 = p.get_future();
```

```
std::thread( [&p]{ p.set_value_at_thread_exit(9); }).detach();
```

```
f1.wait();
```

```
f2.wait();
```

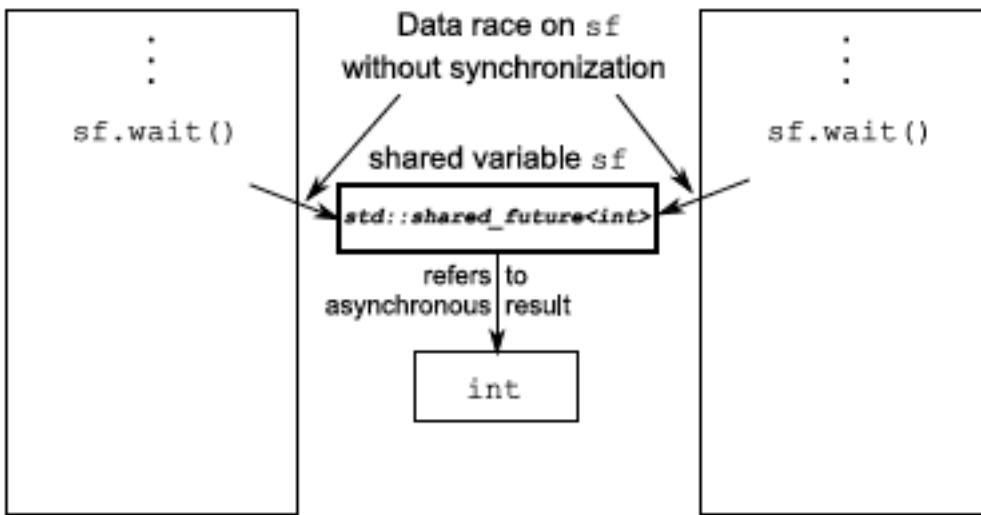
```
f3.wait();
```

# std::shared\_future

PROBLEMA: daca se acceseaza un obiect `std::future` din mai multe threaduri fara sincronizare aditionala => ***data race***.

- `std::future` modeleaza *unique ownership*
  - doar un thread poate sa preia valoarea
- `std::shared_future` permite accesarea din mai multe threaduri
- `std::future` este *moveable* (ownership can be transferred between instances)
- `std::shared_future` este *copyable* (mai multe obiecte pot referi aceeasi stare asociata).

### Thread 1

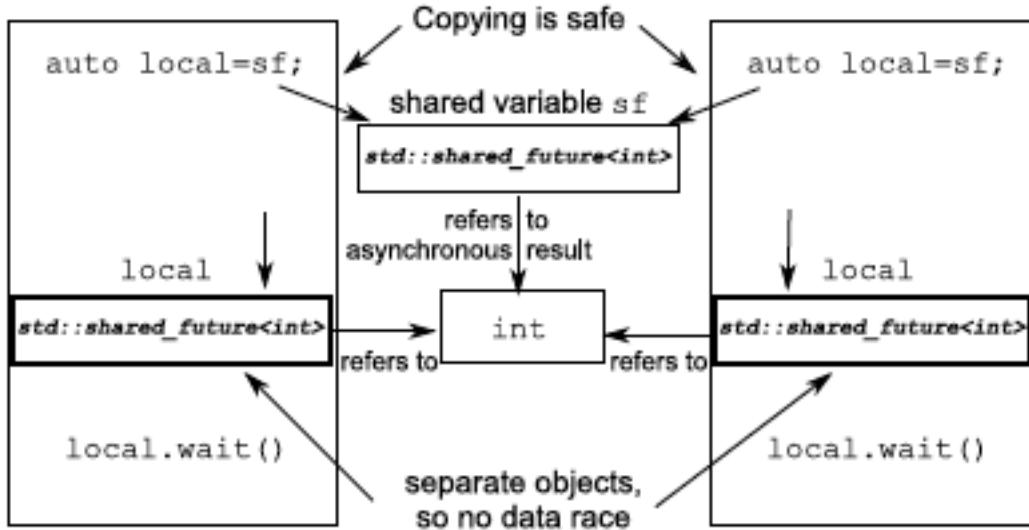


### Thread 2

`std::shared_future`, member functions on an individual object are still unsynchronized.

- To avoid data races when accessing a single object from multiple threads, you must protect accesses with a lock.
- The preferred way to use it would be to take a copy of the object instead and have each thread access its own copy.
- Accesses to the shared asynchronous state from multiple threads are safe if each thread accesses that state through its own `std::shared_future` object.

### Thread 1



# Java

- task ( Runnable vs. Callable)
- Future
- Executor
- CompletableFuture

# Task

- Task = activitate independentă
- Nu depinde de :
  - starea,
  - rezultatul, ori
  - ‘side effects’

ale altor taskuri

=> Concurinta /Paralelism

# Exemplul 1

Aplicatii client server-> task = cerere client

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
    ....  
}
```

# Analiza

- procesare cerere =
  - socket I/O ( read the request + write the response) -> se poate bloca
  - file I/O or make database requests-> se poate bloca
  - Procesare efectiva

*Single-threaded => inefficient*

- Timp mare de raspuns
- Utilizare inefficienta CPU

## Exemplu 2

```
class ThreadPerTaskWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
  
            new Thread(task).start();  
        }  
    }  
    ...  
}
```

# Dezavantaje ale nelimitarii numarului de threaduri create

- **Thread lifecycle overhead**
  - Creare threaduri
- **Resource consumption**
  - Threadurile active consuma resursele sistemului (memorie)
  - Multe threaduri inactive bloacheaza spatiu de memorie -> probleme – garbage collector
  - Multe threaduri => probleme cu CPU-> costuri de performanta
- **Stability**
  - exista o limita a nr de threaduri care se pot crea (depinde de platforma)  
-> OutOfMemoryError.

# Executori

- Task = unitate logica
- Thread -> un mecanism care poate executa taskurile asincron
- Interfata/obiect Executor
  - Mecanism de decuplare a submitterii unui task de executia lui
  - Suport pentru monitorizarea executiei
  - Se bazeaza pe sablonul producator-consumator

```
public interface Executor {  
    void execute(Runnable command);  
}
```

# Exemplu 3

```
class TaskExecutionWebServer {  
    private static final int NTHREADS = 50;  
    private static final Executor exec= Executors.newFixedThreadPool(NTHREADS);  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            exec.execute(task);  
        }  
    }  
    ...  
}
```

# Adaptare – task per thread

```
public class ThreadPerTaskExecutor implements Executor {  
    public void execute(Runnable r) {  
        new Thread(r).start();  
    };  
}
```

```
public class WithinThreadExecutor implements Executor {  
    public void execute(Runnable r) {  
        r.run();  
    };  
}
```

//Executor care executa taskurile direct in threadul apelant (synchronously).

# *Execution policy*

“what, where, when, how” pentru executia taskurilor

= instrument de management al resurselor

- In ce thread se executa un anumit task?
- In ce ordine se aleg taskurile pentru executie (FIFO, LIFO, priority)?
- Cate taskuri se pot executa concurrent?
- Cate taskuri se pot adauga in coada de executie?
- Daca sistemul este supracincarat - *overloaded*, care task se va alege pentru anulare si cum se notifica aplicatia care l-a trimis?
- Ce actiuni trebuie sa fie facute inainte si dupa executia unui task?

# Thread pool

- Un executor care gestioneaza un set omogen de threaduri = *worker threads*
- Se foloseste
  - *work queue*  
*pentru stocare* task-uri
- Worker thread =>
  - cerere task din *work queue*,
  - Executie task
  - Intoarcere in starea de asteptare task.

# Variante - Java

- [newFixedThreadPool](#).

A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected Exception).

- [newCachedThreadPool](#).

A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

- [newSingleThreadExecutor](#).

A single-threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO, priority order).

- [newScheduledThreadPool](#).

A fixed-size thread pool that supports delayed and periodic task execution, similar to Timer.

# Runnable vs. Callable

- abstract computational tasks:
  - Runnable
  - Callable
    - Returns a value
- Task
  - Start
  - [eventually] terminates
- Task Lifecycle:
  - created
  - submitted
  - started
  - completed
- Anulare (cancel)
  - Taskurile submise dar nepornite se pot anula
  - Taskurile pornite se pot anula doar daca raspund la intreruperi
  - Taskurile terminate nu sunt influente de ‘cancel’.

# Interfetele Callable si Future

```
public interface Callable<V> {  
    V call() throws Exception;  
}  
public interface Future<V> {  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
  
    V get() throws InterruptedException, ExecutionException, CancellationException;  
  
    V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException,  
        CancellationException, TimeoutException;  
}
```

# Java

## Future

FutureTask -> A cancellable asynchronous computation.

## CompletableFuture

# apel direct fara executor

```
public class Test {  
    public static class AfisareMesaj implements Callable<String>{  
        private String msg;  
        public AfisareMesaj(String m){  
            msg = m;  
        }  
        public String call(){  
            String threadName = Thread.currentThread().getName();  
            // System.out.println(msg +" "+threadName);  
            return msg + " "+threadName;  
        }  
    }  
    public static void main(String a[]){  
        FutureTask<String> fs = new FutureTask<String>(new AfisareMesaj("TEST"));  
        fs.run();  
        try {  
            System.out.println(fs.get());  
        } catch (InterruptedException | ExecutionException e2) {  
            e2.printStackTrace();  
        }  
    }  
}
```

# Suma numere consecutive – afisare rezultate

```
public class MyRunnable implements Runnable {  
    private final long countUntil;  
  
    MyRunnable(long countUntil) {  
        this.countUntil = countUntil;  
    }  
  
    @Override  
    public void run() {  
        long sum = 0;  
        for (long i = 1; i < countUntil; i++) {  
            sum += i;  
        }  
        System.out.println(sum);  
        //global_variable = sum;  
    }  
}
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
    private static final int NTHREADS = 10;

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
        for (int i = 0; i < 500; i++) {
            Runnable worker = new MyRunnable(10000000L + i);
            executor.execute(worker);
        }
        // This will make the executor accept no new threads
        // and finish all existing threads in the queue
        executor.shutdown();
        // Wait until all threads are finished
        executor.awaitTermination(); !!!!!!!!!!!!!!!
        System.out.println("Finished all threads");
    }
}
```

# Recommended termination of Executors

```
void shutdownAndAwaitTermination(ExecutorService pool) {  
    pool.shutdown(); // Disable new tasks from being submitted  
    try {  
        // Wait a while for existing tasks to terminate  
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {  
            pool.shutdownNow(); // Cancel currently executing tasks  
            // Wait a while for tasks to respond to being cancelled  
            if (!pool.awaitTermination(60, TimeUnit.SECONDS))  
                System.err.println("Pool did not terminate");  
        }  
    } catch (InterruptedException ie) {  
        // (Re-)Cancel if current thread also interrupted  
        pool.shutdownNow();  
        // Preserve interrupt status  
        Thread.currentThread().interrupt();  
    }  
}
```

# Exemplu: Futures & Callable

## Suma de numere consecutive – acumulare

```
import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {
    @Override
    public Long call() throws Exception {
        long sum = 0;
        for (long i = 0; i <= 10000; i++) {
            sum += i;
        }
        return sum;
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableFutures {
    private static final int NTHREADS = 10;
    private static final int MAX = 200;

    public static void main(String[] args) {
        ExecutorService executor =
            Executors.newFixedThreadPool(NTHREADS);
        List<Future<Long>> list =
            new ArrayList<Future<Long>>();
        for (int i = 0; i < MAX; i++) {
            Callable<Long> worker = new MyCallable();
            Future<Long> submit =
                executor.submit(worker);
            list.add(submit);
    }
}
```

```
long sum = 0;
System.out.println(list.size());
// now retrieve the result
for (Future<Long> future : list) {
    try {
        sum += future.get();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}
System.out.println(sum);
executor.shutdownNow();
}
```

# CompletableFuture (from docs...)

- A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.
- When two or more threads attempt to complete or cancel a CompletableFuture, only one of them succeeds.
- CompletableFuture implements interface CompletionStage with the following policies:
  - Actions supplied for dependent completions of non-async methods may be performed by the thread that completes the current CompletableFuture, or by any other caller of a completion method.
  - All async methods without an explicit Executor argument are performed using the ForkJoinPool.commonPool() (unless it does not support a parallelism level of at least two, in which case, a new Thread is created to run each task).
    - To simplify monitoring, debugging, and tracking, all generated asynchronous tasks are instances of the marker interface CompletableFuture.AynchronousCompletionTask.
  - All CompletionStage methods are implemented independently of other public methods, so the behavior of one method is not impacted by overrides of others in subclasses.

# runAsync

```
CompletableFuture<Void> future = CompletableFuture.runAsync(  
    () -> {  
        try {    TimeUnit.SECONDS.sleep(1);  }  
        catch (InterruptedException e) {    throw new IllegalStateException(e);  }  
  
        System.out.println("I'll run in a separate thread than the main thread.");  
    }  
);  
  
future.get();
```

# supplyAsync

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(  
    new Supplier<String>() {  
        @Override  
        public String get() {  
            try { TimeUnit.SECONDS.sleep(5); }  
            catch (InterruptedException e) { throw new IllegalStateException(e); }  
  
            return "Result of the asynchronous computation";  
        }  
    }  
);  
String result = future.get();
```

# Variants of runAsync() and supplyAsync()

```
static CompletableFuture<Void> runAsync(Runnable runnable)
```

```
static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
```

```
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

```
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
```

- ForkJoinPool.commonPool()

# thenApply

```
// Create a CompletableFuture
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(
    () -> { try { TimeUnit.SECONDS.sleep(1); }
        catch (InterruptedException e) { throw new IllegalStateException(e); }

    return "Ana";}
);

// Attach a callback to the Future using thenApply()
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(
    name -> { return "Hello " + name;}
);

// Block and get the result of the future.
System.out.println(greetingFuture.get());
```

- `.thenApply` - takes a Function<T,R> as an argument

## .thenApply(..... ).thenApply(

```
CompletableFuture<String> welcomeText = CompletableFuture.supplyAsync(  
    () -> {  
        try {    TimeUnit.SECONDS.sleep(1);  }  
        catch (InterruptedException e) {    throw new IllegalStateException(e);  }  
        return "Ana";})  
.thenApply(name -> {  return "Hello " + name;})  
.thenApply(greeting -> {  return greeting + ", Welcome ! ";}  
);  
  
System.out.println(welcomeText.get());
```

# thenApply() variants

class CompletableFuture<T>

methods:

<U> CompletableFuture<U> **thenApply**(Function<? super T,? extends U> fn)

<U> CompletableFuture<U> **thenApplyAsync**(Function<? super T,? extends U> fn)

<U> CompletableFuture<U> **thenApplyAsync**(Function<? super T,? extends U> fn, Executor executor)

# thenAccept

```
public static String method1(){  
    System.out.println("salutare");  
    return "salutare";  
}  
  
static void method2(String arg){  
    System.out.println("greetings " +arg);  
}  
  
public static void main( String[] args ) throws InterruptedException{  
    System.out.println("Main thread running... thread id: " +  
    Thread.currentThread().getId());  
    CompletableFuture.supplyAsync(ExempleCompletableFuture::method1).  
        thenAccept(ExempleCompletableFuture::method2);  
    System.out.println("Main thread finished");  
}
```

thenApply returns result of current stage whereas thenAccept does not

# Variante - metode

<b>Method</b>	<b>Async method</b>	<b>Arguments</b>	<b>Returns</b>
thenRun()	thenRunAsync()	—	—
thenAccept()	thenAcceptAsync()	Result of previous stage	—
thenApply()	thenApplyAsync()	Result of previous stage	Result of current stage
thenCompose()	thenComposeAsync()	Result of previous stage	Future result of current stage
thenCombine()	thenCombineAsync()	Result of two previous stages	Result of current stage
whenComplete()	whenCompleteAsync()	Result or exception from previous stage	—

# Exemplu complex

[GitHub - atomix/atomix: A reactive Java framework for building fault-tolerant distributed systems](#)

[atomix/PartitionedDistributedCollectionProxy.java at master · atomix/atomix · GitHub](#)

# Curs 8

Programare Paralela si Distribuita

OpenMP

- OpenMP
  - <http://www.openmp.org>
- OpenMP 5.0
  - <https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf>
- Tutoriale:
  - **TutorialOpenMP.pdf** pentru OpenMP 3.0 in “Class Materials” - Cursuri

# Motivatie

- managementul explicit al threadurilor este de multe ori dificil si poate fi automatizat
  - Consideram exemplul: ‘adunare vectori’
    - Din codul sequential putem deduce ce trebuie executat in parallel - ciclul for
    - conversia spre codul paralel se poate face mecanic
      - trebuie doar sa specificam ca respectivul ciclu se poate face in parallel
      - lasam compilatorul sa realizeze transformarea
    - OpenMP executa aceste automatizari!!!

# Ce este OpenMP?

- Acronimul - OpenMP ?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP => Application Program Interface (API) pentru ***multi-threaded, shared memory parallelism.***
  - Componente:
    - Compiler Directives,
    - Runtime Library Routines,
    - Environment Variables
- OpenMP
  - directive-based method to invoke parallel computations on share-memory multiprocessors

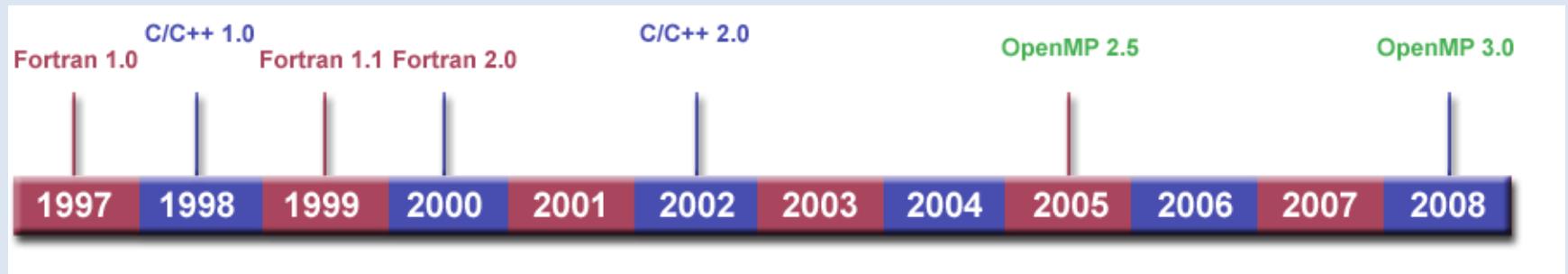
# History of OpenMP

- The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.
- Led by the **OpenMP Architecture Review Board** (ARB). Original ARB members included: (*Disclaimer: all partner names derived from the [OpenMP web site](#)*)
  - Compaq / Digital
  - Hewlett-Packard Company
  - Intel Corporation
  - International Business Machines (IBM)
  - Kuck & Associates, Inc. (KAI)
  - Silicon Graphics, Inc.
  - Sun Microsystems, Inc.
  - U.S. Department of Energy ASCI program

# Other Contributors

- Endorsing application developers
  - ADINA R&D, Inc.
  - ANSYS, Inc.
  - Dash Associates
  - Fluent, Inc.
  - ILOG CPLEX Division
  - Livermore Software Technology Corporation (LSTC)
  - MECALOG SARL
  - Oxford Molecular Group PLC
  - The Numerical Algorithms Group Ltd.(NAG)
- Endorsing software vendors
  - Absoft Corporation
  - Edinburgh Portable Compilers
  - GENIAS Software GmbH
  - Myrias Computer Technologies, Inc.
  - The Portland Group, Inc. (PGI)

# OpenMP Release History



# Scop: OpenMP

- Standardizare
  - furnizeaza un standard pentru arhitecturi/platforme de tip “shared memory”
- Productivitate
  - stabileste un set limitat de directive simple
  - se poate ajunge la un nivel semnificativ de paralelizare doar folosind 3 or 4 directives.
- Simplu de folosit
  - paralelizare incrementală
  - permite -> coarse-grain and fine-grain parallelism
- Portabilitate
  - Suporta Fortran (77, 90, and 95), C, si C++
- Public forum pentru API si participare

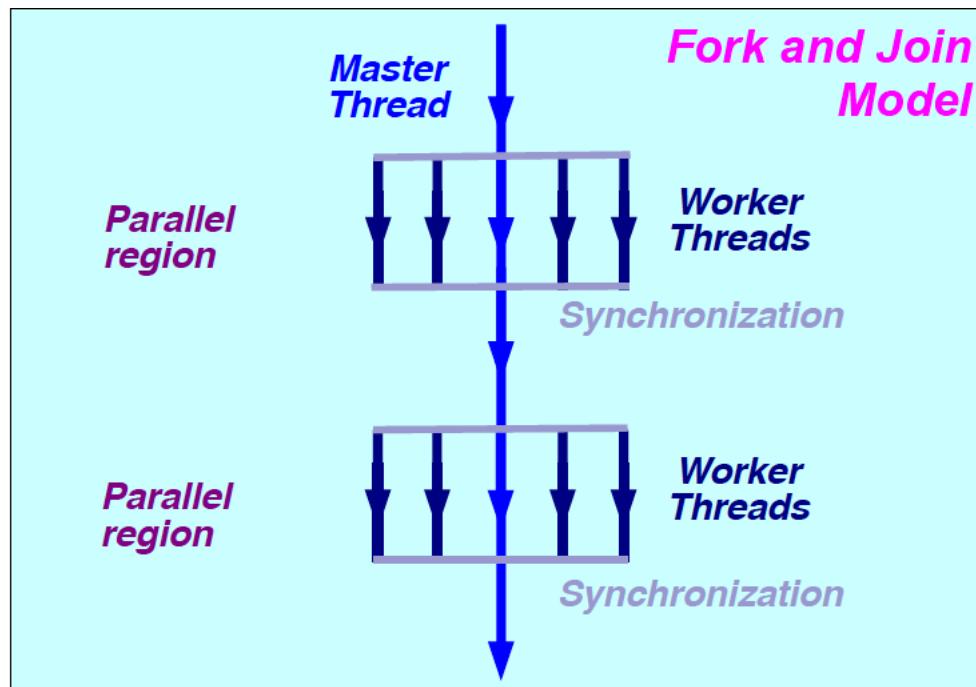
# OpenMP?

- OpenMP API are specificatii pentru C/C++ si Fortran.
- OpenMP nu modifica codul secvential initial
  - Fortran -> comments
  - C/C++ -> pragmas
- OpenMP website: <http://www.openmp.org>
  - diverse tutoriale
    - slide-urile folosesc imagini din acestea

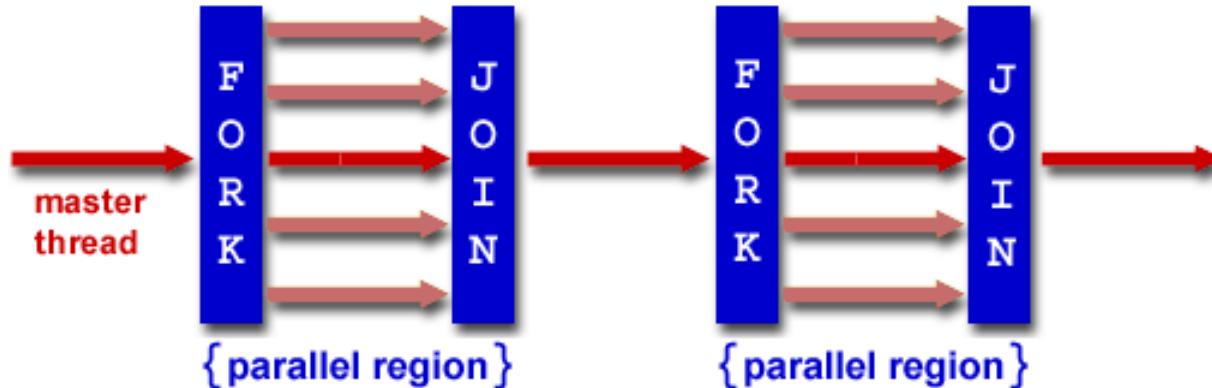
# Compilare si executie

- gcc 4.2 and above suporta OpenMP 3.0
  - **gcc –fopenmp a.c**
- executie – ca si la programele secentiale
  - ‘a.out’
- Compiler documentation
  - IBM: [www-4.ibm.com/software/ad/fortran](http://www-4.ibm.com/software/ad/fortran)
  - Cray: <http://docs.cray.com/> (Cray Fortran Reference Manual)
  - Intel: [www.intel.com/software/products/compilers/](http://www.intel.com/software/products/compilers/)
  - PGI: [www.pgroup.com](http://www.pgroup.com)
  - PathScale: [www.pathscale.com](http://www.pathscale.com)
  - GNU: [gnu.org](http://gnu.org)

# OpenMP Model



# OpenMP execution model



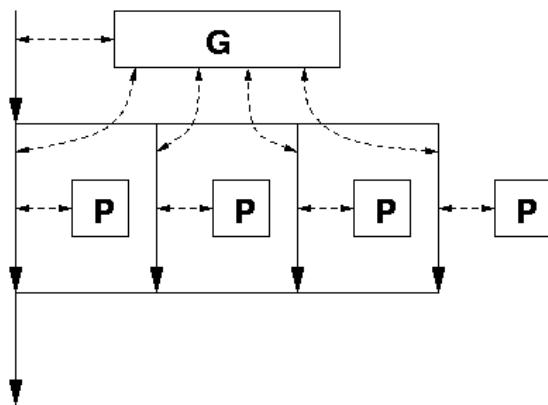
- OpenMP foloseste modelul ‘fork-join’ de executie paralela
  - Toate programele OpenMP incep cu un singur thread - **master thread**.
  - master thread se executa secvential pana cand intalneste o regiune paralela - **parallel region**, cand creeaza un set de threaduri **team of parallel threads** (FORK).
  - Atunci cand regiunea paralela se termina threadurile se sincronizeaza si se ‘termina’ (JOIN).
    - implementarile pot folosi un thread pool care gestioneaza setul de threaduri care se folosesc in regiunea paralela – este posibil sa mentina aceste threaduri

# OpenMP- structura generala cod

```
#include <omp.h>
main () {
    int var1, var2, var3;
    Serial code
    ...
    /* Beginning of parallel section. Fork a team of threads. Specify variable scoping*/
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        /* Parallel section executed by all threads */
        ...
        /* All threads join master thread and disband*/
    }
    Resume serial code
    ...
}
```

# Data model

## Variabile: private + shared



P = private data space  
G = global data space

- global data space – accesibile de catre toate threadurile (**shared variables**).
- private space – accesibile doar la nivel de thread (**private variables**)
  - există variatii care depend de modul de initializare

# Memory Model

- OpenMP furnizeaza
  - "relaxed-consistency" +
  - "temporary" view of thread memory
- adica thread-urile pot face "cache" pe date si nu se mentine o consistenta exacta cu memoria globala tot timpul.

=>

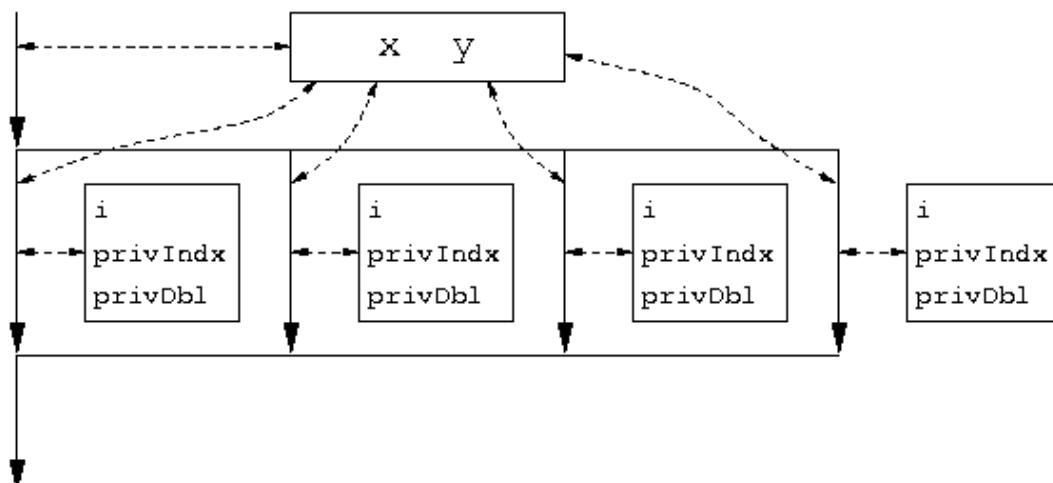
- Atunci cand este critic ca toate threadurile sa vada identic o variabila partajata este responsabilitatea programatorului sa asigure aceasta prin comanda FLUSH  
...

```

int y[MAX];
#pragma omp parallel for private( privIndx, privDbl )
for ( i = 0; i < arraySize; i++ ) {
    for ( privIndx = 0; privIndx < 16; privIndx++ ) {
        privDbl = ( (double) privIndx ) / 16;
        y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) ) + cos( privDbl );
    }
}

```

Parallel for loop index is  
Private by default.



execution context for "arrayUpdate\_II"

# OpenMP Programming Model

- Directivele OpenMP in C and C++ se bazeaza pe directivele de compilare `#pragma`
- Format:

```
#pragma omp directive [clause list]
```
- programele OpenMP se executa seventional pana la intalnirea unei regiuni paralele (parallel directive)

```
#pragma omp parallel [clause list]
/* structured block */
```
- Threadul care intalneste/executa directive paralela devine master pentru grupul de threaduri care o executa si are ID 0

# Clauze

- [clause list] specifică
  - *conditional parallelization*
  - *number of threads*
  - *data handling*
- **Conditional Parallelization:**
  - clause if (scalar expression)
- **Degree of Concurrency:**
  - clause num\_threads (integer expression)
- **Data Handling:**
  - clause private (variable list)
  - clause firstprivate (variable list)
    - private și initializează valoarea corespunzătoare variabilei (cf nume) înainte de regiunea paralela
  - clause shared (variable list)

# Exemplu de translatare a unui cod OpenMP.

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    (
        // parallel segment
    )
    // rest of serial segment
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (....., internal_thread_fn_name, ...);
    for (i = 0; i < 8; i++)
        pthread_join (.....);
    // rest of serial segment
}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
    // parallel segment
]
```

Corresponding Pthreads translation

Code inserted by the OpenMP compiler

# OpenMP Code Structure – C/C++

```
#include <omp.h>
main () {
    int var1, var2, var3;
```

*Serial code*

```
.  
. .
```

*Beginning of parallel section. Fork a team of threads. Specify variable scoping*

```
#pragma omp parallel private(var1, var2) shared(var3)
{
```

*Parallel section executed by all threads*

```
.  
. .
```

*All threads join master thread and disband*

```
}
```

*Resume serial code*

```
.  
. .
```

```
}
```

# PARALLEL Region – Number of Threads

- Numarul de threaduri care se folosesc intr-o regiune paralela depinde de urmatorii factori in urmatoarea ordine de precedenta :
  1. **IF** clause
  2. **NUM\_THREADS** clause
  3. apel **omp\_set\_num\_threads()** library function  
se apeleaza in sectiune de cod seriala inainte de o regiune parallel (`void omp_set_num_threads(int num_threads)`)
  4. setare **OMP\_NUM\_THREADS** environment variable
  5. implementare implicita (eventual) – nr de core-uri.
- Threads sunt numerotate de la 0 (master thread) to N-1

# OpenMP runtime environment

- `omp_get_num_threads`
- `omp_get_thread_num`
- `omp_in_parallel`
- Routines related to locks
- .....

# OMP\_GET\_NUM\_THREADS() and OMP\_GET\_THREAD\_NUM()

- OMP\_GET\_NUM\_THREADS()
  - Returneaza numarul de thread-uri care se executa intr-o regiune paralela curenta

C/C++

```
#include <omp.h>
int omp_get_num_threads(void)
```

- OMP\_GET\_THREAD\_NUM()
  - Returneaza numarul threadului care apeleaza functia si care actioneaza intr-o regiune paralela
  - returneza o valoare intre OMP\_GET\_NUM\_THREADS-1.
  - master thread -> thread 0.

## OMP\_GET\_MAX\_THREADS()

- Returneaza valoare maxima care poate fi returnata de un apel al functiei OMP\_GET\_NUM\_THREADS
- in general reflecta numarul de threaduri setat prin variabila de mediu OMP\_NUM\_THREADS environment variable sau de functia OMP\_SET\_NUM\_THREADS()
- Poate fi apelata atat din regiune parala cat si din sectiune secentiala

# OpenMP environment variables

- OMP\_NUM\_THREADS
- OMP\_SCHEDULE

# OMP\_GET\_THREAD\_LIMIT() and OMP\_GET\_NUM\_PROCS()

- OMP\_GET\_THREAD\_LIMIT()
  - New with OpenMP 3.0
  - Returns the maximum number of OpenMP threads available to a program
- OMP\_GET\_NUM\_PROCS()
  - Returns the number of processors that are available to the program

# Exemplificare

```
int a=1, b=2, c=3, d;  
#pragma omp parallel if (is_parallel== 1) num_threads(8)  
  \  
  private (a) shared (b) firstprivate(c) lastprivate(d)  
  
{  
  d = omp_get_thread_num();  
  /* structured block */  
}  
d=?
```

# PRIVATE and SHARED Clauses

- PRIVATE Clause
  - un nou obiect in fiecare thread
  - toate referintele la obiectul original sunt redirectionate
  - trebuie sa se asume ca nu sunt initializate
- SHARED Clause
  - o variabila *shared* exista doar intr-o locatie de memorie
  - este responsabilitatea programatorului sa asigure accesul correct (eventual via CRITICAL sections daca este necesar)

# FIRSTPRIVATE / LASTPRIVATE Clauses

- FIRSTPRIVATE Clause
  - Combina PRIVATE clause cu automatic initialization
- LASTPRIVATE Clause
  - Combina PRIVATE clause cu o copierea valorii din threadul care face ultima iteratie sau sectiune in variabila initiala din codul sequential anterior

# PRIVATE Variables

```
main()
{
    int A = 10;
    int B, C;
    int n = 20;
    #pragma omp parallel
    {
        #pragma omp for private(i) firstprivate(A) lastprivate(B)
        for (int i=0; i<n; i++)
        {
            ....
            B = A + i; /* A undefined unless declared firstprivate */
            ....
        }
        C = B; /* B undefined unless declared lastprivate */
    } /* end of parallel region */
}
```

# Restrictii pentru PARALLEL Region

- parallel region bloc – trebuie sa fie un bloc structurat
  - nu se poate intra sau iesi fortat (not branch into or out of a parallel region) – no goto
- doar o clauza IF
- doar o clauza NUM\_THREADS

# Exemplu - PARALLEL Region

```
#include <omp.h>
main () {
int nthreads, tid;
/* Fork a team of threads with each thread having a private tid variable */
omp_set_num_threads(4);
#pragma omp parallel private(tid)
{
/* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0) {
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
} /* All threads join master thread and terminate */
}
```

# Reduction Clause in OpenMP

- reduction clause specifica cum se pot combina valorile locale fiecarui thread (ale unor variabile private) intr-o singura valoare in threadul master atunci cand regiunea paralela se termina

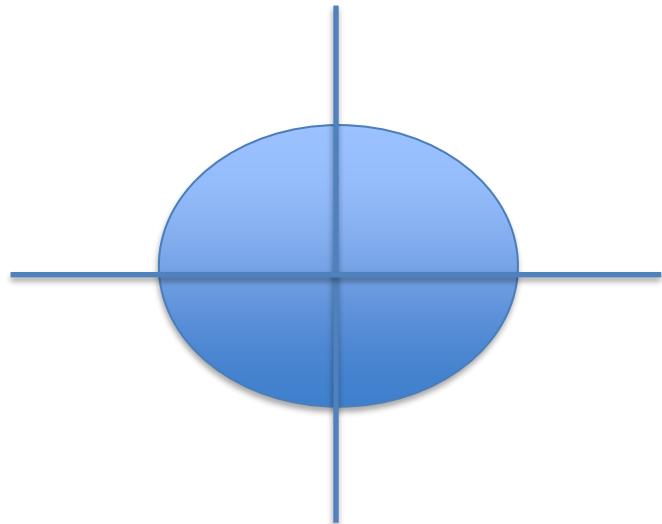
`reduction (operator: variable list)`

- operator: +, \*, -, &, |, ^, &&, and ||
- OpenMP generareaza cod astfel incat sa nu apara *race condition*

```
sum = 0.0;  
#pragma parallel default(none) shared (n, x) private (l) reduction(+ : sum)  
{  
for(l=i_start; l<i_final; l++) sum = sum + x(l);  
}
```

# Approximare Pi – secvential

```
double PI(long npoints) {  
    double sum = 0;  
    int num_threads = omp_get_num_threads();  
    sum = 0;  
    double rand_no_x, rand_no_y;  
    unsigned long seed = time(NULL);  
    srand(seed);  
    for (long i = 0; i < npoints; i++) {  
        rand_no_y = (double)(rand()) / (double)(RAND_MAX);  
        rand_no_x = (double)(rand()) / (double)(RAND_MAX);  
        if (((rand_no_x ) * (rand_no_x ) + (rand_no_y ) * (rand_no_y )) < 1) sum++;  
    }  
    return 4*sum / npoints;  
}
```



# OpenMP Programming: Example

```
/*
 ****
An OpenMP version of a threaded program to compute PI.
**** */

double PI(long npoints, int no_threads) {
    double sum = 0;
#pragma omp parallel shared (npoints) reduction(+: sum) num_threads(no_threads)
{
    int num_threads = omp_get_num_threads();
    long sample_points_per_thread = npoints / num_threads;
    sum = 0;
    double rand_no_x, rand_no_y;
std::time_t nt = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
thread_local unsigned long seed = time(&nt);
srand(seed);
    for (long i = 0; i < sample_points_per_thread; i++) {
        rand_no_y = (double)(rand()) / (double)(RAND_MAX);
        rand_no_x = (double)(rand()) / (double)(RAND_MAX);
        if (((rand_no_x ) * (rand_no_x ) + (rand_no_y ) * (rand_no_y )) < 1) sum++;
    }
}
return 4*sum / npoints;
}
```

# Exemplu REDUCTION Clause

```
#include <omp.h>

main () {
int i, n, chunk;
float a[100], b[100], result;
/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++) {
    a[i] = i * 1.0;
    b[i] = i * 2.0;
}
#pragma omp parallel for default(shared) private(i) \
    schedule(static,chunk) reduction(+:result)
for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
// final regiune paralela
printf("Final result= %f\n",result);
}
```

# Work-sharing constructs

- `#pragma omp for [clause ...]`
- `#pragma omp section [clause ...]`
- `#pragma omp single [clause ...]`
- distributie automata intre threaduri
- trebuie sa fie incluse in regiuni paralele
- nu este implicit bariera la inceput dar este o bariera implicita la iesire daca nu se specifica altfel prin clauza **nowait**
- The work is distributed over the threads

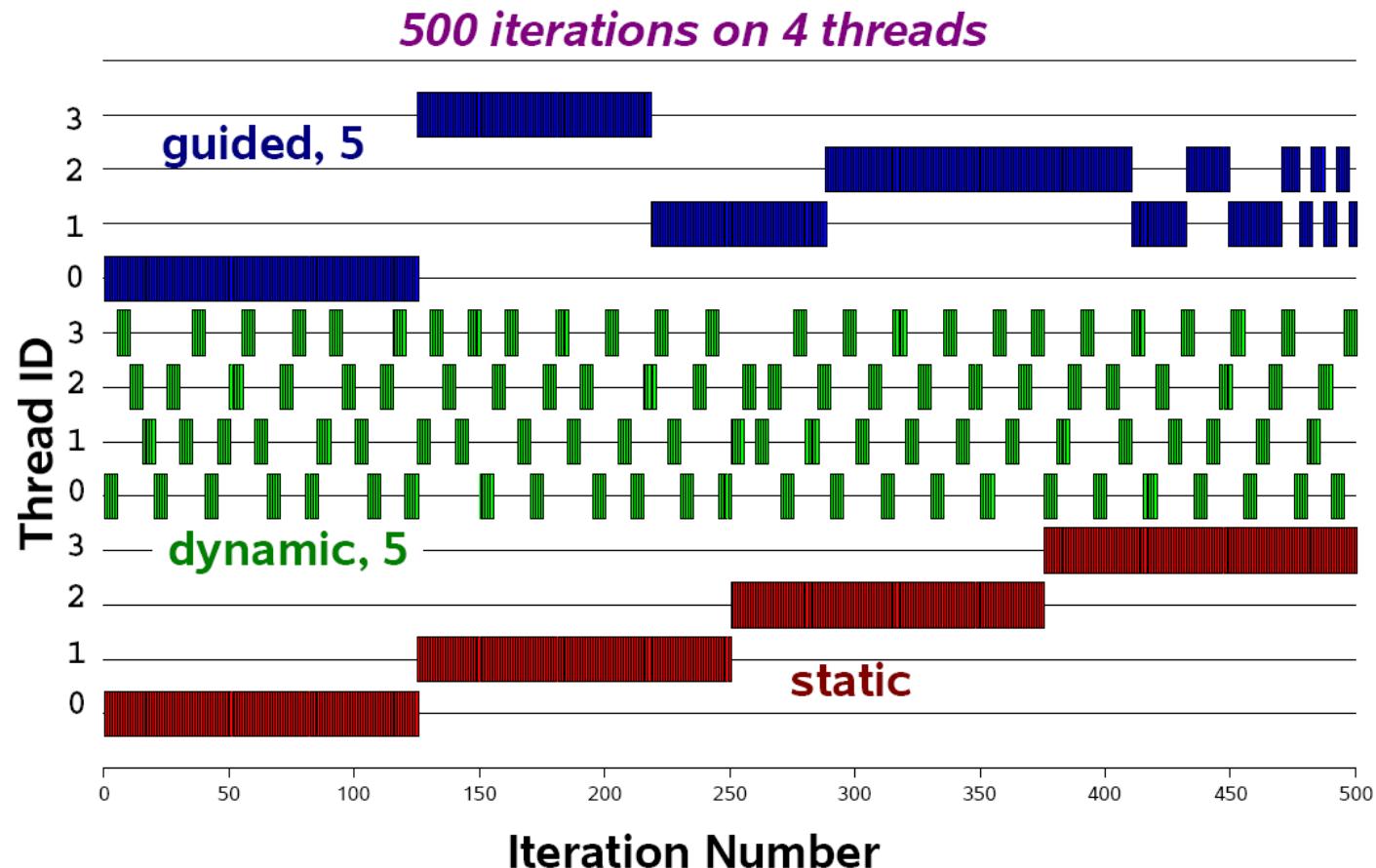
## omp for directive

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
}

/*-- End of parallel region --*/
(implied barrier)
```

- Schedule clause (decide how the iterations are executed in parallel):
   
**schedule (static | dynamic | guided [, chunk])**



# section directive

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

        #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];
    }
    /*-- End of sections --*/
}

/*-- End of parallel region --*/
```

```
#pragma omp parallel  
#pragma omp for  
for (...)
```

Single PARALLEL loop

```
#pragma omp parallel for  
for (....)
```

```
#pragma omp parallel  
#pragma omp sections  
{ ... }
```

Single PARALLEL sections

```
#pragma omp parallel sections  
{ ... }
```

# Synchronization: barrier

```
For(=0; I<N; I++)  
    a[I] = b[I] + c[I];
```

```
For(I=0; I<N; I++)  
    d[I] = a[I] + b[I]
```

Both loops are in parallel region  
With no synchronization in between.  
What is the problem?

Fix:

```
For(I=0; I<N; I++)  
    a[I] = b[I] + c[I];
```

```
#pragma omp barrier
```

```
For(I=0; I<N; I++)  
    d[I] = a[I] + b[I]
```

# Critical session

```
int sum=0  
For(I=0; I<N; I++) {  
    .....  
    sum += A[I];  
    .....  
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {  
    .....  
    #pragma omp critical  
    {  
        sum += A[I];  
    }  
    .....  
}
```

# Sequential Matrix Multiply

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
    for (k=0; k<n; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

# OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

# OpenMP Matrix Multiply - **collapse**

```
#pragma omp parallel for private(j, k) collapse(2)
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0;
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

se paralelizeaza  
ambele instructiuni  
for

# FLUSH Directive

- Identifica un punct de sincronizare la care implementarea trebuie sa furnizeze o vedere consistenta a memoriei
- Variabilele care sunt vizibile threadurilor (globale) trebuie sa fie actualizare (written back to memory) daca sunt valori cashuite in threaduri
- Se instruieste compilatorul ca la acel punct o variabila trebuie sa fie “written to/read from the memory system” – nu mai poate fi tinuta intr-un registru CPU local
  - este posibil ca variabilele sa fie tinute intr-un registru atunci cand se executa un ciclu (loop) pentru a se eficientiza codul

## FLUSH Directive (2)

C/C++: `#pragma omp flush (list) newline`

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, the pointer itself is flushed, not the object to which it points.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; i.e., compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

## FLUSH Directive (3)

- The FLUSH directive is implied for the directives shown in the table below.
  - The directive is not implied if a NOWAIT clause if present.

Fortran	C/C++
BARRIER	barrier
END PARALLEL	parallel – upon entry and exit
CRITICAL and END CRITICAL	critical – upon entry and exit
END DO	
END SECTIONS	ordered – upon entry and exit
END SINGLE	
ORDERED and END ORDERED	for – upon exit sections – upon exit single – upon exit

# Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
```

```
#pragma omp single [clause list]  
structured block
```

```
#pragma omp master  
structured block
```

```
#pragma omp critical [(name)]  
structured block
```

```
#pragma omp ordered  
structured block
```

## work construct - task

- foarte eficient
- se bazeaza pe o abordare de tip ‘thread-pool’
- avantaj fata de *section* – un task se poate tine in ‘asteptare’ (nefinalizat) pana cand alte taskuri se executa => implementare divide&impera

# OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- all lock routines also have a nested lock counterpart for recursive mutexes.

# rezumat

- `#pragma omp parallel`
  - regiune paralela
  - clauze:
- `#pragma omp for`
  - clauze
  - reduce
- `#pragma omp sections`
  - clauze
- `#pragma omp barrier`
- `#pragma omp critical`

# Curs 9

Programare Paralela si Distribuita

Metode de evaluare a performantei programelor paralele

Granularitate

Scalabilitate

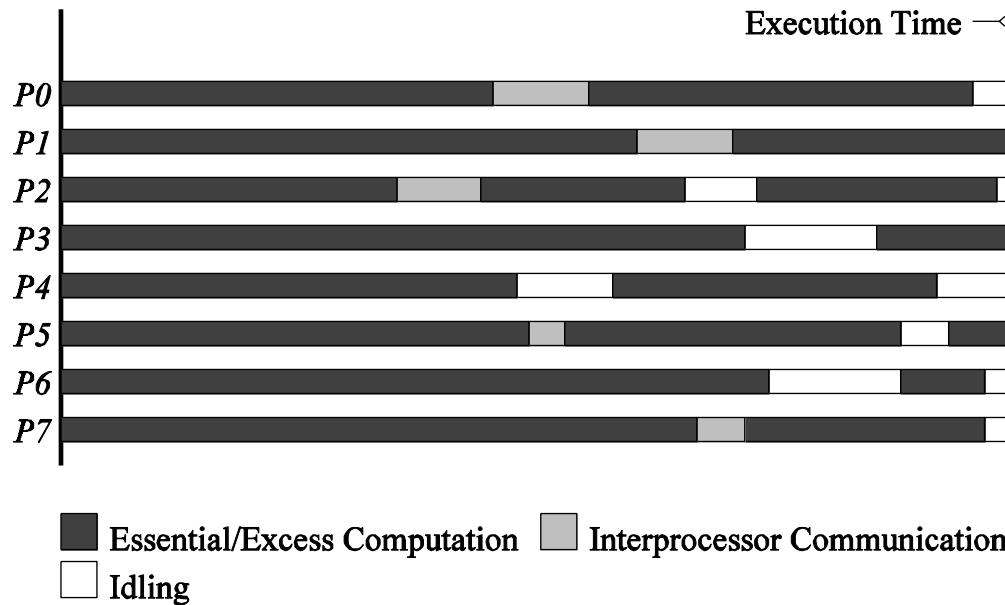
# Complexitate – consideratii generale

- Daca in cazul algoritmilor secentiali performanta este masurata in termenii complexitatilor timp si spatiu, in cazul algoritmilor paraleli se folosesc si alte masuri ale performantei, care au in vedere toate resursele folosite.
  - ***Numarul de procesoare*** in cazul programarii paralele => o resursa importanta
- Pentru compararea corecta a variantei paralele cu cea seriala, trebuie
  - sa se precizeze arhitectura sistemului de calcul paralel  
**(sistem paralel = program + arhitectura pe care se executa)**
  - sa se aleaga algoritmul serial cel mai bun si
  - sa se indice daca exista conditionari ale performantei algoritmului datorate volumului de date.

# Observatii

- In calculul paralel, obtinerea unui timp de executie mai bun nu inseamna neaparat utilizarea unui numar minim de operatii, asa cum este in calculul serial.
- Factorul memorie nu are o importanta atat de mare in calculul paralel (relativ).
- In schimb, o resursa majora in obtinerea unei performante bune a algoritmului paralel o reprezinta numarul de procesoare folosite.
- Daca timpul de executie a unei operatii aritmetice este mult mai mare decat timpul de transfer al datelor intre doua elemente de procesare, atunci intarzierea datorata retelei este nesemnificativa, dar, in caz contrar, timpul de transfer joaca un rol important in determinarea performantei programului.

# Timp de executie



Timpul de executie al unui program paralel masoara perioada care s-a scurs intre momentul initierii primului proces si momentul cand toate procesele au fost terminate.

# Timp de executie vs Complexitate timp

- În timpul executiei fiecare procesor executa
  - operatii de calcul,
  - de comunicatie, sau
  - este in asteptare.

- Timpul total de executie se poate obtine din formula:

$$t_p = (\max i : i \in \overline{0, p-1} : T_{\text{calcul}}^i + T_{\text{comunicatie}}^i + T_{\text{asteptare}}^i))$$

- sau in cazul echilibrarii perfecte ale incarcarii de calcul pe fiecare procesor din formula:

$$t_p = \frac{1}{p} \sum_0^{p-1} (T_{\text{calcul}}^i + T_{\text{comunicatie}}^i + T_{\text{asteptare}}^i)$$

# Evaluarea teoretica a complexitatii-timp

- Ca si in cazul programarii secentiale, pentru a dezvolta algoritmi paraleli eficienti trebuie sa putem face o evaluare a performantei inca din faza de proiectare a algoritmilor.
- Complexitatea timp pentru un algoritm paralel care rezolva o problema  $P(n)$  cu dimensiunea  $n$  a datelor de intrare este o functie  $T$  care depinde de  $n$ , dar si de numarul de procesoare  $p$  folosite.
- Pentru un algoritm paralel, un pas elementar de calcul se considera a fi o multime de operatii elementare care pot fi executate in paralel de catre o multime de procesoare.
- Complexitatea timp a unui pas elementar se poate considera a fi  $O(1)$ .
- Complexitatea timp a unui algoritm paralel este data de numararea atat a pasilor de calcul necesari dar si a pasilor de comunicatie a datelor/acces la memorie.

# Overhead

- $T_{all}$  = timpul total (insumarea timpului pentru toate elementele de procesare).
- $T_s$  = timp serial
- $T_{all} - T_s$  = timp total in care toate procesoarele sunt implicate in operatii care nu sunt strict legate de scopul problemei  
non-goal computation work  
-> total overhead.
- $T_{all} = p \cdot T_p$  ( $p$  = nr. procesoare).
- $T_o = p \cdot T_p - T_s$

## Accelerarea (“speed-up”),

- Accelerarea notata cu  $S_p$ , este definita ca raportul dintre timpul de executie al celui mai bun algoritm serial cunoscut, executat pe un calculator monoprocesor si timpul de executie al programului paralel echivalent, executat pe un sistem de calcul paralel.
- Daca se noteaza cu  $t_1$  timpul de executie al programului serial, iar  $t_p$  timpul de executie corespunzator programului paralel, atunci:

$$S_p(n) = \frac{t_1(n)}{t_p(n)}.$$

- $n$  reprezinta dimensiunea datelor de intrare,
- $p$  numarul de procesoare folosite.

# Variante

- **relativa**, cand ts este timpul de executie al variantei paralele pe un singur procesor al sistemului paralel;
- **reală**, cand se compara timpul executiei paralele cu timpul de executie pentru varianta seriala cea mai rapida, pe un procesor al sistemului paralel;
- **absolută**, cand se compara timpul de executie al algoritmului paralel cu timpul de executie al celui mai rapid algoritm serial, executat de procesorul serial cel mai rapid;
- **asimptotica**, cand se compara timpul de executie al celui mai bun algoritm serial cu functia de complexitate asimptotica a algoritmului paralel, in ipoteza existentei numarului necesar de procesoare;
- **relativ asimptotica**, cand se foloseste complexitatea asimptotica a algoritmului paralel executat pe un procesor.

**Analiza asimptotica** (considera dimensiunea datelor n si numarul de procesoare p foarte mari) ignora termenii de ordin mic, si este folositoare in procesul de constructie al programelor performante.

# Eficienta

- Eficienta este un parametru care măsoară gradul de folosire a procesoarelor.
- Eficienta este definită ca fiind:

$$E = Sp/p$$

- Dacă accelerarea este cel mult egală cu  $p$  se deduce că valoarea eficientei este subunitară.

# Legea lui Amdahl

- Afirma că accelerarea procesării depinde de raportul partii secentiale față de cea paraleizabilă:

seq = fractia calcului secential; (e.g 20%=> seq=20/100)

par = fractia calcului paralelizabil; (e.g 80%=> par=80/100)

Se consideră **calculul serial**  $T_s = \text{seq} + \text{par} = 1$  unitate

Speedup =  $1/(\text{seq} + \text{par}/p)$

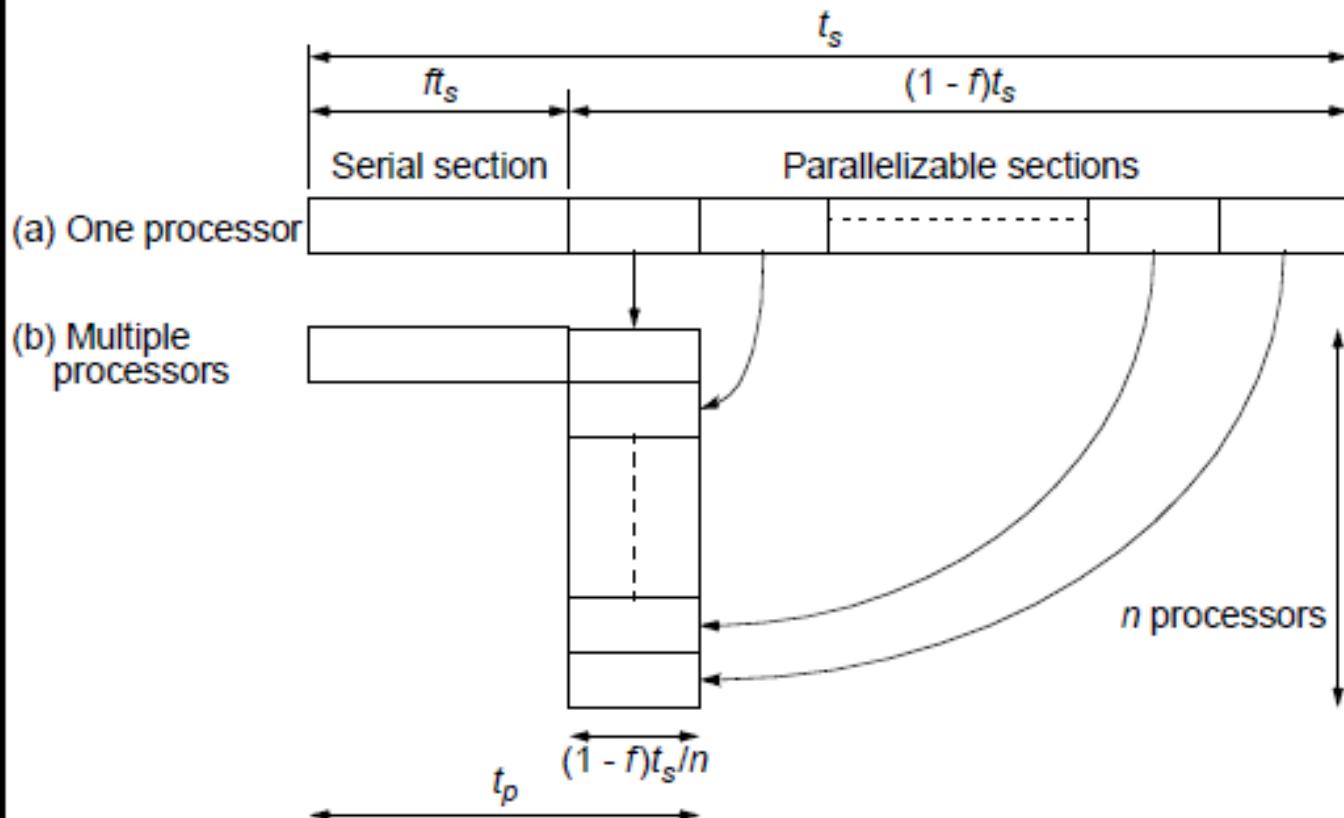
par =  $(1 - \text{seq})$ , p= # procesoare

- $p \rightarrow \text{infinit} \Rightarrow S \sim 1/\text{seq}$  (e.g.  $S \sim 100/20=5$ )
- Limita superioară a accelerării este data de fractia partii secentiale.
- *ATENȚIE: Nu se face analiza în funcție de dimensiunea problemei!*

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1 + (n-1)f}$$

Slide 37

## Maximum Speedup - Amdahl's law



# Legea lui Gustafson

- Considera ca atunci cand dimensiunea problemei creste, partea seriala se micsoreaza in procent :
- $m = \text{dimensiunea problemei}$ ,  $p = \# \text{ procesoare}$ ,  
 $\text{seq}(m) = \text{fractia calcului secvential}$ ;  
 $\text{par}(m) = \text{fractia calcului paralel}$ ;

Considerand ca **programul paralel se executa intr-o unitate de timp** :

$$T_p = \underline{\text{seq}(m) + \text{par}(m)} = 1$$

$$T_s = \text{seq}(m) + p * \text{par}(m)$$

**Atunci**

- $\text{speedup} = T_s / T_p = \text{seq}(m) + p * \text{par}(m)$   
 $\text{speedup} = \text{seq}(m) + p(1 - \text{seq}(m))$

Daca  $\text{seq}(m) \rightarrow 0$  atunci cand  $m \rightarrow \infty \Rightarrow$  se obtine ~ accelerare liniara.

Legea lui Gustafson – optimista

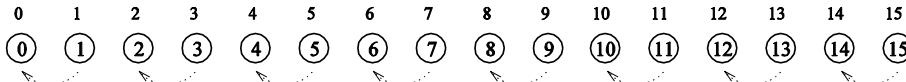
Legea lui Amdahl - pesimista

- Legea lui Gustafson -> presupune ca partea seriala (costul ei) ramane constanta – nu creste odata cu cresterea problemei.
- Legea lui Amdahl -> presupune ca **procentul** partii secentiale este constant - nu depinde de dimensiunea problemei

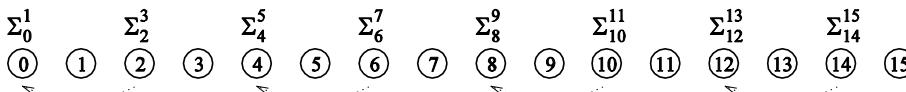
# Exemplu

- Adunarea a  $n$  numere
- Daca  $n =$  putere a lui 2  $\Rightarrow T_p = \log n$

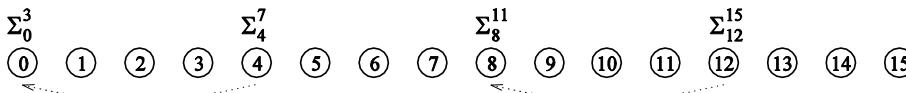
# Adunare – log n pasi



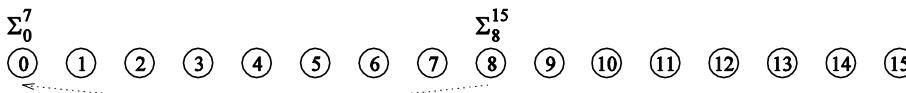
(a) Initial data distribution and the first communication step



(b) Second communication step



(c) Third communication step



(d) Fourth communication step



(e) Accumulation of the sum at processing element 0 after the final communication

$n=16; p=16s \dots$

## Exemplu (continuare)

=>

- $t_c = \text{timp calcul pt o operatie de adunare}$
- $T_{com} = t_s + t_w$  pt o operatie de comunicatie (per word)
  - sunt  $O(n)$  operatii de comunicatie dar si operatiile de comunicatie se pot executa simultan  $T_{com} = \Theta(\log n)$

$$T_p = \Theta(\log n)$$

- Stim ca  $T_s = \Theta(n)$
- Speedup  $S = \Theta(n / \log n)$  ( $p=n \Rightarrow E = \Theta(1 / \log n)$ )

# Accelerare – superliniara?

- $S = 0$  (programul nu se termina niciodata).
- $S < p$  (teoretic)
  - În caz contrar :

oricare procesor ar fi implicat in calcule pentru rezolvarea problemei un timp

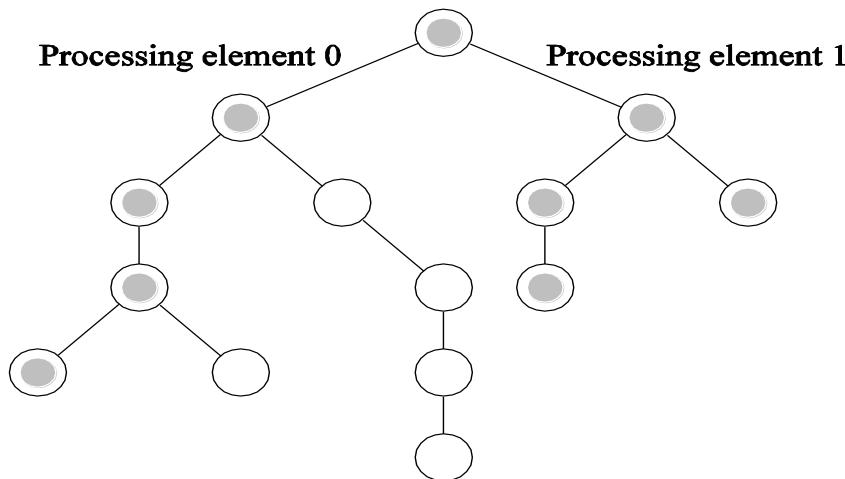
$$T < T_s / p$$

*in conditiile in care  $T_s$  depinde de numarul de operatii (N) care se efectueaza*

$$T < (N/p)$$

*=> se efectueaza mai putine operatii in total*

# Superlinear Speedups



Cautare intr-un arbore nestructurat.

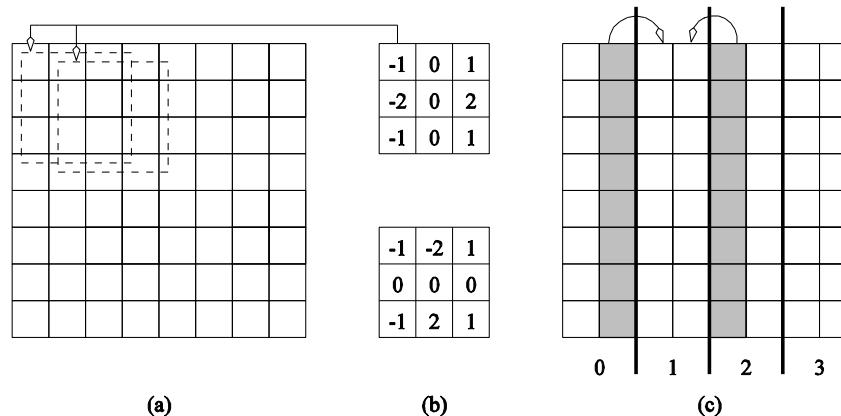
# Exemplu

Problema : *filtrare imagini = model distribuit*

- modificari pe celule de  $3 \times 3$  pixeli.

Daca o operatie aritmetica necesita pentru calcul  $t_c$ ,

timpul serial pt o imagine de  $n \times n$  este  $T_s = 9t_c n^2$ .



- Partitionare verticala =>  $n^2 / p$  pixels.
- Marginea fiecarui segment =>  $2n$  pixels.
- Nr de valori care trebuie comunicate =  $2n \Rightarrow 2(t_s + t_w n)$ .

*Timp de comunicatie:*  $t_{com} = t_s + t_w * (\text{message\_size})$   
 $(t_s - \text{timp de start al unei comunicatii})$

*calculul executat de fiecare proces:*

$$T_p^s = 9 t_c n^2 / p$$

# Evaluare metrici

- Timpul paralel:

$$T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

- Accelerarea si eficienta:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

# Costul

- Costul se defineste ca fiind produsul dintre timpul de executie si numarul maxim de procesoare care se folosesc.

$$C_p(n) = t_p(n) \cdot p$$

- Aceasta definitie este justificata de faptul ca orice aplicatie paralela poate fi simulata pe un sistem secvential, situatie in care unicul procesor va executa programul intr-un timp egal cu  $O(C_p(n))$ .
- O aplicatie paralela este optima din punct de vedere al costului, daca valoarea acestuia este egala, sau este de acelasi ordin de marime cu timpul celei mai bune variante secventiale ---  $C_p = O(t_s)$ .
- aplicatia este eficienta din punct de vedere al costului daca  $C_p = O(t_s \log p)$ .

# Costul unui sistem paralel (algoritm +sistem)

- Cost =  $p \times T_p$
- Costul reflecta suma timpului pe care fiecare procesor il petrece in rezolvarea problemei.
- Un sistem paralel se numeste optimal daca costul rezolvării unei probleme pe un calculator paralel este asymptotic egal cu costul serial.
- $E = T_s / p T_p \Rightarrow$  pentru sisteme cost optimal  $\Rightarrow E = O(1)$ .
- Cost  $\sim work \sim processor\text{-}time product$ .

## Exemplu: Adunare n numere pe un model distribuit

- $T_p = (t_c + t_{com}) \log n$  (pt  $p = n$ ).

$t_c$  timpul necesar unei operatii de adunare;

$t_{com}$ - timpul necesar unei operatii de comunicatie;

- $C = p T_p = O(n \log n)$
- $T_s = \Theta(n) \Rightarrow$  nu este cost optimal
- Cum se poate optimiza?
  - se micsoreaza  $p$ ;  $p=n/k$
  - se considera  $p$  segmente ( $\text{dim} = n/p$ ) pentru care se calculeaza suma sequential
  - se foloseste calculul de tip arbore pentru insumarea celor  $p$  sume locale
  - $T_p = t_c n/p + (t_c + t_{com}) \log p$
  - $C = t_c n + (t_c + t_{com}) p * \log p \Rightarrow$  daca  $p * \log p = O(n)$  atunci cost optimal

# Scalabilitate

- **Scalabilitatea este un parametru calitativ care caracterizeaza atat sistemele paralele (numar de procesoare, unitati de memorie) cat si aplicatiile paralele!**

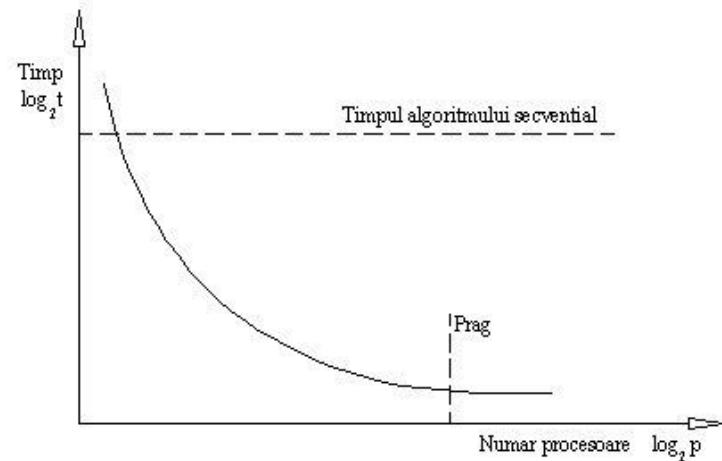
## Scalabilitatea aplicatiilor paralele

- *Un program se poate scala a.i. sa foloseasca mai multe procesoare?*
  - Adica???
  - Cum se evaluateaza scalabilitatea?
  - Cum se evaluateaza beneficiile aduse de scalabilitate?
- Evaluare performantei:
  - Daca se dublaaza nr de procesoare la ce ar trebui sa ne asteptam?
  - Cu cat trebuie sa creasca dimensiunea problemei daca dublam numarul de procesoare astfel incat sa obtinem aceeasi eficienta?
  - Este scalabilitatea liniara? (raport  $p_1/p_2 = n_1/n_2$ )
- Evaluarea eficientei corespunzatoare
  - Se pastreaza eficienta pe masura ce creste dimensiunea problemei?
    - cat de mult trebuie sa creasca p?

**Scalabilitate aplicatiei: abilitatea unui program paralel sa obtina o crestere de performanta proportionala cu numarul de procesoare si dimensiunea problemei.**

# Scalabilitate

- **Scalabilitatea masoara modul in care se schimba(creste) performanta unui anumit algoritm in cazul in care sunt folosite mai multe elemente de procesare.**
- Un indicator important pentru aceasta este **numarul maxim de procesoare** care pot fi folosite pentru rezolvarea unei probleme.
- In cazul posibilitatii folosirii unui numar mic de procesoare, un program paralel se poate executa mult mai incet decat un program secvential.
  - Diferenta poate fi atribuita comunicatiilor si sincronizariilor care nu apar in cazul unui program secvential. (Overhead)
- Numarul minim de componente pentru o anumita partitionare, poate fi de asemenea un indicator important.



# Scalabilitate

## Definitie

- *Scalabilitatea unui sistem paralel este o masura a capacitatii de a livra o accelerare cu o crestere liniara in functie de numarul de procesoare folosite.*
- Analiza scalabilitatii se face pentru un **sistem (arhitectura + algoritm)**.
- *Evidentiaza cum se extrapoleaza performanta de la probleme si sisteme mici  
=>  
la probleme si configuratii mai mari.*
- Metrici pentru scalabilitate –
  - functia de isoeficienta (isoefficiency)
  - eficienta Isospeed –efficiency
  - Fractia seriala/Serial Fraction  $f$

# weak vs strong scaling analysis

- In weak scaling analysis, we evaluate the speedup, efficiency or the running time of a parallel algorithm in points  $(n, p)$  where we ensure that the problem size per processor remains constant.
  - A common practice in weak scaling analysis consists in doubling both the size of the problem and the number of processors.
  - If the running time or the efficiency remains constant, then the algorithm is scalable.
- In strong scaling analysis, we are interested in determining how far we can remain efficient given a fixed problem size.
  - Therefore, for a fixed problem size, we increase the number of processors until we observe the changes in the efficiency.

# Exemple

- Suma de 2 vectori (memorie partajata) = scalabilitate foarte buna  
 $n$  operatii independente  $\Rightarrow p_{\maxim} = n$   
daca  $n$  creste si  $p$  poate creste  $\Rightarrow$  eficienta ramane la fel

Ideal

$$S = n/(n/p) = p; E = 1$$

(!!!daca se ignora timpul de creare threaduri/procese; distributia datelor!!!)

- Suma a  $n$  numere folosind  $p$  procesoare (model distribuit)  
 $S = n t_c / (t_c n/p + (t_c + t_{com}) \log p) = np t_c / (t_c n + (t_c + t_{com}) p \log p)$   
 $E = n t_c / (t_o n + (t_c + t_{com}) p \log p)$   
 $p_{\maxim} = n \Rightarrow E = n t_c / (t_c n + (t_c + t_{com}) n \log n)$

# Filtru pe imagine

- Timpul paralel:

$$T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

- Accelerarea si eficienta:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

- Daca  $p$  creste atunci eficienta scade – cat de mult?

$$(2t_w p n / 9t_c n^2) = (2t_w p / 9t_c n) < 1$$

- $p_{\maxim} = n \Rightarrow (9t_c n) / (9t_c n + 2t_s + 2n t_w) \quad (n \rightarrow \infty \Rightarrow E \sim 1)$

# Granularitate

- **Granularitatea** (“grain size”) este un parametru calitativ care caracterizeaza atat
  - sistemele paralele (numar de procesoare, unitati de memorie)
  - cat si
  - aplicatiile paralele.
- **Granularitatea aplicatiei** se defineste ca dimensiunea minima a unei unitati secentiale dintr-un program, exprimata in numar de instructiuni.
  - Prin unitate secentuala se intlege o parte din program in care nu au loc operatii de sincronizare sau comunicare cu alte procese.
- Fiecare flux de instructiuni are o anumita granularitate.
- Granularitatea unui algoritm poate fi aproximata ca fiind raportul dintre **timpul total calcul si timpul total de comunicare**.

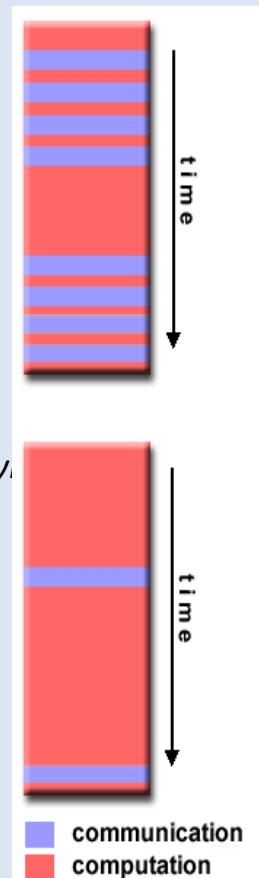
# Granularitatea sistemului

- Pentru un **sistem paralel** dat, există o valoare minima a granularitatii aplicatiei, sub care performanta scade semnificativ. Aceasta valoare de prag este cunoscută ca și **granularitatea sistemului** respectiv.
  - Justificarea constă în faptul că timpul de overhead (comunicații, sincronizări, etc.) devine comparabil cu timpul de calcul paralel.
- De dorit
  - => un calculator paralel să aibă o granularitate mică, astfel încât să poată executa eficient o gamă largă de programe.
  - ⇒ programele parallele să fie caracterizate de o granularitate mare, astfel încât să poată fi executate eficient de o diversitate de sisteme.
- Exceptii: clase de aplicații cu o valoare a granularitatii foarte mică, dar care se executa cu succes pe arhitecturi specifice.
  - aplicațiile sistolice, în care în general o operatie este urmata de o comunicație.
    - aceste aplicații impun însă o structură a comunicațiilor foarte regulată și comunicatii doar intre noduri vecine

# Granularity...cont.

- **Fine-grain Parallelism:**

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

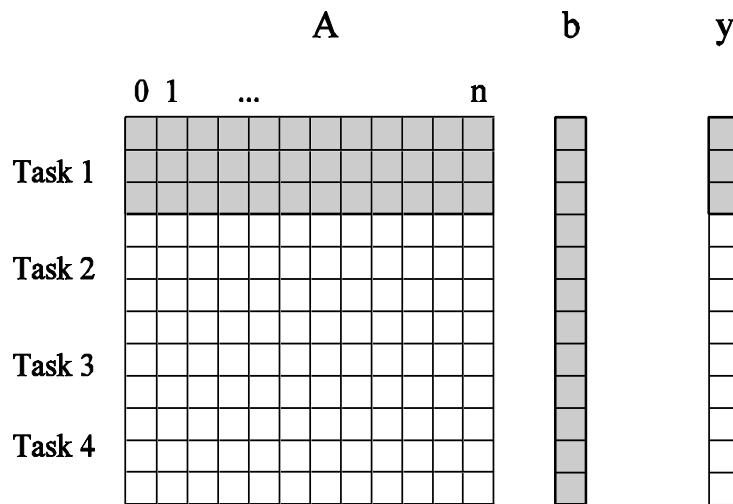


- **Coarse-grain Parallelism:**

- Relatively large amounts of computational work are done between communication/events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

# Granularitate $\Leftrightarrow$ descompunere in taskuri

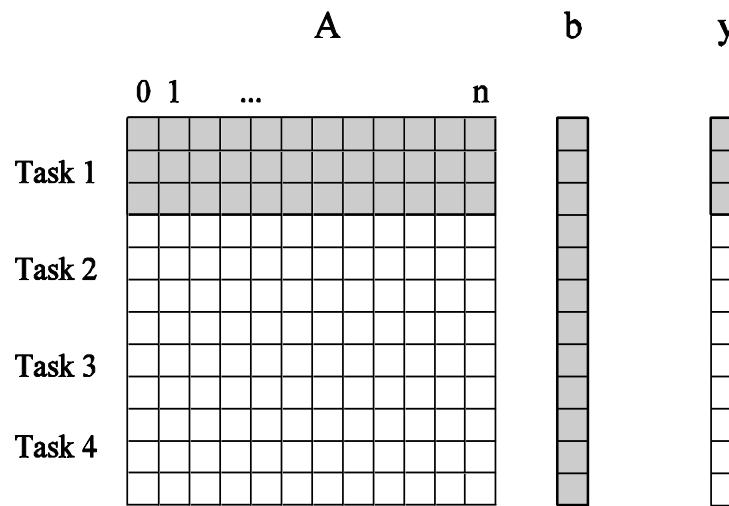
- Numarul de task-uri in care o problema se descompune determina granularitatea.
- numar mare  $\Rightarrow$  fine-grained decomposition
- numar mic  $\Rightarrow$  coarse grained decomposition



Exemplu: inmultire matrice

# Granularitea decompunerii taskurilor

- Granularitatea este **determinata** (*atentie – nu definită*) de numarul de taskuri care se creeaza pt o problema.
- Mai multe taskuri => granularitate mai mica



# Efectul granularitatii asupra performantei

- De multe ori, folosirea a mai putine procesoare imbunatatesta performanta sistemului parallel (ansamblu aplicatie+sistem).
- Folosind mai putine procesoare decat numarul maxim posibil se numeste ***scaling down (for a parallel system)***.
- Modalitatea naiva de scalare este de a considera fiecare element de procesare initiala fi unul virtual si sa se atribuie fiecare procesor virtual unui real (fizic).
- Daca numarul de procesoare scade cu un factor  $n / p$ , calculul efectuat de catre fiecare procesor va creste cu acelasi factor.
- Costul comunicatiei nu creste pentru ca comunicatia intre unele procesoare virtuale se va face in cadrul aceluiasi procesor (real).

# Gradul de paralelism (*DegreeOfParallelisation*)

- DOP al unui algoritm este dat de numarul de operatii care pot fi executate simultan.
  - fina – numar mare de operatii executate in paralel
  - medie
  - Bruta

Exemplu: suma numerelor dintr-un sir=>  $DOP = n/2$

# *DOP(Degree of Parallelism)*

- *Gradul de paralelism DOP* („degree of parallelism”) =
  - (al unui program)
    - numarul de procese care se executa in paralel intr-un anumit interval de timp.
    - Numarul de operatii care se executa in paralel intr-un anumit interval de timp
  - (al unui sistem)
    - numarul de procesoare care pot fi in executie in paralel intr-un anumit interval de timp
- *Profilul paralelismului* = graficul *DOP* in functie de timp (pentru un anumit program).

Depinde de:

- structura algoritmului;
- optimizarea programului;
- utilizarea resurselor;
- conditiile de rulare.

## In concluzie:

- accelerarea indica castigul de viteza de calcul intr-un sistem paralel;
- eficienta masoara partea utila a prelucrarii (lucrului) efectuate de  $n$  procesoare;
- costul masoara efortul necesar in obtinerea unui viteza de calcul mai mare
- scalabilitatea masoara gradul in care o aplicatie poate folosi eficient un numar mai mare de procesoare
- granularitatea depinde de raportul intre operatiile de calcul si cele de comunicatie/sincronizare

Referinte:

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar  
``Introduction to Parallel Computing'',

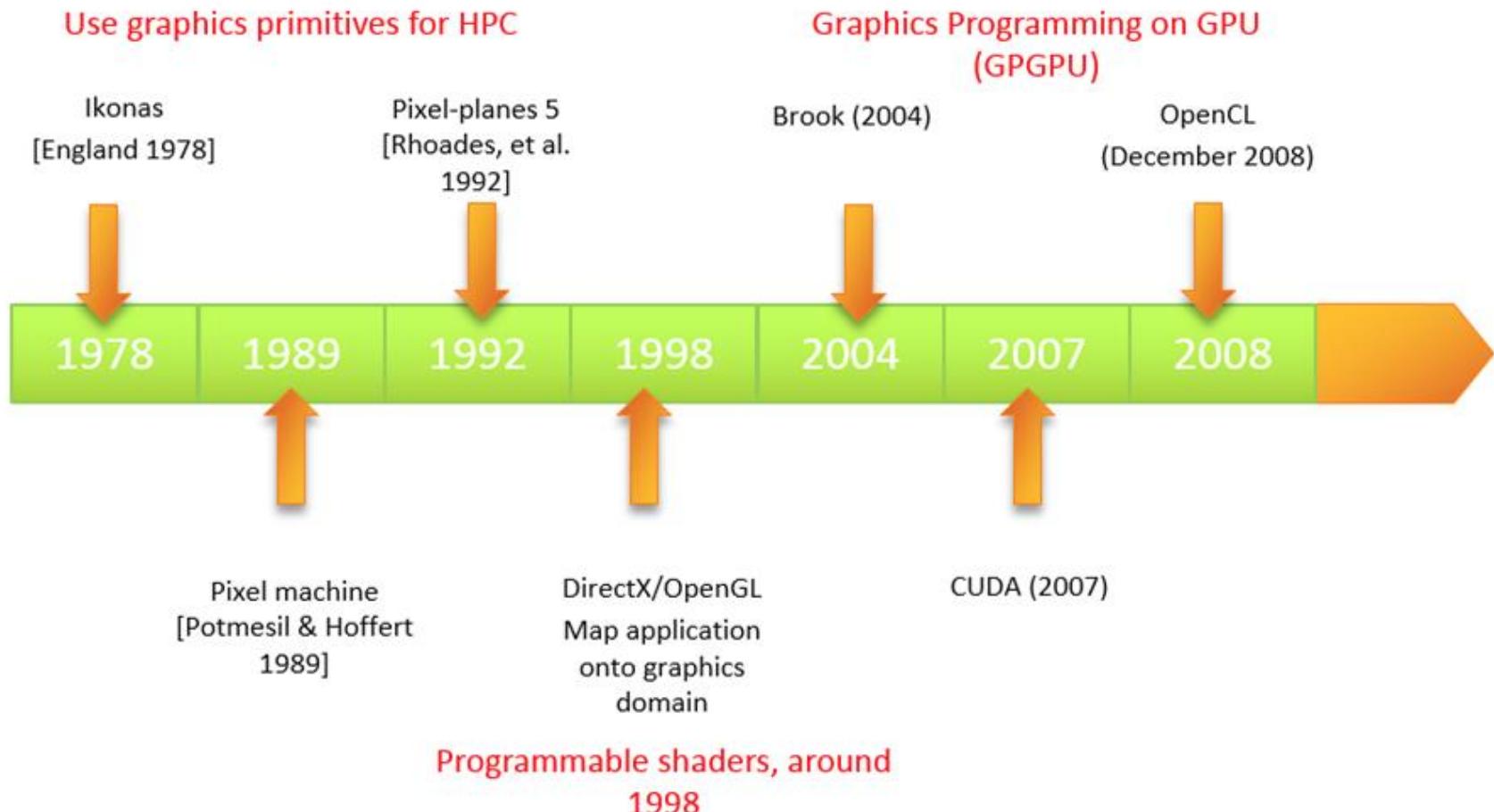
# Curs 11

## Introducere in CUDA

# Ce este CUDA?

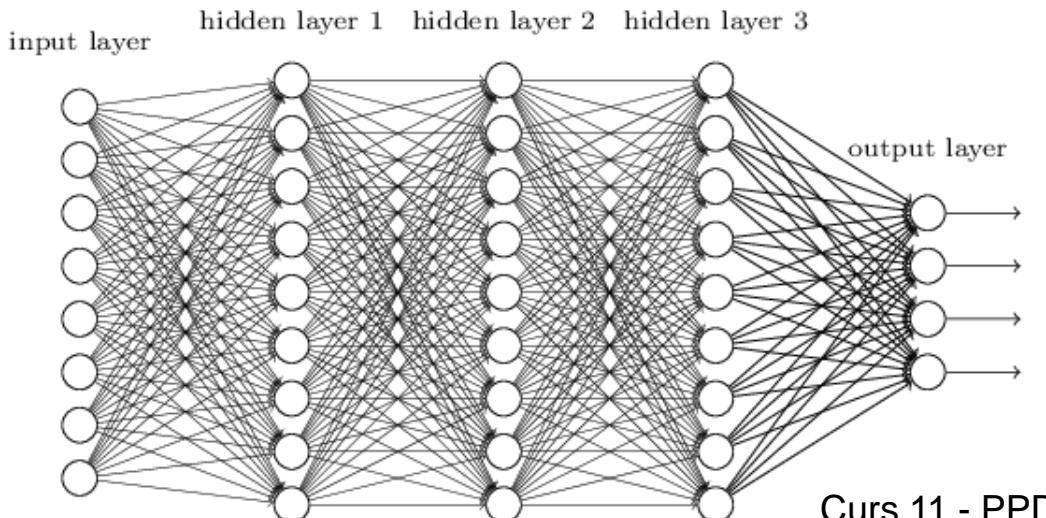
- Compute Unified Device Architecture"
- o platforma de programare paralela->
- Arhitectura care foloseste GPU pt calcul general
  - permite cresterea performantei
- Released by NVIDIA in 2007
- Model de programare
  - Bazat pe extensii C / C++ - pentru a permite ‘heterogeneous programming’
  - API pt gestionarea device-urilor, a memoriei etc.

# Istoric



# Aplicatii

- Bioinformatica
- Calcul financiar
- Deep learning
- Molecular dynamics simulation
- Video and audio coding and manipulation
- 3D imaging and visualization
- Consumer game physics
- virtual reality products
- ...



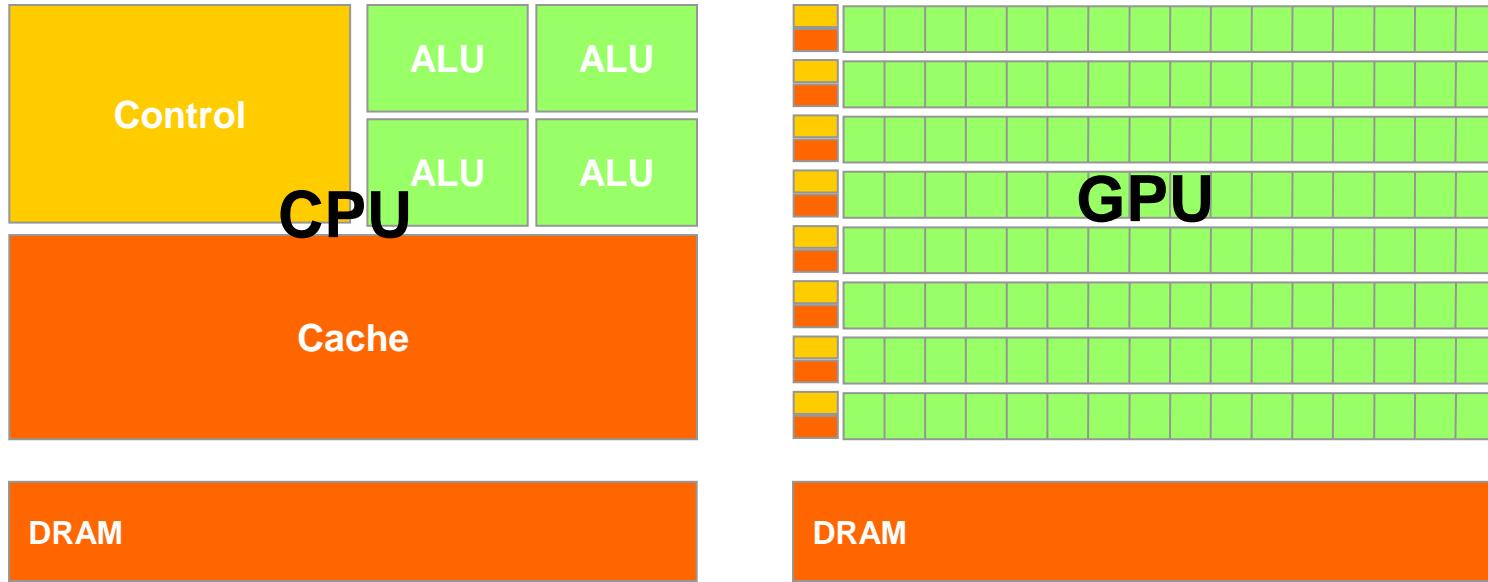
Curs 11 - PPD



# GPGPU

- GPU au devenit mai puternice
  - Mai multă putere de calcul
  - Memory bandwidth (on chip) ridicată
- General Purpose GPU (GPGPU)
- Sute de mii de core-uri în GPU care rulează threaduri în paralel
- core-uri mai slabe dar ... multe...

# CPU vs. GPU



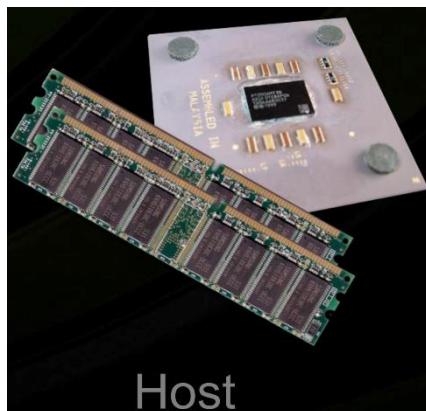
# Terminologie

Host: = CPU si memoria asociata

Device: = GPU si memoria asociata

- device

- Is a coprocessor to the CPU or **host**
- Has its own DRAM (**device memory**)
- Runs many **threads in parallel**
- Is typically a **GPU** but can also be another type of parallel processing device

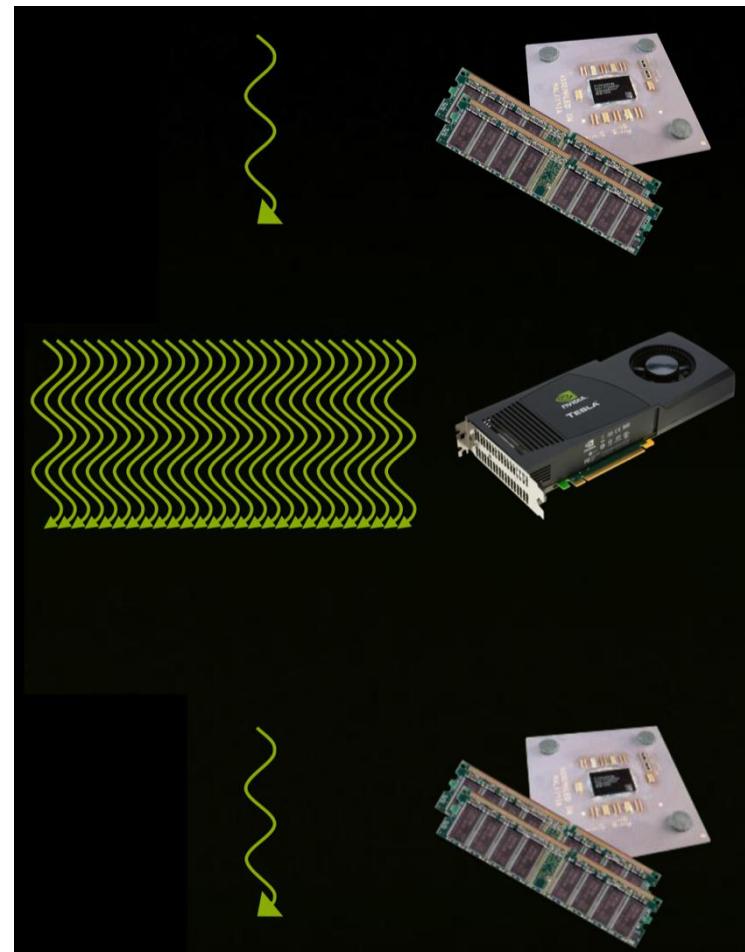
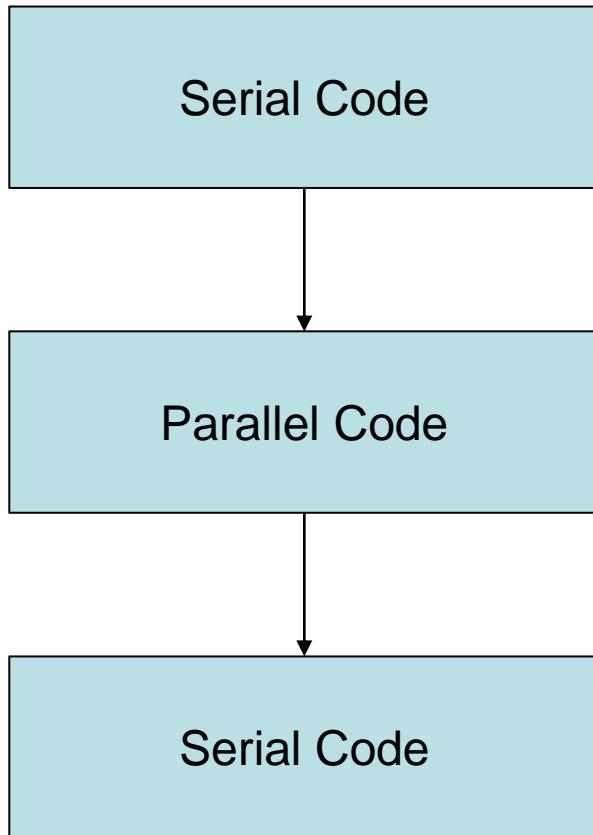


## Diferente intre threadurile GPU si CPU

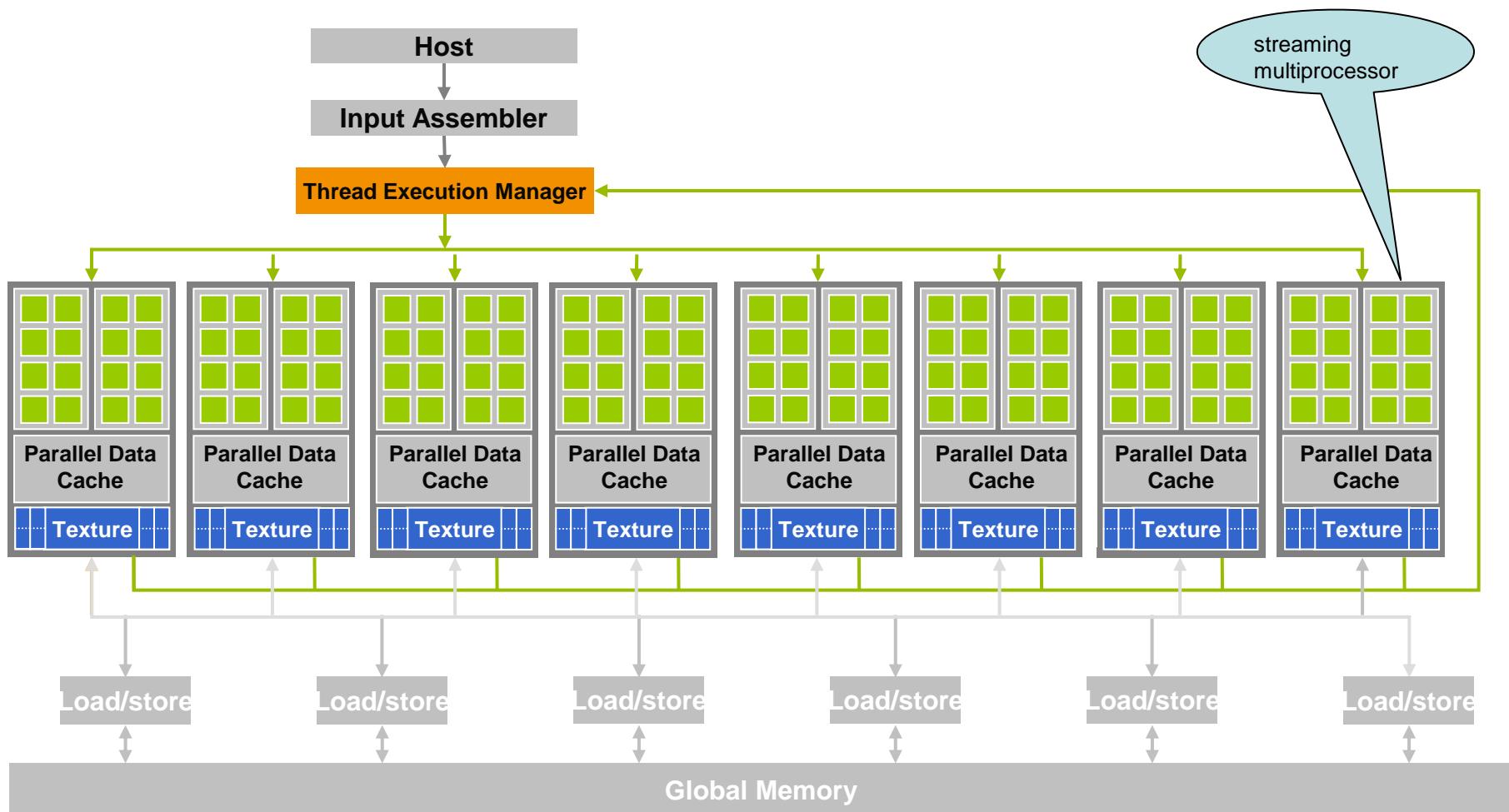
- GPU threads are extremely lightweight
  - Very little creation overhead
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few



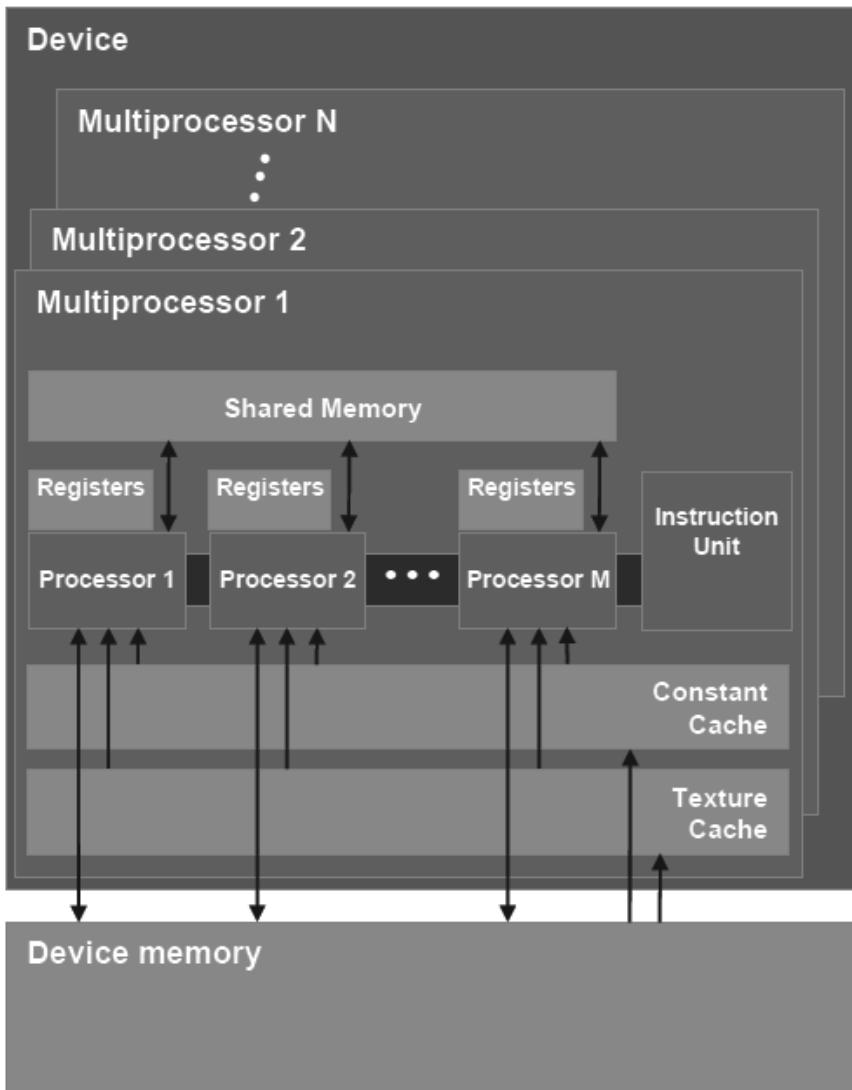
# Heterogeneous Computing



# Architecture of a CUDA-capable GPU



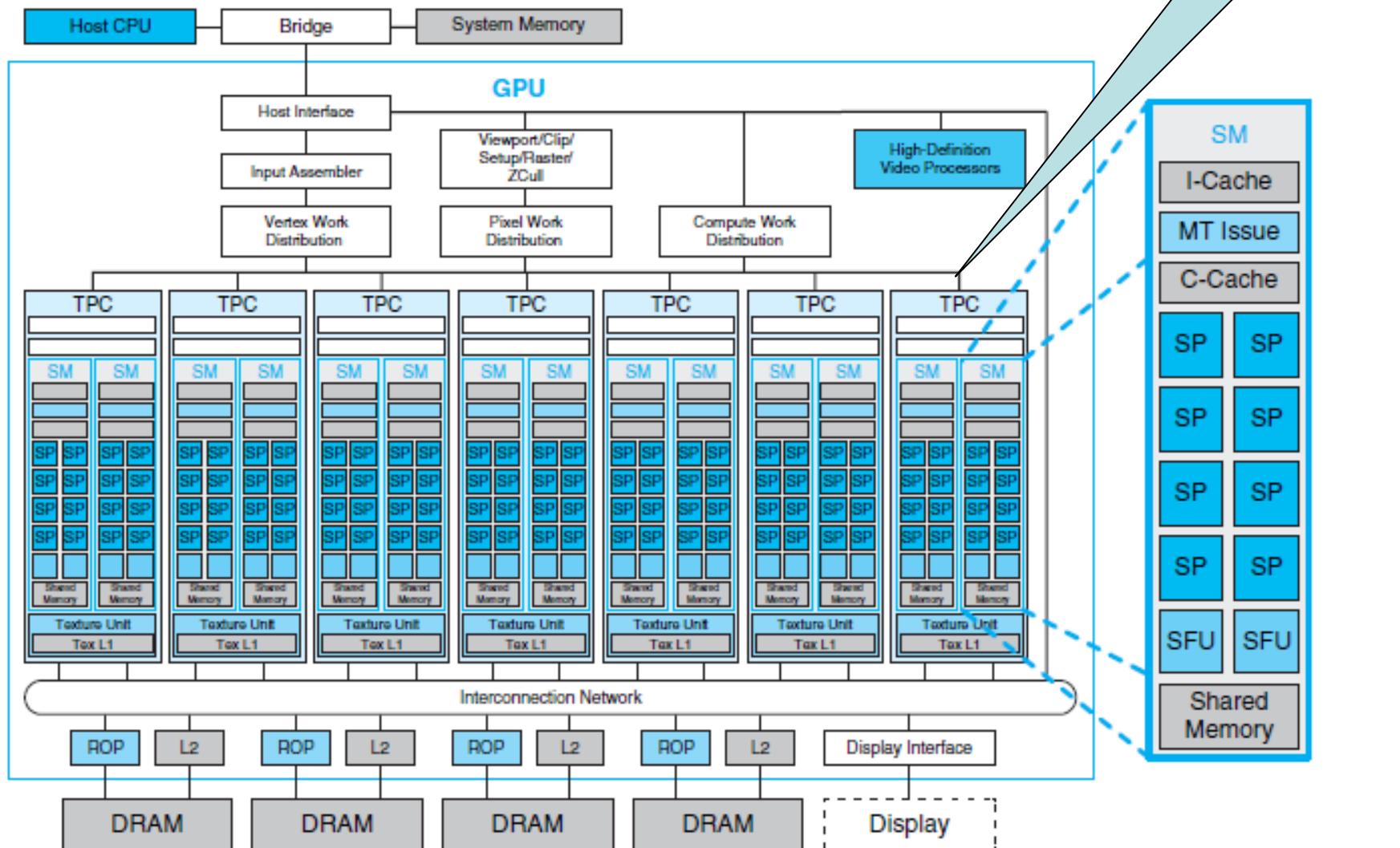
# GPU device

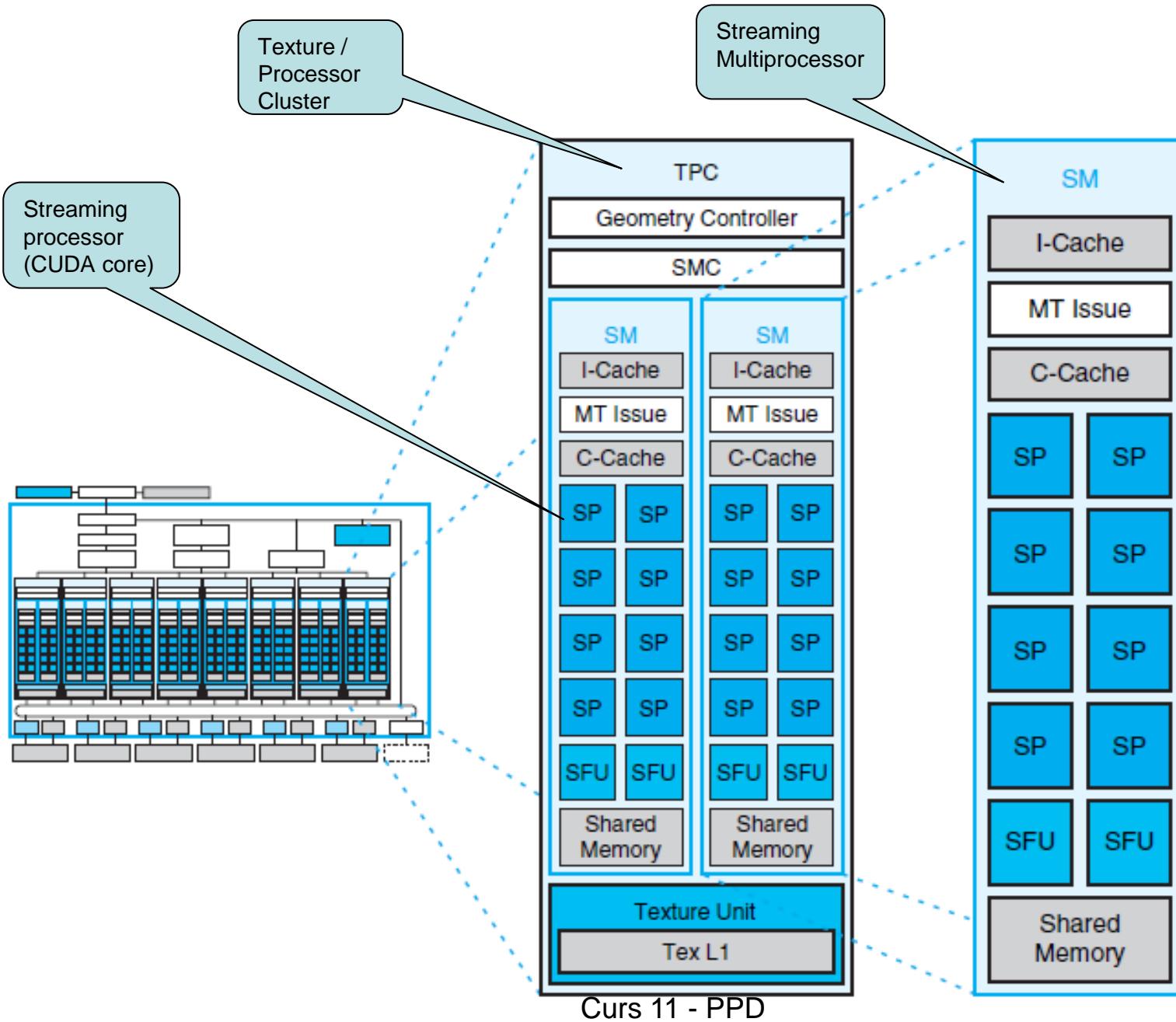


- Global memory
- Streaming Multiprocessors (SM) where each SM has:
  - Control units
  - Registers
  - Execution pipelines
  - Caches

<https://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environment/>

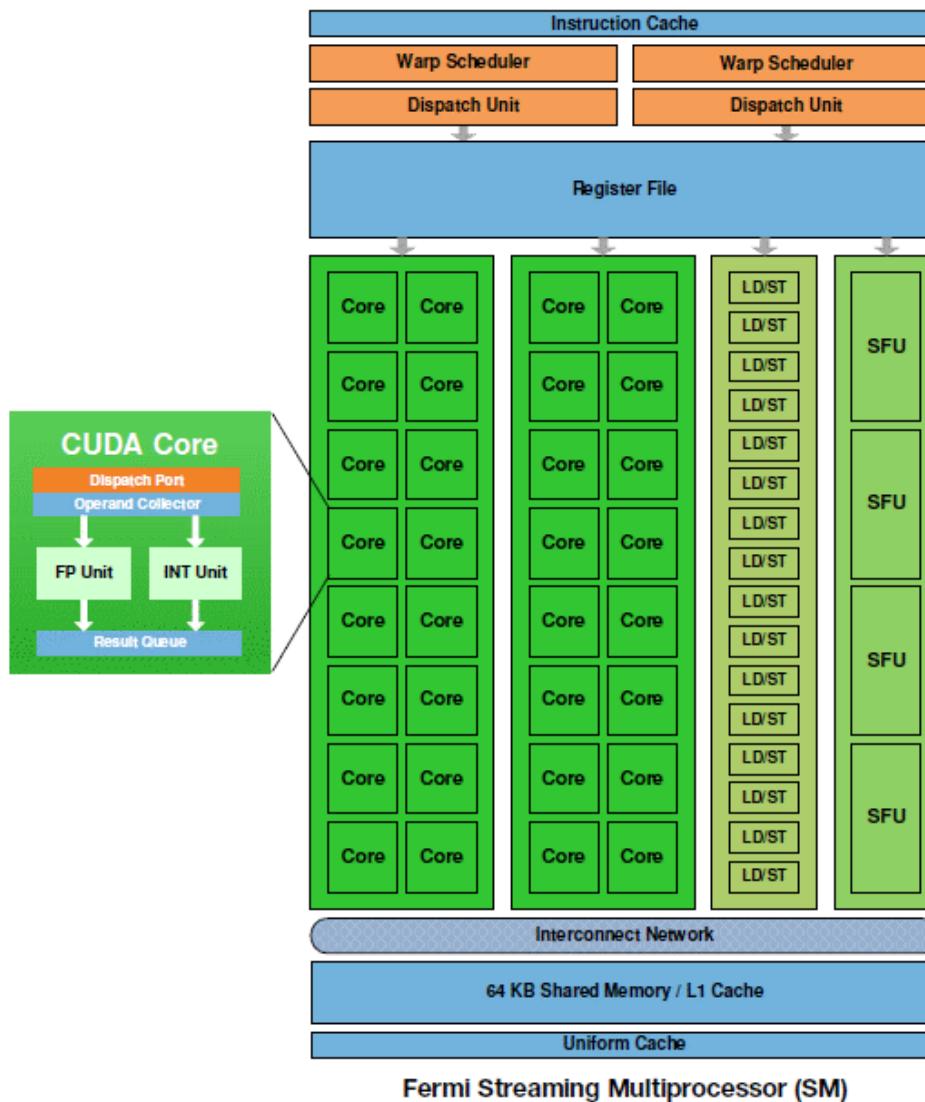
# GeForce 8800 Architecture





In Fermi architecture,  
a SM is made up of two SIMD  
16-way units.

each SIMD 16-way has 16 SPs  
=> a SM in Fermi has 32 SPs or  
32 CUDA cores



# Hardware Requirement

Cerinte pt GPU

- CUDA-capable GPU
  - Lista device-urilor acceptate: <https://developer.nvidia.com/cuda-gpus>
- Instalare-> documentatie
  - <http://docs.nvidia.com/cuda/>

# Fluxul de procesare

- Se copiaza datele in memoria GPU
- Se executa calcul in GPU (mii de threaduri)
- Se copiaza datele din memoria GPU in memoria host
- Kernel: codul GPU care se va executa

# Extended C

- **Declspecs**

- **global, device, shared, local, constant**

```
__device__ float filter[N];
// a global variable in the GPU, not the CPU.

__global__ void convolve (float *image) {

    __shared__ float region[M];
    ...

    region[threadIdx] = image[i];

    __syncthreads()

    ...

    image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

- **Runtime API**

- **Memory, symbol, execution management**

- **Function launch**

# CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function

**!!! Must return `void`**

`__device__` functions cannot have their address taken

## \_\_device\_\_ and \_\_host\_\_

- A routine decorated with **host** instructs the compiler to generate a host-callable entry point (i.e. compile it as host code). Such a routine is host code that can only be called from other host code.
- A routine decorated with **device** instructs the compiler to generate a device-callable entry point (i.e. compile it as device code). Such a routine is device code that can only be called from other device code.
- The only situation where device code can be “called from host code” is the kernel launch, which must be decorated with **global**
- A routine with none of the above decorations is treated by nvcc implicitly as if it were decorated with **host**
- You can use both. If you use both, order does not matter. If you use both, the compiler generates both types of routines described as above, one with a device-callable entry point, and one with a host-callable entry point.

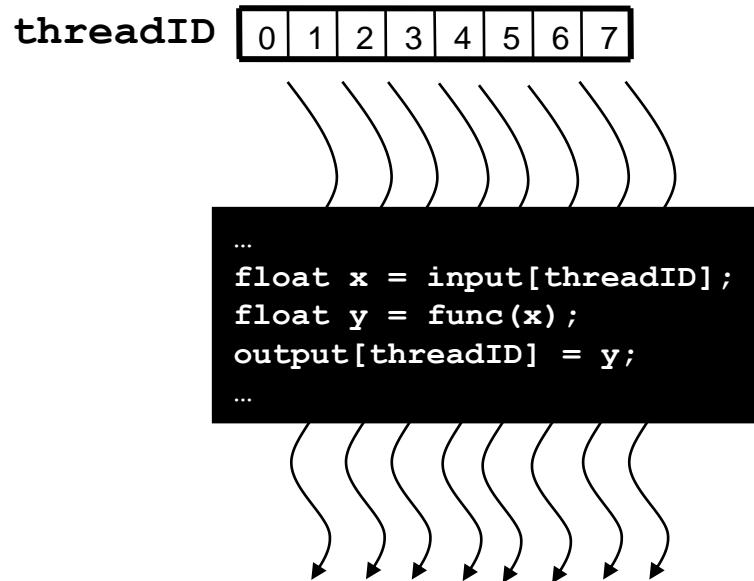
```
class example
{
public:
float a, b;
__device__ __host__ example(float _a, float _b) : a(_a), b(_b) {};
}
```

# CUDA Function Declarations

- For functions executed on the device:
  - *No recursion*
  - *No static variable declarations inside the function*
  - *No variable number of arguments*

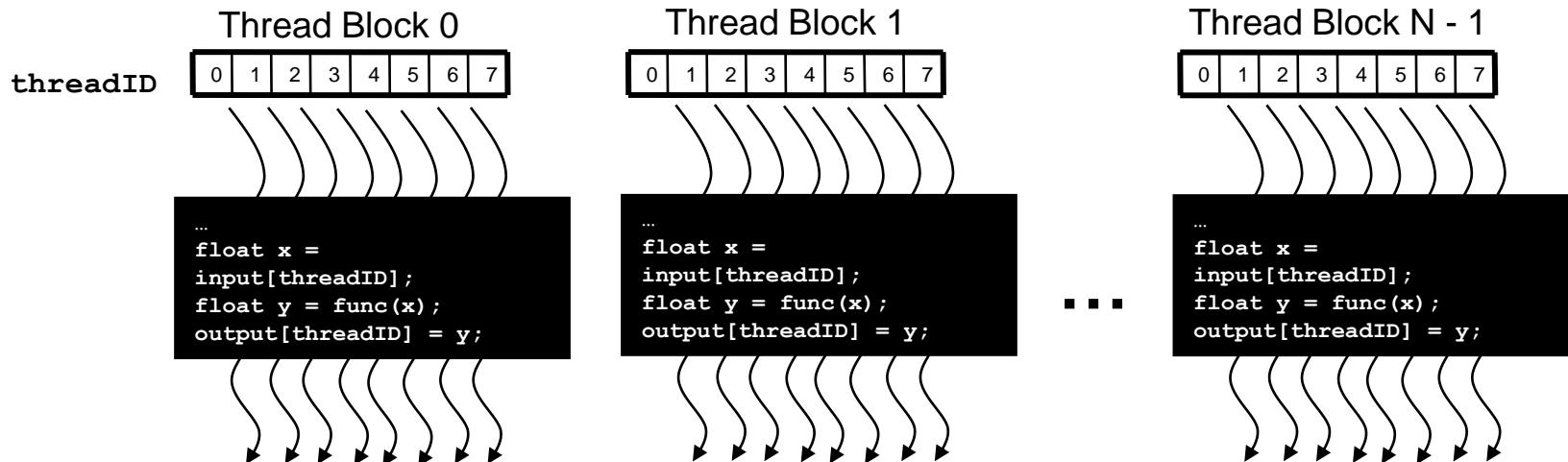
# Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID



# Thread Blocks: Scalable Cooperation

- Threadurile dintr-un bloc pot coopera via
  - **shared memory**
  - **atomic operations**
  - **synchronization barrier**



# Exemplu: Adunare vectori

## Parallel code: kernel

```
__global__ void vectorAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    // Make sure not to go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

# Cod: setup

```
int main( int argc, char* argv[] )  
{  
    // Size of vectors  
    int n = 1<<20;  
    size_t no_bytes= n*sizeof(int);  
    // Host input vectors  
    double *h_a;  double *h_b;  
    //Host output vector  
    double *h_c;  
    // h_a and h_b....[...] ... allocation and init  
    // Device input vectors  
    double *d_a;  double *d_b;  
    //Device output vector  
    double *d_c;  
    // Allocate memory for each vector on GPU  
    cudaMalloc(&d_a, no_bytes);  
    cudaMalloc(&d_b, no_bytes);  
    cudaMalloc(&d_c, no_bytes);  
  
    // Copy host vectors to device  
    cudaMemcpy( d_a, h_a, no_bytes, cudaMemcpyHostToDevice);  
    cudaMemcpy( d_b, h_b, no_bytes, cudaMemcpyHostToDevice);
```

# Cod: apelare kernel, colectare rezultate

```
int blockSize, gridSize;
// Number of threads in each thread block
blockSize = 1024;
// Number of thread blocks in grid
gridSize = (int)ceil((float) n / blockSize );
// Execute the kernel
vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

// Copy array back to host
cudaMemcpy( h_c, d_c, no_bytes, cudaMemcpyDeviceToHost );

// Release device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
//use h_c [...]
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}
```

```
><><><><><><><><><><
```

- Lansare kernel => executie \_\_global\_\_ function  
function <<< >>>

```
vectorAdd<<<3, 4>>>(d_a, d_b, d_c);
```

Functia Kernel

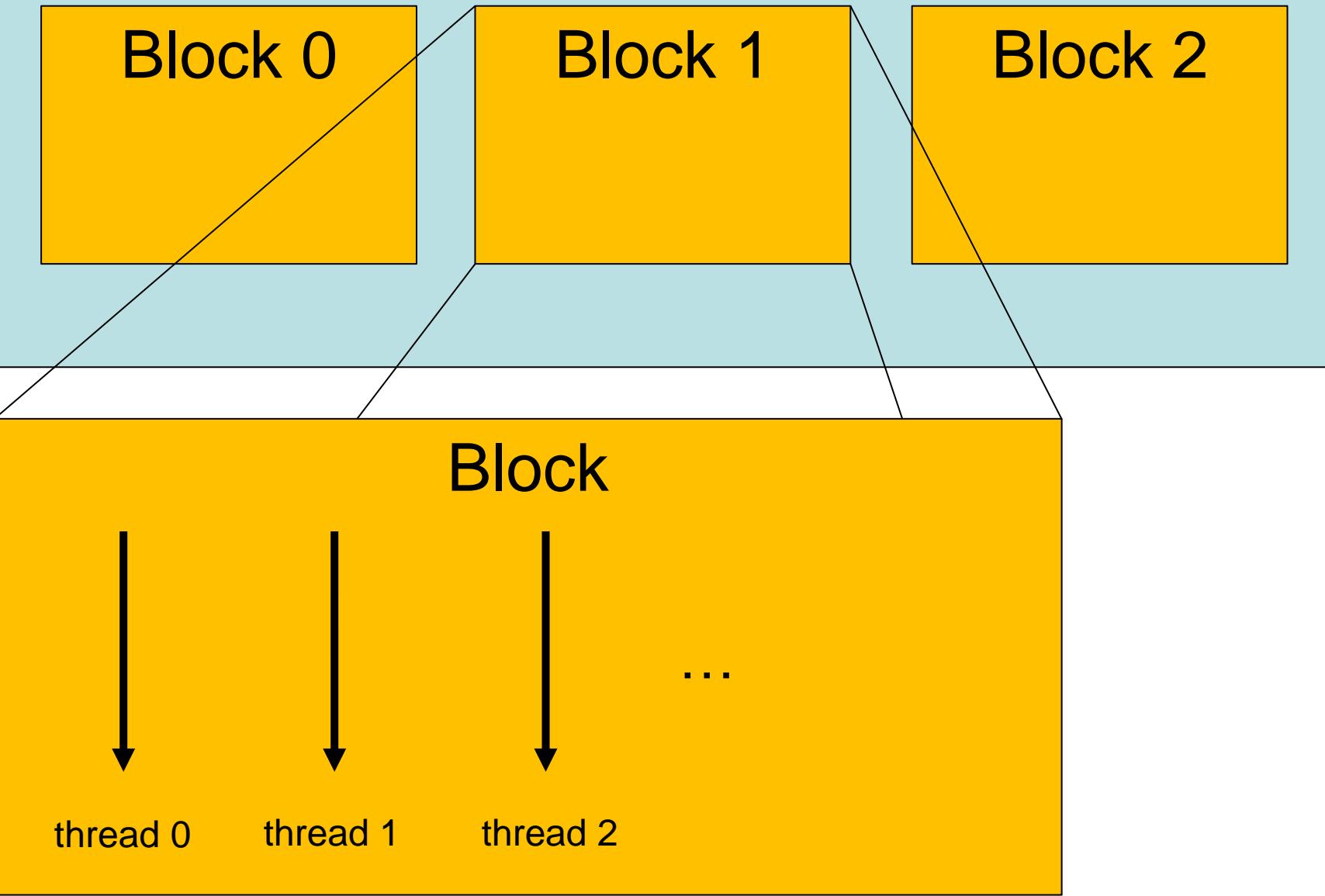
Cate blocuri si cate  
threaduri per bloc

parametrii

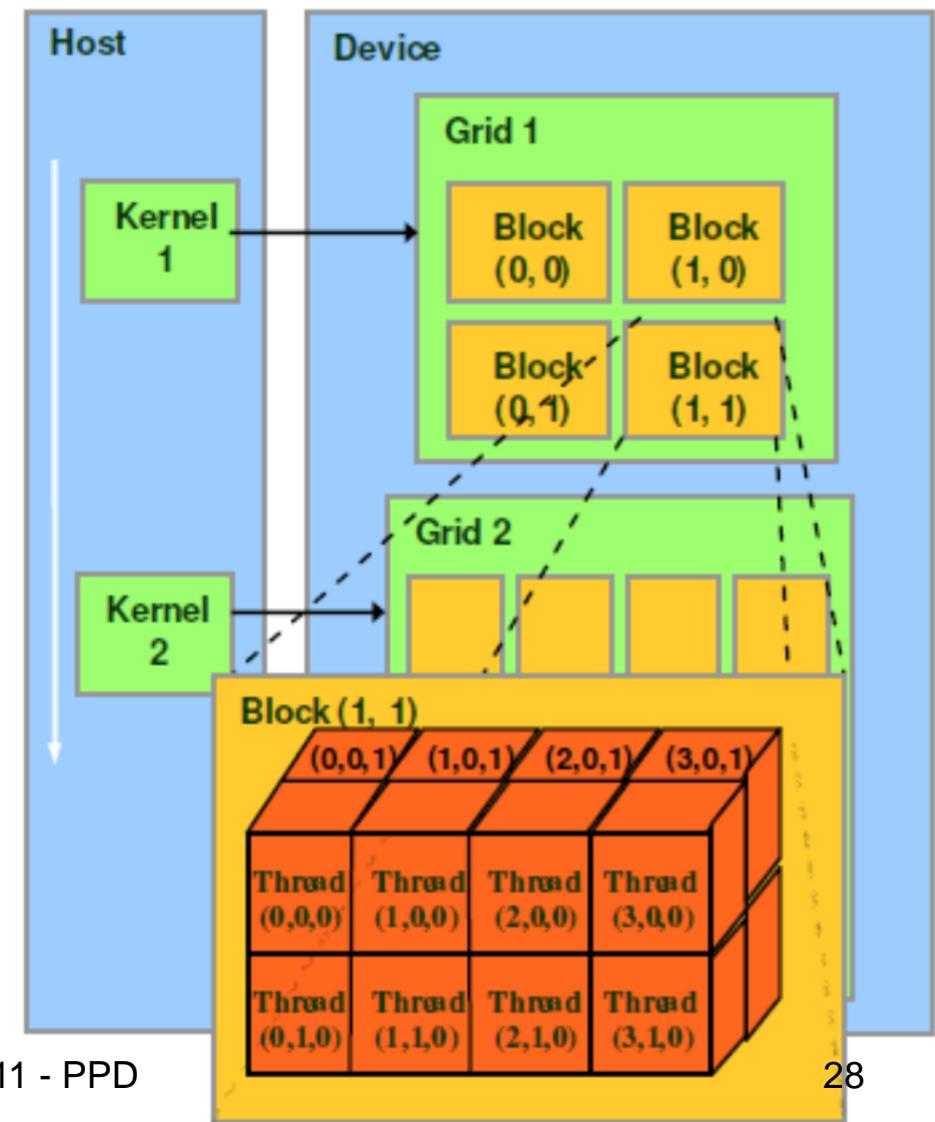
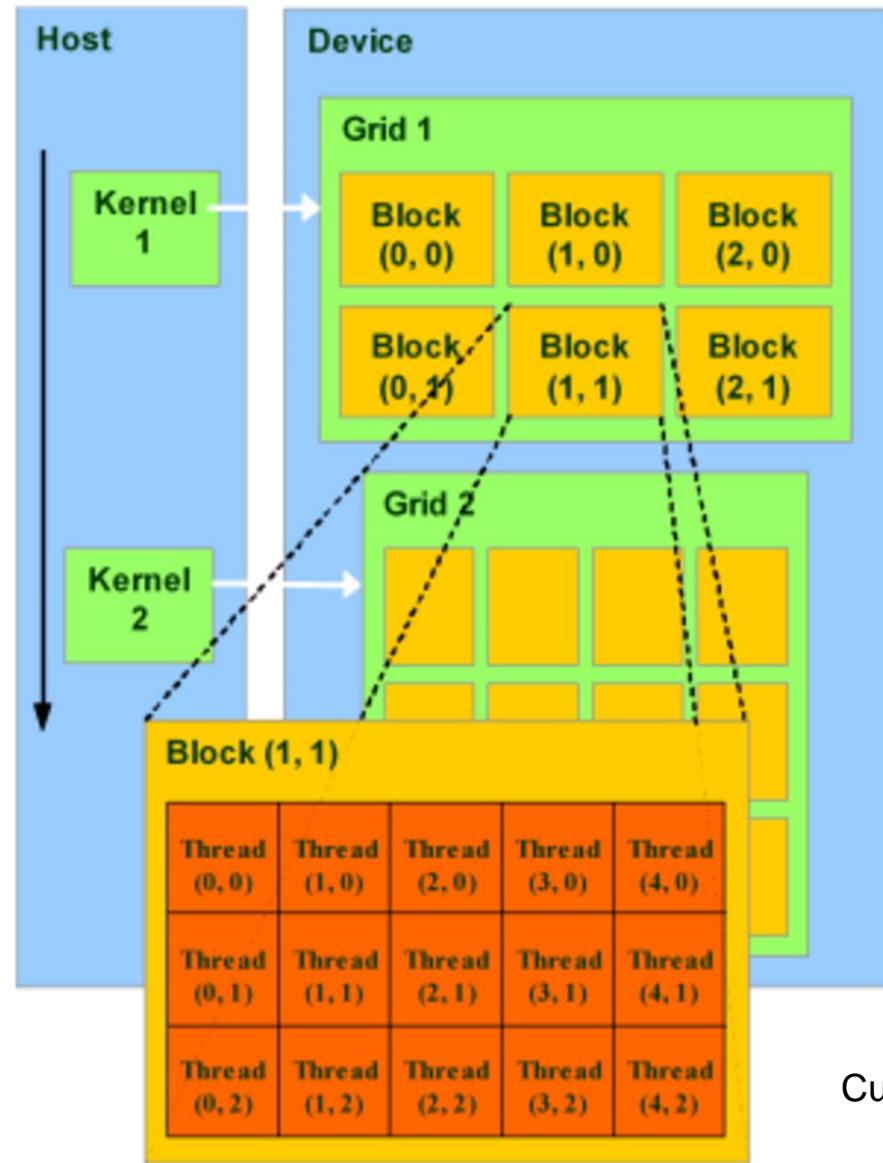
# Thread, Block, Grid

- CUDA foloseste o structura ierarhica :
  - grid
    - block
      - thread

# Grid



- Grid-ul poate sa fie compus din blocuri organizate 1D, 2D sau 3D
- Blocurile pot fi compuse din threaduri organizate 1D, 2D sau 3D



- <<<blocks per grid, threads per block>>>
  - <<<1, 1>>> : a grid with 1 block inside, and one block is consisted of 1 thread.  
Total threads: 1
  - <<<2, 3>>>: a grid with 2 blocks inside, and one block is consisted of 3 threads.  
Total threads: 6

# dim3 struct

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#ifndef __cplusplus
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Ex.

```
dim3 grid(256);           // defines a grid of 256 x 1 x 1 blocks
dim3 block(512,512);      // defines a block of 512 x 512 x 1 threads

foo<<<grid,block>>>(...);
```

# CUDA Built-In Variables

- `blockIdx.x`, `blockIdx.y`, `blockIdx.z` are built-in variables that return the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z` are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by *this* stream processor in *this* (where is called) particular block.
- `blockDim.x`, `blockDim.y`, `blockDim.z` are built-in variables that return the “block dimension” (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

The full global thread ID in x dimension can be computed by:

`x = blockIdx.x * blockDim.x + threadIdx.x;`

## Exemplul 1

- `BlockIdx.x` is the x number of the block
- `BlockDim.x` is the total threads in x dimension (width)
- If we launch `vector_add<<<2, 4>>>`
  - Primul thread (block(0), thread(0)):  
 $\text{idx} = 0 + 0 * 4 = 0$
  - Threadul al 5-lea (block(1), thread(0)):  
 $\text{idx} = 0 + 1 * 4 = 4$



# Exemplul 2

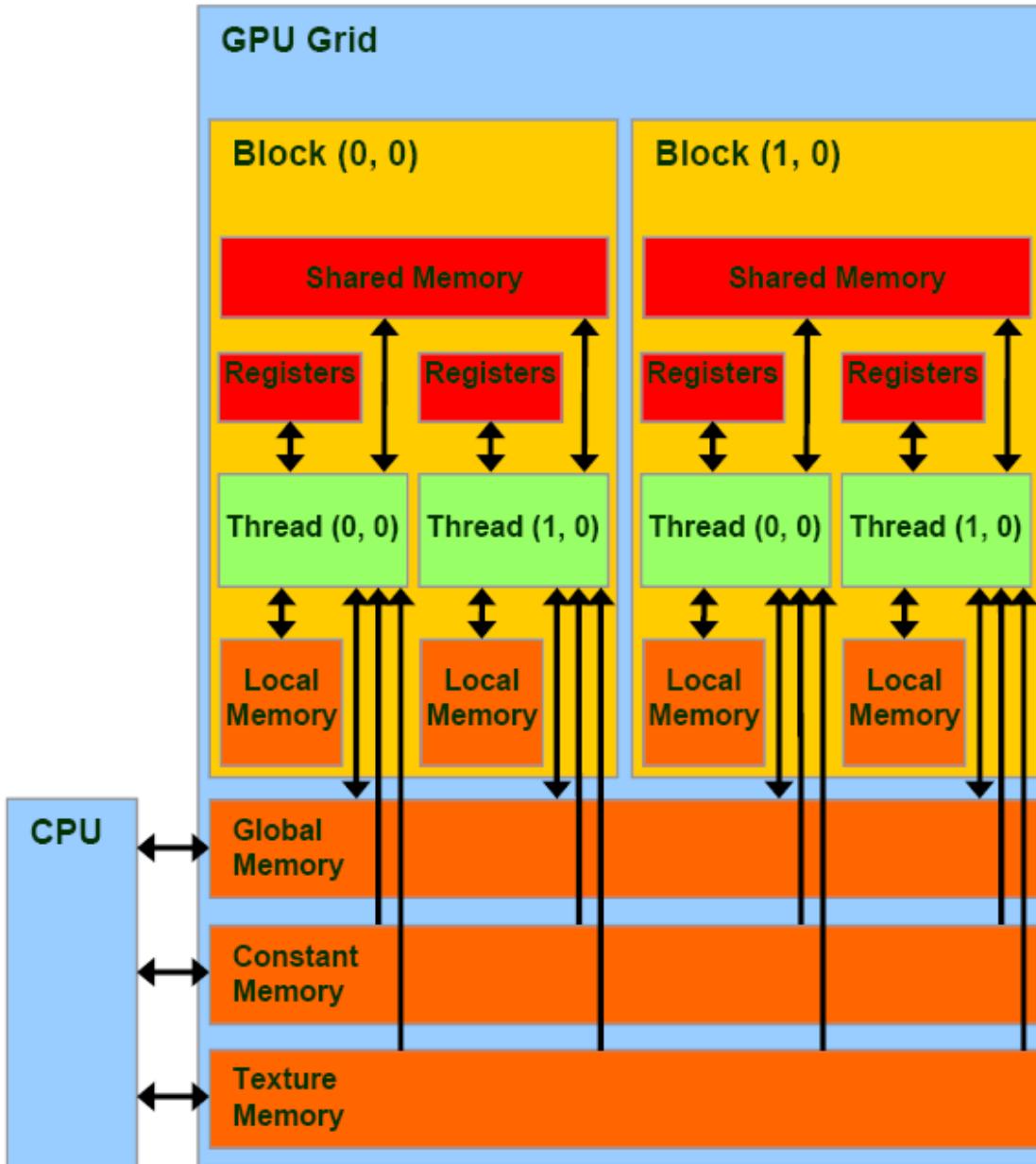
Global Thread ID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

- Assume a hypothetical ID grid and ID block architecture: 4 blocks, each with 8 threads.
- For Global Thread ID 26:
  - $\text{gridDim.x} = 4 \times 1$
  - $\text{blockDim.x} = 8 \times 1$
  - $\text{Global Thread ID} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
  - $= 3 \times 8 + 2 = 26$

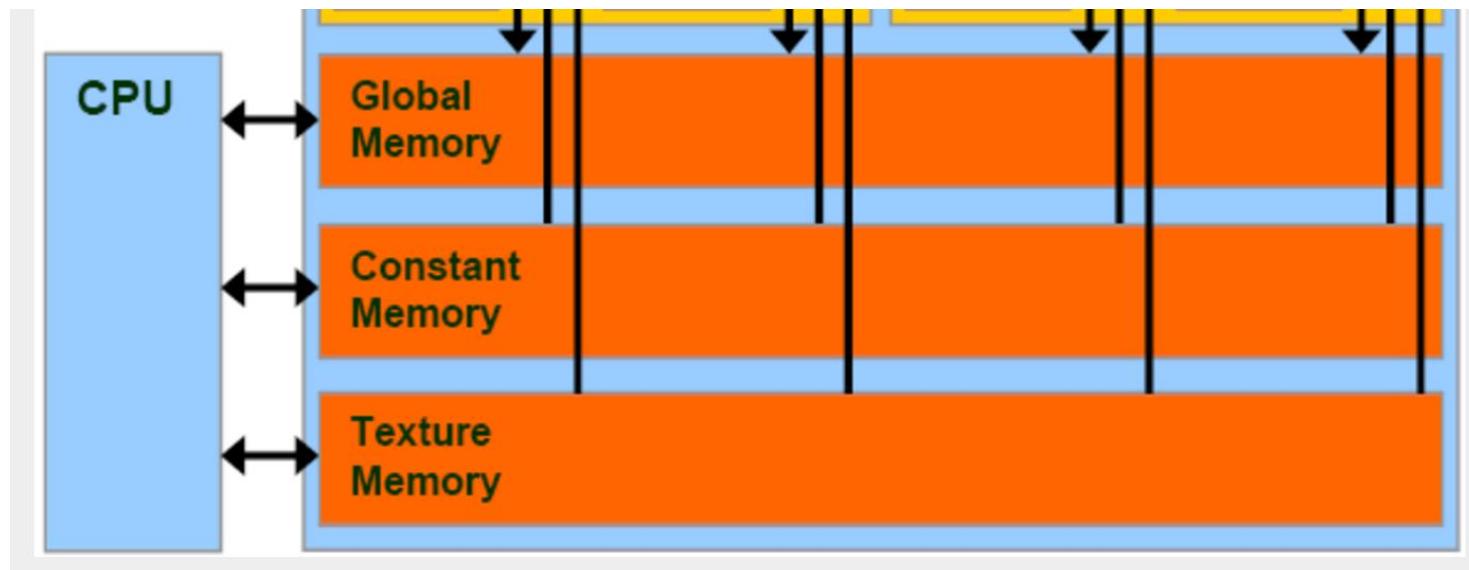
# Memory Types

- CUDA foloseste 5 tipuri de memorie fiecare cu proprietati diferite
- Proprietati:
  - Size
  - Access speed
  - Read/write, read only



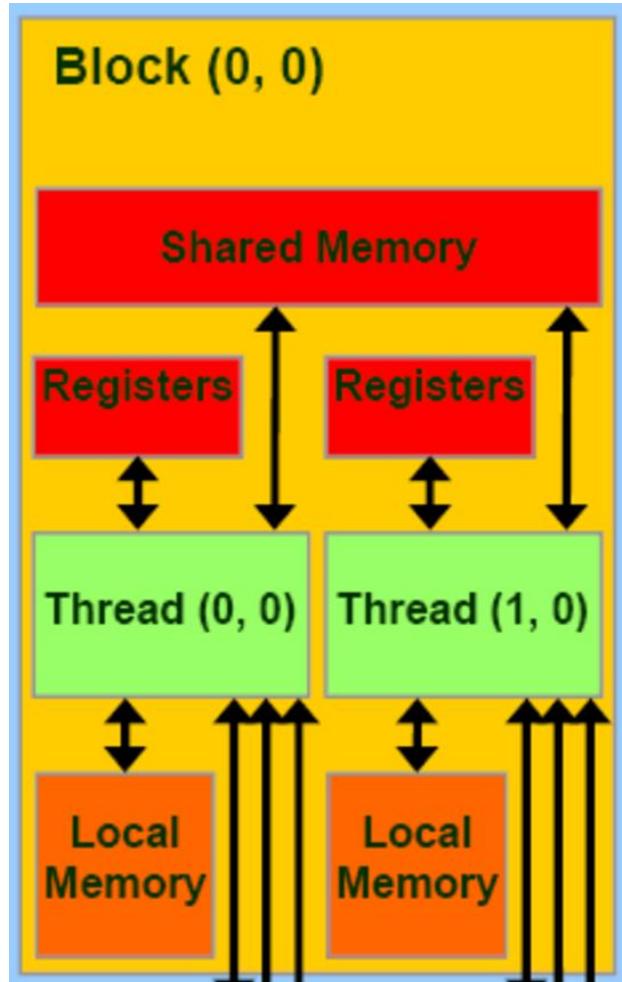
# Memory Types

- Global memory: `cudaMalloc` memory, the size is large, but slow
- Texture memory: read only, cache optimized for 2D access pattern
- Constant memory: slow but with cache (8KB)



# Memory Types

- Local memory:  
local to thread, but it is as slow as global memory
- Shared memory:  
100x fast to global  
memory, it is accessible  
to all threads in one  
block



# Memory Types

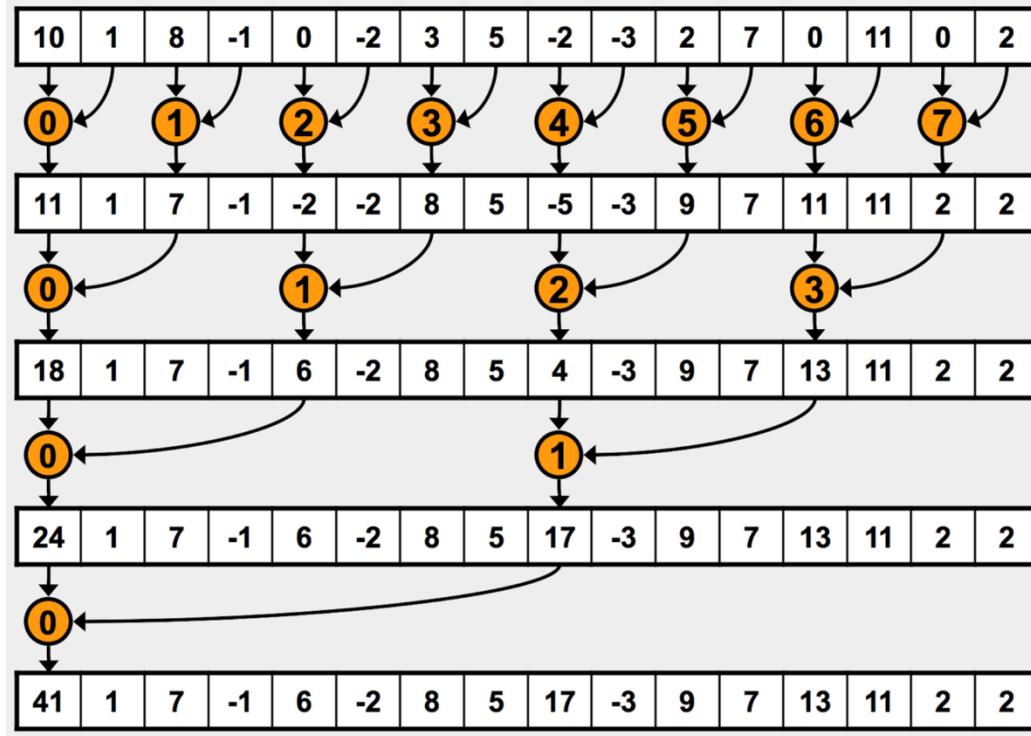
- Shared memory is very fast, but usually only 64KB.
- Actually, shared memory is the same as “L1 cache” of CPU, but controllable by user.
- One block has one shared memory, that’s one reason why we manage the threads in grid and block way!

# CUDA Memories characteristics

To summarize

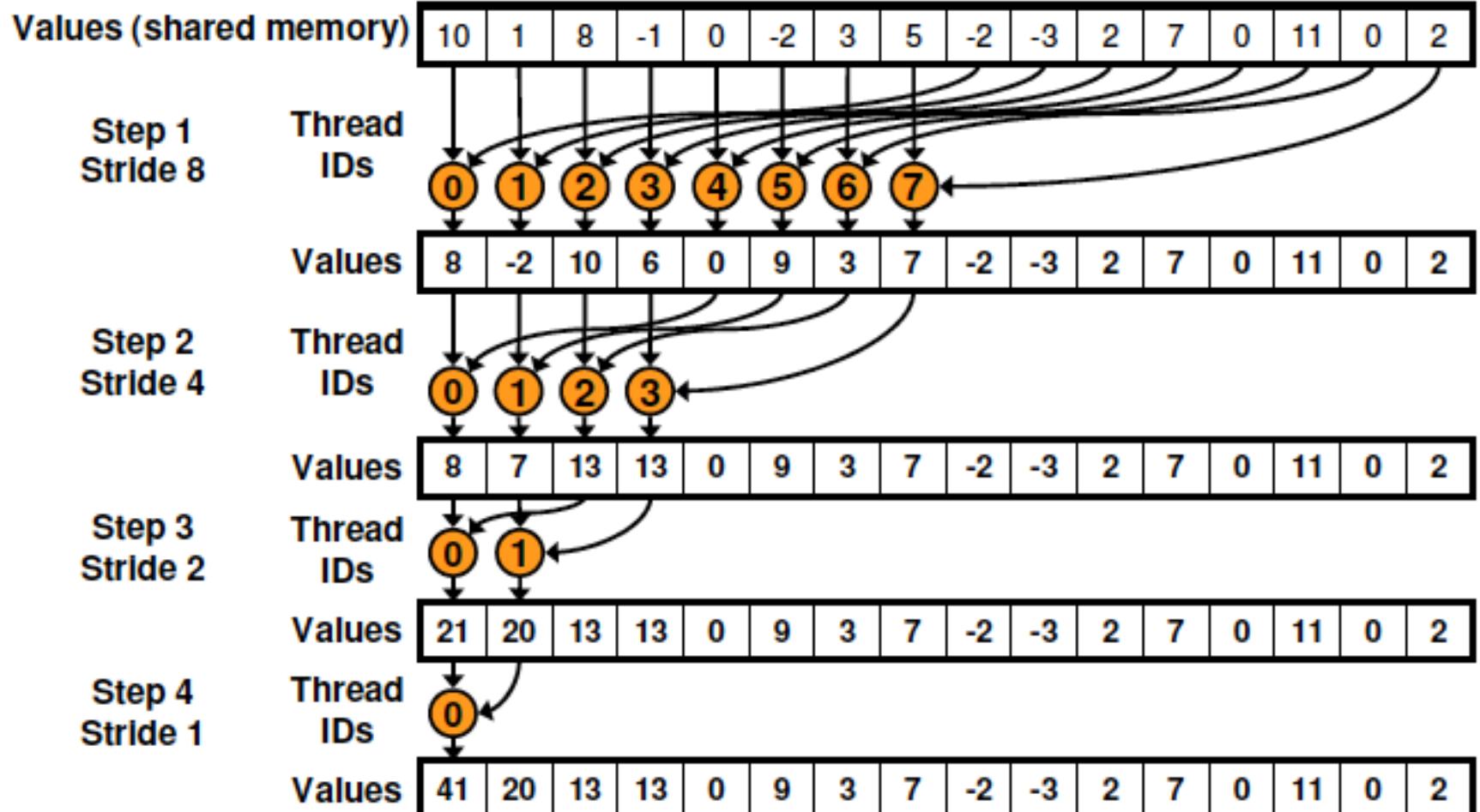
Registers	Per thread	Read-Write	
Local memory	Per thread	Read-Write	
Shared memory	Per block	Read-Write	For sharing data within a block
Global memory	Per grid	Read-Write	Not cached
Constant memory	Per grid	Read-only	Cached
Texture memory	Per grid	Read-only	Spatially cached

# Exemplu: Reduction



- !!! `__syncthreads()` is needed

# O alta varianta... dar similară *(better for CUDA)*



```

#define BLOCK_SIZE 512 // can be changed
#define NUM_OF_ELEMS 1024// can be changed
__global__ void total(float * input, float * output, int len) {
    // Load a segment of the input vector into shared memory
    __shared__ float partialSum[2*BLOCK_SIZE];
    int globalThreadId = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int t = threadIdx.x;
    unsigned int start = 2*blockIdx.x*blockDim.x;
    if ((start + t) < len) {    partialSum[t] = input[start + t];  }
    else {    partialSum[t] = 0.0;  }
    if ((start + blockDim.x + t) < len) {    partialSum[blockDim.x + t] = input[start + blockDim.x + t];  }
    else {    partialSum[blockDim.x + t] = 0.0;  }

    // Traverse reduction tree... (on each block)
    for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
    {
        __syncthreads();
        if (t < stride)    partialSum[t] += partialSum[t + stride];
    }
    __syncthreads();

    // Write the computed sum of the block to the output vector at correct index
    if (t == 0 && (globalThreadId*2) < len)
    {
        output[blockIdx.x] = partialSum[t];
    }
}

```

```

int main(int argc, char ** argv)
{
    int ii;
    float * hostInput; // The input 1D vector
    float * hostOutput; // The output vector (partial sums)
    float * deviceInput;
    float * deviceOutput;
    int numInputElements = NUM_OF_ELEMS; // number of elements in the input list
    int numOutputElements; // number of elements in the output list
    hostInput = (float *) malloc(sizeof(float) * numInputElements);
    //initialization
    for (int i=0; i < NUM_OF_ELEMS; i++) {
        hostInput[i] = i; // set the input values
    }
    numOutputElements = numInputElements / (BLOCK_SIZE<<1);
    if (numInputElements % (BLOCK_SIZE<<1)) { numOutputElements++; }
    hostOutput = (float*) malloc(numOutputElements * sizeof(float));
    //Allocate GPU memory
    cudaMalloc((void **) &deviceInput, numInputElements * sizeof(float));
    cudaMalloc((void **) &deviceOutput, numOutputElements * sizeof(float));
    // Copy memory to the GPU
    cudaMemcpy(deviceInput, hostInput, numInputElements * sizeof(float), cudaMemcpyHostToDevice);
}

```

```

// Initialize the grid and block dimensions here
dim3 DimGrid( numOutputElements, 1, 1); //numOutputElements = no of blocks!
dim3 DimBlock(BLOCK_SIZE, 1, 1);
//each block compute a local sum - results are stored into deviceOutput[numOutputElements]
//*****
// Launch the GPU Kernel here
total<<<DimGrid, DimBlock>>>(deviceInput, deviceOutput, numInputElements);
//*****
// Copy the GPU memory back to the CPU here
cudaMemcpy(hostOutput, deviceOutput, numOutputElements * sizeof(float), cudaMemcpyDeviceToHost);
/* Reduce output vector on the host*/
for (ii = 1; ii < numOutputElements; ii++) {
    hostOutput[0] += hostOutput[ii];
}
printf("Reduced Sum from GPU = %f\n", hostOutput[0]);
// Free the GPU memory here
cudaFree(deviceInput);
cudaFree(deviceOutput);
free(hostInput);
free(hostOutput);
return 0;
}

```

# Coordonare -> Host & Device

- Kernel-urile sunt pornite asincron
- Controlul este returnat catre CPU imediat
- CPU necesita sincronizare inainte sa foloseasca rezultatele obtinute pe device
- `cudaMemcpy()` blocheaza CPU pana cand copierea se finalizeaza.
  - Copierea incepe atunci cand toate apelurile CUDA anterioare s-au terminat;
- `cudaMemcpyAsync()`      *asynchronous* -> nu blocheaza CPU
- `cudaDeviceSynchronize()` blocheaza CPU pana toate apelurile CUDA se finalizeaza.

# NVIDIA GPU Execution Model

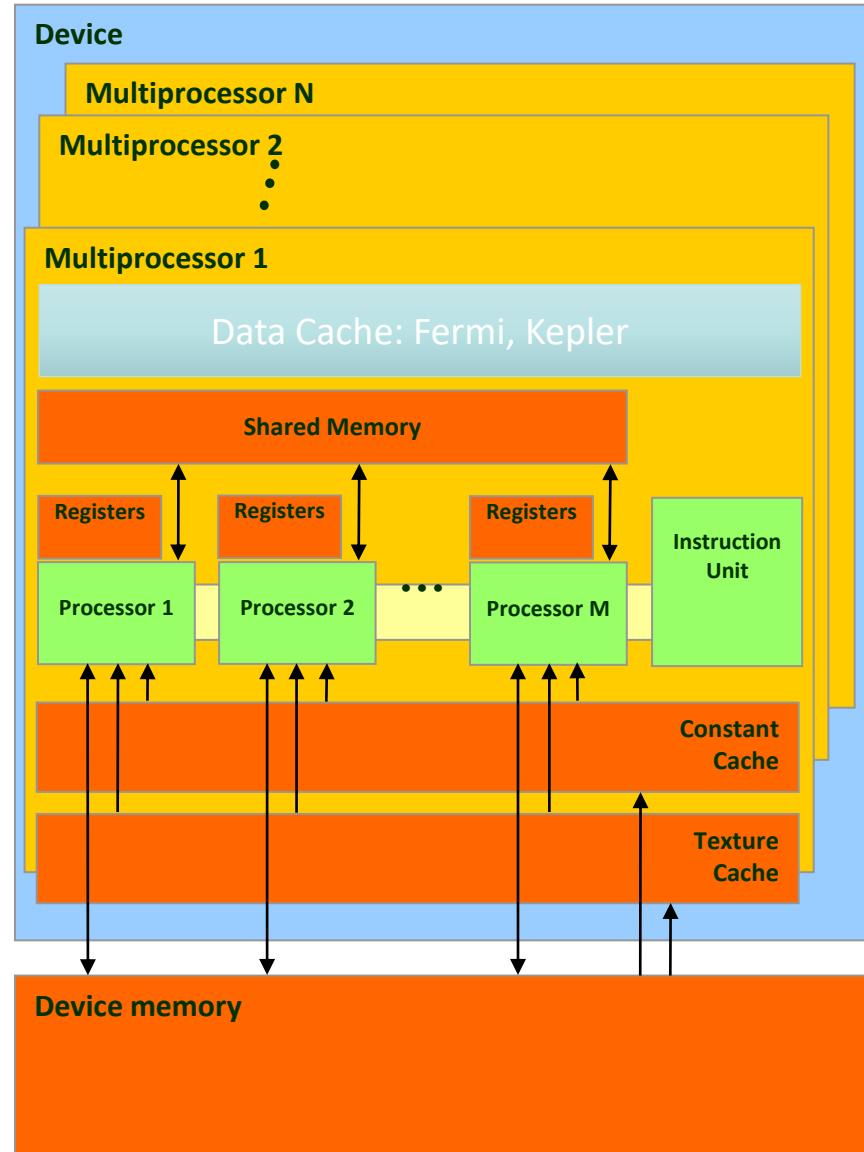
I. SIMD Execution of  
warpsize=M threads  
(from single block)

II. Multithreaded Execution across different instruction streams within block

- Also, possibly across different blocks if there are more blocks than SMs

III. Each block mapped to single SM

- No direct interaction across SMs



# SIMT = Single-Instruction Multiple Threads

- Model introdus de catre Nvidia
- Combina executia de tip SIMD din interiorul unui Block (pe un SM) cu executia SPMD intre Block-uri (distribuita pe /across SMs)

# Organizare - Structurare

- În GPU, unitatea de procesare este SP (streaming processor);
  - Mai multe SP și alte componente formează un SM (streaming multiprocessor);
  - Mai multe SM formează un TPC (texture processing cluster)
- În CUDA, putem spune că
  - un grid este procesat de catre întreg device-ul GPU,
  - un block este procesat de catre un SM, and
  - un thread este procesat de catre un SP.

## WRAP

- ❖ fiecare SM are 8 SP si pot fi 4 instructiuni in executie – pipelined => 32 threads
  - Cate 32 threads compun un **wrap**.
    - Daca se alege un numar de threaduri care nu se divide cu 32 atunci restul va forma un wrap (-> ineficient)
  - De fiecare data un **SM proceseaza doar un wrap si** astfel ca daca sunt mai putin de 32 threads intr-un wrap atunci unele SP nu sunt folosite.
  - **The warp size is the number of threads running concurrently on an SM.**
    - De fapt threadurile ruleaza si in paralel dar si pipelined
      - fiecare SM contine cate 8 SP
      - cea mai rapida instructiune necesita 4 cicluri (cycles).
      - => fiecare SP poate avea 4 instructiuni in propriul pipeline, deci avem un total de  $8 \times 4 = 32$  instructiuni care se executa concurrent.
  - In interiorul unui warp, thread-urile au indecsi secentiali:
    - Primul 0..31, urmatorul 32..63 s.a.md. Pana la numarul total de threaduri dintr-un block.[\[http://cuda-programming.blogspot.ro/2013/01/what-is-warp-in-cuda.html\]](http://cuda-programming.blogspot.ro/2013/01/what-is-warp-in-cuda.html)

# Efect-> wrap

- Omogenitatea threadurilor dintr-un wrap are un efect important asupra performantei calculului (*computational throughput*).
  - Daca toate threadurile executa aceeasi instructiune atunci toate SP dintr-un SM pot executa aceeasi instructiune in paralel.
  - Daca un thread dintr-un presupus wrap executa o instructiune diferita de celealte, atunci acel wrap trebuie sa fie partitionat in grupuri de threaduri bazat pe instructiunile care urmeaza sa fie executate; apoi grupurile se executa unul dupa altul.
    - Aceasta serializare reduce ‘throughput-ul’
      - pe masura ce threadurile devin tot mai heterogene se impart in grupuri tot mai mici.
- Rezulta ca este important sa se pastreze omogenitatatea pe cat posibil!

## Threads in Blocks

- Atunci cand un thread asteapta date, unitatea SM va alege un alt thread pentru a fi executat – astfel se ascunde latenta de acces la memorie.
- Astfel definirea a mai multe threaduri dintr-un block pot ascunde mai mult latenta;
  - Dar mai multe thread-uri intr-un block inseamna ca memoria partajata per threaduri este mai mica.
- Recomandarea NVIDIA: un block necesita cel putin 196 threaduri pentru a ascunde latenta corespunzatoare accesului la memorie.

# Optimizare

- Evitarea copiilor/transferurilor dintre memoriile CPU si GPU
- Folosire shared memory – acces rapid
- Alegerea potrivita a numarului de blocuri
- Array alignment ( alignment at 64 byte boundary)
- Continuous memory access
- Folosirea functiilor din CUDA API

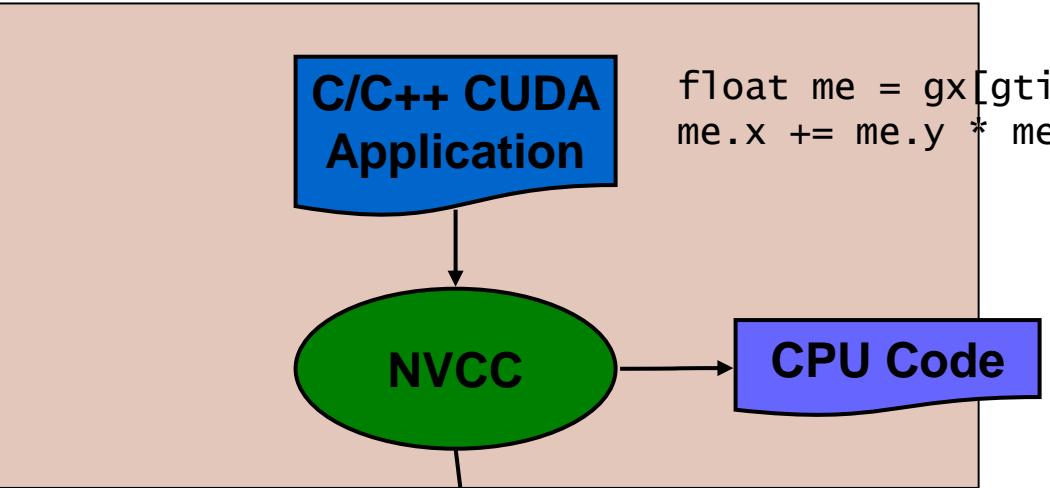
# Floating Point Operations

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

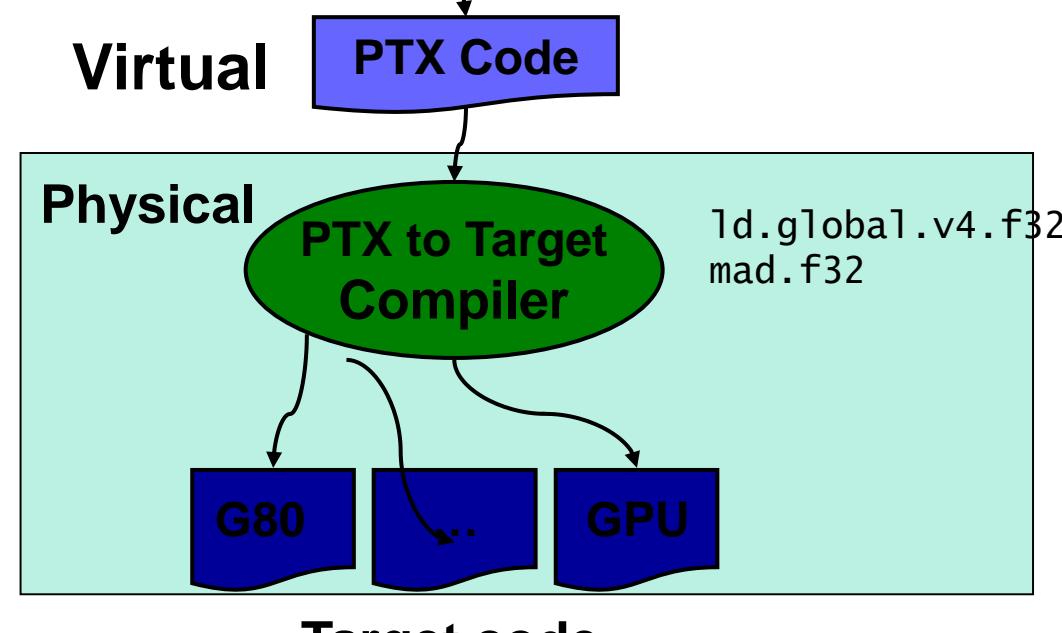
COMPILEARE

EXECUTIE

# Compiling a CUDA Program



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state



# Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

# Referinte prezentare

Prezentarea este bazata pe slide-uri din urmatoarele referinte:

- David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010. ECE 498AL, University of Illinois, Urbana-Champaign
- <http://cuda-programming.blogspot.ro/2013/01/what-is-constant-memory-in-cuda.html>
- Li Sung-Chi. Taiwan Evolutionary Intelligence Laboratory. 2016/12/14 Group Meeting Presentation
- Cyril Zeller. CUDA C/C++ Basics. Supercomputing 2011 Tutorial, NVIDIA Corporation  
<http://www.nvidia.com/docs/io/116711/sc11-cuda-c-basics.pdf>

# Curs 12

Programare Paralela si Distribuita

Strategii de Partitionare/Descompunere  
Sabloane de proiectare

# Partitionarea

- Problema partitionarii are în vedere împărțirea problemei de programare în componente care se pot executa concurențial.
- Aceasta nu implica o divizare directă a programului într-un număr de componente egal cu numărul de procesoare disponibile.
- Cele mai importante scopuri ale partitionarii sunt legate de:
  - scalabilitate,
  - abilitatea de a ascunde întâzierea (*latency*) datorată rețelei sau accesului la memorie și
  - realizarea unei granularități cât mai mari.
- Sunt de preferat partitionările care furnizează mai multe componente decât procesoare, astfel încât să se permită **ascunderea întâzierii**:
  - Un task poate fi blocat, sau va aștepta, până când
    - un mesaj care conține informația dorită va ajunge;
    - o condiție care este necesară ajunge să fie adevarată;
    - ...
  - Dacă există și alte componente program disponibile, procesorul poate continua calculul => multiprogramare => execuție concurențială

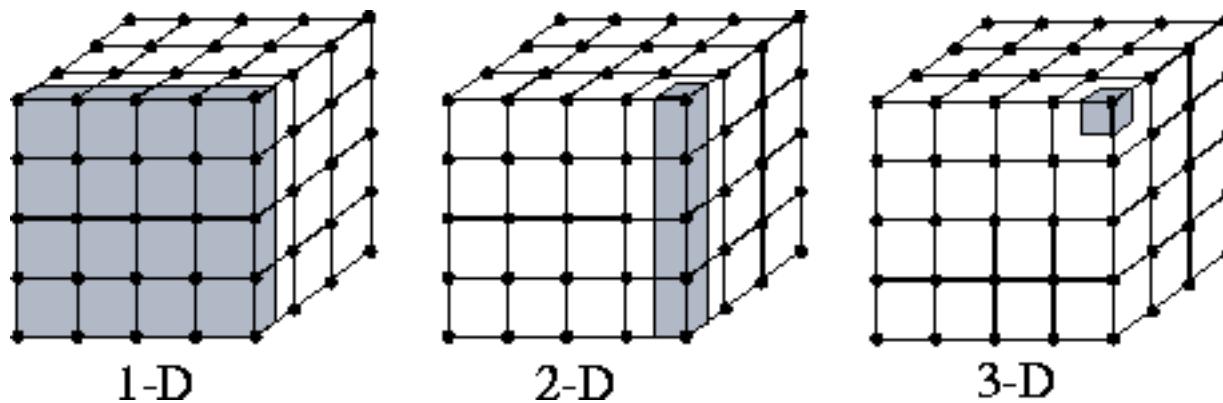
# Strategii de partitionare

- Există două strategii principale de partitionare:
  - descompunerea domeniului de date și
  - descompunerea funcțională.
- În funcție de acestea putem considera aplicații paralele bazate pe:
  - descompunerea domeniului de date – paralelism de date, și
  - aplicații paralele bazate pe descompunerea funcțională.
- Cele două tehnici pot fi folosite însă și împreună:
  - de exemplu se incepe cu o descompunere funcțională și după identificarea principalelor funcții se poate folosi descompunerea domeniului de date pentru fiecare în parte.

# Descompunerea domeniului de date

- Este aplicabila atunci cand domeniul datelor este mare si regulat.
- Ideea centrala este de a divide domeniul de date, reprezentat de principalele structuri de date, in componente care pot fi manipulate independent.
- Apoi se partitioneaza operatiile, de regula prin asocierea calculelor cu datele asupra carora se efectueaza.
- Astfel, se obtine un numar de activitati de calcul, definite de un numar de date si de operatii.
  - Este posibil ca o operatie sa solicite date de la mai multe activitati. In acest caz, sunt necesare comunicatii.

# Exemplificare



# Descompunerea domeniului de date

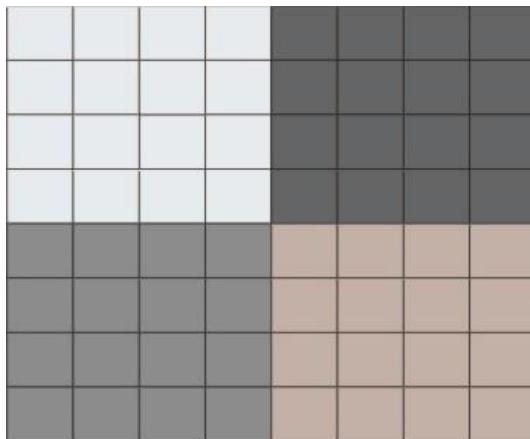
- Sunt posibile diferite variante, in functie de structurile de date avute in vedere.
- Datele care se partitioneaza sunt in general datele de intrare, dar pot fi considerate deasemenea si datele de iesire sau intermediare.
  - Trebuie avute in vedere in special structurile de date de dimensiuni mari sau cele care sunt accesate in mod frecvent.
- Faze diferite ale calculului pot impune partitionari diferite ale acelasi structuri de date
  - =>redistribuiriri ale datelor
    - in acest caz trebuie avute in vedere de asemenea si costurile necesare redistribuirii datelor.

# Distributii de date

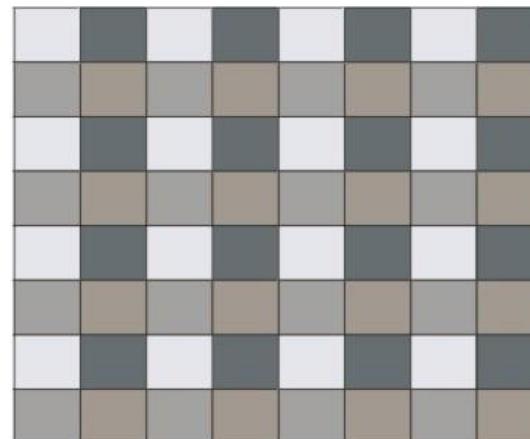
- Partitionarea datelor conduce la anumite distributii ale datelor per procese si de aici implicit si per procesoare.
- Exista mai multe tehnici de partitionare a datelor, care pot fi exprimate si formal prin functii definite pe multimea indicilor datelor de intrare cu valori in multimea indicilor de procese.
- Cele mai folosite tehnici de partitionare sunt prin “taiere” si prin “incretire” care corespund distributiilor liniara si ciclica.

# Distributii de date

- De exemplu, pentru o matrice A de dimensiune  $8 \times 8$ , partitionarea sa in  $p = 4$  parti:
  - prin tehnica taierei (distr. liniara) conduce la partitionarea arata de Figura (a), iar
  - prin tehnica incretirii (distr. ciclica) conduce la partitionarea arata de Figura (b).

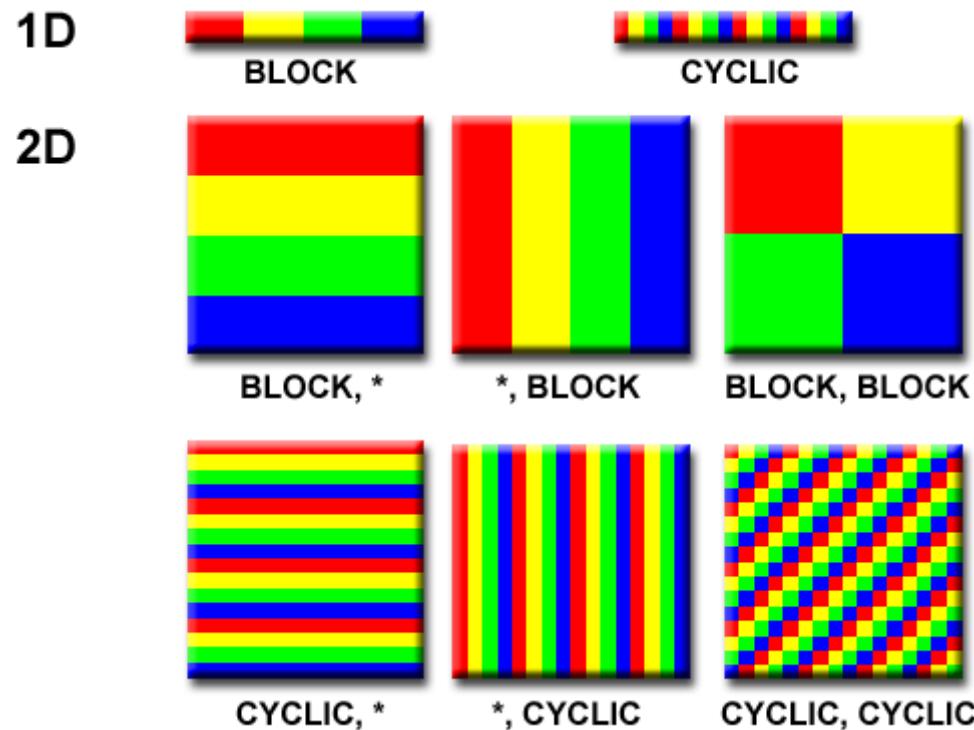


(a) Taiere (distributie bloc)



(b) Incretire (distributie grid)

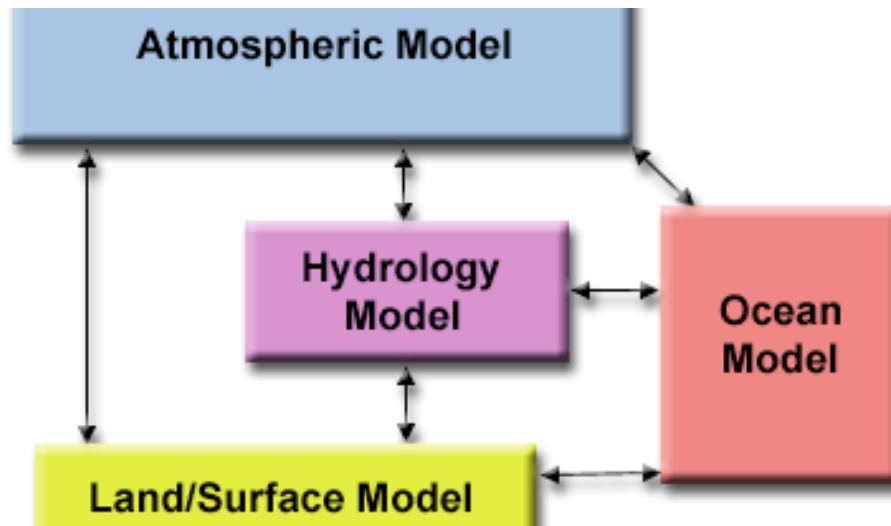
# Variante



# Descompunerea functionala

- Descopunerea functionala este o tehnica de partitionare folosita atunci cand aspectul dominant al problemei este functia, sau algoritmul, mai degraba decat operatiile asupra datelor.
- Obiectivul este descompunerea calculelor in activitati de calcul cat mai fine.
- Dupa crearea acestora se examineaza cerintele asupra datelor.
- **Focalizarea asupra calculelor** poate revela uneori o anumita structura a problemei, de unde oportunitati de optimizare, care nu sunt evidente numai din studiul datelor.
- In plus, ea are un rol important ca si tehnica de structurare a programelor.
- Aceasta varianta de descompunere nu conduce in general la o granulatie fina a sarcinilor de calcul, care se executa in paralel.

# Exemplificare



# Cerinte generale pentru partitionare

- Task-urile obtinute sunt de dimensiuni comparabile.
- Scalabilitatea poate fi obtinuta.
  - Aceasta inseamna ca numarul de sarcini de calcul sunt definite in functie de dimensiunea problemei;
  - deci cresterea dimensiunii datelor implica cresterea numarului de sarcini de lucru.
- Intazierile pot fi reduse prin multitasking.
- Granularitatea aplicatiei este suficient de mare astfel incat sa poate fi implementata cu succes pe diferite arhitecturi.

# SABLOANE DE PROIECTARE

... in general sabloanele de proiectare se bazeaza pe...

- Descompunere functionala
- Descompunere domeniului de date
- Fluxul de date

Observatie:

In literatura exista mai multe clasificari de sabloane de proiectare paralela dar sunt cateva sabloane de baza care se regasesc in majoritatea referintelor.

# Sabloane de baza

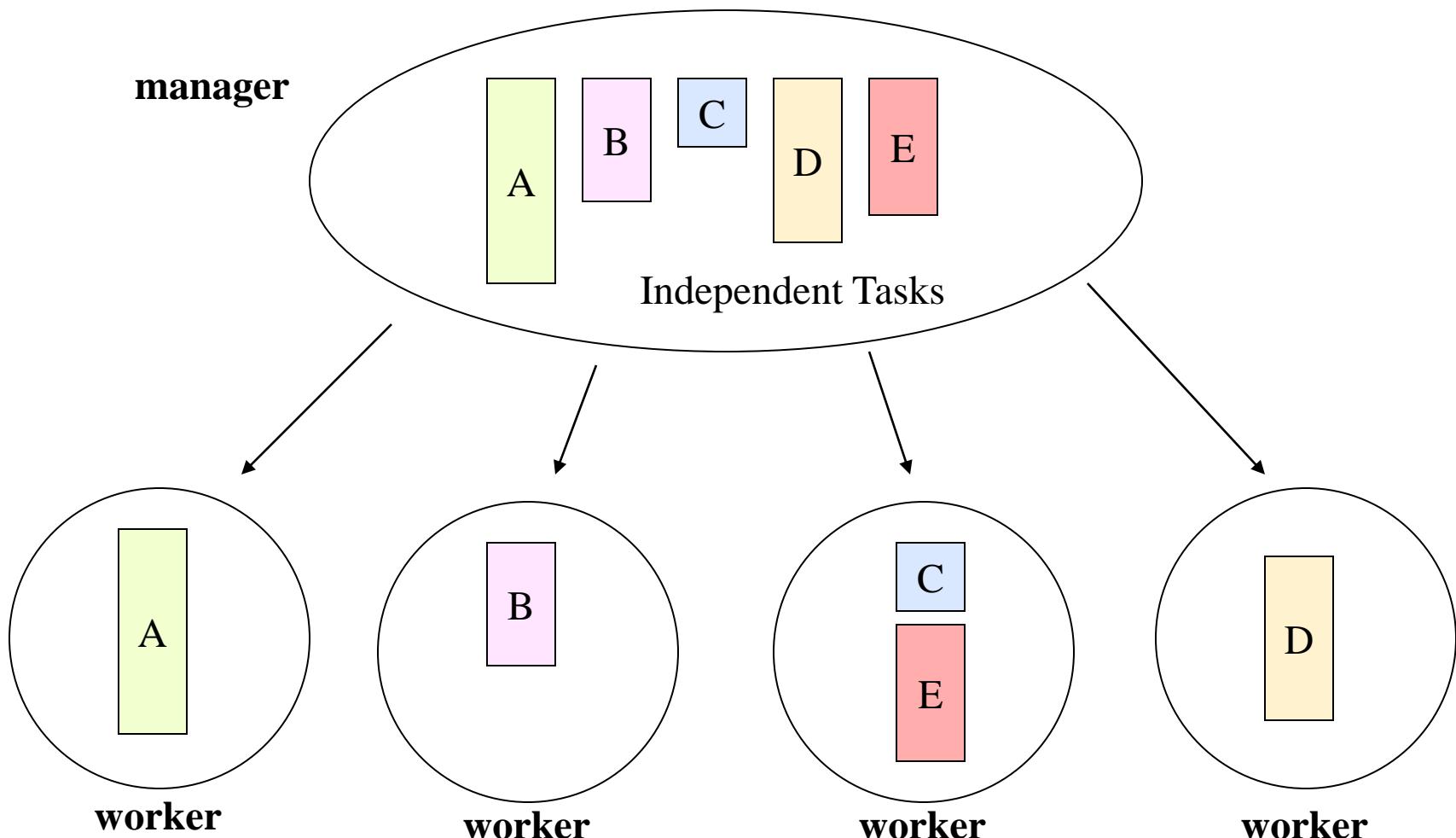
Parallel Programming Patterns  
**Eun-Gyu Kim** 2004

- Embarassingly Parallel – e.g. suma de vectori
  - /Master-Slave
- Replicable
- Repository
- Divide&Conquer
- Pipeline
- Recursive Data
- Geometric Decomposition
- IrregularMesh

# Variante pentru sabloane bazate pe descompunerea datelor sau a taskurilor

- Descompunere
  - geometrica ... -> caracter static
  - recursiva
  - exploratorie ... -> caracter dinamic
  - speculativa

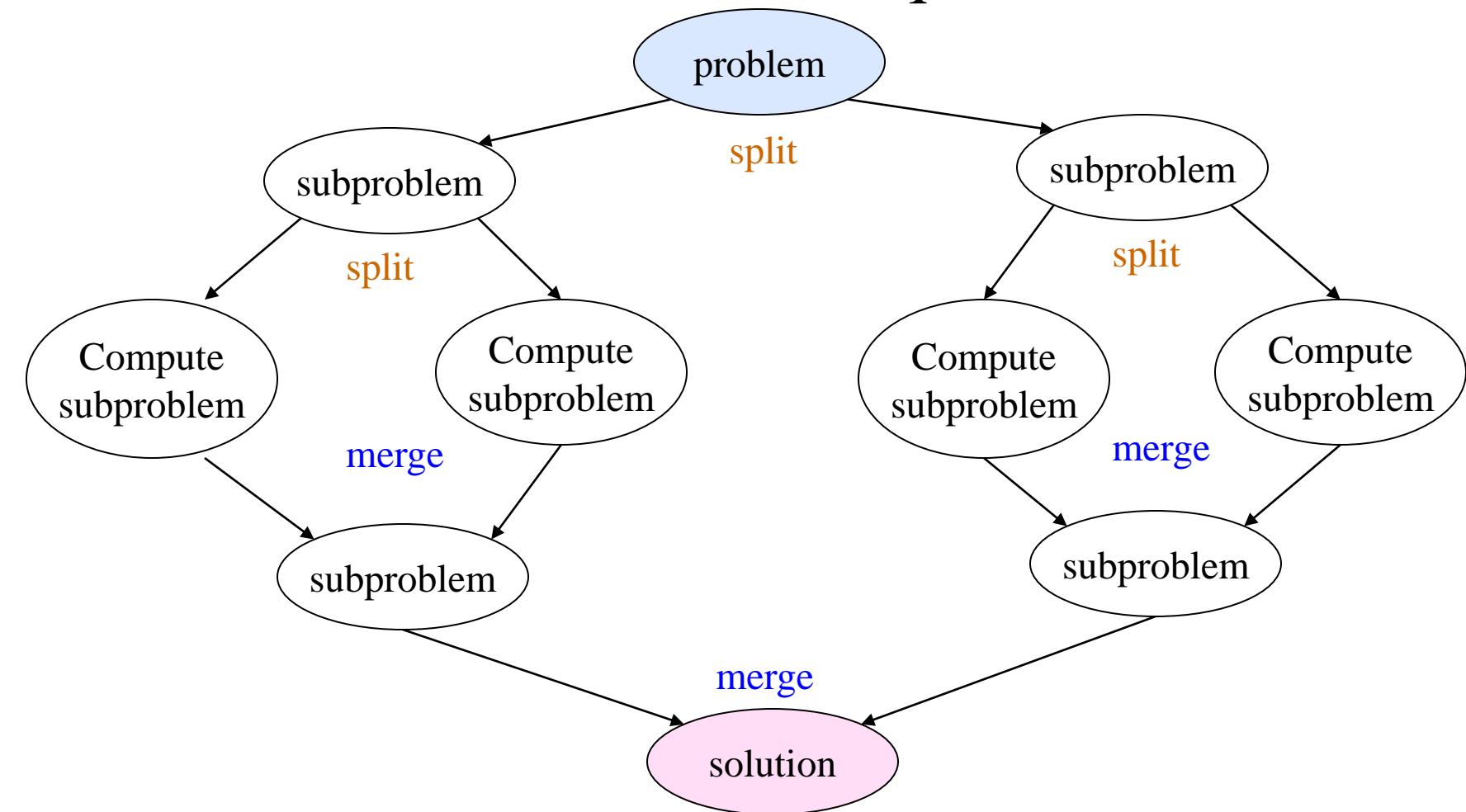
# *Master-slave (master-worker)*



# Descompunere Recursiva

- In general pentru probleme care se pot rezolva prin **divide & impera**

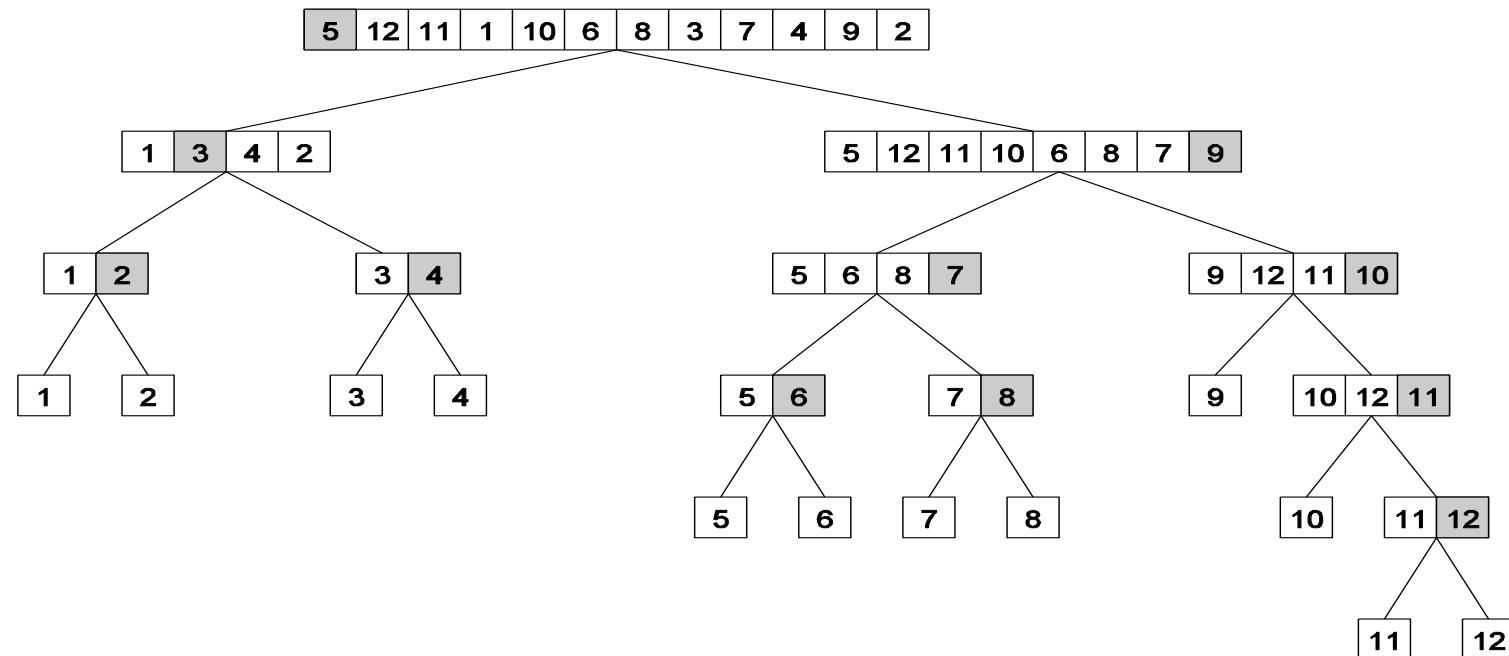
# *Divide & Conquer*



\* Nivelurile de descompunere trebuie sa fie ajustate corespunzator.

# Recursive Decomposition: Exemplul 1

Quicksort



Fiecare sublista reprezinta un task.

# *Recursive Decomposition: Exemplul 2*

Cautarea minimului:

```
1. procedure SERIAL_MIN (A, n)
2. begin
3.   min = A[0];
4.   for i := 1 to n – 1 do
5.     if (A[i] < min) min := A[i];
6.   endfor;
7.   return min;
8. end SERIAL_MIN
```

# Recursive Decomposition

Rescriere

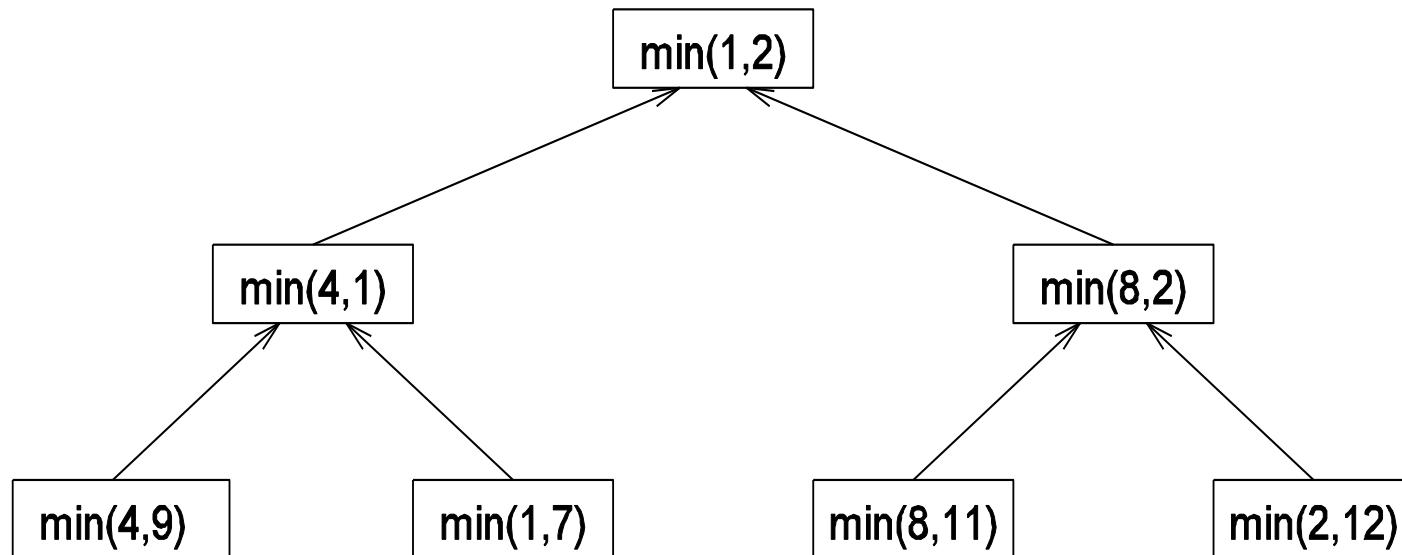
```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3.   if ( n = 1 ) then
4.     min := A [0] ;
5.   else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN ( &(A[n/2]), n - n/2 );
8.     if (lmin < rmin) then
9.       min := lmin;
10.    else
11.      min := rmin;
12.    endelse;
13.  endelse;
14.  return min;
15. end RECURSIVE_MIN
```

se pot  
executa in  
paralel

# *Recursive Decomposition*

$\{4, 9, 1, 7, 8, 11, 2, 12\}$ .

## - task dependency graph

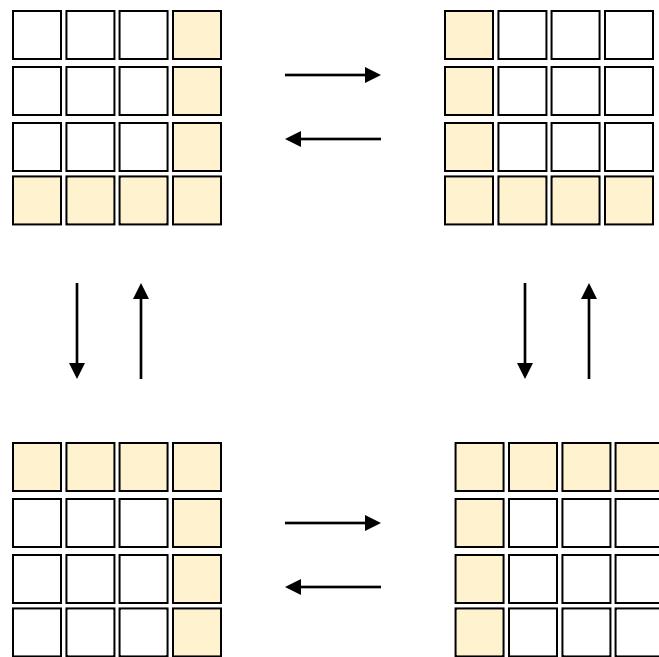


# *Data Decomposition (geometric )*

- Identificarea datelor implicate in calcul.
- Partitionarea datelor pe taskuri.
  - Diferite modalitati care implica impact important pt performanta.

# Descompunere Geometrica

Exista dependente dar comunicarea se face intr-un mod predictibil (geometric) -> vecini.



Neighbor-To-Neighbor communication

# *Output Data Decomposition: Exemplu*

inmultire de matrice

$n \times n$  matricele  $\mathbf{A} \times \mathbf{B} = \mathbf{C}$ .

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

# *Output Data Decomposition: Exemplu*

- nu există doar o descompunere unică – variante!

<b>Decomposition I</b>	<b>Decomposition II</b>
Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$	Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$
Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$	Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$
Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,1} \mathbf{B}_{1,2}$	Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,2} \mathbf{B}_{2,2}$
Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,2} \mathbf{B}_{2,2}$	Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,1} \mathbf{B}_{1,2}$
Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,1} \mathbf{B}_{1,1}$	Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,2} \mathbf{B}_{2,1}$
Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,2} \mathbf{B}_{2,1}$	Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,1} \mathbf{B}_{1,1}$
Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$	Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$
Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$	Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$

# *Output Data Decomposition: Exemplu*

Problema: numararea instantelor unor itemi intr-o baza de date de tranzactii.  
Output= itemset frequencies  
se partitioneaza intre taskuri.

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

(b) Partitioning the frequencies (and itemsets) among the tasks

Database Transactions	Itemsets	Itemset Frequency	Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1	A, B, C, E, G, H	C, D	1
B, D, E, F, K, L	D, E	3	B, D, E, F, K, L	D, K	2
A, B, F, H, L	C, F, G	0	A, B, F, H, L	B, C, F	0
D, E, F, H	A, E	2	D, E, F, H	C, D, K	0
F, G, H, K,			F, G, H, K,		
A, E, F, K, L			A, E, F, K, L		
B, C, D, G, H, L			B, C, D, G, H, L		
G, H, L			G, H, L		
D, E, F, K, L			D, E, F, K, L		
F, G, H, L			F, G, H, L		

task 1    task 2

# *Output Data Decomposition: Exemplu*

Analiza:

- Daca baza de tranzactii este replicata pe procese fiecare task se poate executa fara comunicatii.
- Daca baza de tranzactii este distribuita pe procese (memory ...) atunci fiecare task calculeaza frecvente partiale care trebuie ulterior aggregate.

# *Input Data Partitioning*

- Exemplu: minim intr-o lista, sortare,....
- Task <=> partitie input
- Procese ulterioare pot adauga/ combina rezultatele parțiale.

# *Input Data Partitioning: Exemplu*

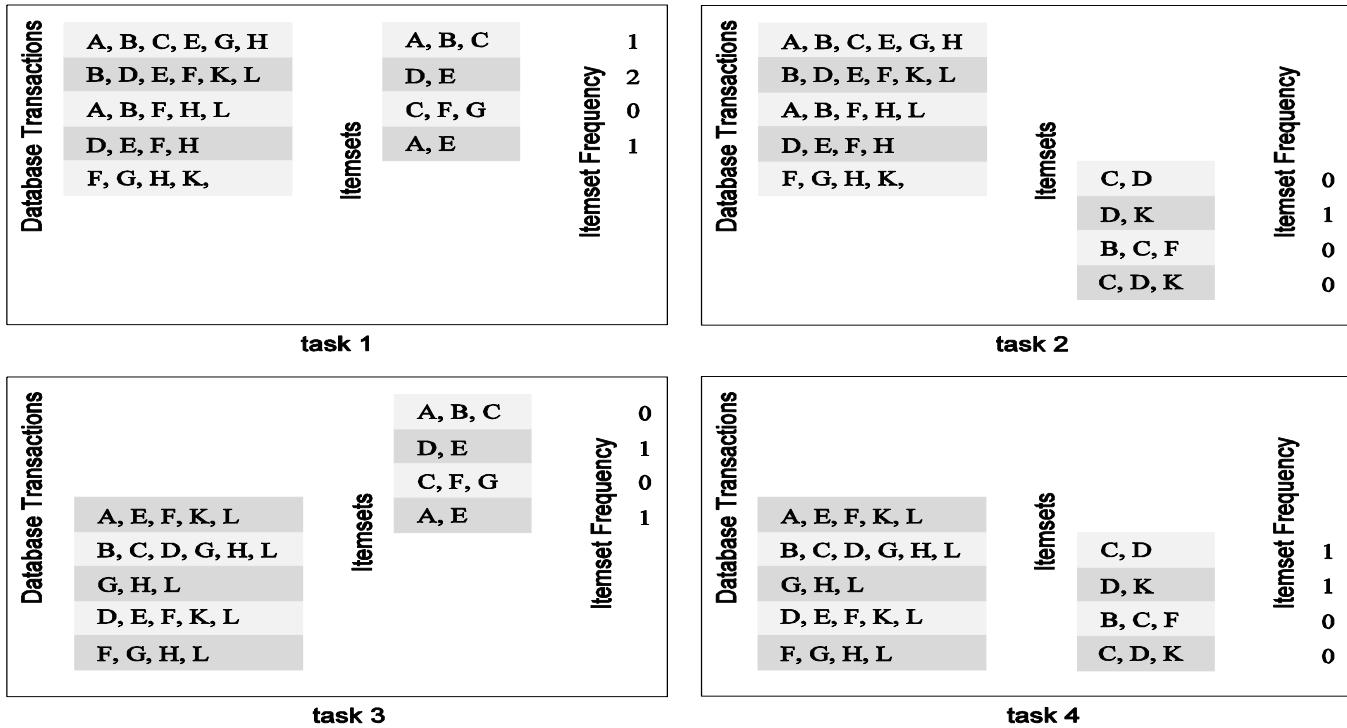
**Partitioning the transactions among the tasks**

Database Transactions	Itemsets	Itemset Frequency	Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H B, D, E, F, K, L A, B, F, H, L D, E, F, H F, G, H, K,	A, B, C D, E C, F, G A, E C, D D, K B, C, F C, D, K	1 2 0 1 0 1 0 0	A, E, F, K, L B, C, D, G, H, L G, H, L D, E, F, K, L F, G, H, L	A, B, C D, E C, F, G A, E C, D D, K B, C, F C, D, K	0 1 0 1 1 1 0 0

task 1    task 2

# Partitioning Input and Output Data

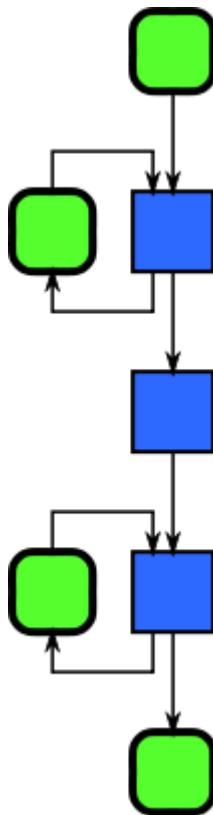
**Partitioning both transactions and frequencies among the tasks**



# *The Owner Computes Rule*

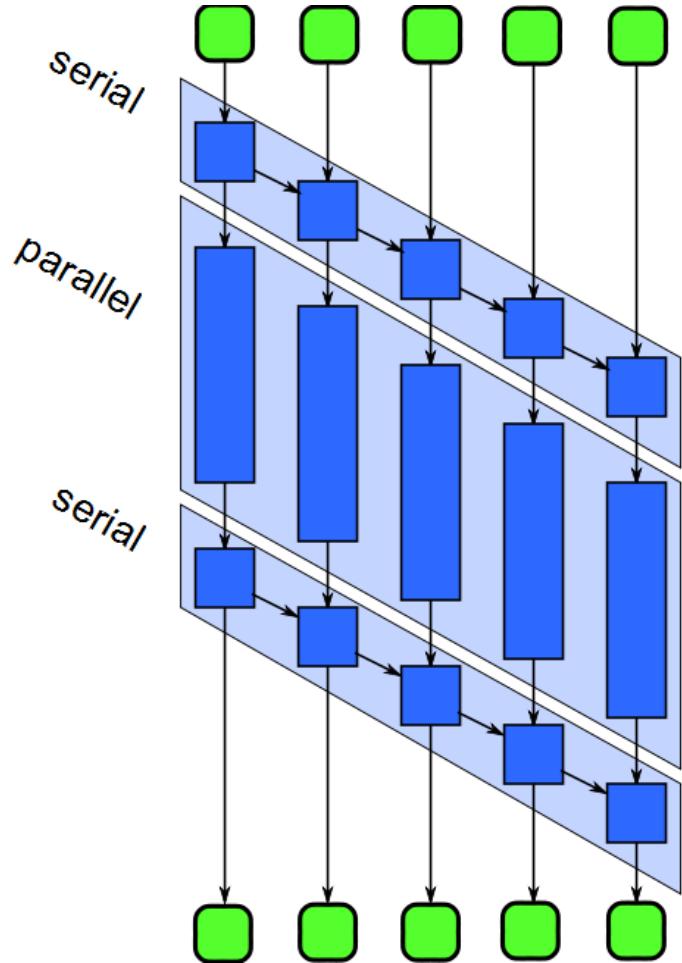
- Procesul care are date asignata lui este responsabil de calculele asociate acelei date
- Diferente:
  - input data decomposition
  - output data decomposition
- Obs: similar sablon GRASP: Expert

# Pipeline - sablon de programare paralela



- *Pipeline* – o secventa de stagii care transforma un flux de date
- Unele stagii pot sa stocheze stare
- Datele pot fi “consumate” si produse incremental.

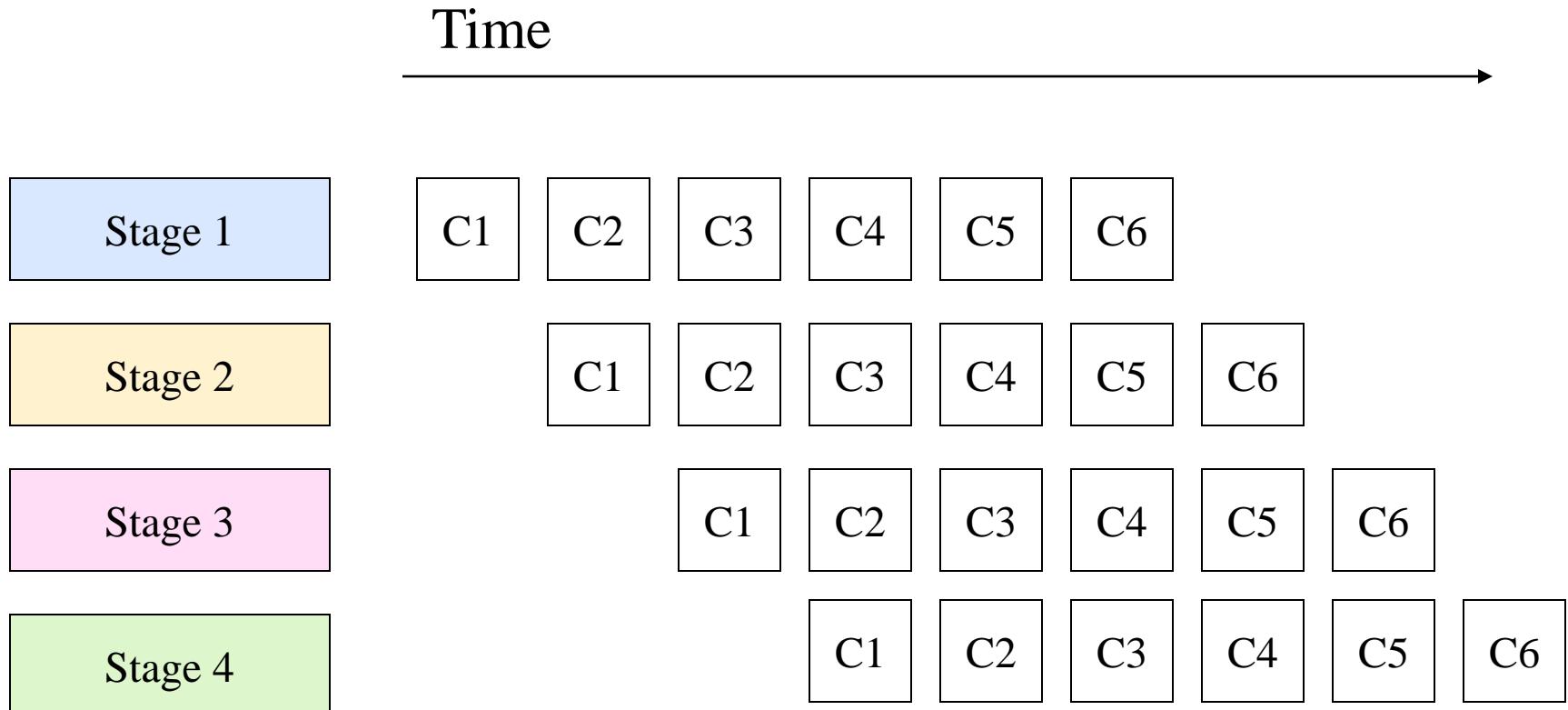
# Pipeline



- Paralelizarea pipeline se face prin
  1. Executia diferitelor stagii in paralel
  2. Executia multiplelor copii ale stagilor fara stare in paralel

# Pipeline

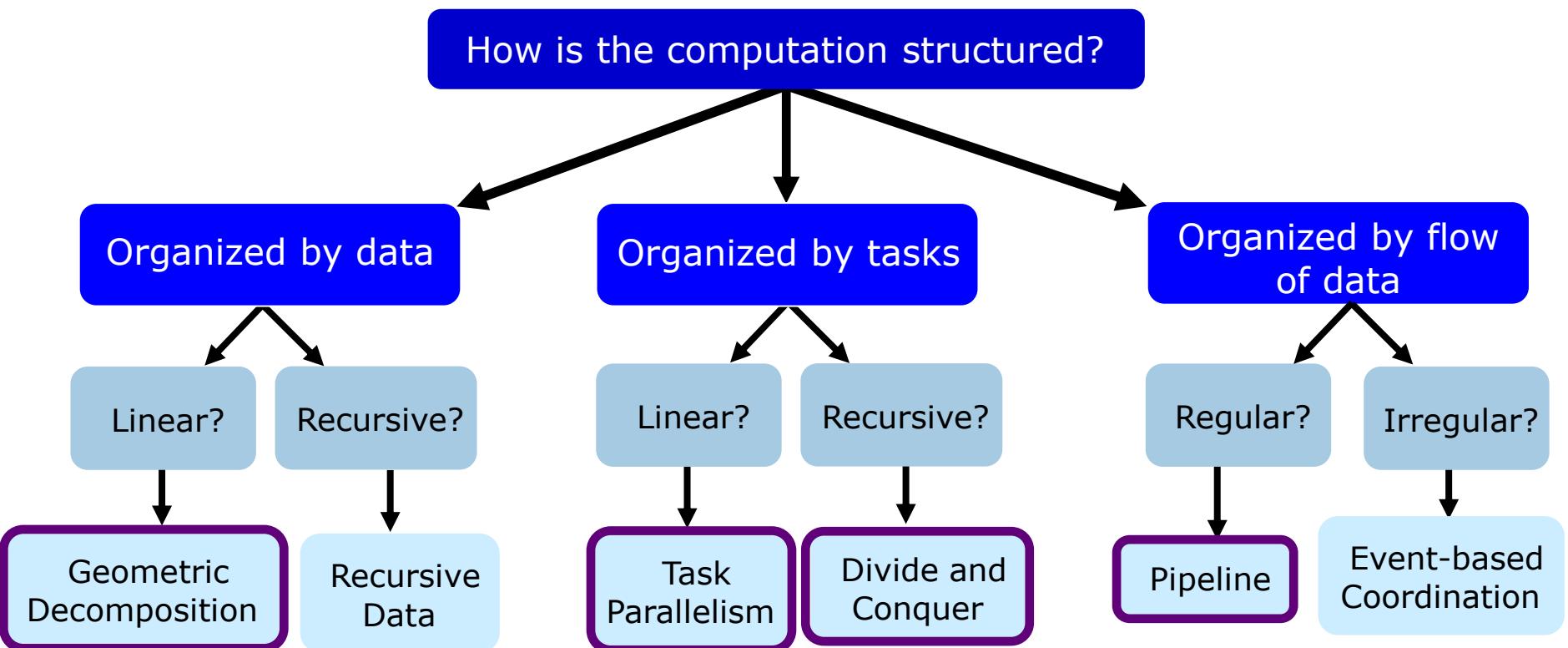
*A series of ordered but independent computation stages need to be applied on data, where each output of a computation becomes input of subsequent computation.*



# Sumar - descompunere

- Nu există doar o singura rețetă pentru descompunere
- Se pot aplica un set de tehnici comune pe o clasa de probleme mai vastă.
- *data decomposition (geometric decomposition)*
- *recursive decomposition*
- *exploratory decomposition*
- *speculative decomposition*
- *Pipelines... (se poate obține prin descompunerea fluxului de date)*

# Algorithm Structure Design Space



**Patterns for Parallel Programming.** Mattson, Sanders, and Massingill (2005).

## Design Space

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

## The Evolving Design

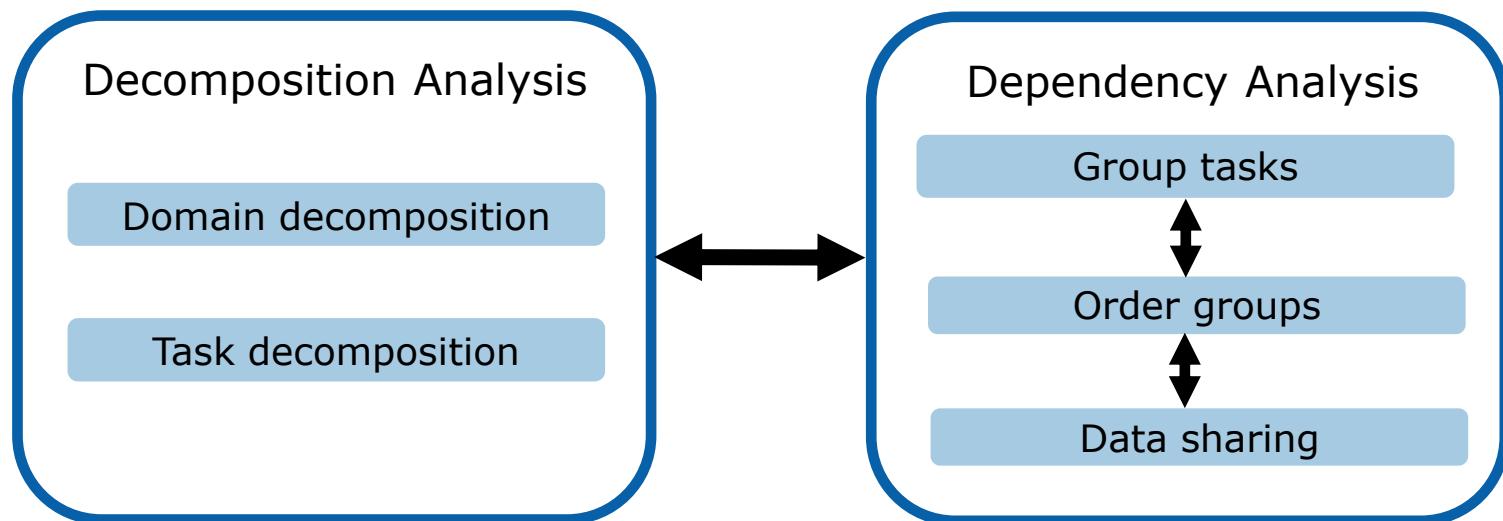
Tasks, shared data, partial orders

Thread/process structures, schedules

Source Code organization, Shared data

Messages, synchronization, spawn

# Cautare in spatiul de proiectare concurenta



# Applications

## Structural Patterns

Pipe-and-Filter  
Agent-and-Repository  
Process-Control  
Event-Based/Implicit-Invocation  
Arbitrary-Static-Task-Graph

Model-View-Controller  
Iterative-Refinement  
Map-Reduce  
Layered-Systems  
Puppeteer

## Computational Patterns

Unstructured-Grids  
Structured-Grids  
Graphical-Models  
Finite-State-Machines  
Backtrack-Branch-and-Bound  
N-Body-Methods  
Circuits  
Spectral-Methods  
Monte-Carlo

## Parallel Algorithm Strategy Patterns

Task-Parallelism  
Divide and Conquer

Data-Parallelism  
Pipeline

Discrete-Event  
Geometric-Decomposition  
Speculation

## Implementation Strategy Patterns

SPMD  
Fork/Join  
Program structure

Kernel-Par.  
Loop-Par.  
Vector-Par.

Actors  
Work-pile

Shared-Queue  
Shared-Map  
Shared-Data

Partitioned-Array  
Partitioned-Graph  
Data structure

## Parallel Execution Patterns

Coordinating Processes  
Stream processing

Shared Address Space Threads  
Task Driven Execution

Referinte:

``Introduction to Parallel Computing''

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar  
2003

Ian Foster

Designing and Building Parallel Programs, Addison Wesley, 2, Addison-Wesley Inc.,  
Argonne National Laboratory (<http://www.mcs.anl.gov/~itf/dbpp/>)

Parallel Programming Patterns

Eun-Gyu Ki, 2004

Patterns for Parallel  
Programming. Mattson,  
Sanders, and Massingill  
(2005).

# Curs 13

Distributed Computing Patterns

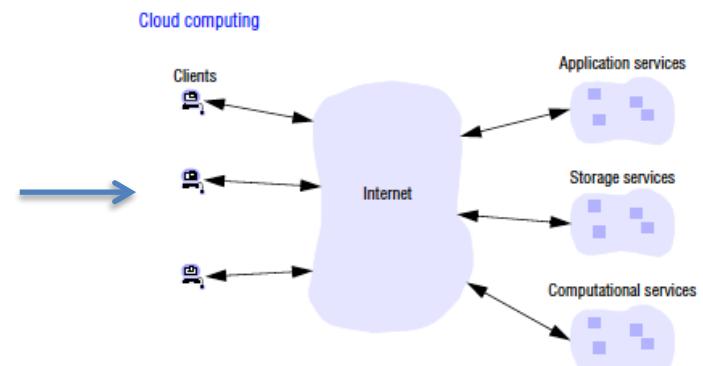
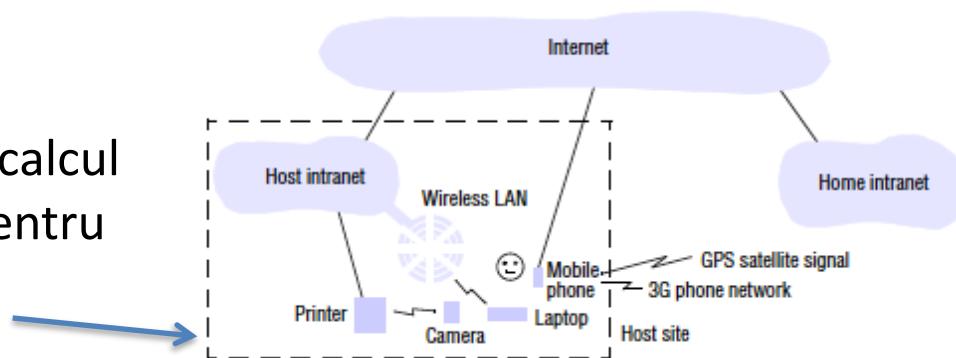
# Distributed systems

- Un sistem distribuit poate fi definit ca fiind format din **componente hardware si software localizate intr-o retea de calculatoare care comunica si isi coordoneaza actiunile doar bazat pe transmitere de mesaje.**
- Calcul distribuit (Distributed-Computing) = rezolvarea unor probleme folosind sisteme distribuite

# Tendinte

Influențe semnificative:

- emergenta tehnologiilor de retea de scara largă
- emergenta necesitatilor crescute de calcul cuplata cu dorinta de asigura suport pentru mobilitatea utilizatorilor
- cresterea cererii de servicii multimedia
- perspectiva asupra sistemelor distribuite ca fiind o utilitate publica



# Caracteristici

## Concurrenta:

- *se lucreaza cu programe care se executa concurrent si care partajeaza resurse*

## No global clock:

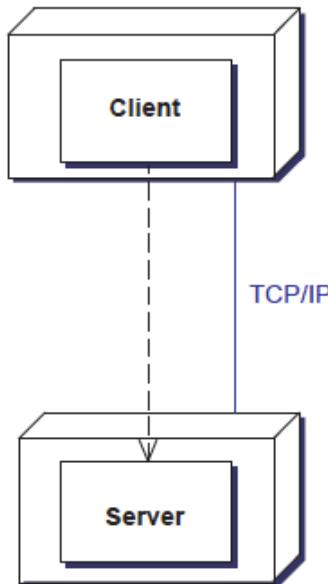
- *nu exista notiunea de timp global*
- aceasta este o consecinta directa a faptului ca singura modalitate de comunicare este transmiterea de mesaje prin retea

## Independent failures:

**Fiecare componenta a sistemului poate ‘cadea’ in mod independent lasand celelalte in starea de executie (‘run’).**

- toate retelele de calculatoare pot esua(‘fail’) si este responsabilitatea proiectantilor sistemului sa gestioneze efectele in aceste cazuri si sa asigure masuri de reorganizare.
- Aceste tipuri de probleme conduc la izolarea computerelor conectate prin aceste retele aflate in starea de ‘fault’ – dar nu inseamna si oprirea celorlalte computere;
- Programele pot sa continue sa functioneze dar nu pot detecta daca reteaua a cazut sau este doar incetinita.
- Similar, ‘caderea’ unui computer, sau oprirea neasteptata a unui software undeva in sistem (*a crash*), nu este imediat facuta cunoscuta celorlalte componente cu care acesta comunica

# Client-Server pattern



- O componentă de tip server care furnizează servicii către mai multe componente client.
- O componentă client cere servicii de la componentă server.
- Serverele sunt active permanent ‘ascultand’ cererile de la clienti.

# Starea(State) in sablonul Client-server

Clientii si serverele lucreaza in general in sesiuni ('sessions').

–**stateless server** -- starea unei sesiuni (**session state**) este gestionata de catre client. Aceasta stare (client) este trimisa impreuna cu fiecare cerere. In aplicatiile web, *session state* poate fi stocata ca si parametrii URL, in campuri ascunse sau folosind cookies (obligatoriu pentru arhitecturile REST folosite pentru aplicatii web).

–**stateful server** -- starea unei sesiuni (**session state**) este mentinuta la nivel de server si este asociata cu ID clientului

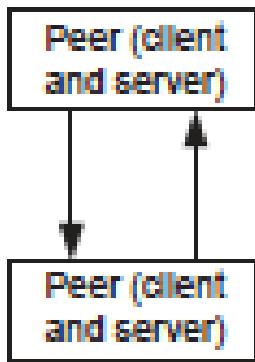
Modalitatea de gestionare a starii unei sesiuni influenteaza tranzactiile, scalabilitatea si gestiunea erorilor (*fault handling*).

- Tranzactiile trebuie sa fie
  - atomice si sa asigure consistenta starii,
  - izolate (sa nu afecteze alte tranzactii)
  - durabile
- *fault handling* => starea mentinuta la nivelul clientului implica faptul ca toata informatia se va pierde in cazul in care clientul esueaza (*fails*).
- Securitatea poate fi afectata daca starea se menține la nivel de client pentru ca informatie se transmite de fiecare data (la fiecare *request*).
- Scalabilitatea poate fi redusa daca starea se menține la nivelul serverului '*in-memory*' – multi client, multe cereri => necesar de memorie crescut.

# Peer-to-peer pattern

- Poate fi privit ca si un sablon Client-Server **simetric**:
  - un nod (*peer*) poate functiona ca si un client – care cere servicii de la alte componente sau ca si server care furnizeaza servicii pentru altii.
  - rolul unui nod se poate schimba in mod dinamic
- Atat clientii cat si servere folosesc in mod uzual multithreading.
- Serviciile pot fi implicite (de exemplu prin intermediul unui stream de conectare) in locul unei cerere (request) trimise prin invocare directa.
- Un *peer* care actioneaza ca si server isi pot informa colegii (peers) care activeaza ca si client de aparitia unor evenimente; clientii pot fi informati folosind de exemplu o magistrala de evenimente (event-bus).

# Exemple



- the distributed search engine Scienonet,
- multi-user applications like a drawing board,
- peer-to-peer file-sharing like Gnutella or BitTorrent.

# Caracteristici

- **Performanta** creste atunci cand numarul de noduri creste dar scade atunci cand sunt prea putine

## Avantaje

- Nodurile pot folosi capacitatea intregului sistem chiar daca fiecare are o capacitate proprie limitata.
  - este un **cost individual mic cu beneficiu mare** obtinut prin partajare
- Overhead-ul de administrare este scazut pentru ca retelele peer-to-peer se organizeaza intern (self-organizing)
- Asigura **scalabilitate** foarte buna si este rezilienta la esec/caderea(failure) componentelor individuale.
- Configurarea sistemului se poate schimba **dinamic**: un *peer* poate intra sau pleca in timp ce sistemul functioneaza.

## Dezavantaje

- nu exista garantia calitatii serviciilor (*no guarantee about quality of service*) deoarece nodurile coopereaza voluntar
- nu exista garantia securitatii (*security is difficult to guarantee*) deoarece nodurile coopereaza voluntar

## Forwarder-Receiver

- **Communication Pattern**
  - Forwarder-Receiver furnizeaza in mod transparent comunicarea inter-proces pentru sistemele sistem cu model de interactiune de tip peer-to-peer.
  - Foloseste *forwarders* si *receivers* pentru a decupla nodurile de mecanismul de comunicare de baza (the underlying mechanism).

# Architectural patterns

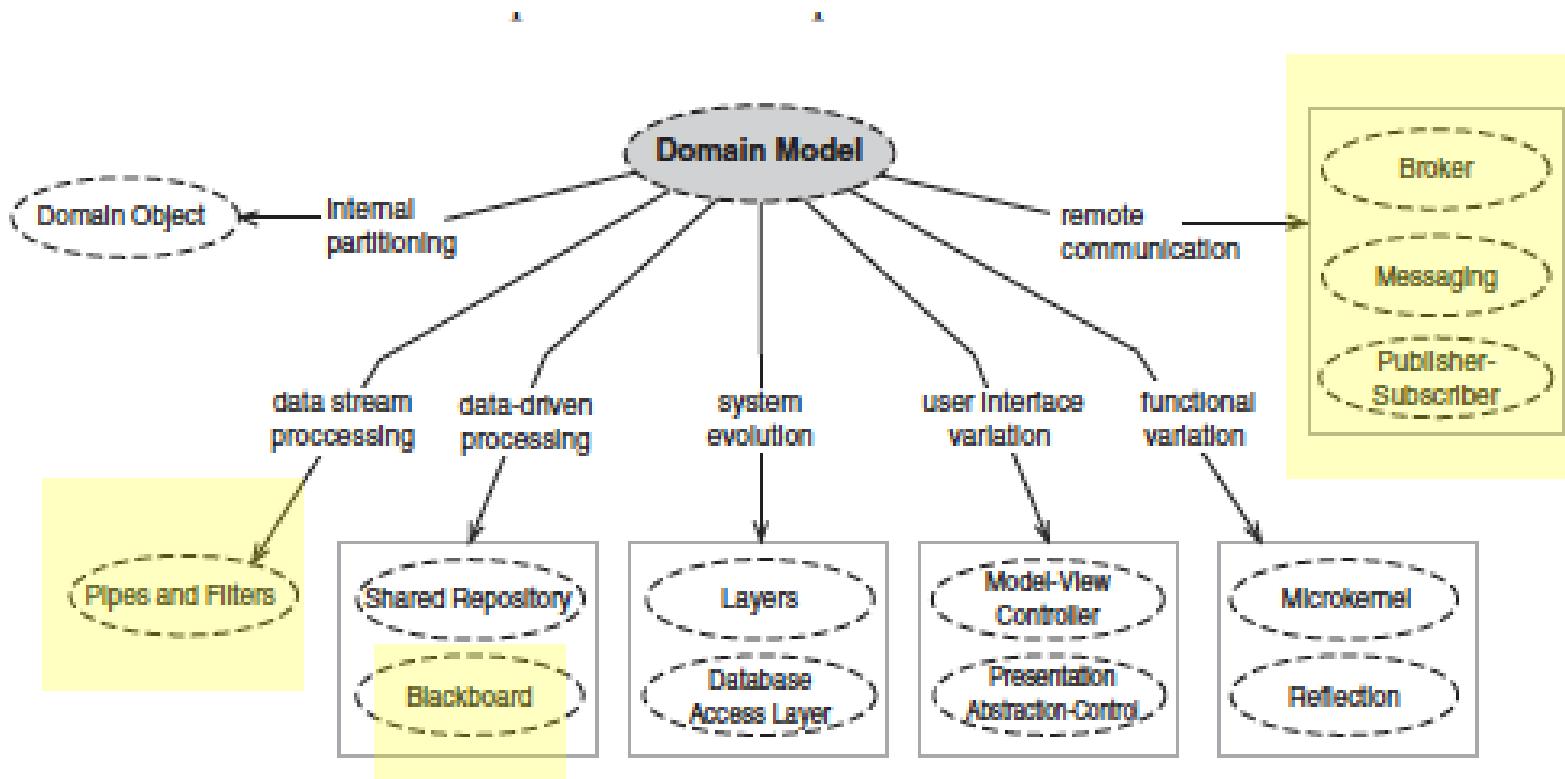
Frank Buschmann. Kevlin Henney. Douglas C. Schmidt. ***Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed-Computing.*** Wiley & Sons, 2007

Un sablon arhitectural este un concept care

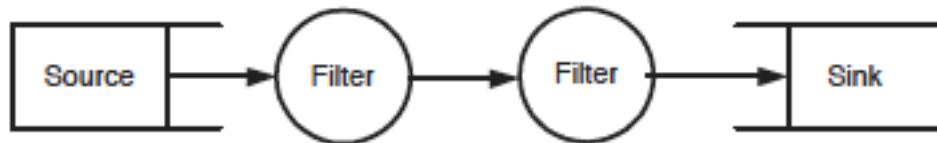
**rezolva si delimitaaza anumite elemente esentiale de coeziune ale unei arhitecturi software.**

- Domain Model
- Layers
- Model-View-Controller
- Presentation-Abstraction-Control
- Microkernel
- Reflection
- Pipes and Filters
- Shared Repository
- Blackboard
- Domain Object

# Conexiunea cu Domain Layer

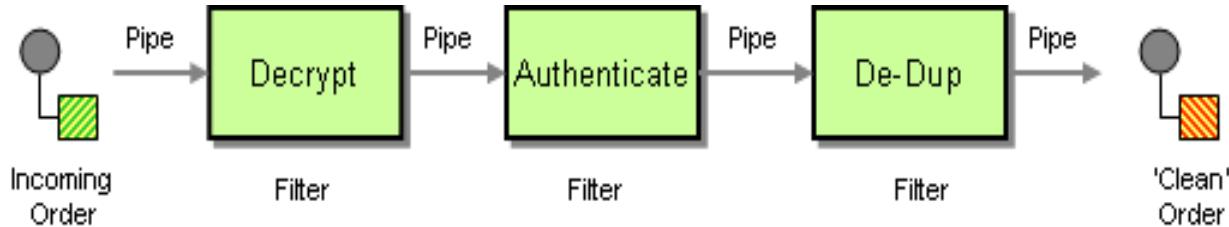


# Pipe-Filter Pattern



- furnizeaza pentru un sistem o structura care produce un ***stream de date***
- fiecare pas de procesare este incapsulat intr-o componenta de tip ***filter***
- Datele sunt transferate prin pipes
  - *pipe* – leaga 2 filtre
  - *pipes* pot fi utilizate pentru sincronizare sau pentru buffering

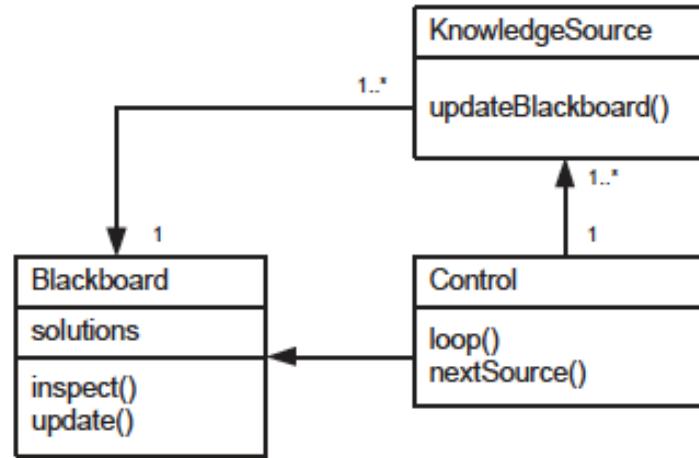
# Exemplu



- se poate utiliza pentru a divide taskuri de procesare mari intr-o secvență de componente independente mai mici de procesare (Filters) care sunt conectate prin canale (Pipes).
- fiecare filtru expune o interfață simplă – primește mesaje de la *inbound pipe*, procesează mesajul și publică rezultatul la *outbound pipe*.
- pentru că toate componentele utilizează aceeași interfață externă ele pot fi compuse în diferite soluții prin conectarea componentelor la pipe-uri diferite
- se pot adăuga filtre sau rearanja în secvențe noi fără a se schimba filtrele (în interior).

# Blackboard Pattern

arata cum se poate rezolva o problema complexa precum recunoasterea de imagini sau de limbaj prin impartirea in subsisteme mai mici specializate care pot rezolva problema impreuna.



- mai multe subsisteme specializate asambleaza cunostinte pentru a construi partial sau aproximativ solutia
- ***Toate componentele au acces la ansamblul de datele partajate = the blackboard.***
- Componentele pot produce date noi care sunt adaugate pe blackboard.
- Componentele cauta anumite date pe blackboard, folosind e.g. pattern matching.

# Componente

- Class
    - Blackboard
  - Responsibility
    - Manages central data
  - Collaborators
    - no
- 
- Class
    - Knowledge Source
  - Responsibility
    - Evaluates its own applicability
    - Computes a result
    - Updates Blackboard
  - Collaborators
    - Blackboard
- 
- Class
    - Control
  - Responsibility
    - Monitors Blackboard
    - Schedules knowledge source activations
  - Collaborators
    - Blackboard
    - Knowledge Source

# Descriere generală

- Blackboard permite mai multor procese (agenti) să comunice prin citirea și scrierea de cereri și informații către un depozit de date globale.
- **Fiecare agent participant are o expertiza în propriul domeniu**, și are un anumit tip de cunoaștere referitoare la rezolvarea problemei (knowledge source) care se poate aplica la o parte a problemei, i.e., problema nu se poate rezolva de către un singur agent.
- **Agentii comunică strict prin intermediul unei *common blackboard*** care este vizibil tuturor agentilor.
- Atunci când o problema parțială trebuie rezolvată potențialii agenti care ar putea să o rezolve sunt listati.
- **O unitate de control** este responsabilă pentru selectarea agentilor și atribuirea de sarcini acestora.

# Exemplu: speech recognition

## Ciclul principal de rezolvare

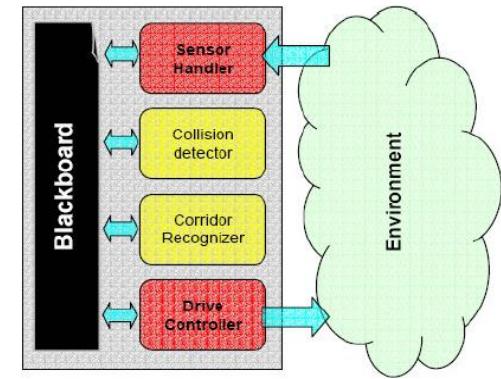
- **Control** calls **nextSource()** to select the next knowledge source
- **nextSource()** looks at the blackboard and determines which knowledge sources to call
- For example, **nextSource()** determine that **Segmentation**, **Syllable Creation** and **Word Creation** are candidate
- **nextsource()** invokes the **condition-part** of each candidate knowledge source
- The **condition-parts** of candidate knowledge source inspect the blackboard to determine if and how they can contribute to the current state of the solution
- The **Control** chooses a knowledge source to invoke and a **set of hypotheses** to be worked on (according to the result of the condition parts and/or control data)
- Apply the **action-part** of the knowledge source to the **hypothesis**
- *New contents are updated in the blackboard*

# Robot example

- An Experimental robot is equipped with four agents:
  - Sensor Handler Agent,
  - Collision Detector Agent,
  - Corridor Recognizer Agent and
  - Drive Controller Agent

(Includes the control software)

- Agents and blackboard form the control system. Agent cooperation is reached by means of the blackboard. Blackboard is used as a central repository for all shared information.
- Only two agents have an access to the environment: Sensor Handler Agent and Drive Controller Agent.
- There is no global controller for all of these agents, so each of them independently tries to make a contribution to the system during the course of navigation.
- Basically each of the four agents executes its tasks independently using information on the blackboard and posts any result back to the blackboard.



# Avantaje & Dezavantaje

- Avantaje
  - potrivit pentru diverse surse de date/procesare
  - potrivit pentru medii distribuite
  - **potrivit pentru planificarea taskurilor si a deciziilor**
  - **potrivit pentru abordari de rezolvare in echipa** – se posteaza subcomponente si rezultate partiale
  - util pentru probleme pentru care nu sunt cunoscute strategii de rezolvare deterministe => **opportunistic problem solving**.
- Dezavantaje
  - Scump
  - Dificil de a determina partitionarea pentru knowledge
  - Control unit poate fi foarte complexa

# ....patterns legate de infrastructura de comunicare

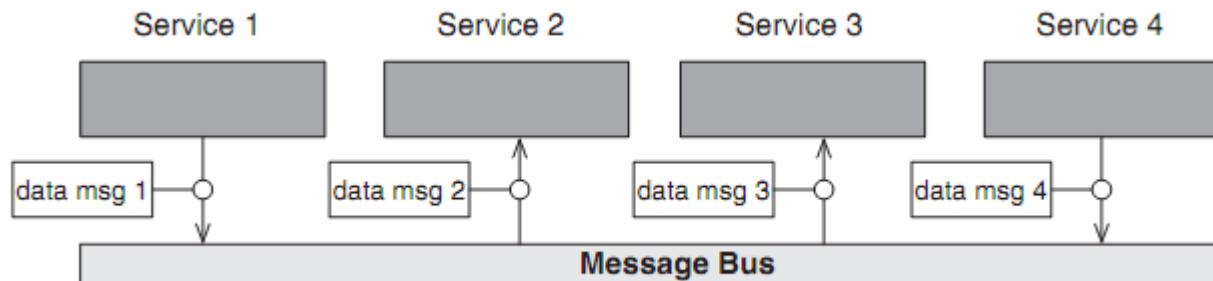
- **Messaging**
  - Distribution Infrastructure
- **Publisher-Subscriber**
  - Distribution Infrastructure
- **Broker**
  - Distribution Infrastructure
- **Client Proxy**
  - Distribution Infrastructure
- **Reactor**
  - Event Demultiplexing and Dispatching
- **Proactor**
  - Event Demultiplexing and Dispatching

# Messaging Pattern

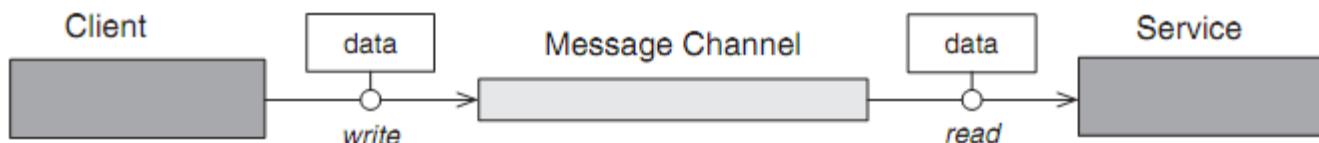
- Unele sisteme distribuite sunt compuse din servicii care sunt dezvoltate independent.
- Pentru a forma un sistem coherent aceste servicii trebuie sa interactioneze intr-un mod fiabil fara a implica totusi o dependenta stransa intre ele.
- **Solutie:** Conectarea printr-un canal de mesaje (*a message bus*) care sa permita transferul de date **asincron**.
  - mesajele se codifica astfel incat comunicarea sa se poata face fara sa fie nevoie de sa fie cunoscute toate informatiile legate de tipurile de date.

# Messaging

- Message-based communication supports *loose coupling*



- *Mesajele* contin doar datele care pot fi interschimbate intre setul de clienti si servicii – nu si cine este interesat de acestea.  
=> conectarea clientilor si serviciilor printr-un canal care permite schimbul de mesaje “Message Channel”.

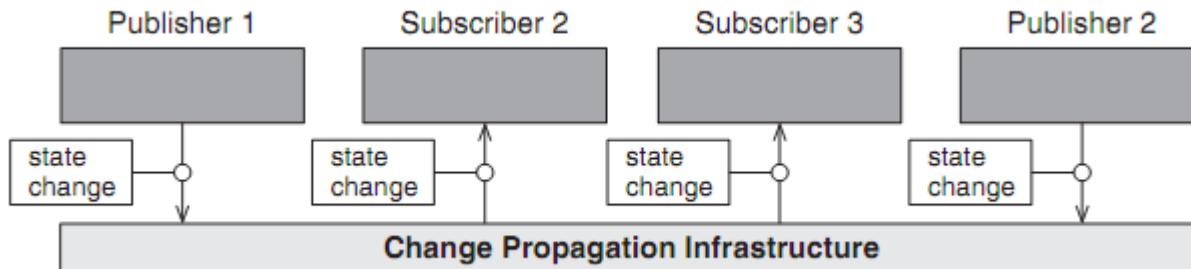


# Publisher-Subscriber Pattern

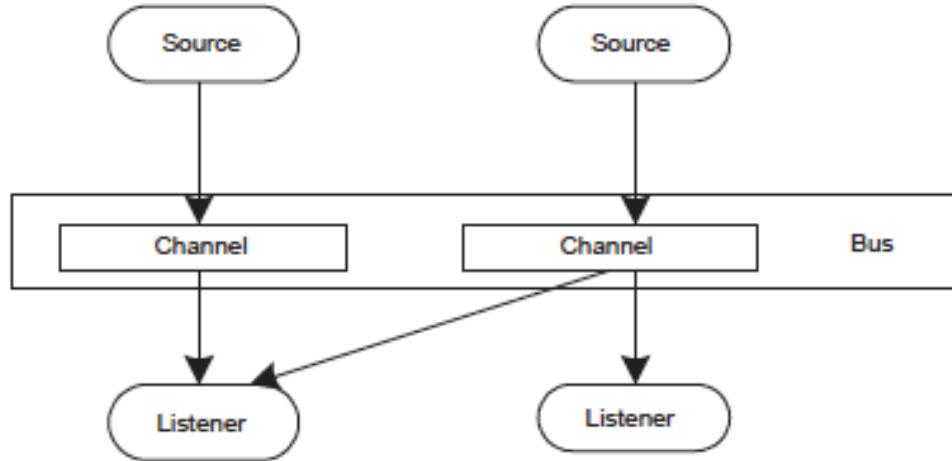
- Componentele din anumite aplicatii distribuite sunt cuplate slab si opereaza in general independent.
- pentru a propaga informatii in asemenea aplicatii se poate folosi un mechanism de notificare prin care sa se faca informari referitoare la modificari de stare sau referitoare la aparitia unor evenimente.
- **Solutie:** Definerea unei infrastructuri de propagare a informatiei care permite editorilor sa disemineze evenimente care contin informatie care ar putea sa intereseze alti actori.
  - se notifica abonatii inregistrati pentru a primi anumite tipuri de notificari

# Publisher-Subscriber

- Editorii inregistreaza ce tipuri de evenimente publica.
- Abonatii inregistreaza de ce tipuri de evenimente sunt interesati.
- Infrastructura foloseste informatiile inregistrate pentru a transmite prin retea evenimentele de la editori la abonatii inregistrati a fi interesati.



# Event-Bus Pattern

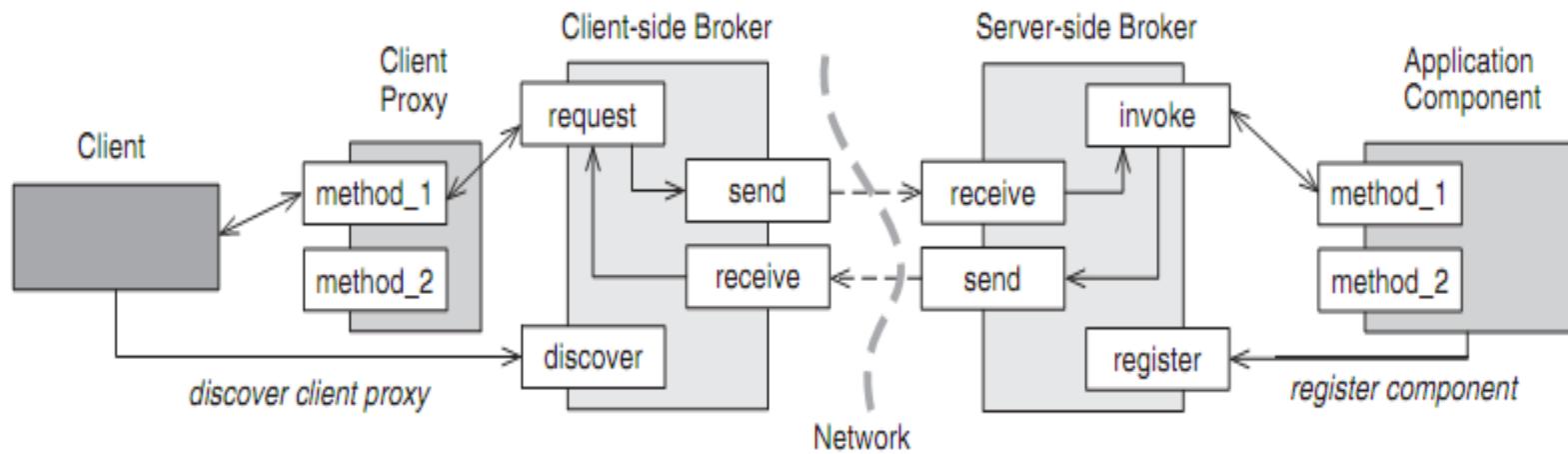


- Sursele de evenimente (*Event sources*) publică mesaje pe canale particulare pe un (*event bus*)
- Event listeners se abonează la anumite canale.
- Listenerii sunt notificați referitor la canalele care sunt publicate pe un canal la care s-au abonat.
- Generarea și notificarea de mesaje este **asincronă** :
  - un event source generează un mesaj – nu așteaptă ca acesta să fie primit de către listeneri

# Broker Pattern

- Sistemele distribuite pot avea dificultati (provocari) care nu apar in sistemele cu un proces
  - Important – codul aplicatiile nu trebuie sa trateze aceste probleme in mod direct => intermediari (brokers)
- aplicatiile trebuie simplificate prin modularizare *modular programming model* care poate ascunde detaliile de retea si locatie
- **Solutie:** Folosirea unei federatii de brokeri
  - *brokers to separate and encapsulate the details of the communication infrastructure in a distributed system from its application functionality.*
- Definirea unui model de tip *component-based programming model prin care clientii sa poate sa invoce metode sau servicii remote ca si cum ar fi locale*

# Broker



# Client-Proxy Pattern

- Atunci cand se construieste o infrastructura de tip broker *client-side* pentru o componenta *remote* trebuie sa se asigure o abstractizare care permite clientilor sa acceseze acele componente folosind ***remote method invocation***.
- un “Client Proxy / Remote Proxy” reprezinta o componenta remote- in the spatial de adrese al clientului.
- Proxy ofera o interfata identica care mapeaza invocarile de metode specifice catre functionalitatea orientate catre transmiterea de mesaje ale brokerului.
- ***Proxies allow clients to access remote component functionality as if they were collocated.***

# Event Demultiplexing & Dispatching

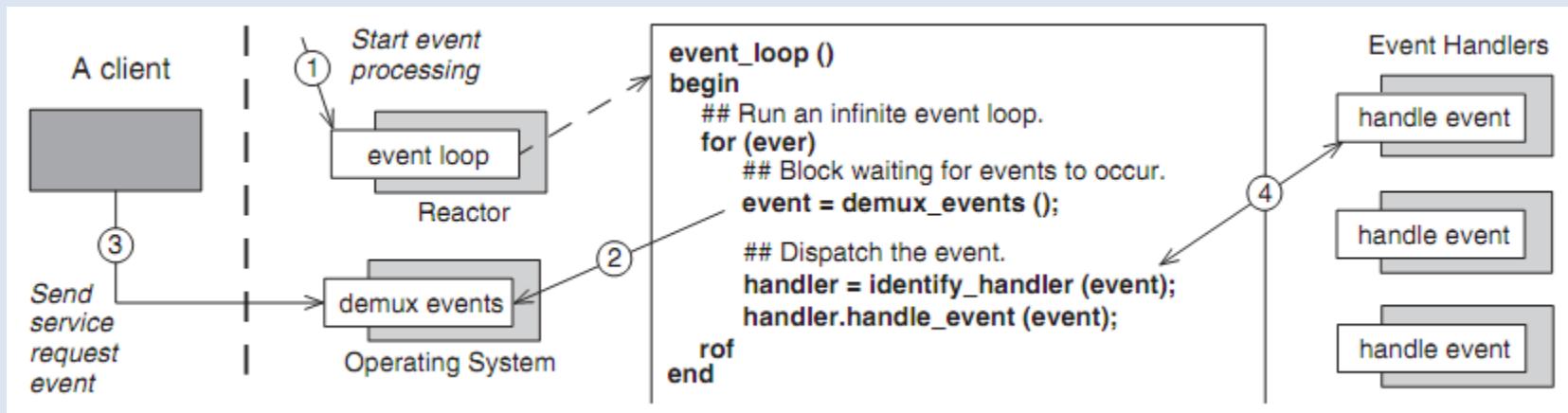
- At its heart, distributed computing is all about *handling and responding to events* received from the network.
- There are patterns that describe different approaches for initiating, receiving, demultiplexing, dispatching, and processing events in distributed and networked systems.
- ...two of these patterns:
  - Reactor ... synchronous
  - Proactor ... asynchronous

# Reactor Pattern

- Event-driven software often
  - receives service request events from multiple event sources, which it demultiplexes and dispatches to event handlers that perform further service processing.
- Events can also arrive simultaneously at the event-driven application.
  - To simplify software development, events should be processed sequentially | synchronously.

# Reactor

- **Solution:** Provide an event handling infrastructure that waits on multiple event sources simultaneously for service request events to occur, but only demultiplexes and dispatches **one event at a time** to a corresponding event handler that performs the service.



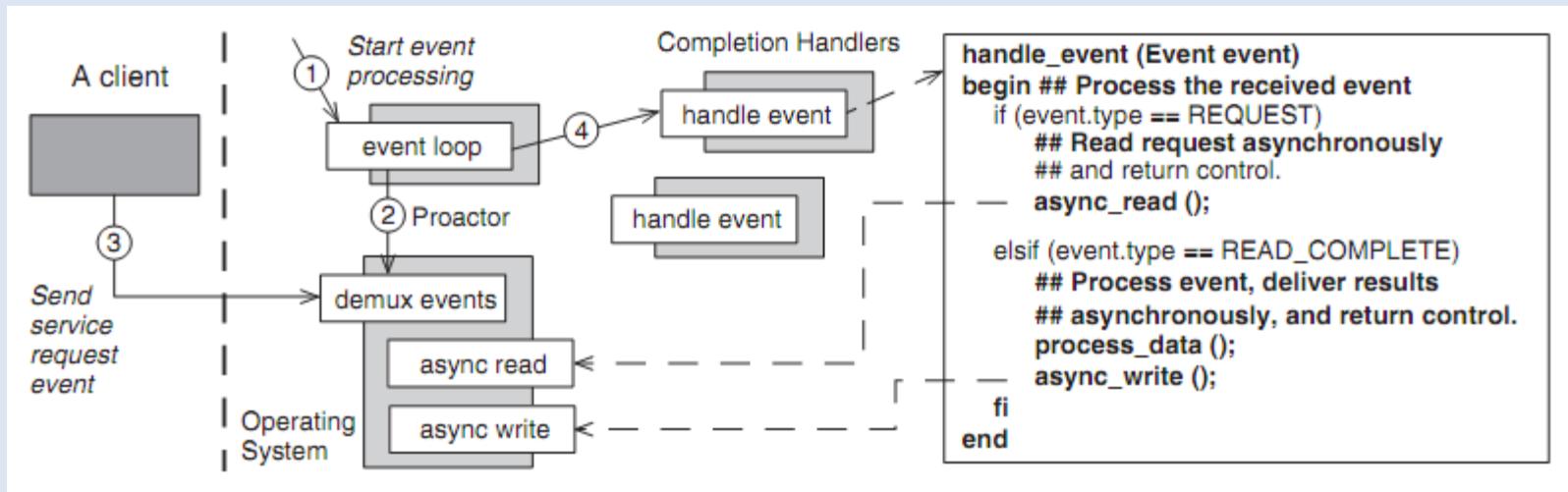
- It defines an **event loop** that uses an **operating system** event demultiplexer to wait **synchronously** for service request events.
- By **delegating the demultiplexing of events to the operating system**, the reactor can wait for multiple event sources simultaneously without a need to multi-thread the application code.

# Proactor Pattern

- To achieve the required performance and throughput, event-driven applications must often be able ***to process multiple events simultaneously.***
- However, **resolving this problem via multi-threading, may be undesirable, due to the overhead of synchronization, context switching and data movement.**
- **Solution:**
  - ***Split an application's functionality into***
    - ***asynchronous operations*** that perform activities on event sources and
    - ***completion handlers*** that ***use the results of asynchronous operations to implement application service logic.***
  - Let the operating system execute the asynchronous operations, but
  - execute ***the completion handlers in the application's thread of control.***

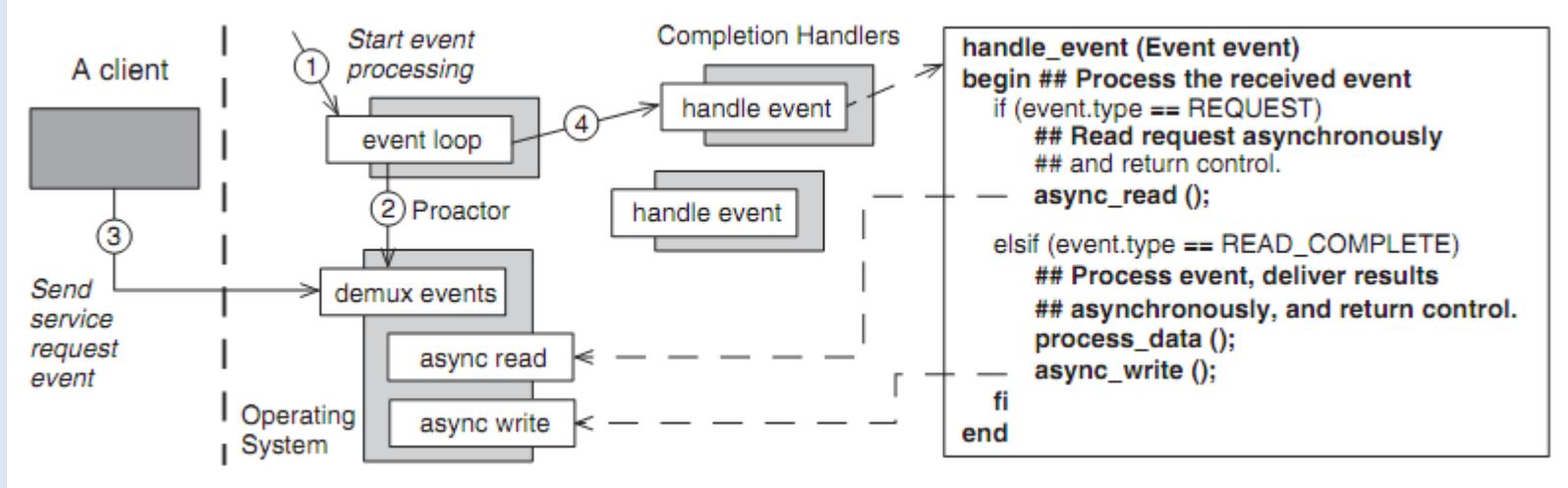
# Proactor

- A proactor component coordinates the collaboration between completion handlers and the operating system.
  - It defines an event loop that uses an operating system event demultiplexer to wait synchronously for events that indicate the completion of asynchronous operations to occur.



- Initially all completion handlers ‘proactively’ call an asynchronous operation to wait for service request events to arrive, and then run the event loop on the proactor.

# Proactor



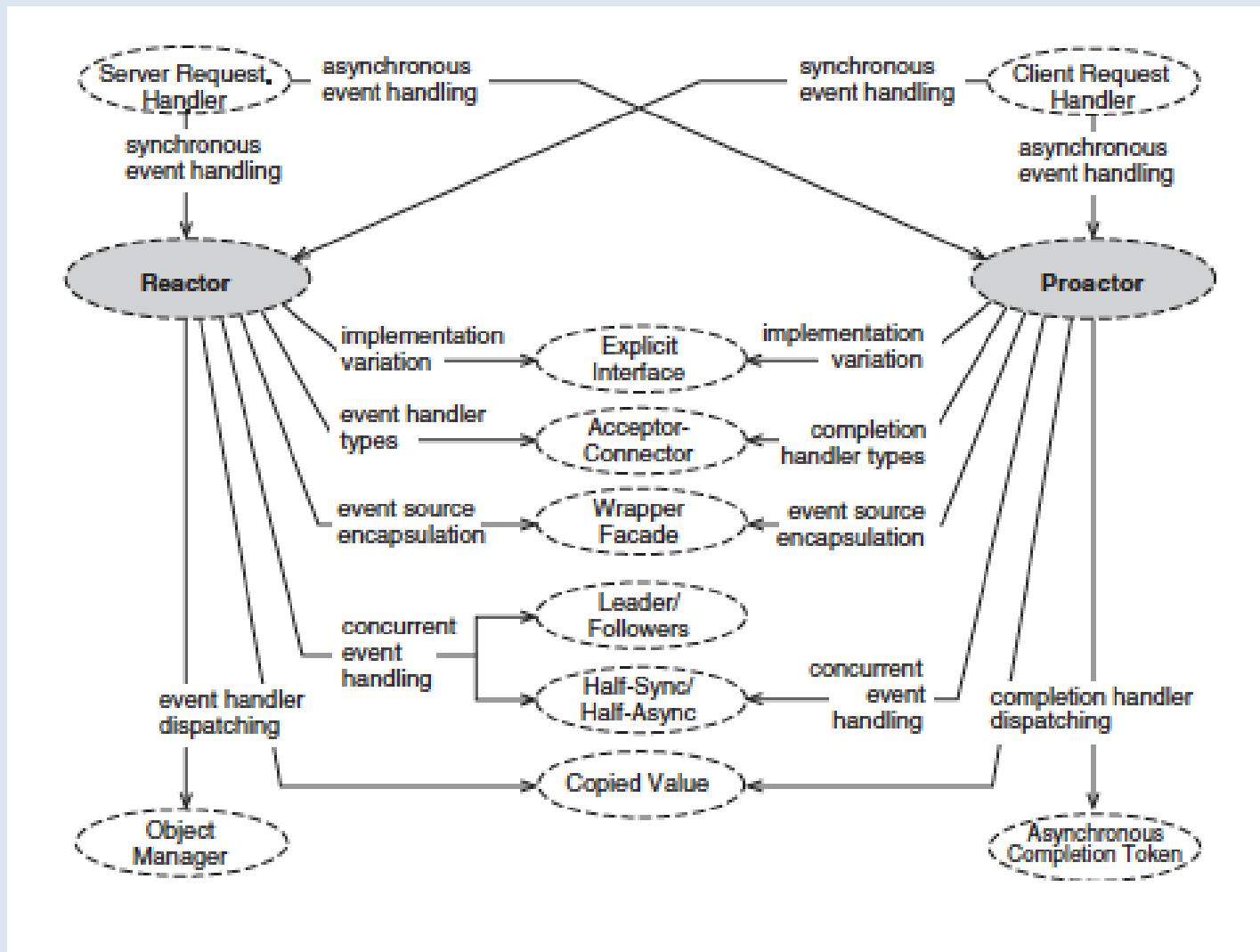
- When such an event arrives, the proactor dispatches the result of the completed asynchronous operation to the corresponding completion handler.
- This handler then continues its execution, which may invoke another asynchronous operation.

# Reactor vs. Proactor

- Although both patterns resolve essentially the same problem in a similar context, and also use similar patterns to implement their solutions, the concrete event-handling infrastructures they suggest are distinct, due to the orthogonal forces to which each pattern is exposed.
- REACTOR focuses on simplifying the programming of event-driven software.
  - It implements a **passive** event demultiplexing and dispatching model in which services wait until request events arrive and then react by processing the events synchronously without interruption.
  - While this model scales well for services in which the duration of the response to a request is short, it can introduce performance penalties for long-duration services, since executing these services synchronously can unduly delay the servicing of other requests.

# Reactor vs. Proactor

- PROACTOR is designed to maximize event-driven software performance.
  - It implements a more *active* event demultiplexing and dispatching *model* in which services divide their processing into multiple self-contained parts and
  - *proactively initiate asynchronous execution* of these parts.
  - This design allows multiple services to execute concurrently, which can increase quality of service and throughput.
- REACTOR and PROACTOR are not really equally weighted alternatives, but rather are *complementary patterns* that trade-off programming simplicity and performance.
  - Relatively simple event-driven software can benefit from a REACTOR-based design, whereas
  - PROACTOR offers a more efficient and scalable event demultiplexing and dispatching model.



# Middleware

- In computer science, a middleware is a software layer that resides between the application layer and the operating system.
- Its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, to coordinate how parts of applications are connected and how they inter-operate.
- Middleware also enables and simplifies the integration of components developed by different technology suppliers.

# Middleware

Examples of a middleware for distributed object-oriented enterprise systems: CORBA, SOAP.

- Despite their detailed differences, middleware technologies typically follow one or more of three different communication styles:
  - Messaging
  - Publish/Subscribe
  - Remote Method Invocation

# Referinte

- Frank Buschmann. Kevlin Henney.Douglas C. Schmidt.**Pattern-Oriented Software Architecture**,Volume 4: A Pattern Language for Distributed-Computing.Wiley & Sons, 2007
- George Coulouris. Jean Dollimore. Tim Kindberg.Gordon Blair. DISTRIBUTED SYSTEMS: Concepts and Design. Fifth Edition. Addison-Wesley.