

# Curs 1

Programare Paralela si Distribuita

# Continutul cursului

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

## 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*

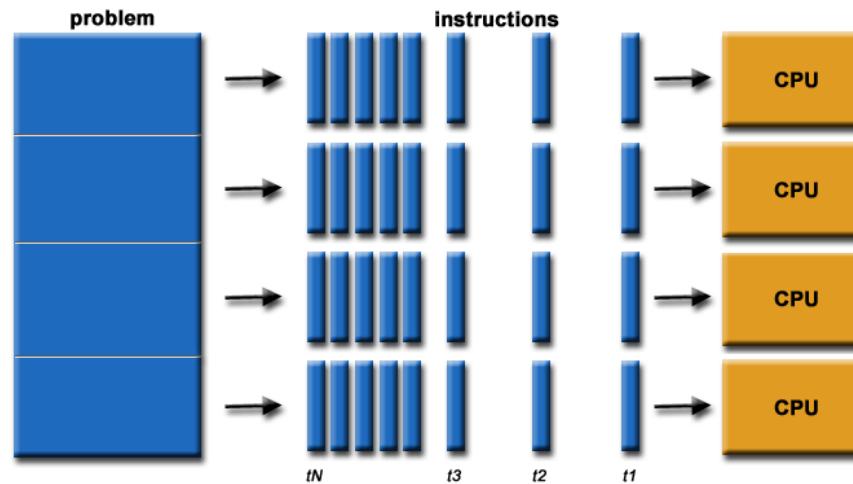
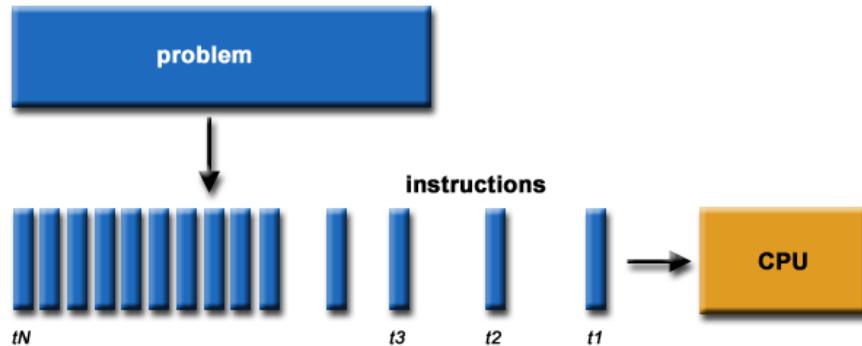
- Laborator
  - (40%) NL->Programe/proiecte **NL >=4.5**
  - (10%) NS-> Exercitii in timpul laboratorului (“in class”)
- (10%) NT -> Test practic
- (40%) NE -> Examen teoretic **NE>=4.5**
  
- Informatii curs
  - <http://www.cs.ubbcluj.ro/~vniculescu/didactic/PPD/CursPPD.html>

# 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 ...
- Paralelismul este natural.
- 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.

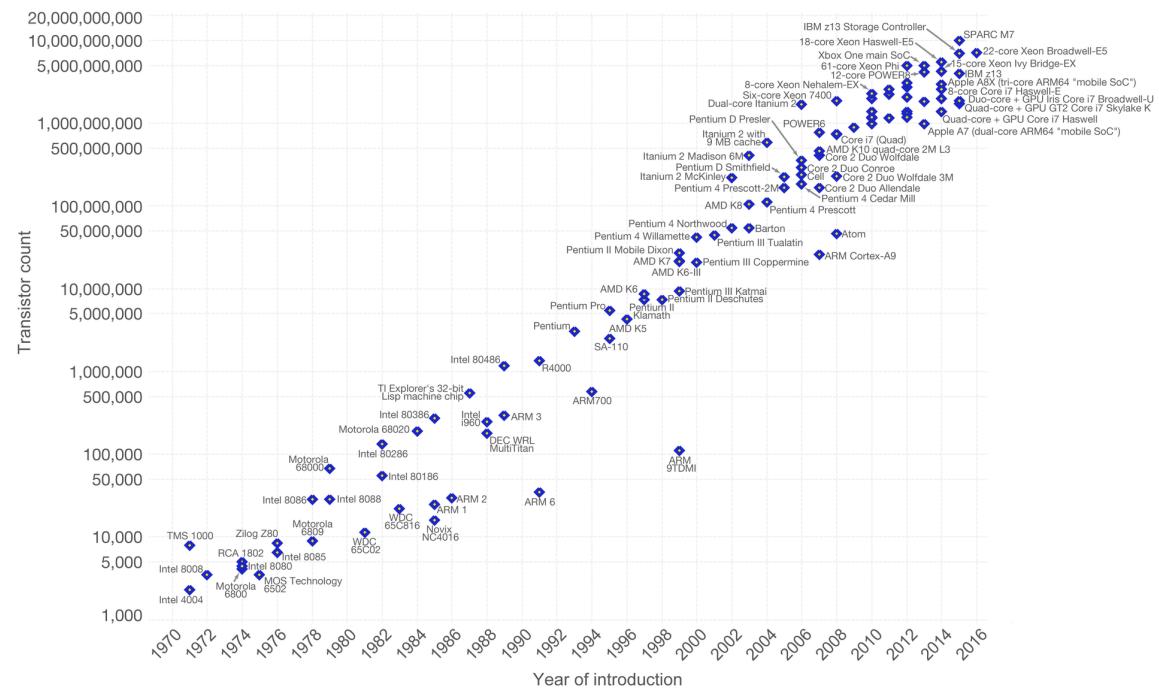
- Legea lui Moore:  
numărul de tranzistori  
care pot fi plasati pe un singur  
circuit integrat  
(per square inch chip)  
se dubleaza la fiecare 2 ani.

- Impune costuri mari.

- #### • Limitari economice

## Moore's Law – The number of transistors on integrated circuit chips (1971-2016) Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



# Istoric

- Cresterea performantei procesor prin cresterea frecventei ceasului CPU (CPU clock frequency)
  - Ridind Moore's law
- Probleme : incalzirea puternica a chipurilor!
  - Frecventa ceas mai mare  $\Rightarrow$  consum electric mai mare  
*(Pentium 4 heat sink    ↶ Frying an egg on a Pentium 4)*
- 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;
- in anumite cicluri;

## 3. paralelism la nivel de instrucțiune:

- intre diferite faze de execuție ale unei instrucțiuni;

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

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

- Arhitecturile curente se bazeaza tot mai mult pe paralelism la nivel hardware pentru a imbunatati performanta :
  - Multiple execution units
  - Pipelined instructions
  - Multi-core

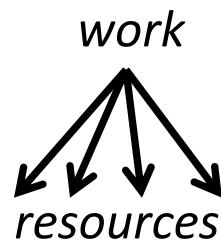
# Paralelism <-> Concurinta

- 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

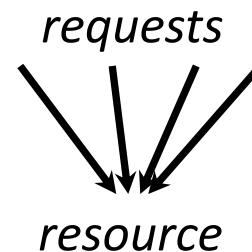
## Paralelism:

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

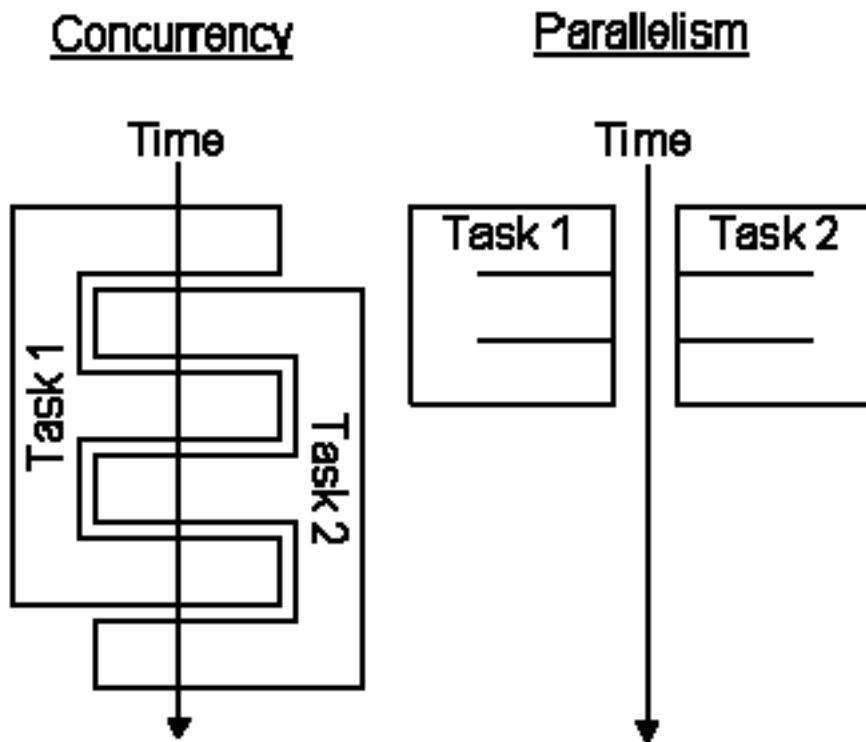
# Concurrenta si Paralelism

- 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 paralelism:
  - 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)

# 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, ...

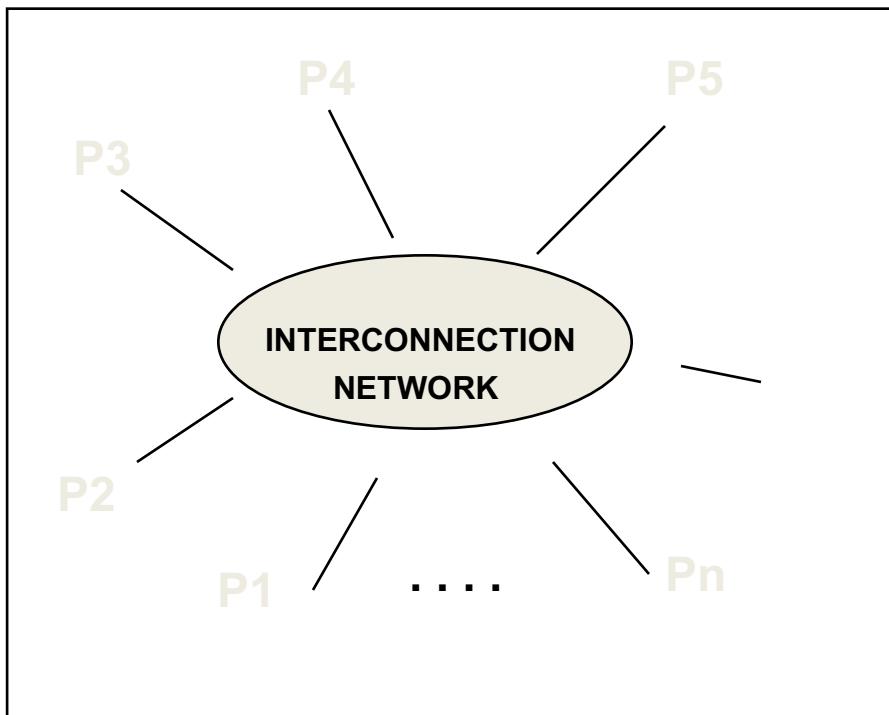
# De ce sa studiem programare paralela?

- 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 FRECVENTA 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 NECESSARA

## **PARALLEL COMPUTERS: COOPERARE IN SENS “POSITIV”**

**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 pt 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, adică 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

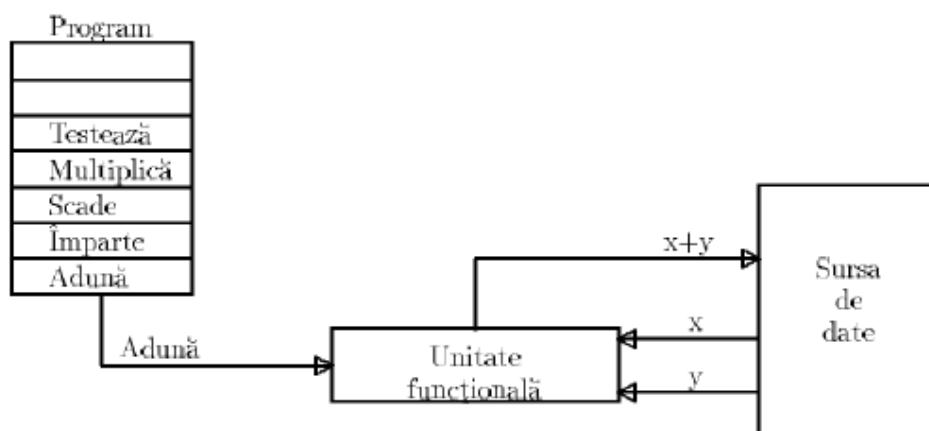
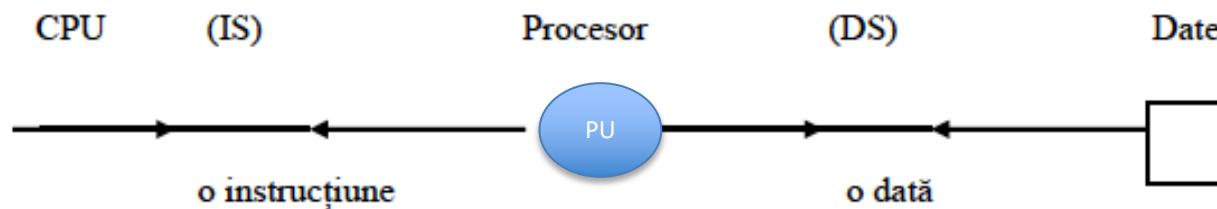
- SISD: sistem cu un singur flux de instrucțiuni și un singur flux de date;
- SIMD: sistem cu un singur flux de instrucțiuni și mai multe fluxuri de date;
- MISD: sistem cu mai multe fluxuri de instrucțiuni și un singur flux de date;
- MIMD: cu mai multe fluxuri de instrucțiuni și mai multe fluxuri de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

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

# SISD(Single instruction stream, single data stream)

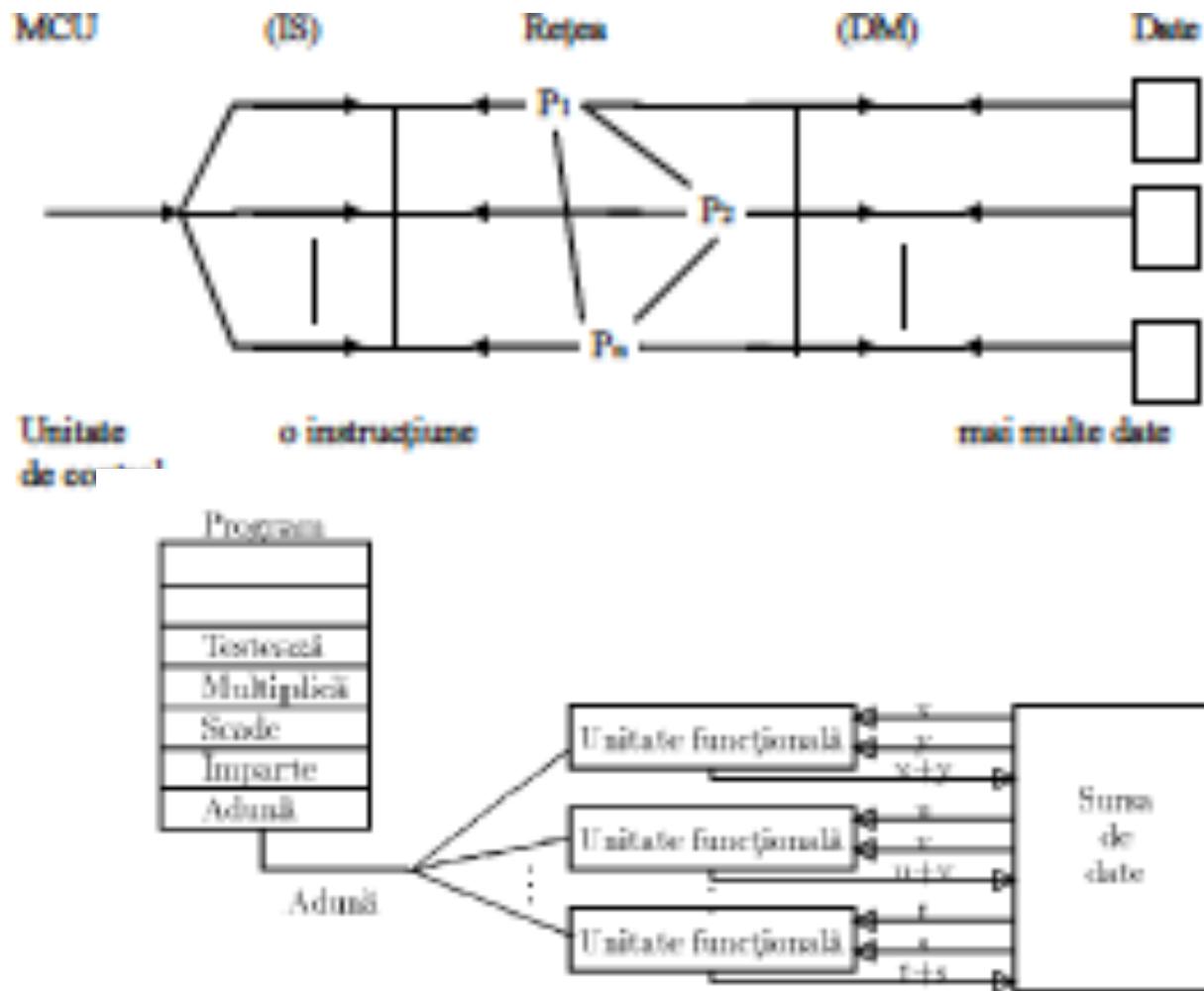
Flux de instrucțiuni singular, flux de date singular (SISD)-

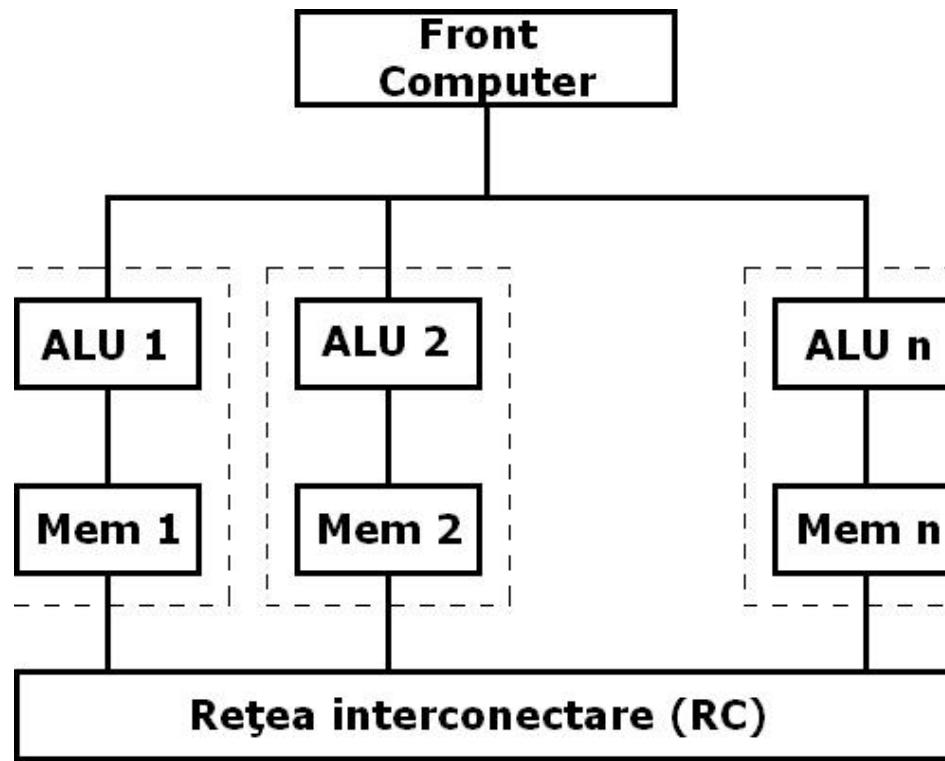
- microprocesoarele clasice cu arhitecturi von Neumann
- Funcționare ciclică: preluare instr., stocare rez. în mem. , etc.



# SIMD (Single instruction stream, multiple data stream)

Flux de instrucțiuni singular, flux de date multiplu





# Executie conditionala in 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

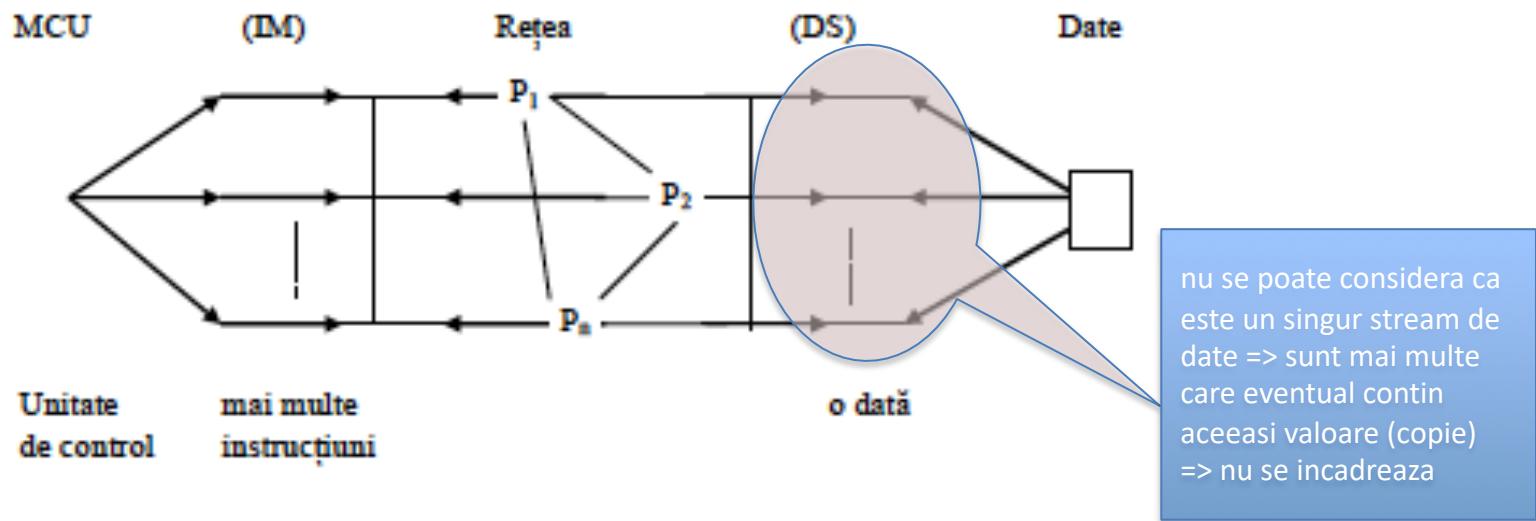
Step 2

(b)

# MISD (multiple instruction stream, single data stream)

Flux de instrucțiuni multiplu, flux de date singular

- multime vida



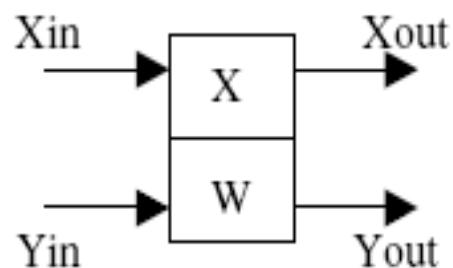
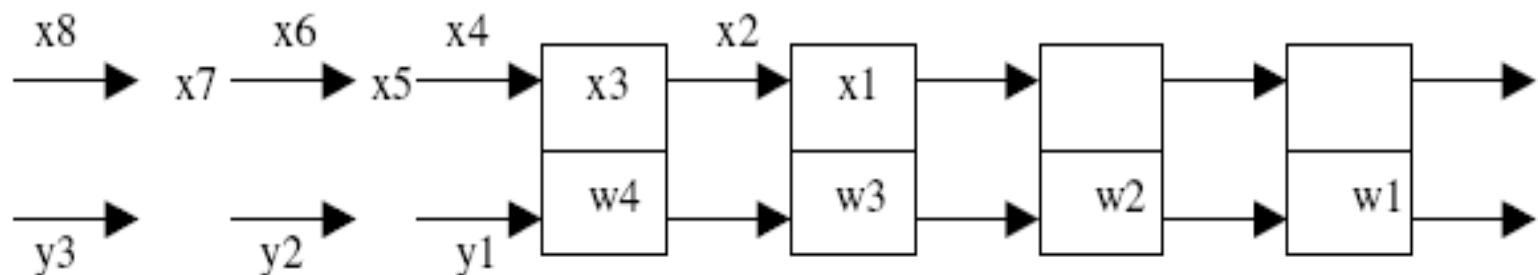
? ~ procesoare pipeline:

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, utilizând o rețea liniară de elemente de prelucrare:

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



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

*Rețea liniară pentru calcularea conoluțiilor.*

# 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

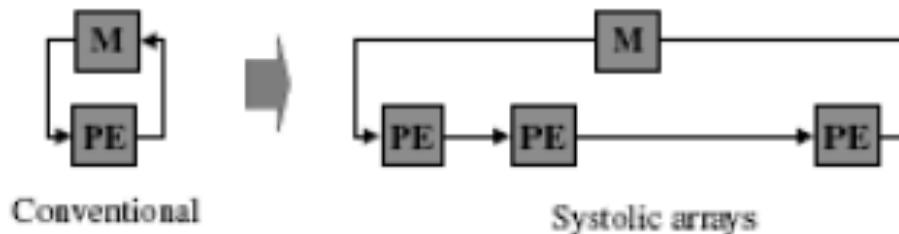
## Initial motivation

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

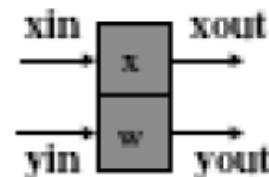
$$y_i = \sum_{j=1}^n a_{ij}x_j, i = 1, \dots, n$$



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

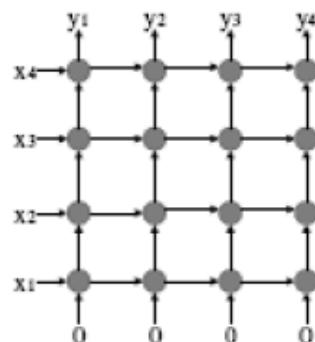


$$x_{\text{out}} = x$$

$$x = x_{\text{in}}$$

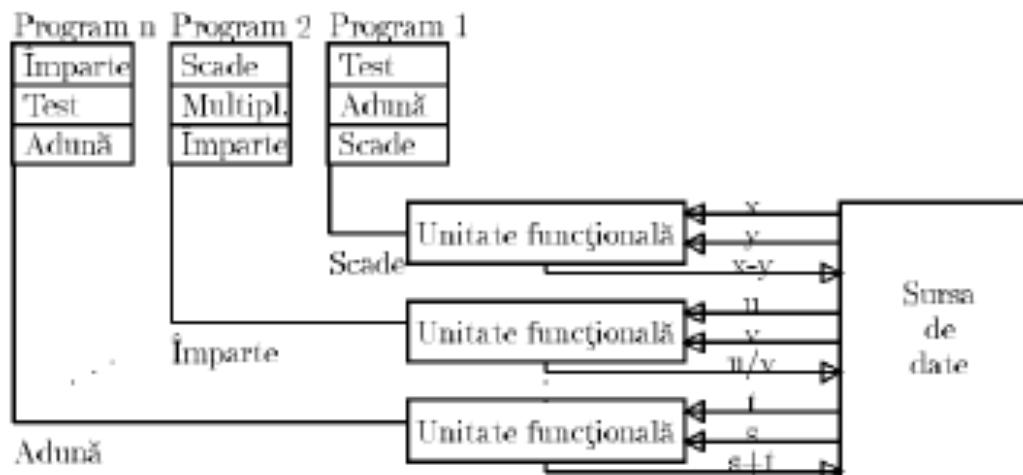
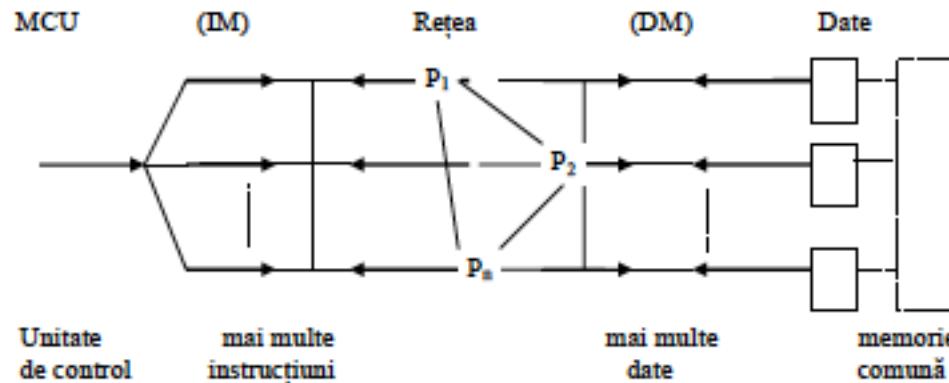
$$y_{\text{out}} = y_{\text{in}} + w * x_{\text{in}}$$

## Systolic Array Representation of Matrix Multiplication



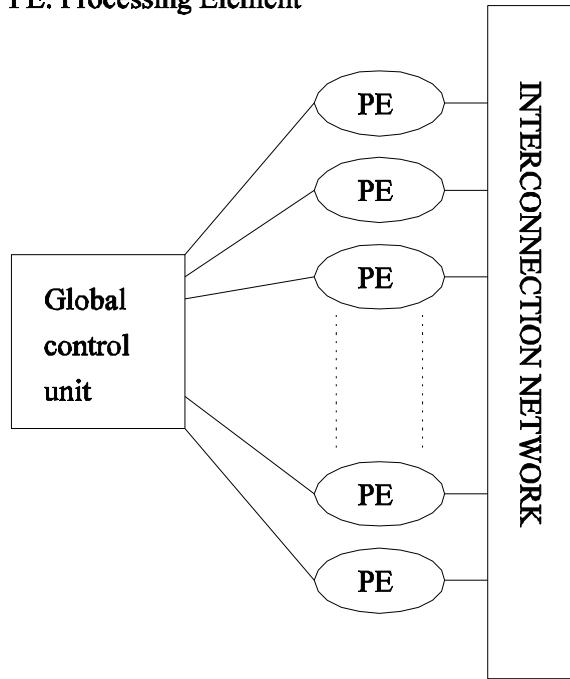
# MIMD (multiple instruction stream, multiple data stream)

Flux de instrucțiuni multiplu, flux de date multiplu

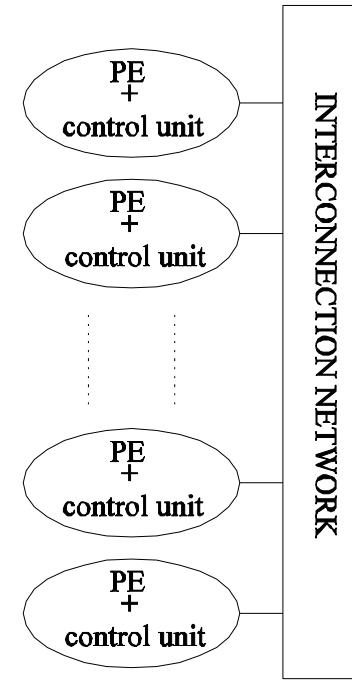


# SIMD versus MIMD

PE: Processing Element

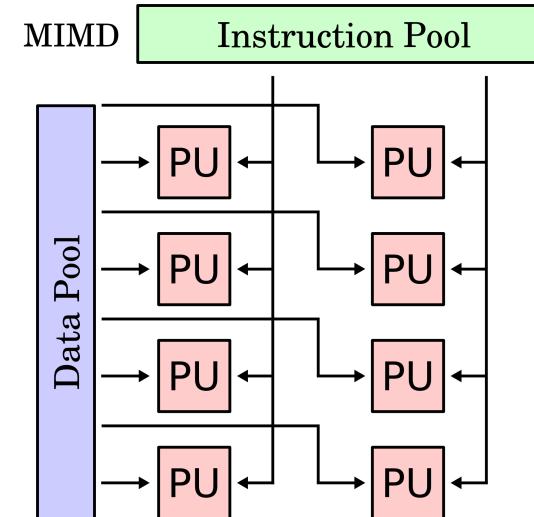
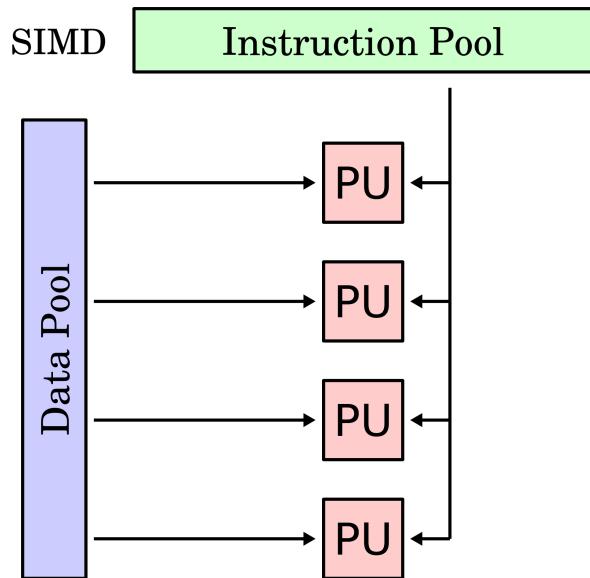
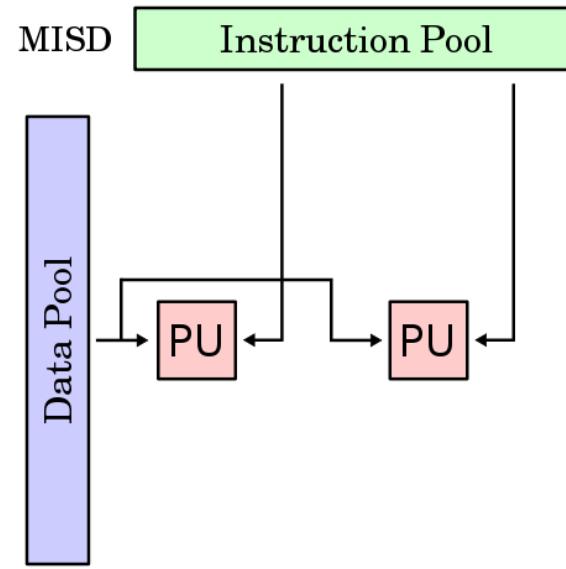
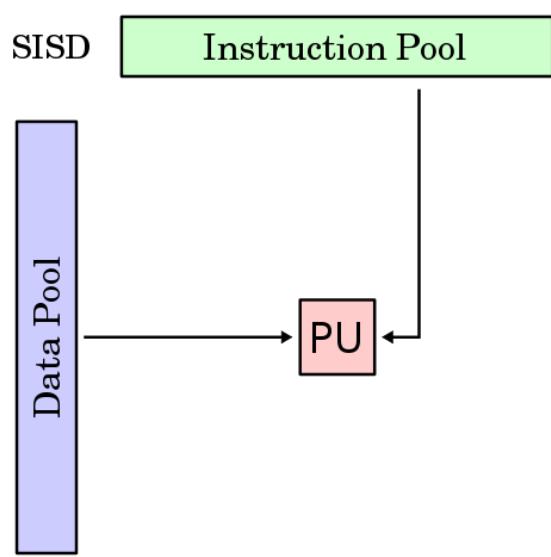


(a)



(b)

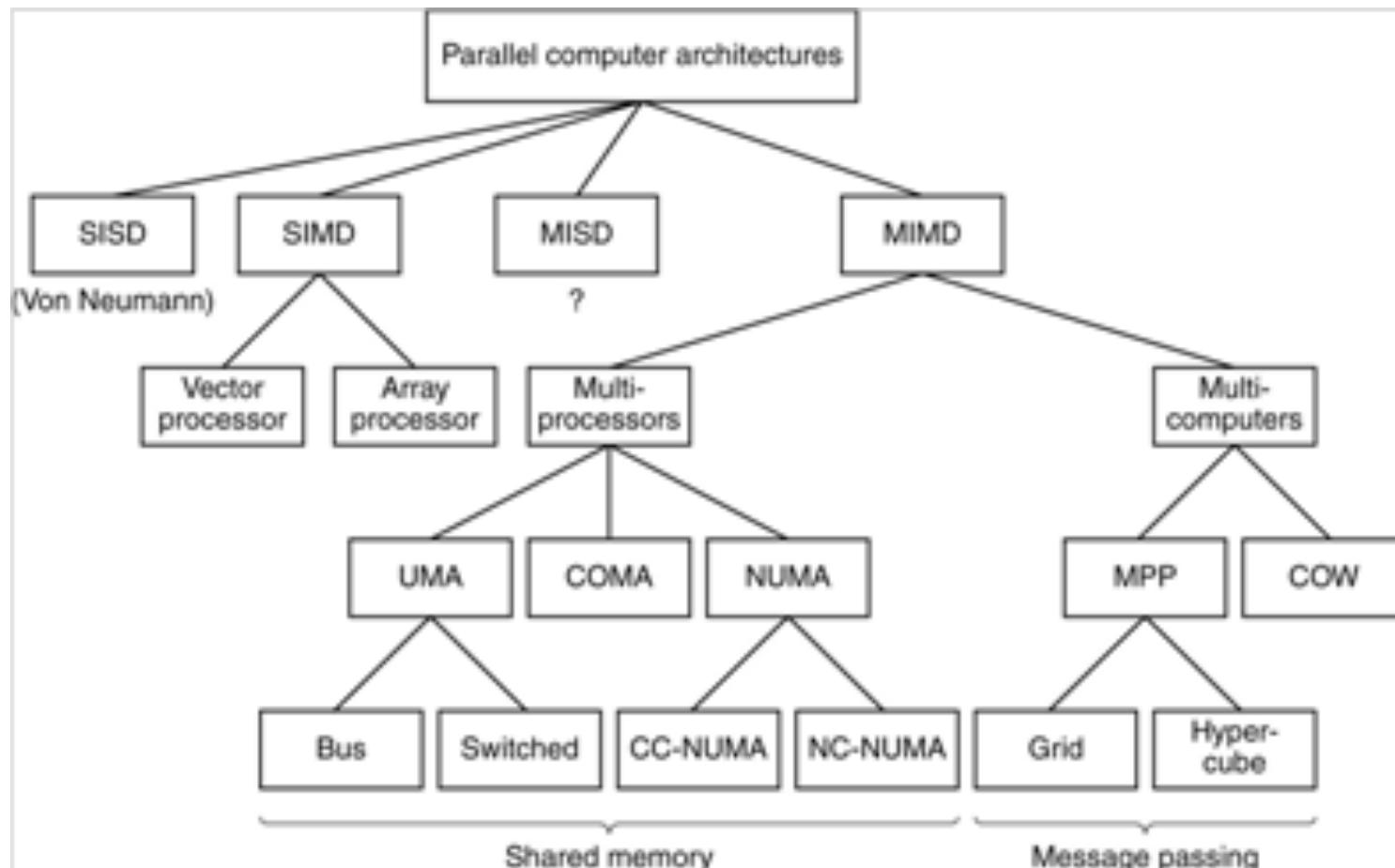
# Scheme Comparative – clasificare Flynn



# Paralelizare la nivel hardware – istoric

- Etapa 1 (1950s): executie secv 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

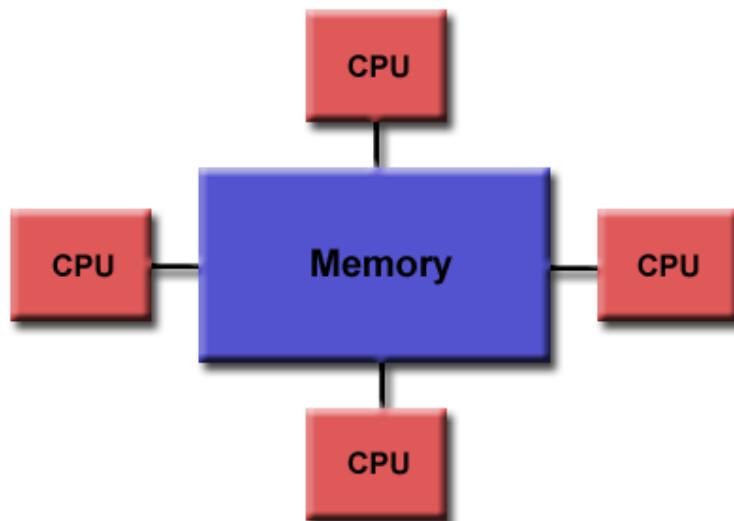
- Clasificare in functie de tipul de memorie
  - partajata
  - distribuita
  - hibrida

# 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**.

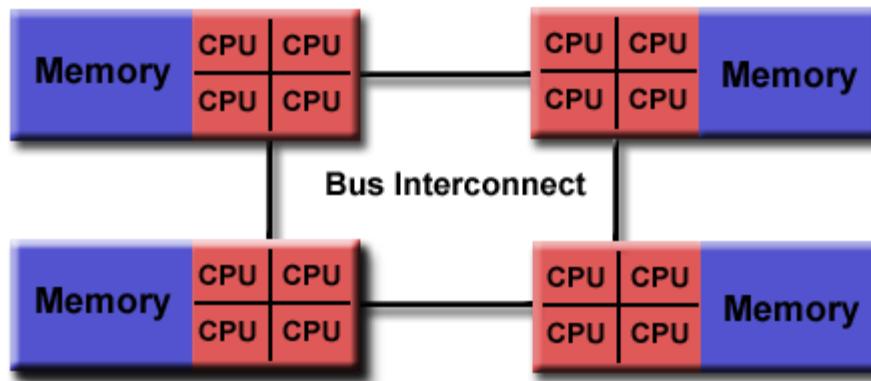
# Shared Memory (UMA)

- **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.



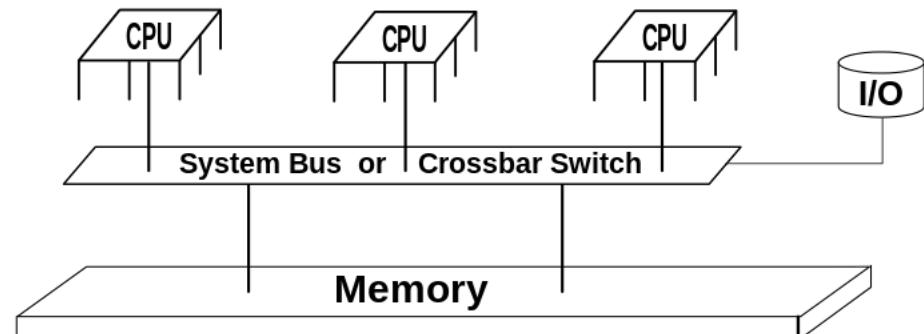
# Non-Uniform Memory Access (NUMA):

- Se obtine deseori prin unirea a 2 sau mai multe arh. 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



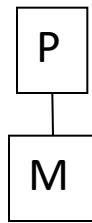
# ***SMP Symmetric multiprocessor computer***

- 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

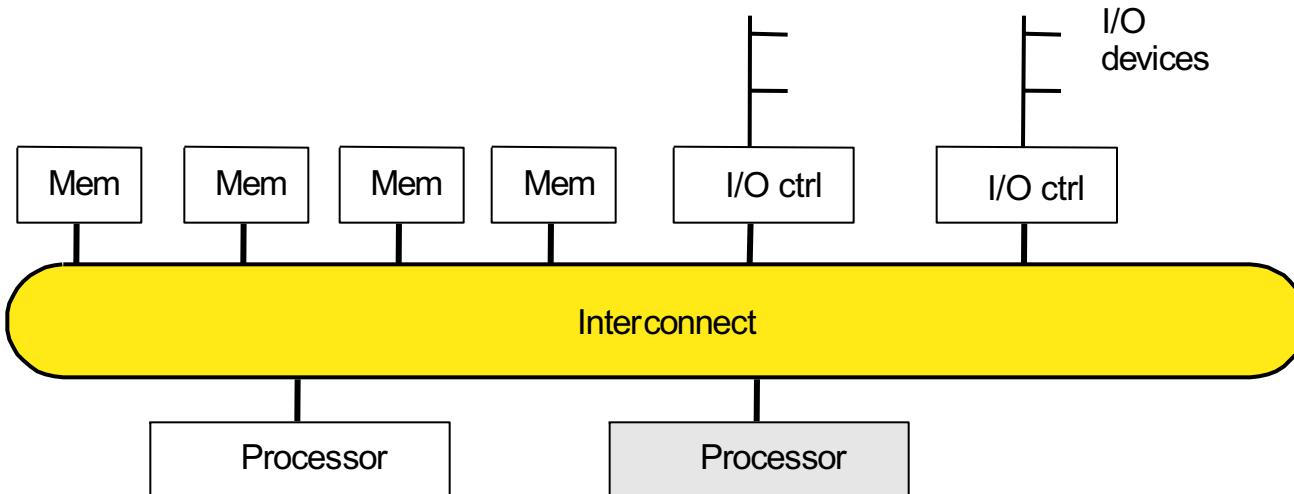
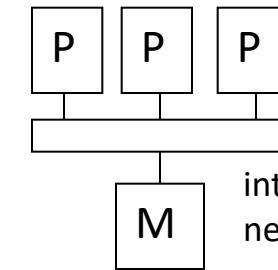
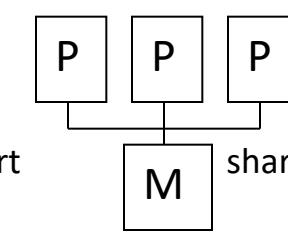
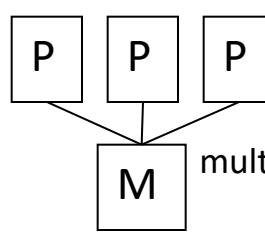


# Shared Memory Multiprocessors (SMP) - overview

Single processor

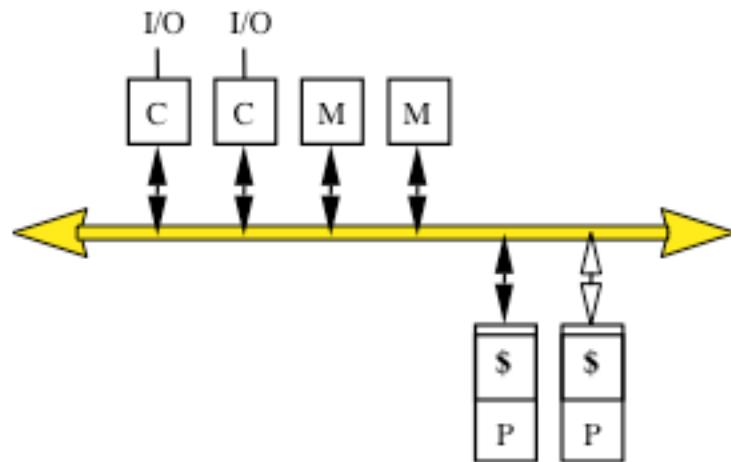


Multiple processors

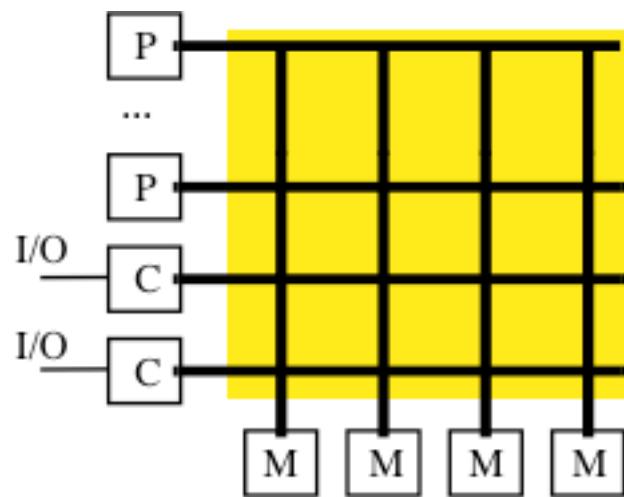


# Bus-based SMP(Symmetric Multi-Processor)

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



# Crossbar SMP



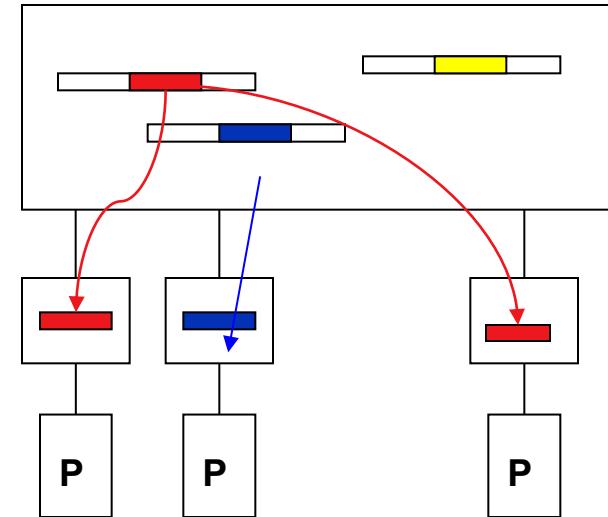
# Parametrii de performanta corespunzatori accesului la memorie

- 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*

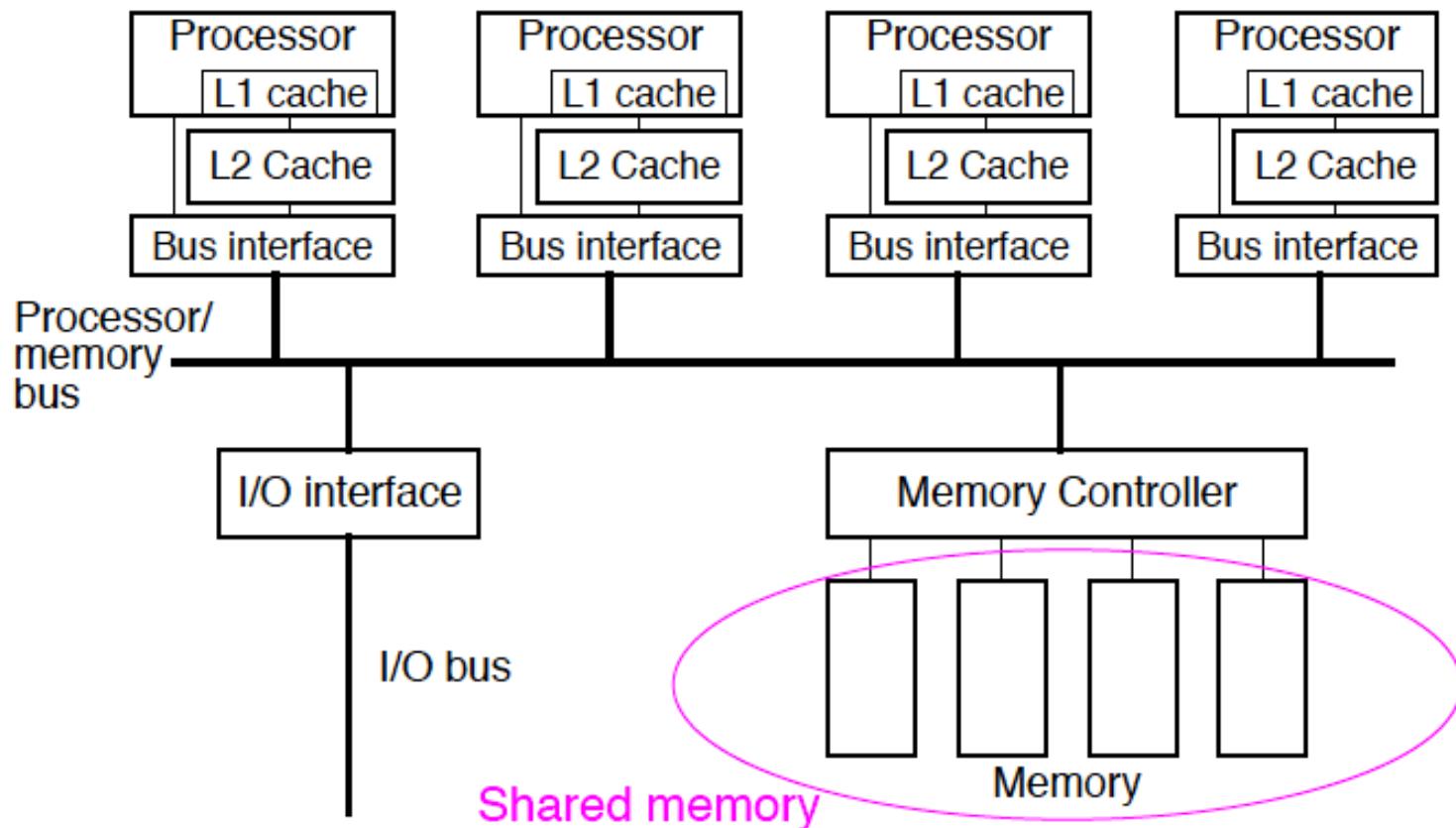
- Folosirea memorilor cache intr-un system de tip SPM introduce probleme legate de ***cache coherency***:

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

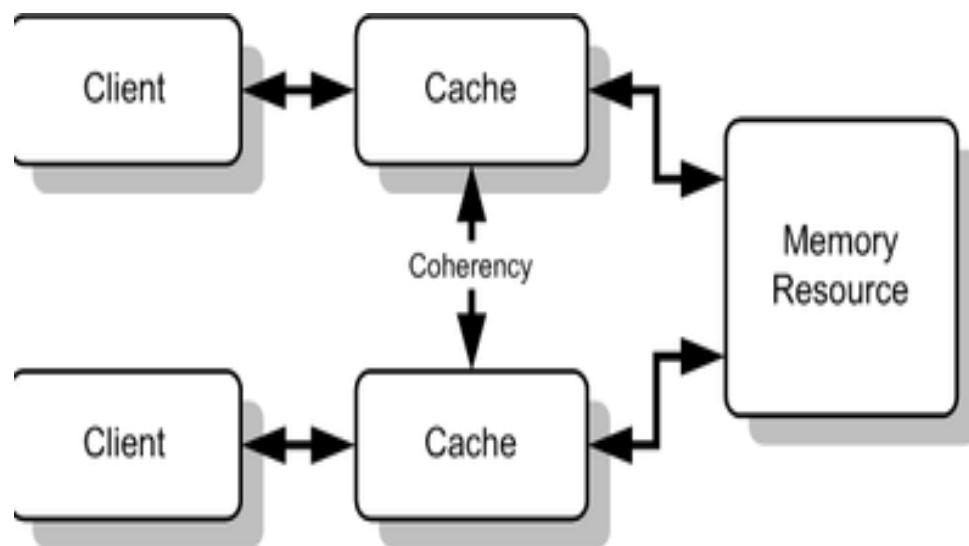


- *coherency* diminueaza scalabilitatea
  - shared memory systems=> maximum 60 CPUs (2016).

# 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

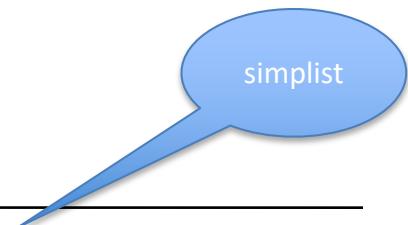
- Coerenta implica faptul ca scrierile la o locatie devin vizibile tuturor procesoarelor in aceeasi ordine .
  - cum se stabileste ordinea dintre o citire si o scriere?
    - Sincronizare (event based)
    - Implementarea unui protocol hardware pentru *cache coherency*.
    - Protocolul se poate baza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

$P_1$                        $P_2$

---

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

A = 1;           while (flag == 0); /* spin idly */
flag = 1;        print A;
```



# 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:**

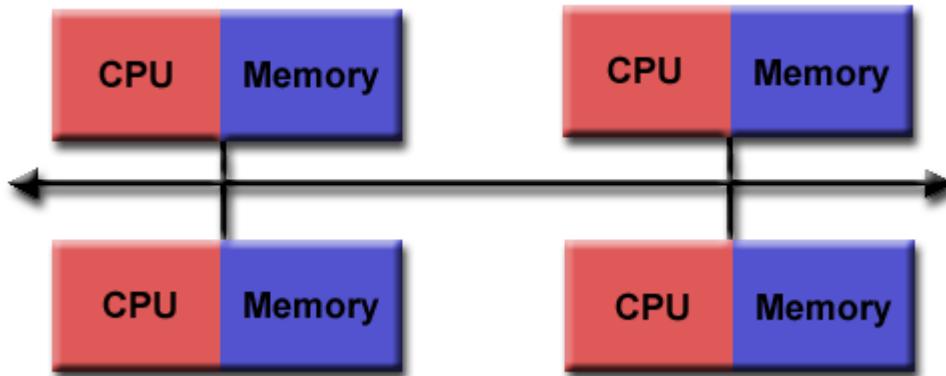
- *Global address space*
- *Partajare date rapida si uniforma*

## **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:**

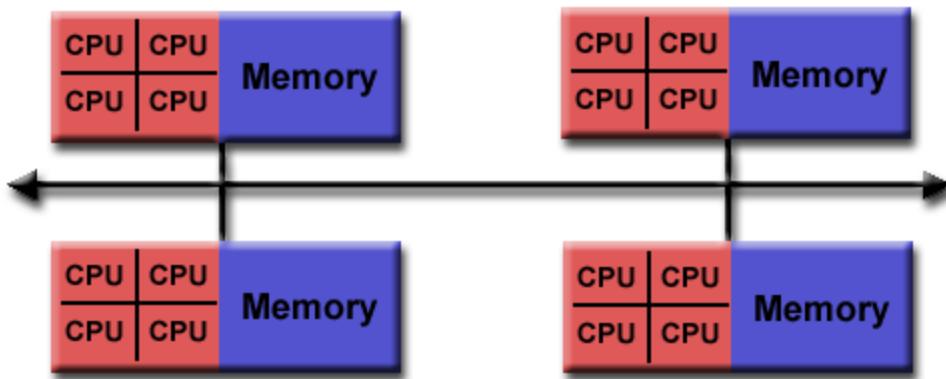
- Memorie scalabila – odata cu cresterea nr de procesoare
- Cost redus – retele

## **Dezavantaje:**

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

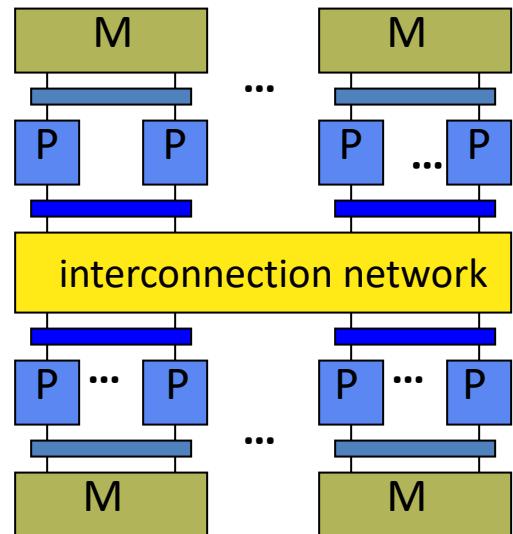
# Hybrid Distributed-Shared Memory

- 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)

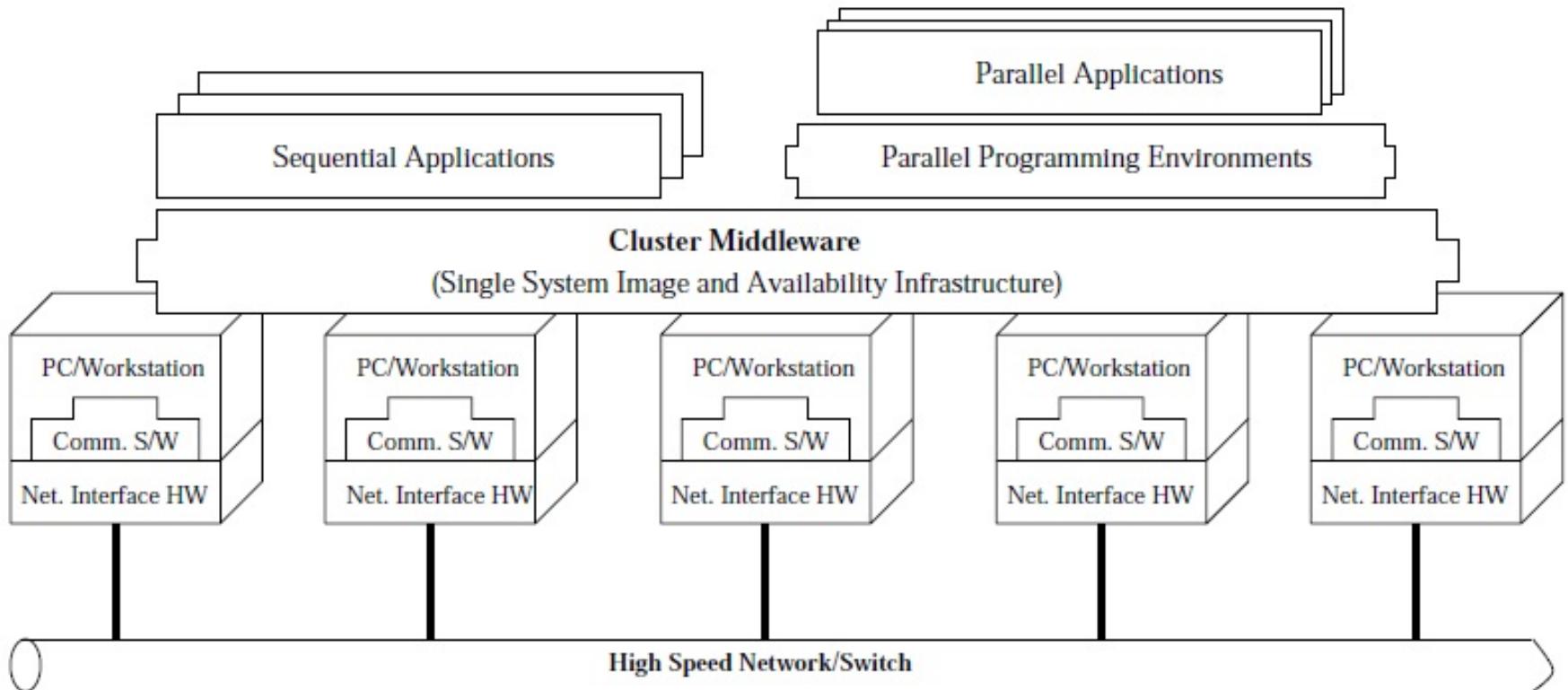
- Fiecare nod este un sistem independent care are local:
  - Memorie fizica
  - Spatiu de adresare
  - Disc local si conexiuni la retea
  - Sistem de operare
- *MPP (massively parallel processing) is the coordinated processing of a program by multiple processors that work on different parts of the program, 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.*

# 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



# 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*

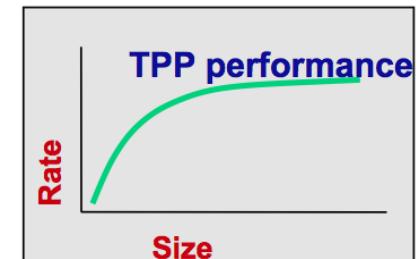
# Scalabilitatea sistemelor de calcul

- 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*)

# 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

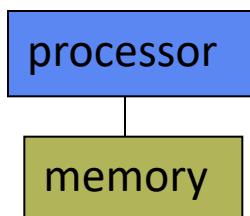
Vedere actuală asupra tipurilor de arhitecturi paralele

# Parallel Architecture Types

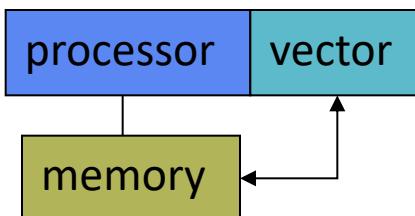
imagini preluate de la course pres. Introduction to Parallel Computing CIS 410/510, Univ. of Oregon

- Uniprocessor

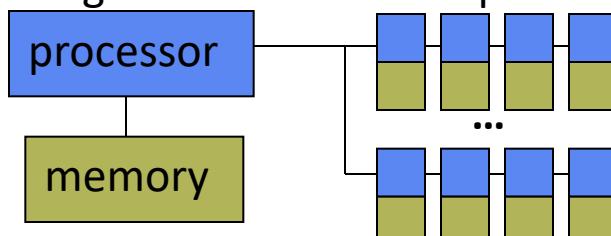
- Scalar processor



- Vector processor



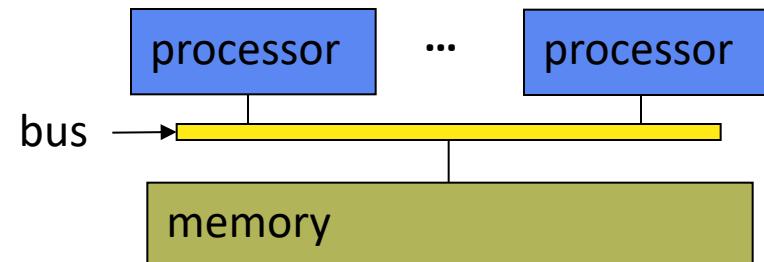
- Single Instruction Multiple Data



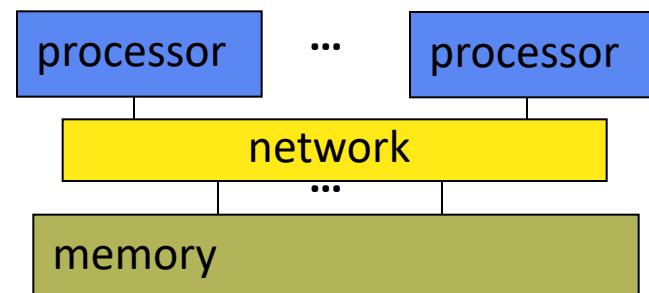
- Shared Memory

- Multiprocessor (SMP)

- Shared memory address space
  - Bus-based memory system

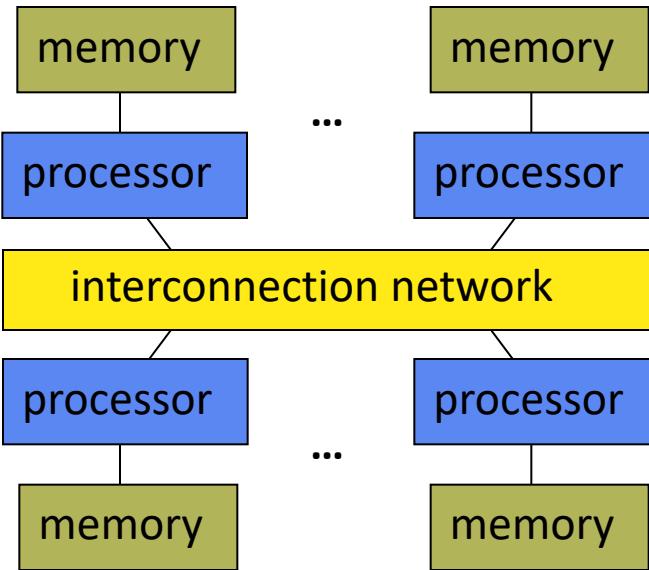


- Interconnection network

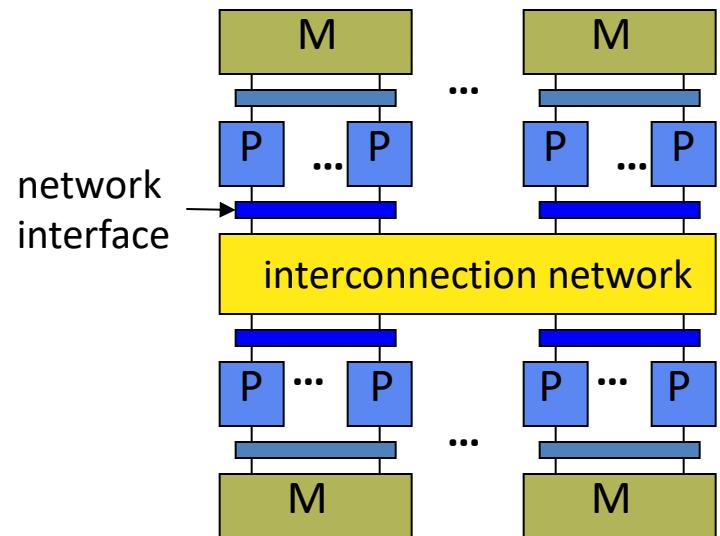


# Parallel Architecture Types (2)

- 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

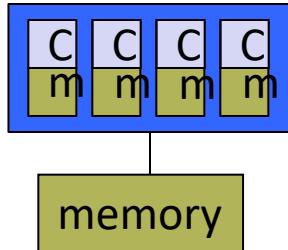


- Can also be regarded as MPP if processor number is large

# Parallel Architecture Types (3)

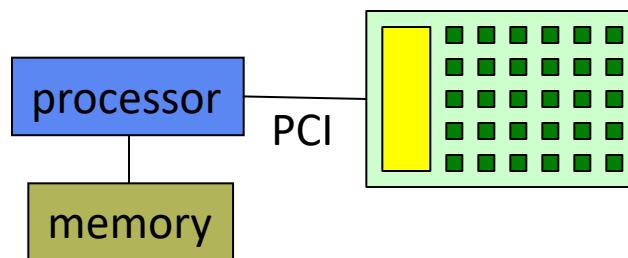
## □ Multicore

- Multicore processor

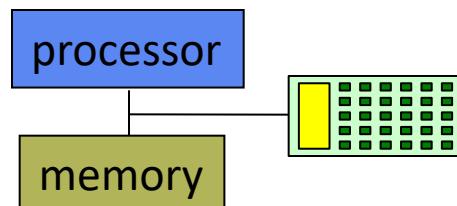


cores can be hardware multithreaded (hyperthread)

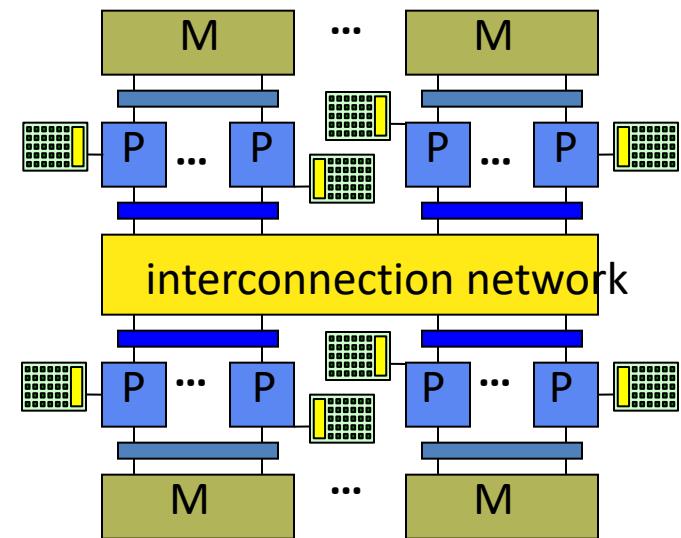
- GPU accelerator



- “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						
	IF	ID	EX	MEM	WB		
1							
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

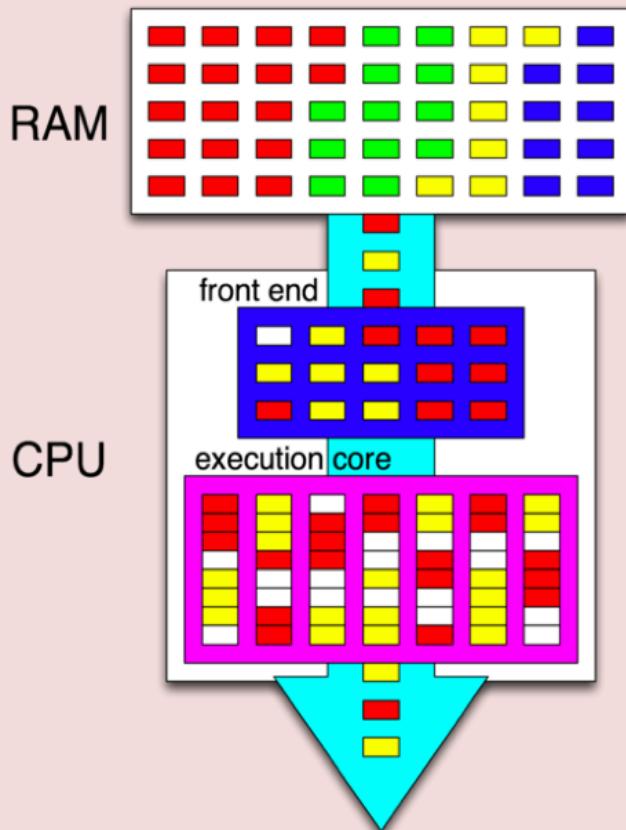
# ILP in a superscalar architecture

- The primary difference between a **superscalar** and a **pipelined** processor is (a superscalar processor is also pipeline) 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 pipelines 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 a 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.

## PARALLELISM LA NIVEL DE PROGRAM

# Paralelism implicit SAU explicit

- *Implicit Parallelism*

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

- *Explicit Parallelism*

Programatorul specifica explicit paralelismul in codul sursa prin  
constructii 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 Modele de Programare si Arhitecturi**

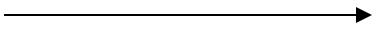
Exemplificare pe problema concreta:

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:

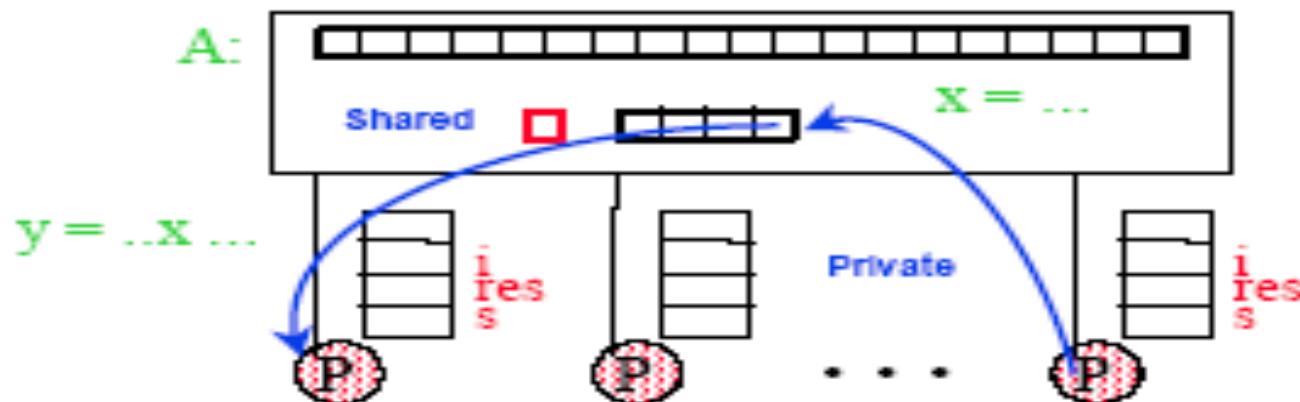
- partajate: valorile  $A[i]$  si suma finala;
- private: evalurile individuale de functii si sumele partiale

## 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.

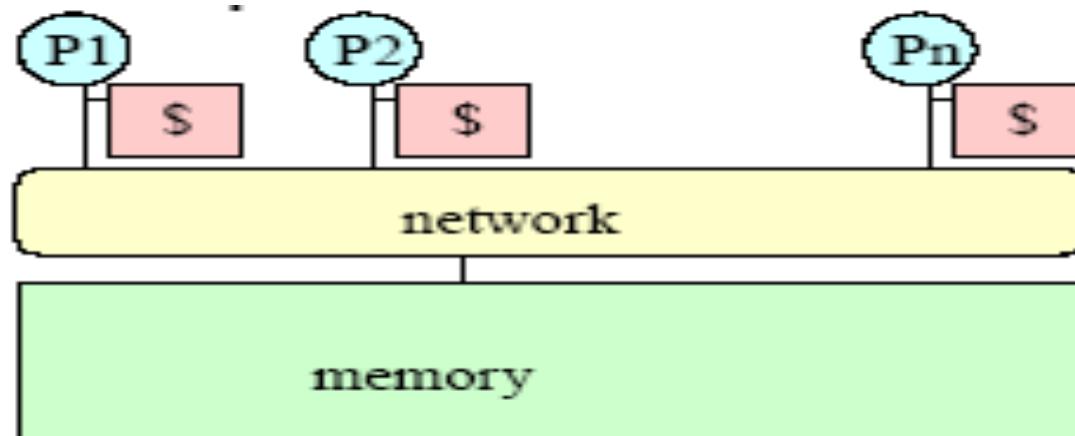
**Comunicatia** dintre firele de executie: **prin citirea/scrierea variabilelor partajate**.

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



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

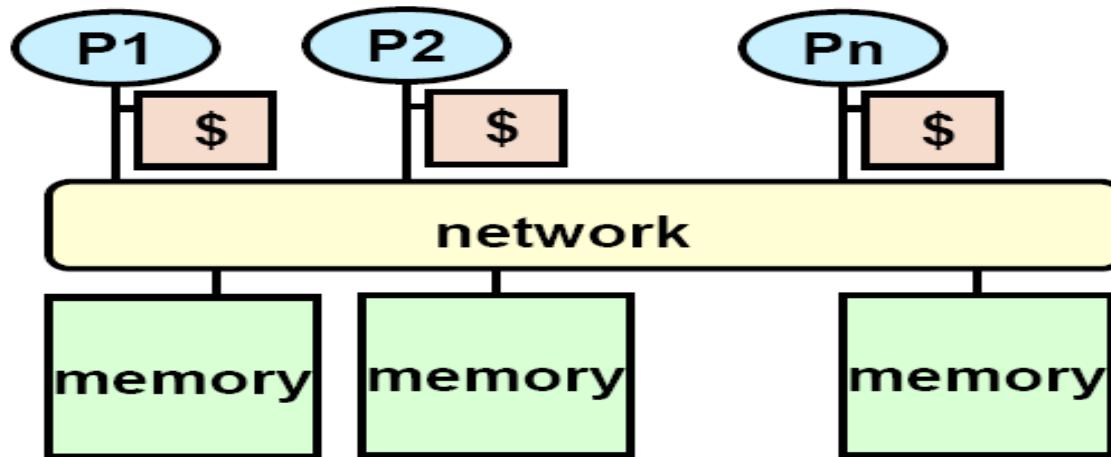
Exemple: sisteme de la Sun, DEC, Intel (Millennium), SGI Origin.



Variante ale acestui model:

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

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



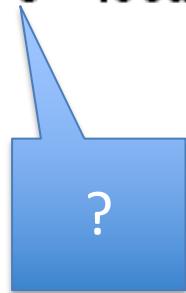
b) masina cu spatiu partajat de adrese (memoriile cache inlocuite cu memorii locale). Exemplu: Cray T3E.

*single address space*

O posibila solutie pentru rezolvarea problemei:

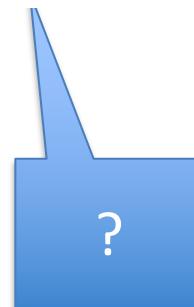
### Thread 1

```
[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
```

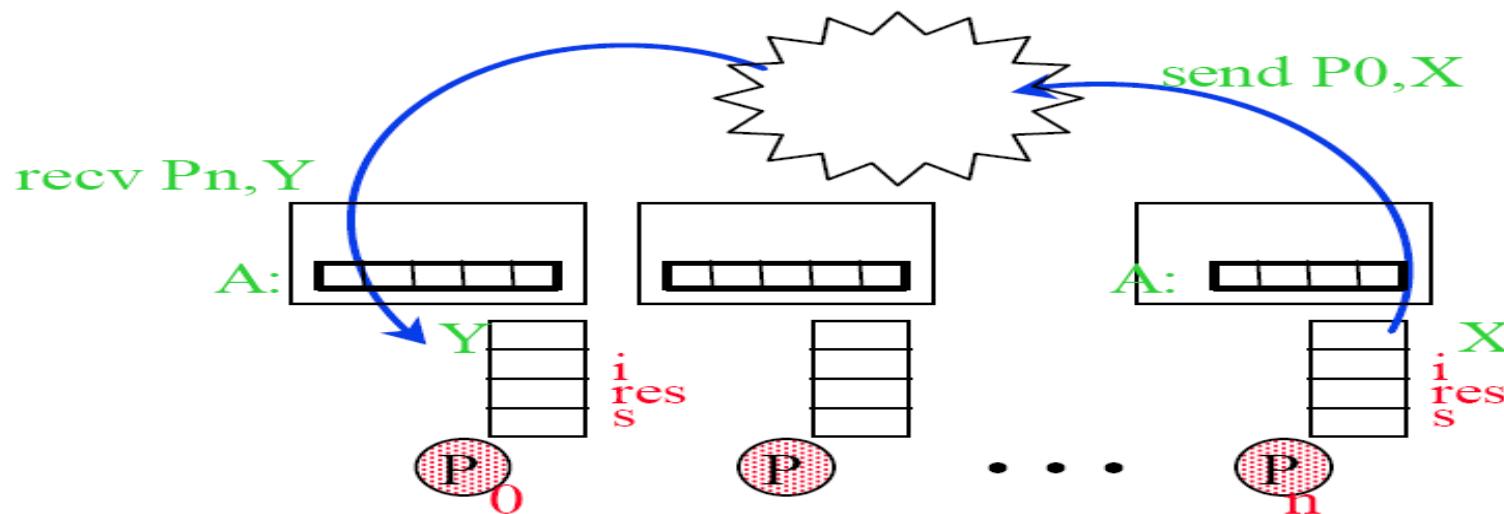
## 2) Modelul de programare: transfer de mesaje.

**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).

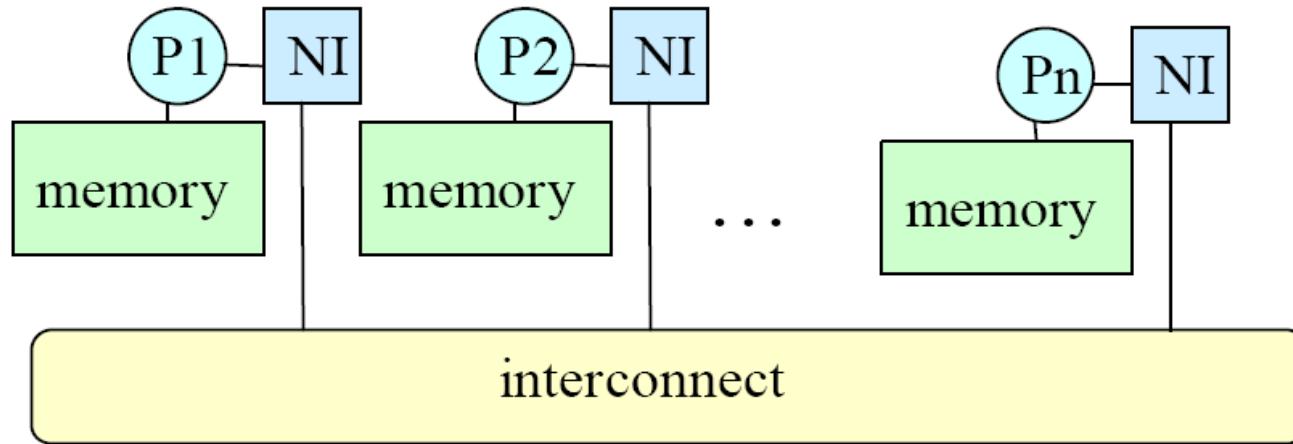
**Coordonarea**: transfer de mesaje

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



Există biblioteci standard (exemplu: MPI și 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**

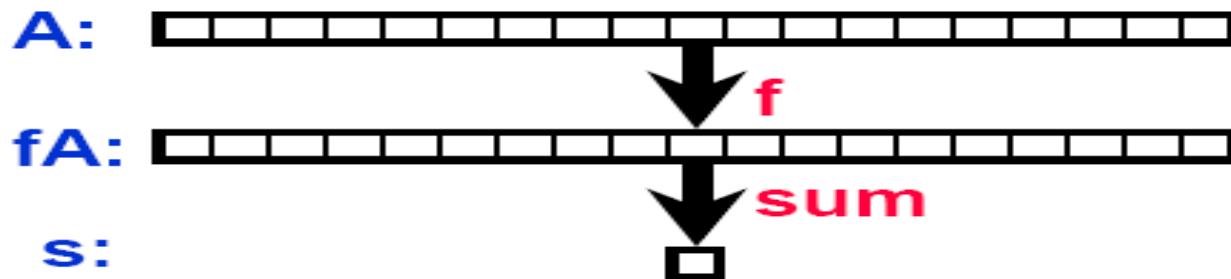
**s = xlocal + xremote**



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.

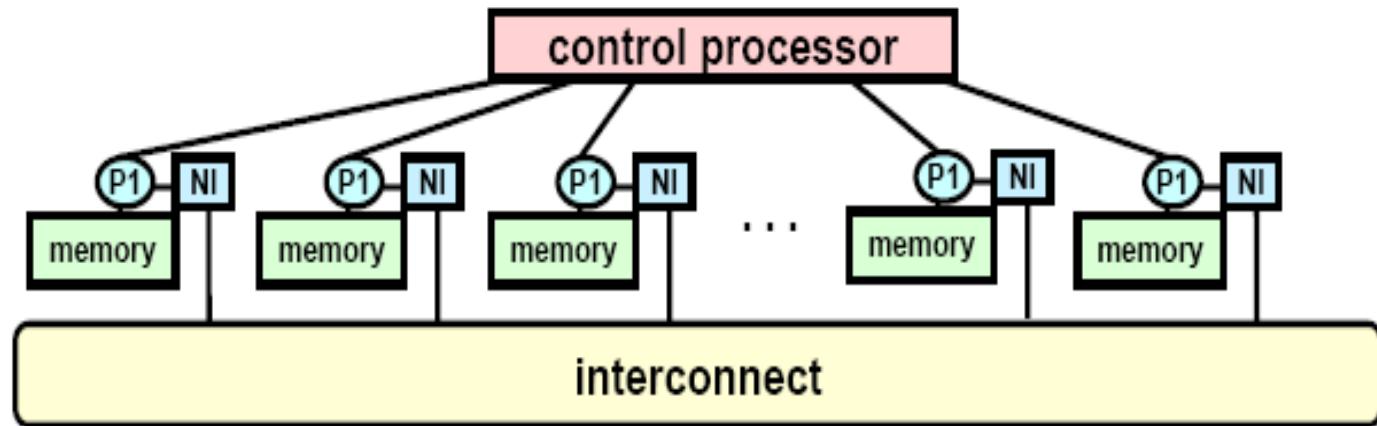


map + reduce

**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).

**4) Modelul 4 de masina: 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).

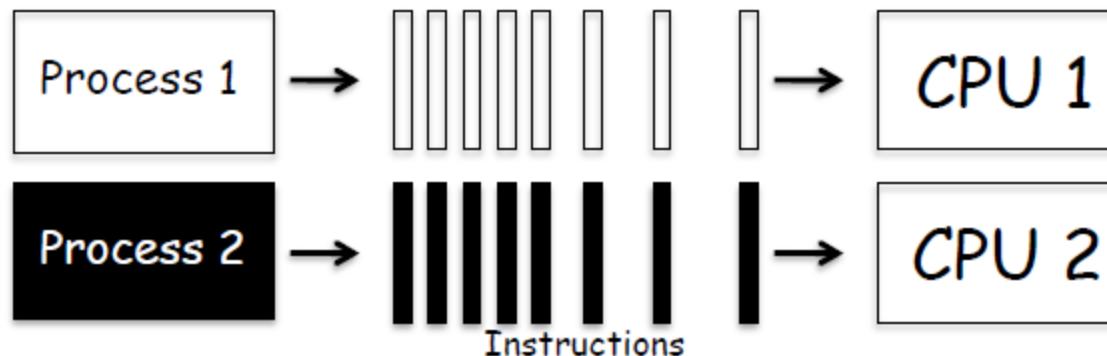
### **Model de programare:**

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

## Procese versus Fire de executie

# *Multiprocessing -> Paralelism*

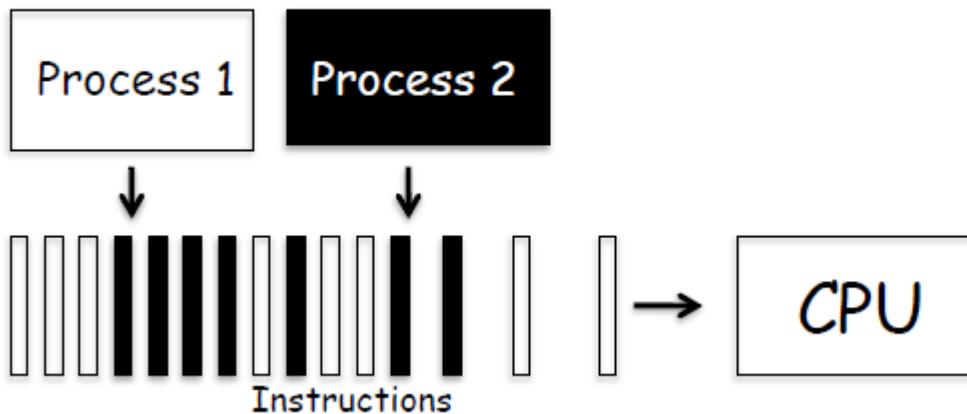
- *Multicore processors*



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

# Multitasking-> Concurenta

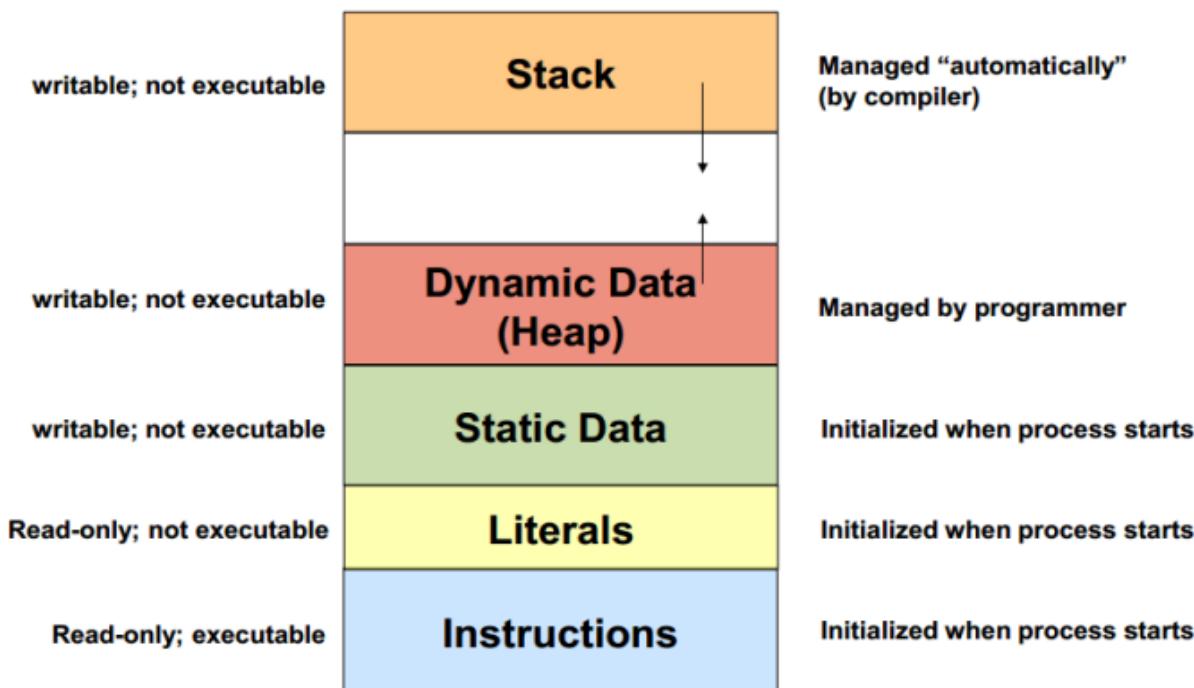
- Sistemul de operare schimba executia intre diferite taskuri
- Resursa comună = CPU



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

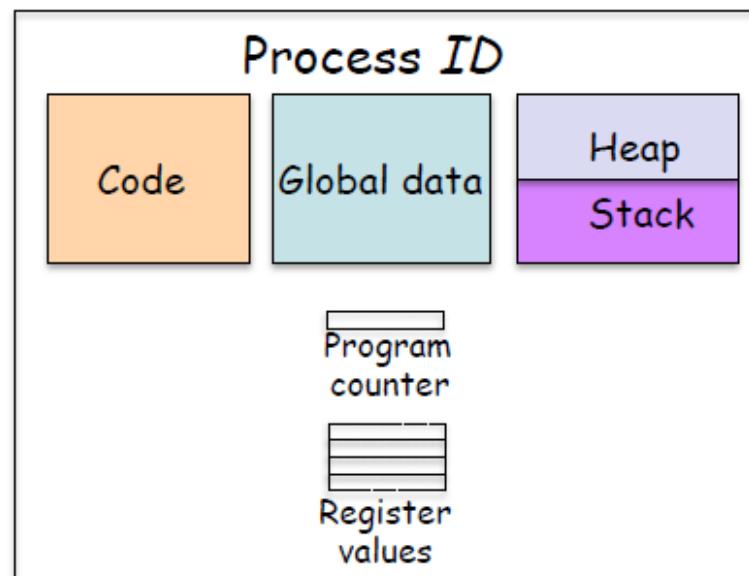
# Procese

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



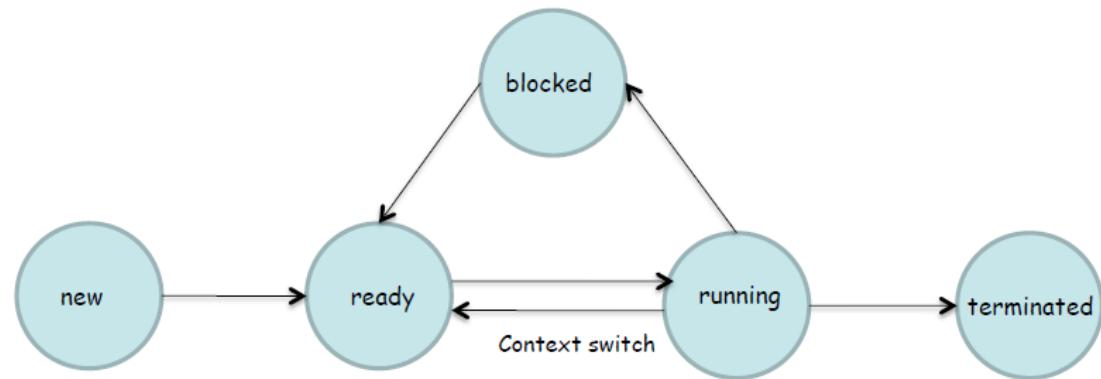
# Procese in sisteme de operare

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



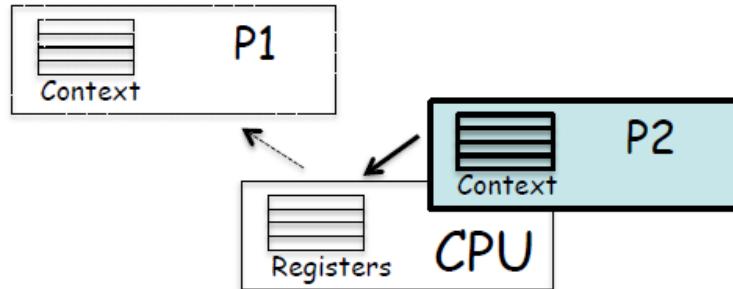
# *Scheduler*

- 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*

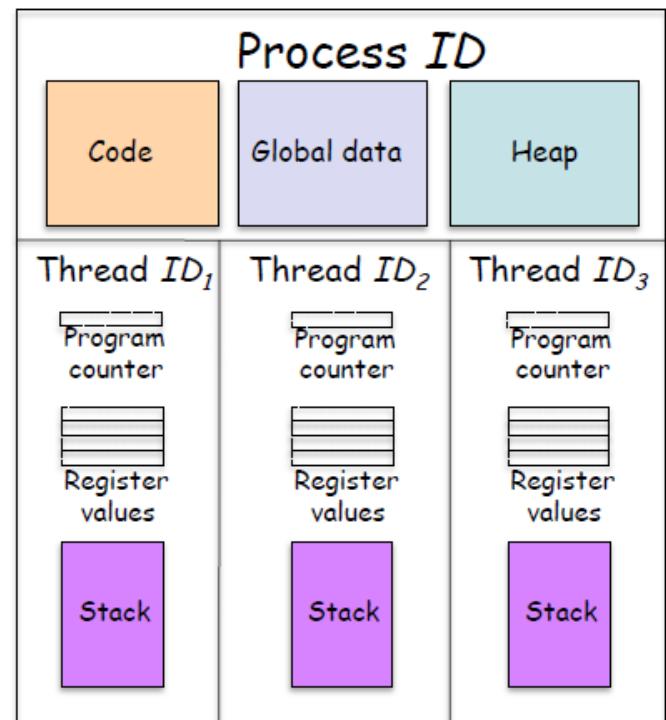
- Atunci cand *scheduler*-ul schimba procesul executat de o unitate de procesare



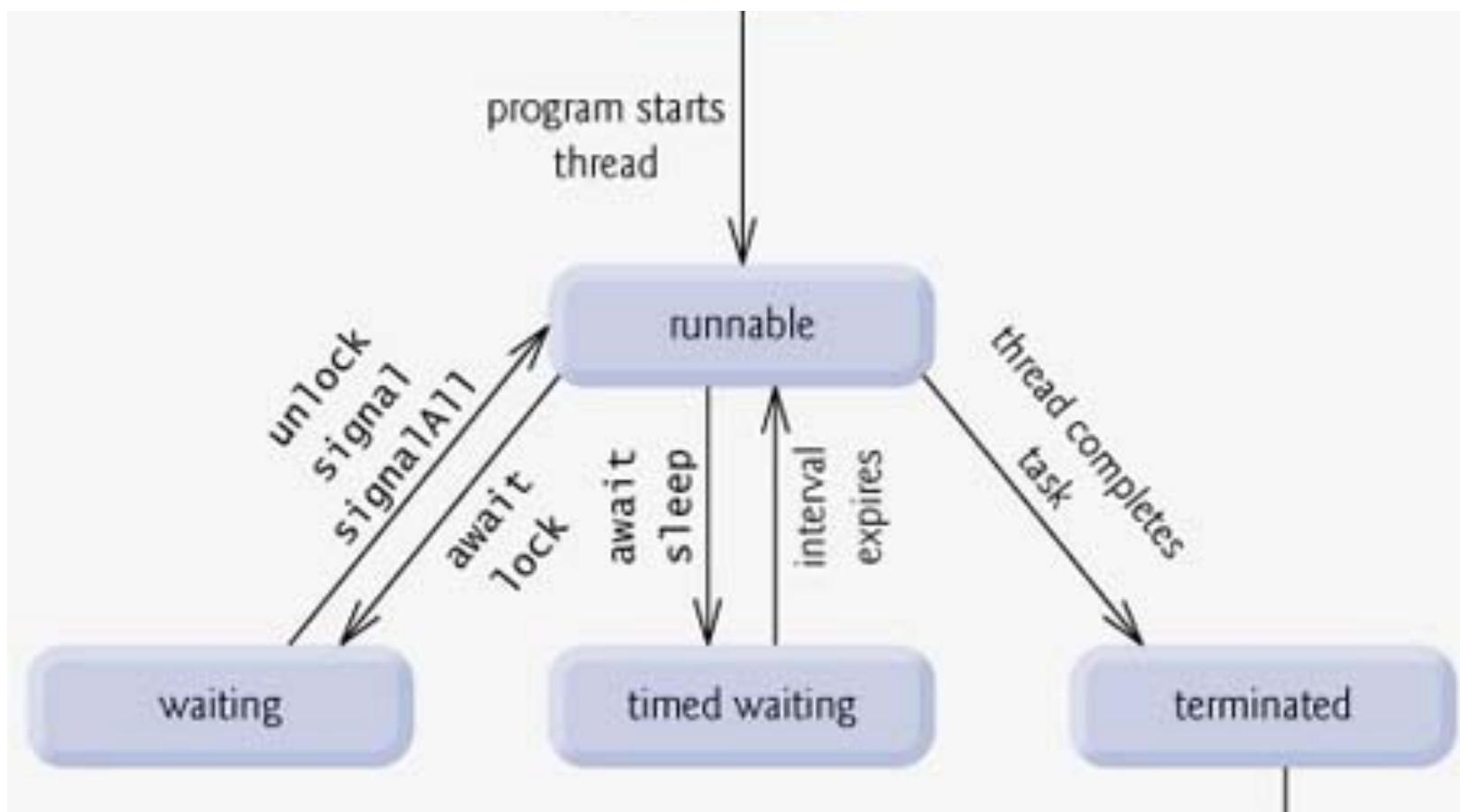
- Actiuni asociate:
  - $P1.state := \text{ready}$
  - Se salveaza valoarea registrilor in memorie dar si context al lui P1
  - Se foloseste contextul lui P2 pt a seta registrii
  - $P2.state := \text{running}$
  - in plus -> switching memory address space:
    - include: memory addresses, page tables, 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 apara intr-un sistem (electronic, software,...) in care **comportamentul este dependent de ordinea in 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 accesizeze o locatie de memorie care este in acelasi timp accesata pentru scriere de catre alt thread.

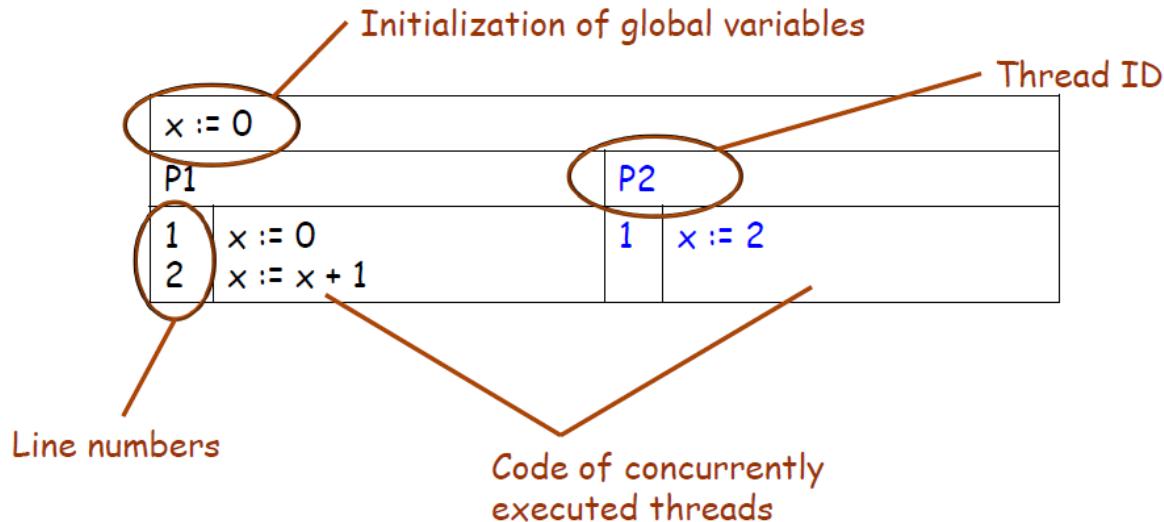
# Deterministic versus non-deterministic computation

- 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

- Secvente 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	$x := 0$		
2	$\text{temp} := x$		
3	$\text{temp} := \text{temp} + 1$		
4	$x := \text{temp}$	1	$x := 2$

- 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--;
```

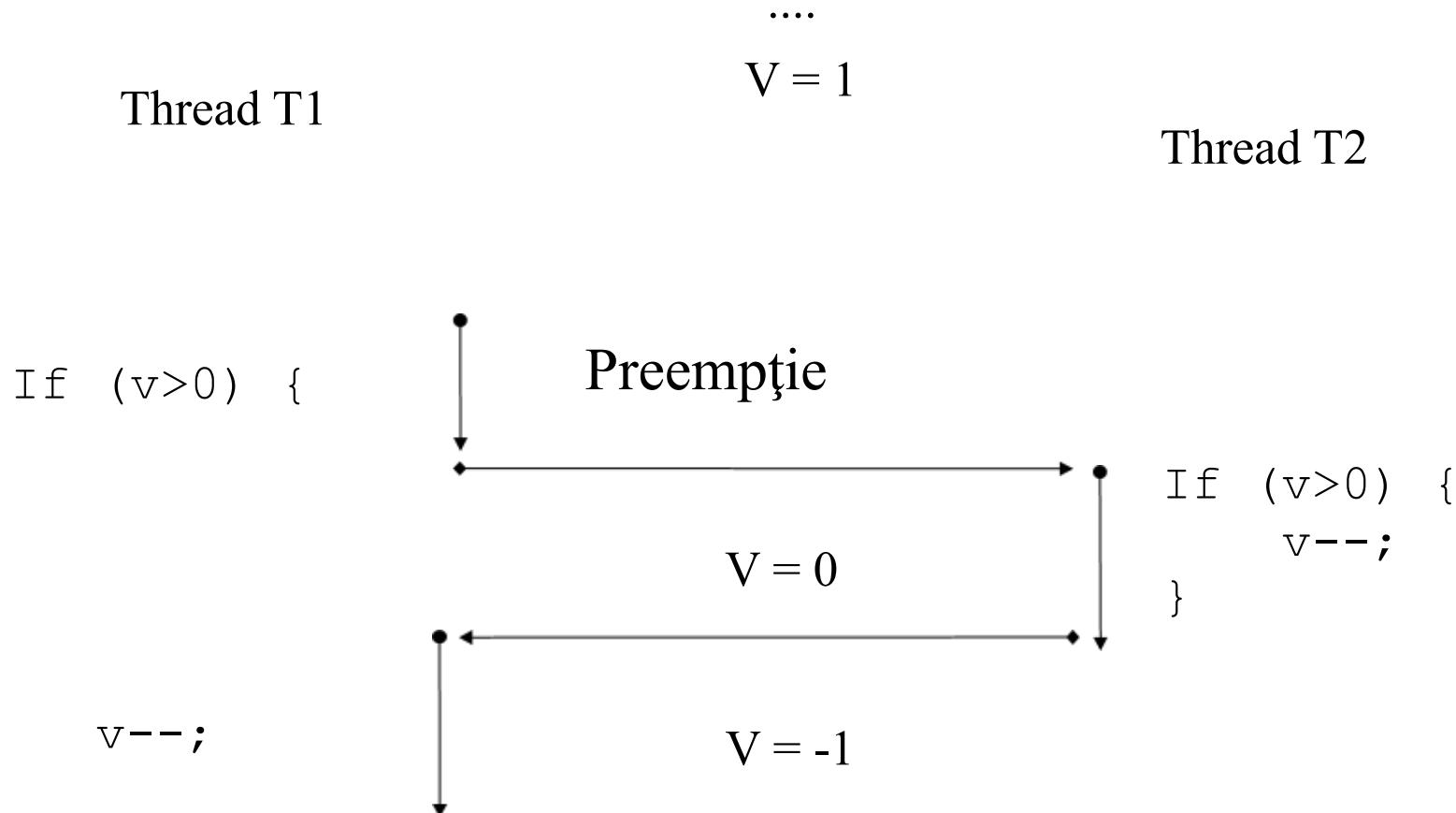
Thread T1

```
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 and has to be executed as an atomic action.

```
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

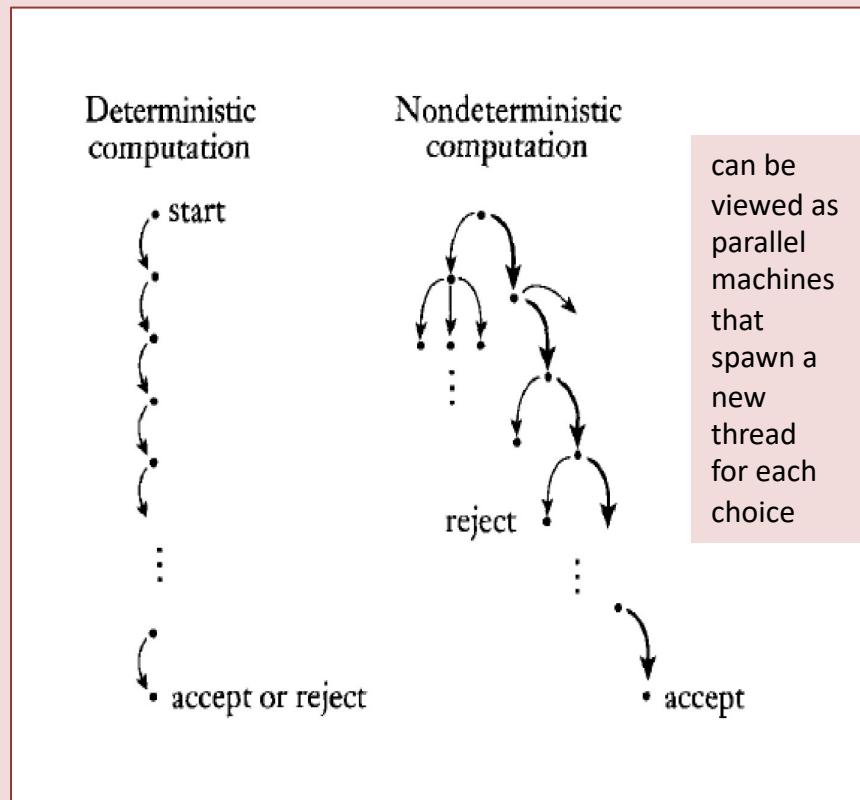
# Solutii

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

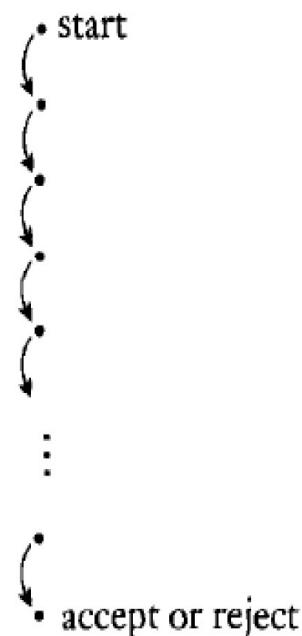
# Deterministic versus non-deterministic computation

- 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.*

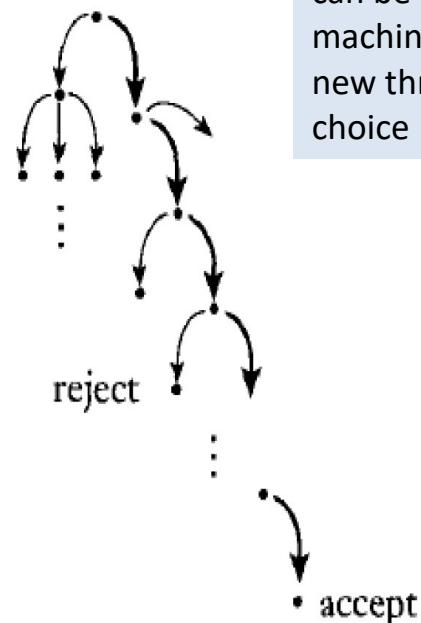


# Deterministic versus non-deterministic computation

Deterministic  
computation



Nondeterministic  
computation

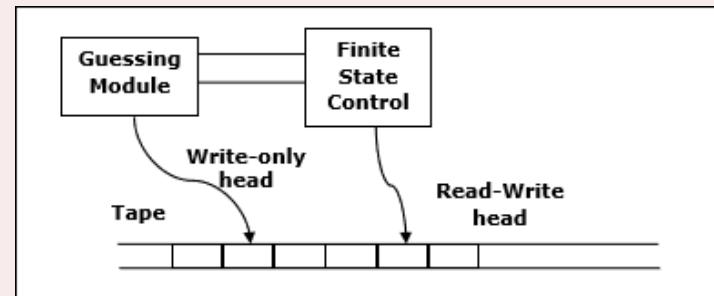
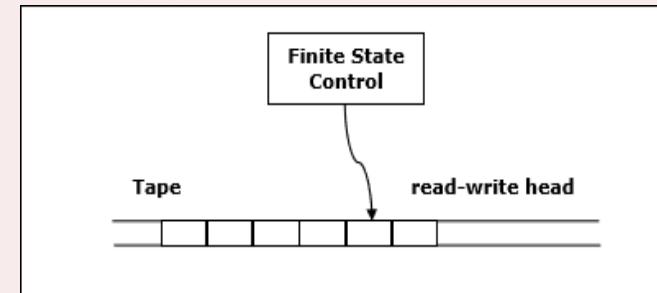


can be viewed as parallel  
machines that spawn a  
new thread for each  
choice

# 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

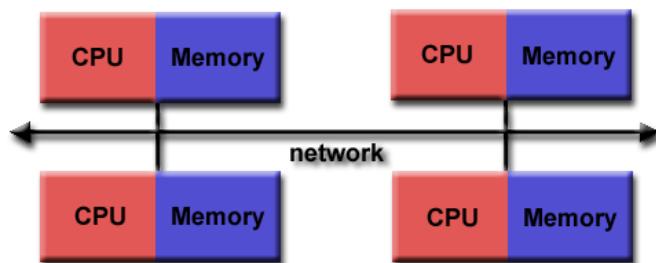
# Implementari

**Exemple:**

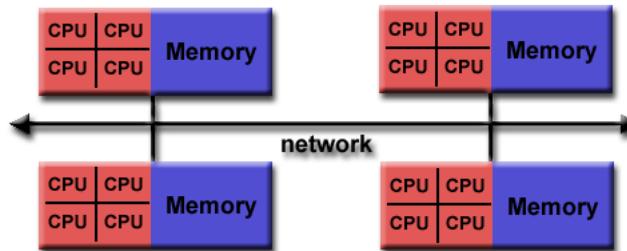
- **MPICH –**
- **Open MPI –**
- **IBM MPI –**
- **IntelMPI (not free)**
- **Links:**  
<http://www.dcs.ed.ac.uk/home/trollius/www.osc.edu/mpi/>

# Modelul de programare

Initial doar pt DM



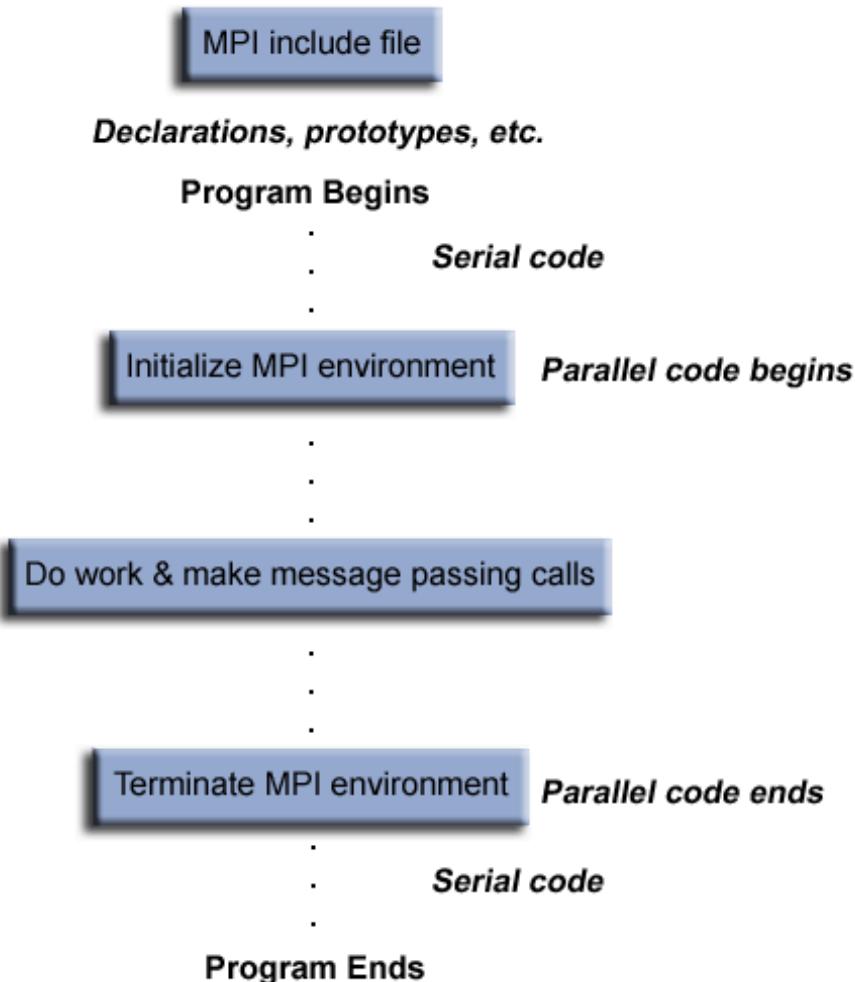
Ulterior si pt SM



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

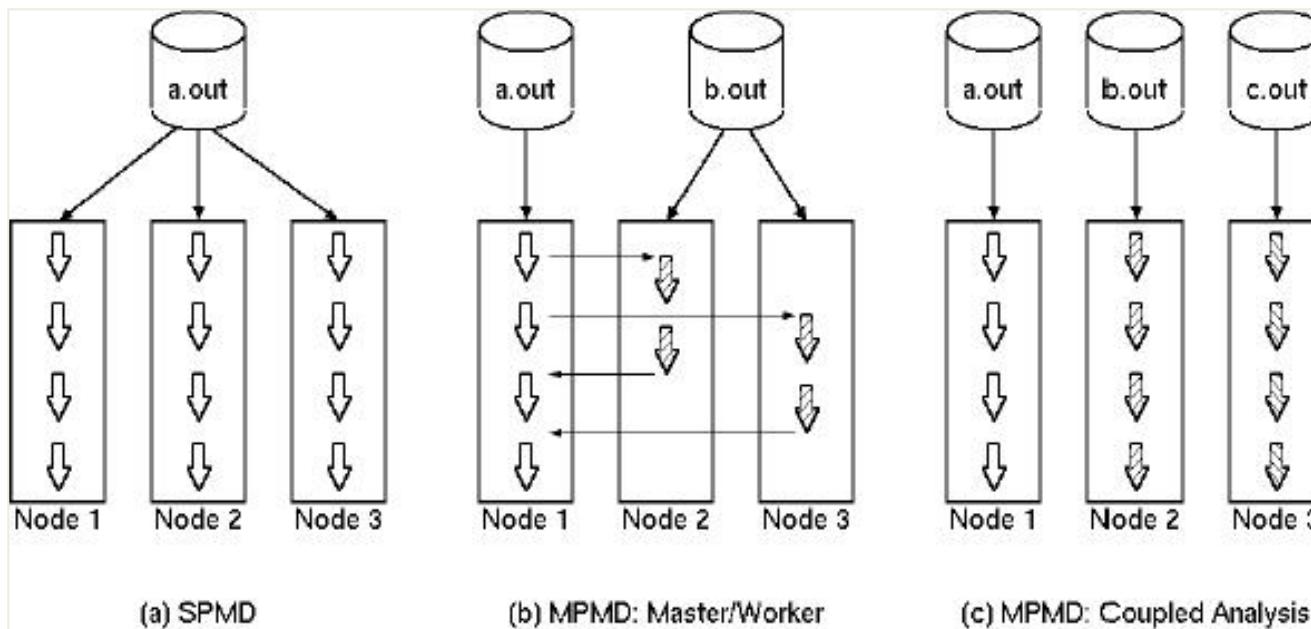
- **Intr-un comunicator, fiecare proces are un identificator unic, rang. El are o valoare intreaga, unica in sistem, atribuita la initializarea mediului.**
- **Utilizat pentru a specifica sursa si destinatia mesajelor.**
- **De asemenea se foloseste pentru a controla executia programului (daca rank=0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

# SPMD/MPMD

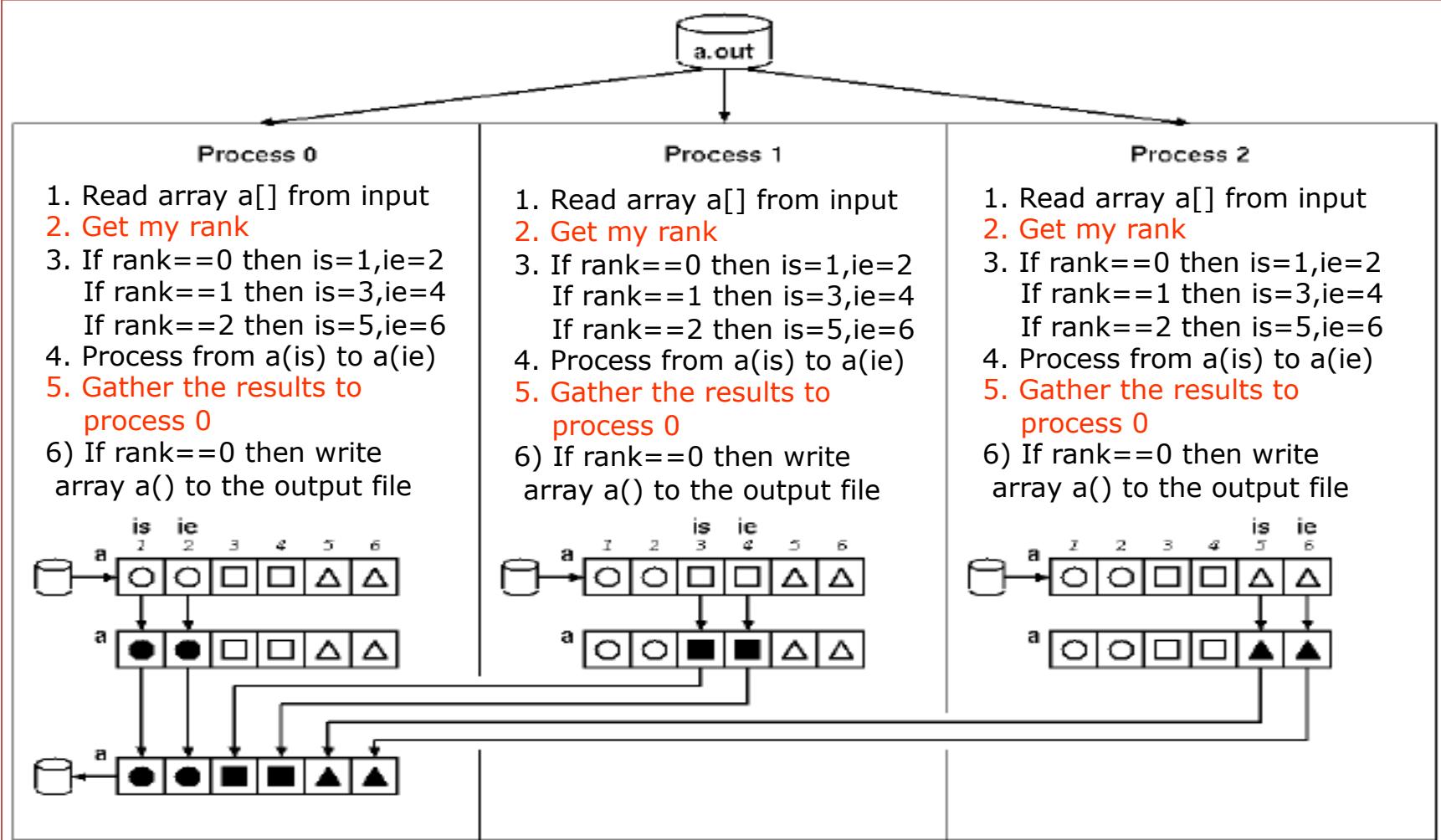
## Modele de calcul paralel in sisteme cu memorie distribuita

SPMD (Single Program Multiple Data) (a)

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



# Modelul SPMD



# 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 mediul de lucru**

```
#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();
}
```

# 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*

### *Operatii blocante*

O operatie de *send blocanta* va returna doar atunci cand zona de date ce a fost trimisa poate fi reutilizata, fara sa afecteze datele primite de destinatar.

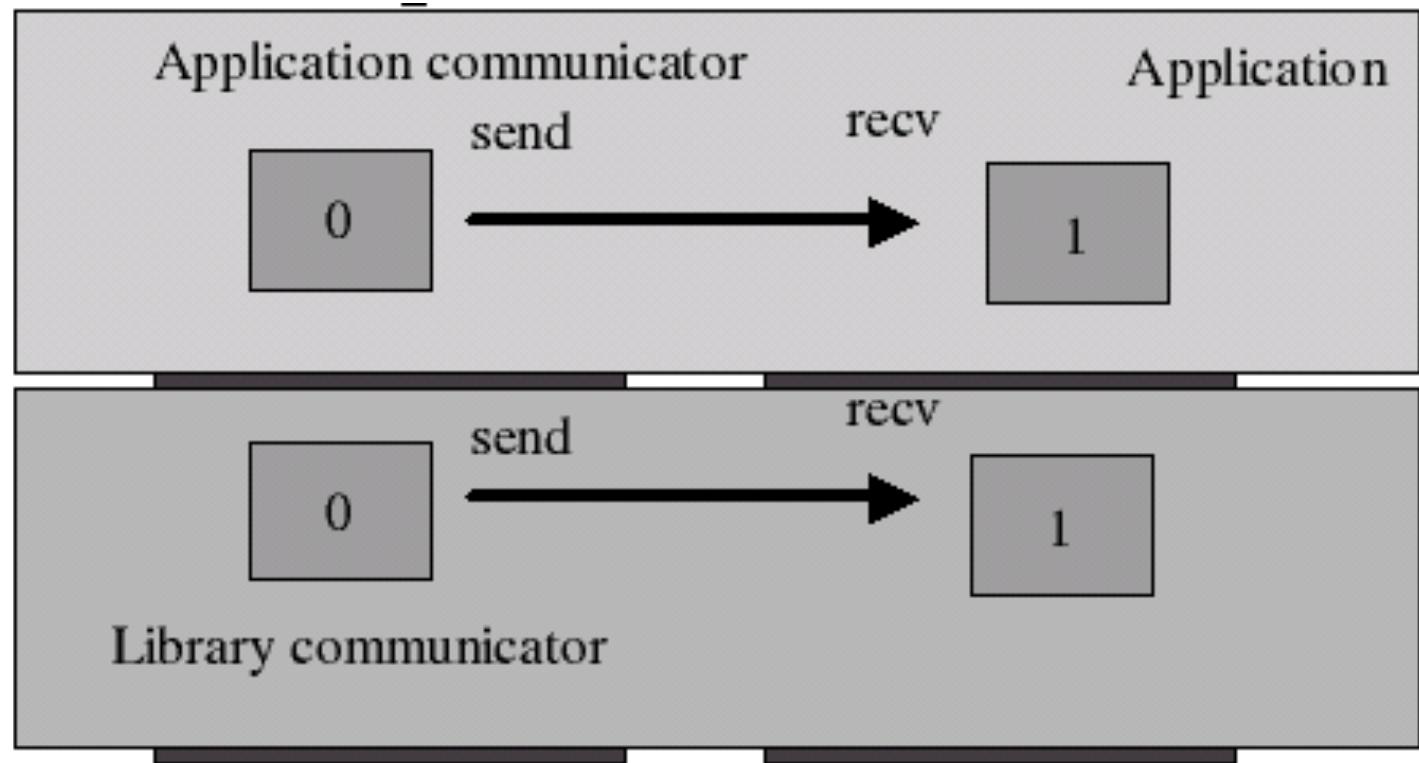
O operatie de *send blocanta* poate fi :

- sincrona : va returna doar atunci cand datele au ajuns efectiv la destinatar
  - asincrona : se utilizeaza o zona de tampon din sistem pentru salvarea datelor ce urmeaza a fi trimise
- O operatie *receive blocanta* va “returna” doar dupa ce datele au fost primite si pot fi folosite in program.

### *Operatii ne-blocante*

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

# 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.
  - MPI garanteaza doar ca mesajele intre doua procese A si B vor ajunge in ordinea trimisa.
  - 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:
  - rangul procesului transmitator
  - rangul procesului receptor
  - un tag (marcaj)
  - un comunicator.
- Comunicatorul stabileste grupul de procese in care se face transmiterea.

## **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 – tag – este un intreg specificat de catre programator, pentru a se putea face distinctie intre mesaje receptionate de la acelasi proces transmitator.
  - 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);
}
}
```

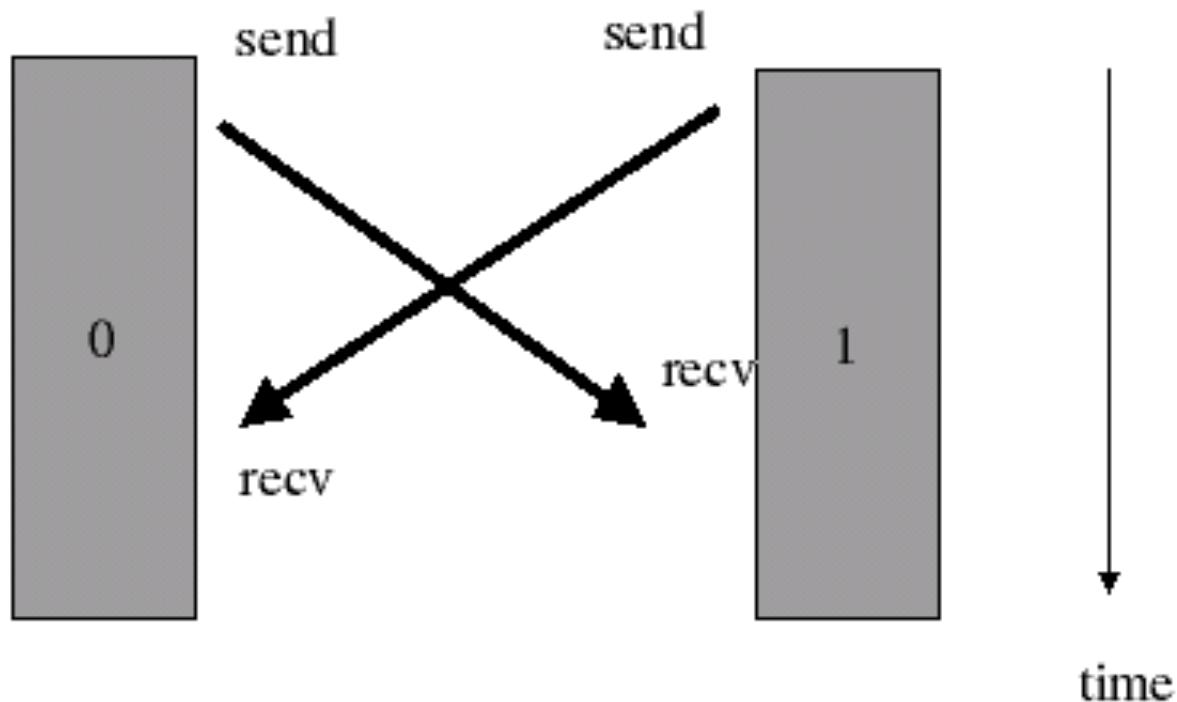
## Exemplu operatii blocante (cont)

```
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

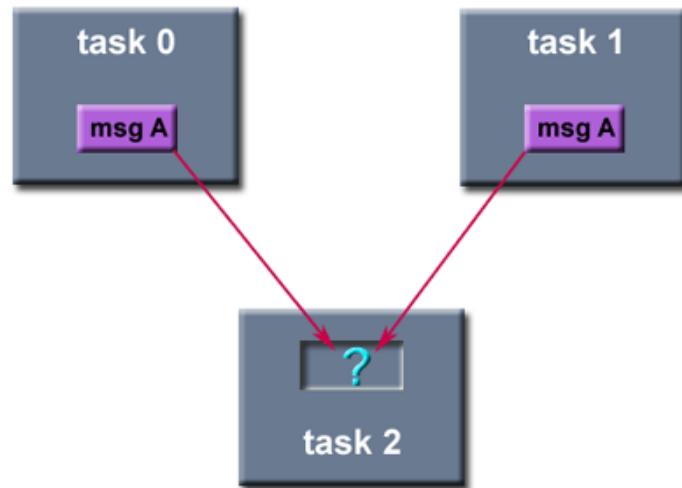
- Scenariu:
  - Presupunem ca avem doua procese in cadrul carora comunicatia se face dupa urmatorul protocol
    - Primul proces trimit date catre cel de-al doilea si asteapta raspunsuri de la acesta.
    - Cel de-al doilea proces trimit date catre primul si apoi asteapta raspunsul de la acesta.
  - 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



# Fairness

- *MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".*
- *Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.*



# Curs 5

## MPI

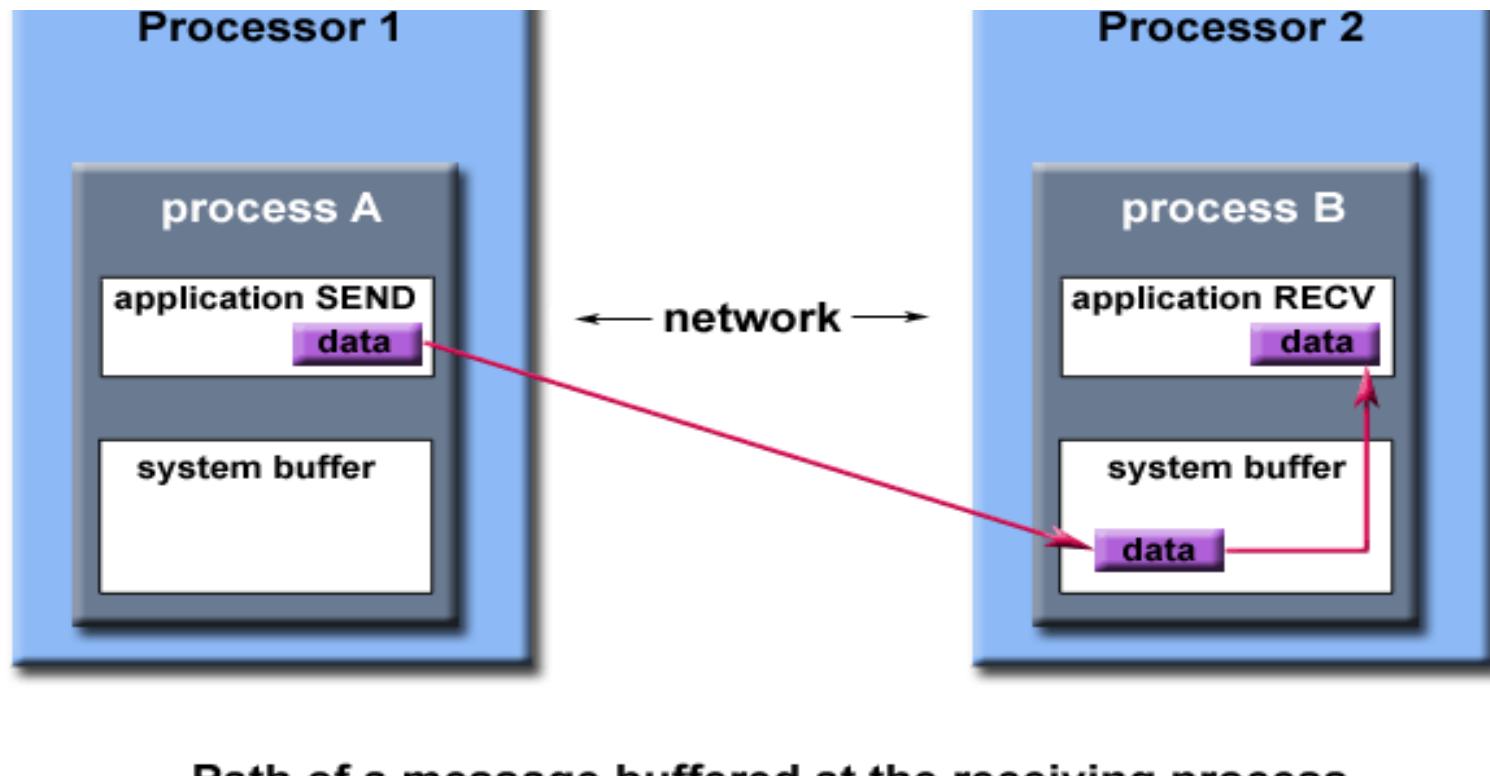
- *Operatii blocante vs ne-blocante*
- operatii de comunicare colectiva

# 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>

# Folosire buffere (decizie a implementarii MPI)



# MPI\_Irecv

- Starts a standard-mode, nonblocking receive.

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
```

- Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request).
- The request can be used later to query the status of the communication or wait for its completion.
- ***A nonblocking receive call indicates that the system may start writing data into the receive buffer.***
- ***The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.***
- A receive request can be determined being completed by calling the [MPI\\_Wait](#), [MPI\\_Waitany](#), [MPI\\_Test](#), or [MPI\\_Testany](#) with request returned by this function.

# MPI\_Isend

- -Starts a standard-mode, nonblocking send.

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,  
MPI_Request *request)
```

- MPI\_Isend starts a standard-mode, nonblocking send.
- Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request).
- The request can be used later to query the status of the communication or wait for its completion.
- ***A nonblocking send call indicates that the system may start copying data out of the send buffer.***
- ***The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.***
- A send request can be determined being completed by calling the [MPI\\_Wait](#), [MPI\\_Waitany](#), [MPI\\_Test](#), or [MPI\\_Testany](#) with request returned by this function.

# MPI\_Wait

- Waits for an MPI send or receive to complete.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- A call to MPI\_Wait returns when the operation identified by request is complete.
- If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI\_Wait and the request handle is set to MPI\_REQUEST\_NULL.
- The call returns, in status, information on the completed operation.
- If your application does not need to examine the *status* field, you can save resources by using the predefined constant MPI\_STATUS\_IGNORE as a special value for the *status* argument.

# MPI\_Test

- Tests for the completion of a specific send or receive.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

- A call to `MPI_Test` returns `flag = true` if the operation identified by `request` is complete.
  - In this case, the `status` object is set to contain information on the completed operation;
  - if the communication object was created by a nonblocking send or receive, then it is deallocated and the `request` handle is set to `MPI_REQUEST_NULL`.
- The call returns `flag = false`, otherwise.
  - In this case, the value of the `status` object is undefined.
- `MPI_Test` is a local operation.
- If your application does not need to examine the `status` field, you can save resources by using the predefined constant `MPI_STATUS_IGNORE` as a special value for the `status` argument.
- The functions [`MPI\_Wait`](#) and `MPI_Test` can be used to complete both sends and receives.

# Exemple

- Hello world – async messages
  - p procese
  - procesele cu id > 0 trimit mesaje catre procesul 0
  - procesul 0 le preia in ordinea venirii – le adauga intr-un string
  - dupa ce a preluat toate mesajele afiseaza stringul in care aceastea au fost concatenate

## **MPI\_Sendrecv**

- trimite si receptioneaza un mesaj

**MPI\_Sendrecv (&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf, recvcount, recvtype, source, recvtag, comm, &status)**

- *Send a message and post a receive before blocking!!!*
- *Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.*
- *useful for executing a shift operation across a chain of processes*

# A quick overview of other MPI's send modes

MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete). (*In the following, we use "send buffer" for the user-provided buffer to send.*)

- **MPI\_Send**
  - **MPI\_Send** will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- **MPI\_Bsend**
  - May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- **MPI\_Ssend**
  - will not return until matching receive posted
- **MPI\_Rsend**
  - May be used ONLY if matching receive already posted. User responsible for writing a correct program.
- **MPI\_Isend**
  - Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see `MPI_Request_free`). Note also that while the `I` refers to immediate, there is no performance requirement on `MPI_Isend`. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.
- **MPI\_Ibsend**
  - buffered nonblocking
- **MPI\_Issend**
- **Synchronous nonblocking.** Note that a Wait/Test will complete only when the matching receive is posted.
- **MPI\_Irsend**
  - As with `MPI_Rsend`, but nonblocking.

Note that "nonblocking" refers ONLY to whether the data buffer is available for reuse after the call. No part of the MPI specification, for example, mandates concurrent operation of data transfers and computation.

Some people have expressed concern about not having a single "perfect" send routine. But note that in general you can't write code in Fortran that will run at optimum speed on both Vector and RICS/Cache machines without picking different code for the different architectures. MPI at least lets you express the different algorithms, just like C or Fortran.

## Recommendations

The best performance is likely if you can write your program so that you could use just `MPI_Ssend`; in that case, an MPI implementation can completely avoid buffering data. Use `MPI_Send` instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data. (Unfortunately, one vendor has chosen to have `MPI_Send` emphasize buffering over performance; on that system, `MPI_Ssend` may perform better.) If nonblocking routines are necessary, then try to use `MPI_Isend` or `MPI_Irecv`. Use `MPI_Bsend` only when it is too inconvenient to use `MPI_Isend`. The remaining routines, `MPI_Rsend`, `MPI_Issend`, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

# **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 ceilalți 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)

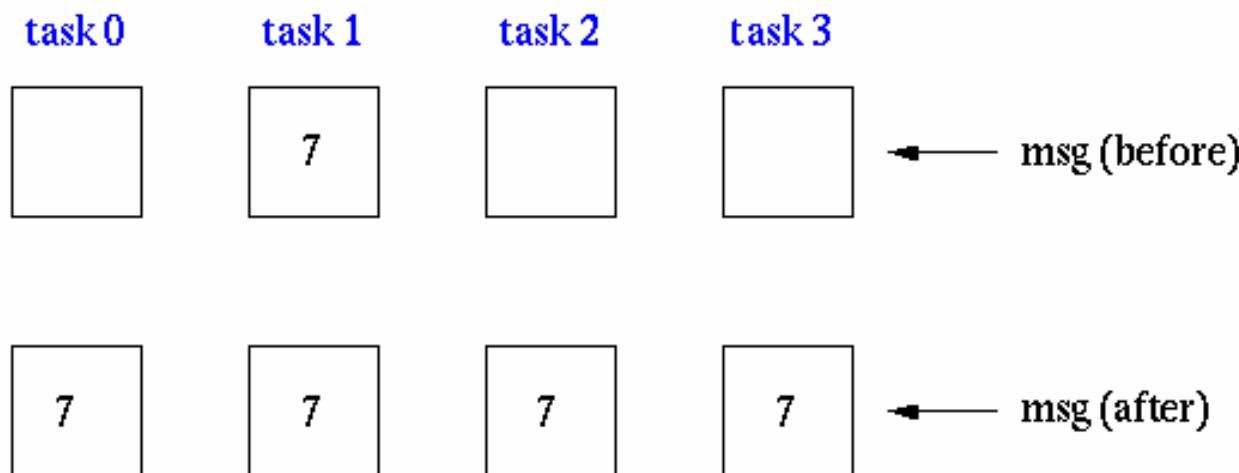
MPI\_BARRIER (comm,ierr)

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

## 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);
```

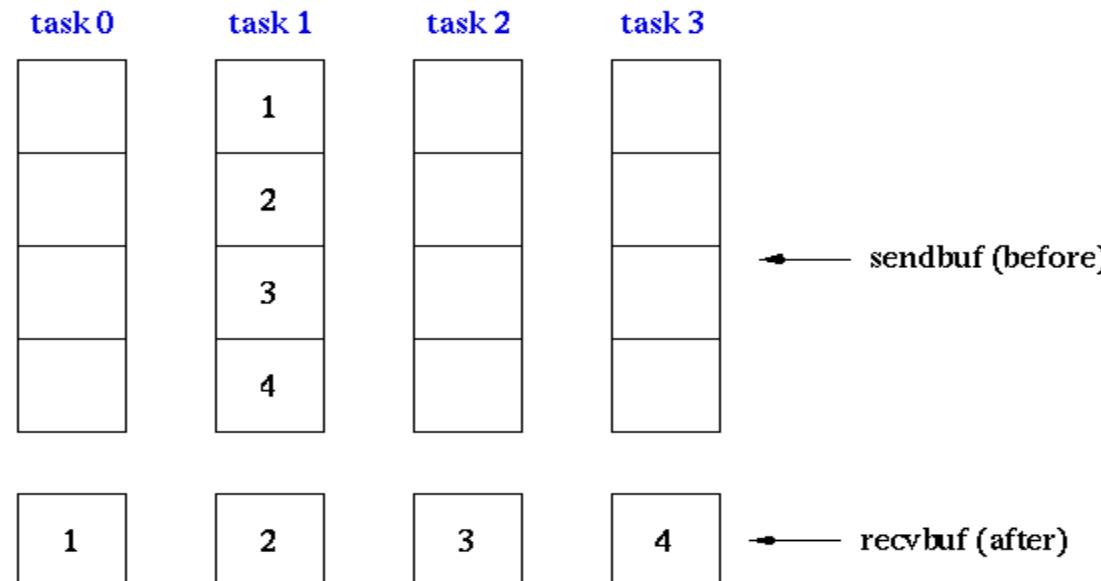


# Operatii colective

## 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);
```

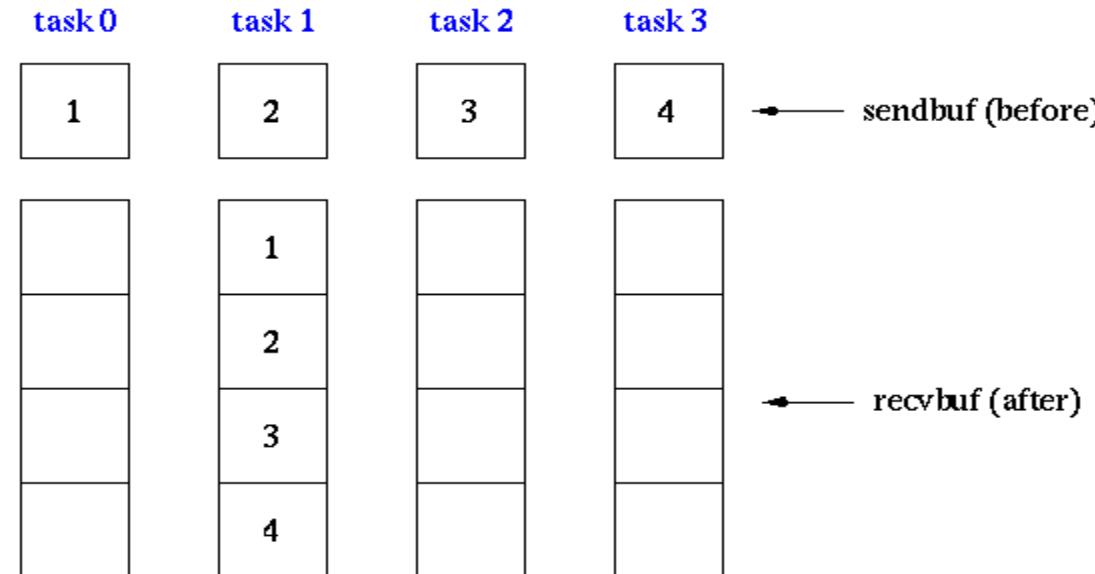


# Operatii colective

## 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);
```

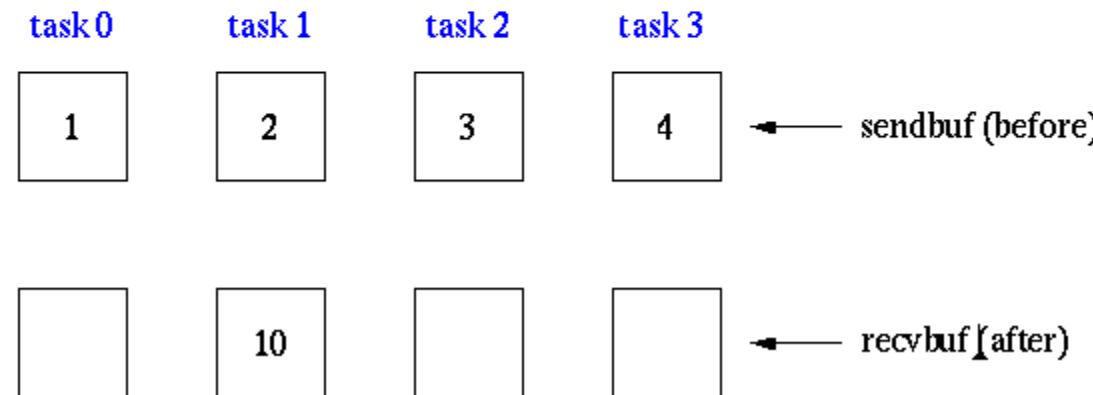


# Operatii colective

## 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 4

## 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
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

# *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
  - 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();  
  
    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 apare deadlock?

In ce situatii?

# *Safety - 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*).
  - Strong fairness
    - daca o actiune este infinit de des accesibilila (*unfinetely often enabled*) dar nu obligatoriu in mod continuu atunci trebuie sa fie executata infinit de des (*infinetely often*).
- ***Liveness***
  - "something good eventually happens"
  - *a program eventually terminates*

# Forme de sincronizare

- *excluderea mutuală*: se evită utilizarea simultană de către mai multe procese a unui resurse critice.
  - O resursă este critică dacă poate fi utilizată doar de către singur proces la un moment dat
  - *arbitrarea*: se evită accesul simultan din partea mai multor procesoare la aceeași locație de memorie.
    - se realizează o secvențializare a accesului, impunând așteptarea până când procesul care a obținut acces și-a încheiat activitatea asupra locației de memorie.
- *sincronizarea pe condiție*: se amână execuția unui proces până când o anumită condiție devine adevărată;

# 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:

# 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 instrucț.:

P(s)

regiune critică

V(s)

Rest. procesului

- Cerinte de atomicitate:
  - Testarea
  - Incrementare/decrementarea valorii
- 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  $P(s)/down$  si  $V(s)/up$

# 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:

$count \geq 0$

$count = k + \#up - \#down$

- $k \geq 0$ : 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 protejeaza modificarile var count

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

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

```
count := k
```

```
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

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 clasă) 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 initializare 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** 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/setează **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 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: strong semaphore

entry : SEMAPHORE

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

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

$\text{entry.down}$   
 $\text{body}_r$   
 $\text{entry.up}$

end

# Variabile conditionale in monitoare

- 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 ofera 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
- **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).

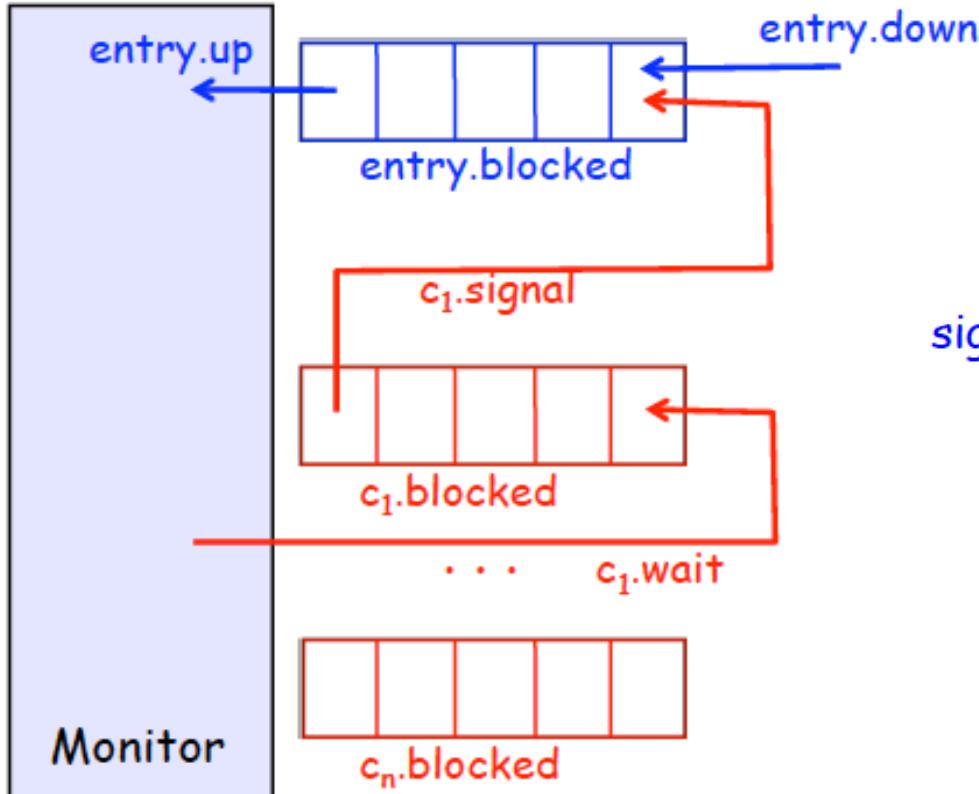
# Schita Implementare

```
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:
  - Procesul de semnalizare continua si procesul notificat este mutat la intrarea monitorului;
  - Procesul care semnalizeaza lasa monitorul si procesul semnalizat continua.

# Signal & Continue

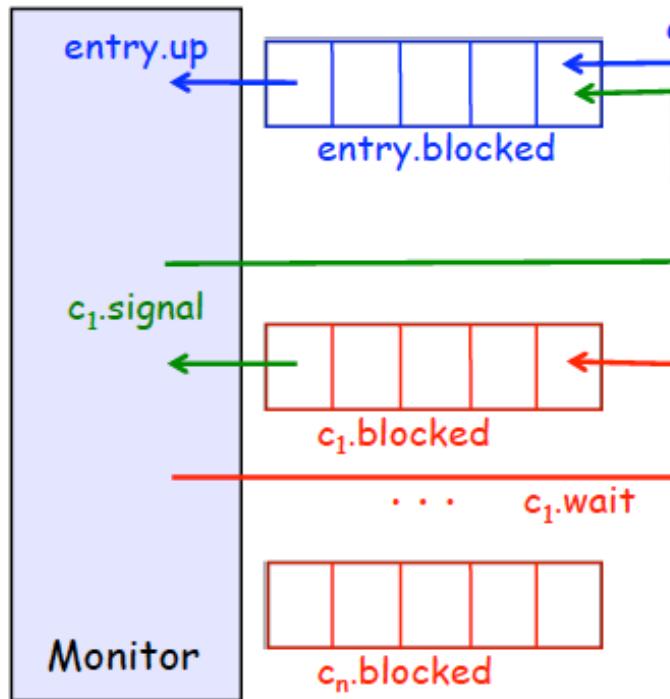


Pentru fiecare  
conditie => o  
coada

```
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
- 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 în 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*:

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)

- O abstractizare care permite sincronizarea conditionala;  
Operatii: **wait**; **signal** ; [broadcast]
- O variabila conditionala **C** este asociata cu
  - o variabila de tip **Lock – m**
  - o coada
- Thread t apel **wait** =>
  - suspenda t si il adauga in coada lui **C** + deblocheaza **m** (op atomica)
- Atunci cand t isi reia executia **m** se blocheaza
- Thread v apel **signal** =>
  - se verifica daca este vreun thread care asteapta si il activeaza

Legatura cu monitor:

- Variabile conditionale pot fi asociate cu lacatul unui monitor (monitor lock);
  - Permit threadurilor sa astepte in interiorul unei sectiuni critice eliberand 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();  
}  
(1)  
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 );

wait\_for      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);

wait\_until      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 7

## 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
    }
}
```

# 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;  
    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

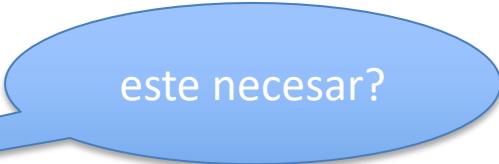
```
public class Counter {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  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

# Exemplu

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```



este necesar?

# Transformare => 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

```
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.

# *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

- 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) –
    - ceva similar ~~ sleep.
    - poate sa produca blocaj!

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

```
...
```

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

# Exemplificari

- `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;

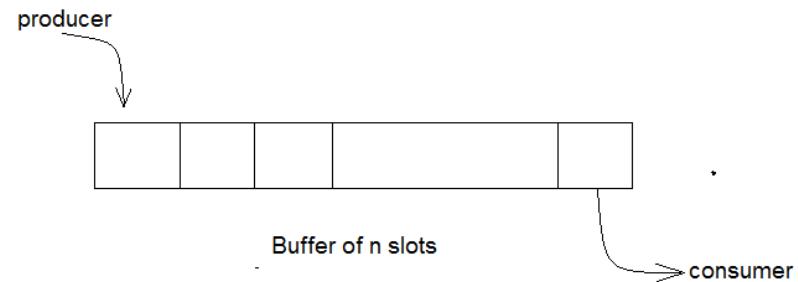
/* Method used by the consumer to access the shared data */
    public synchronized int get() {
        while (available == false) {
            try {
                wait();          // Consumer enters a wait state until notified by the Producer
            } catch (InterruptedException e) { }
        }
        available = false;
        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 {
                wait();          // Producer who wants to store contents enters
                                  // a wait state until notified by the Consumer
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        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(long time, TimeUnit unit)</a> 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);
```

# ReadWriteLock

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

# Curs 8

## Programare Paralela si Distribuita

Java:

Lock, Condition, Semaphore, ReadWriteLock, Exchange  
Intreruperi

Exemple: deadlock, producator-consumator

# 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(long time, TimeUnit unit)</a> 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();  
    }  
}
```

# Intreruperi

- O intrerupere (*interrupt*) este o indicatie pentru un thread ca ar trebui sa se opreasca si ... sa faca altceva (de ex. sa se termine).

`public void interrupt()`

`public static boolean interrupted()`

`public boolean isInterrupted()`

*“There is no way in Java to terminate a thread unless the thread exits by itself.”*

# Intreruperi

- mecanismul de intrerupere foloseste un flag intern -> the *interrupt status*.
- Atunci cand se apeleaza `Thread.interrupt` se seteaza acest flag.
- Atunci cand se verifica intreruperea prin metoda statica `Thread.interrupted`, *<interrupt status>* este sters.
- Metoda nestatica `isInterrupted`, care este folosita de catre un thread pt a verifica statusul (*interrupt status*) al altuia nu schimba flagul.
- Prin conventie, orice metoda care se termina (exit) aruncand o exceptie de tip `InterruptedException` sterge "interrupt status".
- Totusi este posibil ca acesta sa fie imediat setat din nou de catre alt thread care invoca o metoda *interrupt*.

# Exemplu

```
public class SimpleThreads {  
  
    static void threadMessage(String message) {  
        String threadName = Thread.currentThread().getName();  
        System.out.format("%s: %s %n", threadName, message);  
    }  
  
    private static class MessageLoop implements Runnable {  
        public void run() {  
            String importantInfo[] = {  
                "Studentii sunt prezenti.",  
                "Examenul este greu.",  
                "Vacanta este asteptata.",  
                "Există concurență."  
            };  
            try {  
                for (int i = 0; i < importantInfo.length; i++) {  
                    // Pause for 4 seconds  
                    Thread.sleep(4000);  
                    // Print a message  
                    threadMessage(importantInfo[i]);  
                }  
            } catch (InterruptedException e) {  
                threadMessage("I wasn't done!");  
            }  
        }  
    }  
}
```

```

public static void main(String args[])
throws InterruptedException {

    // Delay, in milliseconds before
    // we interrupt MessageLoop
    // thread (default one hour).

    long patience = 1000 * 60 * 60;

    threadMessage(
        "Starting MessageLoop thread");

    long startTime = System.currentTimeMillis();

    Thread t = new Thread(new MessageLoop());
    t.start();

    threadMessage(
        "Waiting for MessageLoop thread to finish");
}

while (t.isAlive()) {
    threadMessage("Still waiting...");

    t.join(1000);

    if ((System.currentTimeMillis() - startTime) > patience)
        && t.isAlive())
    {
        threadMessage("Tired of waiting!");
        t.interrupt();
        // Shouldn't be long now
        // -- wait indefinitely

        t.join();
    }
}
threadMessage("Finally!");
}

```

# Interactiuni: Waits, Notification, Interruption

- Notificările nu se pot pierde din cauza intreruperilor.
- Presupunem ca un set de threaduri **s** este în **wait set** a lui **m**, și alt thread executa notificare pe **m**.

Atunci fie:

- cel puțin un thread din **s** ieșe normal din **wait**, sau
- **toate** threadurile din **s** ies din **wait** aruncând excepție **InterruptedException**
- Dacă un thread este atât intrerupt cât și trezit prin notificare și el ieșe din **wait** aruncând o excepție atunci un alt thread din **wait set** va fi notificat.
- Dacă thread **t** a fost sters din **wait set** a lui **m** din cauza unei intreruperi atunci **interrupted status** al lui **t** este setat la **false** și se ieșe din **wait** cu aruncarea unei excepții de tip **InterruptedException**.

# Conditionare - reguli

- folosirea operatiilor `wait` doar in cicluri care se termina atunci cand anumite conditii logice sunt indeplinite!
- fiecare thread trebuie sa determine o ordine intre evenimentele care pot cauza ca el sa fie sters din *wait set*.

De exemplu: daca  $t$  este in *wait set* al obiectului  $o$ , atunci cand apare atat o intrerupere a lui  $t$  cat si o notificare a lui  $o$ , trebuie sa existe o ordine intre aceste evenimente.

- ÷ Daca *interrupt* este considerata prima, atunci pana la urma tiese din `wait` aruncand `InterruptedException`, si alt thread din *wait set* a lui  $o$  (daca mai exista vreunul) va primi notificarea.
- ÷ Daca sunt invers ordonate atunci tiese normal din `wait` si intreruperea este in asteptare (pana se verifica starea).

## Concluzii – legate de interacțiune intreruperi cu mecanismul wait-notify

Un thread  $t$  poate fi sters din *wait set* al unui obiect  $o$  ca urmare a uneia din urm. actiuni (si apoi isi va relua executia) :

- ◆ O actiune *notify* asupra lui  $o$  in care  $t$  este selectat spre a fi sters din *wait set*.
- ◆ O actiune *notifyAll* asupra lui  $o$  .
- ◆ O actiune de intrerupere realizata de  $t$ .
- ◆ Trecerea timpului (argument) specificat la apelul lui *wait*.
- ◆ Operatii "spurious wake-ups" (*no apparent reason*) –/rare.

([https://en.wikipedia.org/wiki/Spurious\\_wakeup](https://en.wikipedia.org/wiki/Spurious_wakeup))

## *Nested monitor lockout*

- Thread 1 blocheaza A
  - Thread 1 blocheaza B (in timp se ramane blocajul pe A)
    - Thread 1 decide sa astepte un semnal de la un alt thread
    - Thread 1 apeleaza **B.wait()** => elibereaza B dar nu si pe A.
- Thread 2 trebuie sa blocheze atat pe A cat si pe B (in ordine) pentru a trimite un semnal catre Thread 1.
- Thread 2 nu poate bloca pe A (pt ca Thread 1 are blocajul lui A).
- Thread 2 ramane blocat indefinit asteptand ca A sa fie eliberat.
- Thread 1 ramane blocat indefinit asteptand un semnal de la Thread 2, si astfel nu elibereaza pe A, etc.

# Diferenta intre Deadlock si *Nested monitor lockout*

- In cazul *deadlock*, 2 (sau mai multe) threaduri se asteapta unul pe altul sa elibereze blocajul.
- In cazul *nested monitor lockout*:
  - Thread 1 blocheaza A, si asteapta un semnal de la Thread 2.
  - Thread 2 asteapta sa-l blocheze pe A pentru a trimite semnalul catre Thread 1.

# 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);
```

# ReadWriteLock

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

## public interface **ReadWriteLock**

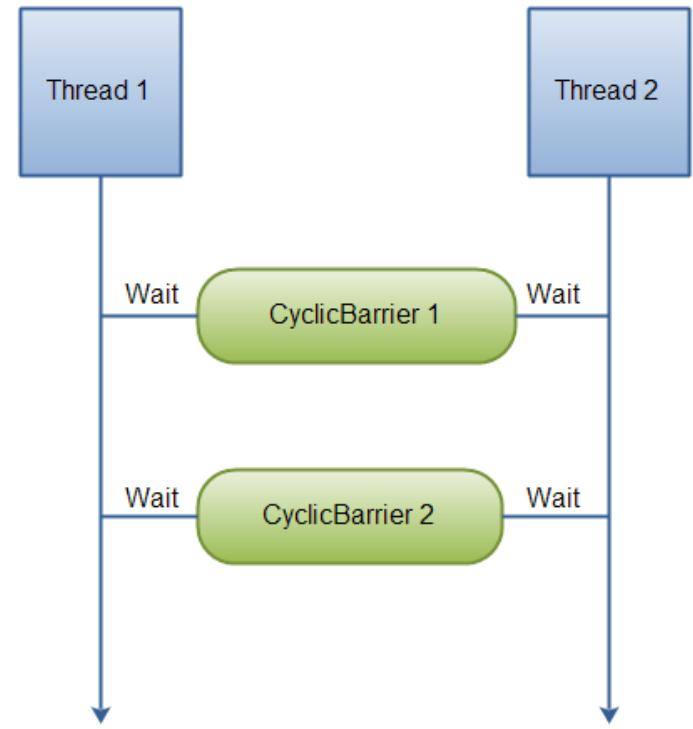
- A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing.
- The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.
- The write lock is exclusive.

# 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)`

# java.util.concurrent.CountDownLatch

## (java documentation)

- A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
- A **CountDownLatch** is initialized with a given count. The **await** methods block until the current **count** reaches zero due to invocations of the **countDown()** method, after which all waiting threads are released and any subsequent invocations of await return immediately.
  - This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier.
- A CountDownLatch is a versatile synchronization tool and can be used for a number of purposes.
  - A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate: all threads invoking **await** wait at the gate until it is opened by a thread invoking **countDown()**.
  - A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.
- A useful property of a CountDownLatch is that it doesn't require that threads calling **countDown** wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass.

## CountDownLatch (Java)

```
final CountDownLatch latch = new CountDownLatch(5);
```

```
// making two threads for 2 services
```

```
Thread serviceOneThread = new Thread(new ServiceOne(latch, 2));
```

```
Thread serviceTwoThread = new Thread(new ServiceTwo(latch, 3));
```

```
serviceOneThread.start();
```

```
serviceTwoThread.start();
```

```
// latch waits till the count becomes 0
```

```
// this way it can make sure that the execution of main thread only
```

```
// finishes once 2 services have executed
```

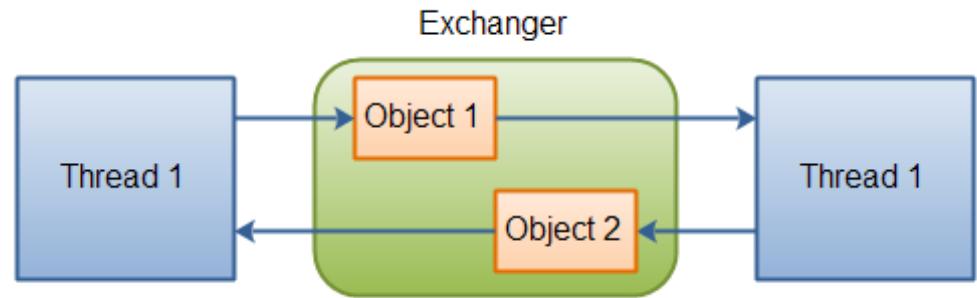
```
try {
```

```
    latch.await(); ...
```

# 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 i le 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 A for B  
Thread-1 exchanged B for A

# 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;  
            this.object = this.exchanger.exchange (this.object);  
  
            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.

# Exemplu

```
public class SynchronousQueueDemo { public static void main(String args[]) {  
    final SynchronousQueue<String> queue = new SynchronousQueue<String>();  
    Thread producer = new Thread("PRODUCER") {  
        public void run() {  
            String event = "FOUR";  
            try { queue.put(event); // thread will block here  
                System.out.printf("[%s] published event : %s %n",  
                    Thread.currentThread().getName(), event); }  
            catch (InterruptedException e) {  
                e.printStackTrace(); } } };  
    producer.start(); // starting publisher thread  
  
    Thread consumer = new Thread("CONSUMER") {  
        public void run() {  
            try { String event = queue.take(); // thread will block here  
                System.out.printf("[%s] consumed event : %s %n",  
                    Thread.currentThread().getName(), event); }  
            catch (InterruptedException e) { e.printStackTrace(); }  
        }  
    };  
    consumer.start(); // starting consumer thread  
}
```

## Sincronizare <-> comunicare

Programele comunică între ele nu numai pentru a-și comunica informații sub formă de mesaje ci și pentru a se sincroniza.

=>

Un semnal de sincronizare poate fi considerat și el că este un mesaj fără conținut ce se transmite între programe.

Cum se realizează acest lucru?

Rendez- vous

Process-view

## Procese secvențiale comunicante. Rendez-vous simetric.

task-ul A: o comandă de emitere mesaj → SUSPENDARE ←

task-ul B: o comandă de recepție de mesaj;

task-ul B: o comandă de recepție de mesaj → SUSPENDARE ←

task-ul A: o comandă de emisie de mesaj;

task-uri sincronizate → datele (mesajul) sunt transferate

# Procese distribuite. Rendez-vous asimetric

- Comunicarea și sincronizarea între programele concurente se realizează în acest caz similar cu apelarea prin nume de către programul emițător a unei proceduri incluse în programul receptor,
  - lista cu parametrii asociați acestui apel fiind folosită ca un “canal” de comunicare pentru transmiterea de date între cele două programe.
- Doar programul apelant trebuie să cunoască numele programului apelat, nu și invers.

# Curs 9

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 progr. 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
```

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)
```

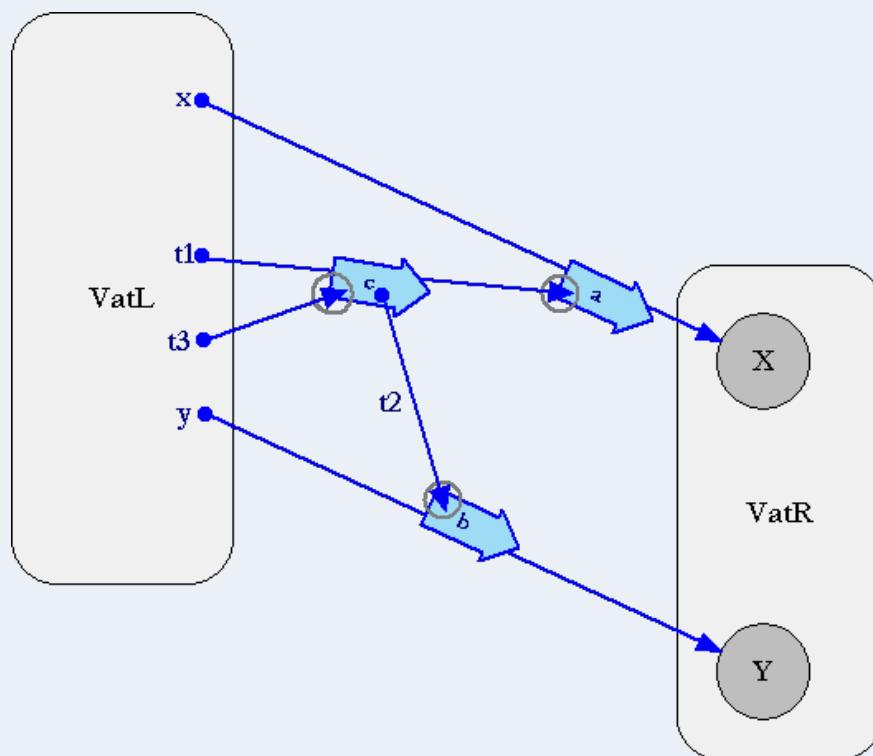
Daca x, y, t1, si t2 sunt localizate pe aceeasi masina remote atunci se poate rezolva in 1 round-trip.

- 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)
  - 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 prin transmiterea unui mesaj (se asteapta pana la primirea mesajului)
- 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 a packaged\_task*
- (2) *future from an async()*
- (3) *future from a promise*

# packaged\_task

- `std::packaged_task` object = wraps a 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

- 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)

# async

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

# 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();
```

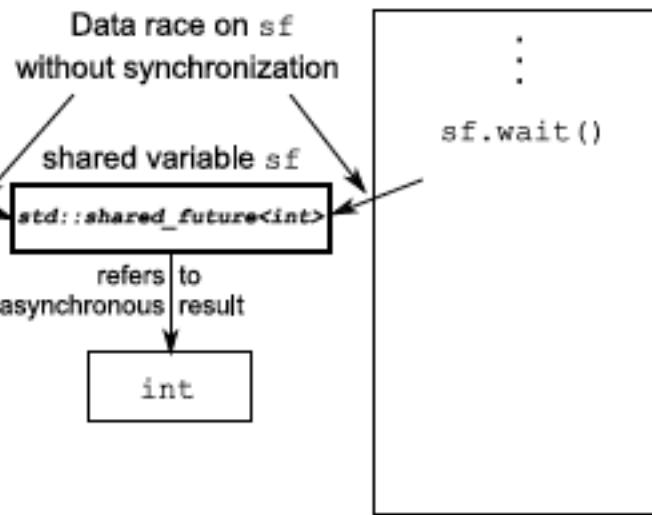
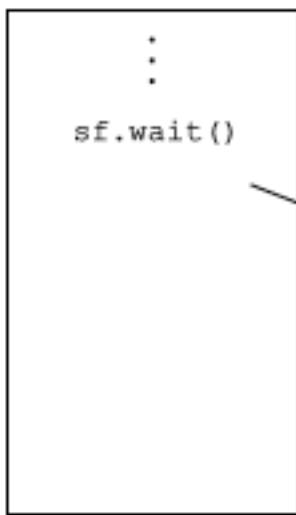
```
f3.wait();
```

# 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

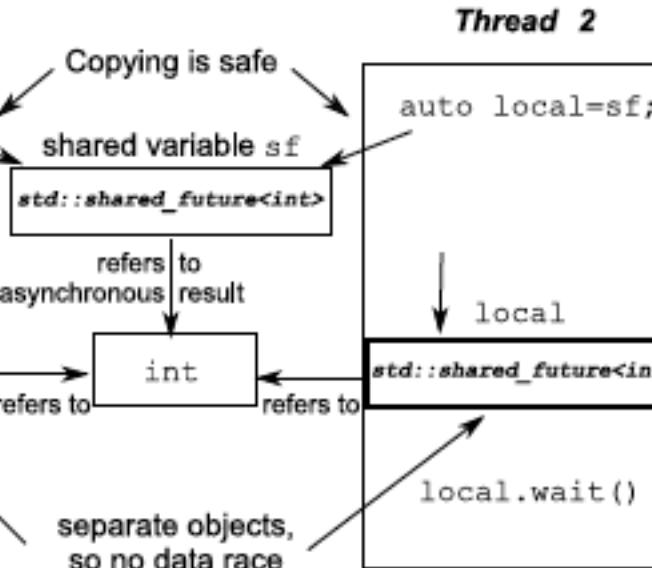
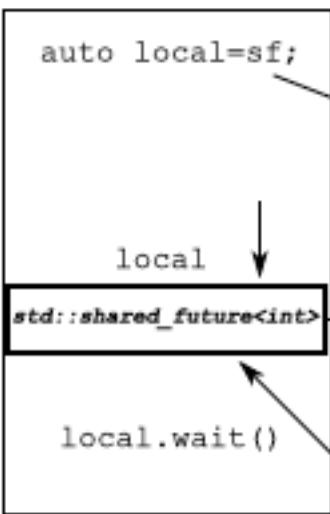


### Thread 2

`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();  
    };  
}
```

# *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

- [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
    - Return 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");
    }
}
```

# 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 <= 100; 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.shutdown();
}
```

# 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(1); }  
            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

```
CompletableFuture.supplyAsync(  
    () -> {  
        return ProductService.getProductDetail(productId);  
    }  
)  
.thenAccept(  
    product -> {  
        System.out.println("Got product detail from remote service " + product.getName());  
    }  
);
```

- thenAccept() takes a Consumer<T> and returns CompletableFuture<Void>.
- It has access to the result of the CompletableFuture on which it is attached.

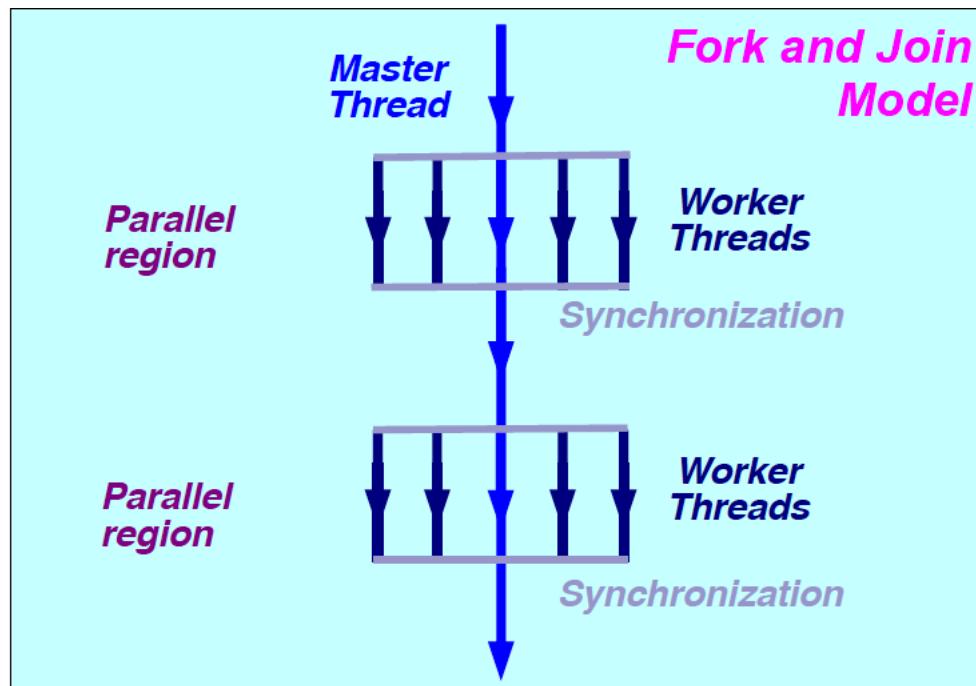
# Curs 10

Programare Paralela si Distribuita

OpenMP

- OpenMP
  - <http://www.openmp.org>
- Tutorial:
  - **TutorialOpenMP.pdf** in  
“Class Materials” - Cursuri

# OpenMP Model



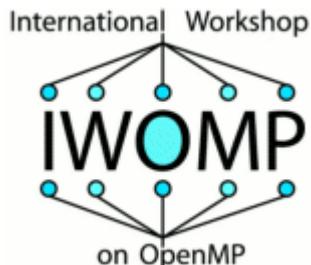
# Agenda

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

# OpenMP Runtime Functions

- `omp_set_num_threads`: Set number of threads
- `omp_get_num_threads`: Number of threads in team
- `omp_get_max_threads`: Max num of threads for parallel region
- `omp_get_thread_num`: Get thread ID

# *An Overview of OpenMP*



## Ruud van der Pas

Senior Staff Engineer  
SPARC Microelectronics  
Oracle  
Santa Clara, CA, USA

IWOMP 2011

Chicago, IL, USA  
June 13-15, 2011



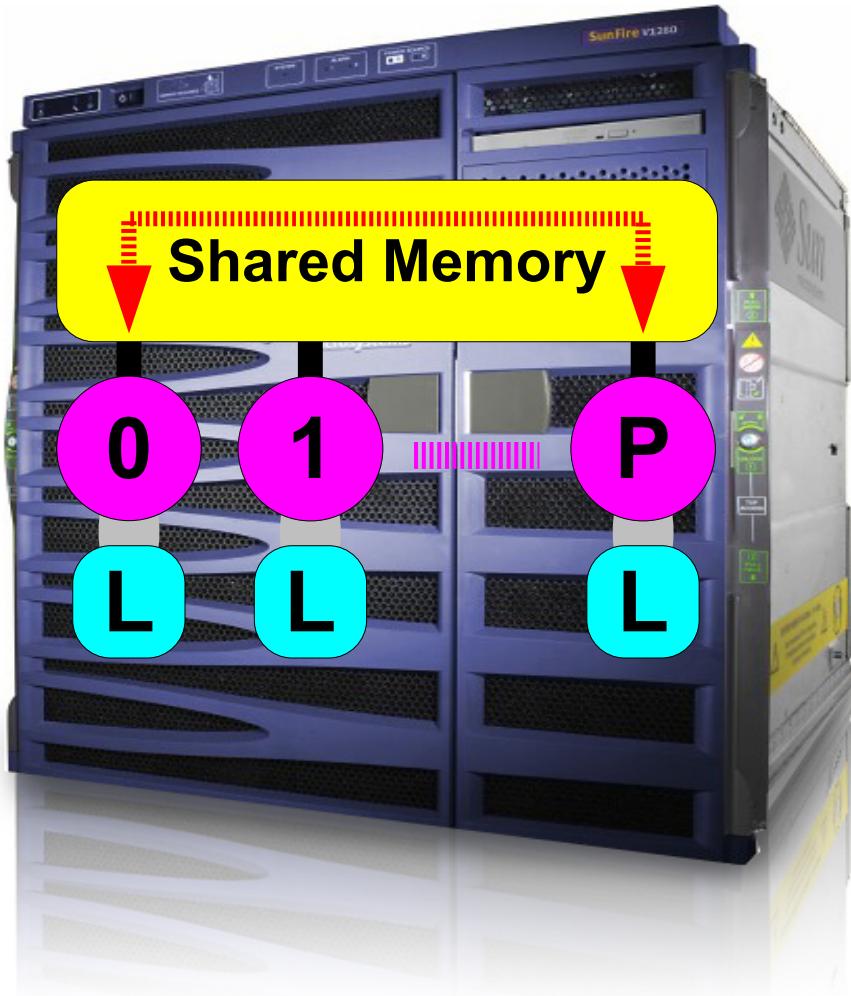
ORACLE®

# Outline

- Getting Started with OpenMP
- Using OpenMP
- What's New in OpenMP 3.1

# Getting Started With OpenMP





# OpenMP<sup>TM</sup>

<http://www.openmp.org>



<http://www.compunity.org>

ORACLE®

# OpenMP®

THE OPENMP® API SPECIFICATION FOR PARALLEL PROGRAMMING

[Subscribe to the News Feed](#)[»» OpenMP Specifications](#)[»About the OpenMP ARB](#)[»Compilers](#)[»Resources](#)[»Discussion Forum](#)

## Events

»IWOMP 2011 - 7th International Workshop on OpenMP, June 13 - 15, 2011, Chicago USA

## Input Register

Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.

[»webmaster@openmp.org](mailto:webmaster@openmp.org)

## Search OpenMP.org

Google™ Custom Search

[Archives](#)

## OpenMP News

»IWOMP 2011 - June 13-15, 2011 Chicago



The 7th annual International Workshop on OpenMP (IWOMP) is dedicated to the promotion and advancement of all aspects of parallel programming with the OpenMP API. It is the premier forum to discuss the latest developments in parallel computing and how they relate to the OpenMP parallel programming model.

# http://www.openmp.org

and third days will consist of technical papers and panel sessions during which research ideas and results will be presented and discussed.

A complete list of tutorials at IWOMP11: [Tutorials](#)

A complete list of activities during IWOMP11: [Workshop program](#)

Registration for IWOMP 2011 is now open.

Posted on May 5, 2011

## The OpenMP API

supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

[»Read about OpenMP.org](#)

## Get

[»OpenMP specs](#)

[»Use](#)  
[»OpenMP Compilers](#)

[»Learn](#)



ORACLE®

# Shameless Plug - “Using OpenMP”

**“Using OpenMP”**  
*Portable Shared Memory  
Parallel Programming*

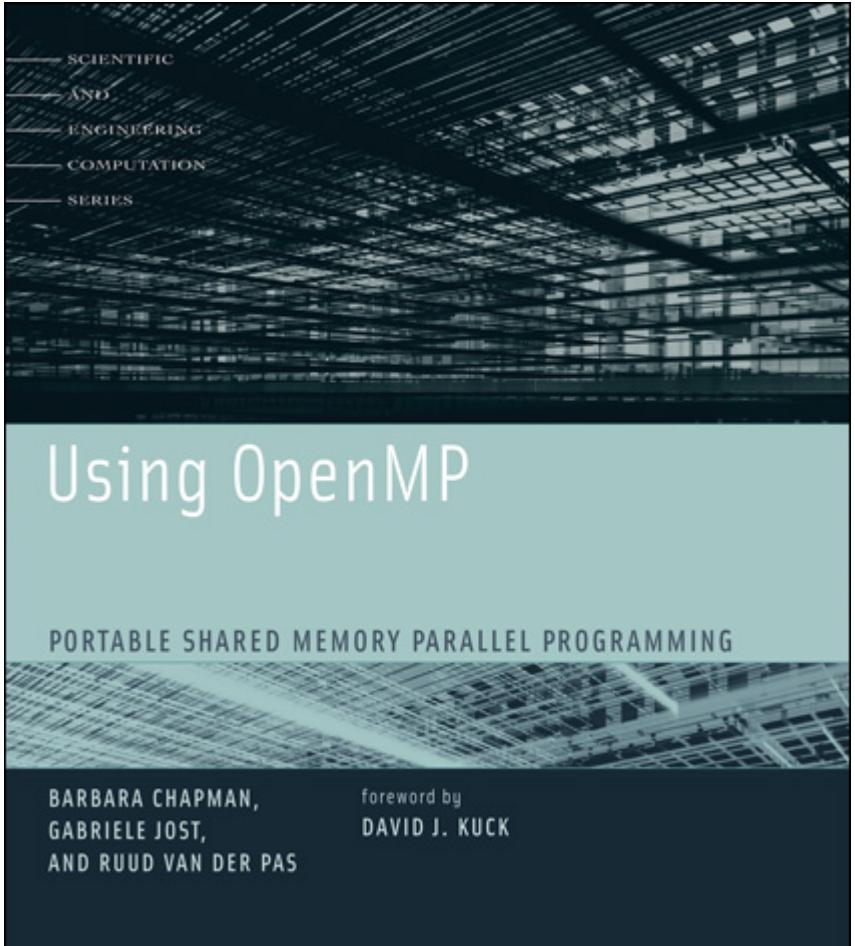
*Chapman, Jost, van der Pas*

MIT Press, 2008

ISBN-10: 0-262-53302-2

ISBN-13: 978-0-262-53302-7

List price: 35 \$US



All 41 examples are available NOW!

As well as a forum on <http://www.openmp.org>



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

»» OpenMP Specifications  
» About OpenMP  
» Compilers

» R  
» D

E  
» N  
(p  
W  
13

### Input Register

Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.  
[»webmaster@openmp.org](mailto:webmaster@openmp.org)

### Search OpenMP.org

Custom Search  
Search

## Download Book Examples and Discuss

Ruud van der Pas, one of the authors of the book *Using OpenMP - Portable Shared Memory Parallel Programming* by Chapman, Jost, and van der Pas, has made 41 of the examples in the book available for download and your use.

These source examples are available as a free download [»here](#) (a zip file) under the BSD license. Each source comes with a copy of the license. Please do not remove this.

**Get**  
» OpenMP specs

**Use**  
» OpenMP Compilers

**Learn**

**Download the examples and discuss in forum:**

[http://www.openmp.org/wp/2009/04/  
download-book-examples-and-discuss](http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss)

To make things easier, each source directory has a make file called "Makefile". This file can be used to build and run the examples in the specific directory. Before you do so, you need to activate the appropriate include line in file Makefile. There are include files for several compilers and Unix based Operating Systems (Linux, Solaris and Mac OS to precise).

These files have been put together on a best effort basis. The User's Guide that is bundled with the examples explains this in more detail.

Also, we have created a new forum, [»Using OpenMP - The Book and Examples](#), for discussion and feedback.

Posted on April 2, 2009

SARAH CHAPMAN,  
GABRIELLE JOST,  
AND RUUD VAN DER PAS  
Foreword by  
DAVID J. KUCK

» Using OpenMP – the book  
» Using OpenMP – the examples  
» Using OpenMP – the forum  
» Wikipedia  
» OpenMP Tutorial  
» More Resources

**Discuss**

# What is OpenMP?

- ***De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran***
- ***Consists of Compiler Directives, Run time routines and Environment variables***
- ***Specification maintained by the OpenMP Architecture Review Board (<http://www.openmp.org>)***
- ***Version 3.0 has been released May 2008***
  - ***The upcoming 3.1 release will be released soon***

## Members

### Permanent Members of the ARB:

- AMD (Roy Ju)
- Cray (James Beyer)
- Fujitsu (Matthijs van Waveren)
- HP (Uriel Schafer)
- IBM (Kelvin Li)
- Intel (Sanjiv Shah)
- NEC (Kazuhiro Kusano)
- The Portland Group, Inc. (Michael Wolfe)
- Oracle Corporation (Nawal Copty)
- Microsoft (-)
- Texas Instruments (Andy Fritsch)
- CAPS-Entreprise (Francois Bodin)

### Auxiliary Members of the ARB:

- ANL (Kalyan Kumaran)
- ASC/LLNL (Bronis R. de Supinski)
- cOMPunity (Barbara Chapman)
- EPCC (Mark Bull)
- LANL (John Thorp)
- NASA (Henry Jin)
- RWTH Aachen University (Dieter an Mey)

## Directors

***OpenMP is widely supported by industry, as well as the academic community***

»Wikipedia  
»OpenMP  
»More Results

## Discuss

»User Forum  
Ask the experts  
answers to your  
OpenMP questions

## Recent Posts

- IWOMP 2011
- Parallel Computing Engineering PPCE
- 3.1 Draft Published
- OpenMP 3.1 Orleans
- IWOMP 2011 Available

# When to consider OpenMP?

11

- ***Using an automatically parallelizing compiler:***
  - ***It can not find the parallelism***
    - ✓ ***The data dependence analysis is not able to determine whether it is safe to parallelize ( or not)***
  - ***The granularity is not high enough***
    - ✓ ***The compiler lacks information to parallelize at the highest possible level***
- ***Not using an automatically parallelizing compiler:***
  - ***No choice, other than doing it yourself***

ORACLE®

# Advantages of OpenMP

12

- ***Good performance and scalability***
  - *If you do it right ....*
- ***De-facto and mature standard***
- ***An OpenMP program is portable***
  - *Supported by a large number of compilers*
- ***Requires little programming effort***
- ***Allows the program to be parallelized incrementally***

ORACLE®

# OpenMP and Multicore

13

***OpenMP is ideally suited for multicore architectures***

***Memory and threading model map naturally***

***Lightweight***

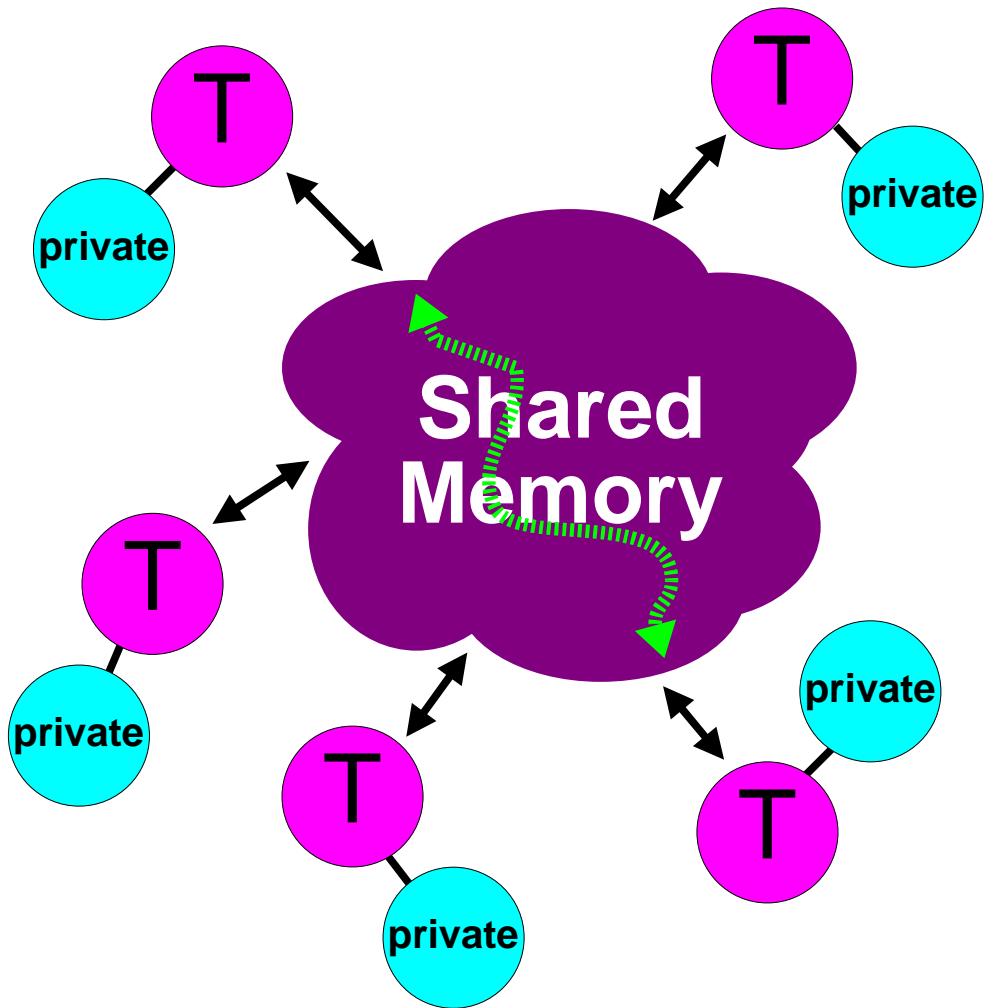
***Mature***

***Widely available and used***

**ORACLE®**

# The OpenMP Memory Model

14



- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is mostly implicit

ORACLE®

# Data-sharing Attributes

15

- In an OpenMP program, data needs to be “labeled”
- Essentially there are two basic types:
  - Shared - There is only one instance of the data
    - Threads can read and write the data simultaneously unless protected through a specific construct
    - All changes made are visible to all threads
      - But not necessarily immediately, unless enforced .....
  - Private - Each thread has a copy of the data
    - No other thread can access this data
    - Changes only visible to the thread owning the data

ORACLE®

# Private and shared clauses

16

## private (list)

- ✓ *No storage association with original object*
- ✓ *All references are to the local object*
- ✓ *Values are undefined on entry and exit*

## shared (list)

- ✓ *Data is accessible by all threads in the team*
- ✓ *All threads access the same address space*

ORACLE®

# About storage association

17

- Private variables are undefined on entry and exit of the parallel region
- A private variable within a parallel region has no storage association with the same variable outside of the region
- Use the **firstprivate** and **lastprivate** clauses to override this behavior
- We illustrate these concepts with an example

ORACLE®

# The firstprivate and lastprivate clauses

18

## firstprivate (list)

- ✓ *All variables in the list are initialized with the value the original object had before entering the parallel construct*

## lastprivate (list)

- ✓ *The thread that executes the sequentially last iteration or section updates the value of the objects in the list*

ORACLE®

# Example firstprivate

```

n = 2; idx = 4;

#pragma omp parallel default(none) private(i,TID) \
    firstprivate(idx) shared(n,a)
{ TID = omp_get_thread_num();
  idx = idx + n*TID;
  for(i=idx; i<idx+n; i++)
    a[i] = TID + 1;
} /*-- End of parallel region --*/

```

	TID = 0			TID = 1			TID = 2			
index	0	1	2	3	4	5	6	7	8	9
value					1	1	2	2	3	3



# Example lastprivate

20

```
#pragma omp parallel for default(None) lastprivate(a)
for (int i=0; i<n; i++)
{
    .....
    a = i + 1;
    .....
}
// End of parallel region

b = 2 * a; // value of b is 2*n
```

# The default clause

21

`default ( none | shared )`

C/C++

`default (none | shared | private | threadprivate )`

Fortran

`none`

- ✓ *No implicit defaults; have to scope all variables explicitly*

`shared`

- ✓ *All variables are shared*
- ✓ *The default in absence of an explicit "default" clause*

`private`

- ✓ *All variables are private to the thread*
- ✓ *Includes common block data, unless THREADPRIVATE*

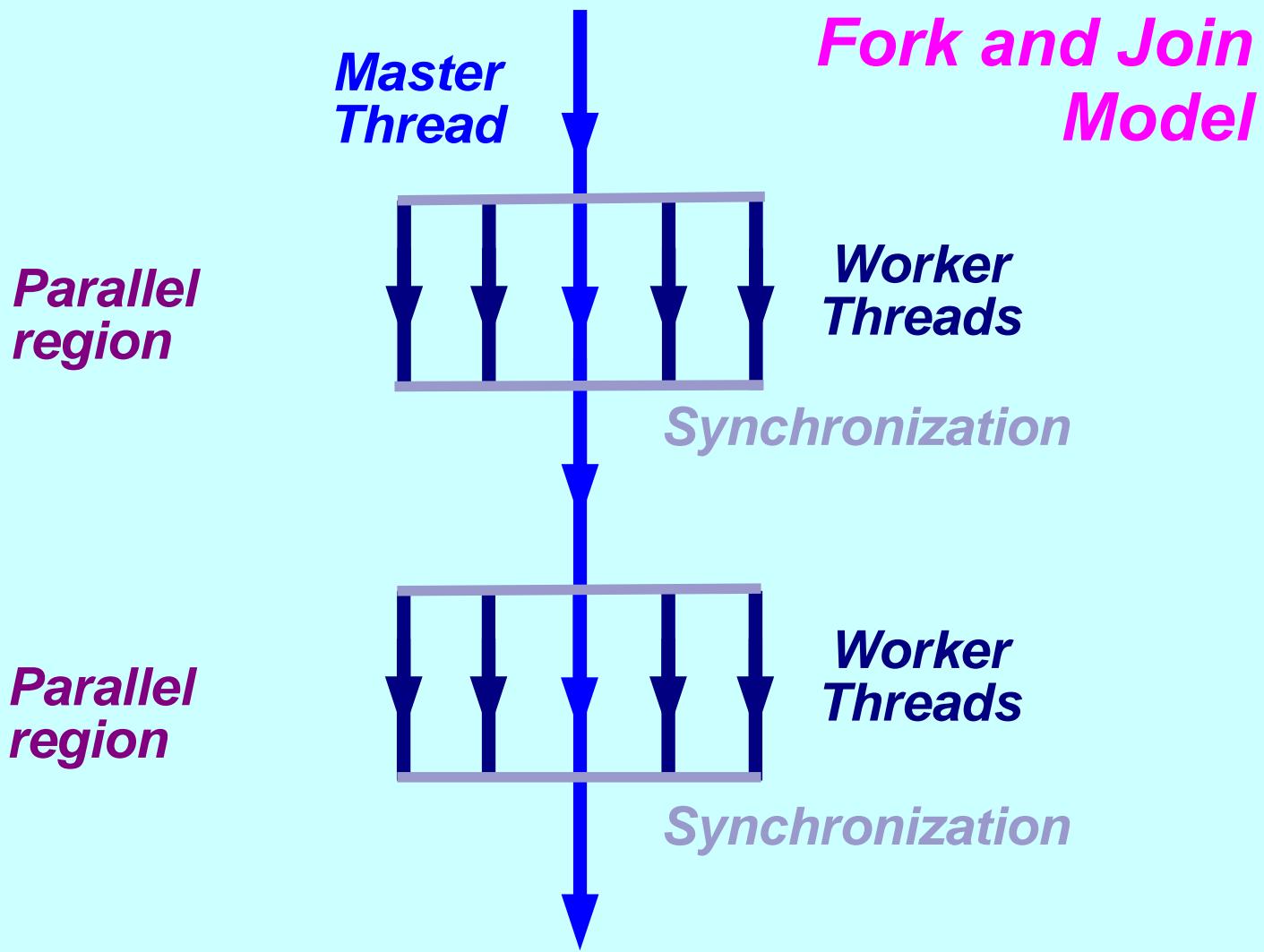
`firstprivate`

- ✓ *All variables are private to the thread; pre-initialized*

ORACLE®

# The OpenMP Execution Model

22



ORACLE®

# Defining Parallelism in OpenMP

23

- *OpenMP Team := Master + Workers*
- *A Parallel Region is a block of code executed by all threads simultaneously*
  - ▣ *The master thread always has thread ID 0*
  - ▣ *Thread adjustment (if enabled) is only done before entering a parallel region*
  - ▣ *Parallel regions can be nested, but support for this is implementation dependent*
  - ▣ *An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially*

ORACLE®

# The Parallel Region

24

**A parallel region is a block of code executed by all threads in the team**

```
#pragma omp parallel [clause[,] clause] ...
{
    "this code is executed in parallel"
} // End of parallel section (note: implied barrier)
```

```
!$omp parallel [clause[,] clause] ...
"this code is executed in parallel"
!$omp end parallel (implied barrier)
```

ORACLE®

# Parallel Region - An Example/1

25

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[ ]) {
    printf("Hello World\n");
    return(0);
}
```

ORACLE®

# Parallel Region - An Example/1

26

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[ ]) {

    #pragma omp parallel
    {
        printf("Hello World\n");

    } // End of parallel region

    return(0);
}
```

ORACLE®

# Parallel Region - An Example/2

27

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

ORACLE®

# The if clause

28

## if (scalar expression)

- ✓ *Only execute in parallel if expression evaluates to true*
- ✓ *Otherwise, execute serially*

```
#pragma omp parallel if (n > some_threshold) \
    shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        x[i] += y[i];
} /*-- End of parallel region --*/
```

ORACLE®

# Nested Parallelism

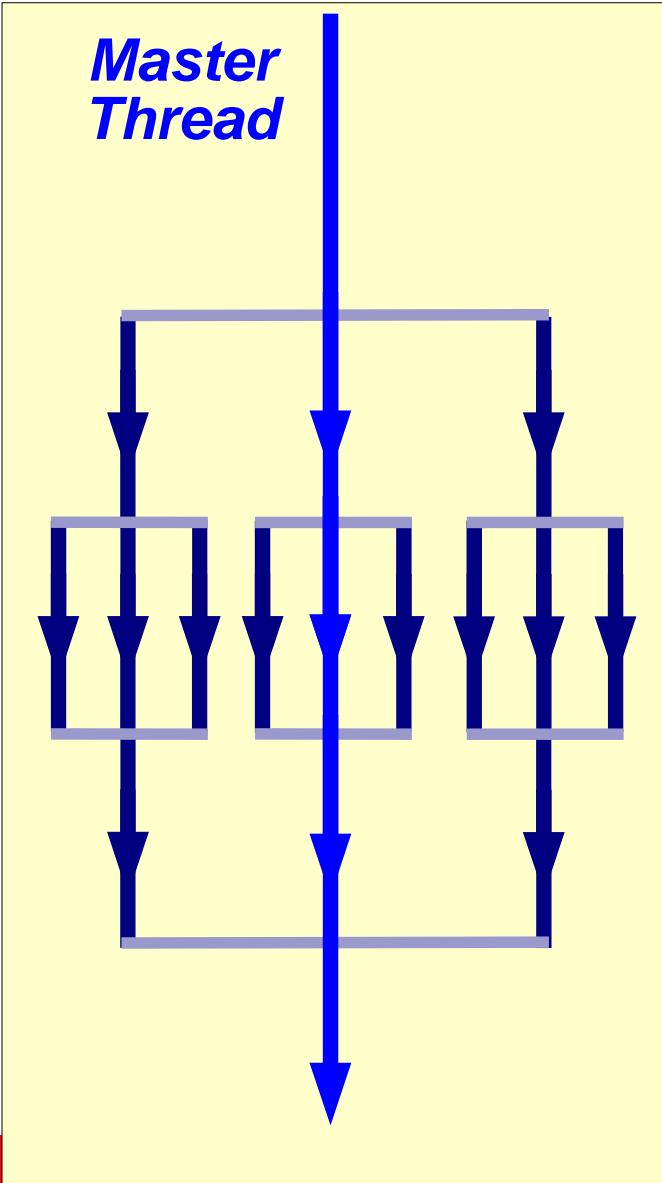
29

3-way parallel

9-way parallel

3-way parallel

**Note:** nesting level can be arbitrarily deep



Outer parallel region

Nested parallel region

Outer parallel region

ORACLE®

# Nested Parallelism Support/1

30

- ❑ ***Environment variable and runtime routines to set/get the maximum number of nested active parallel regions***

*OMP\_MAX\_ACTIVE\_LEVELS*

*omp\_set\_max\_active\_levels()*

*omp\_get\_max\_active\_levels()*

- ❑ ***Environment variable and runtime routine to set/get the maximum number of OpenMP threads available to the program***

*OMP\_THREAD\_LIMIT*

*omp\_get\_thread\_limit()*

ORACLE®

# Nested Parallelism Support/2

31

- ***Per-task internal control variables***
  - *Allow, for example, calling `omp_set_num_threads()` inside a parallel region to control the team size for next level of parallelism*
- ***Library routines to determine***
  - *Depth of nesting*  
`omp_get_level()`  
`omp_get_active_level()`
  - *IDs of parent/grandparent etc. threads*  
`omp_get_ancestor_thread_num(level)`
  - *Team sizes of parent/grandparent etc. teams*  
`omp_get_team_size(level)`

ORACLE®

# A More Elaborate Example

32

```
#pragma omp parallel if (n>limit) default(none) \
    shared(n,a,b,c,x,y,z) private(f,i,scale)
{
```

```
f = 1.0;
```

```
#pragma omp for nowait
```

```
for (i=0; i<n; i++)
    z[i] = x[i] + y[i];
```

```
#pragma omp for nowait
```

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

```
....
```

```
#pragma omp barrier
```

```
scale = sum(a,0,n) + sum(z,0,n) + f;
```

```
....
```

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

Statement is executed by all threads

**parallel loop**  
 (work is distributed)

**parallel loop**  
 (work is distributed)

**synchronization**

Statement is executed by all threads

ORACLE®

# Using OpenMP



# Using OpenMP

34

- We have just seen a glimpse of OpenMP
- To be practically useful, much more functionality is needed
- Covered in this section:
  - Many of the language constructs
  - Features that may be useful or needed when running an OpenMP application
- Note that the tasking concept is covered in a separate section

ORACLE®

# Components of OpenMP

35

## *Directives*

- ◆ *Parallel region*
- ◆ *Worksharing constructs*
- ◆ *Tasking*
- ◆ *Synchronization*
- ◆ *Data-sharing attributes*

## *Runtime environment*

- ◆ *Number of threads*
- ◆ *Thread ID*
- ◆ *Dynamic thread adjustment*
- ◆ *Nested parallelism*
- ◆ *Schedule*
- ◆ *Active levels*
- ◆ *Thread limit*
- ◆ *Nesting level*
- ◆ *Ancestor thread*
- ◆ *Team size*
- ◆ *Wallclock timer*
- ◆ *Locking*

## *Environment variables*

- ◆ *Number of threads*
- ◆ *Scheduling type*
- ◆ *Dynamic thread adjustment*
- ◆ *Nested parallelism*
- ◆ *Stacksize*
- ◆ *Idle threads*
- ◆ *Active levels*
- ◆ *Thread limit*

# Directive format

- **C: directives are case sensitive**
  - *Syntax:* #pragma omp directive [clause [clause] ...]
- **Continuation:** use \ in pragma
- **Conditional compilation:** \_OPENMP macro is set
- **Fortran: directives are case insensitive**
  - *Syntax:* sentinel directive [clause [[,] clause]...]
  - *The sentinel is one of the following:*
    - ✓ !\$OMP or C\$OMP or \*\$OMP      (*fixed format*)
    - ✓ !\$OMP                                (*free format*)
- **Continuation:** follows the language syntax
- **Conditional compilation:** !\$ or C\$ -> 2 spaces

# The reduction clause - Example

37

```

    sum = 0.0
 !$omp parallel default(none) &
 !$omp shared(n,x) private(i)
 !$omp do reduction (+:sum)
   do i = 1, n
     sum = sum + x(i)
   end do
 !$omp end do
 !$omp end parallel
 print *,sum
  
```

**Variable SUM is a shared variable**

- ☞ **Care needs to be taken when updating shared variable SUM**
- ☞ **With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided**

ORACLE®

# The reduction clause

**reduction ( operator: list )**

**C/C++**

**reduction ( [operator | intrinsic] ) : list )**

**Fortran**

- ✓ **Reduction variable(s) must be shared variables**
- ✓ **A reduction is defined as:**

**Fortran**

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr_list)
x = intrinsic (expr_list, x)
```

**C/C++**

```
x = x operator expr
x = expr operator x
x++, ++x, x--, --x
x <binop> = expr
```

**Check the docs  
for details**

- ✓ **Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed**
- ✓ **The reduction can be hidden in a function call**

# Fortran - Allocatable Arrays

39

- Fortran allocatable arrays whose status is “currently allocated” are allowed to be specified as **private**, **lastprivate**, **firstprivate**, **reduction**, or **copyprivate**

```
integer, allocatable,dimension (:) :: A
integer i

allocate (A(n)) ←

!$omp parallel private (A)
do i = 1, n
    A(i) = i
end do
...
!$omp end parallel
```

ORACLE®

# Barrier/1

40

*Suppose we run each of these two loops in parallel over i:*

```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

*This may give us a wrong answer (one day)*

**Why ?**

ORACLE®

# Barrier/2

41

*We need to have updated all of a[ ] first, before using a[ ] \**

```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
```

*wait !*

---

*barrier*

```
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

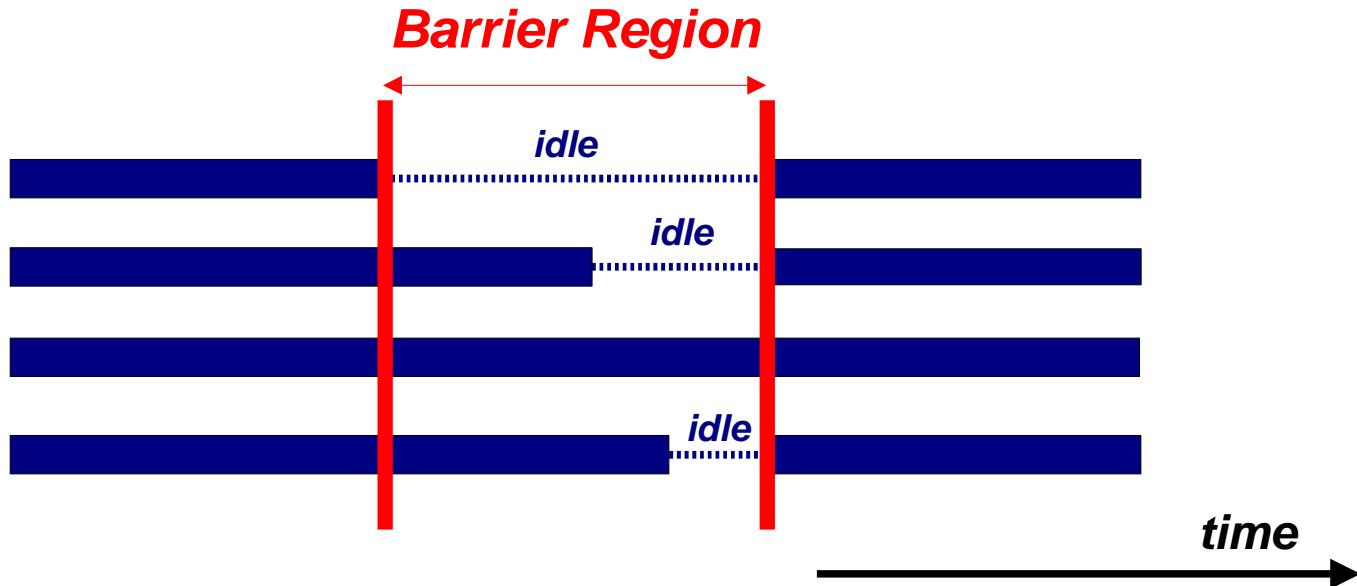
*All threads wait at the barrier point and only continue when all threads have reached the barrier point*

*\*) If there is the guarantee that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case*

ORACLE®

# Barrier/3

42



*Barrier syntax in OpenMP:*

```
#pragma omp barrier
```

```
! $omp barrier
```

ORACLE®

# When to use barriers ?

43

- ❑ *If data is updated asynchronously and data integrity is at risk*
- ❑ *Examples:*
  - *Between parts in the code that read and write the same section of memory*
  - *After one timestep/iteration in a solver*
- ❑ *Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors*
- ❑ *Therefore, use them with care*

ORACLE®

# The nowait clause

44

- ❑ *To minimize synchronization, some directives support the optional nowait clause*
  - *If present, threads do not synchronize/wait at the end of that particular construct*
- ❑ *In C, it is one of the clauses on the pragma*
- ❑ *In Fortran, it is appended at the closing part of the construct*

```
#pragma omp for nowait
{
    :
}
```

```
!$omp do
    :
!
!$omp end do nowait
```

ORACLE®

# The Worksharing Constructs

45

```
#pragma omp for
{
    ....
}

!$OMP DO
    ....
!$OMP END DO
```

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

 !$OMP SECTIONS
    ....
 !$OMP END SECTIONS
```

```
#pragma omp single
{
    ....
}

 !$OMP SINGLE
    ....
 !$OMP END SINGLE
```

- ☞ *The work is distributed over the threads*
- ☞ *Must be enclosed in a parallel region*
- ☞ *Must be encountered by all threads in the team, or none at all*
- ☞ *No implied barrier on entry; implied barrier on exit (unless nowait is specified)*
- ☞ *A work-sharing construct does not launch any new threads*

ORACLE®

# The Workshare construct

46

**Fortran has a fourth worksharing construct:**

```
!$OMP WORKSHARE
```

```
<array syntax>
```

```
!$OMP END WORKSHARE [NOWAIT]
```

**Example:**

```
!$OMP WORKSHARE
```

```
    A(1:M) = A(1:M) + B(1:M)
```

```
!$OMP END WORKSHARE NOWAIT
```

ORACLE®

# The omp for/do directive

47

```
#pragma omp for [clauses]
for (.....)
{
    <code-block>
}
```

```
!$omp do [clauses]
do ...
    <code-block>
end do
 !$omp end do[nowait]
```

*The iterations of the loop are distributed over the threads*

ORACLE®

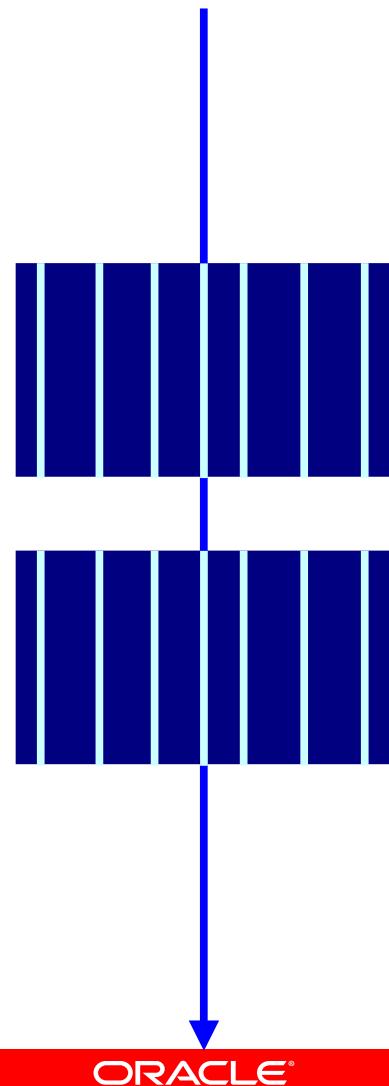
# The omp for directive - Example

48

```
#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)
```



# C++: Random Access Iterator Loops

49

*Parallelization of random access iterator loops is supported*

```
void iterator_example()
{
    std::vector vec(23);
    std::vector::iterator it;

#pragma omp for default(none)shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```

ORACLE®

# Loop Collapse

50

- Allows parallelization of perfectly nested loops without using nested parallelism
- The **collapse** clause on for/do loop indicates how many loops should be collapsed
- Compiler forms a single loop and then parallelizes it

```
!$omp parallel do collapse(2) ...
  do i = il, iu, is
    do j = jl, ju, js
      do k = kl, ku, ks
        .....
      end do
    end do
  end do
 !$omp end parallel do
```

ORACLE®

# The schedule clause/1

51

```
schedule ( static | dynamic | guided | auto [, chunk] )
schedule ( runtime )
```

## static [, chunk]

- ✓ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✓ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*
  - *Details are implementation defined*
- ✓ *Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region*

ORACLE®

# The schedule clause/2

52

## *Example static schedule*

*Loop of length 16, 4 threads:*

Thread	0	1	2	3
<i>no chunk*</i>	1-4	5-8	9-12	13-16
<i>chunk = 2</i>	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

*\*) The precise distribution is implementation defined*

ORACLE®

# The schedule clause/3

53

## dynamic [, chunk]

- ✓ *Fixed portions of work; size is controlled by the value of chunk*
- ✓ *When a thread finishes, it starts on the next portion of work*

## guided [, chunk]

- ✓ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

## auto

- ✓ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

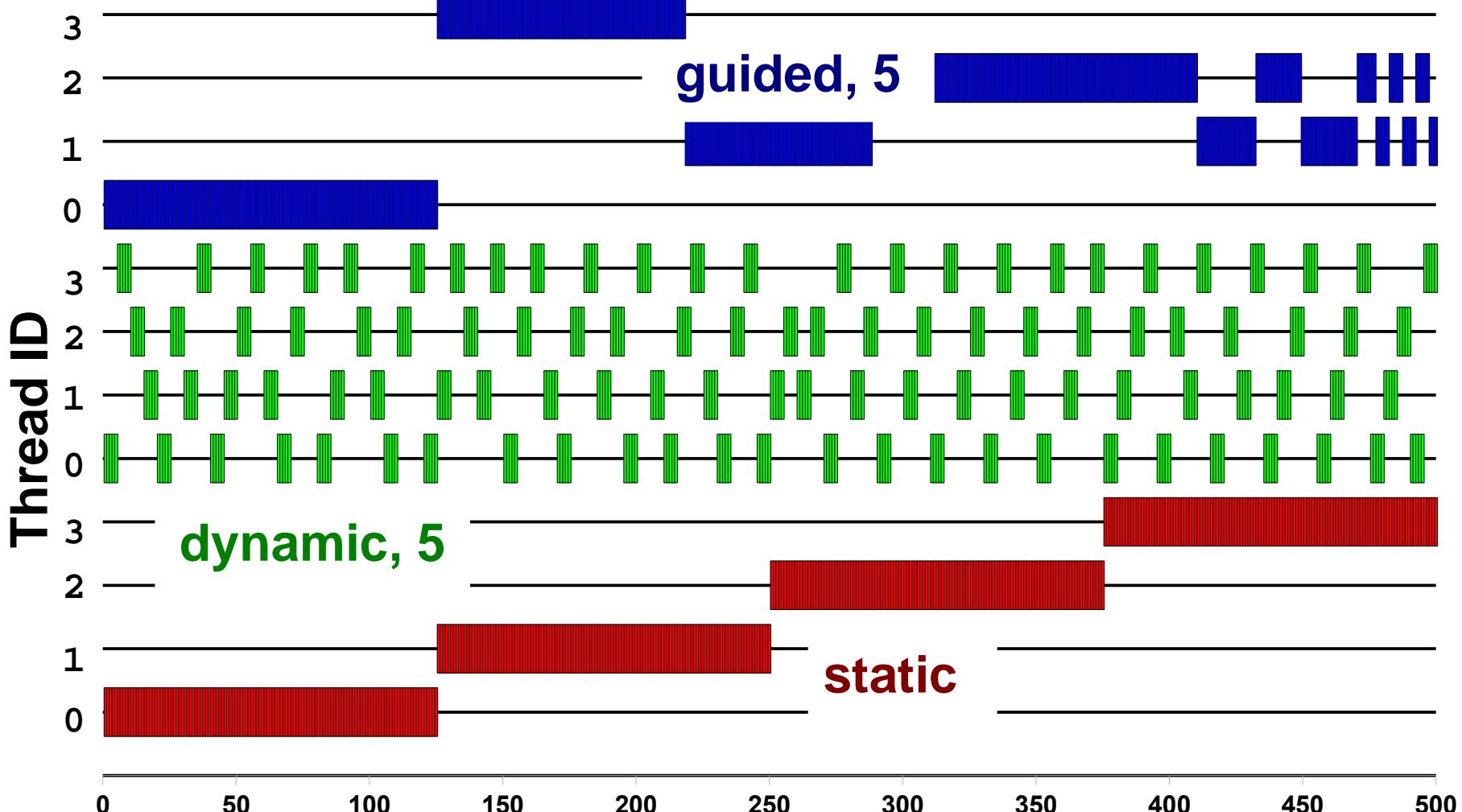
## runtime

- ✓ *Iteration scheduling scheme is set at runtime through environment variable OMP\_SCHEDULE*

ORACLE®

# Experiment - 500 iterations, 4 threads

54



Iteration Number

ORACLE®

# Schedule Kinds Functions

55

- **Makes schedule (runtime) more general**
- **Can set/get schedule it with library routines:**

`omp_set_schedule()`  
`omp_get_schedule()`

- **Also allows implementations to add their own schedule kinds**

ORACLE®

# Parallel sections

56

```
#pragma omp sections [clauses]
{
    #pragma omp section
    { . . . . }
    #pragma omp section
    { . . . . }
    . . .
}
```

*Individual section  
blocks are executed  
in parallel*

```
!$omp sections [clauses]

 !$omp section
 { . . . . }
 !$omp section
 { . . . . }
 . . .
 !$omp end sections [nowait]
```

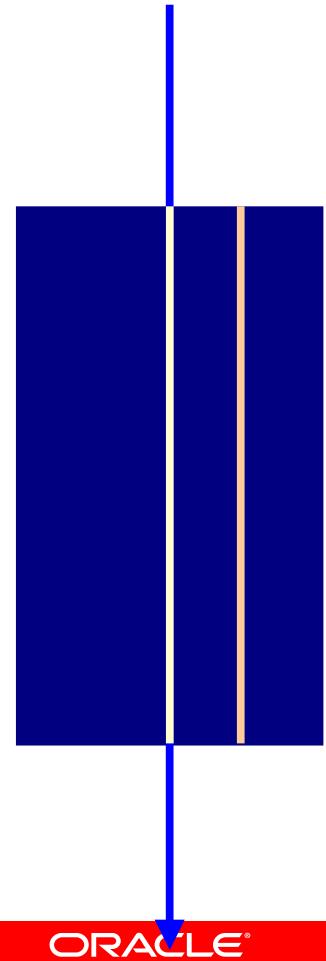
ORACLE®

# The Sections Directive - Example

57

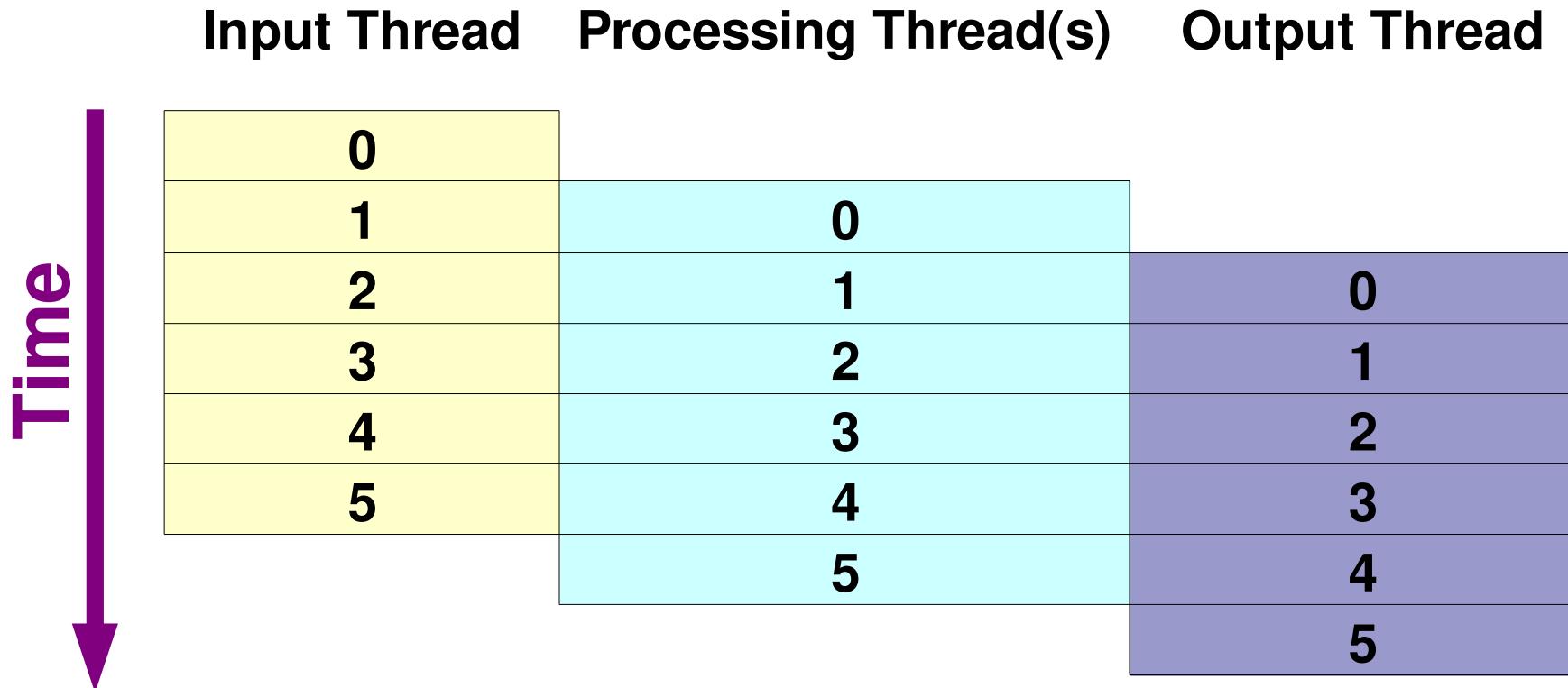
```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

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



# Overlap I/O and Processing/1

58



ORACLE®

# Overlap I/O and Processing/2

59

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
}
} /*-- End of parallel sections --*/
```

**Input Thread**

**Processing  
Thread(s)**

**Output Thread**

# The Single Directive

60

*Only one thread in the team executes the code enclosed*

```
#pragma omp single [private] [firstprivate] \
    [copyprivate] [nowait]
{
    <code-block>
}
```

```
!$omp single [private] [firstprivate]
<code-block>
!$omp end single [copyprivate] [nowait]
```

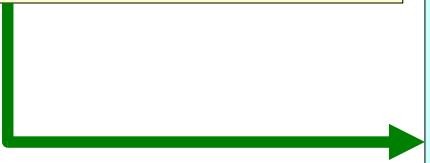
ORACLE®

# Single processor region/1

61

## *Original Code*

```
.....
"read A[0..N-1]";
.....
```



***Only one thread executes the single region***

***This construct is ideally suited for I/O or initializations***

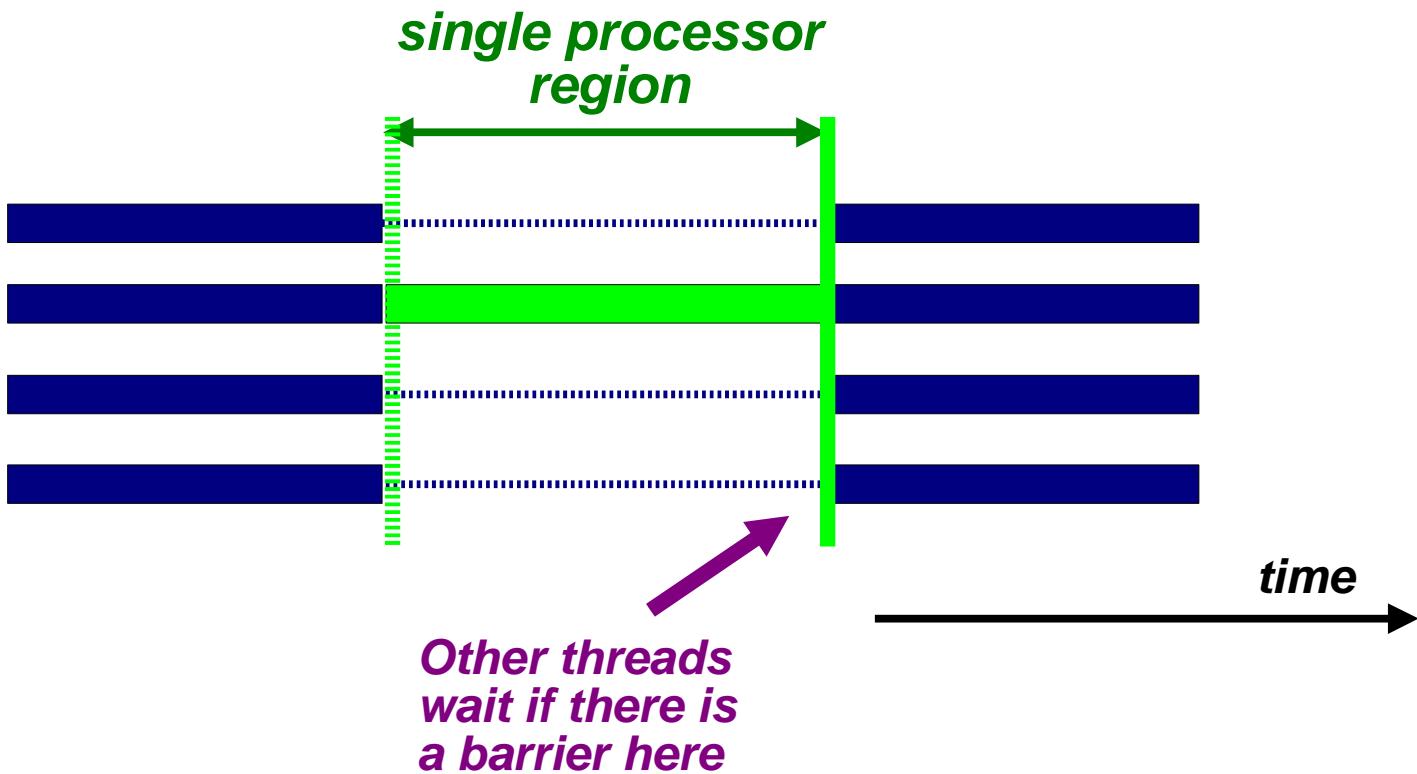
```
#pragma omp parallel \
  shared (A)
{
  .....
#pragma omp single nowait
{ "read A[0..N-1]; }

  .....
#pragma omp barrier
  "use A"
}
```

***Parallel Version***

# Single processor region/2

62



ORACLE®

# Combined work-sharing constructs

63

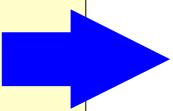
```
#pragma omp parallel
#pragma omp for
for (...)
```

```
!$omp parallel
 !$omp do
 ...
 !$omp end do
 !$omp end parallel
```

```
!$omp parallel
 !$omp workshare
 ...
 !$omp end workshare
 !$omp end parallel
```

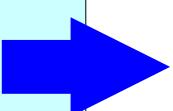
```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
!$omp parallel
 !$omp sections
 ...
 !$omp end sections
 !$omp end parallel
```



*Single PARALLEL loop*

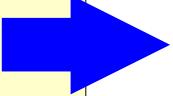
```
#pragma omp parallel for
for (...)
```



```
!$omp parallel do
...
 !$omp end parallel do
```

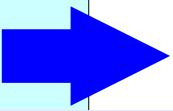
*Single WORKSHARE loop*

```
:omp parallel workshare
...
 !$omp end parallel workshare
```



*Single PARALLEL sections*

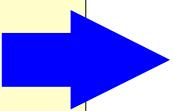
```
#pragma omp parallel
sections
, ,
```



```
!$omp parallel sections
```

...

```
!$omp end parallel sections
```



# Orphaning

64

```
:
#pragma omp parallel
{
  :
  (void) dowork();
  :
}
```

```
void dowork()
{
  :
  #pragma omp for
  for (int i=0;i<n;i++)
  {
    :
  }
  :
}
```

orphaned  
work-sharing  
directive

- *The OpenMP specification does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned*
- *That is, they can appear outside the lexical extent of a parallel region*

ORACLE®

# More on orphaning

65

```
(void) dowork(); !- Sequential FOR

#pragma omp parallel
{
    (void) dowork(); !- Parallel FOR
}
```

```
void dowork()
{
#pragma omp for
for (i=0;....)
{
    :
}
}
```

- When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored

ORACLE®

# Example - Parallelizing Bulky Loops

66

```
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...
    b = ... a ...
    c[i] = ....
    .....
    for (j=0; j<m; j++)
    {
        <a lot more code in this loop>
    }
    .....
}
```

ORACLE®

# Step 1: “Outlining”

```
for (i=0; i<n; i++) /* Parallel loop */
{
  (void) FuncPar(i,m,c,...)
}
```

*Still a sequential program*

*Should behave identically*

*Easy to test for correctness*

*But, parallel by design*

```
void FuncPar(i,m,c,...)
{
  float a, b; /* Private data */
  int j;
  a = ...
  b = ... a ...
  c[i] = ....
  .....
  for (j=0; j<m; j++)
  {
    <a lot more code in this loop>
  }
  .....
}
```



# Step 2: Parallelize

68

```
#pragma omp parallel for private(i) shared(m,c,...)

for (i=0; i<n; i++) /* Parallel loop */
{
    (void) FuncPar(i,m,c,...)
} /*-- End of parallel for --*/
```

*Minimal scoping required*

*Less error prone*

```
void FuncPar(i,m,c,...)
{
    float a, b; /* Private data */
    int j;
    a = ...
    b = ... a ...
    c[i] = ....
    .....
    for (j=0; j<m; j++)
    {
        <a lot more code in this loop>
    }
    .....
}
```

# Additional Directives/1

69

```
#pragma omp master  
{<code-block>}
```

```
!$omp master  
    <code-block>  
 !$omp end master
```

```
#pragma omp critical [ (name) ]  
{<code-block>}
```

```
!$omp critical [ (name) ]  
    <code-block>  
 !$omp end critical [ (name) ]
```

```
#pragma omp atomic
```

```
!$omp atomic
```

ORACLE®

# The Master Directive

70

***Only the master thread executes the code block:***

```
#pragma omp master  
{<code-block>}
```

```
!$omp master  
    <code-block>  
 !$omp end master
```

***There is no implied barrier on entry or exit !***

ORACLE®

# Critical Region/1

71

*If sum is a shared variable, this loop can not run in parallel by simply using a “#pragma omp for”*

```
for (i=0; i < n; i++) {
    . . .
    sum += a[i];
    . . .
}
```

```
#pragma omp parallel for
for (i=0; i < n; i++) {
    . . .
    #pragma omp critical
        {sum += a[i];}
    . . .
}
```

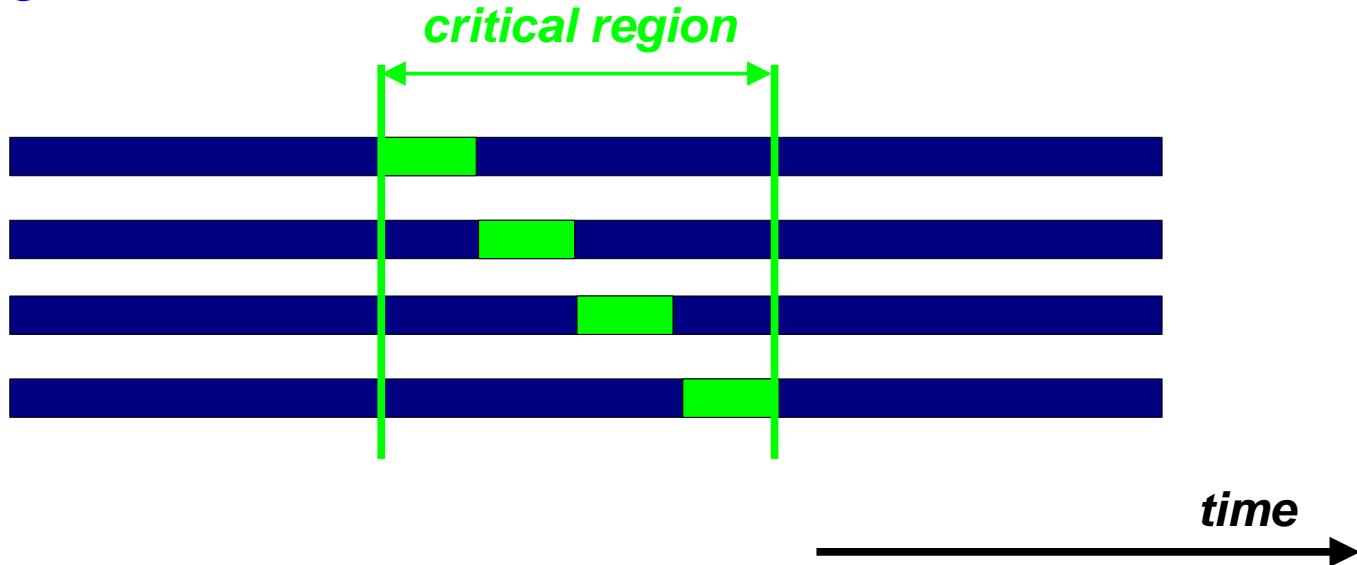
*All threads execute the update, but only one at a time will do so*

ORACLE®

# Critical Region/2

72

- ***Useful to avoid a race condition, or to perform I/O (but that still has random order)***
- ***Be aware that there is a cost associated with a critical region***



ORACLE®

# Critical and Atomic constructs

73

**Critical: All threads execute the code, but only one at a time:**

```
#pragma omp critical [ (name) ]
{<code-block>}
```

```
!$omp critical [ (name) ]
<code-block>
```

```
!$omp end critical [ (name) ]
```

**There is no implied barrier on entry or exit !**

**Atomic: only the loads and store are atomic ....**

```
#pragma omp atomic
<statement>
```

```
!$omp atomic
<statement>
```

**This is a lightweight, special form of a critical section**

```
#pragma omp atomic
a[indx[i]] += b[i];
```

ORACLE®

# Additional Directives/2

74

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
<code-block>
!$omp end ordered
```

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

ORACLE®

# Additional Directives/2

***The enclosed block of code is executed in the order in which iterations would be executed sequentially:***

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
<code-block>
 !$omp end ordered
```

***May introduce serialization (could be expensive)***

***Ensure that all threads in a team have a consistent view of certain objects in memory:***

```
#pragma omp flush
[(list)]
```

```
!$omp flush [(list)]
```

***In the absence of a list, all visible variables are flushed***

# The flush directive

76

## Thread A

```
x = 0
```

```
■  
■  
■  
■
```

```
x = 1
```

```
■  
■  
■  
■
```

## Thread B

```
while (x == 0)  
{  
    "wait"  
}
```

*If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)*

ORACLE®

# Implied Flush Regions/1

77

- During a barrier region
- At exit from worksharing regions, unless a nowait is present
- At entry to and exit from parallel, critical, ordered and parallel worksharing regions
- During `omp_set_lock` and `omp_unset_lock` regions
- During `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` and `omp_test_nest_lock` regions, if the region causes the lock to be set or unset
- Immediately before and after every task scheduling point

# Implied Flush Regions/2

78

- At entry to and exit from atomic regions, where the list contains only the variable updated in the atomic construct
- A flush region is not implied at the following locations:
  - At entry to a worksharing region
  - At entry to or exit from a master region

ORACLE®

# *OpenMP and Global Data*

# Global data - An example

80

```

program global_data
  ....
  include "global.h"
  ....
 !$omp parallel do private(j)
    do j = 1, n
      call suba(j)
    end do
 !$omp end parallel do
  .....
  
```

*file global.h*

```
common /work/a(m,n),b(m)
```

```
subroutine suba(j)
  .....
  include "global.h"
  .....
```

do i = 1, m  
 b(i) = j  
 end do

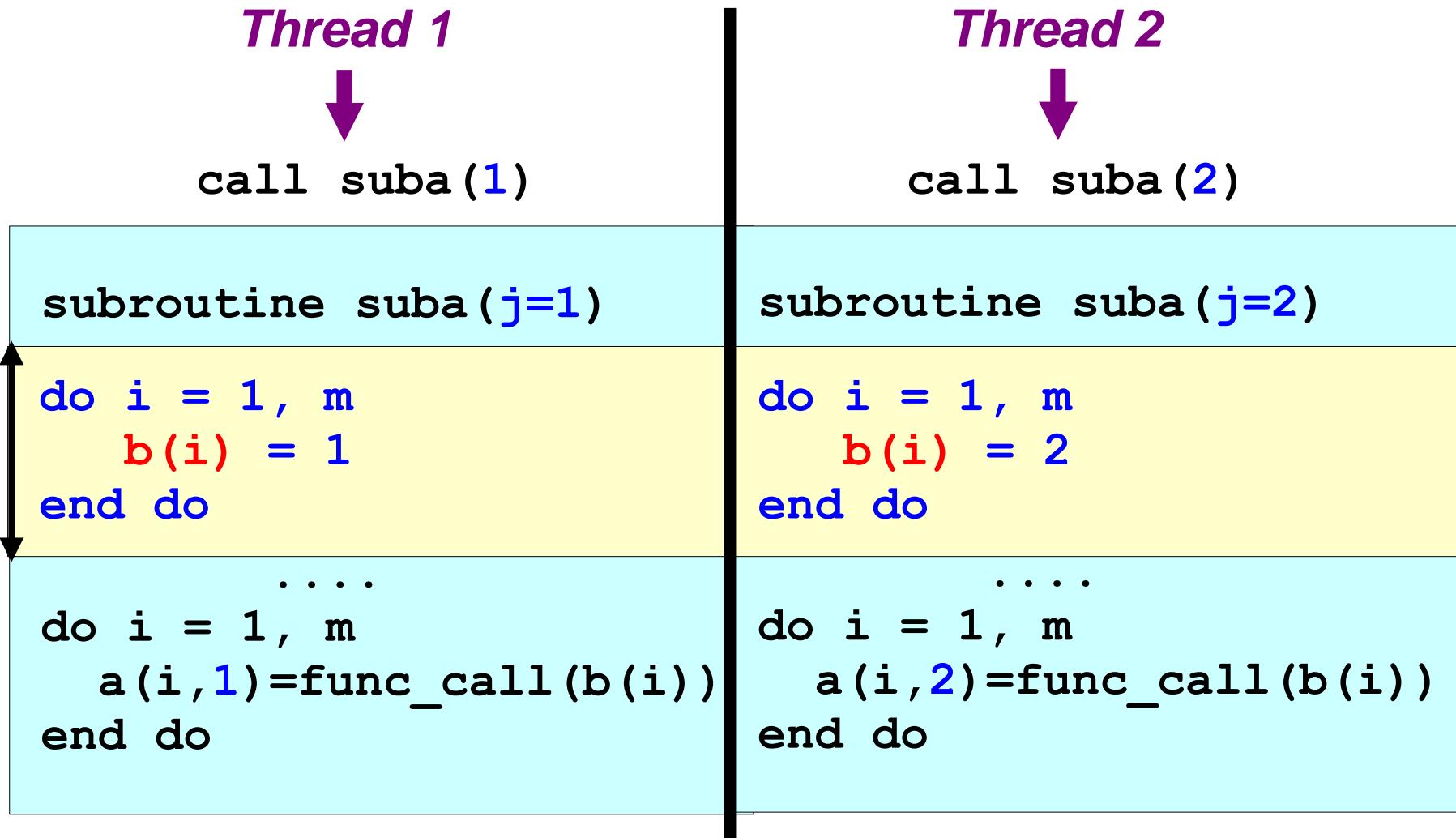
do i = 1, m  
 a(i,j) =  
 func\_call(b(i))  
 end do

```
return
end
```

**Data Race !**

# Global data - A Data Race!

81



ORACLE®

# Example - Solution

82

```
program global_data
    ...
    include "global_ok.h"
    ...
!$omp parallel do private(j)
    do j = 1, n
        call suba(j)
    end do
!$omp end parallel do
    ...
```

- By expanding array B, we can give each thread unique access to its storage area
- Note that this can also be done using dynamic memory (allocatable, malloc, ....)

```
integer, parameter:: file global_ok.h
nthreads=4
common /work/a(m,n)
common /tprivate/b(m,nthreads)
```

```
subroutine suba(j)
    ...
    include "global_ok.h"
    ...
TID = omp_get_thread_num() + 1
do i = 1, m
    b(i,TID) = j
end do

do i = 1, m
    a(i,j)=func_call(b(i,TID))
end do

return
end
```

ORACLE

# About global data

83

- Global data is shared and requires special care
- A problem may arise in case multiple threads access the same memory section simultaneously:
  - Read-only data is no problem
  - Updates have to be checked for race conditions
- It is your responsibility to deal with this situation
- In general one can do the following:
  - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel
  - Manually create thread private copies of the latter
  - Use the thread ID to access these private copies
- Alternative: Use OpenMP's `threadprivate` directive

ORACLE®

# The threadprivate directive

84

```
#pragma omp threadprivate (list)  
!$omp threadprivate (/cb/ [,/cb/] ...)
```

- *Thread private copies of the designated global variables and common blocks are created*
- *Several restrictions and rules apply when doing this:*
  - *The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)*
    - ✓ *Oracle implementation supports changing the number of threads*
  - *Initial data is undefined, unless copyin is used*
  - .....
- *Check the documentation when using threadprivate !*

ORACLE®

# Example - Solution 2

85

```

program global_data
  ....
  include "global_ok2.h" →
  ....
 !$omp parallel do private(j)
   do j = 1, n
     call suba(j)
   end do
 !$omp end parallel do
  .....
  stop
end
  
```

file global\_ok2.h

```

common /work/a(m,n)
common /tprivate/b(m)
!$omp
threadprivate(/tprivate/)
  
```

```

subroutine suba(j)
  .....
  include "global_ok2.h" ←
  .....
  
```

```

do i = 1, m
  b(i) = j
end do

do i = 1, m
  a(i,j) = func_call(b(i))
end do

return
end
  
```

- ☞ **The compiler creates thread private copies of array B, to give each thread unique access to its storage area**
- ☞ **Note that the number of copies is automatically adjusted to the number of threads**

ORACLE®

# The copyin clause

86

## copyin (list)

- ✓ *Applies to THREADPRIVATE common blocks only*
- ✓ *At the start of the parallel region, data of the master thread is copied to the thread private copies*

### Example:

```
common /cblock/velocity
common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel      &
!$omp default (private) &
!$omp copyin ( /cblock/, zfield )
```

ORACLE®

# C++ and Threadprivate

87

- ❑ *As of OpenMP 3.0, it has been clarified where/how threadprivate objects are constructed and destructed*
- ❑ *Allow C++ static class members to be threadprivate*

```
class T {  
public:  
    static int i;  
    #pragma omp threadprivate(i)  
    ...  
};
```

ORACLE®

# *OpenMP Runtime Routines*

# OpenMP Runtime Functions/1

## Name

`omp_set_num_threads`  
`omp_get_num_threads`  
`omp_get_max_threads`

`omp_get_thread_num`  
`omp_get_num_procs`  
`omp_in_parallel`  
`omp_set_dynamic`

`omp_get_dynamic`  
`omp_set_nested`

`omp_get_nested`  
`omp_get_wtime`  
`omp_get_wtick`

## Functionality

**Set number of threads**

**Number of threads in team**

**Max num of threads for parallel region**

## Get thread ID

**Maximum number of processors**

**Check whether in parallel region**

**Activate dynamic thread adjustment**

*(but implementation is free to ignore this)*

**Check for dynamic thread adjustment**

**Activate nested parallelism**

*(but implementation is free to ignore this)*

**Check for nested parallelism**

**Returns wall clock time**

**Number of seconds between clock ticks**

C/C++ : Need to include file `<omp.h>`

Fortran : Add “use omp\_lib” or include file “omp\_lib.h”

# OpenMP Runtime Functions/2

90

## Name

`omp_set_schedule`

`omp_get_schedule`

`omp_get_thread_limit`

`omp_set_max_active_levels`

`omp_get_max_active_levels`

`omp_get_level`

`omp_get_active_level`

`omp_get_ancestor_thread_num` *Thread id of ancestor thread*

`omp_get_team_size (level)` *Size of the thread team at this level*

## Functionality

*Set schedule (if “runtime” is used)*

*Returns the schedule in use*

*Max number of threads for program*

*Set number of active parallel regions*

*Number of active parallel regions*

*Number of nested parallel regions*

*Number of nested active par. regions*

C/C++ : Need to include file `<omp.h>`

Fortran : Add “use omp\_lib” or include file “`omp_lib.h`”

ORACLE®

# OpenMP locking routines

91

- ***Locks provide greater flexibility over critical sections and atomic updates:***
  - Possible to implement asynchronous behavior
  - Not block structured
- ***The so-called lock variable, is a special variable:***
  - C/C++: type `omp_lock_t` and `omp_nest_lock_t` for nested locks
  - Fortran: type `INTEGER` and of a `KIND` large enough to hold an address
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, and behavior is undefined, in case a lock variable is used without the appropriate initialization***

ORACLE®

# Nested locking

92

- *Simple locks: may not be locked if already in a locked state*
- *Nestable locks: may be locked multiple times by the same thread before being unlocked*
- *In the remainder, we discuss simple locks only*
- *The interface for functions dealing with nested locks is similar (but using nestable lock variables):*

## Simple locks

`omp_init_lock`  
`omp_destroy_lock`  
`omp_set_lock`  
`omp_unset_lock`  
`omp_test_lock`

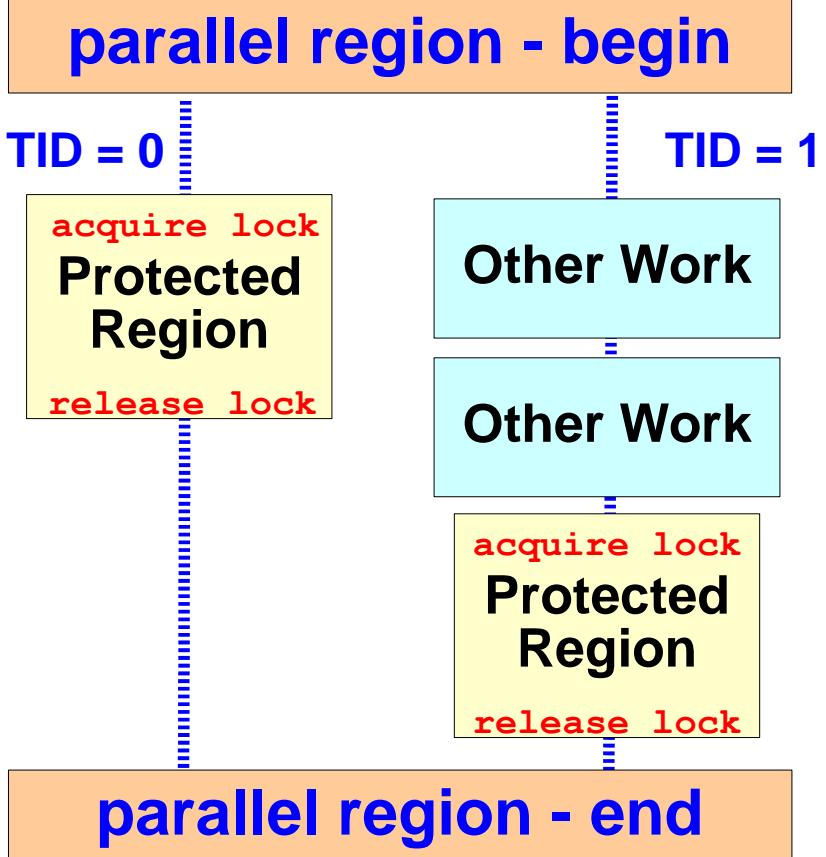
## Nestable locks

`omp_init_nest_lock`  
`omp_destroy_nest_lock`  
`omp_set_nest_lock`  
`omp_unset_nest_lock`  
`omp_test_nest_lock`

ORACLE®

# OpenMP locking example

93



- ◆ *The protected region contains the update of a shared variable*
- ◆ *One thread acquires the lock and performs the update*
- ◆ *Meanwhile, the other thread performs some other work*
- ◆ *When the lock is released again, the other thread performs the update*

ORACLE®

# Locking Example - The Code

94

```

Program Locks
  ...
Call omp_init_lock (LCK)
Initialize lock variable

!$omp parallel shared(LCK)
Check availability of lock
  (also sets the lock)

Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
  Call Do_Something_Else()
End Do

Call Do_Work()
Release lock again

Call omp_unset_lock (LCK)
Remove lock association

!$omp end parallel

Call omp_destroy_lock (LCK)

Stop
End
  
```



# Example output for 2 threads

95

```

TID: 1 at 09:07:27 => entered parallel region
TID: 1 at 09:07:27 => done with WAIT loop and has the lock
TID: 1 at 09:07:27 => ready to do the parallel work
TID: 1 at 09:07:27 => this will take about 18 seconds
TID: 0 at 09:07:27 => entered parallel region
TID: 0 at 09:07:27 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:32 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:37 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:42 => WAIT for lock - will do something else for 5 seconds
TID: 1 at 09:07:45 => done with my work
TID: 1 at 09:07:45 => done with work loop - released the lock
TID: 1 at 09:07:45 => ready to leave the parallel region
TID: 0 at 09:07:47 => done with WAIT loop and has the lock
TID: 0 at 09:07:47 => ready to do the parallel work
TID: 0 at 09:07:47 => this will take about 18 seconds
TID: 0 at 09:08:05 => done with my work
TID: 0 at 09:08:05 => done with work loop - released the lock
TID: 0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
  
```

Used to check the answer

**Note: program has been instrumented to get this information**

ORACLE®

# *OpenMP Environment Variables*

ORACLE®

# OpenMP Environment Variables

97

OpenMP environment variable	Default for Oracle Solaris Studio
<code>OMP_NUM_THREADS n</code>	1
<code>OMP_SCHEDULE "schedule,[chunk]"</code>	static, “N/P”
<code>OMP_DYNAMIC { TRUE   FALSE }</code>	TRUE
<code>OMP_NESTED { TRUE   FALSE }</code>	FALSE
<code>OMP_STACKSIZE size [B K M G]</code>	4 MB (32 bit) / 8 MB (64-bit)
<code>OMP_WAIT_POLICY [ACTIVE   PASSIVE]</code>	PASSIVE
<code>OMP_MAX_ACTIVE_LEVELS</code>	4
<code>OMP_THREAD_LIMIT</code>	1024

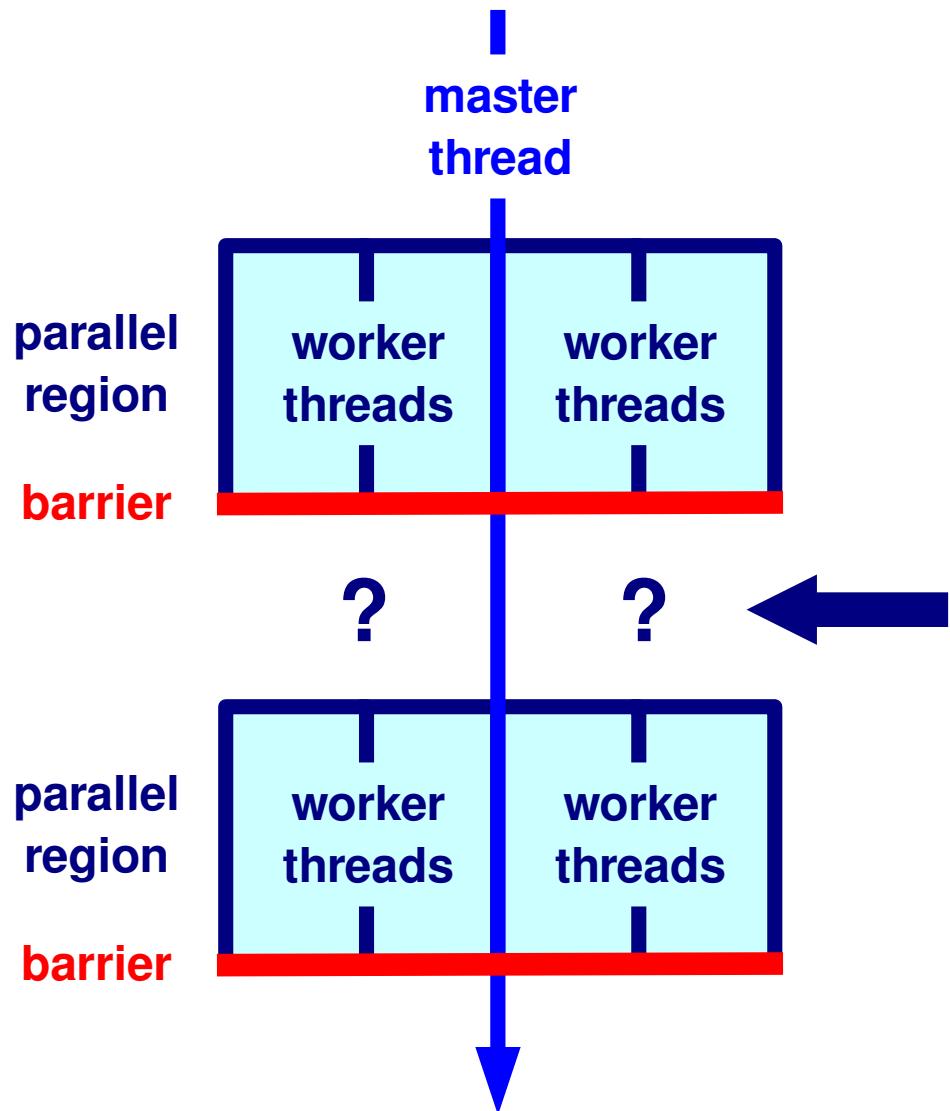
**Note:**

*The names are in uppercase, the values are case insensitive*

**ORACLE®**

# Implementing the Fork-Join Model

98



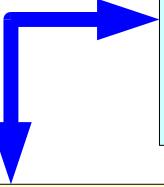
*Use the `OMP_WAIT_POLICY` environment variable to control the behaviour of idle threads*

ORACLE®

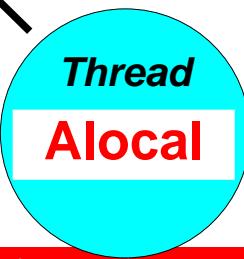
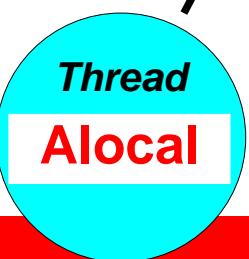
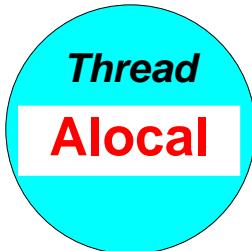
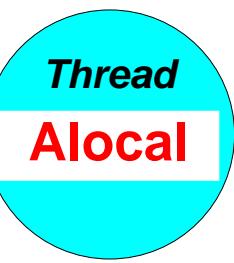
# About the Stack

99

```
void myfunc(float *Aglobal)
{
    int Alocal;
    .....
}
```



```
#omp parallel shared(Aglobal)
{
    (void) myfunc(&Aglobal);
}
```



**Variable Alocal is in private memory,  
managed by the thread owning it,  
and stored on the so-called stack**

ORACLE®

# Tasking In OpenMP



# Tasking in OpenMP

101

- When any thread encounters a **task construct**, a new explicit task is generated
  - Tasks can be nested
- Execution of explicitly generated tasks is assigned to one of the threads in the current team
  - This is subject to the thread's availability and thus could be immediate or deferred until later
- Completion of the task can be guaranteed using a **task synchronization** construct

ORACLE®

# The Tasking Construct

102

## Define a task:

```
#pragma omp task
```

```
!$omp task
```

A **task** is a specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

A **task region** is a region consisting of all code encountered during the execution of a task.

The **data environment** consists of all the variables associated with the execution of a given task. The data environment for a given task is constructed from the data environment of the generating task at the time the task is generated.

ORACLE®

# Task Completion in OpenMP

103

- **Task completion** occurs when the end of the structured block associated with the construct that generated the task is reached
- Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of **task synchronization constructs**
  - Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits
- A task synchronization construct is a **taskwait** or a **barrier** construct

ORACLE®

# Task Completion

104

Explicitly wait on the completion of child tasks:

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

ORACLE®

# Example/1

105

```
#include <stdlib.h>
#include <stdio.h>
```

```
int main(int argc, char *argv[ ]) {
    printf("A ");
    printf("race ");
    printf("car ");

    printf("\n");
    return(0);
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

*What will this program print ?*

ORACLE®

# Example/2

106

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[ ]) {

    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");

    } // End of parallel region

    printf("\n");
    return(0);
}
```

*What will this program print using 2 threads ?*

ORACLE®

# Example/3

107

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

*Note that this program could for example also print  
“A A race race car car ” or  
“A race A car race car”, or  
“A race A race car car”, .....  
although I have not observed this (yet)*

# Example/4

108

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[ ]) {

#pragma omp parallel
{
  #pragma omp single
  {
    printf("A ");
    printf("race ");
    printf("car ");
  }
} // End of parallel region

printf("\n");
return(0);
}
```

*What will this program print  
using 2 threads ?*

ORACLE®

# Example/5

109

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

*But now only 1 thread  
executes .....*

ORACLE®

# Example/6

110

```
int main(int argc, char *argv[ ]) {  
  
#pragma omp parallel  
{  
    #pragma omp single  
{  
        printf("A ");  
        #pragma omp task  
        {printf("race ");}  
        #pragma omp task  
        {printf("car ");}  
    }  
} // End of parallel region  
  
printf("\n");  
return(0);  
}
```

*What will this program print  
using 2 threads ?*

ORACLE®

# Example/7

111

```
$ cc -fopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$
```

*Tasks can be executed in arbitrary order*

ORACLE®

# Example/8

112

```

int main(int argc, char *argv[ ]) {

#pragma omp parallel
{
  #pragma omp single
  {
    printf("A ");
    #pragma omp task
    {printf("race ");}
    #pragma omp task
    {printf("car ");}
    printf("is fun to watch ");
  }
} // End of parallel region

printf("\n");
return(0);
}

```

*What will this program print  
using 2 threads ?*

ORACLE®

# Example/9

113

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
```

A is fun to watch race car  
\$ ./a.out

A is fun to watch race car  
\$ ./a.out

A is fun to watch car race  
\$

***Tasks are executed at a task execution point***

ORACLE®

# Example/10

114

```

int main(int argc, char *argv[ ]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("car ");}
            #pragma omp task
            {printf("race ");}
            #pragma omp taskwait
            printf("is fun to watch ");
        }
    } // End of parallel region
    printf("\n");return 0;
}

```

*What will this program print  
using 2 threads ?*

ORACLE®

# Example/11

115

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

***Tasks are executed first now***

ORACLE®

# Clauses on the task directive

116

**if(*scalar-expression*)**

if false, create an undeferred task,  
encountering thread must suspend  
the encountering task region, resume  
execution of the current task region  
until the task is completed  
any task can resume after suspension

**untied**

**default(shared | none)**

**private(*list*)**

**firstprivate(*list*)**

**shared(*list*)**

**New in OpenMP 3.1:**

**final(*scalar-expression*)**

**mergeable**

if true, the generated task is a final task  
if the task is an undeferred task or an  
included task, the implementation may  
generate a merged task

ORACLE®

# Task Scheduling Points in OpenMP/1

117

- Threads are allowed to suspend the current task region at a **task scheduling point** in order to execute a different task
  - If the suspended task region is for a **tied** task, the initially assigned thread resumes execution of the suspended task
  - If it is **untied**, any thread may resume its execution

ORACLE®

# Task Scheduling Points/2

118

- Whenever a thread reaches a **task scheduling point**, the implementation may cause it to perform a *task switch*, beginning or resuming execution of a different task bound to the current team
- Task scheduling points are implied at:
  - The point immediately following the generation of an explicit task
  - After the last instruction of a task region
  - In taskwait and taskyield regions
  - In implicit and explicit barrier regions
- In addition to this, the implementation may insert task scheduling points in untied tasks

ORACLE®

# Task Scheduling Points/3

119

- Task scheduling points dynamically divide task regions into parts
- When a thread encounters a task scheduling point, it may do of the following:
  - Begin execution of a tied task bound to the current team
  - Resume any suspended task region bound to the current team to which it is tied
  - Begin execution of an untied task bound to the current team
  - Resume any suspended untied task region bound to the current team
- If more than one of these choices is available, the behavior is undetermined

ORACLE®

# *Tasking Examples Using OpenMP 3.0*

# Example - A Linked List

121

```
.....  
while(my_pointer) {  
  
    (void) do_independent_work (my_pointer);  
  
    my_pointer = my_pointer->next ;  
} // End of while loop  
.....
```

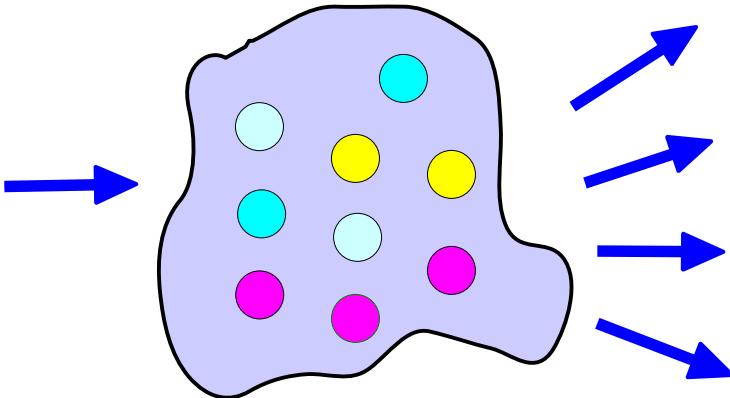
***Hard to do before OpenMP 3.0:  
First count number of iterations,  
then convert while loop to for loop***

ORACLE®

# The Tasking Example

122

Encountering  
thread adds  
task(s) to  
pool



Threads execute  
tasks in the pool

*Developer specifies tasks in application  
Run-time system executes tasks*

ORACLE®

# Example - A Linked List With Tasking

123

```
my_pointer = listhead;  
  
#pragma omp parallel  
{  
    #pragma omp single nowait  
    {  
        while(my_pointer) {  
            #pragma omp task firstprivate(my_pointer)  
            {  
                (void) do_independent_work (my_pointer);  
            }  
            my_pointer = my_pointer->next ;  
        }  
    } // End of single - no implied barrier (nowait)  
} // End of parallel region - implied barrier
```

OpenMP Task is specified here  
(executed in parallel)



ORACLE®

# Example – Fibonacci Numbers

124

***The Fibonacci Numbers are defined as follows:***

$$\begin{aligned} F(0) &= 1 \\ F(1) &= 1 \end{aligned}$$

$$F(n) = F(n-1) + F(n-2) \quad (n=2, 3, 4, \dots)$$

***Sequence:***

***1, 1, 2, 3, 5, 8, 13, 21, 34, ....***

**ORACLE®**

# Recursive Algorithm\*

125

```
long comp_fib_numbers(int n){  
    // Basic algorithm: f(n) = f(n-1) + f(n-2)  
  
    long fnm1, fnm2, fn;  
  
    if ( n == 0 || n == 1 ) return(n);  
  
    fnm1 = comp_fib_numbers(n-1);  
  
    fnm2 = comp_fib_numbers(n-2);  
  
    fn     = fnm1 + fnm2;  
  
    return(fn);  
}
```

*\*) Not very efficient, used for demo purposes only*

ORACLE®

# Parallel Recursive Algorithm

126

```
long comp_fib_numbers(int n){  
    // Basic algorithm: f(n) = f(n-1) + f(n-2)  
  
    long fnm1, fnm2, fn;  
  
    if ( n == 0 || n == 1 ) return(n);  
  
#pragma omp task shared(fnm1)  
{fnm1 = comp_fib_numbers(n-1);}  
  
#pragma omp task shared(fnm2)  
{fnm2 = comp_fib_numbers(n-2);}  
  
#pragma omp taskwait  
fn    = fnm1 + fnm2;  
  
    return(fn);  
}
```

ORACLE®

# Driver Program

127

```
#pragma omp parallel shared(nthreads)
{
    #pragma omp single nowait
    {
        result = comp_fib_numbers(n);
    } // End of single
} // End of parallel region
```

ORACLE®

# Parallel Recursive Algorithm - V2

128

```

long comp_fib_numbers(int n){

    // Basic algorithm: f(n) = f(n-1) + f(n-2)

    long fnm1, fnm2, fn;

    if ( n == 0 || n == 1 ) return(n);
    if ( n < 20 ) return(comp_fib_numbers(n-1) +
                        comp_fib_numbers(n-2));

#pragma omp task shared(fnm1)
{fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
{fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
fn    = fnm1 + fnm2;

return(fn);
}

```

**ORACLE®**

# Performance Example\*

129

```
$ export OMP_NUM_THREADS=1
$ ./fibonacci-omp.exe 40
Parallel result for n = 40: 102334155 (1 threads
                               needed 5.63 seconds)
$ export OMP_NUM_THREADS=2
$ ./fibonacci-omp.exe 40
Parallel result for n = 40: 102334155 (2 threads
                               needed 3.03 seconds)
$
```

*\*) MacBook Pro Core 2 Duo*

ORACLE®

# What's New In OpenMP 3.1 ?



# About OpenMP 3.1

131

- A brand new update on the specifications
  - Following the evolutionary model
- Public comment phase ended May 1, 2011
  - Started February 2011
- It takes time for compilers to support any new standard
  - OpenMP 3.1 is no exception
- OpenMP continues to evolve!

ORACLE®

# New OpenMP 3.1 Features/1

132

- The “reduction” clause for C/C++ now supports the “min” and “max” operators
- Data environment for “firstprivate” extended to allow “intent(in)” in Fortran and const qualified types in C/C++
- Addition of “omp\_in\_final” runtime routine
  - Supports specialization of final or included task regions
- New OMP\_PROC\_BIND environment variable
- Corrections and clarifications
  - Incorrect use of “omp\_integer\_kind” in Fortran interfaces in Appendix D has been corrected
  - Description of some examples expanded and clarified

ORACLE®

# New OpenMP 3.1 Features/2

133

- Additions to tasking
  - The “mergeable” clauses
    - When a mergeabe clause is present on a task construct, and the generated task is undeferred or included, the implementation may generate a merged task instead
      - A merged task is a task whose data environment, inclusive of ICVs, is the same as that of its generating task region

**Note: ICV = Internal Control Variable**

ORACLE®

# New OpenMP 3.1 Features/3

134

- Additions to tasking
  - The “final” clause
    - If true, the generated task will be a final task
    - All tasks generated encountered during execution of a final task will generate included tasks
      - An included task is a task for which execution is sequentially included in the generating task region; that is, it is undeferred and executed immediately by the encountering thread

ORACLE®

# New OpenMP 3.1 Features/4

135

- Additions to tasking
  - The “taskyield” construct has been added
    - Current task can be suspended in favor of execution of another task
    - Allows user defined task switching points
  - This construct includes an explicit task scheduling point

```
#pragma omp taskyield
```

```
!$omp taskyield
```

ORACLE®

# New OpenMP 3.1 Features/5

136

- Enhancements for the “atomic” construct
  - New “update” clause
  - New “read”, “write” and “capture” forms
  - Disallow closely nested parallel regions within “atomic”
    - Clarification of existing restriction

ORACLE®

# **Summary OpenMP**



# Summary OpenMP

138

- OpenMP provides for a small, but yet powerful, programming model
- It can be used on a shared memory system of any size
  - This includes a single socket multicore system
- Compilers with OpenMP support are widely available
- The tasking concept opens up opportunities to parallelize a wider range of applications
- OpenMP continues to evolve!
  - The new 3.1 specifications are again a step forward

ORACLE®

***Thank You And ..... Stay Tuned !***

***[ruud.vanderpas@oracle.com](mailto:ruud.vanderpas@oracle.com)***

# **Hardware and Software Engineered to Work Together**

**ORACLE®**

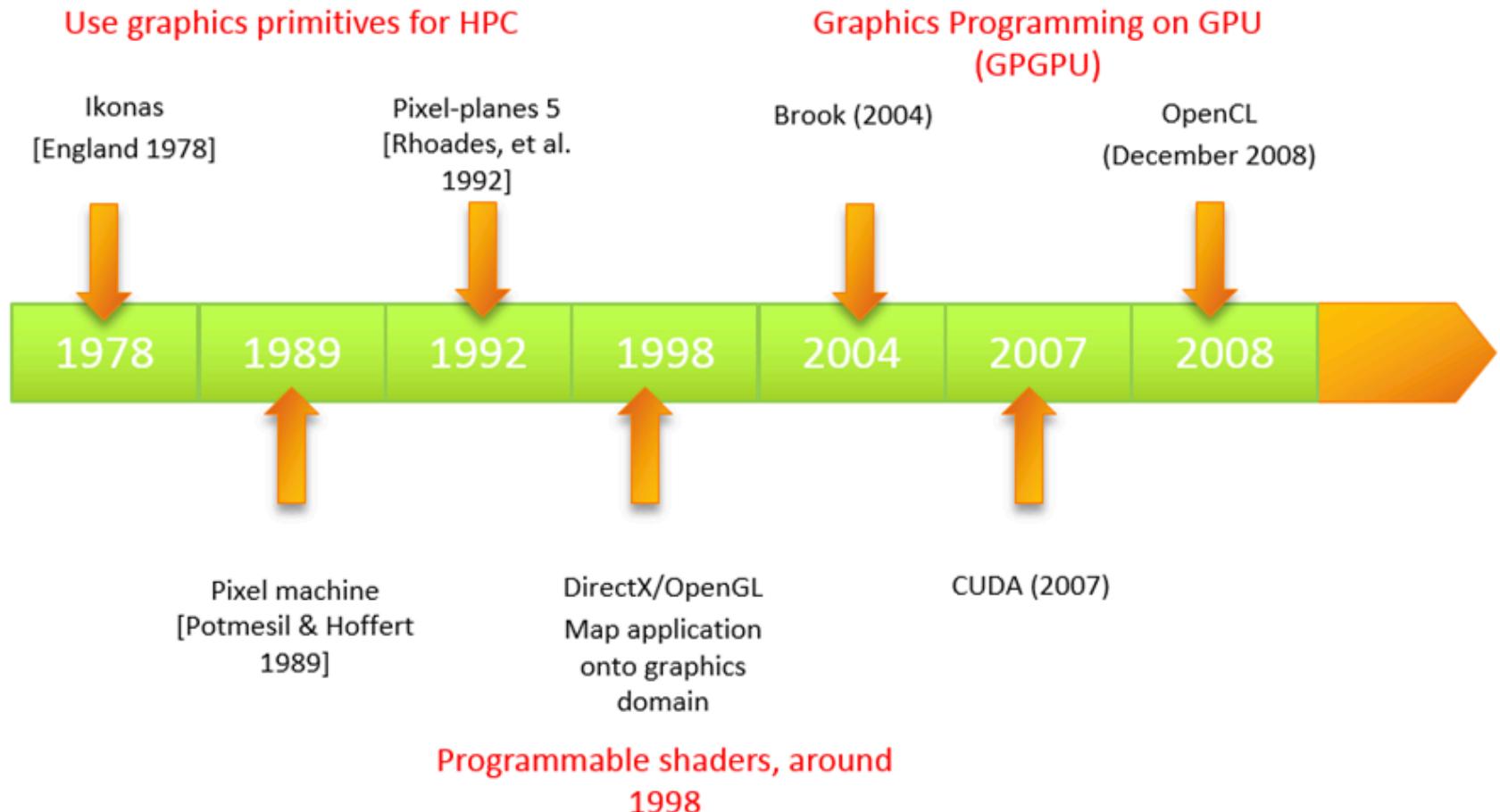
# Curs 11

## Introducere in CUDA

# Ce este CUDA?

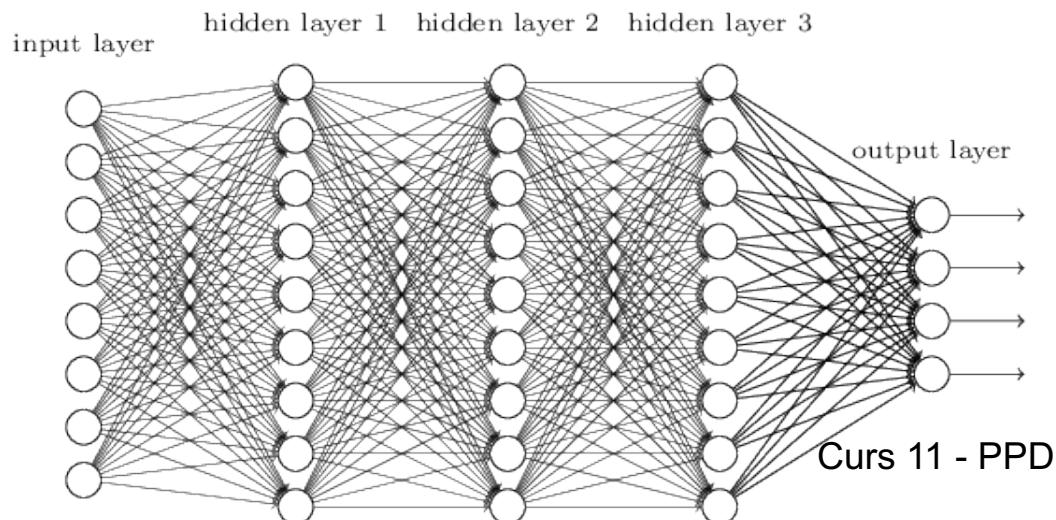
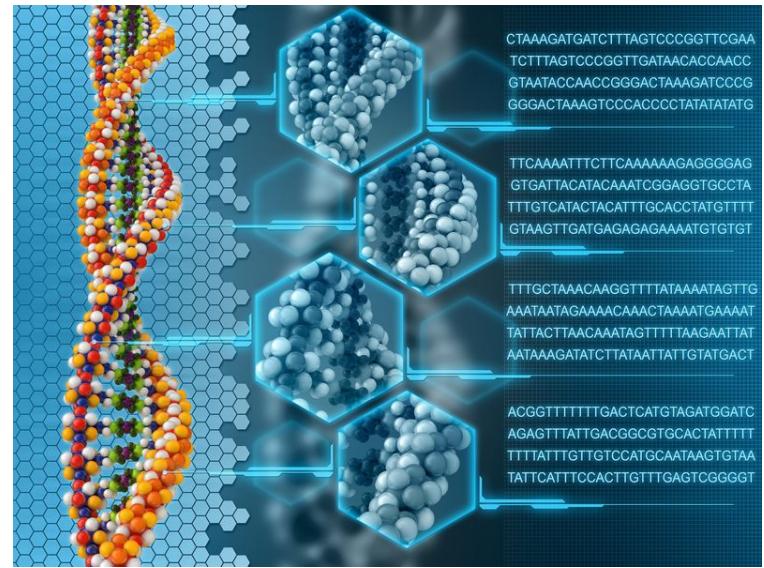
- Compute Unified Device Architecture"
- o platforma de programare paralela->
- Arhitectura care foloseste GPU pt calcul general
  - permite cresterea performantei
- Released by NVIDIA in 2007
- Model de programare
  - Bazat pe extensii C / C++ - pt a permite ‘heterogeneous programming’
  - API pt gestionarea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

# Historic



## Aplicatii

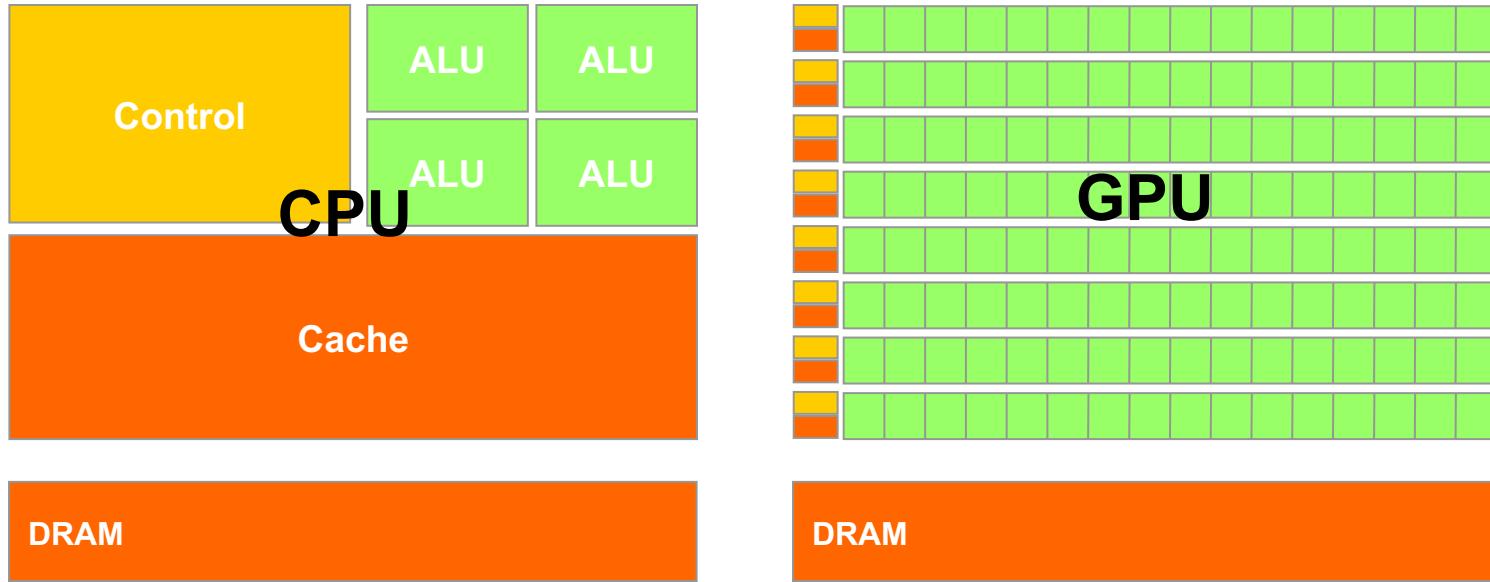
- Bioinformatica
  - Calcul financiar
  - Deep learning
  - Molecular dynamics simulation
  - Video and audio coding and manipulation
  - 3D imaging and visualization
  - Consumer game physics
  - virtual reality products
  - ...



# 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

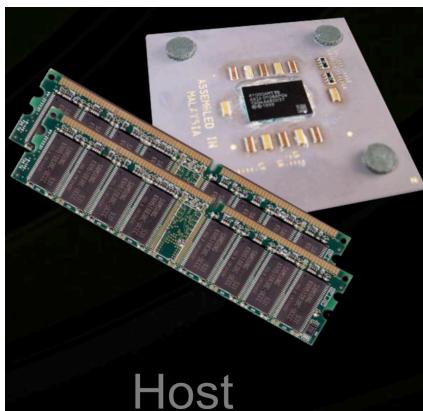
Device: = GPU si memoria asociata

- 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

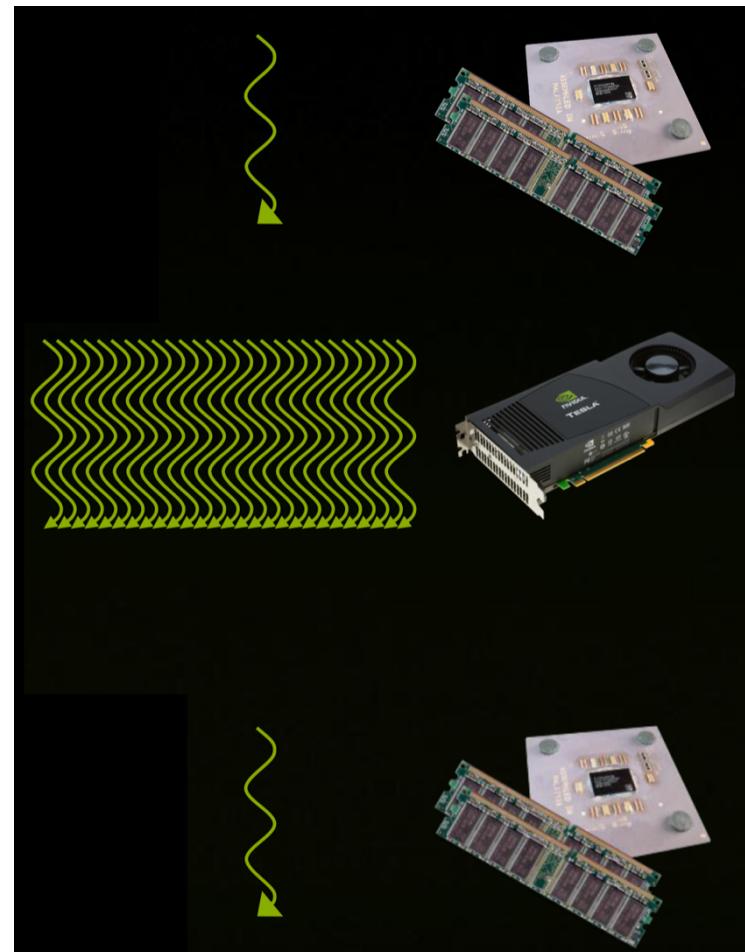
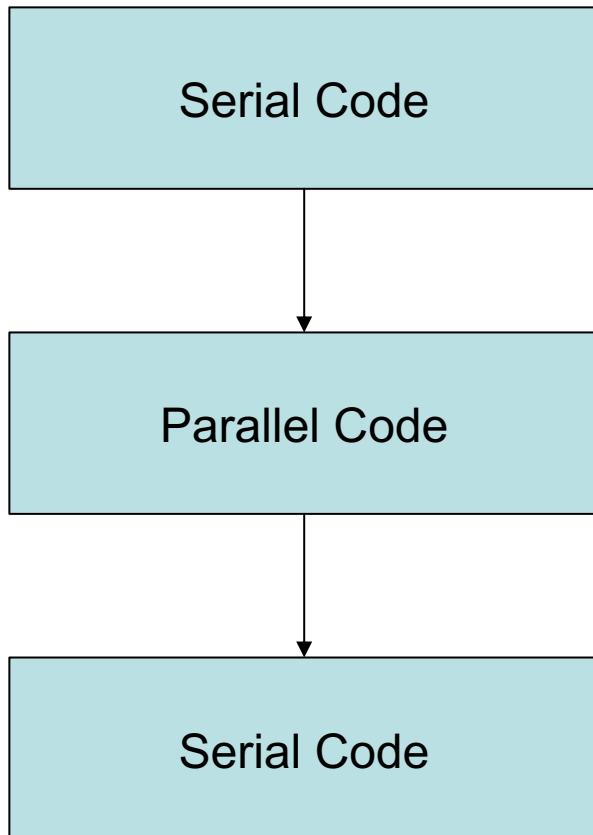


Host

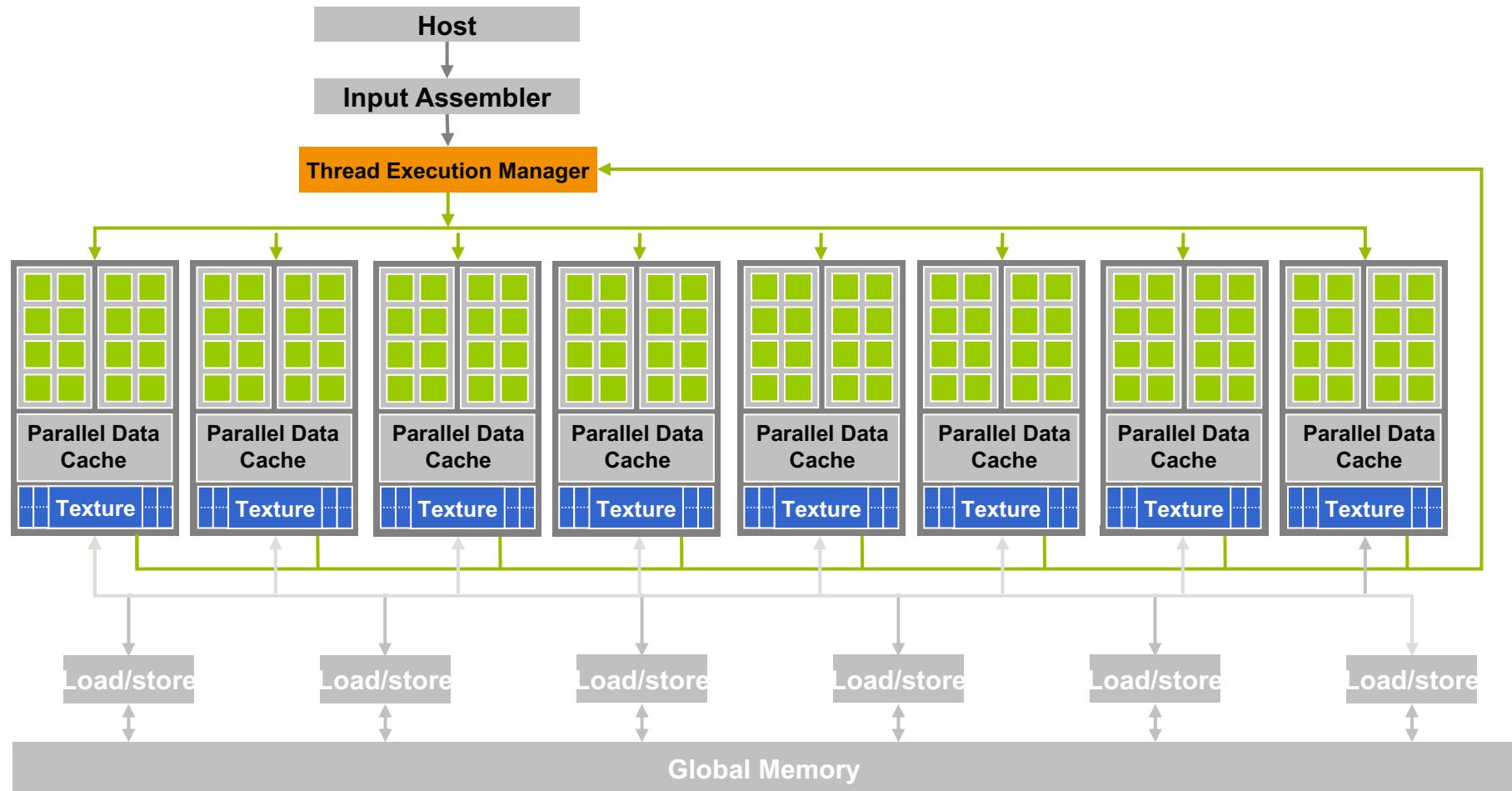


Device

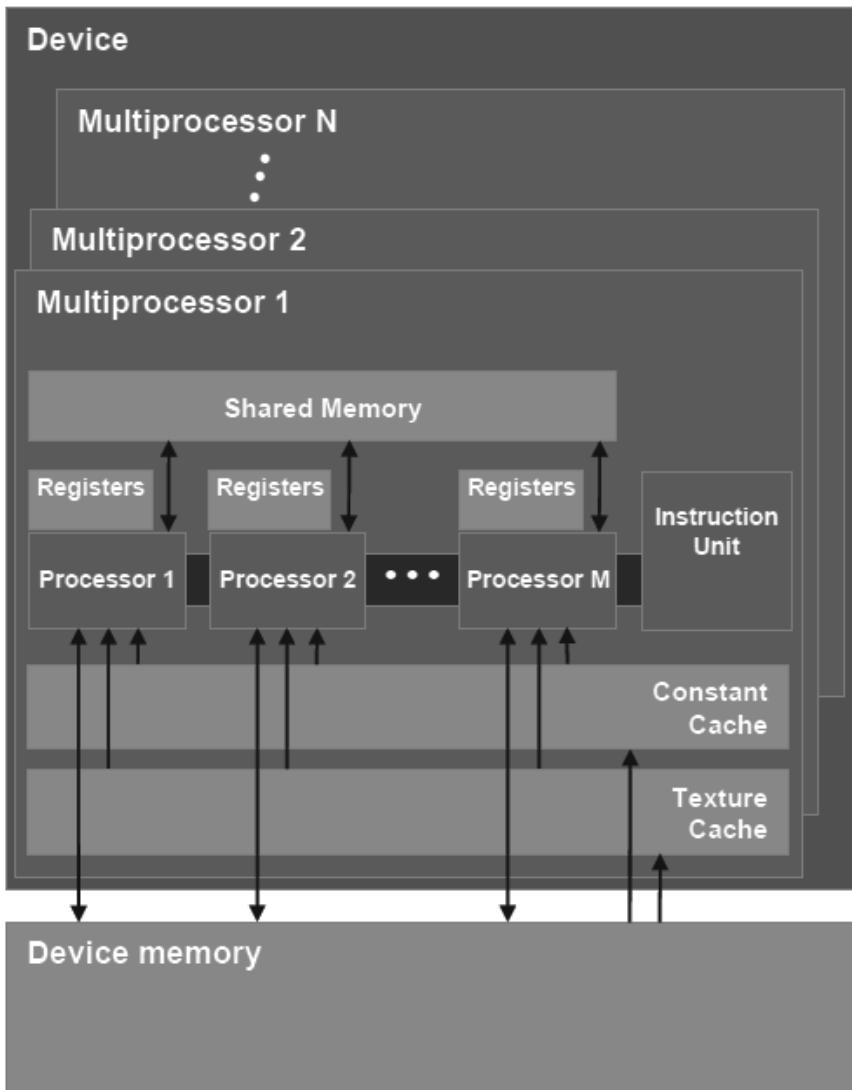
# Heterogeneous Computing



# Architecture of a CUDA-capable GPU



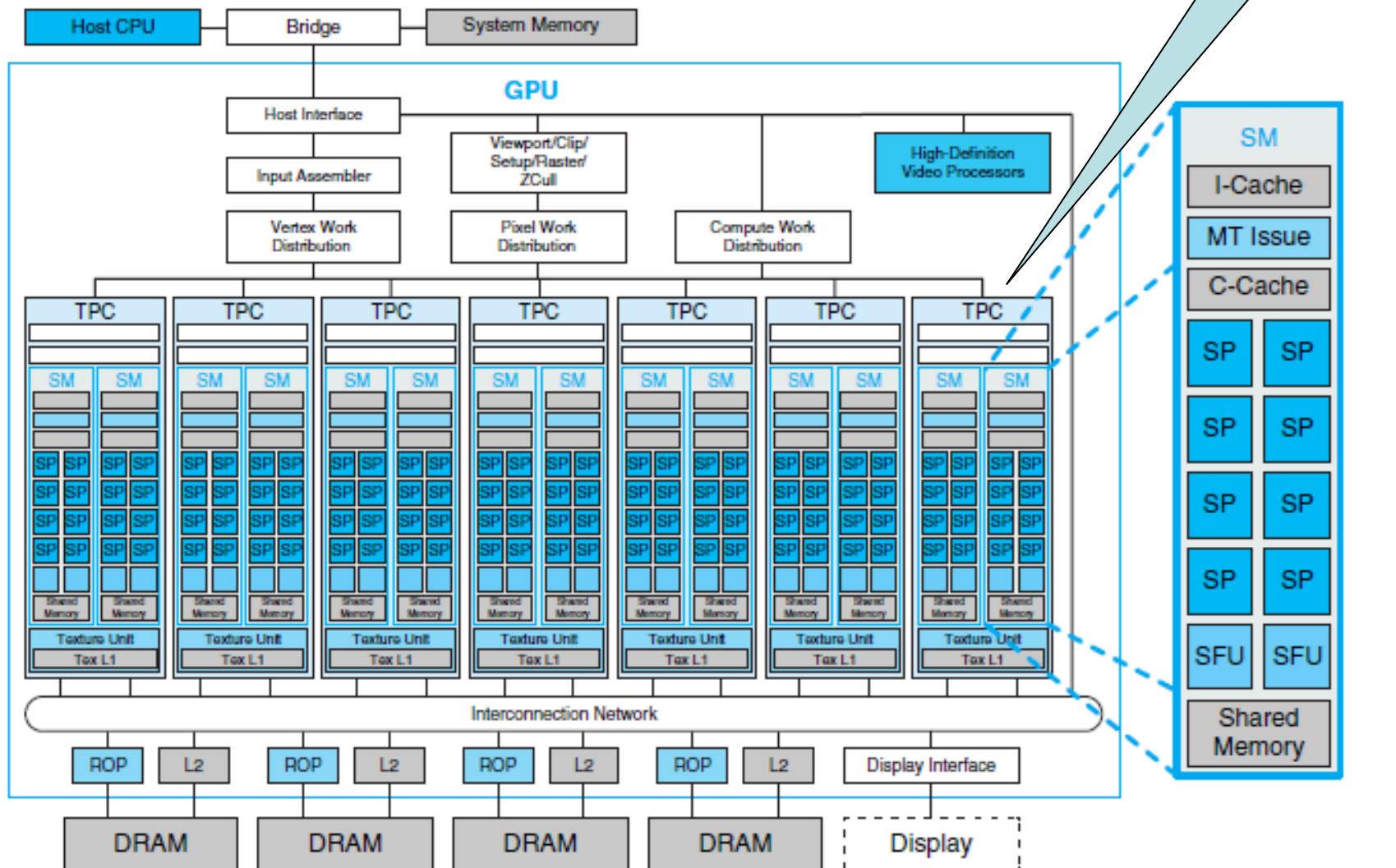
# GPU device

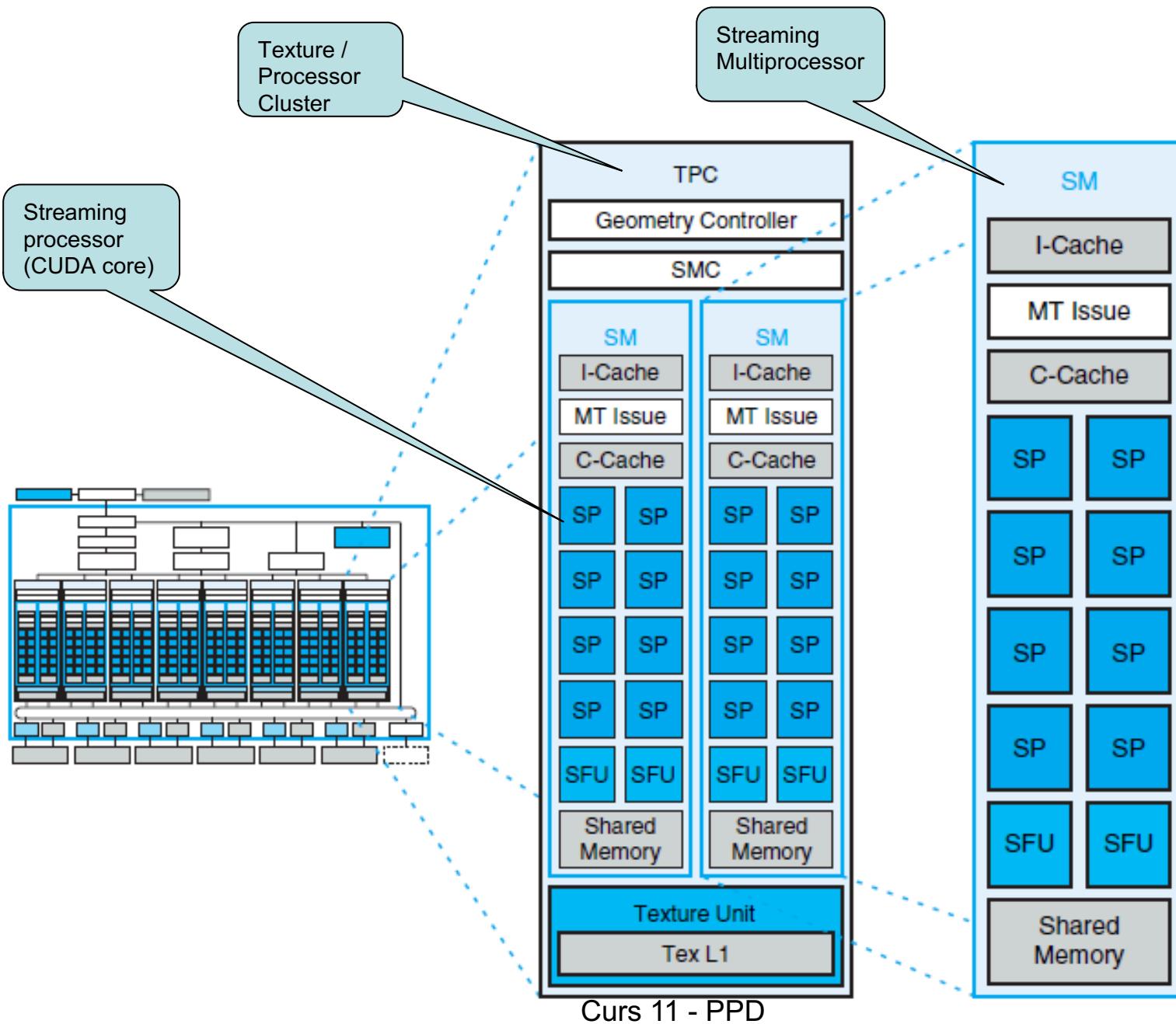


- Global memory
- Streaming Multiprocessors (SM) where each SM has:
  - Control units
  - Registers
  - Execution pipelines
  - Caches

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

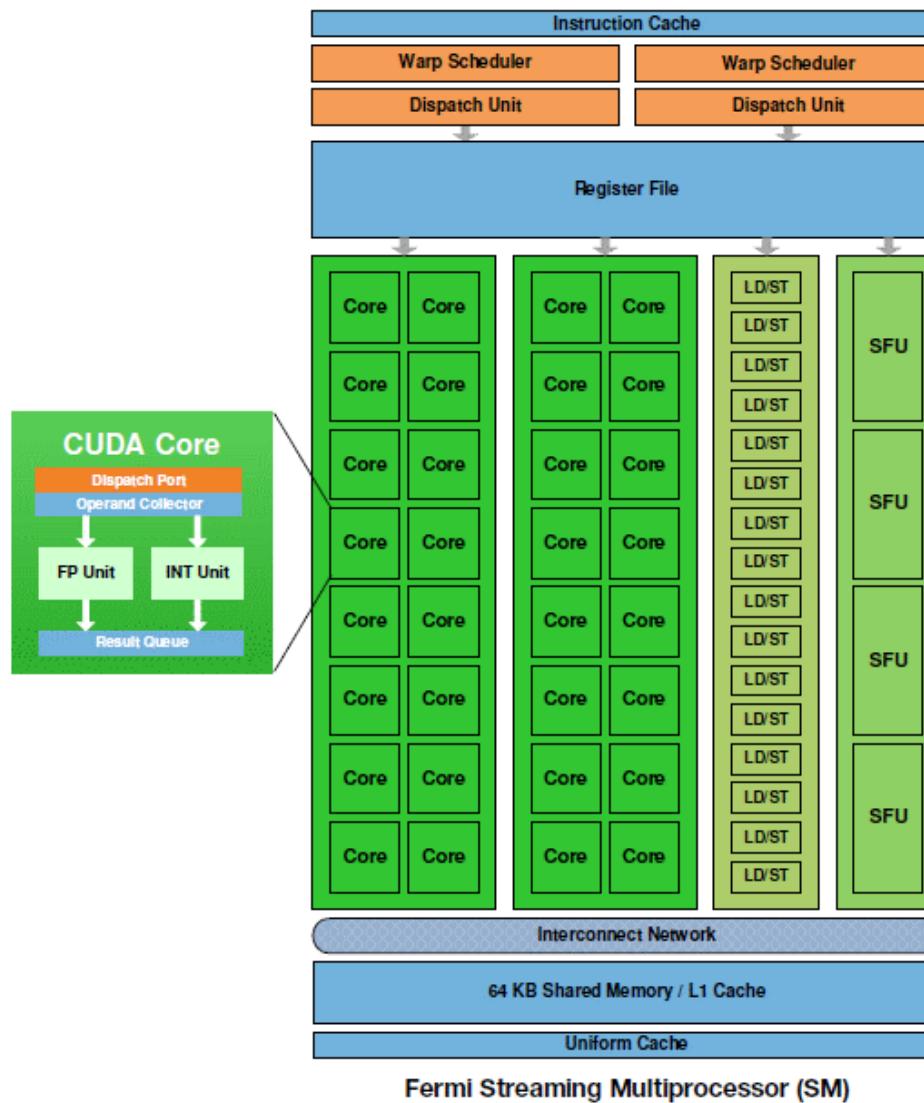
# GeForce 8800 Architecture





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

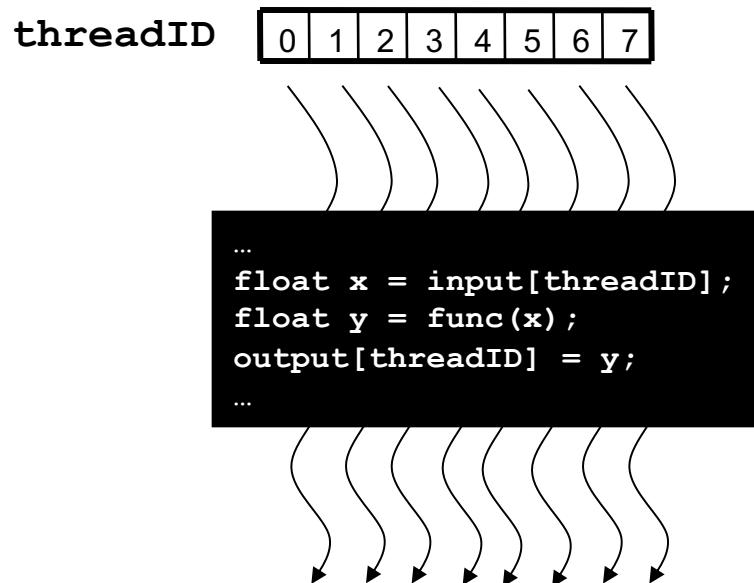
- `__global__` defines a kernel function
  - Must return `void`

# CUDA Function Declarations

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - *No recursion*
  - *No static variable declarations inside the function*
  - *No variable number of arguments*

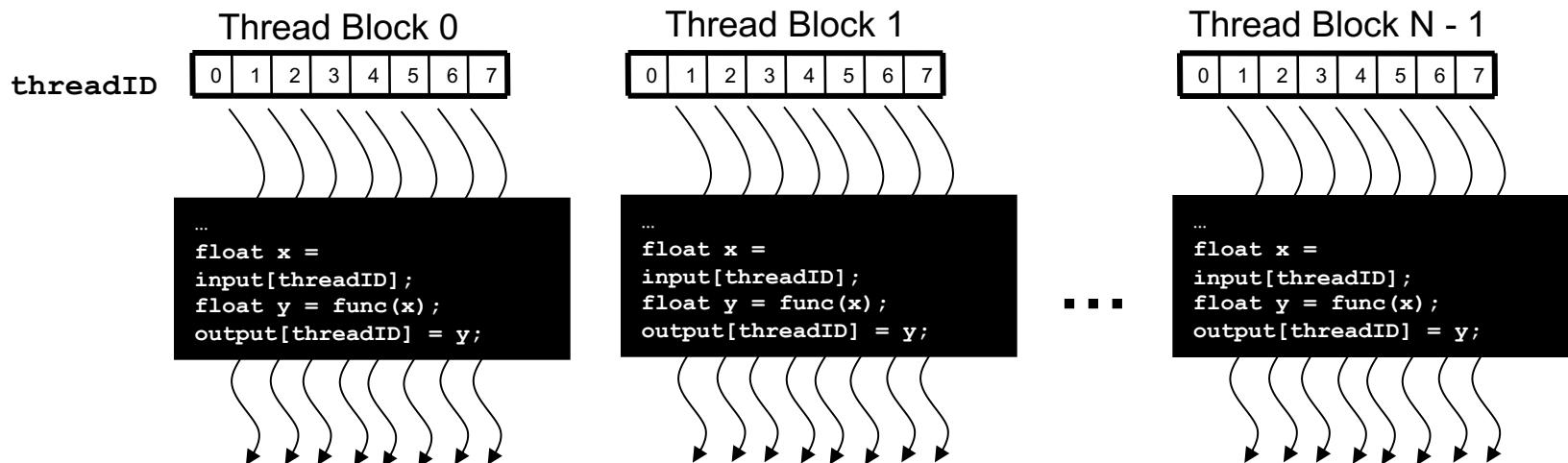
# Arrays of Parallel Threads

- 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

- Thread-urile dintr-un bloc pot coopera via **shared memory, atomic operations, synchronization barrier**
- Thread-urile din blocuri diferite nu pot coopera!



# 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 Serial : setup

```
int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 1<<20;
    // Host input vectors
    double *h_a;  double *h_b;
    //Host output vector
    double *h_c;
    // 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 Serial: 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);
// Release host memory
free(h_a);
free(h_b);
free(h_c);
return 0;
}
```

><><><><><><><><><

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

vectorAdd<<<3, 4>>>(d\_a, d\_b, d\_c);

↑  
Functia Kernel

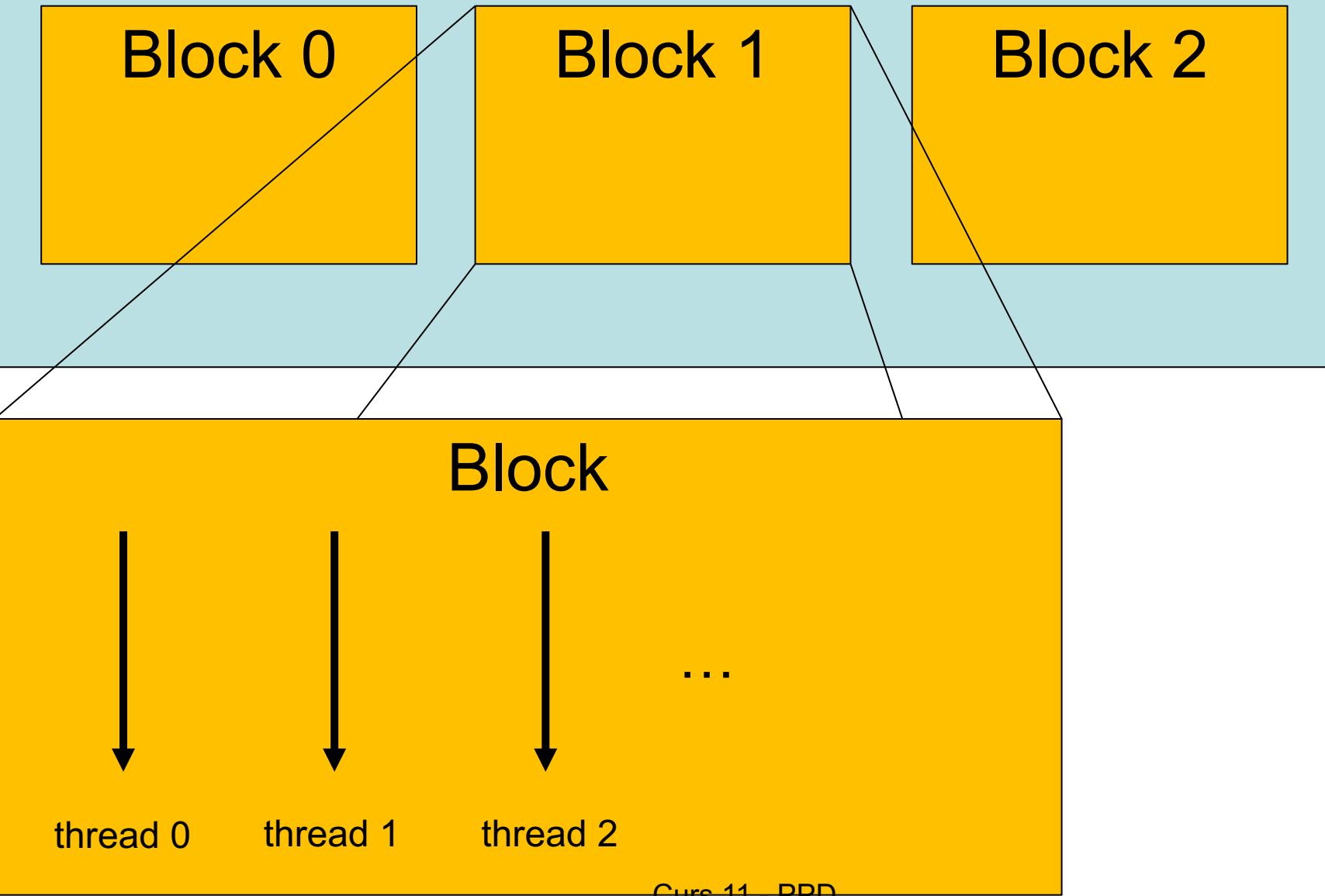
↑  
Cate blocuri si cate  
threaduri per bloc

↑  
parametrii

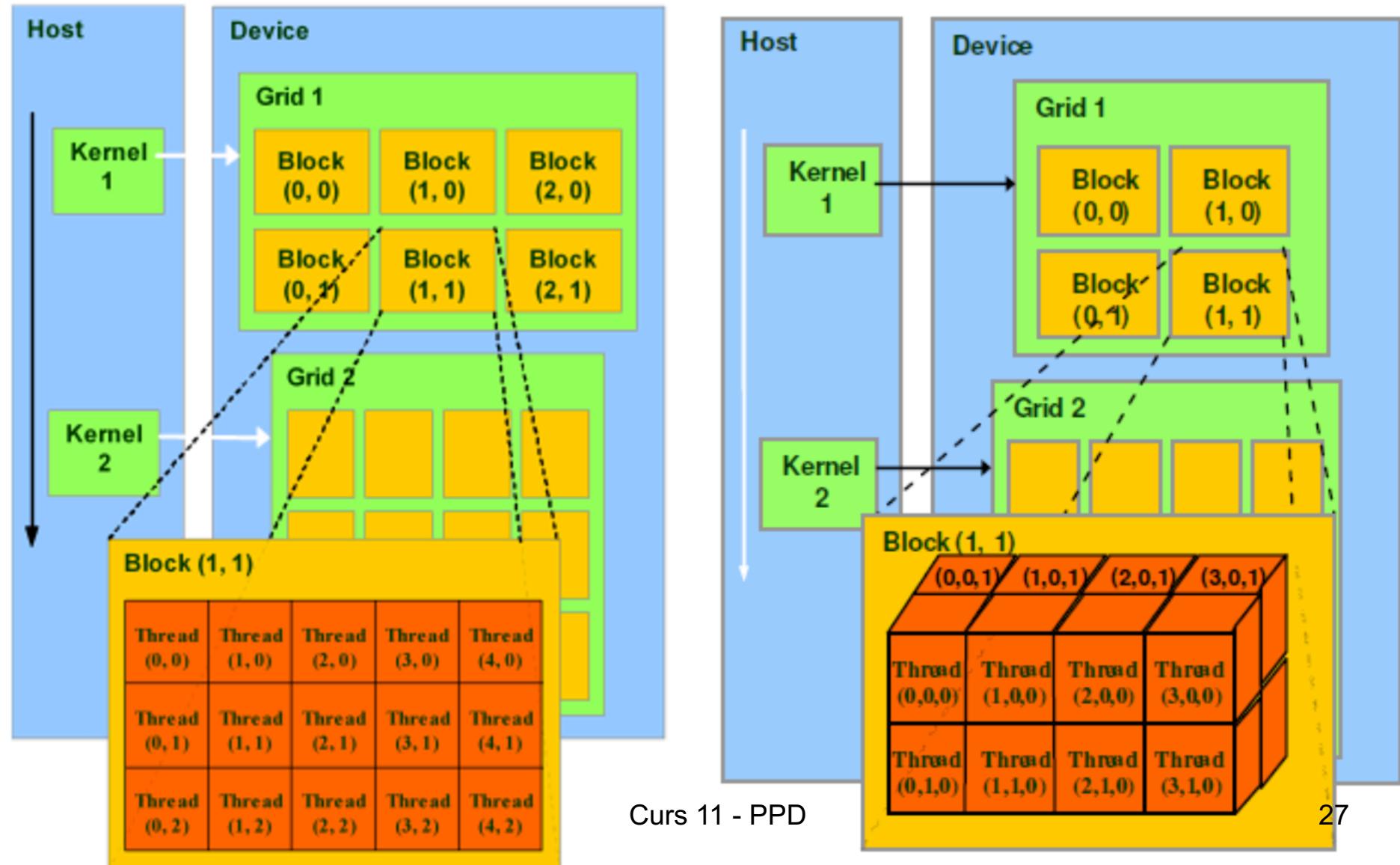
# Thread, Block, Grid

- 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

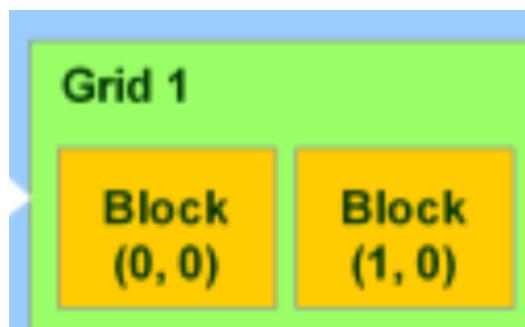
- `blockIdx.x`, `blockIdx.y`, `blockIdx.z` are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z` are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this 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)):  
 $\text{idx} = 0 + 1 * 4 = 4$



Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)

# Exemplul 2

Global Thread ID

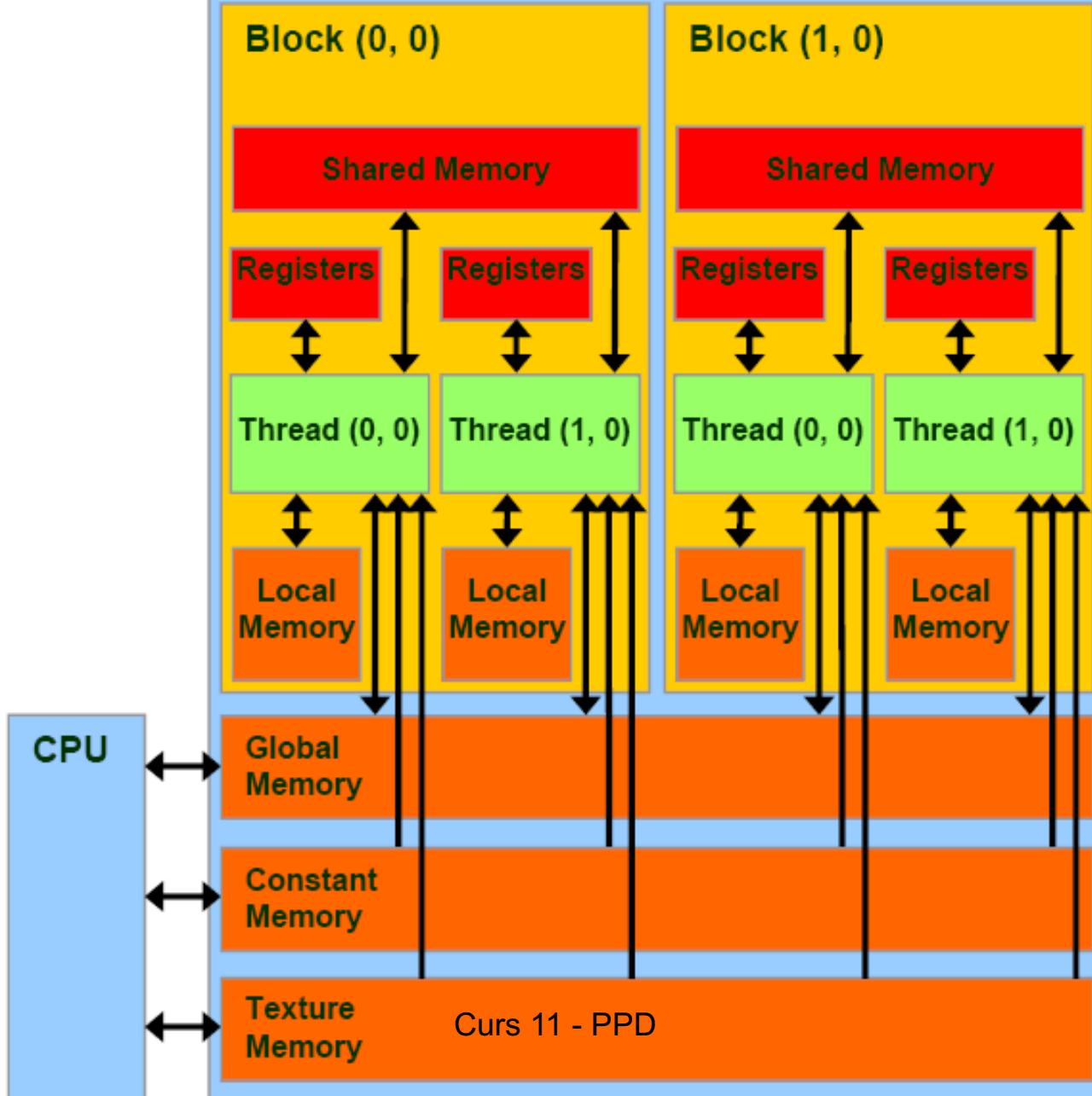
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x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

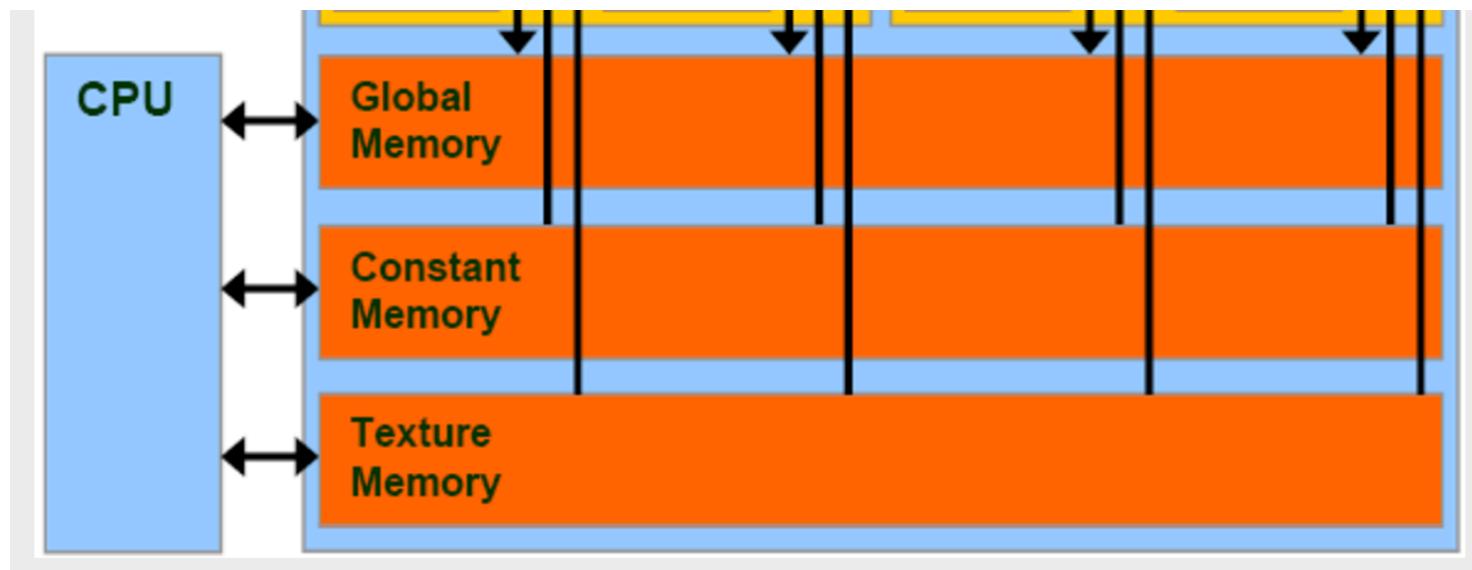
- CUDA foloseste 5 tipuri de memorie fiecare cu proprietati diferite
- Proprietati:
  - Size
  - Access speed
  - Read/write, read only

## GPU Grid



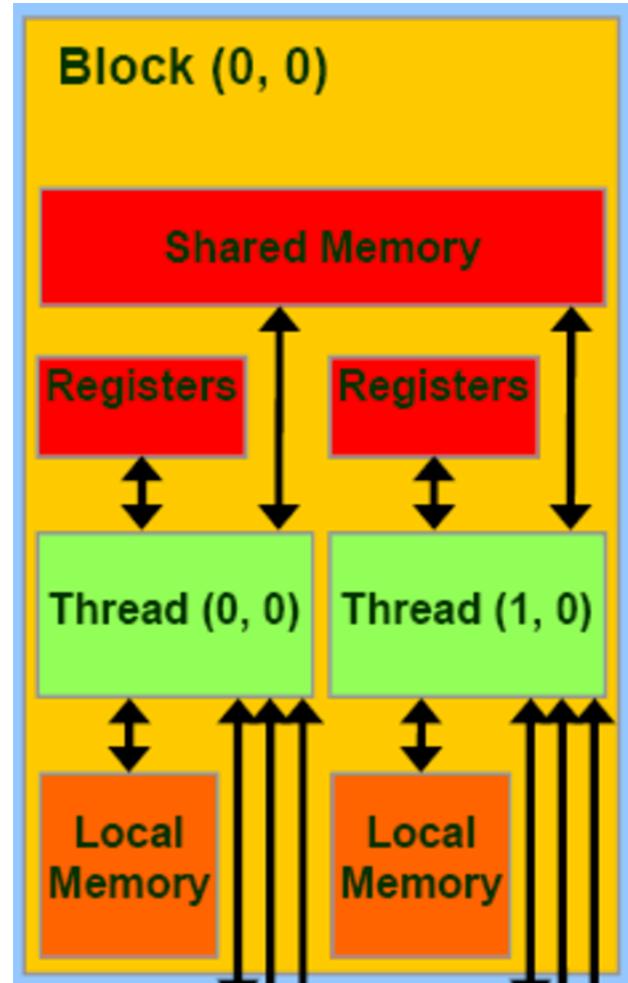
# Memory Types

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



# Memory Types

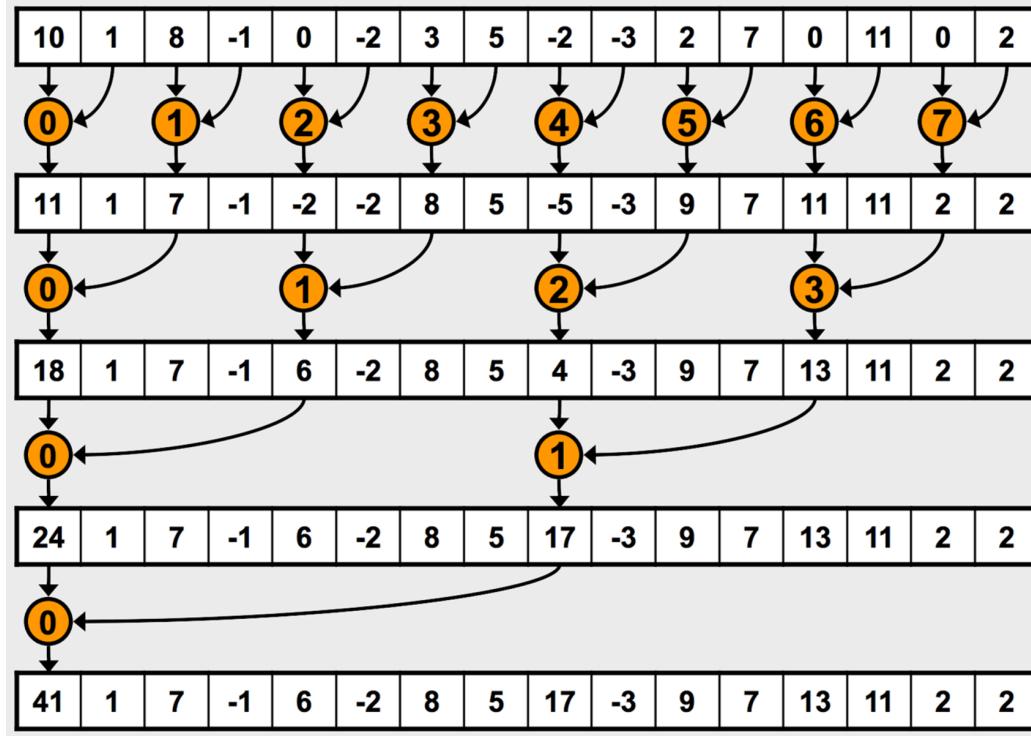
- Local memory:  
local to thread, but it is as slow as global memory
- Shared memory:  
100x fast to global  
memory, it is accessible  
to all threads in one  
block



# Memory Types

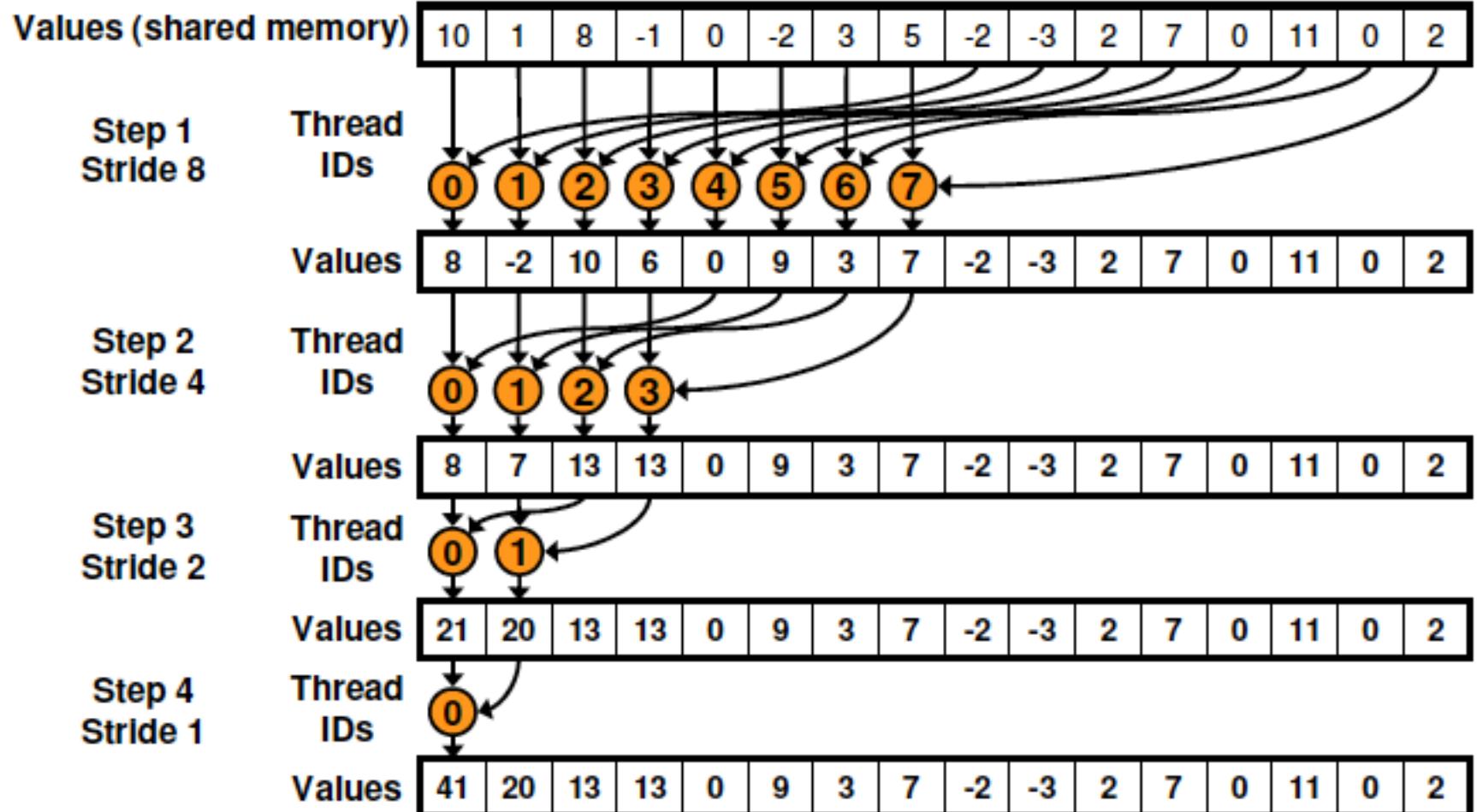
- Shared memory is very fast, but usually only has 49KB (can be configured to 64KB).
- Actually, shared memory is the same as “L1 cache” of CPU, but controllable by user.
- One block has one shared memory, that’s one reason why we manage the threads in grid and block way!

# 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
    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.

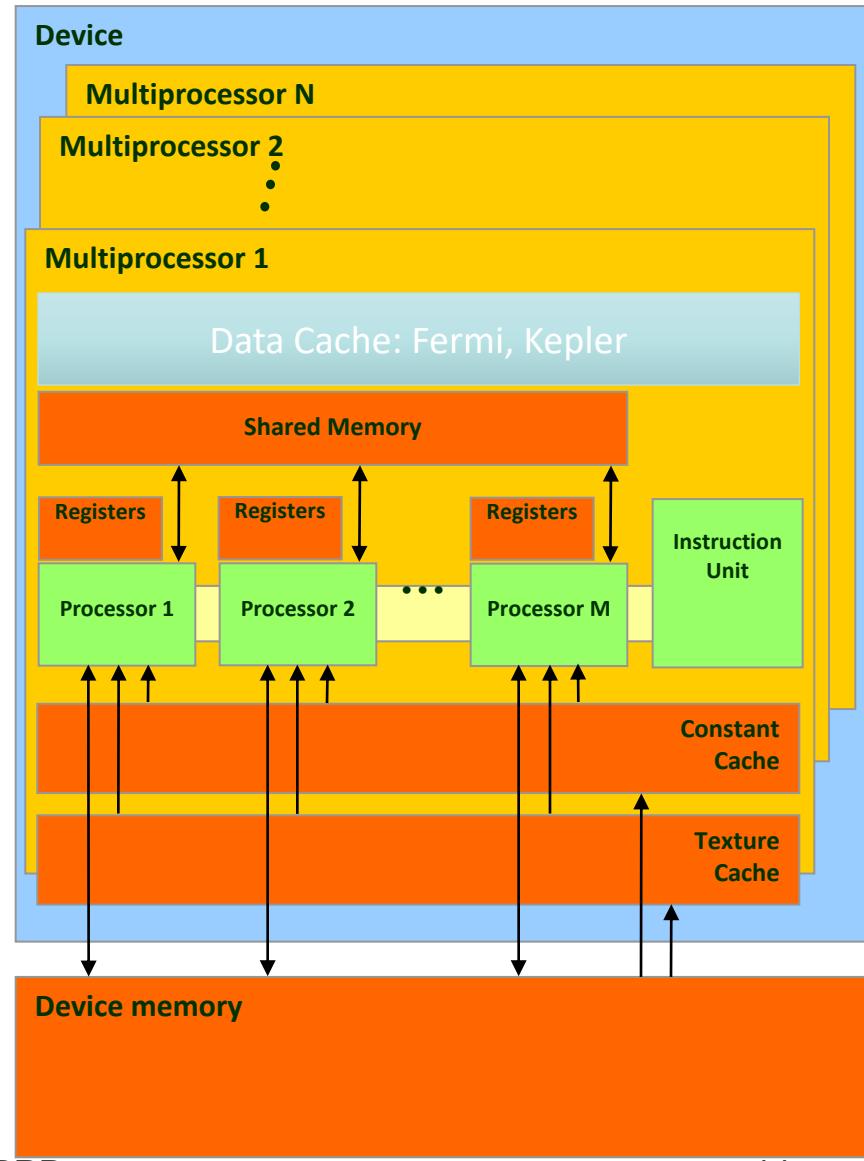
# NVIDIA GPU Execution Model

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

- 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)

- 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
  - Cate 32 threads compun un **wrap**.
    - Daca se alege un numar 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, threadurile au indecsi secventiali:
    - 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 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

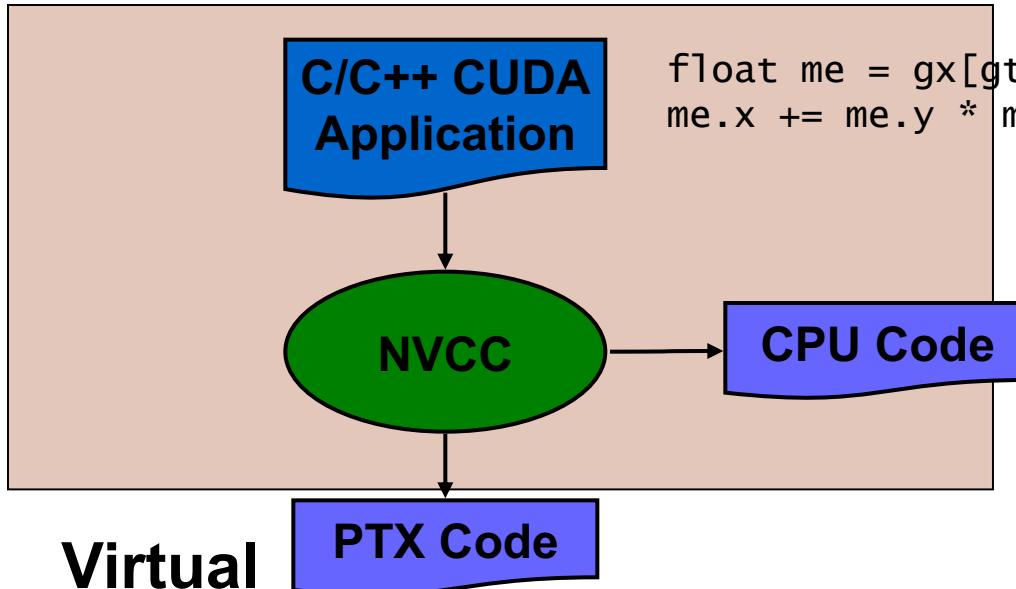
# Floating Point Operations

- 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

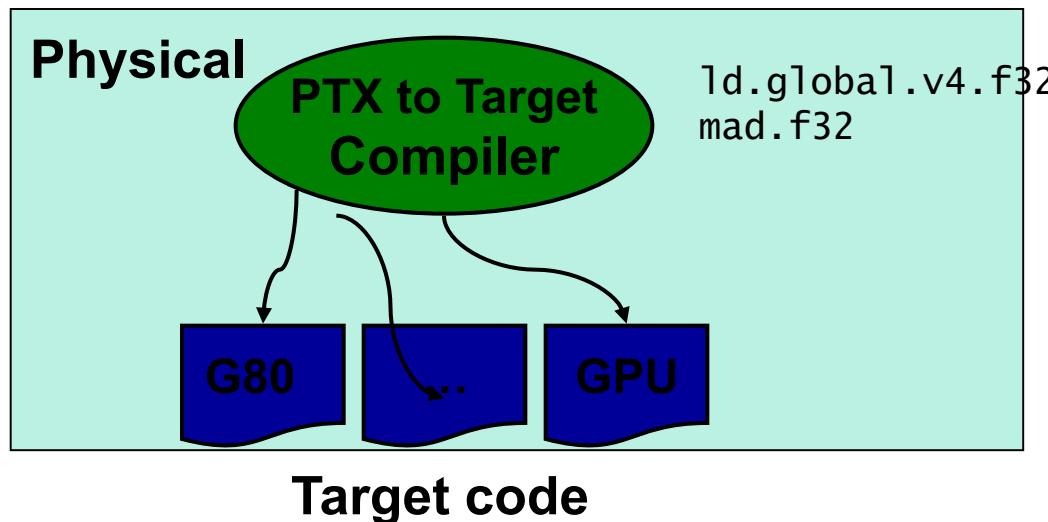
COMPILEARE

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

Metode de evaluare a performantei programelor paralele

Granularitate

Scalabilitate

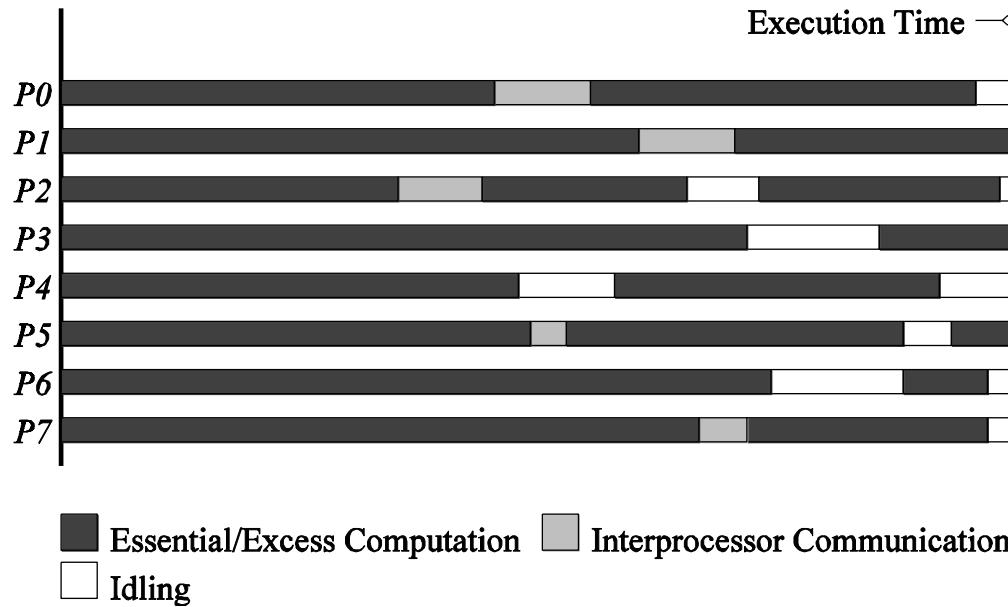
# Complexitate – consideratii generale

- Daca in cazul algoritmilor sesequentiali 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

- În calculul paralel, obținerea unui timp de execuție mai bun nu înseamnă neapărat utilizarea unui număr minim de operații, așa cum este în calculul serial.
- Factorul memorie nu are o importanță atât de mare în calculul paralel (relativ).
- În schimb, o resursă majoră în obținerea unei performante bune a algoritmului paralel o reprezintă numărul de procesoare folosite.
- Dacă timpul de execuție a unei operații aritmetice este mult mai mare decât timpul de transfer al datelor între două elemente de procesare, atunci întârzierea datorată retelei este nesemnificativă, dar, în caz contrar, timpul de transfer joacă un rol important în determinarea performanței 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:

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

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

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

# 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.

# 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;

par = fractia calcului paralelizabil;

Se consideră **calculul serial**  $T_s = 1$  unitate

Speedup =  $1/(seq + par/p)$

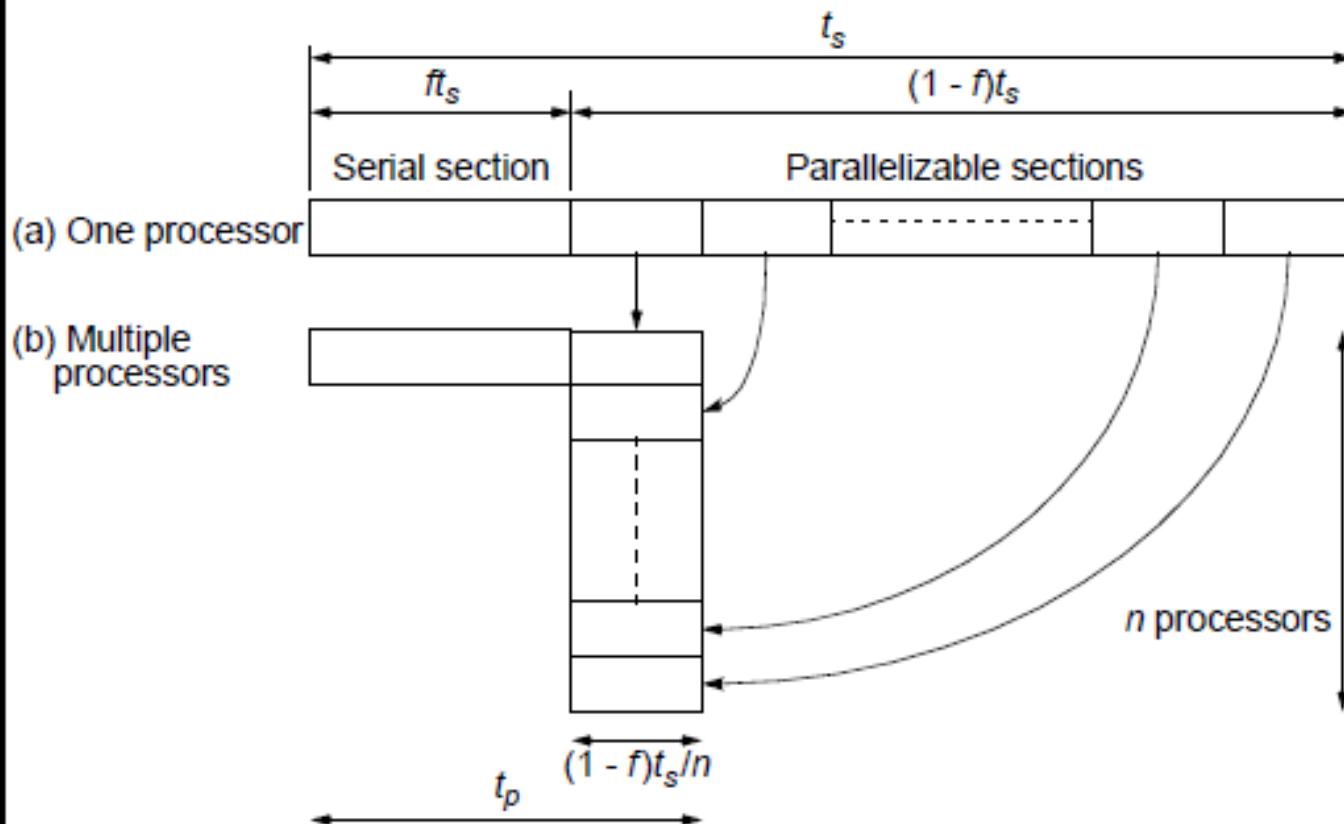
par =  $(1 - seq)$ , p = # procesoare

- $p \rightarrow \text{infinit} \Rightarrow S \sim 1/\text{seq.}$
- Limita superioară a accelerării este data de fractia partii secentiale.
- *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

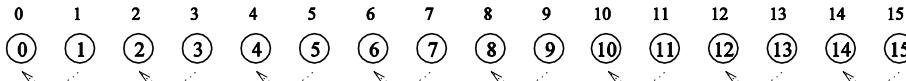
## Legea lui Amdahl - pesimista

- 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

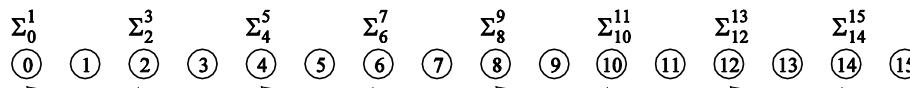
# Exemple

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

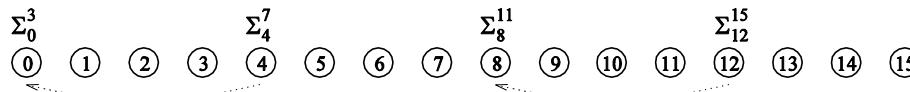
# Adunare – log n pasi



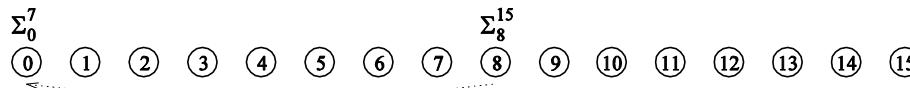
(a) Initial data distribution and the first communication step



(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

$n=16$ ;  $p=16$ s . .

## 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  $T_{com} = \Theta(\log n)$

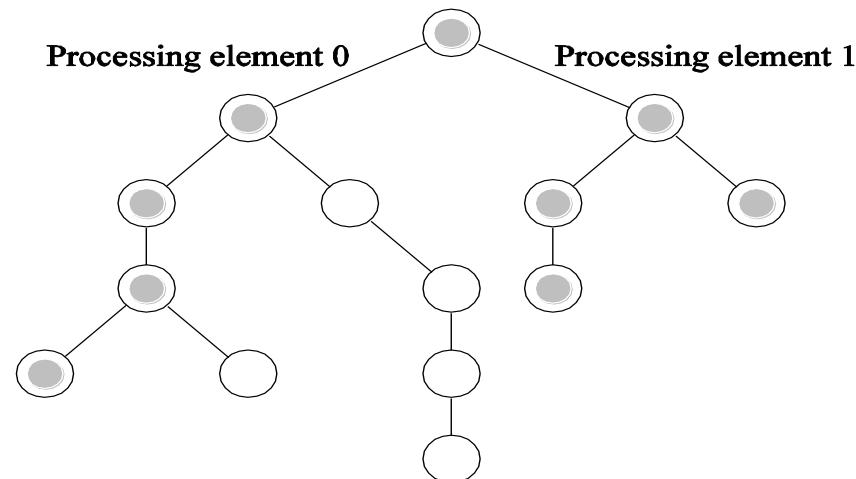
$$T_p = \Theta(\log n)$$

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

# Accelerare – superliniara?

- $S = 0$  (the parallel program never terminates).
- $S < p$  (teoretic)
  - În caz contrar un procesor ar fi implicat în calcule pentru rezolvarea problemei un timp  $T < T_s / p$  *în condițiile în care  $T_s$  depinde de numărul de operații care se efectuează => se efectuează mai puține operații în total*

# Superlinear Speedups



Cautare intr-un arbore nestructurat.

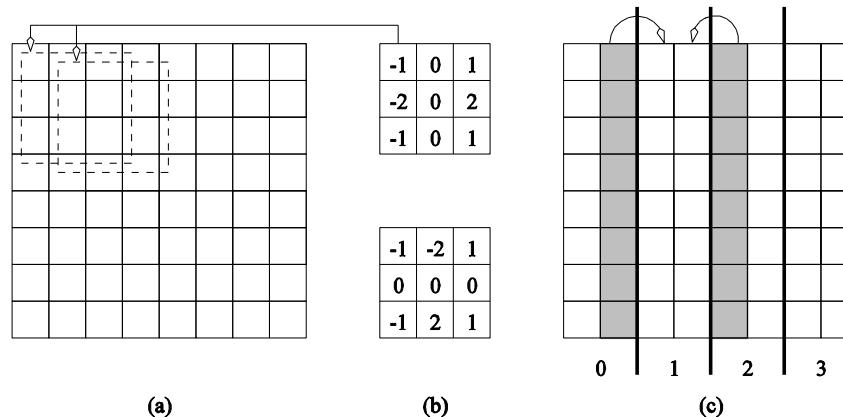
# Exemplu

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

*calculul executat de fiecare process:*

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

# 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;
- aplicatia este **eficienta** din punct de vedere al costului daca  $C_p = O(t_1 \log p)$ .

# 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 rezolvarii 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

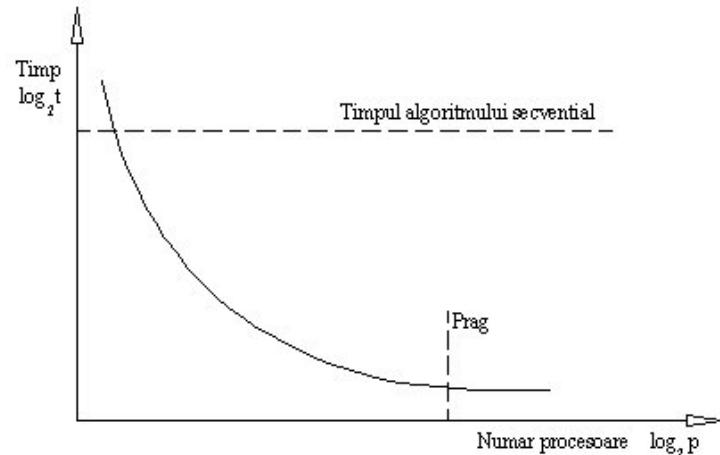
# Scalabilitate

- *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 dubleaza 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 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 a change in the efficiency.

# Exemple

- Suma de 2 vectori = 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?

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

- $p_{\maxim} = n \Rightarrow (2t_w / 9t_c) \sim 1$

# Granularitate

- **Granularitatea** (“*grain size*”) este un parametru calitativ care caracterizeaza atat **sistemele paralele** 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**.

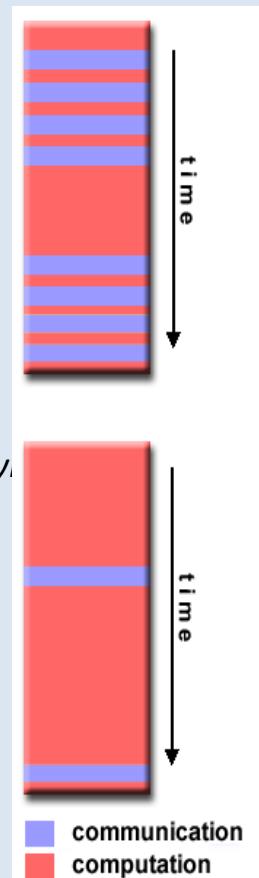
# Granularitatea sistemului

- 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.

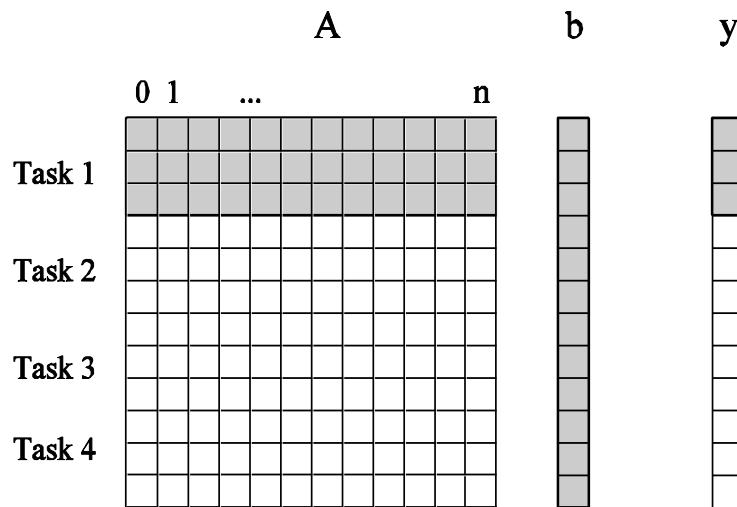


- **Coarse-grain Parallelism:**

- 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

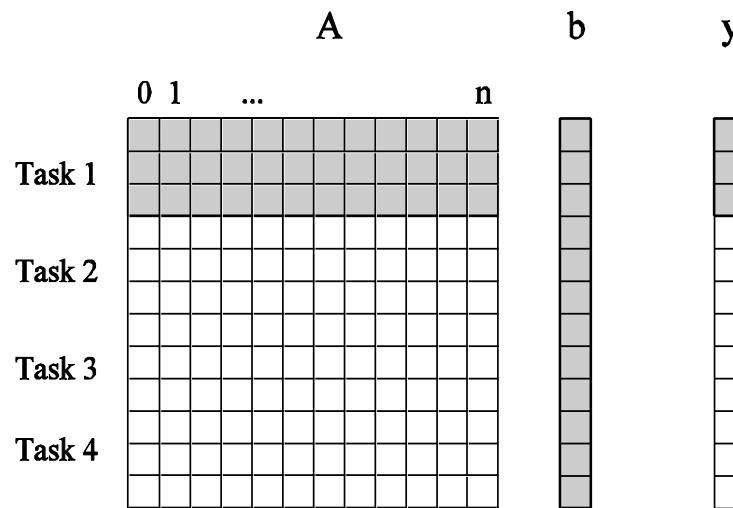
- Numarul de task-uri in care o problema se descompune determina granularitatea.
- numar mare  $\Rightarrow$  fine-grained decomposition
- numar mic  $\Rightarrow$  coarse grained decomposition



Exemplu: inmultire matrice

# Granularitatea decompunerii taskurilor

- Granularitatea este **determinata** 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 (DOP)

- 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

# *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 13

Distributed Computing Patterns

# Distributed systems

- We may define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only bypassing messages .
- Distributed-Computing is the process of solving a problem using a distributed system.

# Characteristics

## Concurrency:

- coordination of concurrently executing programs that share resources is also an important and recurring topic.

## No global clock:

- there is no single global notion of the correct time.
- This is a direct consequence of the fact that the only communication is by sending messages through a network.

## Independent failures:

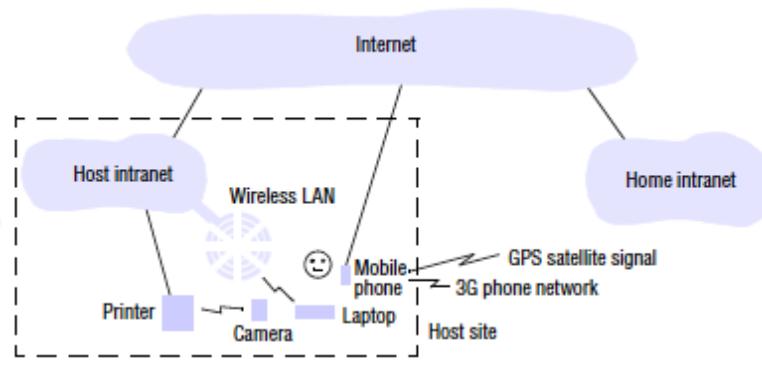
- All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures.
- Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running.
- In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow.
- Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a crash), is not immediately made known to the other components with which it communicates.
- Each component of the system can fail independently, leaving the others still running.

# Trends

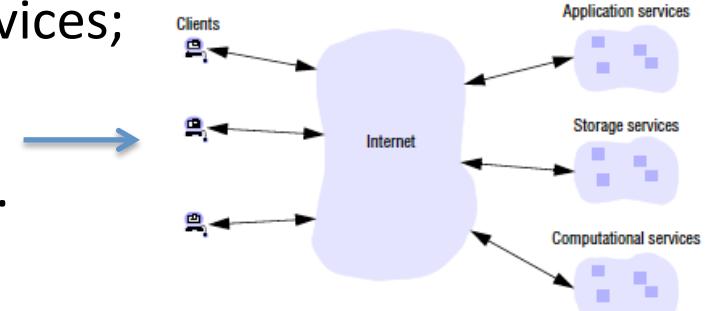
Distributed systems are undergoing a period of significant change and this can be traced back to a number of influential trends:

- the emergence of pervasive networking technology;

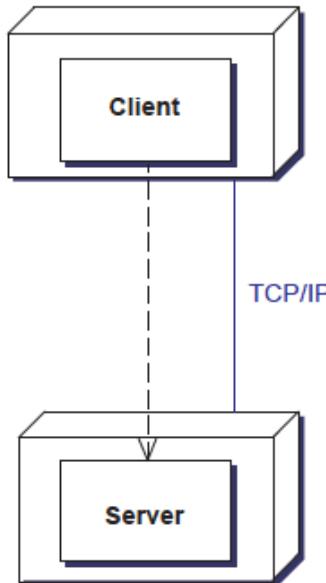
- the emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;



- the increasing demand for multimedia services;
- the view of distributed systems as a utility.



# Client-Server



- A server component provides services to multiple client components.
- A client component requests services from the server component.
- Servers are permanently active, listening for clients.

# State in the Client-server pattern

Clients and servers are often involved in ‘sessions’.

- With a **stateless server**, the session state is managed by the client. This client state is sent with each request. In a web application, the session state may be stored as URL parameters, in hidden form fields, or by using cookies. This is mandatory for the REST architectural style for web applications.
- With a **stateful server**, the session state is maintained by the server, and is associated with a client-id.

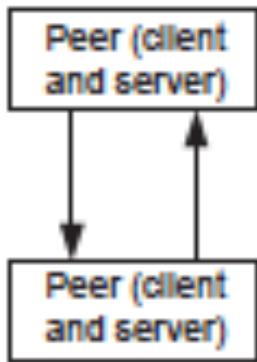
State in the Client-server pattern influences transactions, fault handling and scalability.

- Transactions should be atomic, leave a consistent state, be isolated (not affected by other requests) and durable. These properties are hard to obtain in a distributed world.
- Concerning fault handling, state maintained by the client means for instance that everything will be lost when the client fails.
- Client-maintained state poses security issues as well, because sensitive data must be sent to the server with each request.
- Scalability issues may arise when you handle the server state in-memory: with many clients using the server at the same time, many states have to be stored in memory at the same time as well.

# Peer-to-peer pattern

- it can be seen as a symmetric Client-server pattern:
  - peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers.
- A peer may act as a client or as a server or as both, and it may change its role dynamically.
- Both clients and servers in the peer-to-peer pattern are typically multithreaded. The services may be implicit (for instance through the use of a connecting stream) instead of requested by invocation.
- Peers acting as a server may inform peers acting as a client of certain events. Multiple clients may have to be informed, for instance using an event-bus.

# Examples



- the distributed search engine Scienonet,
- multi-user applications like a drawing board,
- peer-to-peer file-sharing like Gnutella or BitTorrent.

# Characteristics

- An advantage of the peer-to-peer pattern is that nodes may use the capacity of the whole, while bringing in only their own capacity.
  - In other words, there is a low cost of ownership, through sharing.
- Administrative overhead is low, because peer-to-peer networks are self-organizing.
- The Peer-to-peer pattern is scalable, and resilient to failure of individual peers.
- The configuration of a system may change dynamically: peers may come and go while the system is running.
- A disadvantage may be that there is no guarantee about quality of service, as nodes cooperate voluntarily.
  - For the same reason, security is difficult to guarantee.
- Performance grows when the number of participating nodes grows, which also means that it may be low when there are few nodes.

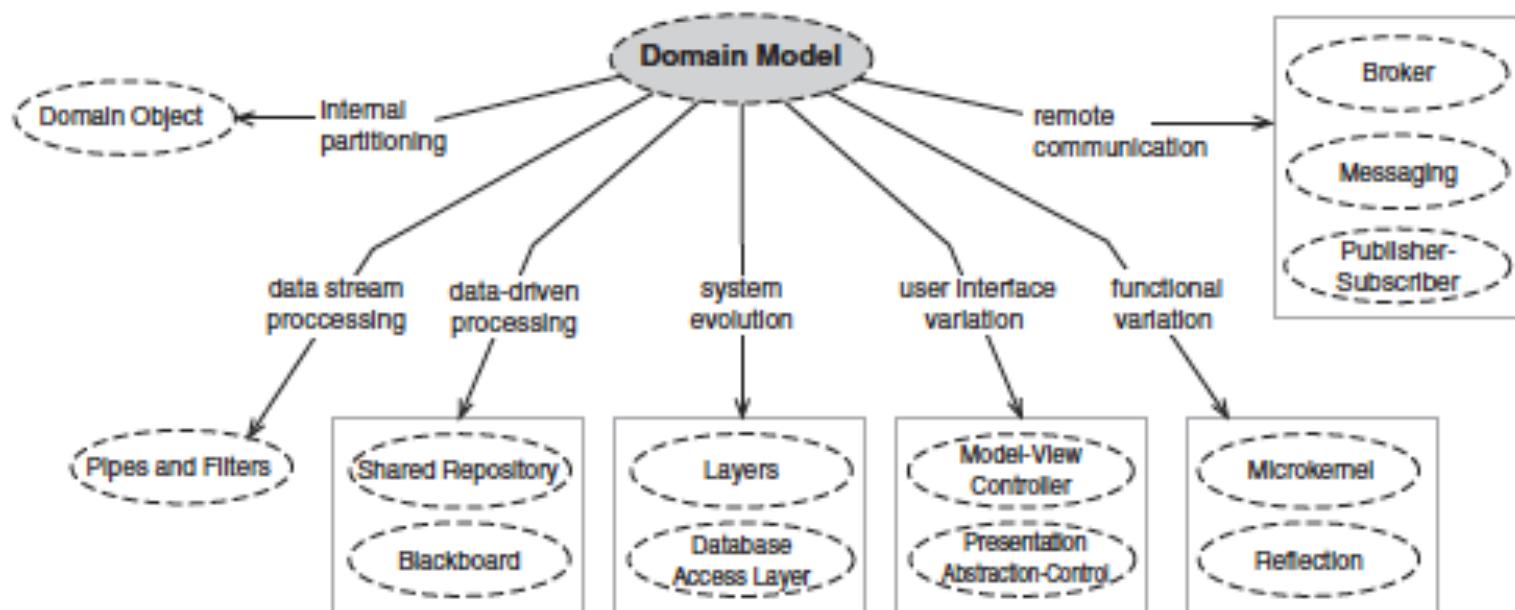
# Architectural patterns

Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. **Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing**. Wiley & Sons, 2007

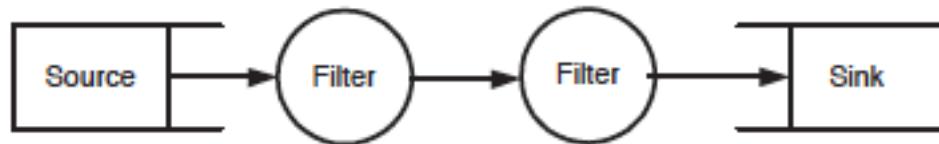
An architectural pattern is a concept that solves and delineates some essential cohesive elements of a software architecture.

- Domain Model
- Layers
- Model-View-Controller
- Presentation-Abstraction-Control
- Microkernel
- Reflection
- Pipes and Filters
- Shared Repository
- Blackboard
- Domain Object

# Connection to the Domain Layer

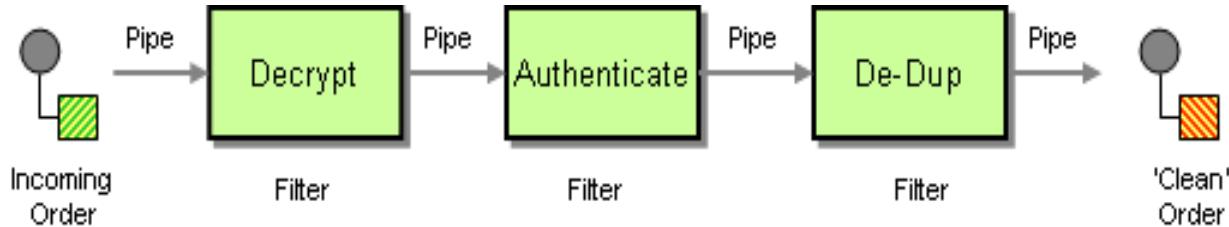


# Pipe-Filter Pattern



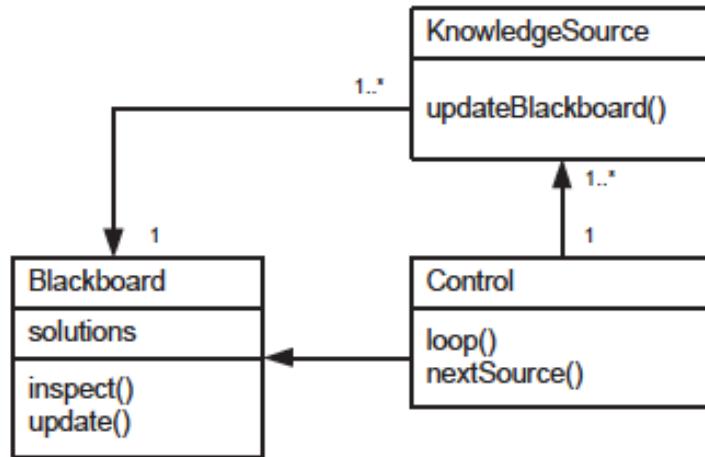
- The Pipe-filter architectural pattern provides a structure for systems that produce a stream of data.
- Each processing step is encapsulated in a filter component (circles).
- Data is passed through pipes (the arrows between adjacent filters).
  - The pipes may be used for buffering or for synchronization.

# Example



- Use the Pipes and Filters architectural style to divide a larger processing task into a sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes).
- Each filter exposes a very simple interface: it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe.
- The pipe connects one filter to the next, sending output messages from one filter to the next.
- Because all component use the same external interface they can be composed into different solutions by connecting the components to different pipes.
- We can add new filters, omit existing ones or rearrange them into a new sequence -- all without having to change the filters themselves.

# Blackboard Pattern



- The Blackboard pattern is useful for problems for which no deterministic solution strategies are known => opportunistic problem solving.
- Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- ***All components have access to a shared data store, the blackboard.***
- Components may produce new data objects that are added to the blackboard.
- Components look for particular kinds of data on the blackboard, and may find these by pattern matching.

- 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

# General description

- Blackboard allows multiple processes (or agents) to communicate by reading and writing requests and information to a global data store.
- Each participant agent has expertise in its own field, and has a kind of problem solving knowledge (knowledge source) that is applicable to a part of the problem, i.e., the problem cannot be solved by an individual agent only.
- Agents communicate strictly through a common blackboard whose contents is visible to all agents.
- When a partial problem is to be solved, candidate agents that can possibly handle it are listed.
- A control unit is responsible for selecting among the candidate agents to assign the task to one of them.

# Example: speech recognition

- The main loop of Control started
  - 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

# Advantages & Disadvantages

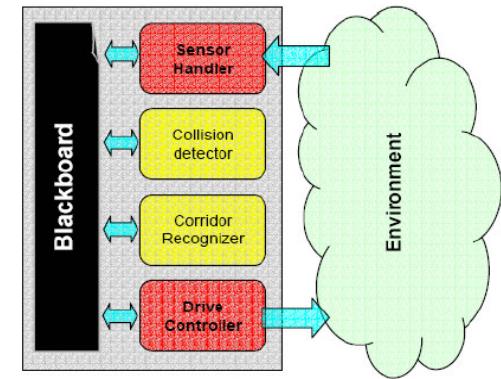
- Advantages
  - Suitable when there are diverse sources of input data
  - Suitable for physically distributed environments
  - Suitable for scheduling and postponement of tasks and decisions
  - Suitable for team problem-solving approaches as it can be used to post problem subsomponents and partial results
- Disadvantages
  - Expensive
  - Difficult to determine partitioning of knowledge
  - Control unit can be very complex

# 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.



# ....patterns related to the communication infrastructure

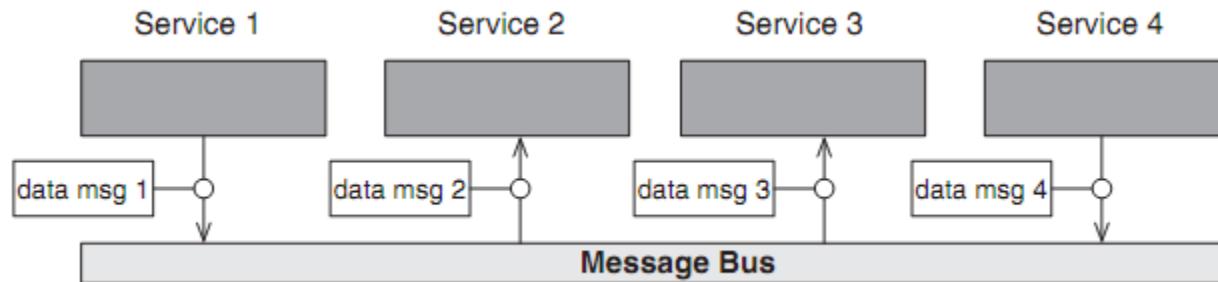
- **Messaging**
  - Distribution Infrastructure
- **Publisher-Subscriber**
  - Distribution Infrastructure
- **Broker**
  - Distribution Infrastructure
- **Client Proxy**
  - Distribution Infrastructure
- **Reactor**
  - Event Demultiplexing and Dispatching
- **Proactor**
  - Event Demultiplexing and Dispatching

# Messaging Pattern

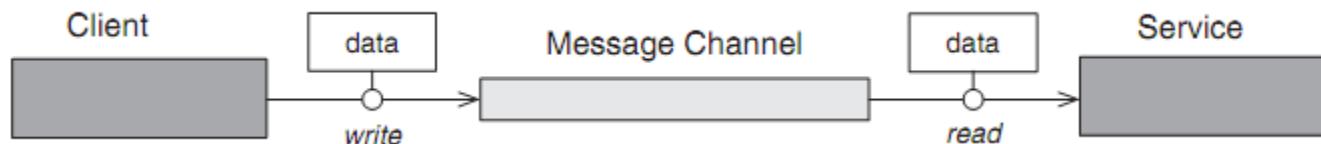
- Some distributed systems are composed of services that were developed independently.
- However, to form a coherent system, these services must interact reliably, but without incurring overly tight dependencies on one another.
- **Solution:** *Connect the services via a message bus* that allows them to transfer data messages **asynchronously**.
  - Encode the messages so that senders and receivers can communicate reliably without having to know all the data type information statically.

# Messaging (continued)

- Message-based communication supports *loose coupling* between services in a distributed system.



- *Messages* only contain the data to be exchanged between a set of clients and services, so they do not know who is interested in them.
  - Therefore, another way is to connect the collaborating clients and services using a message channel that allows them to exchange messages, known as “Message Channel”.

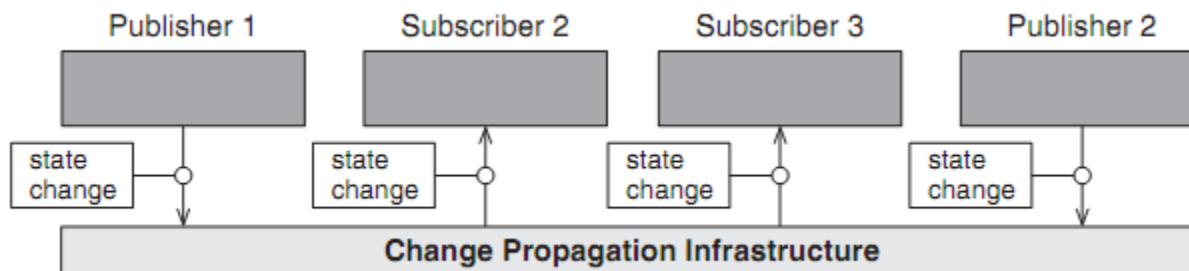


# Publisher-Subscriber Pattern

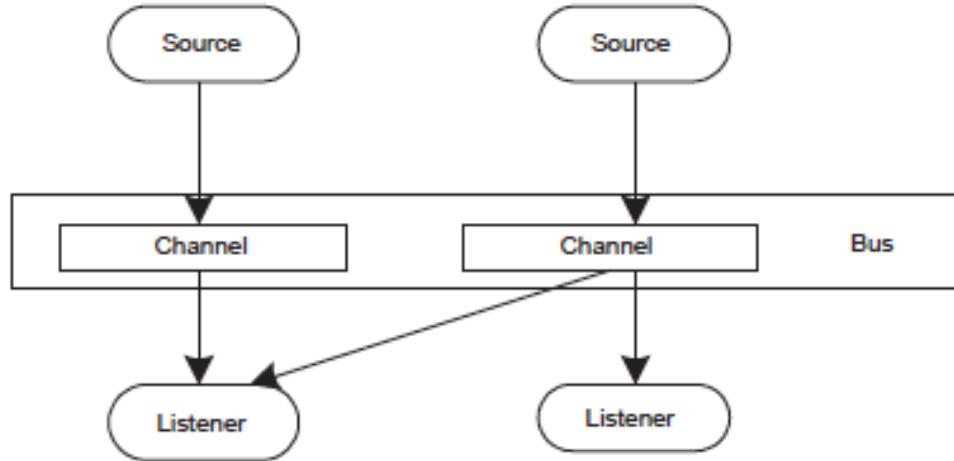
- Components in some distributed applications are loosely coupled and operate largely independently.
- if such applications need to propagate information to some or all of their components, a notification mechanism is needed to inform the components about state changes or other interesting events.
- **Solution:** Define a change propagation infrastructure that allows publishers in a distributed application to ***disseminate events*** that convey information that may be of interest to others.
  - Notify subscribers interested in those events whenever such information is published.

# Publisher-Subscriber (continued)

- Publishers register with the change propagation infrastructure to inform it about what types of events they can publish.
- Similarly, subscribers register with the infrastructure to inform it about what types of events they want to receive.
- The infrastructure uses this registration information to route events from their publishers through the network to interested subscribers.



# Event-Bus Pattern

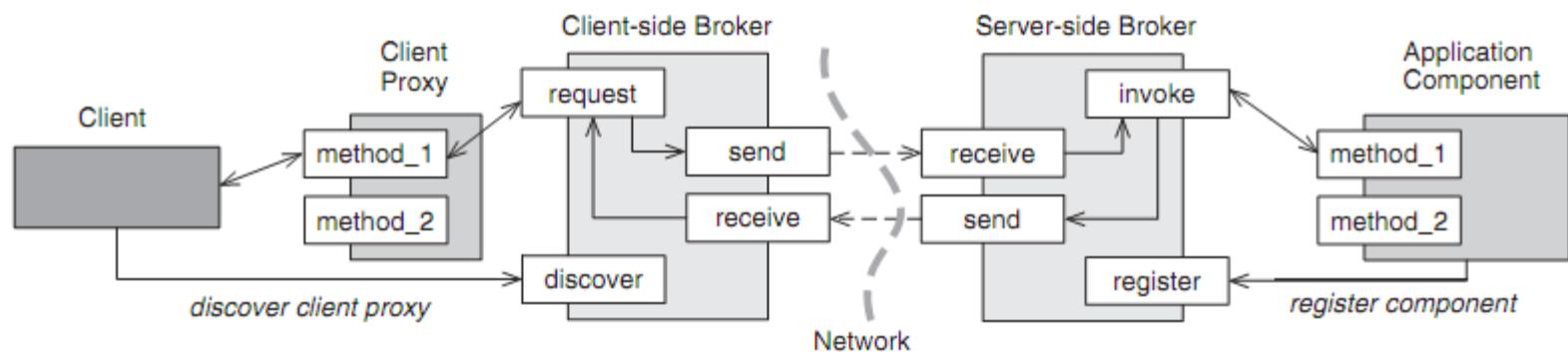


- Event sources publish messages to particular channels on an event bus.
- Event listeners subscribe to particular channels .
- Listeners are notified of messages that are published to a channel to which they have subscribed.
- Generation and notification of messages is asynchronous: an event source just generates a message and may go on doing something else; it does not wait until all event listeners have received the message.

# Broker Pattern

- Distributed systems face many challenges that do not arise in single-process systems.
  - However, application code should not need to address these challenges directly.
- Moreover, applications should be simplified by using a *modular programming model* that shields them from the details of networking and location.
- Solution: Use a federation of *brokers to separate and encapsulate the details of the communication infrastructure* in a distributed system from its application functionality.
- Define a *component-based programming model* so that clients can invoke methods on remote services as if they were local.

# Broker (continued)



- In general a messaging infrastructure consists of two components:
  - A “Requestor” forwards request messages from a client to the local broker of the invoked remote component;
  - While an “Invoker” encapsulates the functionality for receiving request messages sent by a client-side broker and dispatching these requests to the addressed remote components.

# Client-Proxy Pattern

- When constructing a client-side BROKER infrastructure for a remote component we must provide an abstraction that allows clients to access remote components using ***remote method invocation***.
- A “Client Proxy / Remote Proxy” represents a remote-component in the client’s address space.
- The proxy offers an identical interface that maps specific method invocations on the component onto the broker’s message-oriented communication functionality.
- ***Proxies allow clients to access remote component functionality as if they were collocated.***

# Event Demultiplexing & Dispatching

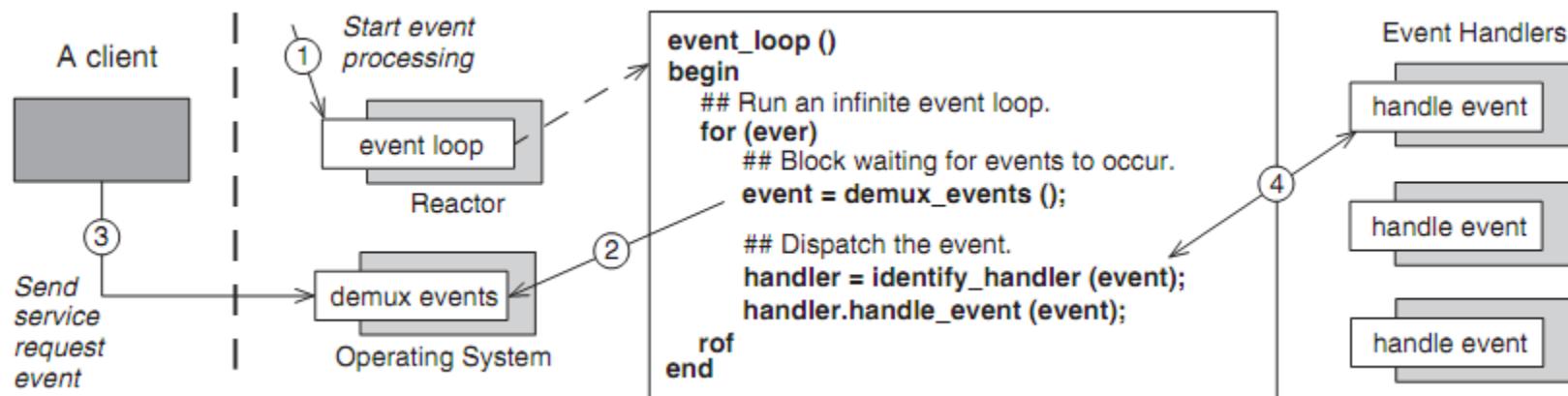
- At its heart, distributed computing is all about *handling and responding to events* received from the network.
- There are patterns that describe different approaches for initiating, receiving, demultiplexing, dispatching, and processing events in distributed and networked systems.
- ...two of these patterns: Reactor ; Proactor ;

# 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 (continued)

- **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.



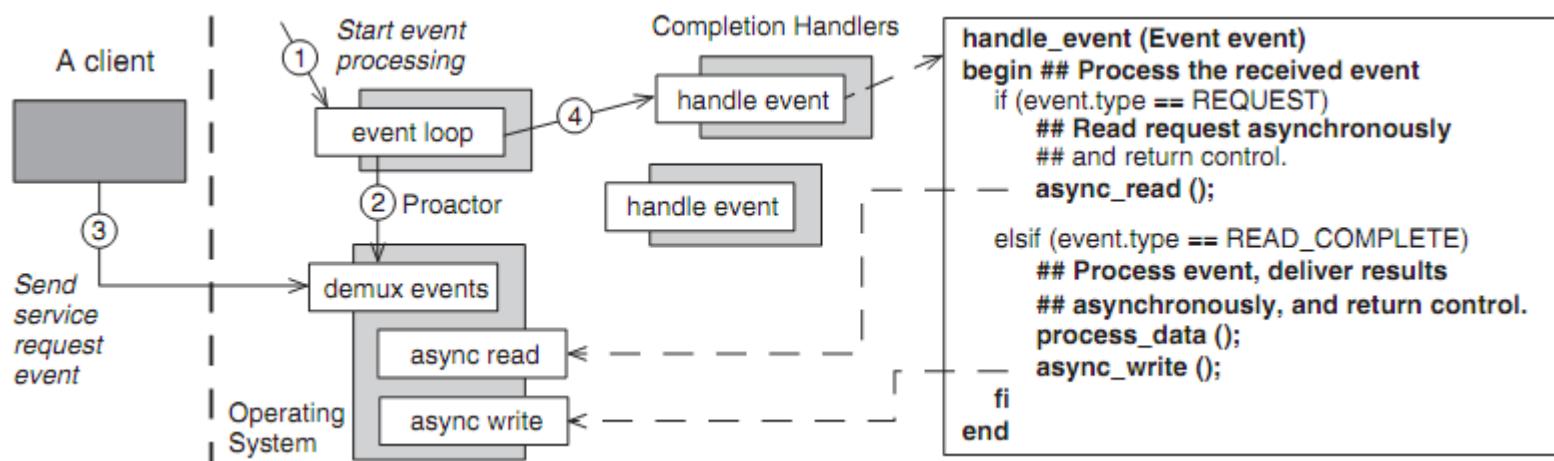
- It defines an **event loop** that uses an **operating system** event demultiplexer to wait synchronously for service request events.
- By **delegating the demultiplexing of events to the operating system**, the reactor can wait for multiple event sources simultaneously without a need to multi-thread 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.***

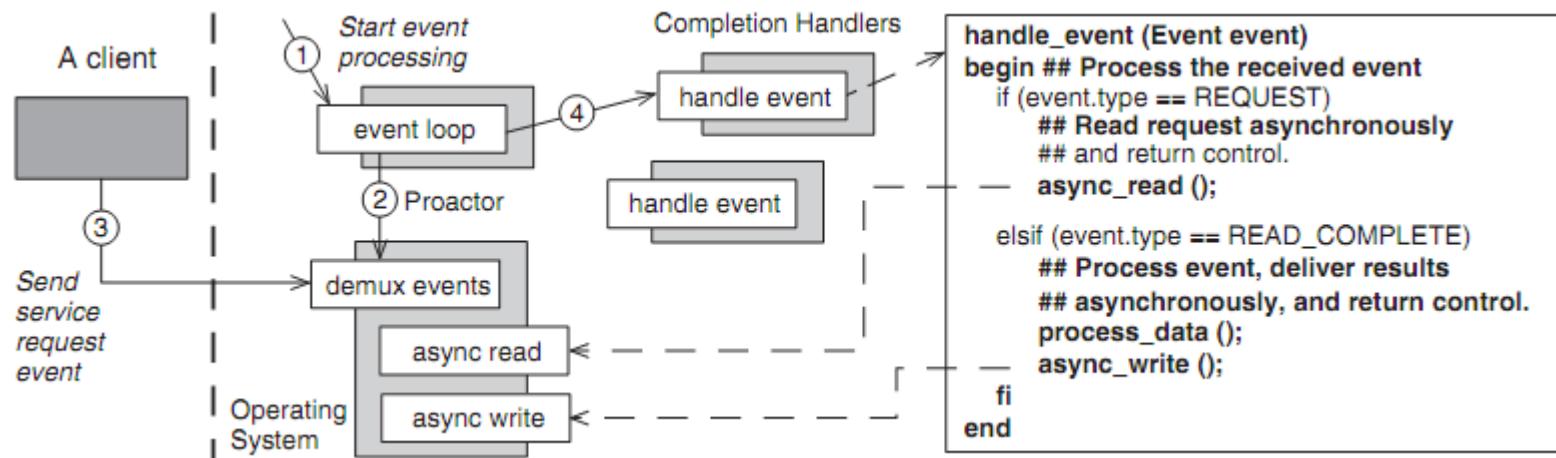
# Proactor (continued)

- 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 (continued)



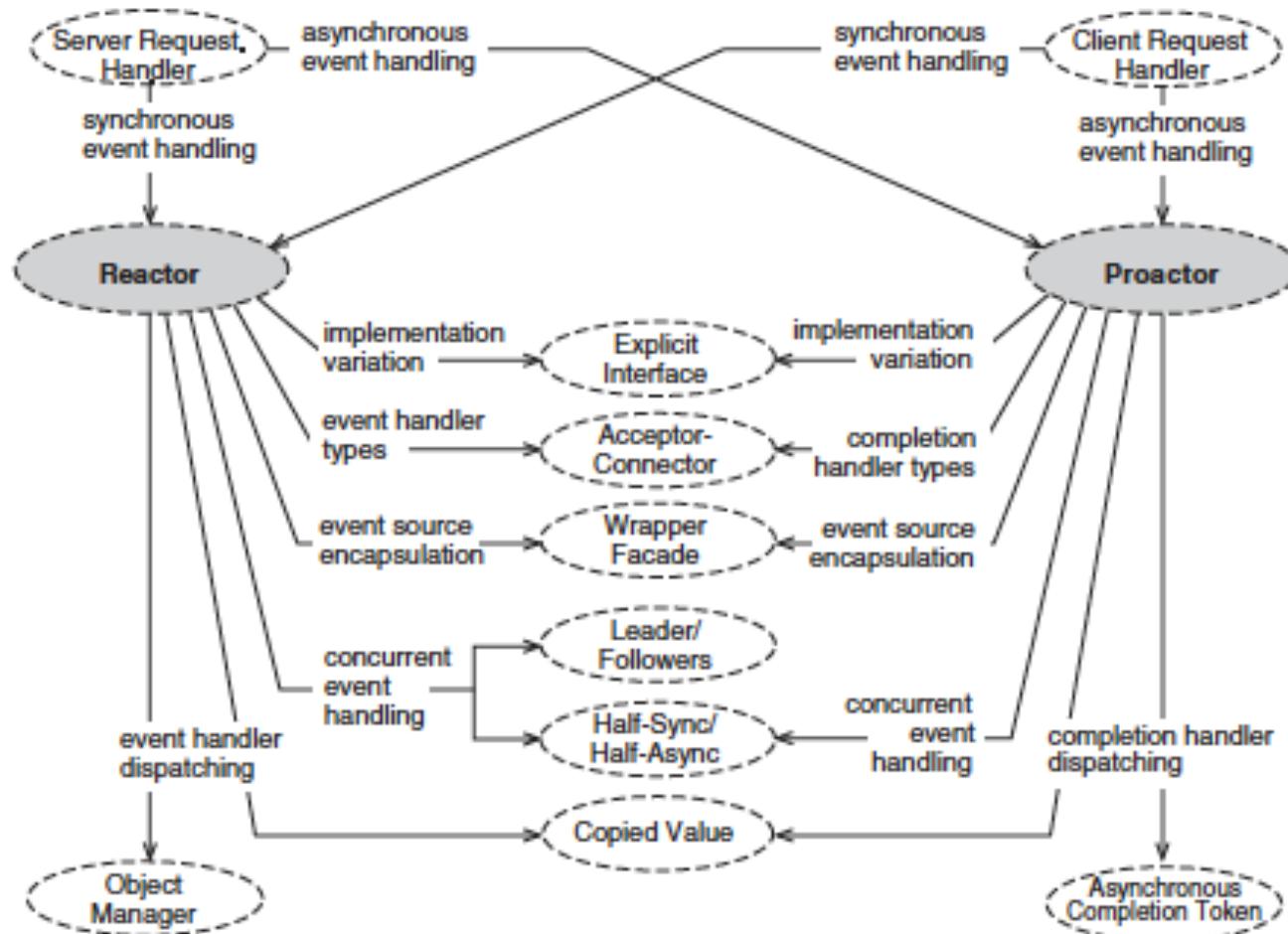
- When such an event arrives, the proactor dispatches the result of the completed asynchronous operation to the corresponding completion handler.
- This handler then continues its execution, which may invoke another asynchronous operation.

# 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 (cont.)

- 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 (continued)

- An Example of a middleware for distributed object-oriented enterprise systems is **CORBA**.
- 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.