# Algoritmi paraleli
# Curs 1

Vlad Olaru

vlad.olaru@fmi.unibuc.ro

Calculatoare si Tehnologia Informatiei

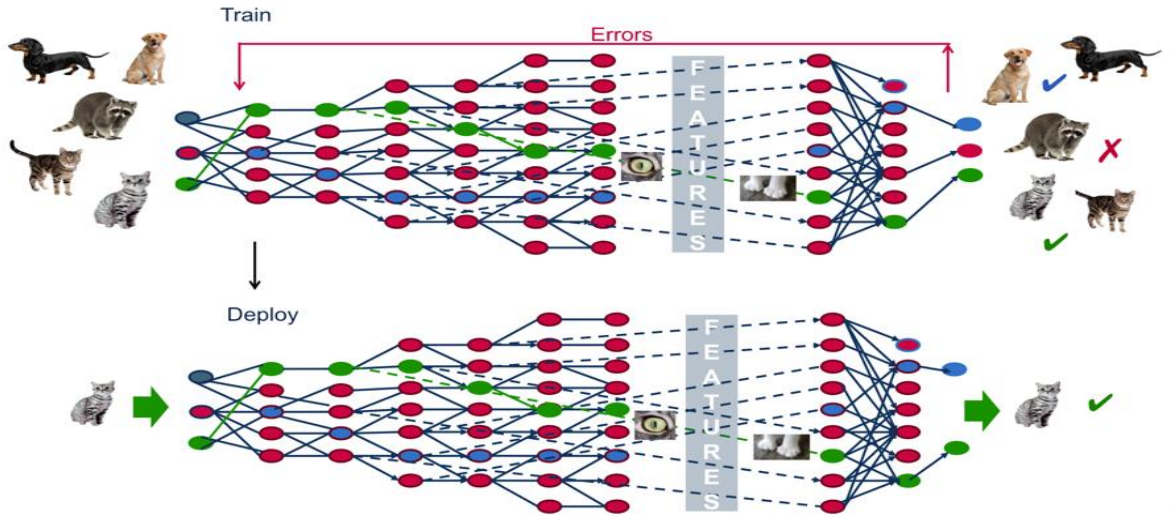
Universitatea din Bucuresti

# Organizare

- examinare in doua (chiar trei) etape
  - examen partial scris: 30 pcte
  - examen final: 70 pcte
- examenul partial
  - se promoveaza cu cel putin 15 pcte
  - fara examen partial promovat nu se intra in examenul final
  - saptamana a 6-a la curs (tentativ)
- examen final
  - prezentarea unui proiect de algoritmi paraleli implementat in MPI
    - echipe 2-3 studenti
  - doua etape:
    - prezentarea specificatiei proiectului (cca 10 min / echipa), saptamana a 7-a (tentativ) la laborator
    - sesiune: prezentarea propriu-zisa a proiectului + demo (cca 30 min/echipa)
- laborator: programare in MPI

# Motivatie calcul paralel

- cerinte sporite de calcul pentru intregi clase de probleme
  - calcul stiintific (simulari, decriptare genom)
  - Web/Internet (retele de socializare, indexare motoare de cautare)
  - deep nets
  - seturi de date pe scara larga (large-scale datasets): calcul paralel + acces paralel la date
  - Large Language Models (LLM) > 600 mil parametri

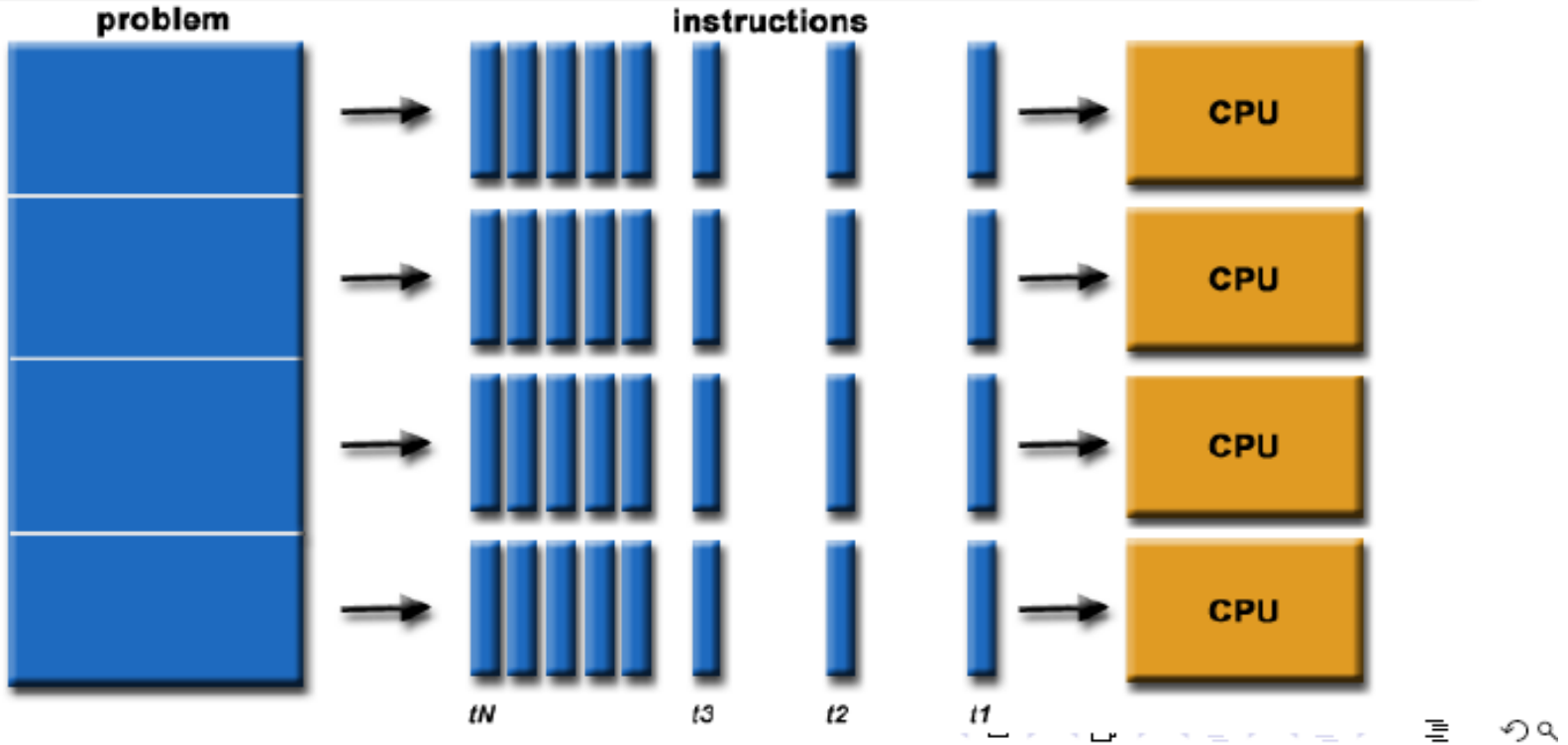



https://mapr.com/blog/demystifying-ai-ml-dl/

# Calcul paralel

- utilizarea simultana a mai multor unitati de procesare pentru rezolvarea unei singure probleme

Plot obtained from https://computing.llnl.gov/tutorials/parallel_comp/



# Arhitecturi paralele

- taxonomia Flynn

1. **SISD** – *Single Instruction stream, Single Data stream*

- arhitectura istorica, secventiala a masinilor uniprocesor

- relevanta actuala
  - sisteme de timp real uniprocesor (non multi-core) fara cache, fara executie speculativa a instructiunilor, samd, ceea ce faciliteaza analiza WCET (worst case execution time)
  - safety-critical systems (aviatie, radare, centrale nucleare, etc), eg MILS

2. **MISD** – *Multiple Instruction streams, Single Data stream*

- utilizata rar, ex: sisteme tolerante la defecte

# Arhitecturi paralele (cont.)

3. **SIMD** – *Single Instruction stream, Multiple Data streams*

- exemplu tipic: arhitecturi vectoriale tip GPU (heartbeat synchronized)

- in particular, pentru arhitecturile vectoriale se foloseste si termenul de **SIMT:** *Single Instruction, Multiple Threads* (NVIDIA)

- *predicated / masked SIMD:* executia respectiv ne-executia (skip) operatiei vectoriale asupra unui element particular din vector (in functie de valoarea sa) poate fi controlata de un predicat

  ex:

  ```
  c[i] = a[i] / b[i]   if b[i] != 0;
  ```

- model computational folosit in framework-uri de lucru din AI (caffe, pytorch, tensorflow, etc)

# Arhitecturi paralele (cont.)

4. **MIMD** – *Multiple Instruction streams, Multiple Data streams*

- restul echipamentelor moderne de calcul, de la smartphones la calculatoare paralele/distribuite
- arhitecturi multiprocesor UMA/NUMA, clustere, grid-uri, arhitecturi masiv paralele (CRAY)
- cunoscute si informal sub numele de sisteme multiprocesor (MP)
- subdiviziune:

a. **SPMD** – *Single Program, Multiple Data streams*

- difera de SIMD, instructiunile aceluiasi program se executa independent, nesincronizat, asupra unor date diferite
- cel mai uzual model de programare paralela

b. **MPMD** – *Multiple Programs, Multiple Data streams*

- multiple procesoare independente executa simultan cel putin doua tipuri de programe
- ex: master/slave programs, Intel MPI library MPMD launch

# Tipuri de paralelism

- definit de **granularitatea** G a task-ului

$$G = T_{comp} / T_{comm}$$

unde $T_{comp}$ = timpul de calcul al unui task

$T_{comm}$ = timpul necesar schimbului de date intre procesoare

1. *paralelism cu granularitate fina* (*fine-grained parallelism*)
   - programul paralel este impartit in multe task-uri mici, fiecare asignate individual unui procesor
   - asigura o incarcare echilibrata a sistemului multiprocesor
   - in general creste overhead-ul de comunicare si sincronizare
   - potrivit pentru arhitecturi paralele cu latenta de comunicare mica, cum sunt sistemele MIMD cu memorie partajata (shared memory / tightly coupled MPs)

# Tipuri de paralelism (cont.)

2. *paralelism cu granularitate mare* (*coarse-grained parallelism*)
   - programul paralel este impartit in task-uri mari
   - majoritatea procesarii are loc pe cateva procesoare
   - poate induce dezechilibru in incarcarea  sistemului multiprocesor
   - in general  overhead-ul de comunicare si sincronizare este mic, pentru ca task-urile nu prea interactioneaza
   - potrivit pentru arhitecturi paralele cu latenta de comunicare mare
     - eg, sistemele MIMD cu memorie distribuita (message-passing / loosely coupled MPs)

# Tipuri de paralelism (cont.)

- *concluzie*: performanta unei solutii de paralelizare a unui program depinde de alegerea unei granularitati potrivite pentru arhitectura paralela aflata la dispozitie

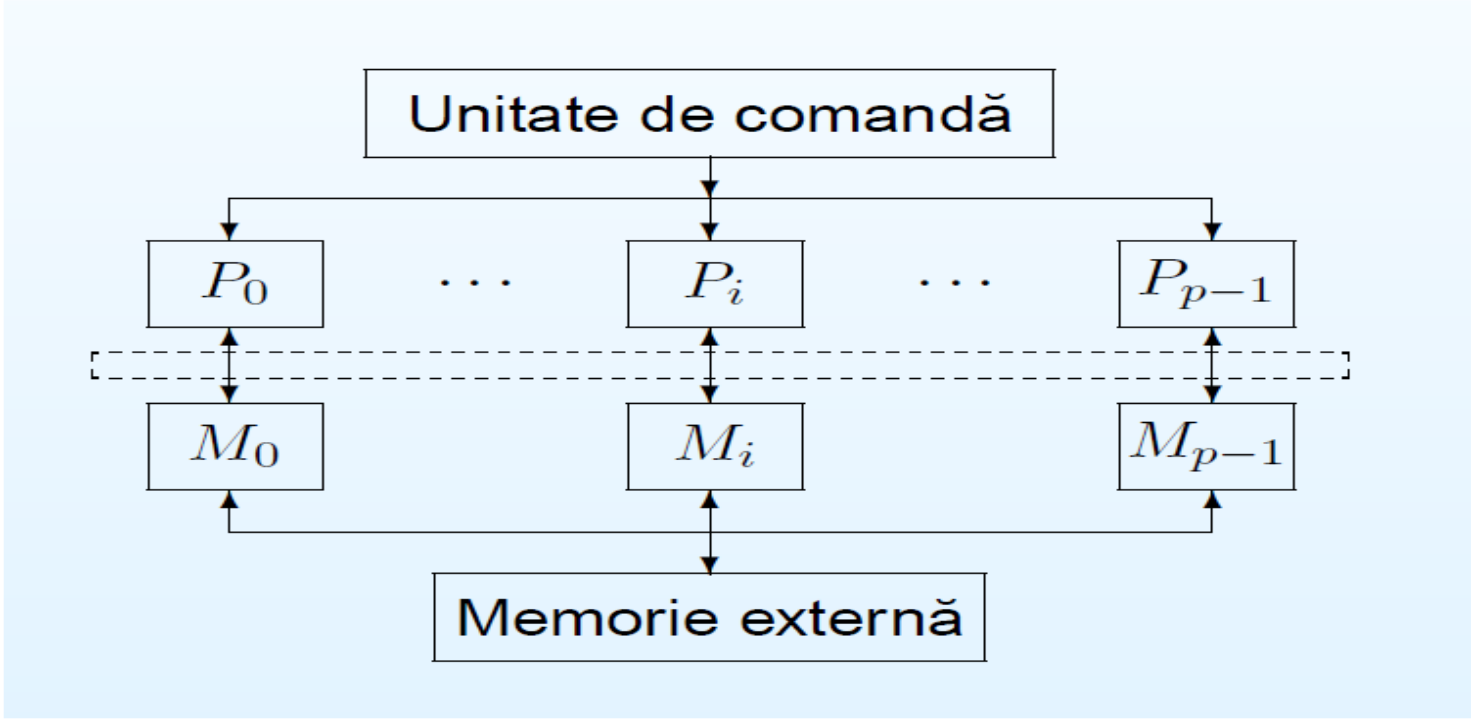
– programe paralele cu granularitate fina vor rula suboptimal pe arhitecturi paralele cu memorie distribuita
  – comunicatie intensa si scumpa

– programe paralele cu granularitate mare nu vor folosi optimal arhitecturile cu memorie partajata
  – majoritatea calculului se va face pe procesoare putine
  – nu se exploateaza potentialul de paralelism al arhitecturii

- obs: alegerea unei solutii optimale de paralelizare nu e intotdeauna posibila
  – ex: N-body problem pt DSM (Distributed Shared Memory)

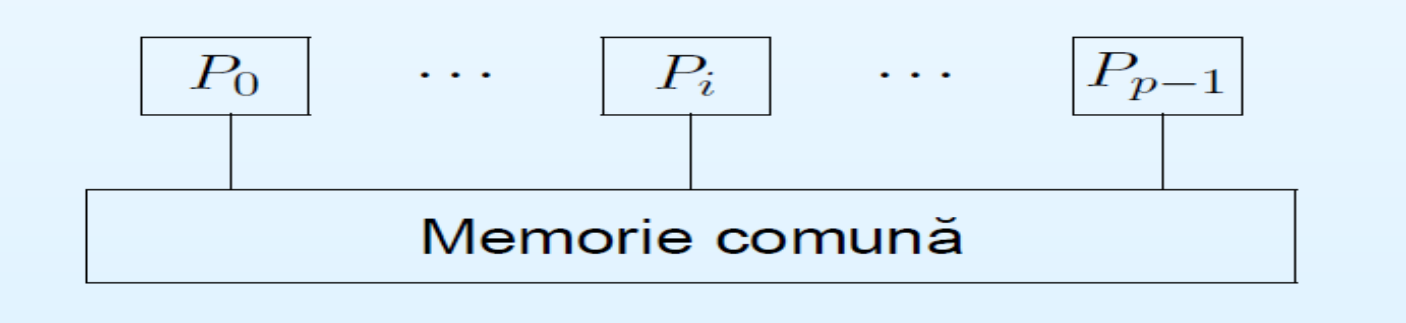
# Arhitecturi SIMD

- fiecare procesor $P_i$ executa aceeasi instructiune asupra unei date din memoria locala $M_i$

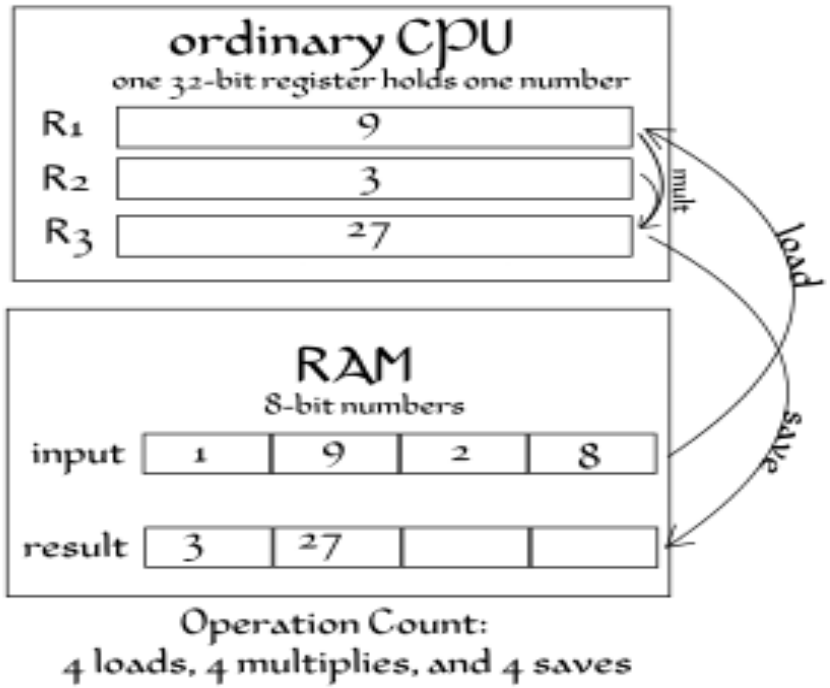


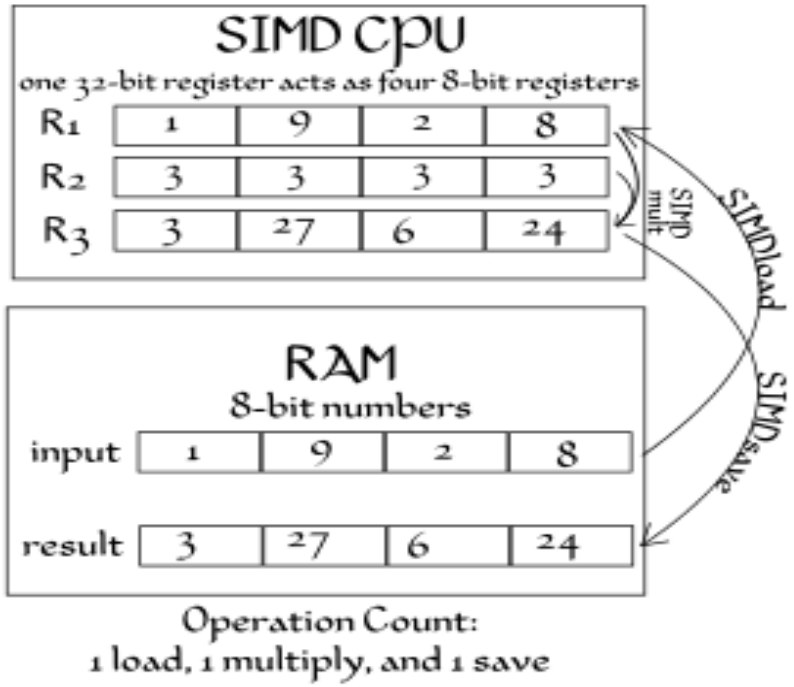
# Arhitecturi SIMD (cont.)

- memoriile locale procesoarelor sunt rapide
- procesoarele sunt legate de memoriile locale prin retele de interconectare configurabile (orice procesor poate accesa orice modul de memorie)
- in general, procesoare simple
- operatii vectoriale
  - vectori de lungime multiplu al numarului de procesoare $p$
  - se executa intr-un singur tact pe $p$ procesoare (eg, adunare vectori)
- problema critica: accesul la date pentru alimentarea memoriilor locale



# Arhitecturi SIMD (cont.)




https://en.wikipedia.org/wiki/SIMD

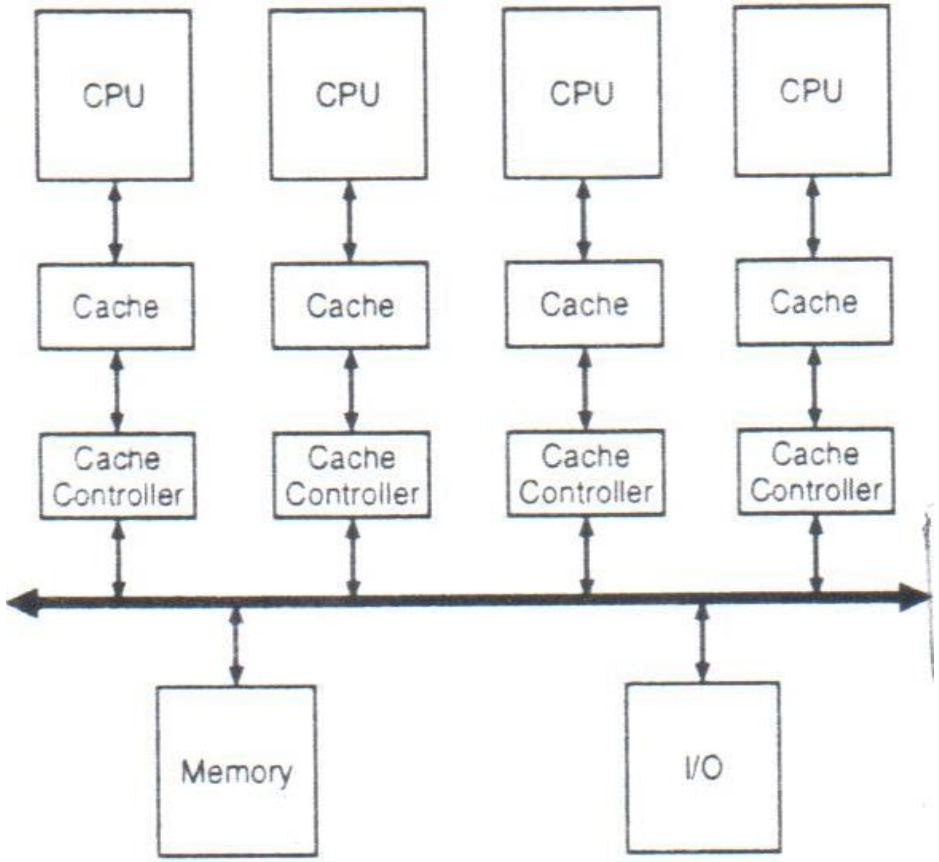
# Arhitecturi MIMD

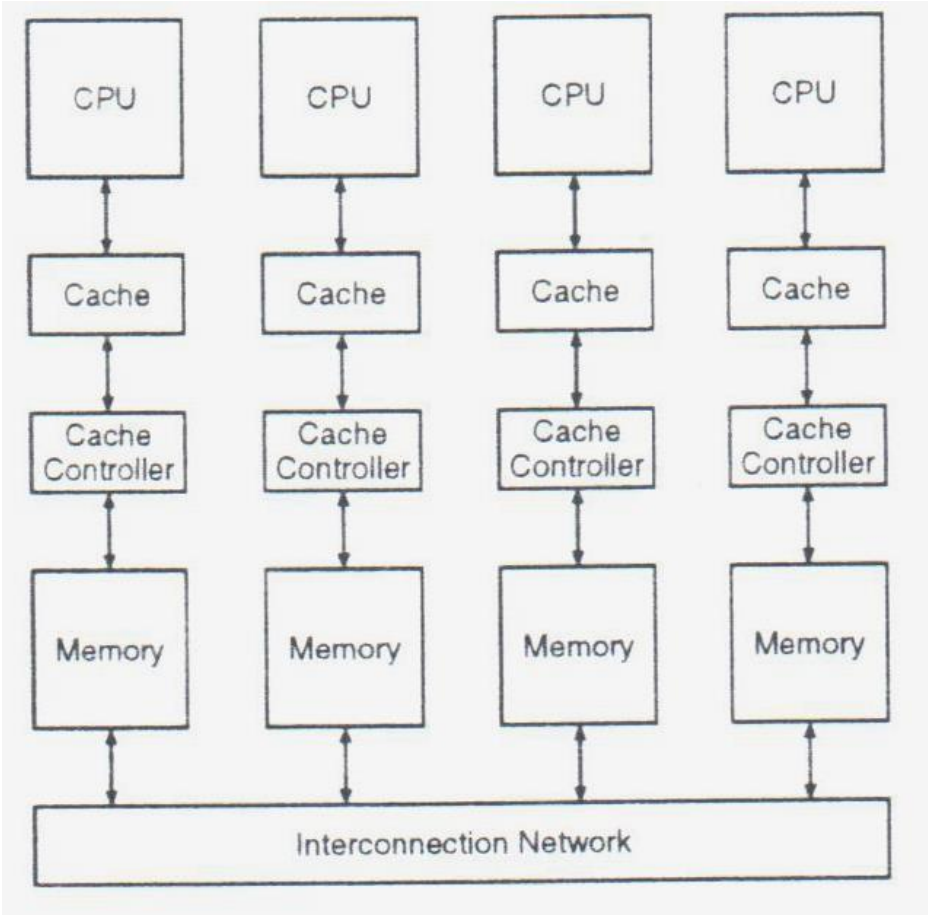
- referite uzual ca sisteme multiprocesor (MP)
- felul in care se interconecteaza procesoarele individuale in sistemele multiprocesor determina in general si tipul de acces la memorie
  - Uniform Memory Access (UMA)
  - Non-Uniform Memory Access (NUMA)
  - No Remote Memory Access (NORMA)
- in sistemele UMA/NUMA procesoarele comunica prin variabile partajate de memorie accesate sincronizat (cu ajutorul lock-urilor, de pilda)
- in sistemele NORMA accesul coordonat la date se face prin schimb de mesaje (message passing)

# MP cu memorie partajata

- fiecare procesor executa propriul flux de instructiuni
- datele se afla in memoria comuna
- uzual procesoarele au cache-uri locale
- operatii paralele la nivel de bloc
- avantaj: comuncatie simpla prin memoria comuna
- dezavantaj: scalabilitatea
  - cresterea numarului de procesoare induce cresterea numarului de conflicte la memorie => scaderea vitezei de calcul

# MP cu memorie partajata



# MP cu o singura magistrala

- magistrala unica reprezinta principala limitare pt. cresterea nr. de CPU
- folosirea cache-urilor reduce traficul pe magistrala si permite cresterea nr de procesoare
- existenta cache-urilor locale => copii multiple ale aceleiasi locatii de memorie => potential de inconsistenta daca un CPU modifica datele (locale)
- accesul uncached la datele partajate nu e indicat
  - scade viteza accesului la date
  - creste traficul pe magistrala ceea ce incetineste si accesul celorlalte procesoare inclusiv la datele nepartajate
- ideal, cand un procesor citeste date partajate pot exista copii multiple si in alte cache-uri, dar la scriere e nevoie de acces exclusiv la date urmat de invalidarea celorlalte copii existente

# MP cu o singura magistrala (cont.)

- solutia: protocoale de mentinere a coerentei cache-urilor locale
- ex de protocol uzual: *snoop coherency*
  - controlerele cache-urilor monitorizeaza magistrala pt traficul care afecteaza datele din cache-urile lor
  - *read miss*: controlerul localizeaza o copie actualizata a datelor (posibil aflata in cache-ul altui CPU)
  - *write*: exista doua protocoale posibile, *write-invalidate* si *write-update*
  - *write-invalidate* invalideaza toate copiile celorlalte procesoare inainte de a scrie datele in cache-ul local (eg. Intel MESI)
  - *write-update* scrie datele modificate pe magistrala printr-o operatie de broadcast
    - controlerele de cache isi pot actualiza copiile locale ale datelor la valoarea modificata

# Suport HW pentru sincronizare

- multe procecesoare ofera suport HW pt implementarea sectiunilor critice
- uniprocesoare – pot dezactiva intreruperile
  - codul current se executa fara a fi interrupt (preempted)
  - ineficient in sisteme multiprocesor
- instructiuni HW atomice
  - instructiuni procesor speciale executate *in mod atomic (neintrerupt)*
  - fie testeaza si modifica (*test-and-modify)* continutul unui cuvant de memorie, **Test-and-Set (TAS)**
  - fie schimba (*swap)* continutul a doua cuvinte de memorie (sau mai multe), **Compare-and-Swap** (**CAS**, cu varianta **CAS2**)

# Test-and-set (TAS)

- definitie

```
int tas(int *lock)
  {
        int rv = *lock;
        *lock = 1;
        return rv;
  }
```

- proprietati
  - executie atomica
  - intoarce valoarea originala a parametrului pasat functiei
  - seteaza noua valoarea a parametrului pe 1

# Spinlock-uri cu TAS

- lock (lacat, zavor), concept (abstractie) de nivel inalt pt. protectia sectiunilor critice
- ofera doua operatii: *acquire* (apelata la intrarea in sectiune critica) si respectiv *release* (apelata la iesirea din sectiune critica)
- cand un proces/thread detine lock-ul, celelalte sunt in busy waiting ("spinning in the while loop") => numele de *spinlocks*
- pt a se reduce busy waiting, sectiunile critice implementate cu spinlocks trebuie sa fie foarte scurte

```
int lock = 0;
void acquire(int *lock)
{
        while(tas(lock))
                ;
}
void release(int *lock)
{
        *lock = 0;
}
```

# Instructiuni atomice multiprocesor

- se folosesc instructiuni atomice de tip TAS/CAS
- TAS in sisteme multiprocesor
  - cand un CPU executa TAS, arbitrul de magistrala da drept de folosinta exclusiva a magistralei procesorului respectiv pe durata executiei instructiunii
  - toate celelalte surse generatoare de accese de memorie sunt blocate pe durata TAS
  - se executa ciclul read-modify-write
  - la final, se deblocheaza magistrala pt uzul altor procesoare

# Consecinte ciclu read-modify-write

- intr-un astfel de ciclu se citeste si se scrie o valoare in mod atomic, iar datele nu sunt cached

=> operatie mai costisitoare decat operatiile de citire/scriere obisnuite

=> magistrala e ocupata mai mult timp

=> operatiile TAS afecteaza semnificativ rata de transfer pe magistrala

- alte consecinte privesc implementarea spinlock-urilor in sisteme multiprocesor

# Spinlock-uri TAS MP

```
void acquire(char *lock_ptr)
{
        disable_interrupts();
        while(tas(lock_ptr))
                ;
}
void release(char *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

# Observatii

- dezactivarea intreruperilor in *acquire* impiedica pierderea procesorului pe durata detinerii lock-ului
- pierderea procesorului ar insemna ca procese/thread-uri de pe alte CPU nu pot intra in sectiune critica din cauza unui proces/thread care nu ruleaza !
- pe durata detinerii unui lock, celelalte procesoare executa TAS in bucla si consuma inutil largimea de banda a magistralei afectand procesoare care nu sunt implicate in accesul la sectiunea critica !
- la *release*, procesorul care detine lock-ul concureaza cu celelalte procesoare pt accesul la magistrala inainte de a putea ceda lock-ul
- solutie: test-and-test-and-set (TATAS)

# TATAS

- traficul pe magistrala se poate reduce daca procesoarele cicleaza in *acquire* pe o copie locala a spinlock-ului
- pt ca *release* e in fond o operatie de scriere a spinlock-ului, protocolul de coerenta snoop detecteaza scrierea si invalideaza copiile locale ale celorlalte procesoare (in cazul protocolului write-invalidate)
- cand celelalte procesoare acceseaza din nou spinlock-ul => cache miss
- la cache miss copia locala se va actualiza si va reflecta starea de lock free
- procesorul incearca din nou sa execute operatia TAS sperand ca acum lock-ul e probabil free

# Spinlock-uri TATAS

```
void acquire(char *lock_ptr)
{
        disable_interrupts();
        while(*lock_ptr || tas(lock_ptr))
                ;
}
void release(char *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

*Obs: codul se bazeaza pe evaluarea scurtcircuitata a conditiilor in C*

# TATAS vs TAS

- pt sectiuni critice scurte si protocol write-invalidate, costurile sunt asemanatoare
  - dupa ce un CPU elibereaza lock-ul si altul il obtine, dureaza pana cand celelalte procesoare ajung sa cicleze pe copia locala a lock-ului
  - in tot acest timp, magistrala e saturata cu trafic de invalidare, read miss si TAS
- secventa tipica de evenimente
  - eliberare lock => invalidare copii din cache-urile altor CPU
  - invalidarile genereaza read miss pt toate CPU care cicleaza, toate citesc noua valoare a lock-ului de pe magistrala si ordinea e seriala !
  - primul CPU care obtine noua valoare a lock-ului executa TAS si obtine lock-ul
  - restul CPUs primesc noua valoare, executa TAS si esueaza
  - fiecare TAS invalideaza copiile locale din cache-uri si forteaza un read miss

# TATAS cu backoff

- competitia pt. magistrala partajata seamana cu protocolul de transmisie a datelor pe Ethernet (CSMA/CD)
  - analogia nu e stricta: pe Ethernet toti transmitatorii simultani esueaza, la TAS pe MP arbitrul de magistrala garanteaza un castigator
- analogia sugereaza insa scheme de backoff pt. reducerea competitiei la magistrala a procesoarelor care cicleaza
- metoda introduce o intarziere (delay) inainte de a incerca din nou operatia TAS
- intarzierea poate fi statica sau dinamica
- intarzierea statica:
  - fiecare CPU are un slot
  - merge bine pentru multe CPU-uri
  - intarzie nejustificat un singur CPU chiar daca lock-ul e liber

# TATAS cu backoff (cont.)

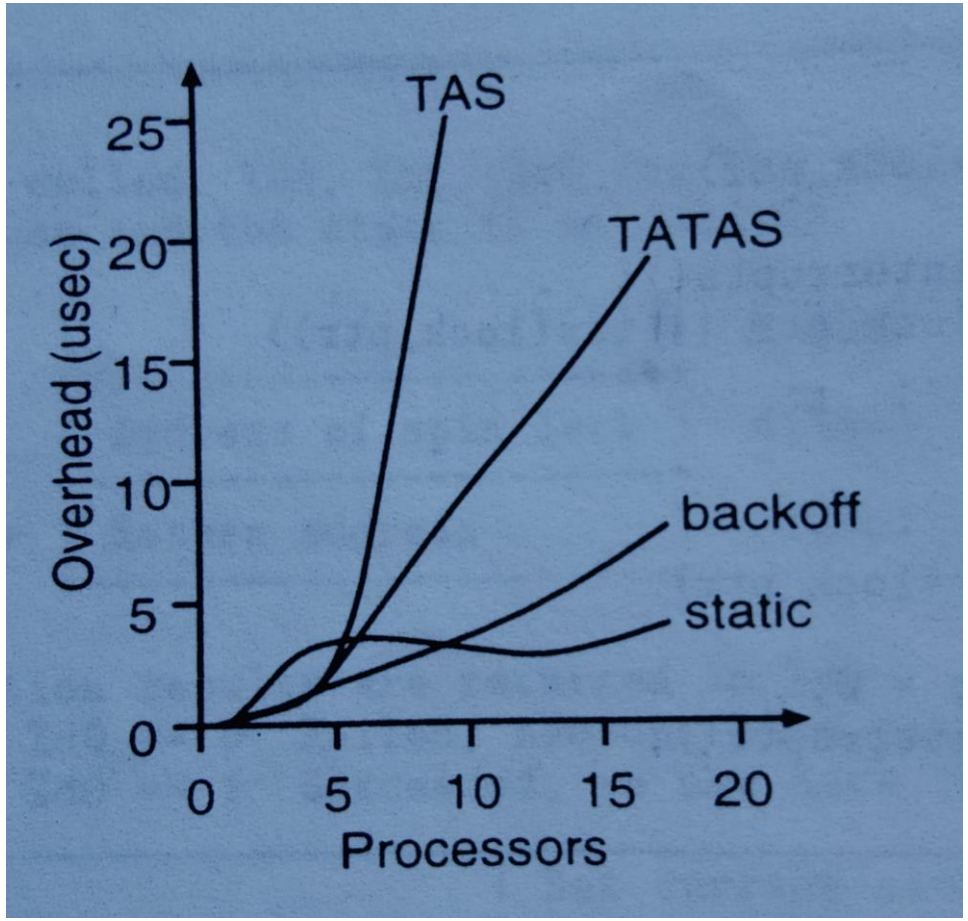
- intarzierea dinamica:
  - la inceput, toate CPU-urile aleg o intarziere mica => coliziuni
  - dupa detectarea coliziunilor, fiecare CPU mareste intarzierea
  - overhead-ul intarzierilor mici de la inceput face metoda mai costisitoare decat alocarea statica a intarzierilor
- obs: folosirea protocolului *write-update* poate imbunatati performanta TATAS prin reducerea traficului pe magistrala
  - cand un lock e eliberat, noua sa valoare poate fi transmisa prin broadcast
  - fiecare CPU care monitorizeaza magistrala isi poate astfel actualiza copia locala fara a mai genera *read miss*

# Spinlock-uri TATAS cu backoff

```
void acquire(char *lock_ptr)
{
        disable_interrupts();
        while(*lock_ptr || tas(lock_ptr))
        {
                while(*lock_ptr)
                        ;
                delay();
        }
}
```

```
void release(char *lock_ptr)
{
        *lock_ptr = 0;
        enable_interrupts();
}
```

# Performanta TAS



# Dezavantaje instructiuni atomice

- instructiunile atomice simplifica IPC pe multiprocesoare, dar au si dezavantaje
  - pot avea performanta redusa daca sunt folosite neglijent
  - daca un proces/thread pierde CPU sau e fortat sa astepte o perioada lunga (eg, page fault) in timp ce detine lock-ul, celelalte procese nu pot avansa pana cand detinatorul lock-ului revine pe CPU si elibereaza lock-ul
  - daca un proces/thread se termina anormal in timp ce detine un lock, nici un alt proces/thread nu poate intra in sectiune critica
  - inversarea prioritatilor – cand un proces/thread cu prioritate mica detine lock-ul si impiedica un proces cu prioritate marea sa intre in sectiune critica
- solutie: sincronizare *wait-free* (*lock-free*), Herlihy 1991

# Sincronizare wait-free (lock-free)

- idee centrala: se incearca executia operatiei, dar se lasa datele intr-o stare consistenta daca operatia esueaza
  - analogie cu tranzactiile din baze de date
  - model de control optimist al concurentei
- se pot folosi instructiuni HW de tipul CAS/CAS2 sau LL/SC
- model Herlihy:
  - sistem cu *n* procese secventiale care comunica prin memorie partajata
  - *obiect concurent* = ({tip+valori}, set operatii, specificatie secventiala)
  - *obiect concurent neblocant* – procesul care executa una dintre operatiile sale trebuie o termine intr-un nr finit de pasi
  - *obiect concurent wait-free* – fiecare proces din sistem trebuie sa termine operatia intr-un nr finit de pasi

# Sincronizare wait-free (cont.)

- conditia de operare *nonblocanta* - unele procese/threaduri vor face intotdeauna progres indiferent de intarzierile sau opririle fortate ale altor procese/threaduri
- conditia de operare *wait-free* e mai puternica => toate procesele/threadurile **care nu sunt oprite** fac progres indiferent de terminarile anormale sau intarzierile altor procese/threaduri
- ambele conditii **exclud** folosirea sectiunilor critice ca metoda de implementare
  - proces/thread care se termina anormal (cu eroare) in mijlocul unei sectiuni critice va bloca pt totdeauna celelalte procese care asteapta sa intre in sectiunea critica
- **rezultat teoretic**: *este imposibil sa se construiasca implementari neblocante sau wait-free ale tipurilor de date simple folosind operatii de citire/scriere, TAS, fetch-and-add, swap de memorie cu registre* (eg, instructiunea Intel *xchg*)

# Semantica compare-and-swap (CAS)

- definitie

```
int compare_and_swap(int *value, int expected, int new_value)
 {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
     return temp;
 }
```

- proprietati
  - executie atomica
  - intoarce valoarea originala a parametrului **`value`**
  - seteaza variabila **`value`** la valoarea parametrului **`new_value`** doar daca e indeplinita conditia **`*value == expected`**, adica, schimbul de valori se face doar daca e indeplinita conditia

# Incrementare atomica wait-free folosind CAS

```
1       void increment(atomic_int *v)
2       {
3               int temp;
4               do {
5                       temp = *v;
6               }while (temp != (compare_and_swap(v,temp,temp+1));
7       }
```

Scenariu de executie:

- pp initial v = 10
- pp. proces de prioritate mica $P_l$ executa codul primul
- $P_l$ pierde CPU intre liniile 5 si 6, moment in care un proces cu prioritate mare $P_h$ incepe executia
- starea $P_l$ este v=10, temp=10, iar $P_h$ incheie cu succes incrementarea => v = 11
- cand $P_l$ se reia, **cas(11,10,11)** esueaza si se re-executa linia 5 => temp = v = 11 si **cas(11, 11, 12)** reuseste = > v = 12

Obs: spre deosebire de TAS care modifica operandul intotdeauna, CAS/CAS2 modifica operandul *doar daca reuseste comparatia* !

# Suport HW RISC pentru sectiuni critice

- TAS, CAS specifice procesoarelor CISC, dar secvente atomice de tip *read-modify-write* nu se pot implementa pe procesoare RISC (arhitecturi load/store)
- operatii speciale RISC: Load Linked (LL) si Store Conditional (SC)
- LL incarca o variabila din memorie intr-un registru CPU si apoi verifica activ daca variabila din memorie este modificata de alte procesoare
- SC verifica daca au existat modificari ale variabilei de memorie de la ultimul LL
  - daca nu, valorea registrului  se stocheaza in memorie cu success (variabila din memorie este modificata) iar continutul registrului este setat pe 1
  - daca da, stocarea esueaza (variabila de memorie ramane nemodificata) iar valoarea registrului se seteaza pe 0

# Incrementare atomica wait-free cu LL/SC

- in fapt, o forma elementara de memorie tranzactionala (zona de memorie restransa la un byte/word)
- functioneaza cf. principiilor de control optimist al concurentei din baze de date
- simuleaza executia unei tranzactii din baze de date cf principiilor ACID (Atomicity, Consistency, Isolation, Durability)

```
1   increment:
2       ll      $2, count           ; incarca valoarea counter-ului
3       addu    $3, $2, 1           ; incrementeaza valoare counter
4       sc      $3, count           ; incearca sa stocheze noua valoare
5       beq     $3, zero, increment         ; zero inseamna esec
6       j       $31                 ; return din rutina
```

# Proprietati ACID incrementare LL/SC

- *atomicitate* - modificarile (incrementarea *counter-ului*) sunt executate ca si cand operatia ar fi atomica, i.e. fie toate modificarile sunt executate, fie niciuna
- *consistenta* - datele (*counter-ul*) sunt intr-o stare consistenta cand tranzactia incepe si cand se termina, i.e. *counter-ul* e incrementat corect, chiar daca se executa mai multe tranzactii simultan, eventual intretesut
- *izolare* - starea intermediara a tranzactiei (apelul procedurii si valorile registrelor) e invizibila altor tranzactii (altor apeluri ale aceleiasi proceduri)
  - tranzactiile concurente (apelurile concurente ale procedurii *increment*) par a fi serializate
- *durabilitate* – dupa incheierea cu succes a tranzactiei (apelul procedurii *increment*), datele sunt salvate in memoria principala RAM si modificarea e finala
  - analogia cu bazele de date se opreste aici, stocarea in RAM nu e persistenta

# Comparatie CAS vs. LL/SC

- daca au aparut modificari ale counter-ului, SC va esua garantat, chiar daca valoarea initial citita de LL a fost restaurata
  - ex: problema ABA
- daca se incearca aceeasi secventa de operatii cu CAS, adica
  - se citeste valoarea *counter-ului*
  - apoi se executa CAS
  - daca vechea valoare a fost restaurata, nu se vor detecta modificarile intermediare aparute intre operatia de read si cea de CAS

    => semantica LL/SC e mai puternica decat CAS
- atat CAS cat si LL/SC se pot folosi pentru implementarea sincronizarii *wait-free*

# Memorie tranzactionala

- software transactional memory, Herlihy & Moss 1993
- am vazut deja ex. LL/SC & CAS
- exploateaza ideea de tranzactii din baze de date (v. proprietati ACID) si controlul optimist al concurentei
- un thread termina modificarile operate pe o zona de memorie partajata fara a se coordona cu alte threaduri
- se inregistreaza fiecare operatie de citire/scriere intr-un log
- la finalul tranzactiei se incearca operatia de *commit*
  - daca reuseste (i.e., nici o alta tranzactie nu a apucat sa faca un *commit* reusit intre timp), tranzactia e validata si modificarile devin permanente
  - daca esueaza, tranzactia e abandonata (aborted) si se re-executa de la inceput pana reuseste (*transaction roll-back*)
- beneficiu major: grad crescut de concurenta, thread-urile nu au nevoie sa se sincronizeze prin lock-uri la accesul memoriei partajate

# Suport HW pt. memorie tranzactionala

- ex. Intel TSX (Transactional Synchronization Extension), extensie ISA x86, microarhitectura Haswell (2013)
- accesibila prin intermediul a doua interfete
  - HLE (Hardware Lock Elision) – backward compatibility
  - RTM (Restricted Transactional Memory)
- HLE adauga doua prefixuri de instr. XACQUIRE & XRELEASE
  - se pot folosi doar pt anumite instructiuni care trebuie prefixate explicit cu LOCK
  - permite executia optimista a sectiunii critice sarind scrierea lock-ului, a.i. acesta apare liber pt alte threaduri
  - o tranzactie esuata determina reluarea operatiei de la instructiunea prefixata cu XACQUIRE, dar instructiunea se va reexecuta ca si cand nu a fost prefixata cu XACQUIRE

# Suport HW pt. memorie tranzactionala (cont.)

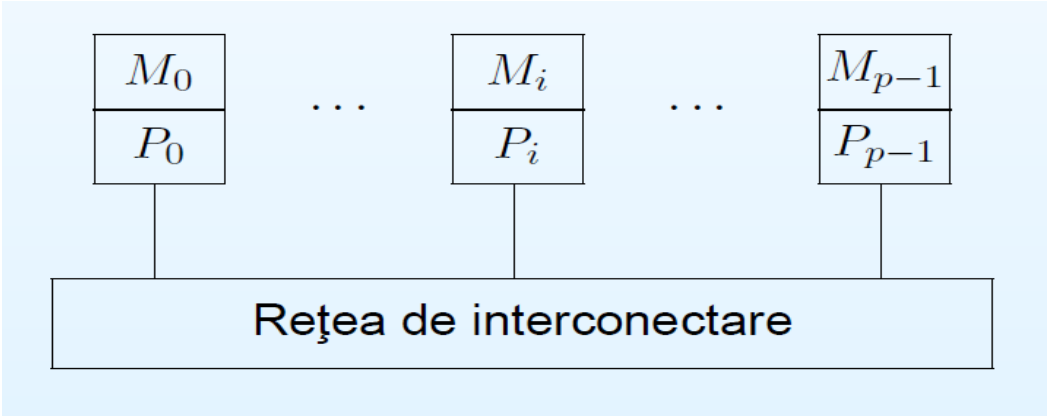
- RTM adauga la ISA trei noi instructiuni
  - XBEGIN – marcheaza inceputul zonei de memorie tranzactionala
  - XEND - marcheaza sfarsitul zonei de memorie tranzactionala
  - XABORT – abandoneaza explicit o tranzactie
  - esecul tranzactiei redirecteaza executia catre codul specificat de instr. XBEGIN, iar codul de eroare e stocat in registrul EAX
- instructiunea XTEST permite testarea starii procesorului (este sau nu in mijlocul executiei unei regiuni de memorie tranzactionala)

# RTM elision pt. pthread_mutex_lock

```
void elided_lock_wrapper(lock) {
        if(_xbegin() == _XBEGIN_STARTED) { // start tranzactie
                if(lock  e liber)
                        return;         // executa sectiunea critica in tranzactie
                _xabort(0xff); // abandoneaza tranzactia
        }
        obtine lock
}
void elided_unlock_wrapper(lock) {
        if (lock e liber)
                _xend();                // comite tranzactia
        else
                elibereaza lock;
}
```

# Arhitecturi NORMA

- arhitecturi MIMD cu memorie distribuita (s.n. si *sisteme distribuite*)
- fiecare procesor are memorie locala proprie
- comunicatia intre procesoare se face schimb de mesaje (*message passing*) peste o retea de interconectare
- operatii paralele la nivel de bloc
- comunicatia prin mesaje necesita algoritmi dedicati



# Arhitecturi NORMA (cont.)

- topologii de retea de comunicatie
  - fixe: array (inel), grid (tor), hipercub
  - configurabile: switch, magistrala
- uzual, topologia e transparenta utilizatorului
- suport pentru comunicatie: biblioteci message passing
- avantaj: nr de procesoare foarte mare

# Modelare topologii de interconectare

- se folosesc grafuri neorientate
- procesoarele sunt noduri, iar arcele linii de comunicatie interprocesor
- graf neorientat G = (V, E)
  - V = multimea nodurilor (*vertices*)
  - E = multimea arcelor (*edges*), E ⊂ V x V
- *ordinul* grafului = nr de noduri (echivalent, nr de procesoare)
- graf simplu = graf fara cicluri (aciclic)

# Grafuri

- noduri *adiacente* (vecine): exista arc intre ele
- *gradul unui nod*:  nr de arce incidente nodului (echivalent, nr de noduri adiacente nodului)
- *gradul unui graf* ($\Delta$): gradul maxim al nodurilor din graf
- graf *regulat*: nodurile din graf au acelasi grad
- nr de arce ale unui graf regulat: $p\Delta/2$

# Grafuri

- *drum* (*cale*) intre doua noduri $x, y \in V$: secventa de noduri $x_0$, $x_1$, …, $x_k$ a.i. $x = x_0$ si $y = x_k$ si $(x_i, x_{i+1}) \in E, \forall\, 0 \leq i < k$
- drum *elementar*: nodurile $x_1$, …, $x_{k-1}$ sunt distincte
- in arhitecturi NORMA, un drum este o cale de comunicatie intre doua procesoare
- *lungimea* unui drum = nr de arce ($k$ in notatia de mai sus)
- *distanta* intre doua noduri *d(x,y)* = lungimea celui mai scurt drum dintre ele
- *diametrul* grafului ($D$) = $\max_{i,j} d(i,j)$ unde $i, j \in V$
  - i.e., distanta dintre cele mai indepartate noduri din graf

# Grafuri

- *ciclu*: drum elementar in care nodul initial si cel final coincid
- *ciclu hamiltonian*: ciclu care trece exact o singura data prin fiecare nod al grafului
  - lungime egala cu ordinul grafului
- graf *conex*: oricare doua noduri sunt legate prin cel putin un drum
- *arbore*: graf conex aciclic

# Dezideratе topologie hardware

- graf conex
- graf regulat cu grad mic: procesoare identice cu putine cai de comunicatie (usor de realizat)
- nr total de arce relativ mic (un arc revine la o conexiune fizica, electrica pe placa sau intre placi)

# Inelul

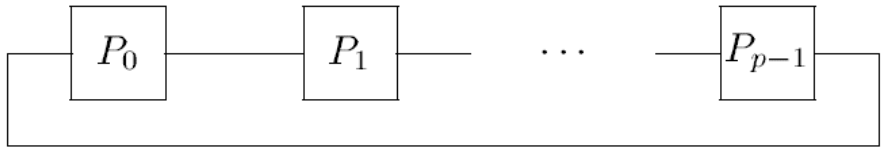
- procesorul $P_i$ legat de $P_j$ unde

$$j = (i-1)\ mod\ p \quad (P_j \text{ e vecinul din stanga})$$

$$sau$$

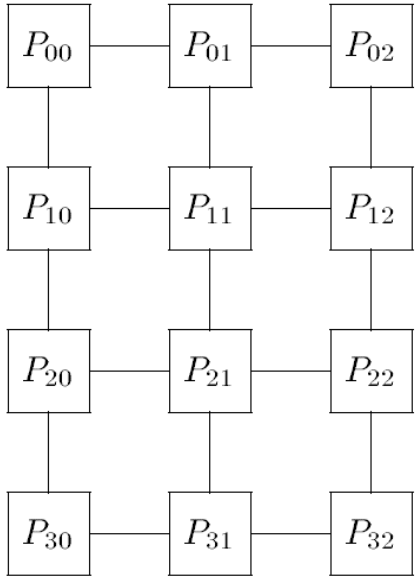
$$j = (i+1)\ mod\ p \quad (P_j \text{ e vecinul din dreapta})$$

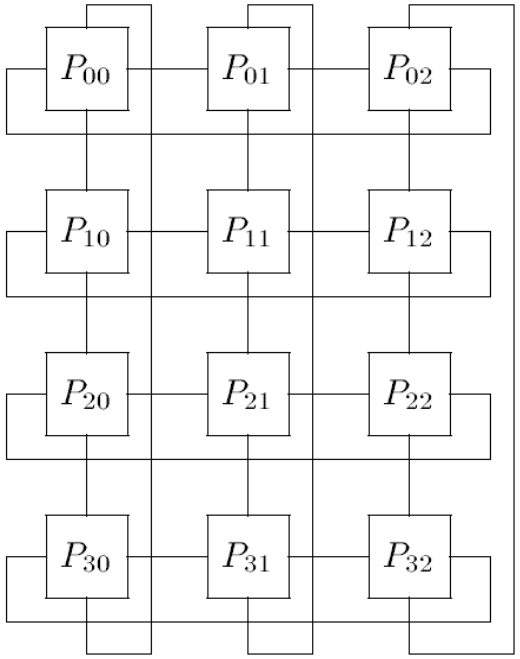
- grad: $\Delta = 2$
- nr de arce = $p$
- diametru D = $floor(\frac{p}{2})$



# Grid (tor)

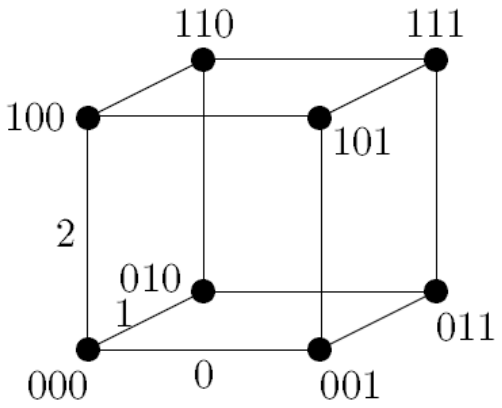
- grid patrat: $\sqrt{p} * \sqrt{p}$
- grad: $\Delta = 4$
- nr de arce = $2p$
- diametru D = $2 * floor(\frac{p}{2})$

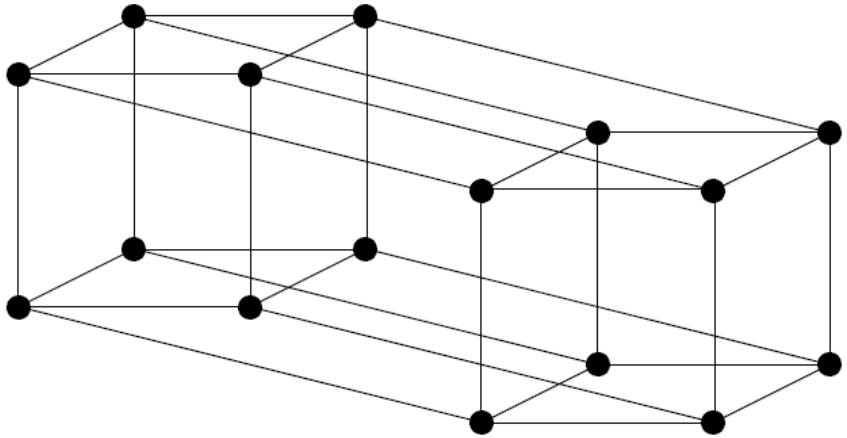


# Hipercub

- topologie avantajoasa pentru unii algoritmi
- $H_d$ = hipercub de dimensiune $d$, cu $p = 2^d$ procesoare
- fiecare procesor are $d$ vecini
- $P_i$ legat de $P_j$ daca reprezentarile binare ale lui $i$ si $j$ difera printr-un singur bit



# Hipercub (cont.)

- grad: $\Delta = d = log(p)$
- nr de arce = $2p$
- diametru $D = p$
- $d(i,j)$ = distanta Hamming dintre codurile binare ale lui $i$ si $j$
- definitie recursiva: $H_d$ compus din doua sub-cuburi $H_{d-1}$ unite prin arce care conecteaza nodurile din aceeasi pozitie in sub-cuburi
  - nodurile din primul sub-cub primesc 0 in cea mai semnificativa pozitie a reprezentarii lor binare, iar cele din al doilea sub-cub 1

# Sumar topologii

| Topologie | Δ | D |
|---|---|---|
| Inel | *2* | $O(p)$ |
| Grid | *4* | $O(\sqrt{p})$ |
| Hipercub | $log\ p$ | $O(\log p)$ |