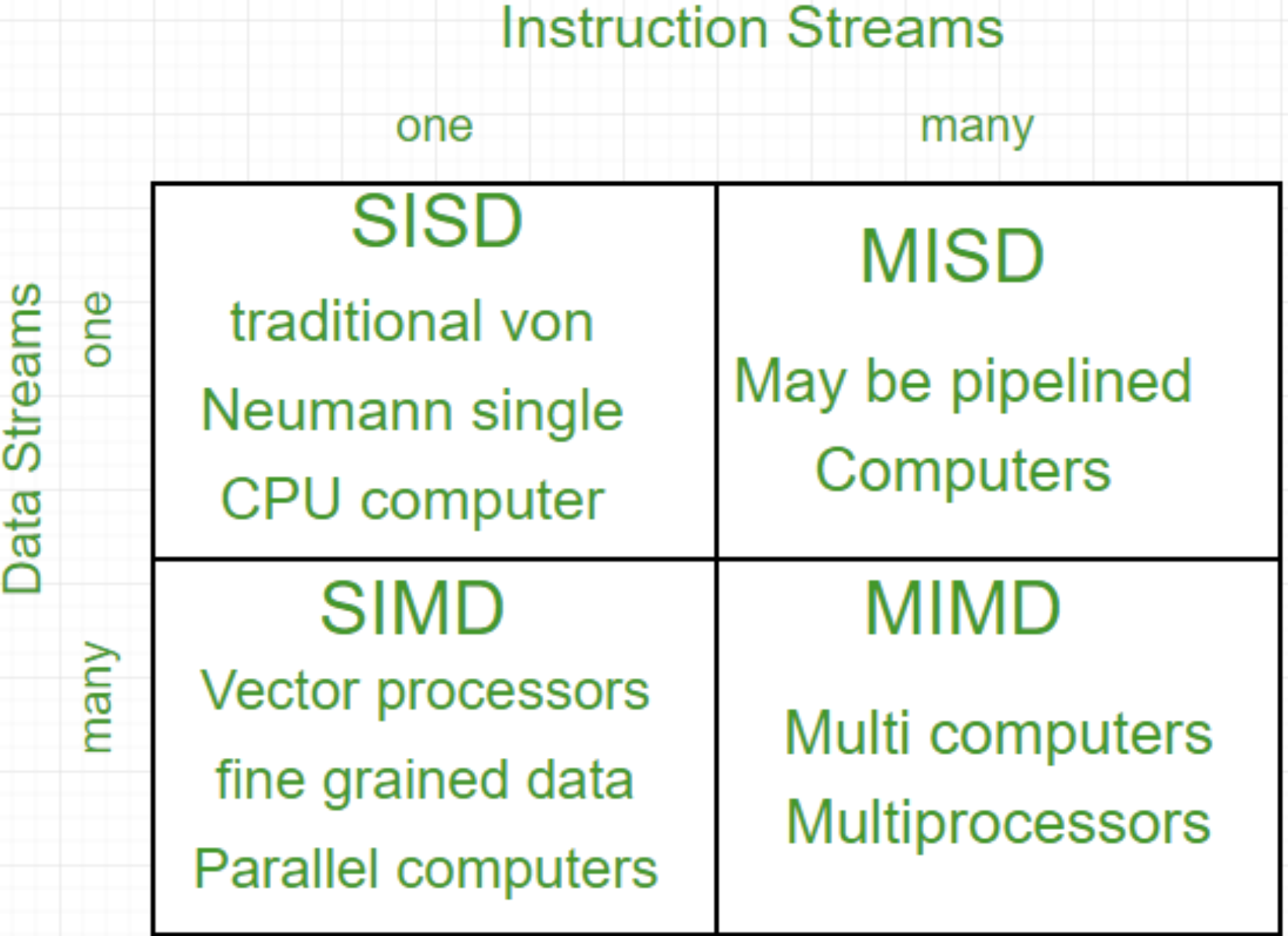
Teorie PPD

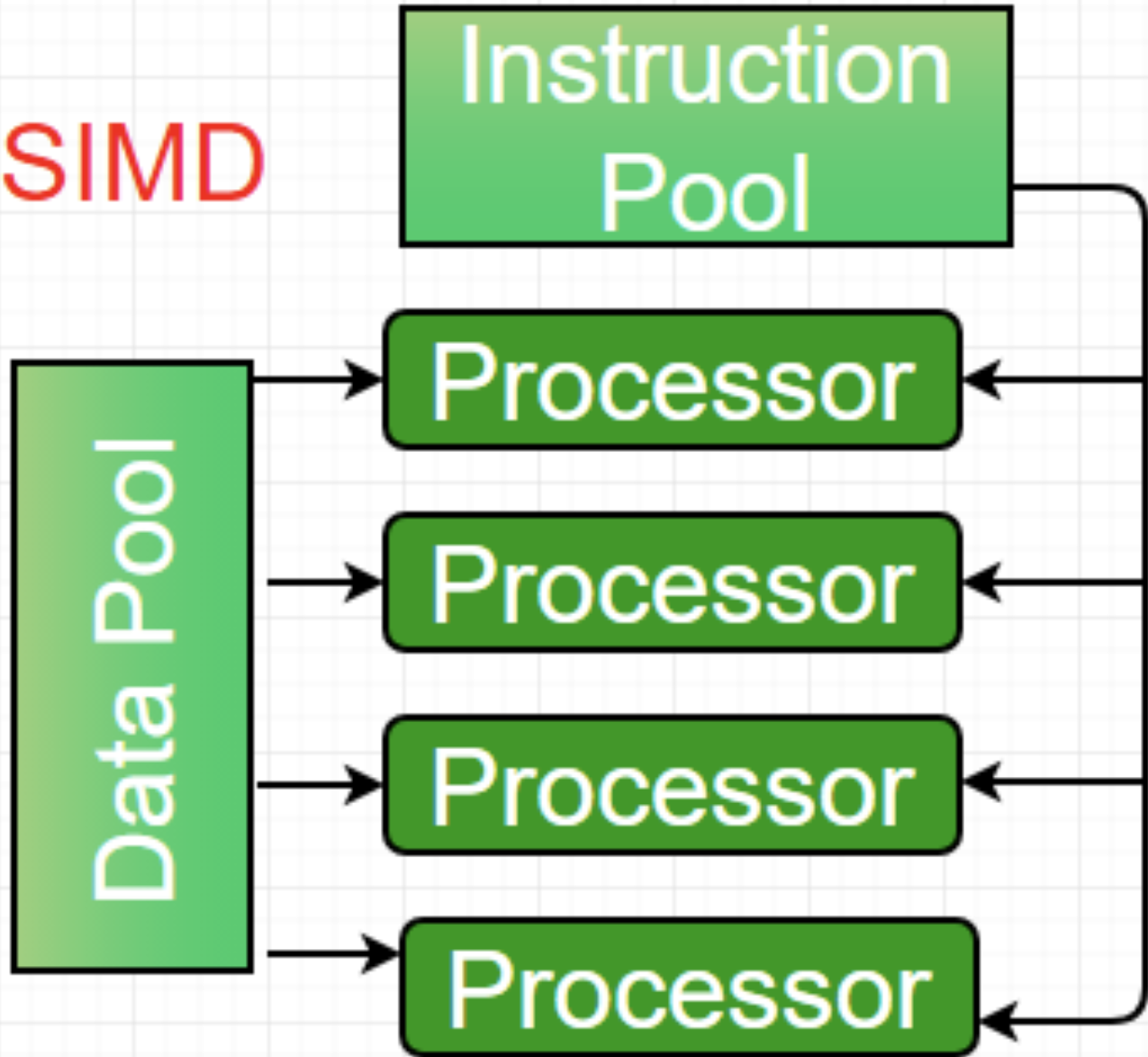
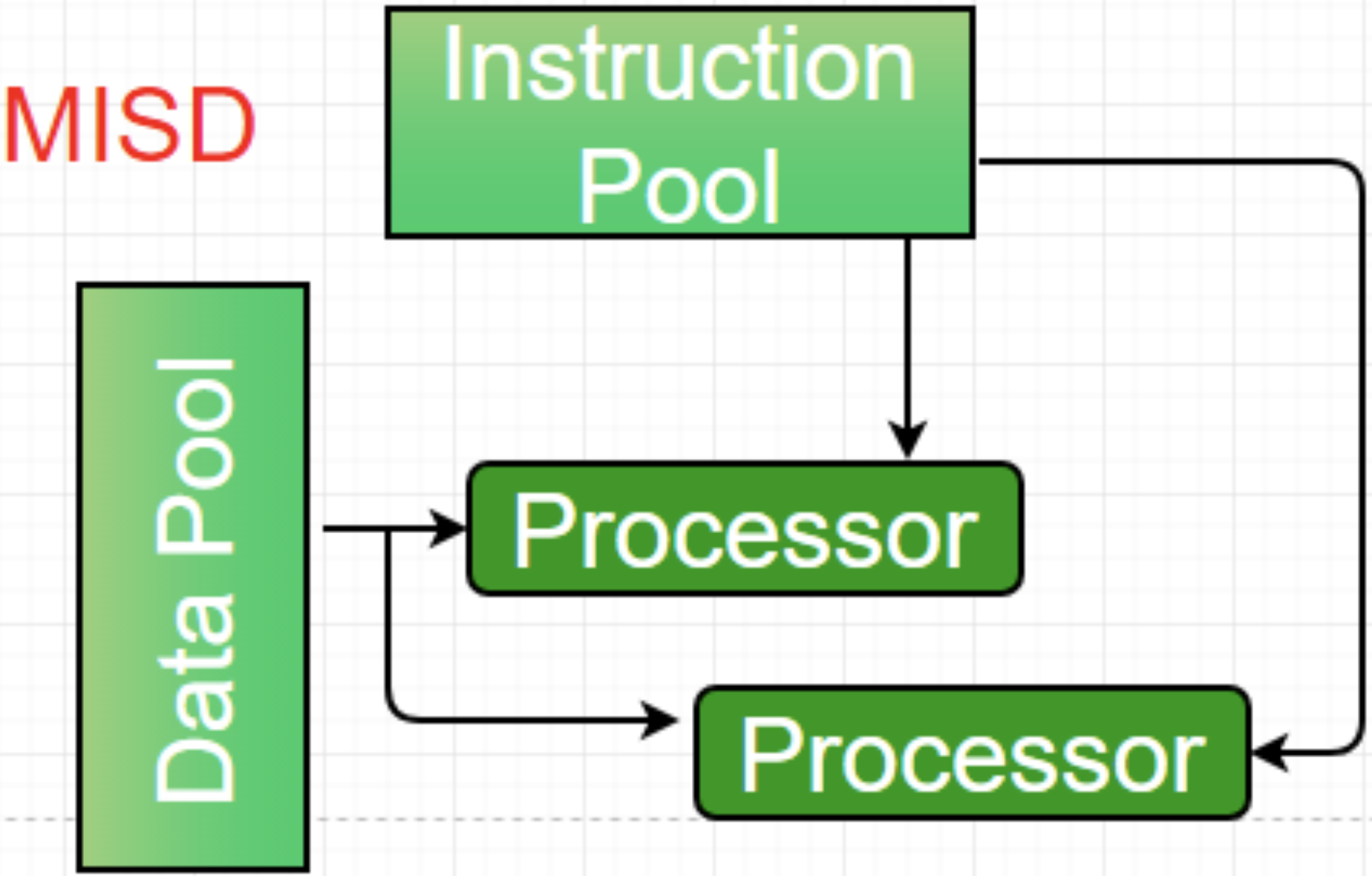
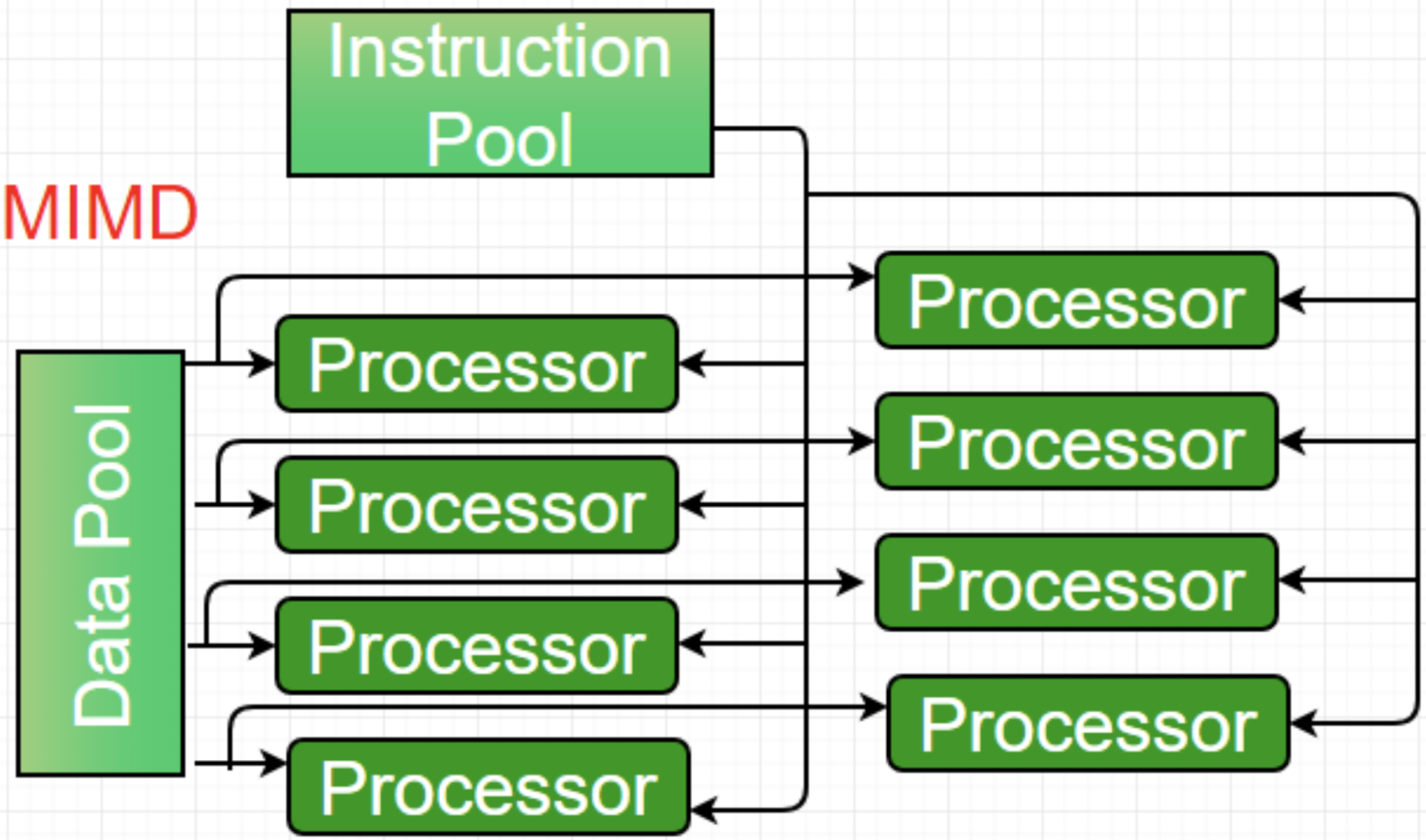
# Clasificarea Flynn

Calcul paralel = împărțirea proiectului în părți mai mici, iar toate părțile lucrează în același timp.

- fiecare parte a programului este conține instrucțiuni sau bucăți de date, iar cei care efectuează partea respectivă se numesc procesoare.

- instrucțiunile din fiecare parte sunt executate simultan pe diferite unități centrale de procesare (CPU).

Aici intervine Taxonomia lui Flynn. Aceasta împarte sistemele de calcul în 4 categorii. Fiecare categorie descrie cum procesează instrucțiunile și datele.

1. SISD (Instrucțiune unică, Date unice)
   1. ex.: o singură persoană care lucrează la un proiect de la început până la sfârșit, sarcină cu sarcină, fără ajutor - majoritatea calculatoarelor obișnuite.
   2. funcționează pe rând și este mai lentă pentru sarcini complexe.
2. SIMD (Instrucțiune unică, Date multiple)
   1. ex.: membrii unei echipe care fac același lucru în același timp, dar fiecare persoană are propriul set de date (fiecare taie câte o parte diferită dintr-un buștean) - calculatoare folosite în știință, care fac multe calcule cu vectori și matrice.
   2. ok pentru operațiuni care repetă aceleași instrucțiuni pe date diferite.
3. MISD (Instrucțiuni multiple, Date unice)
   1. ex.: mai multe persoane fac lucruri diferite pe același material (fiecare analizează un buștean din perspective diferite: unul măsoară, altul desenează, altul îl taie).
   2. nu este comun și nici eficient în cele mai multe cazuri.
4. MIMD (Instrucțiuni multiple, Date multiple)
   1. ex.: fiecare persoană lucrează la sarcini diferite pe materiale diferite (unul construiește un perete, altul montează ușa) - calculatoarele moderne și supercalculatoare, ca sistemele de la IBM.
   2. cel mai flexibil model și poate face orice tip de sarcină.
   3. MIMD cu memorie partajată - toți lucrează pe aceeași foaie (un singur set de date). Dacă cineva schimbă ceva, toată lumea vede schimbarea. (ușor de coordonat, dacă se strică ceva, toată echipa este afectată)
   4. MIMD cu memorie distribuită - fiecare are foaia proprie și lucrează independent, Comunicarea între membrii echipei se face prin mesaje. (greu de coordonat, mai sigur și mai ușor de extins)
   5. MIMD hibrid (aparent există și asta)

# UMA vs NUMA - shared memory

UMA și NUMA sunt două concepte legate de modul în care procesoarele dintr-un sistem multiprocesor (MIMD) accesează memoria.

### UMA (Uniform Memory Access)

* 1. toate procesoarele au acces egal la aceeași memorie. Fiecare procesor poate ajunge la orice parte a memoriei cu aceeași viteză.
  2. simplu de gestionat, ușor de programat, performanță previzibilă, deoarece toate procesoarele au acces egal la memorie.
  3. nu este scalabil pentru un număr mare de procesoare.
  4. exemple de sisteme: calculatoare cu memorie partajată (SMP - Symmetric Multi-Processing).

### NUMA (Non-Uniform Memory Access)

* 1. fiecare procesor are propria sa memorie locală (mai rapidă) și poate accesa și memoria altor procesoare (mai lent). Accesul la memorie nu este egal din punct de vedere al vitezei.
  2. ex.: o echipă unde fiecare persoană are propriul birou cu notițele sale (memoria locală). Dacă cineva are nevoie de informații de la altcineva, trebuie să meargă la biroul colegului, ceea ce durează mai mult.
  3. este scalabil, performanță mai bună pentru aplicații care folosesc doar memoria locală.
  4. mai dificil de programat (trebuie gestionat ce memorie accesează fiecare procesor), accesul la memoria altor procesoare este mai lent.
  5. exemple de sisteme: calculatoare cu memorie distribuită (supercomputere, servere mari).

Cache Coherent (Coerența cache) - asigură că toate procesoarele dintr-un sistem multiprocesor (MIMD) au o versiune actualizată a datelor din cache.

Dacă un procesor schimbă o valoare în memoria cache, celelalte procesoare trebuie să știe de schimbare, pentru a evita inconsistențele. (ca atunci când mai multe persoane folosesc o copie a aceluiași document. Dacă cineva modifică documentul, toți ceilalți trebuie să primească actualizarea.)

Latency (Latența) - timpul în care o dată ajunge la procesor după ce s-a inițiat cererea.

Latența mică înseamnă un sistem mai rapid. (ca timpul de așteptare după ce ai dat o comandă la restaurant. Cu cât aștepți mai puțin, cu atât ești mai mulțumit.)

Bandwidth (Lățime de bandă) - cantitatea de date care poate fi transferată între două componente (procesor și memorie) într-un anumit timp.

O lățime de bandă mai mare permite transferuri mai rapide și mai multe date procesate simultan. (ca o autostradă - cu cât mai multe benzi, cu atât mai multe mașini pot trece în același timp.)

# SMP (Symmetric Mulit-Processing)

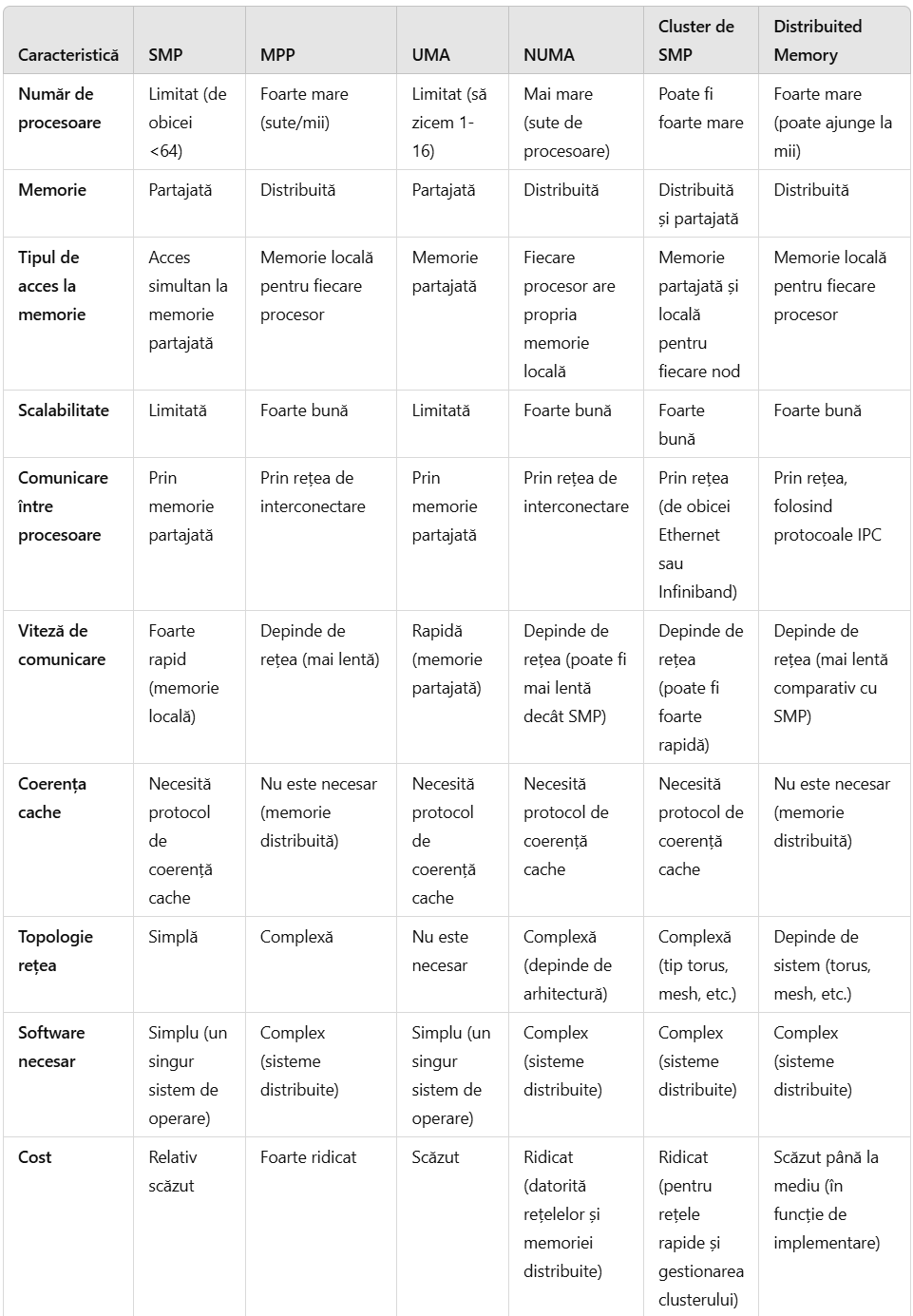
Un **SMP** este un **tip de arhitectură** paralelă în care mai multe procesoare funcționează împreună, având acces egal la o memorie partajată și la un set comun de resurse.

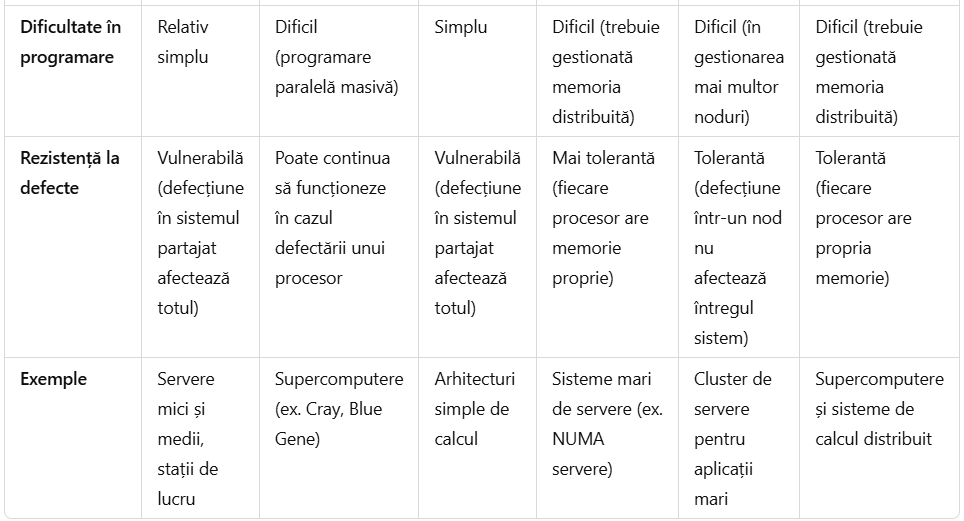
* 1. Toate procesoarele accesează **aceeași memorie principală (RAM)**
  2. Procesoarele **comunică** între ele **prin memorie**, fără a fi nevoie de rețele complexe (problema: pot intra în competiție pentru acces la memorie pe măsură ce numărul de procesoare crește -> contention)
  3. Un singur sistem de operare controlează toate procesele, ceea ce face gestionarea mai simplă decât în arhitecturile distribuite
  4. Sarcinile sunt **partajate**, astfel fiecare procesor poate prelua orice sarcină din coada de procese, deoarece toate au acces la aceleași resurse
  5. Fiecare procesor are **propriul cache** pentru a accelera accesul la datele frecvent utilizate
  6. Sistemele SMP folosesc **protocoale de coerență cache** pentru a asigura că datele stocate în cache-urile procesoarelor sunt consistente cu cele din memorie

# MPP (Massively Parallel Processor)

Un **MPP** este un **tip de arhitectură** paralelă care folosește un **număr foarte mare de procesoare** pentru a rezolva o problemă de calcul complexă. Procesoarele lucrează **independent**, dar colaborează printr-un mecanism de comunicare bine coordonat.

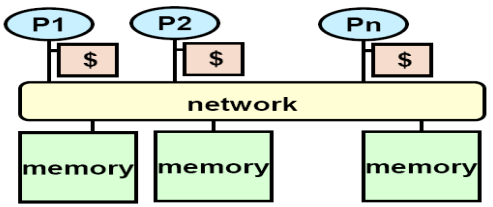
* 1. Fiecare procesor are **memoria sa locală**
  2. Procesoarele comunică între ele prin intermediul unei **rețele de interconectare** foarte rapide
  3. Problema este împărțită în multe **sub-probleme mai mici**, fiecare procesor lucrând pe o parte specifică a datelor.
  4. Sarcinile sunt **independente**, reducând necesitatea comunicării frecvente între procesoare
  5. **Performanță îmbunătățită** datorită faptului că fiecare procesor are memorie proprie și sincronizarea este redusă la minim



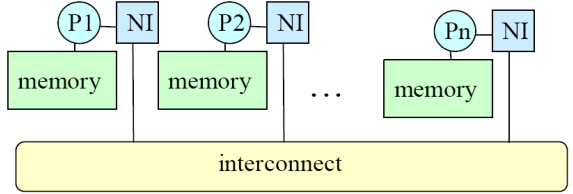


# Modele de programare

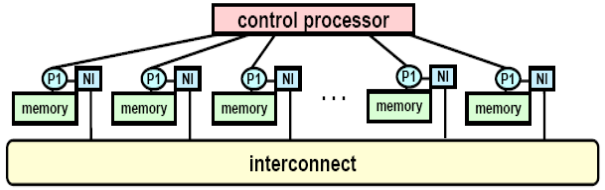
### Spațiu Partajat de Adrese - mai multe procesoare sau noduri dintr-un sistem au acces la aceeași zonă de memorie. Folosit în SMP și UMA. Pot apărea probleme de coerență a datelor (cache), deoarece fiecare procesor ar putea avea o copie locală a acelorași date.



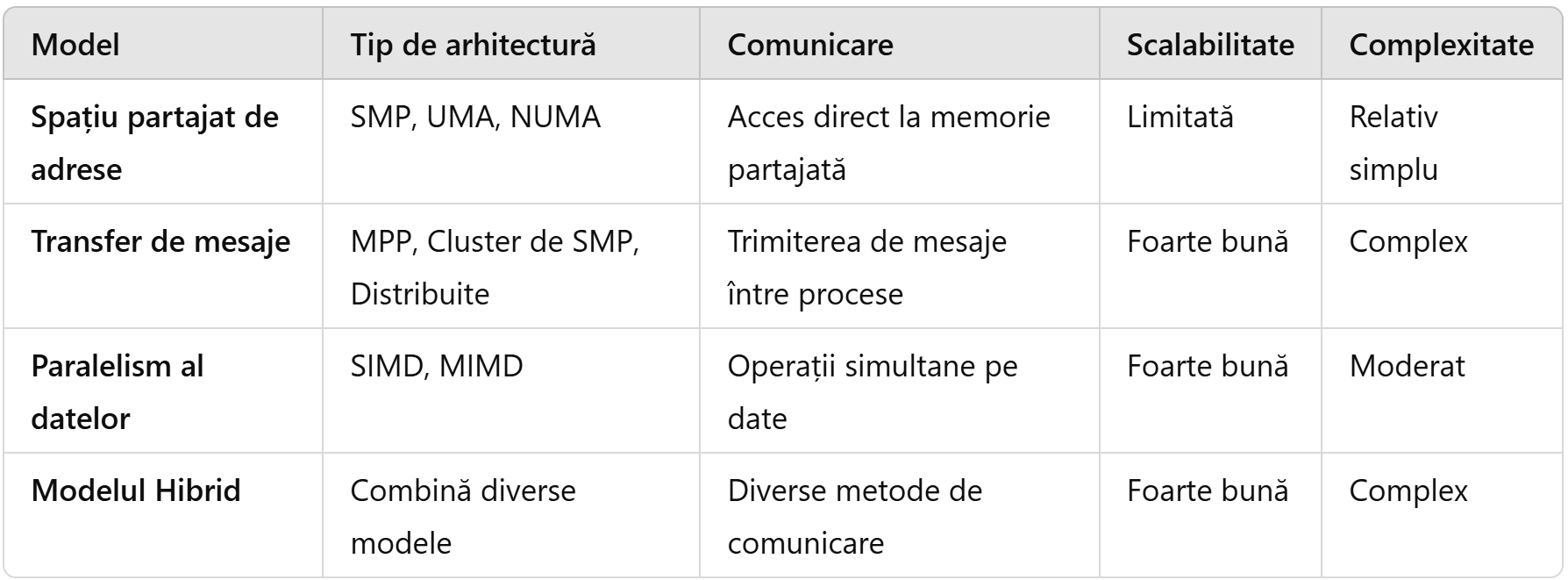
### Transfer de Mesaje - procesele care rulează pe diferite procesoare sau noduri dintr-un sistem comunică între ele prin mesaje, fără a împărți memoria. Folosit în MPP și Cluster de SMP. Este foarte scalabil.



### Paralelism al Datelor - aceleași operații sunt aplicate simultan pe diferite elemente ale unui set de date. Folosit în SIMD. Este scalabil. Sincronizare între procese redusă, overhead de comunicație mic.



### Modelul hibrid - combină două sau mai multe modele de programare pentru a profita de avantajele fiecăruia. De exemplu, poate combina paralelismul datelor cu transferul de mesaje, pentru a maximiza eficiența și scalabilitatea.



Scheduler

- un program care controlează execuția proceselor

- setează stările procesului: new, running, blocked, ready, terminated.

Context switch

- procesul de salvare a stării unui proces și comutarea la alt proces.

- ex.: trecerea de la un editor de text la un joc în execuție.

Threads (Fire de execuție)

- unități mici de execuție ale unui proces care împart resursele acestuia.

- permite rularea în paralel a mai multor sarcini dintr-un proces.

- ex.: un browser poate avea un thread pentru redarea video și altul pentru încărcarea paginii.

Race condition (Condiție de competiție)

- o problemă care apare când mai multe fire de execuție (threads) accesează simultan o resursă partajată fără sincronizare.

- critical - atunci când ordinea în care se modifică variabilele interne determină starea finală a sistemului.

- non-critical - atunci când ordinea în care se modifică variabilele interne NU are impact asupra stării finale a sistemului.

Data Race

- un tip de race condition când două sau mai multe fire de execuție accesează aceeași locație de memorie simultan, iar unul o modifică.

- ex.: două thread-uri actualizează aceeași variabilă fără coordonare.

Secțiune critică

- o parte din cod care accesează resurse partajate și trebuie executată exclusiv de un singur thread la un moment dat.

- previne problemele de *data race*

- ex.: incrementarea unui contor global.

Instrucțiune atomică

- operație complet executată fără întreruperi

- asigură consistența datelor partajate

# MPI

MPI\_INIT - inițializare mediu

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

MPI\_COMM\_SIZE - determină numărul de procese din grupul asociat unui com.

* MPI\_COMM\_SIZE(comm, &size)
* MPI\_COMM\_SIZE(comm, size, ierr)

MPI\_COMM\_RANK - determină rangul procesului apelant în cadrul unui com.

* MPI\_COMM\_RANK(comm, &rank)
* MPI\_COMM\_RANK(comm, rank, ierr)

MPI\_ABORT - oprește toate procesele asociate unui comunicator

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

MPI\_SEND(&buffer, count, type, dest, tag, comm)

MPI\_RECV(&buffer, count, type, source, tag, comm, &status)

MPI\_BARRIER(comm, ierr)

MPI\_BCAST(&msg, count, MPI\_INT, source, MPI\_COMM\_WORLD)

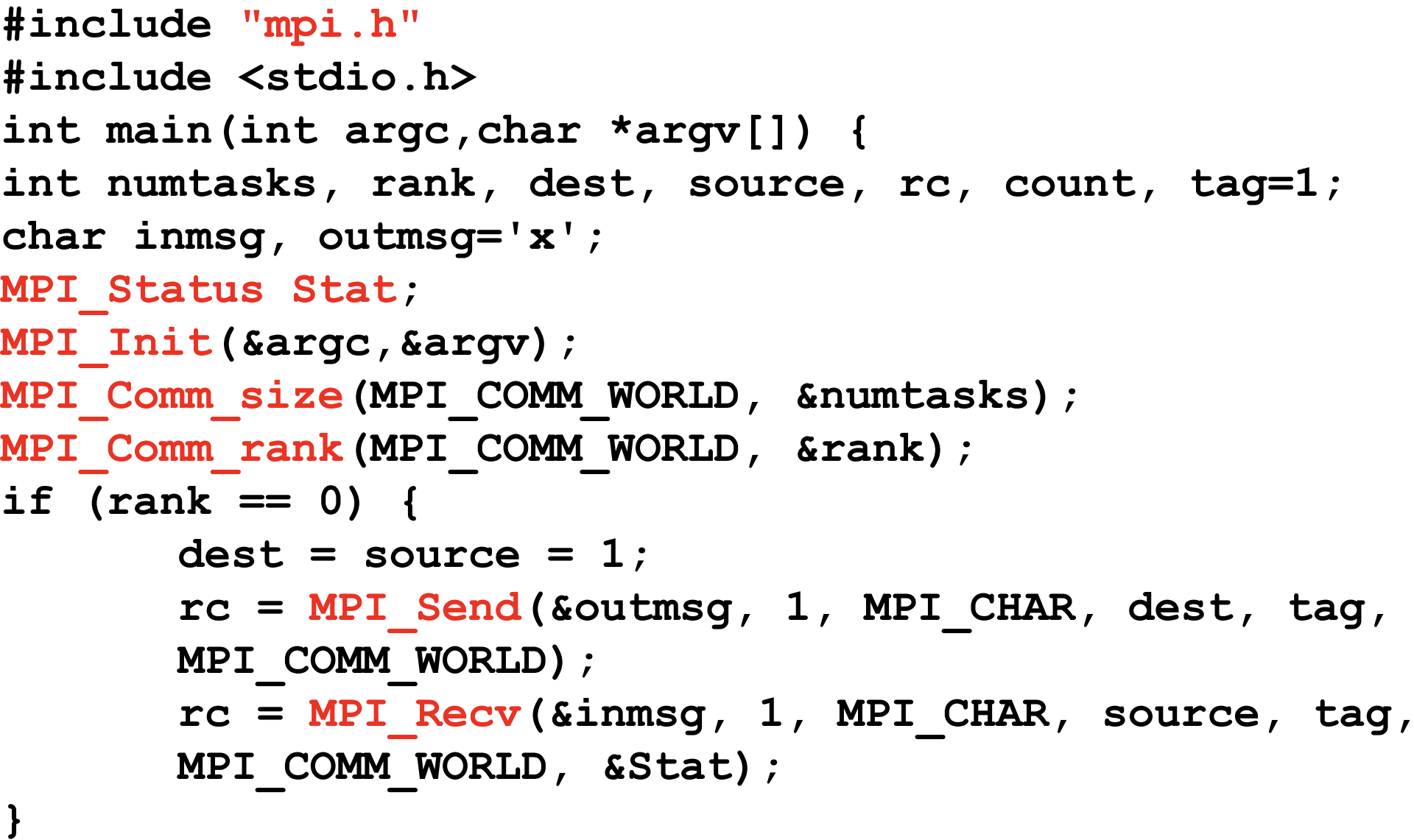
MPI\_SCATTER(sendbuf, sendcnt, MPI\_INT, recvbuf, recvcnt, MPI\_INT, src, MPI\_COMM\_WORLD)

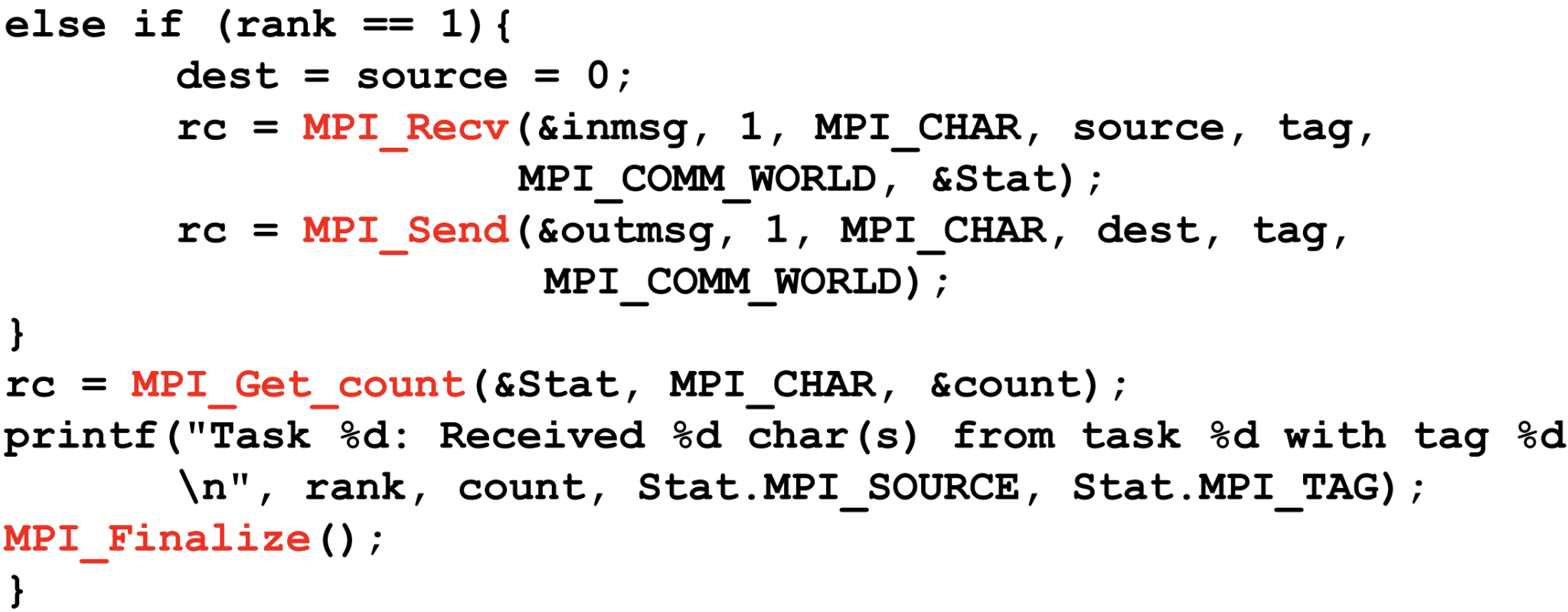
MPI\_GATHER(sendbuf, sendcnt, MPI\_INT, recvbuf, recvcnt, MPI\_INT, src, MPI\_COMM\_WORLD)

MPI\_REDUCE(sendbuf, recvbuf, count, MPI\_INT, MPI\_SUM, dest, comm)

MPI\_Finalize - finalizare mediu MPI

* MPI\_FINALIZE()
* MPI\_FINALIZE(ierr)





# Mecanisme de sincronizare

## Semafor

* mecanism de sincronizare folosit pentru a controla accesul la resurse partajate într-un mediu concurent
* o variabilă care poate lua valori întregi și este accesată prin două operații principale:

- wait() / P() - scade valoare semaforului cu 1. Dacă rezultatul este negativ, thread-ul/procesul este blocat până când semaforul devine pozitiv.

- signal() / V() - crește valoarea semaforului cu 1. Dacă există thread-uri/procese blocate, le deblochează.

Semafor binar

* semafor care poate avea doar două valori: 0 (blocare) sau 1 (acces).
* similar cu un mutex.
* este folosit pentru protejarea unei secțiuni critice astfel încât doar un thread/proces să o acceseze la un moment dat.

Semafor starvation-free

* semafor care garantează că toate thread-urile/procesele care așteaptă accesul la o resursă vor fi servite, prevenind **starvation**
* utilizează o coadă FIFO pentru a asigura ordinea corectă de acces.

Semafor slab (Weak Semaphore)

* semafor care nu garantează ordinea în care thread-urile/procesele sunt servite.
* poate apărea **starvation**, deoarece un thread cu prioritate mică poate aștepta la nesfârșit.

Semafor puternic (Strong Semaphore)

* semafor care garantează servirea thread-urilor **în ordinea** în care au solicitat accesul (**FIFO**).
* elimină starvation.

## Monitor

* mecanism de sincronizare în programarea concurentă care gestionează **accesul sigur** al mai multor thread-uri **la resurse partajate**.
* combină două elemente esențiale:
  + mutual exclusion (excluderea mutuală) - permite accesul la secțiunea critică doar unui thread la un moment dat
  + condition variables (variabile condiționale) - permite thread-urilor să aștepte anumite condiții înainte de a continua execuția
* exemplu simplu: o bancă cu un singur ghișeu pentru clienți.
  + *resursa partajată* - ghișeul băncii
  + *monitor* - manager care decide cine poate folosi ghișeul
  + *thread-uri* - clienții care așteaptă la coadă
  + *excludere mutuală* - doar un client poate interacționa la ghișeu la un moment dat
  + *variabile condiționale* - dacă ghișeul este închis sau nu sunt bani în casierie, clientul trebuie să aștepte până când condiția se schimbă.
* elementele unui monitor
  + lock-uri interne - asigură că doar un thread execută codul monitorului la un moment dat
  + variabile condiționale - permite thread-urilor să aștepte până când o anumită condiție este satisfăcută
    - wait() - blochează thread-ul până când este notificat
    - signal() - notifică un thread care așteaptă că poate continua
* disciplina de semnalizare în monitoare
  + semnalizare înainte (Signal-and-Continue) - thread-ul care face signal() continuă execuția, iar thread-ul notificat așteaptă.
  + semnalizare după (Signal-and-Wait) - thread-ul care face signal() se blochează, iar thread-ul notificat preia controlul imediat.
* în **Java**, fiecare obiect poate fi folosit ca monitor, deoarece Java oferă sincronizare implicită cu **synchronized**.

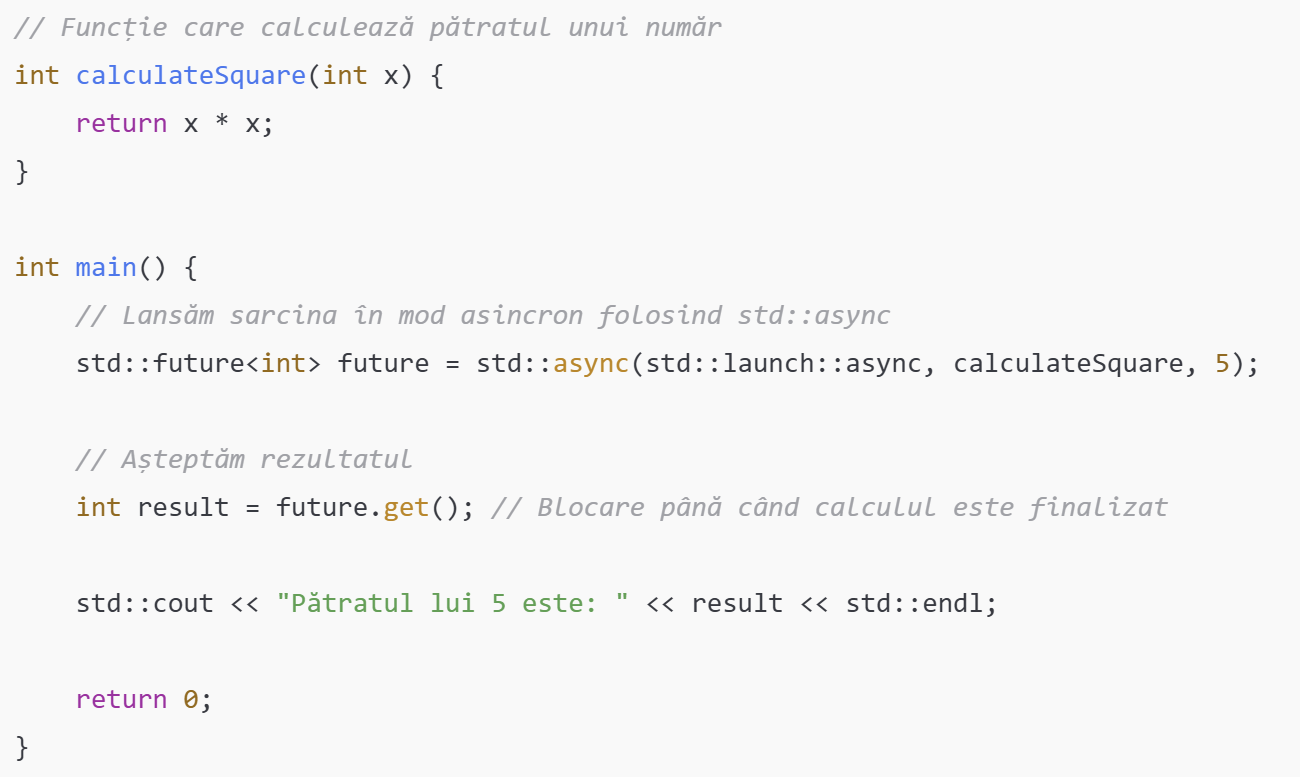
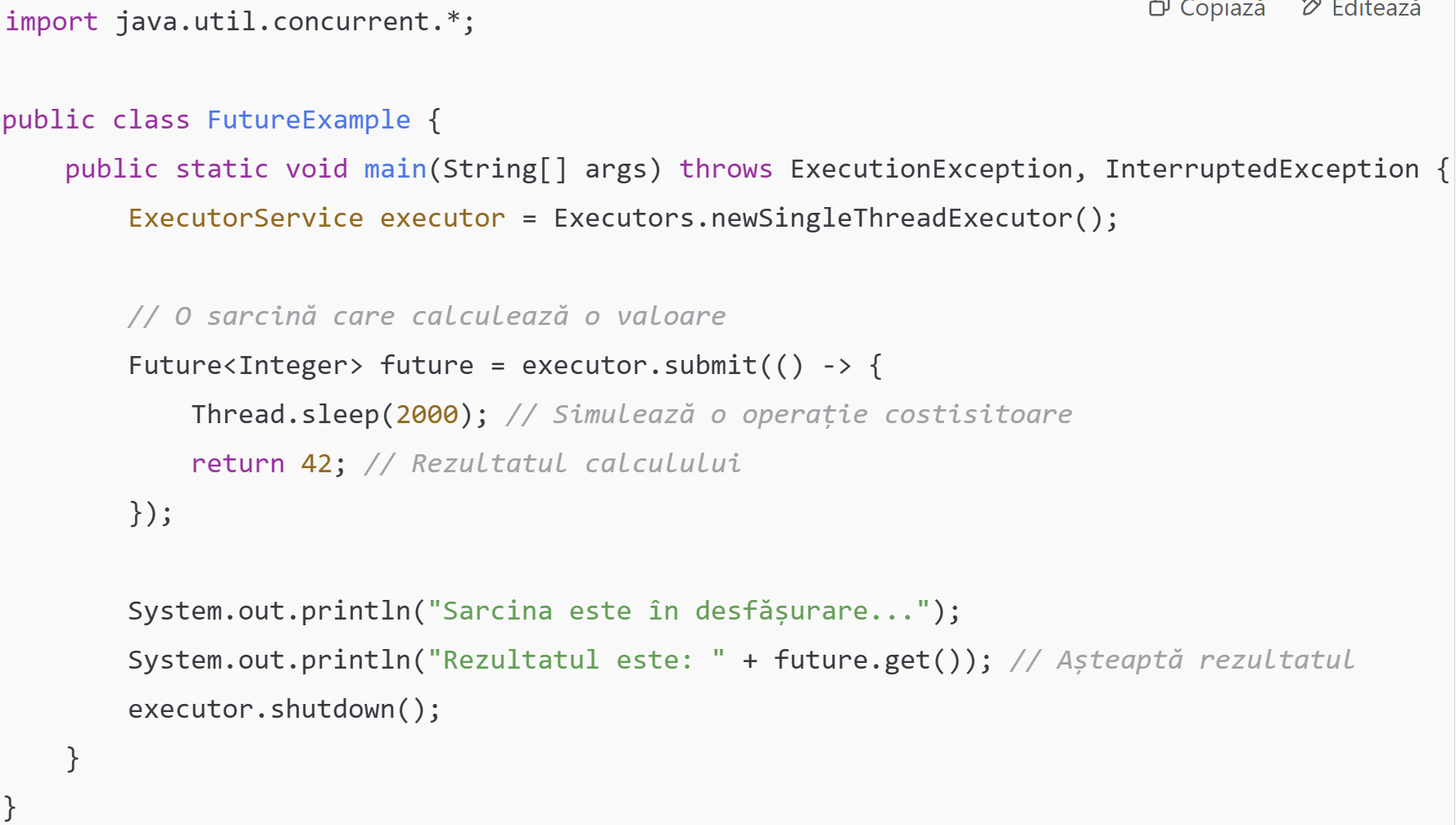


* în **C++**, monitoarele **nu sunt implementate direct**, dar pot fi simulate folosind mutex-uri și condition variables din biblioteca standard <mutex> și <condition\_variable>.



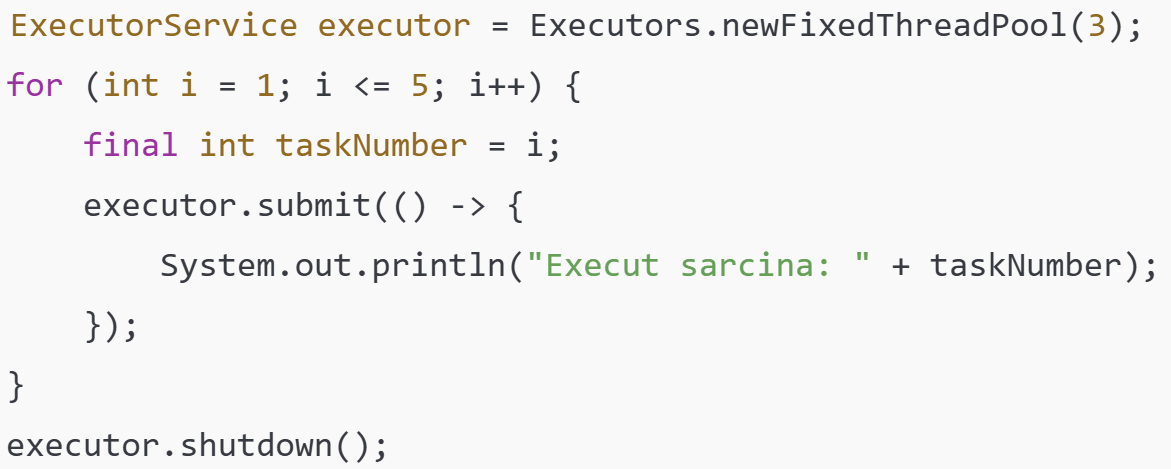
## Future

* **mecanism** care permite **executarea unei sarcini în fundal** și accesarea rezultatului ei mai târziu.
* ca și cum ai comanda o pizza (sarcina ta) și ți s-ar da un bon (Future) cu care poți verifica dacă e gata comanda sau să aștepți până e livrată.



## Executor

* o **abstracție** care decuplează trimiterea unei sarcini de executarea ei. Creează și gestionează thread-uri pentru tine.
* un manager de echipă (Executor) primește task-uri și le distribuie membrilor echipei (thread-uri). Tu, ca superior, doar îi spui managerului ce să facă.



## Thread Pool

* un **grup de thread-uri reutilizabile**, creat pentru a gestiona eficient sarcinile paralele. Folosit pentru a evita costurile de creare și distrugere repetată a thread-urilor.
* elemente: un vector de thread-uri, o coadă de sarcini, variabile de sincronizare (mutex sau condition variables)
* fiecare thread din pool preia sarcini din coadă și le execută
* La un restaurant, bucătarii (thread-urile) sunt disponibili într-o echipă. Dacă vin mai multe comenzi (sarcini), fiecare bucătar preia una, iar tu nu trebuie să angajezi mai mulți bucătari pentru fiecare comandă nouă.

## Runnable vs. Callable

* interfețe care definesc sarcini de executat de un thread.
* Runnable - nu returnează un rezultat și nu aruncă excepții verificate. Spui cuiva să curețe casa și nu îți pasă ce rezultat obții.
* Callable - returnează un rezultat și poate arunca excepții verificate. Spui cuiva să calculeze suma cheltuielilor și aștepți să primești rezultatul.

# OpenMP

* este o **interfață de programare** pentru paralelizarea codului pe sisteme cu memorie partajată.
* **directivele** sale sunt adăugate folosind ***#pragma omp***
* #pragma omp parallel - regiune paralelă
  + creează o regiune paralelă, adică mai multe thread-uri sunt create pentru a executa codul conținut.
  + omp\_get\_thread\_num() returnează ID-ul thread-ului curent.
  + numărul total de thread-uri este controlat prin omp\_set\_num\_threads() sau variabila de mediu OMP\_NUM\_THREADS.
* #pragma omp for
  + distribuie iterările unui for între thread-uri. Este folosit în interiorul unei regiuni paralele.
  + clauze utile:
    - schedule(static) - fiecare thread primește un număr egal de iterări.
    - schedule(dynamic, chunk) - thread-ul primește câte un set de chunk iterări, dar distribuția este dinamică.
    - schedule(guided) - dimensiunea chunk scade pe măsură ce sarcina avansează.
* #pragma omp sections
  + permite mai multor secțiuni independente de cod să fie executate în paralel
  + clauze utile:
    - default(private) - variabile private pentru fiecare thread
    - default(shared) - variabile comune între thread-uri
* #pragma omp barrier
  + sincronizează toate thread-urile. Niciun thread nu trece de barieră până când toate thread-urile nu au ajuns la aceasta.
* #pragma omp critical
  + definește o secțiune critică, în care doar un singur thread poate intra la un moment dat.
* clauze comune:
  + private(var) - fiecare thread are propria copie a variabilei
  + shared(var) - variabila este partajată între toate thread-urile
  + firstprivate(var) - inițializează variabila cu valoarea sa din thread-ul principal
  + lastprivate(var) - actualizează variabila cu valoarea finală obținută de ultimul thread care o accesează
  + reduction(operator: *variable list*) - operator: +, \*, -, &, |, ^, &&, ||.





**Explicații pe pași:**

1. **default(none)**:
   * Folosit pentru a forța declararea explicită a variabilelor ca **shared** sau **private**, ceea ce previne erori.
   * **De ce?** Asigură că programatorul controlează distribuția variabilelor între thread-uri.
2. **shared și private**:
   * **shared(a, b, n, sum)**: Variabilele sunt comune între thread-uri, toate thread-urile văd aceleași valori.
   * **private(prod)**: Fiecare thread are propria copie a lui prod.
   * **firstprivate(prod)**: Fiecare thread își inițializează prod cu valoarea lui din thread-ul principal.
3. **#pragma omp sections**:
   * Permite executarea a două calcule independente:
     + Produsul elementelor din a.
     + Suma elementelor din b.
   * **De ce?** Ideal pentru sarcini independente care pot fi efectuate în paralel.
4. **#pragma omp for cu schedule(dynamic, chunk)**:
   * Distribuie iterările buclei în blocuri dinamice de dimensiune chunk.
   * **De ce?** Sarcinile sunt alocate pe măsură ce thread-urile termină, ceea ce ajută la echilibrarea încărcării.
5. **reduction(+:sum)**:
   * Combină rezultatele parțiale ale fiecărui thread pentru variabila sum folosind operatorul +.
   * **De ce?** Simplifică acumularea rezultatelor din toate thread-urile.
6. **#pragma omp critical**:
   * Asigură că afișarea calculului este protejată, astfel încât doar un singur thread să scrie în consolă la un moment dat.
   * **De ce?** Previne intercalarea mesajelor.
7. **#pragma omp barrier**:
   * Asigură că toate thread-urile au terminat execuția înainte de a trece mai departe.
   * **De ce?** Necesită sincronizare pentru a continua cu suma finală.
8. **#pragma omp atomic**:
   * Folosit pentru operații simple precum incrementarea, evitând secțiuni critice mai complexe.
   * **De ce?** Reduce overhead-ul pentru operații mici.

# Metode de evaluare a performanței

Timp de execuție

* timpul real necesar pentru ca un algoritm să ruleze pe un sistem

Complexitate de timp

* o estimare teoretică a cât durează un algoritm să se execute, în funcție de dimensiunea input-ului. Exprimată cu notația Big-O.

Timpul total de execuție

* Timpul util (timpul efectiv petrecut procesând sarcina)
* Overhead-ul (timpul suplimentar necesar pentru coordonarea proceselor) - timpul pierdut pentru coordonarea proceselor, comunicare între thread-uri, sincronizare.
* *Ttotal = Tutil + Toverhead*

Accelerarea (“speed-up”)

* speed-up arată cât de mult se îmbunătățește timpul de execuție când folosim mai multe resurse (ex. mai multe procesoare)
* *S(p) = Tserial / Tparalel*
* *Tserial* - timpul de execuție cu un singur procesor
* *Tparalel* - timpul de execuție cu *p* procesoare
* ex.: Dacă ai de spălat vase și singur îți ia o oră (*Tserial* = 60 min), iar, cu ajutorul unui prieten, îți ia 30 de minute (*Tparalel* = 30 min), speed-up-ul este *S(2) = 60 / 30 = 2.*

Eficiența

* arată cât de bine sunt utilizate resursele
* *E(p) = S(p) / p*, p - numărul de procesoare
* ex.: Dacă 2 persoane termină spălatul vaselor în 30 de minute (speed-up = 2), eficiența este *E(2) = 2 / 2 = 1* (*Resurse utilizate perfect!*).
* ex.: Dacă 3 persoane lucrează și durează tot 30 de minute (speed-up = 2), eficiența este *E(3) = 2 / 3 = 0.67* (*Resurse parțial irosite!*)

Legea lui Amdahl

* spune că accelerarea unui program paralelizat este limitată de partea care nu poate fi paralelizată.
* f: procentul programului care poate fi paralelizat
* 1 - f: partea care rămâne secvențială
* ex.: Dacă faci pizza:
* 80% din muncă poate fi împărțită (pregătirea ingredientelor).
* 20% e secvențial (coacerea în cuptor).
* Chiar dacă ai 4 prieteni care te ajută, timpul total e limitat de cei 20% din sarcina care nu poate fi împărțită.

Legea lui Gustafson

* spune că dacă dimensiunea problemei crește (ex. mai multe date de procesat), paralelismul devine mai eficient. Spre deosebire de Amdahl, se bazează pe scalabilitate.
* f: partea paralelizabilă
* ex.: Dacă organizezi o petrecere:
* Cu 4 prieteni, poți pregăti mai multe feluri de mâncare simultan, iar eficiența paralelismului crește odată cu numărul sarcinilor.

Costul

* resursele totale consumate pentru a executa o sarcină pe un sistem paralel.
* *T(p)*: timpul de execuție folosind *p* procesoare.
* ex.: Dacă ai de pictat un gard și angajezi 4 persoane:
* Dacă fiecare persoană lucrează timp de 2 ore, costul total este 2⋅4=8 ore-persoane.
* Dacă lucrezi singur timp de 8 ore, costul este același, dar timpul e mai lung.

Scalabilitate

* măsoară **capacitatea unui sistem paralel de a folosi eficient mai multe resurse (procesoare)** pentru a rezolva probleme mai mari sau mai complexe.
* Mod de evaluare - eficiența constantă: Scalabilitatea este bună dacă eficiența rămâne aproape constantă pe măsură ce adaugi procesoare.
* , *E(p)* - eficiența cu *p* procesoare
* ex.: Dacă faci pizza cu mai mulți prieteni și timpul total scade proporțional cu numărul lor (fiecare prieten își face partea eficient), sistemul e scalabil. Dacă timpul nu scade, deși ai mai mulți prieteni, atunci ai probleme de scalabilitate.

Granularitate

* măsoară **dimensiunea sarcinilor paralele** în raport cu overhead-ul de comunicare dintre procesoare.
  + Granularitate fină - sarcini mici, cu multe schimburi de informații între procesoare
  + Granularitate grosieră - sarcini mari, cu puține schimburi de informații
* ex.: Dacă speli vase cu prietenii:
* Granularitate fină: Împărțiți fiecare farfurie separat. Necesită multă coordonare.
* Granularitate grosieră: Fiecare spală câte un tip de veselă (ex. cineva farfurii, altcineva pahare).

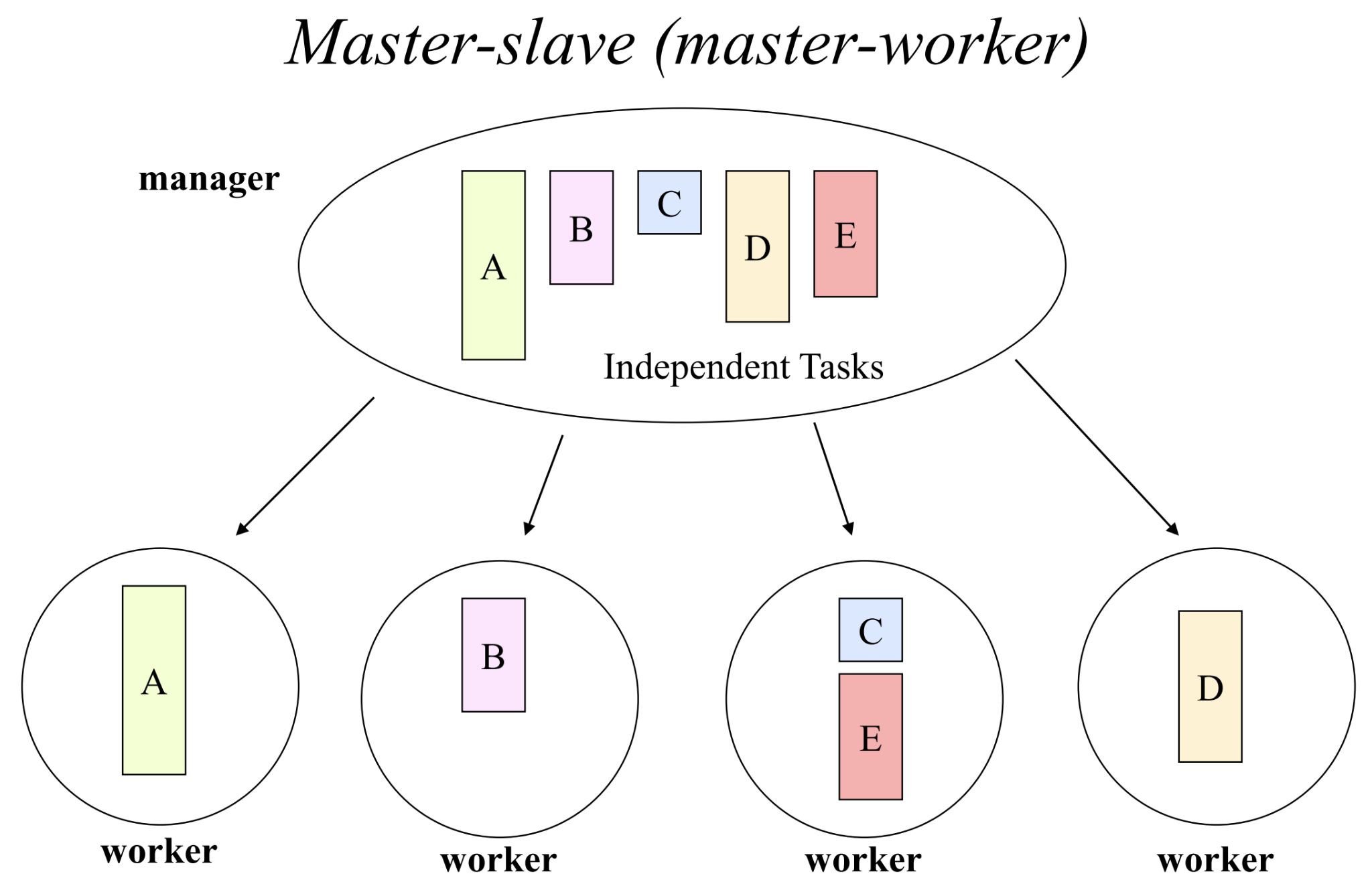
DOP (Grad de paralelism)

* numărul mediu de sarcini care pot fi executate în paralel la un moment dat.
* ex.: Dacă ai de mutat mobilă și sunt 4 prieteni disponibili:
* Dacă doar două piese de mobilier pot fi mutate simultan (deoarece coridorul este îngust), DOP este 2.
* DOP mai mare indică un sistem mai capabil să proceseze mai multe sarcini în paralel.

# Șabloane de proiectare paralelă

## Master-Slave

* un **master** împarte sarcinile în subtask-uri și le atribuie mai multor **slaves** (thread-uri). Slave-urile execută aceste subtask-uri și raportează rezultatele înapoi master-ului, care le combină pentru a obține soluția finală.
* ex.: un șef (master) care împarte sarcini angajaților (slaves). De exemplu, șeful cere fiecărui angajat să completeze o secțiune dintr-un raport, iar apoi șeful reunește toate secțiunile.
* Utilizare: procesarea imaginilor, calculul distribuției pe mai multe regiuni geografice, simulări, OpenMP.
* master-ul poate deveni un bottleneck dacă preia prea multe sarcini de coordonare.
* ineficient pentru probleme foarte mari sau complexe.

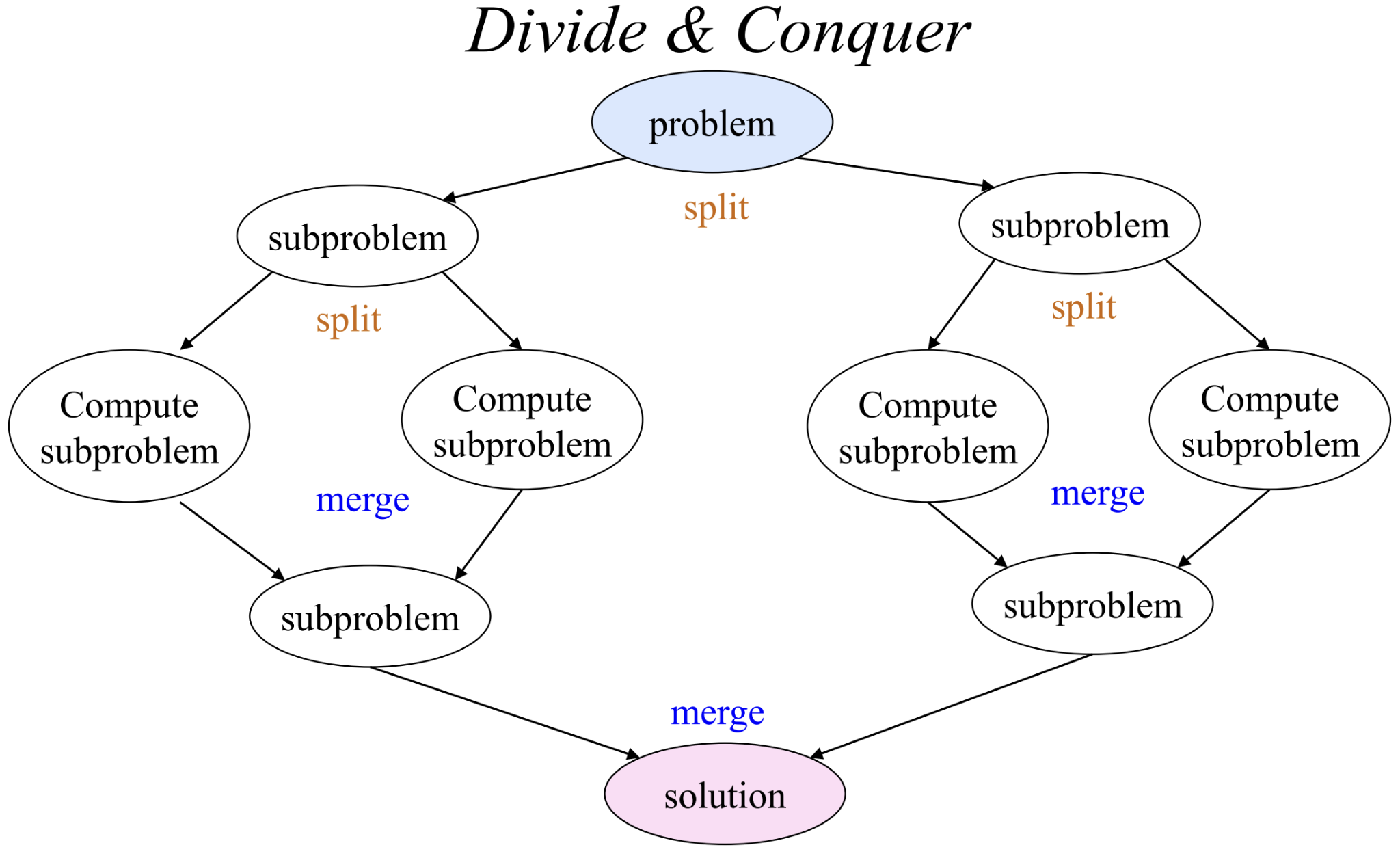


## Divide & Conquer

* problema mare este împărțită în probleme mai mici, fiecare dintre ele fiind rezolvată independent. Rezultatele subtask-urilor sunt apoi combinate pentru a forma soluția completă.
* ex.: Dacă ai o echipă de curățenie pentru o casă mare:

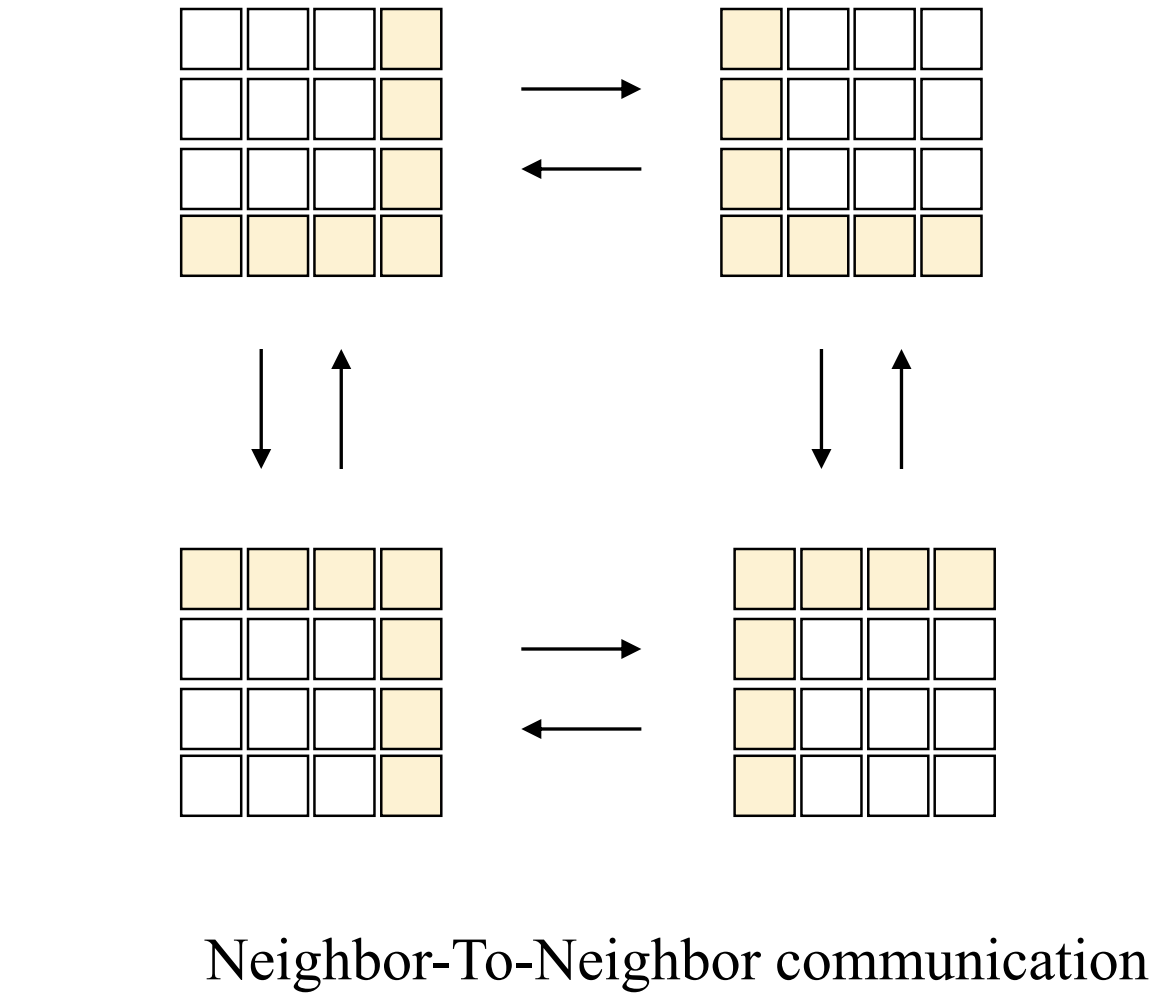
1. Împarți casa în camere.
2. Fiecare persoană curăță o cameră.
3. Rezultatele (camerele curate) sunt reunite pentru a curăța întreaga casă.

* Utilizare: Merge Sort, Quick Sort, FFT (Fast Fourier Transform), OpenMP.
* suprasarcină semnificativă pentru împărțirea și reunirea rezultatelor.
* necesită gestionarea sincronizării pentru a evita conflictele.
* foarte bun pentru probleme recursiv divizibile
* scalabilitate mare, deoarece problema poate fi divizată până la granularitate fină



## Geometric Decomposition

* problema este împărțită în funcție de **datele geometrice** sau **spațiale**. Fiecare thread primește o porțiune de date și lucrează doar pe acestea, comunicând periodic cu celelalte pentru a sincroniza rezultatele.
* ex.: o hartă a unui oraș împărțită în cartiere. Fiecare echipă repară drumurile dintr-un cartier, iar la final se coordonează pentru a verifica că drumurile conectează corect toate cartierele.
* Utilizare: simulări numerice, probleme de rețele.
* necesită împărțirea inteligentă a datelor pentru a evita dezechilibrele de lucru.
* poate avea overhead semnificativ pentru sincronizare.
* scalabilitate excelentă pentru probleme mari.
* reduce comunicațiile excesive între procesoare.

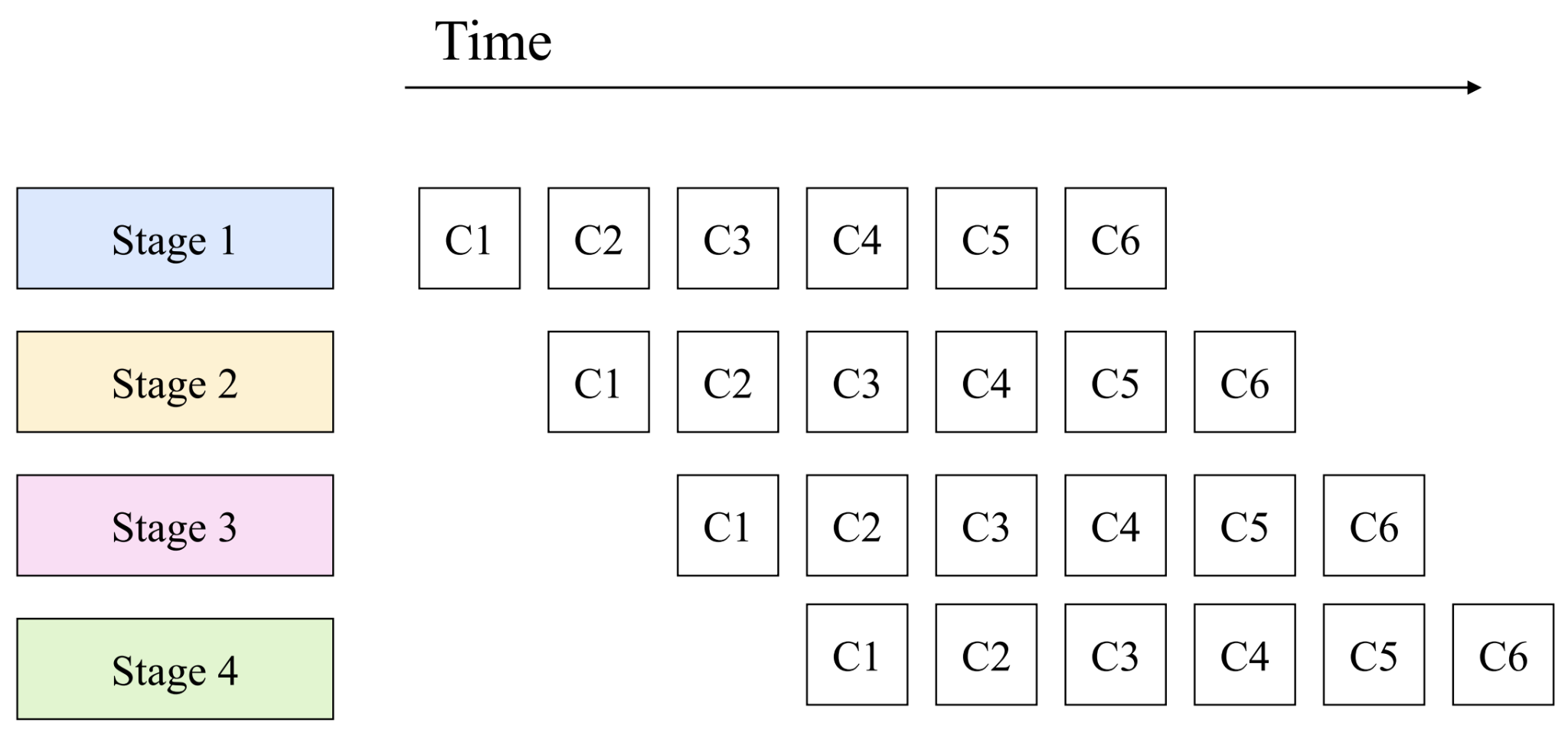


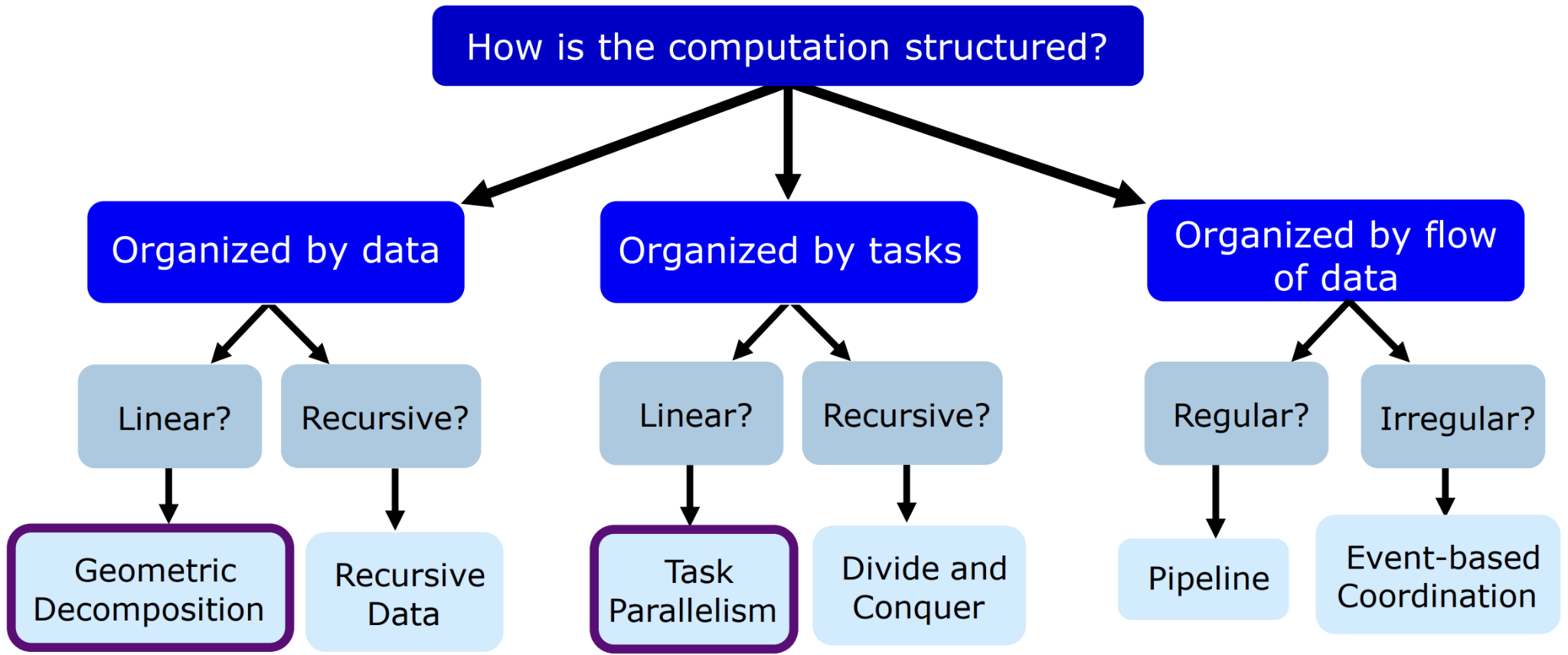
## Pipeline

* problema este împărțită în **etape consecutive** (task-uri) care sunt procesate una după alta, dar în paralel. Fiecare thread prelucrează o etapă și trimite rezultatul mai departe, ca pe o bandă rulantă.
* ex.: Într-o fabrică de automobile:

1. O persoană pune roțile.
2. Alta montează motorul.
3. Alta pictează mașina. Fiecare etapă funcționează în paralel pe mașini diferite.

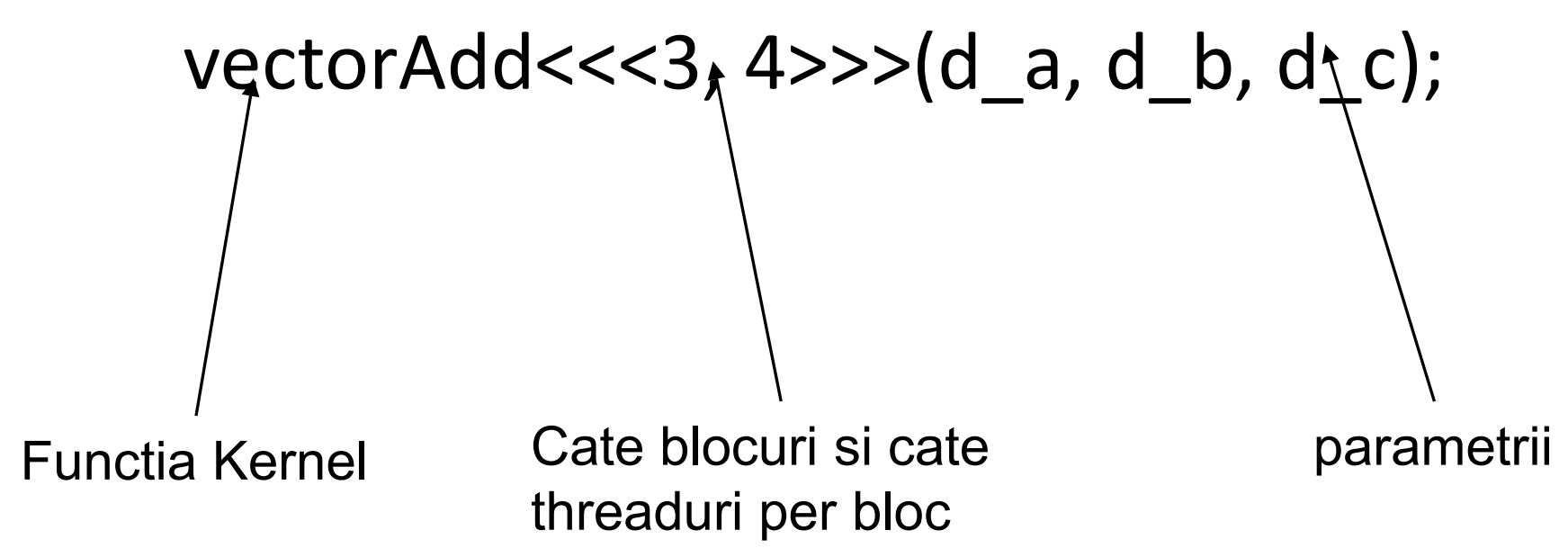
* Utilizare: procesare video, sisteme de procesare audio sau fluxuri de date.
* întârzierile la o etapă pot încetini întregul pipeline.
* necesită sincronizare între etape.
* exploatează bine paralelismul etapizat.
* funcționează eficient pentru fluxuri de date continue.



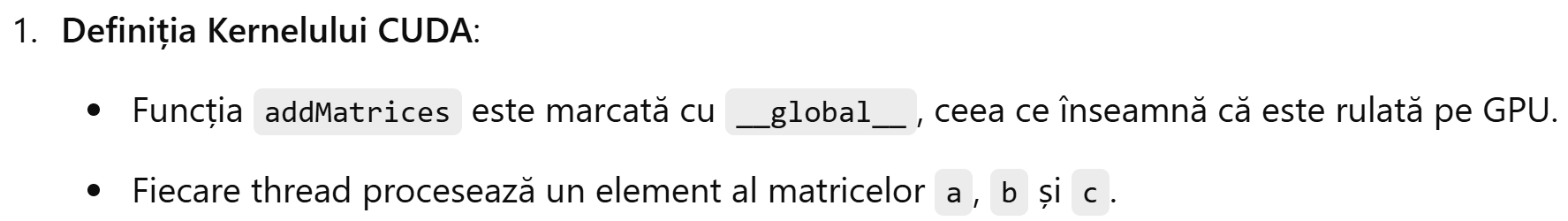


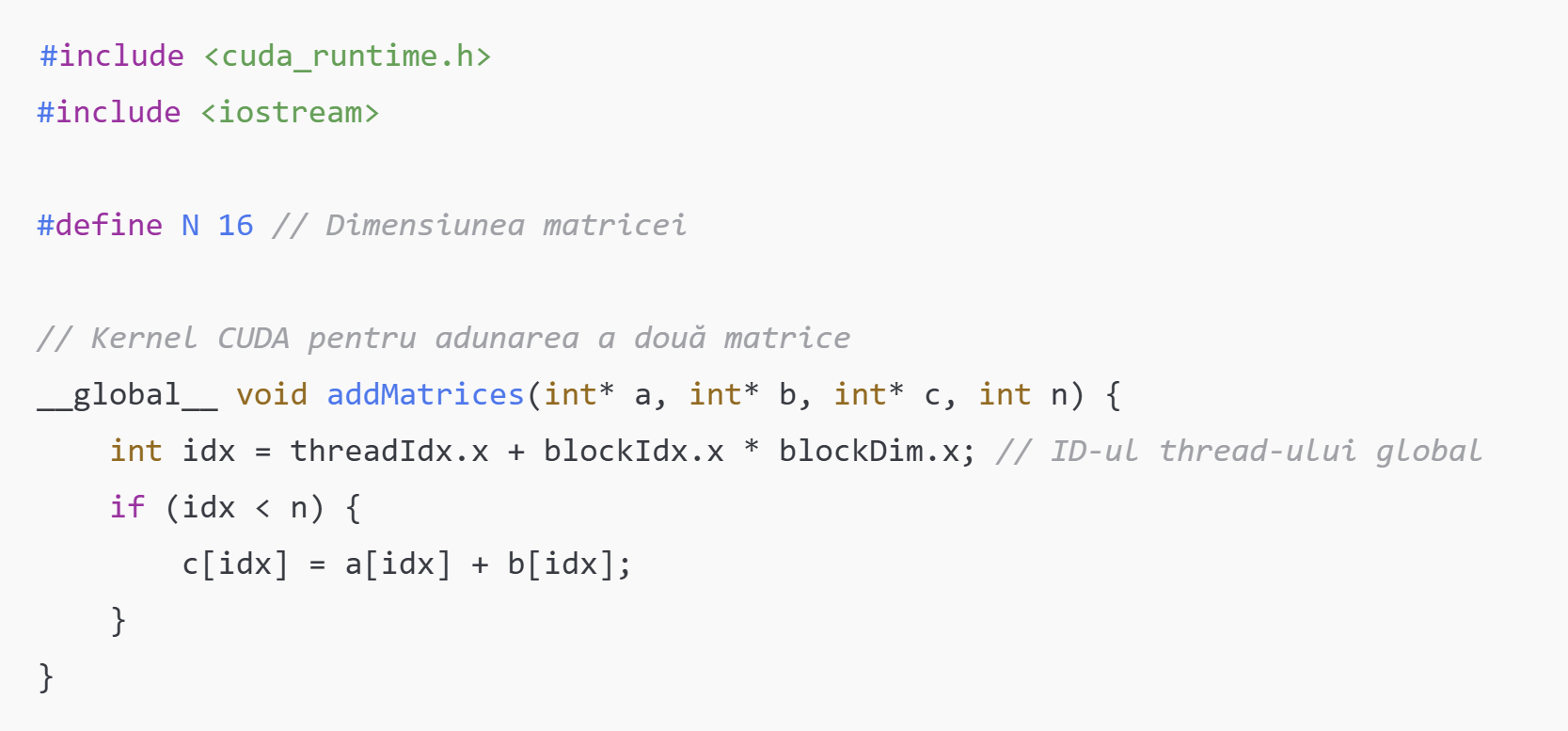
# CUDA

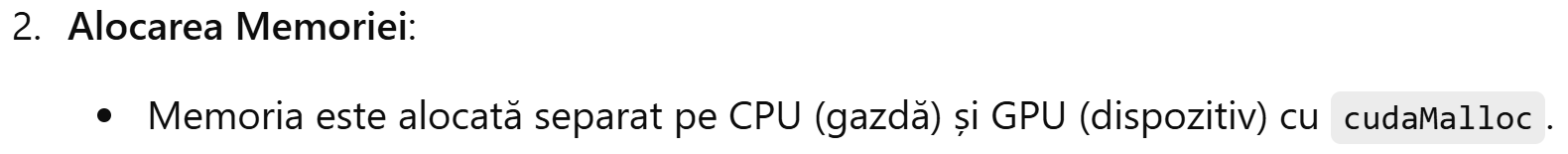
* CUDA este o platformă dezvoltată de NVIDIA pentru programarea GPU-urilor (unități de procesare grafică). Aceasta permite rularea codului pe GPU, ceea ce accelerează execuția pentru probleme care necesită calcule paralele.
* codul CUDA constă din **cod gazdă (rulat pe CPU)** și **cod dispozitiv (device) (rulat pe GPU)**.
* codul pentru GPU este scris sub formă de *kernel*, o funcție specială marcată prin specificatorul \_\_global\_\_.
* kernel-ul este lansat de CPU pe GPU, specificând **numărul de thread-uri** și **blocuri**.
* fiecare thread are un **ID unic**, care determină ce parte din date procesează.

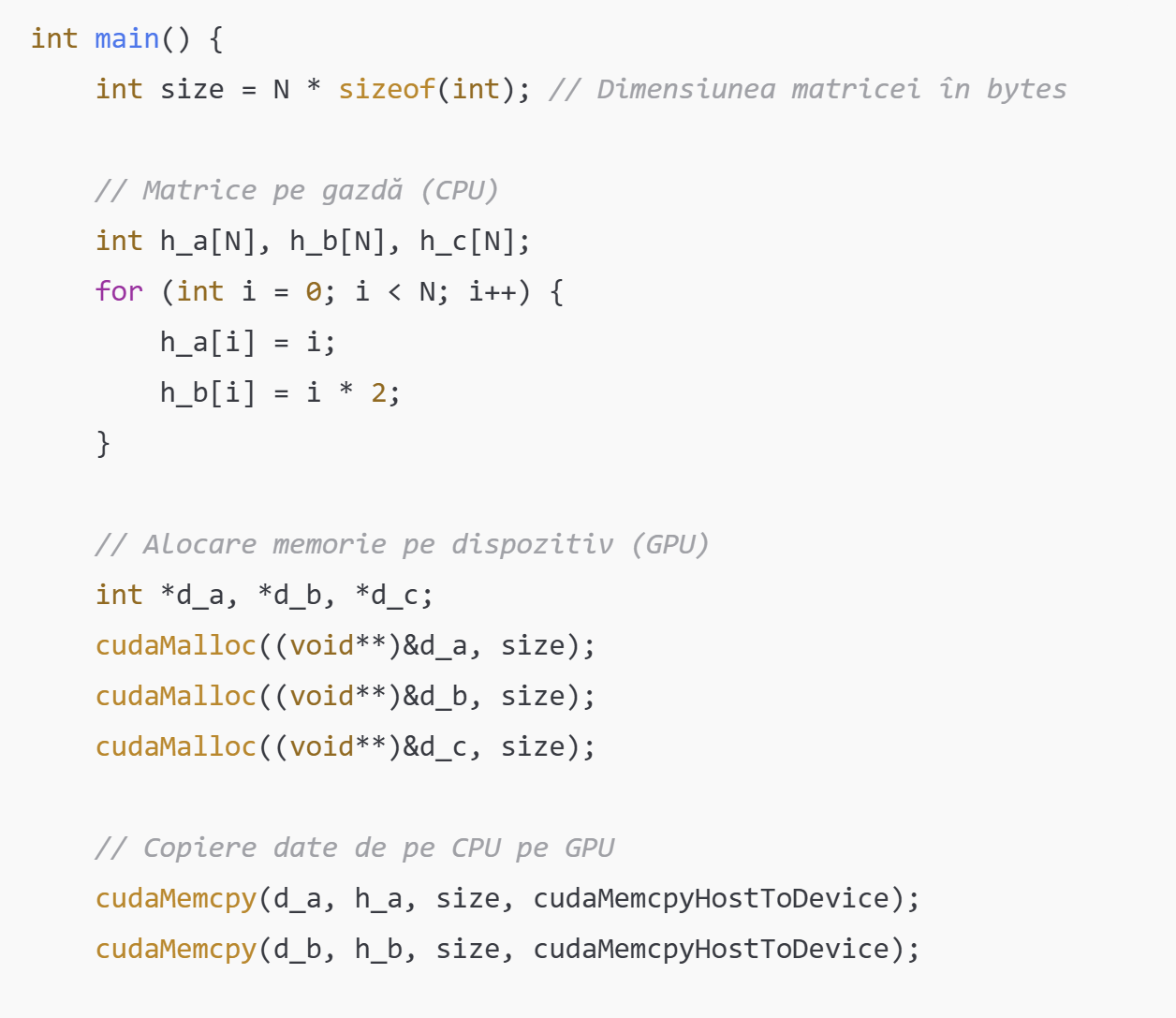


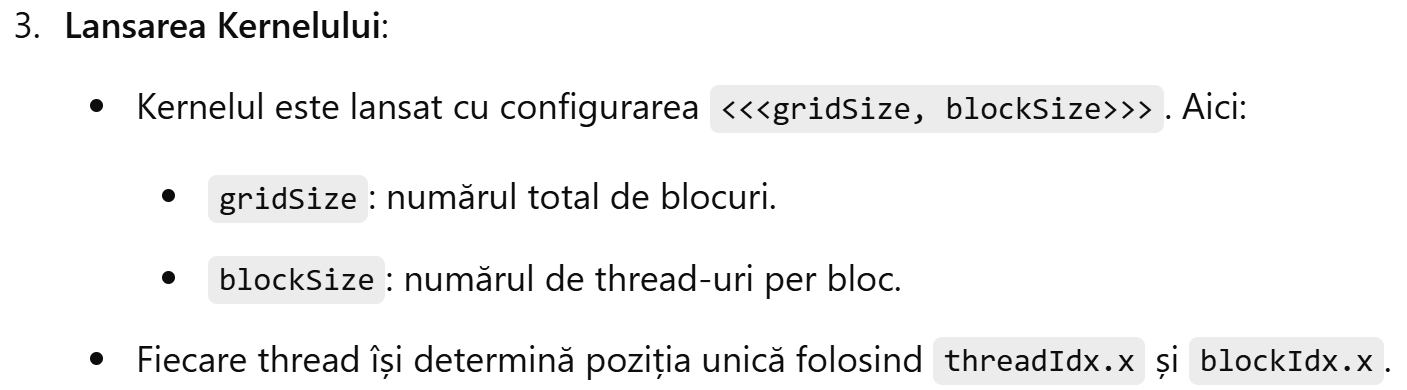
Exemplu

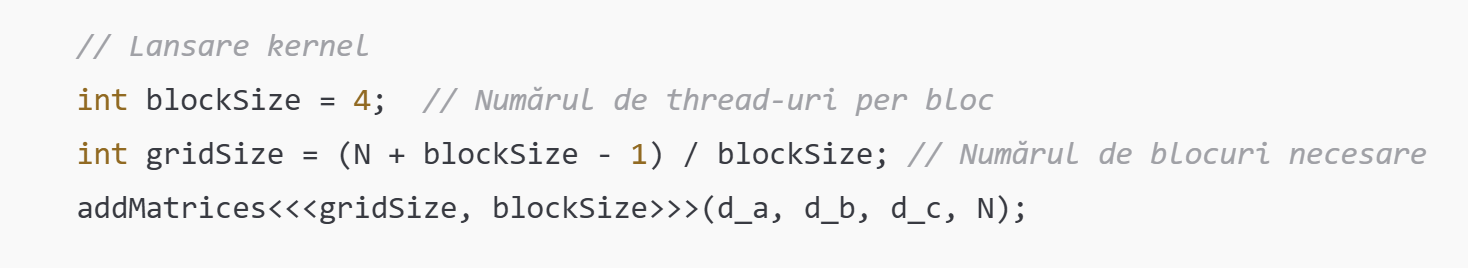




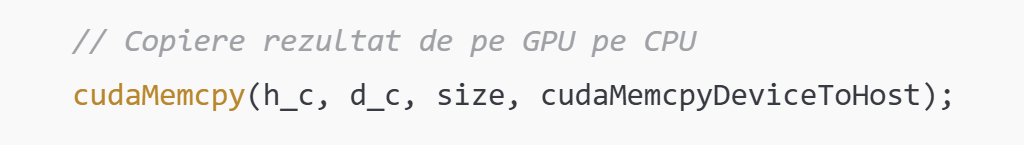


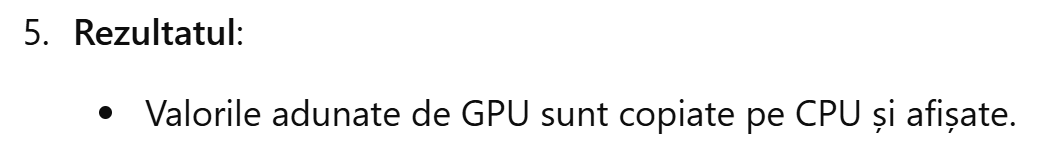


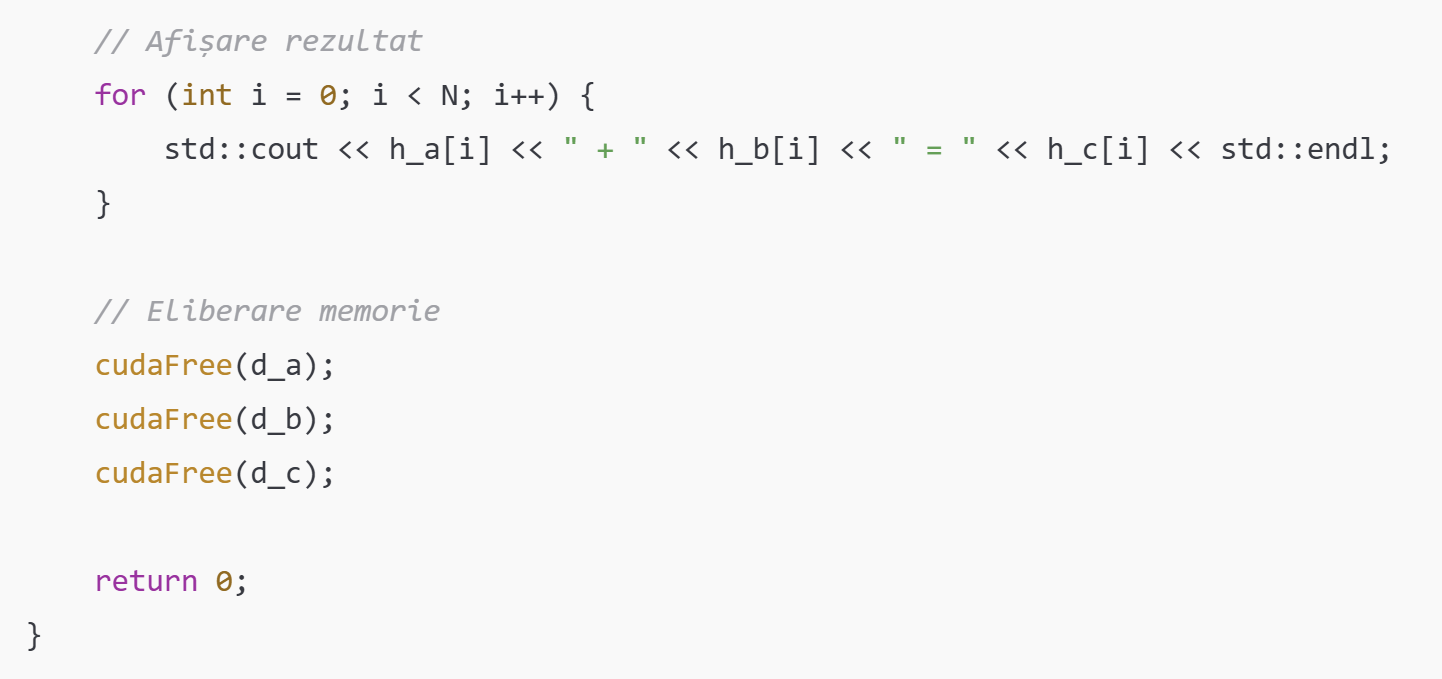












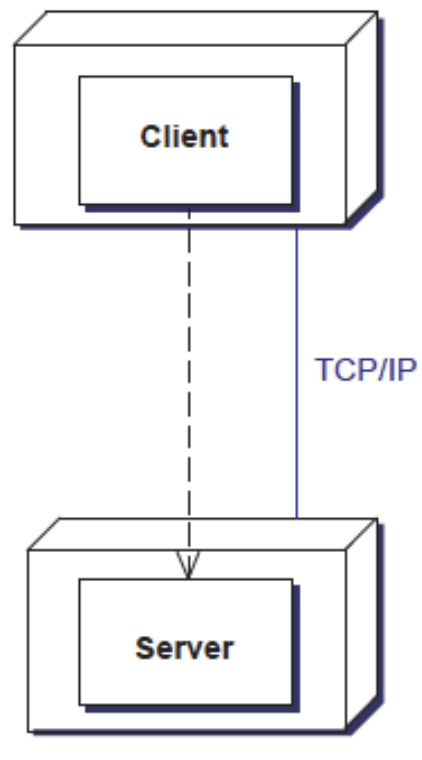
Utilizarea optimă a thread-urilor fără a irosi resurse pe GPU:

* GPU-ul rulează **thread-uri** organizare în **blocuri**
* Fiecare thread dintr-un bloc procesează o porțiune din date
* calculul thread-urilor redundante:
  + **N** elemente de procesat
  + **blockSize** - dimensiunea unui bloc (nr. de thread-uri într-un bloc)
  + **gridSize** - numărul total de blocuri
  + numărul de thread-uri totale lansate este determinat de:
* Dacă N nu este multiplu al , unele thread-uri vor rămâne nefolosite.

# Șabloane de proiectare distribuită

Un **sistem distribuit** poate fi definit ca fiind format din **componente hardware și software** localizate **într-o rețea de calculatoare** care *comunică* și *își coordonează acțiunile* doar bazat pe transmitere de mesaje.

## Client-Server Pattern

* cel mai bun model în sistemele distribuite
  + clientul - solicită servicii sau resurse
  + serverul - procesează cererile și răspunde cu rezultatele
* utilizare: aplicații web, baze de date
* clienții și serverele lucrează în sesiuni
  + stateless server: starea unei sesiuni este gestionată de către client. Această stare este trimisă împreună cu fiecare cerere.
  + stateful server: starea unei sesiuni este menținută la nivel de server și este asociată cu ID-ul clientului
* fault-handling: starea menținută la nivelul clientului implică faptul că toată informația se va pierde în cazul în care clientul eșuează.
* scalabilitate redusă: starea se menține la nivelul serverului ‘in-memory’ => necesar memorie crescut.

## Peer-to-Peer Pattern

* toate nodurile sunt egale => nu există un server central
* fiecare nod poate fi atât client, cât și server
* nodurile comunică direct între ele pentru a partaja resurse sau informații.
* utilizare: torrenting, aplicații descentralizate (blockchain)
* performanța crește odată cu numărul de noduri, dar scade dacă sunt prea puține.
* cost individual mic - nodurile pot folosi capacitatea întregului sistem
* overhead-ul de administrare este scăzut
* scalabilitate foarte bună
* nu există garanția calității serviciilor
* nu există garanția securității

## Communication Pattern: Forwarder-Receiver

* model folosit pentru transferul de mesaje între componente distribuite
* mesajele sunt trimise de un **Forwarder** către un **Receiver**.
* forwarder-ul primește o cerere sau un mesaj și îl transmite mai departe către Receiver (destinatarul final)

## Pipe-Filter Pattern

* model compus dintr-o serie de pași (filtre) care prelucrează datele, conectate prin canale (pipe-uri).
  + datele sunt trimise printr-un pipe de la un filtru la altul
  + fiecare filtru procesează datele și le trece mai departe

## Blackboard Pattern

* model în care mai multe componente (agenți) colaborează printr-un spațiu de lucru comun (Blackboard).
  + fiecare agent își pune rezultatele parțiale pe tabla neagră
  + alt agent poate citi rezultatele și contribui cu informații noi
  + procesul continuă până când problema este rezolvată
* utilizare: Inteligență artificială, sisteme colaborative

Alte șabloane: Messaging Pattern, Publisher-Subscriber Pattern, Event-Bus Pattern, Broker Pattern, Client-Proxy Pattern, Reactor Pattern, Proactor Pattern.

# Noțiuni extra cu exemple Java / C++

## Deadlock

* apare când două sau mai multe procese sunt blocate, fiecare așteptând ca alt proces să elibereze o resursă, ceea ce nu se întâmplă niciodată.
* Două thread-uri:
* Thread 1 deține resursa A și așteaptă resursa B.
* Thread 2 deține resursa B și așteaptă resursa A. Ambele sunt blocate.
* cum se rezolvă:
  + folosește **ordinea resurselor**
  + evită **blocările circulare**

## Livelock

* similar cu deadlock-ul, dar în loc să rămână complet blocate, procesele continuă să lucreze fără progres, schimbându-și constant stările într-o încercare de a rezolva problema.
* Doi oameni care încearcă să treacă unul pe lângă altul pe un coridor îngust:
* Ambii se mișcă în același timp la stânga, apoi la dreapta, fără să treacă.
* cum se rezolvă:
  + fiecare proces **așteaptă un timp diferit** înainte de a încerca din nou
  + asigură **coordonare între procese**

## Starvation

* apare atunci când un proces/thread nu primește acces la resurse pentru o perioadă lungă de timp, deși resursele sunt disponibile, deoarece alte procese prioritizează în mod continuu utilizarea lor. (are întotdeauna prioritate mai mică la resurse)
* O coadă la magazin:
* O persoană timidă așteaptă la coadă, dar continuă să fie ignorată deoarece alții sar peste rând.
* cum se rezolvă:
  + folosirea **priorităților echitabile** (FIFO, Round-Robin)
  + **implementarea unui timer** pentru a oferi șansa unui proces ignorat.