

# Teorie PPD

## 1. Clasificarea Flynn

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

- fiecare parte a programului este conținută instrucțiuni sau bucati 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.

### a. SISD (Instrucțiune unică, Date unice)

- i. ex.: o singură persoană care lucrează la un proiect de la început până la sfârșit, sarcină cu sarcină, fără ajutor - majoritatea calculatoarelor obișnuite.
- ii. funcționează pe rând și este mai lentă pentru sarcini complexe.

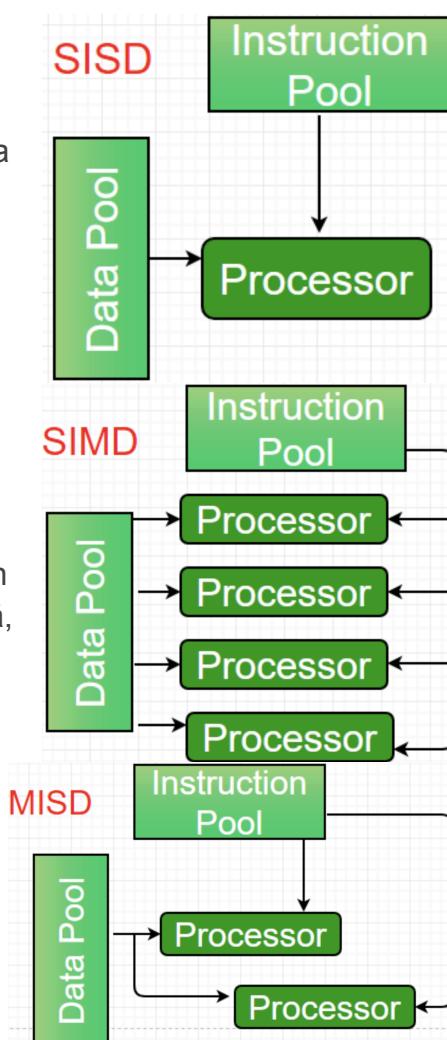
### b. SIMD (Instrucțiune unică, Date multiple)

- i. 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.
- ii. ok pentru operațiuni care repetă aceleași instrucțiuni pe date diferite.

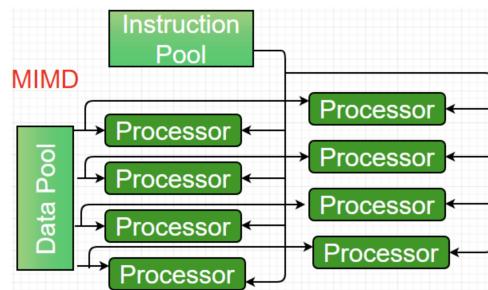
### c. MISD (Instrucțiuni multiple, Date unice)

- i. ex.: mai multe persoane fac lucruri diferite pe același material (fiecare analizează un buștean din perspective diferite: unul măsoară, altul desenează, altul îl taie).

		Instruction Streams
Data Streams	one	many
	one	SISD traditional von Neumann single CPU computer
many	many	MISD May be pipelined Computers
many	one	SIMD Vector processors fine grained data Parallel computers
many	many	MIMD Multi computers Multiprocessors



- ii. nu este comun și nici eficient în cele mai multe cazuri.
- d. MIMD (Instrucțiuni multiple, Date multiple)
  - i. ex.: fiecare persoană lucrează la sarcini diferite pe materiale diferite (unul construiește un perete, altul montează ușă) - calculatoarele moderne și supercalculatoare, ca sistemele de la IBM.
  - ii. cel mai flexibil model și poate face orice tip de sarcină.
  - iii. 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)
  - iv. 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)
  - v. MIMD hibrid (aparent există și asta)



## 2. UMA vs NUMA - shared memory

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

- a. UMA (Uniform Memory Access)
  - i. toate procesoarele au acces egal la aceeași memorie. Fiecare procesor poate ajunge la orice parte a memoriei cu aceeași viteză.
  - ii. simplu de gestionat, ușor de programat, performanță previzibilă, deoarece toate procesoarele au acces egal la memorie.
  - iii. nu este scalabil pentru un număr mare de procesoare.
  - iv. exemple de sisteme: calculatoare cu memorie partajată (SMP - Symmetric Multi-Processing).
- b. NUMA (Non-Uniform Memory Access)
  - i. 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.
  - ii. 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.
  - iii. este scalabil, performanță mai bună pentru aplicații care folosesc doar memoria locală.
  - iv. mai dificil de programat (trebuie gestionat ce memorie accesează fiecare procesor), accesul la memoria altor procesoare este mai lent.

- v. exemple de sisteme: calculatoare cu memorie distribuită (supercomputere, servere mari).

**Cache Coherent (Coerență 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 (Latență)** - timpul în care o dată ajunge la procesor după ce s-a inițiat cererea.

Latență mică înseamnă un sistem mai rapid. (ca timpul de așteptare după ce ai dat o comandă la restaurant. Cu cât așteptă 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.)

### 3. SMP (Symmetric Mult-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.

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

## 4. 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.

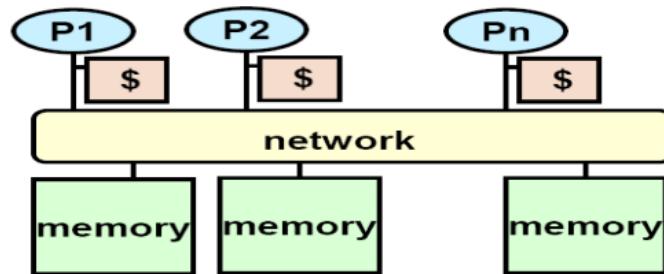
- a. Fiecare procesor are **memoria sa locală**
- b. Procesoarele comunică între ele prin intermediul unei **rețele de interconectare** foarte rapide
- c. Problema este împărțită în multe **sub-probleme mai mici**, fiecare procesor lucrând pe o parte specifică a datelor.
- d. Sarcinile sunt **independente**,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

Caracteristică	SMP	MPP	UMA	NUMA	Cluster de SMP	Distributed Memory
<b>Număr de procesoare</b>	Limitat (de obicei <64)	Foarte mare (sute/mii)	Limitat (să zicem 1-16)	Mai mare (sute de procesoare)	Poate fi foarte mare	Foarte mare (poate ajunge la mii)
<b>Memorie</b>	Partajată	Distribuită	Partajată	Distribuită	Distribuită și partajată	Distribuită
<b>Tipul de acces la memorie</b>	Acces simultan la memorie partajată	Memorie locală pentru fiecare procesor	Memorie partajată	Fiecare procesor are propria memorie locală	Memorie partajată și locală pentru fiecare nod	Memorie locală pentru fiecare procesor
<b>Scalabilitate</b>	Limitată	Foarte bună	Limitată	Foarte bună	Foarte bună	Foarte bună
<b>Comunicare între procesoare</b>	Prin memorie partajată	Prin rețea de interconectare	Prin memorie partajată	Prin rețea de interconectare	Prin rețea (de obicei Ethernet sau Infiniband)	Prin rețea, folosind protocoale IPC
<b>Viteză de comunicare</b>	Foarte rapid (memorie locală)	Depinde de rețea (mai lentă)	Rapidă (memorie partajată)	Depinde de rețea (poate fi mai lentă decât SMP)	Depinde de rețea (poate fi foarte rapidă)	Depinde de rețea (mai lentă comparativ cu SMP)
<b>Coerența cache</b>	Necesită protocol de coerență cache	Nu este necesar (memorie distribuită)	Necesită protocol de coerență cache	Necesită protocol de coerență cache	Necesită protocol de coerență cache	Nu este necesar (memorie distribuită)
<b>Topologie rețea</b>	Simplă	Complexă	Nu este necesar	Complexă (depinde de arhitectură)	Complexă (tip torus, mesh, etc.)	Depinde de sistem (torus, mesh, etc.)
<b>Software necesar</b>	Simplu (un singur sistem de operare)	Complex (sisteme distribuite)	Simplu (un singur sistem de operare)	Complex (sisteme distribuite)	Complex (sisteme distribuite)	Complex (sisteme distribuite)
<b>Cost</b>	Relativ scăzut	Foarte ridicat	Scăzut	Ridicat (datorită rețelelor și memoriei distribuite)	Ridicat (pentru rețele rapide și gestionarea clusterului)	Scăzut până la mediu (în funcție de implementare)

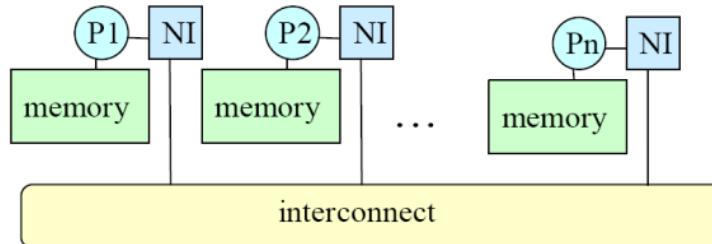
Dificultate în programare	Relativ simplu	Difícil (programare paralelă masivă)	Simplu	Difícil (trebuie gestionată memoria distribuită)	Difícil (în gestionarea mai multor noduri)	Difícil (trebuie gestionată memoria distribuită)
Rezistență la defecte	Vulnerabilă (defecțiune în sistemul partajat afectează totul)	Poate continua să funcționeze în cazul defectării unui procesor	Vulnerabilă (defecțiune în sistemul partajat afectează totul)	Mai tolerantă (fiecare procesor are memorie proprie)	Tolerantă (defecțiune într-un nod nu afectează întregul sistem)	Tolerantă (fiecare procesor are propria memorie)
Exemple	Servere mici și medii, stații de lucru	Supercomputere (ex. Cray, Blue Gene)	Arhitecturi simple de calcul	Sisteme mari de servere (ex. NUMA servere)	Cluster de servere pentru aplicații mari	Supercomputere și sisteme de calcul distribuit

## 5. Modele de programare

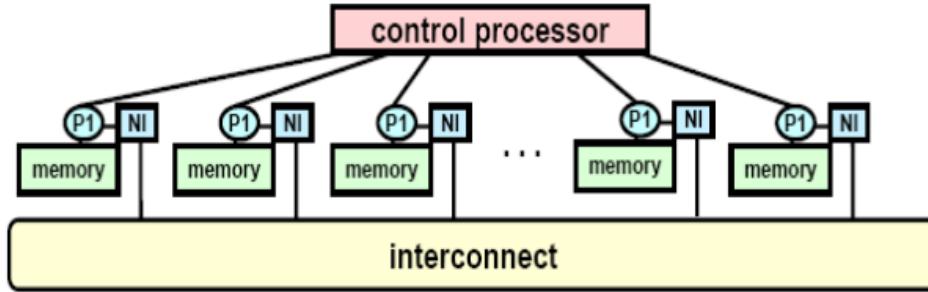
- a. **Spațiu Partajat de Adrese** - mai multe procesoare sau noduri dintr-un sistem au acces la aceeași zonă de memorie. Folosit în SMP și UMA. **Pot apărea probleme de coerentă a datelor (cache), deoarece fiecare procesor ar putea avea o copie locală a acelorași date.**



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



- c. **Paralelism al Datelor** - aceleasi 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.**



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

Model	Tip de arhitectură	Comunicare	Scalabilitate	Complexitate
Spațiu partajat de adrese	SMP, UMA, NUMA	Acces direct la memorie partajată	Limitată	Relativ simplu
Transfer de mesaje	MPP, Cluster de SMP, Distribuite	Trimiterea de mesaje între procese	Foarte bună	Complex
Paralelism al datelor	SIMD, MIMD	Operații simultane pe date	Foarte bună	Moderat
Modelul Hibrid	Combină diverse modele	Diverse metode de comunicare	Foarte bună	Complex

### 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

## 6. 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)

```
#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);  
    }  
    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();  
}
```

## 7. 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 concurrentă 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 \(variabile conditionale\)](#) - 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**.

```
class CounterMonitor {
    private int counter = 0;

    public synchronized void increment() {
        counter++;
        notifyAll(); // Notifică alte fire că s-a schimbat starea
    }

    public synchronized void waitForValue(int value) throws InterruptedException {
        while (counter < value) {
            wait(); // Așteaptă până când counter >= value
        }
    }
}
```

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

```

#include <iostream>
#include <mutex>
#include <condition_variable>

class BasketMonitor {
private:
    int apples = 0;
    const int MAX_APPLES = 10;
    std::mutex mtx;
    std::condition_variable cv;

public:
    void addApple() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this]() { return apples < MAX_APPLES; });
        apples++;
        std::cout << "Added an apple. Total: " << apples << std::endl;
        cv.notify_all();
    }

    void removeApple() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this]() { return apples > 0; });
        apples--;
        std::cout << "Removed an apple. Total: " << apples << std::endl;
        cv.notify_all();
    }
};

```

## 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 îți s-ar da un bon (Future) cu care poți verifica dacă e gata comanda sau să aștepți până e livrată.

```

import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // O sarcină care calculează o valoare
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(2000); // Simulează o operație costisitoare
            return 42; // Rezultatul calculului
        });

        System.out.println("Sarcina este în desfășurare...");
        System.out.println("Rezultatul este: " + future.get()); // Așteaptă rezultatul
        executor.shutdown();
    }
}

```

Java

```

// Funcție care calculează pătratul unui număr
int calculateSquare(int x) {
    return x * x;
}

int main() {
    // Lansăm sarcina în mod asincron folosind std::async
    std::future<int> future = std::async(std::launch::async, calculateSquare, 5);

    // Așteptăm rezultatul
    int result = future.get(); // Blocare până când calculul este finalizat

    std::cout << "Pătratul lui 5 este: " << result << std::endl;

    return 0;
}

```

C++

## 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.

```

ExecutorService executor = Executors.newFixedThreadPool(3);
for (int i = 1; i <= 5; i++) {
    final int taskNumber = i;
    executor.submit(() -> {
        System.out.println("Execut sarcina: " + taskNumber);
    });
}
executor.shutdown();

```

## 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.**

## 8. 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)` - initializează 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: +, \*, -, &, |, ^, &&, ||.

```

#include <iostream>
#include <omp.h>
#include <vector>

int main() {
    const int n = 20; // Dimensiunea array-ului
    const int num_threads = 4; // Numărul de thread-uri
    const int chunk = 5; // Dimensiunea unui "chunk" pentru distribuirea dinamică
    std::vector<int> a(n, 1); // Vector de dimensiune n inițializat cu 1
    std::vector<int> b(n, 2); // Vector de dimensiune n inițializat cu 2
    int sum = 0; // Variabilă pentru suma
    int prod = 1; // Variabilă pentru produs

    // Setăm numărul de thread-uri
    omp_set_num_threads(num_threads);

    // Regiune paralelă
    #pragma omp parallel default(none) shared(a, b, n, sum, num_threads, chunk) private(pr
    {
        // Inițializarea variabilei firstprivate
        prod = 1;

        // Secțiuni independente
        #pragma omp sections
        {
            // Secțiunea 1: calculează produsul elementelor din 'a'
            #pragma omp section
            {
                for (int i = 0; i < n; ++i) {
                    prod *= a[i];
                }
                #pragma omp critical
                {
                    std::cout << "Thread-ul " << omp_get_thread_num() << " a calculat produsul ";
                }
            }

            // Secțiunea 2: calculează suma elementelor din 'b'
            #pragma omp section
            {
                for (int i = 0; i < n; ++i) {
                    #pragma omp atomic
                    sum += b[i];
                }
            }
        }
    }
}

```

```

        #pragma omp critical
        {
            std::cout << "Thread-ul " << omp_get_thread_num() << " a calculat suma
        }
    }

    // Buclă paralelă cu reducere pentru calculul unui total
    #pragma omp for schedule(dynamic, chunk) reduction(+:sum)
    for (int i = 0; i < n; ++i) {
        sum += a[i] * b[i];
    }

    // Barieră pentru sincronizare
    #pragma omp barrier

    // Afisare după sincronizare
    #pragma omp single
    {
        std::cout << "Suma totală după reducere: " << sum << "\n";
    }
}

return 0;
}

```

## 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 threadurile văd aceleași valori.
  - **private(prod):** Fiecare thread are propria copie a lui **prod**.
  - **firstprivate(prod):** Fiecare thread își initializează **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.

## 9. 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.
- $T_{total} = T_{util} + T_{overhead}$

### Accelerarea ("speed-up")

- speed-up arată cât de mult se îmbunătățește timpul de execuție când folosim mai multe resurse (ex. mai multe procesoare)

- $S(p) = T_{\text{serial}} / T_{\text{parallel}}$
- $T_{\text{serial}}$  - timpul de execuție cu un singur procesor
- $T_{\text{parallel}}$  - timpul de execuție cu  $p$  procesoare
- ex.: Dacă ai de spălat vase și singur îți ia o oră ( $T_{\text{serial}} = 60$  min), iar, cu ajutorul unui prieten, îți ia 30 de minute ( $T_{\text{parallel}} = 30$  min), speed-up-ul este  $S(2) = 60 / 30 = 2$ .

### Eficiență

- 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), eficiență este  $E(2) = 2 / 2 = 1$  (*Resurse utilizate perfect!*).
- ex.: Dacă 3 persoane lucrează și durează tot 30 de minute (speed-up = 2), eficiență 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.
- $$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$
- $f$ : 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.
- $$S(p) = p - (1 - f) \cdot p$$
- $f$ : 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.
- $$\text{Cost} = T(p) \cdot p$$
- $T(p)$ : 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  $2 \cdot 4 = 8$  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 - eficiență constantă: Scalabilitatea este bună dacă eficiența rămâne aproape constantă pe măsură ce adaugi procesoare.
- $Scalabilitate = \frac{E(p_2)}{E(p_1)}$ ,  $E(p)$  - eficiență 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. Necesară 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 corridorul este îngust), DOP este 2.
  - DOP mai mare indică un sistem mai capabil să proceseze mai multe sarcini în paralel.

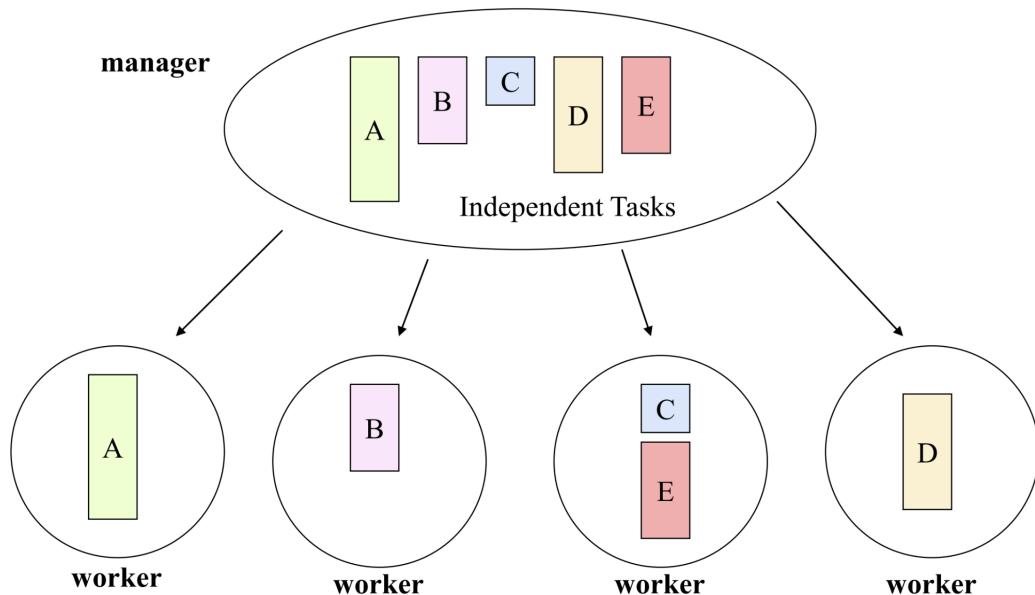
## 10. Şabloane de proiectare paralelă

### 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.
- inefficient pentru probleme foarte mari sau complexe.

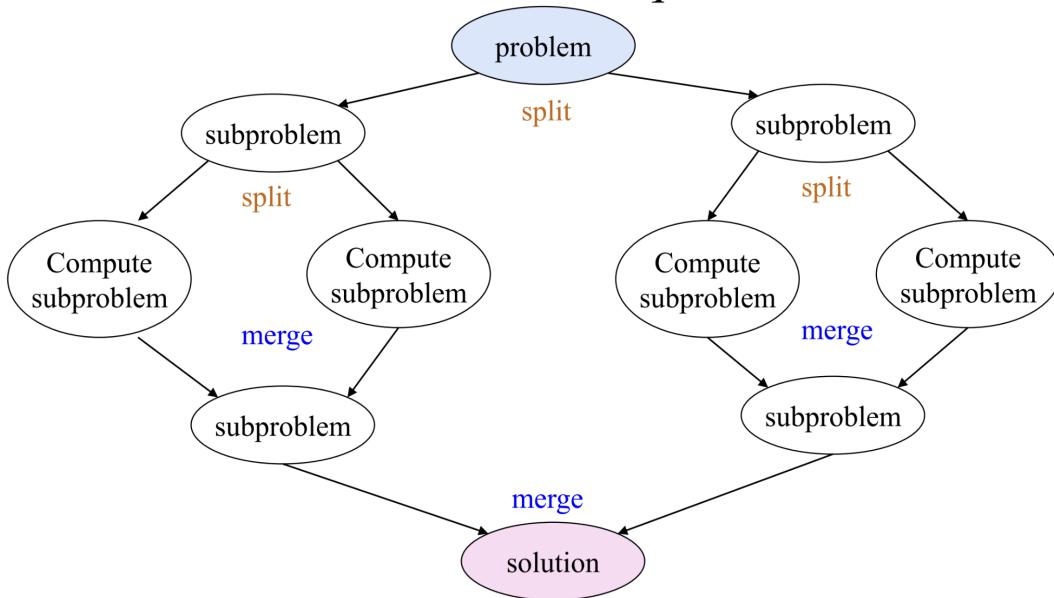
### *Master-slave (master-worker)*



### Divide & Conquer

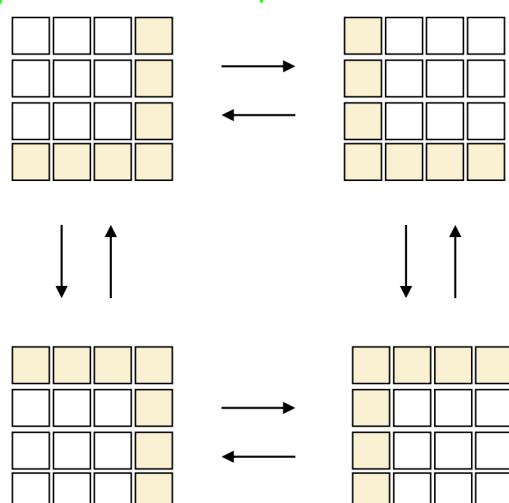
- problema mare este împărțită în probleme mai mici, fiecare dintre ele fiind rezolvată independent. Rezultatele subtask-urilor sunt apoi combinate pentru a forma soluția completă.
- ex.: Dacă ai o echipă de curățenie pentru o casă mare:
  1. Împărți casa în camere.
  2. Fiecare persoană curăță o cameră.
  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

## Divide & Conquer



## Geometric Decomposition

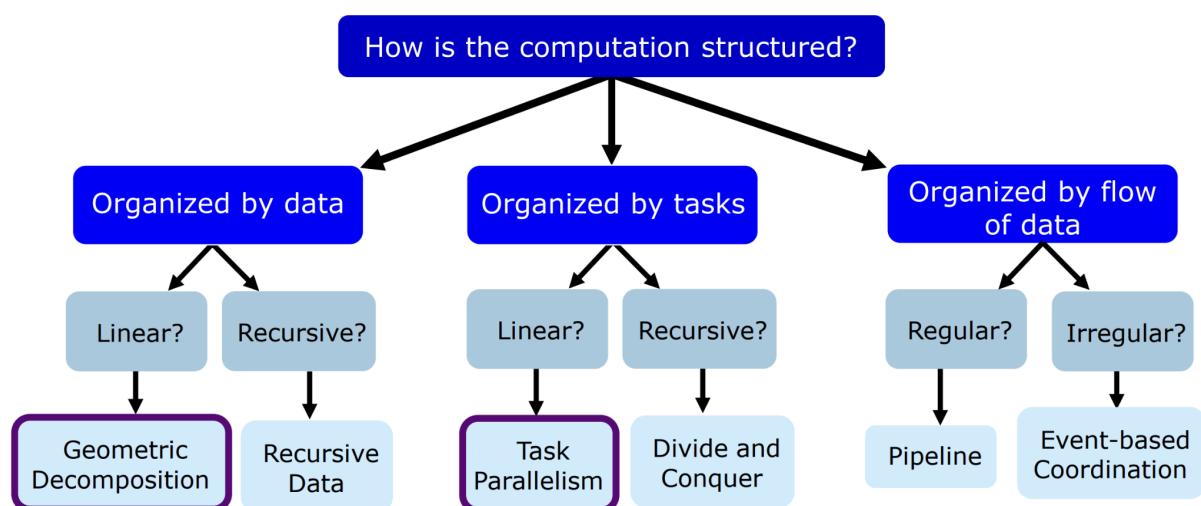
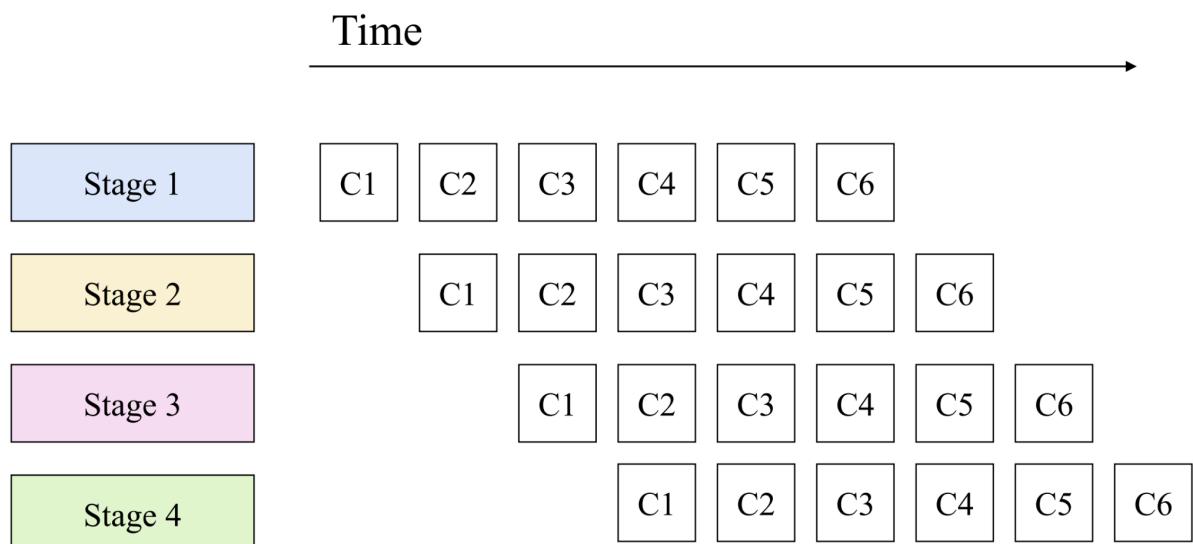
- problema este împărțită în funcție de **datele geometrice** sau **spațiale**. Fiecare thread primește o porțiune de date și lucrează doar pe acestea, comunicând periodic cu celelalte pentru a sincroniza rezultatele.
- ex.: o hartă a unui oraș împărțită în cartiere. Fiecare echipă repară drumurile dintr-un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



Neighbor-To-Neighbor communication

## Pipeline

- problema este împărtită 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. Alte 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.



## 11. 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.

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

Functia Kernel      Cate blocuri si cate threaduri per bloc      parametrii

### Exemplu

#### 1. Definiția Kernelului CUDA:

- Funcția `addMatrices` este marcată cu `_global_`, ceea ce înseamnă că este rulată pe GPU.
- Fiecare thread procesează un element al matricelor `a`, `b` și `c`.

```
#include <cuda_runtime.h>
#include <iostream>

#define N 16 // Dimensiunea matricei

// Kernel CUDA pentru adunarea a două matrice
__global__ void addMatrices(int* a, int* b, int* c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x; // ID-ul thread-ului global
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```

## 2. Alocarea Memoriei:

- Memoria este alocată separat pe CPU (gazdă) și GPU (dispozitiv) cu `cudaMalloc`.

```
int main() {
    int size = N * sizeof(int); // Dimensiunea matricei în bytes

    // Matrice pe gazdă (CPU)
    int h_a[N], h_b[N], h_c[N];
    for (int i = 0; i < N; i++) {
        h_a[i] = i;
        h_b[i] = i * 2;
    }

    // Alocare memorie pe dispozitiv (GPU)
    int *d_a, *d_b, *d_c;
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Copiere date de pe CPU pe GPU
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
```

## 3. Lansarea Kernelului:

- Kernelul este lansat cu configurarea `<<<gridSize, blockSize>>>`. Aici:
  - `gridSize`: numărul total de blocuri.
  - `blockSize`: numărul de thread-uri per bloc.
- Fiecare thread își determină poziția unică folosind `threadIdx.x` și `blockIdx.x`.

```
// Lansare kernel
int blockSize = 4; // Numărul de thread-uri per bloc
int gridSize = (N + blockSize - 1) / blockSize; // Numărul de blocuri necesare
addMatrices<<<gridSize, blockSize>>>(d_a, d_b, d_c, N);
```

## 4. Sincronizarea Datelor:

- Datele sunt transferate de la gazdă la dispozitiv și invers folosind `cudaMemcpy`.

```
// Copiere rezultat de pe GPU pe CPU
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
```

## 5. Rezultatul:

- Valorile adunate de GPU sunt copiate pe CPU și afișate.

```
// Afisare rezultat
for (int i = 0; i < N; i++) {
    std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << std::endl;
}

// Eliberare memorie
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);

return 0;
}
```

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

- GPU-ul rulează **thread-uri** organizate 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:  

$$\text{nr. total threaduri} = gridSize \times blockSize$$
  - Dacă N nu este multiplu al  $gridSize \times blockSize$ , unele thread-uri vor rămâne nefolosite.

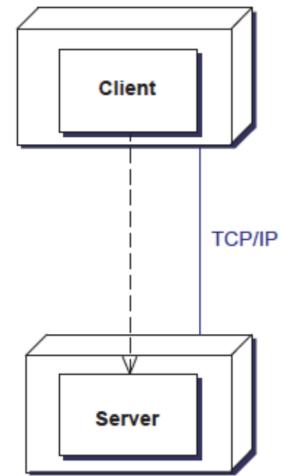
## 12. Şabloane de proiectare distribuită

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

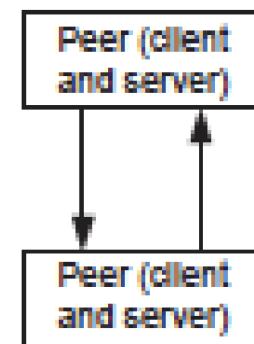
- cel mai bun model în sistemele distribuite

- **clientul** - solicită servicii sau resurse
- **serverul** - procesează cererile și răspunde cu rezultatele
- utilizare: aplicații web, baze de date
- clientii 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 (agente) 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 şablonane: 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 corridor 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 priorizează 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.