

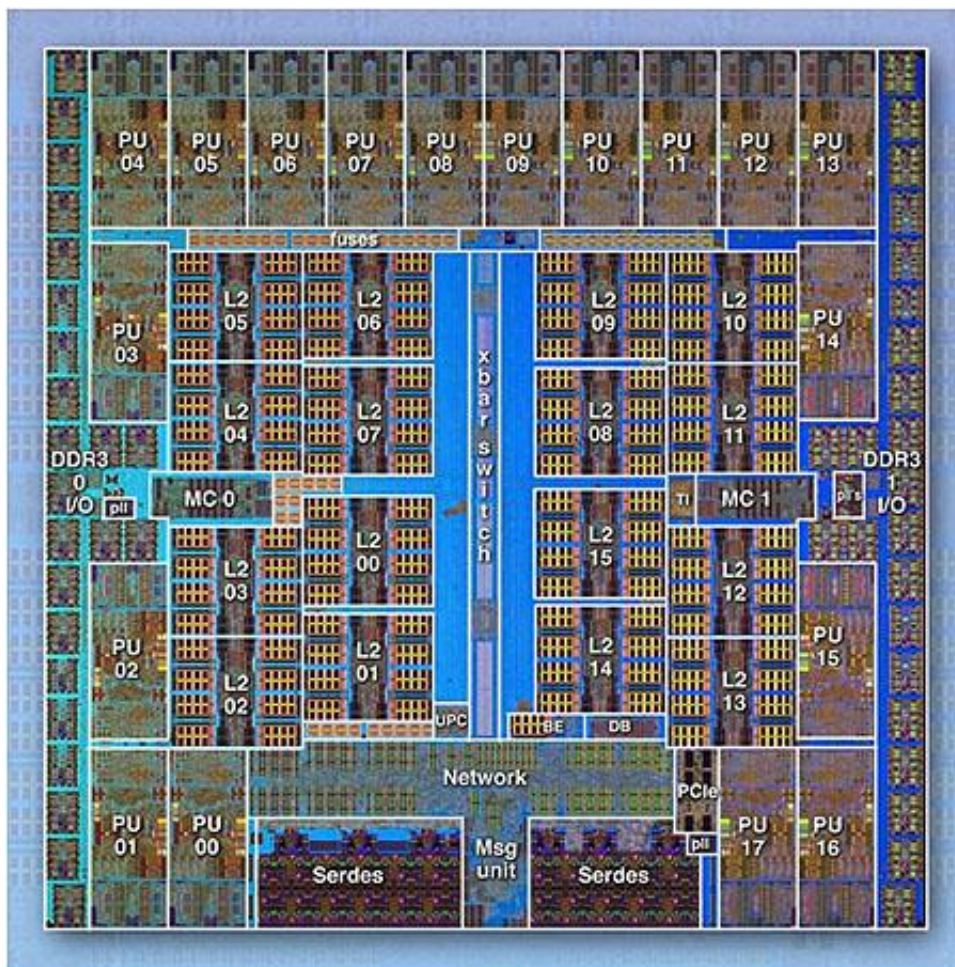
# Programarea multi-core și multi-threading -Exemplificare în Java-



# De ce multi-core?



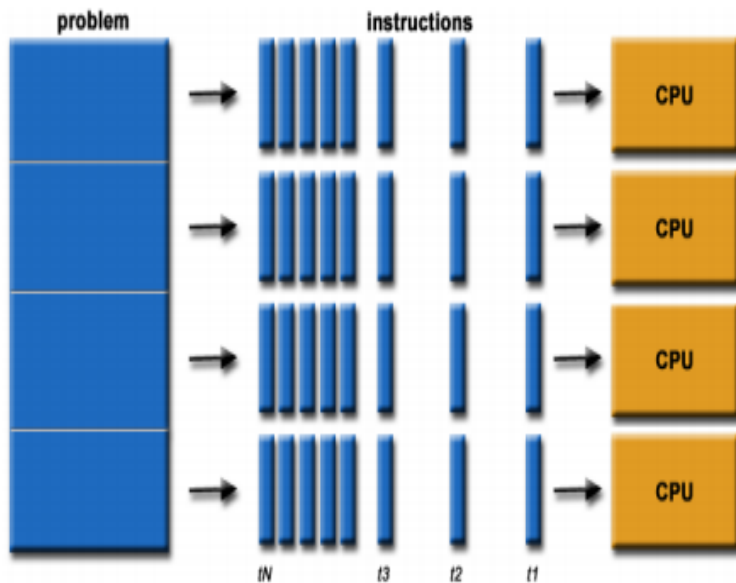
- ⇒ aplicații cât mai rapide și eficiente;
- ⇒ viteze mari de calcul și scalabilitate (trebuie să funcționeze pe un număr variabil de procesoare);
- ⇒ Îmbunătățiri considerabile în ceea ce privește puterea de procesare și viteza de execuție a aplicațiilor software;
- ⇒ existența unor aplicații mult mai complexe decât cele existente;
- ⇒ noi moduri de cercetare și explorare științifice;
- ⇒ În 2001 IBM introduce primul procesor multi-core cu două nuclee de procesare;



➡ Arhitectură multi-core cu 18 core-uri



# 1. Procesarea paralelă



- ⇒ utilizarea simultană a mai multor resurse de calcul pentru a rezolva o problemă computațională;
- ⇒ o problemă este împărțită în mai multe părți;
- ⇒ fiecare parte este divizată într-o serie de instrucțiuni;
- ⇒ instrucțiunile se execută simultan pe diferite procesoare;
- ⇒ se utilizează un mecanism general de control;

- **Resursele de calcul** pot fi:
  - Un calculator cu mai multe procesoare/nuclee
  - Un număr de calculatoare conectate într-o rețea
- **Procesarea paralelă** include tehnici și tehnologii care fac posibil calculul în paralel: hardware, rețele, sisteme de operare, biblioteci, limbaje, compilatoare, algoritmi, etc.
- **Avantajele procesării paralele:**
  - Timp de calcul mai rapid
  - Rezolvarea unor probleme mai mari și mai complexe
  - Folosirea efectivă a resurselor de calcul
  - Costuri reduse
  - Reducerea constrângerilor asociate memoriei
  - Limitările mașinilor seriale
- **Limitele și costurile procesării paralele**

## 2. Limitari ale programarii paralele

**Legea lui Amdahl** definește limitările fundamentale ale programării paralele, oferind o evaluare a performanțelor mașinilor de calcul, în special în cazul utilizării de procesoare multiple – sau cu mai multe core-uri. Se poate defini un Factor de Accelerare, FA, prin care se poate stabili de câte ori mașina îmbunătățită va rula mai repede pornind de la timpii de executie vechi si nou.

$$FA = \frac{T_E V}{T_E N} = \frac{timpexecutievechi}{timpexecutienou}$$

# Programare paralelă

- Worker = Thread = Fir de execuție.
- O bucată de cod care se execută asincron față de restul programului.
- Multi threading = mai multe threaduri executate asincron în cadrul aceleiași aplicații.

# Parallelism in computer science

- Problemă: Doresc să adun toate numerele de la 0 la 100, cum pot face acest lucru cel mai eficient?
- A) Adun numerele unul câte unul până ajung la 100.
- B) Împart cele 100 de numere în 2 categorii 0 -> 49, 50 -> 100, le adun în paralel iar la final adun cele 2 sume.
- C) Împart numerele în 4 categorii și procedez ca la punctul B.



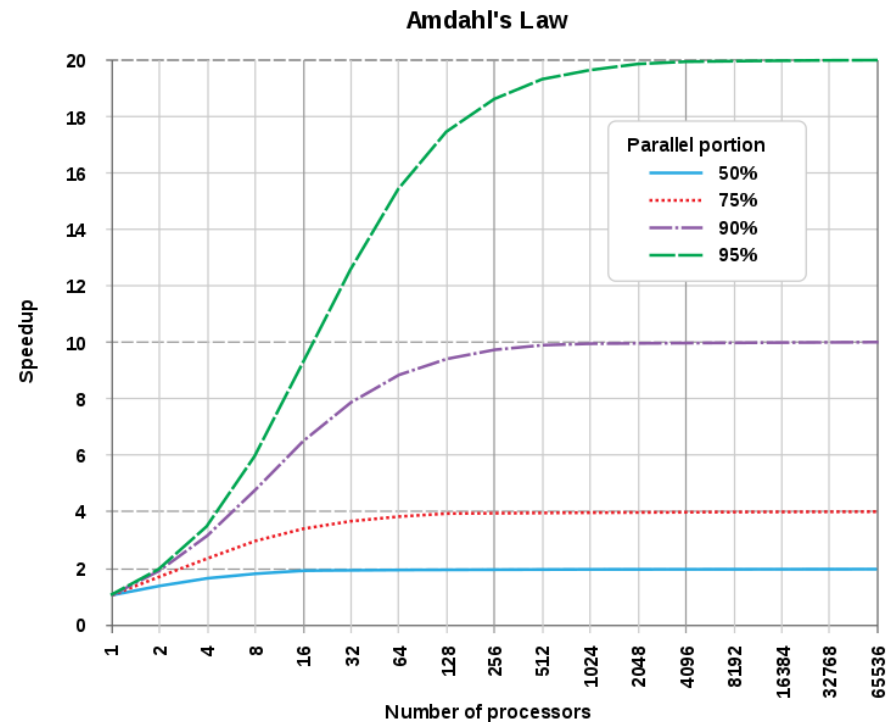
## Parallelism in computer science

- Timpul necesar adunării numerelor 0->100 cu 1 worker = 10s.
- Timpul necesar adunării numerelor cu 1 worker de la 0->50 este 5s, iar de la 0->100 cu 2 worker este ~5s.
- Timpul necesar adunării numerelor cu 1 worker de la 0->25 este de 2.5s iar de la 0->100 cu 4 worker este de ~ 2.5s.

# Parallelism in computer science

- Dacă numărul de workeri  $\Rightarrow \infty$  atunci timpul de execuție  $\Rightarrow 0$  ???
- Legea lui **Amdahl**:  $S = \frac{1}{(1-p) + \frac{p}{N}}$
- S – viteza teoretica
- p – portiunea paralelizabila
- N – numar de core-uri

# Parallelism in computer science



# 3. Parallelism vs. Concurență

## Paralelism:

- ➡ mai multe resurse de calcul pentru a rezolva o problemă mai rapid;
- ➡ mai multe niveluri de paralelism: *processe, thread-uri, rutine, instructiuni, etc;*
- ➡ resurse hardware: *procesoare, nuclee (core-uri), memorii, rețele, etc;*



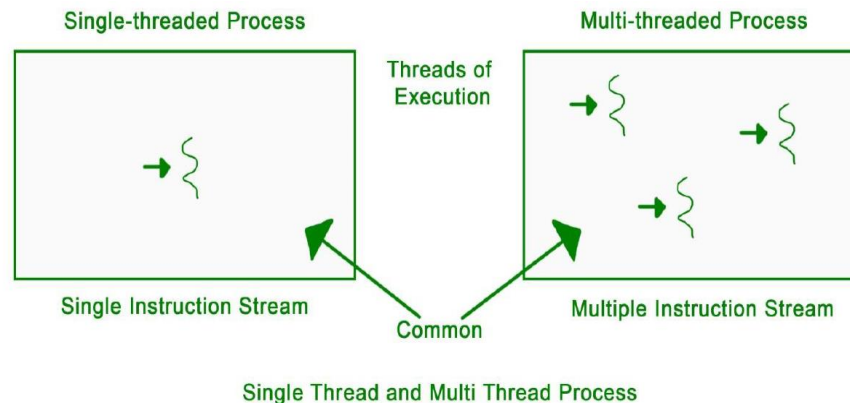
## Concurență:

- ➡ gestiunea corectă și eficientă a accesului la resurse comune;
- ➡ concurența este fundamentală în computer science: *sisteme de operare, baze de date, networking, etc;*

- **Multi-tasking** = capacitatea unui sistem de calcul de a executa mai multe programe în același timp ( $1, \dots, n$  procesoare);
- Execuție paralelă:
  - task-urile se execută efectiv în același timp;
  - este necesară existența de multiple resurse de calcul;
  - task-urile se consideră a fi ***pur paralele*** dacă se execută în același timp (execuție paralelă);
  - task-urile sunt dependente;

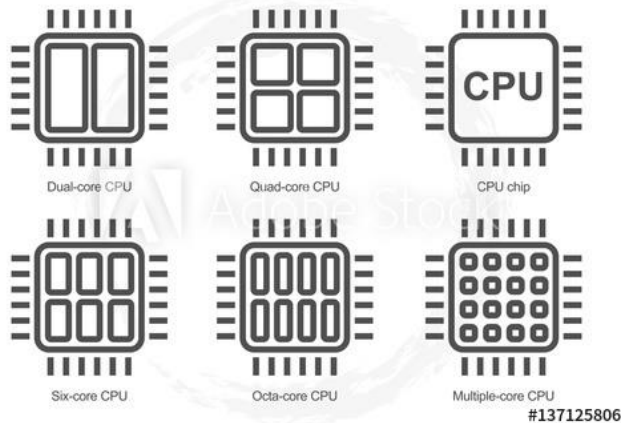
## 4. Thread-uri (Fire de execuție)

- **Thread** = secvență de cod dintr-o aplicație care se rulează separat față de firul de execuție principal al aplicației;
- Un proces este format de obicei din zeci sau sute de thread-uri;
- **Single-threading** = procesarea unei activități la un moment dat;
- **Multi-threading** (extensie a multi-tasking-ului) = capacitatea unui program de a executa mai multe secvențe de cod dintr-un program în același timp;



# 5. Arhitecturi multi-core

- ➡ puterea de procesare a calculatoarelor cunoscute a crescut exponențial;
- ➡ primele procesoare:
  - 4 biți lățimea magistralelor;
  - module de operare;
  - frecvența de tact era de ordinul kiloherților;
- ➡ procesoarele din zilele noastre:
  - mult mai puternice;
  - pot opera pe date de 128 de biți;
  - au frecvențe de tact de până la 5 GHz;
  - au mai multe core-uri;



# 6. Tehnici de paralelizare

- ➡ **Arhitecturi pipeline** – se folosesc de un singur core astfel sunt paralelizate instrucțiuni punându-se într-o coadă și executându-se fiecare într-o manieră secvențială;
- ➡ **Arhitecturi multi-core sau multi-threaded** – folosesc mai multe nuclee deținute de procesor, fiind executate instrucțiuni în paralel în același timp;
- ➡ **Arhitecturi de hyper-threading** – (firma Intel) prin care programatic este vizualizată o clonă a fiecărui nucleu din cadrul procesorului, fizic ele nu există dar din punct de vedere programatic și logic ele servesc drept nuclee adevărate pentru a fi paralelizate;
- ➡ **Arhitecturi distribuite** – mai multe procesoare, nu doar nuclee de procesoare, mai multe sisteme de calcul independente interconectate în scopul execuției unor task-uri comune, în variantă paralelă;
- ➡ **Arii de micro-nuclee paralele** – tehnica mai este numită și CUDA de către cei de la Nvidia, sau *stream processors* de către cei de la AMD, și este utilizată la plăcile grafice;



# 7. Multi-threading (programarea concurentă) în Java

- **Java** = limbaj de programare multi-threading;
- Există cel puțin un fir de execuție (firul metodei *main( )*);
- Firele de execuție partajează resursele proceselor (memorie, fișiere, etc.);

⇒ Avantaj: procesare paralelă, viteză mărită de calcul;

⇒ Dezavantaj: probleme de acces simultan la aceleași resurse;

- **Suport initial pentru multi-threading în Java:**
  - interfața *java.lang.Runnable*;
  - clasa *java.lang.Thread*;
  - clasa *java.lang.ThreadGroup*;
- **Crearea unui fir de execuție;**
- **Clasa Thread:**
  - se creează o clasă derivată din clasa *java.lang.Thread*;
  - se redefineste metoda *public void run( )* moștenită din clasa Thread în clasa derivată;
  - se instanțiază clasa definită;
  - se pornește thread-ul instanțiat prin apelul metodei *start()*;

```

1  public class ThreadExample1 extends Thread {
2
3      public void run() {
4          System.out.println("My name is: " + getName());
5      }
6
7      public static void main(String[] args) {
8          ThreadExample1 t1 = new ThreadExample1();
9          t1.start();
10
11          System.out.println("My name is: " + Thread.currentThread().getName());
12      }
13
14  }

```

➡ ***Thread-0*** - numele thread-ului creat;

```

1  My name is: Thread-0
2  My name is: main

```

➡ ***main*** - numele thread-ului principal care pornește în program;

# Interfața Runnable

```
1  public class ThreadExample2 implements Runnable {  
2  
3      public void run() {  
4          System.out.println("My name is: " + Thread.currentThread().getName());  
5      }  
6  
7      public static void main(String[] args) {  
8          Runnable task = new ThreadExample2();  
9          Thread t2 = new Thread(task);  
10         t2.start();  
11  
12         System.out.println("My name is: " + Thread.currentThread().getName());  
13     }  
14  
15 }
```

## Controlul firelor de execuție:

### ⇒ Pornirea și oprirea firului de execuție:

*public void start()* – executarea corpului metodei *run()*;

*public void stop()* – metoda depreciată; este de preferat ca metoda *run()* să își încheie execuția;

### Punerea pe pauză pentru intervale de timp determinate:

⇒ *public void sleep(long ms);*  
*public void sleep(long ms, int ns);*

```
1 public class NumberPrint implements Runnable {
2
3     public void run() {
4
5         for (int i = 1; i <= 5; i++) {
6
7             System.out.println(i);
8
9             try {
10
11                 Thread.sleep(2000);
12
13             } catch (InterruptedException ex) {
14                 System.out.println("I'm interrupted");
15             }
16         }
17     }
18
19     public static void main(String[] args) {
20         Runnable task = new NumberPrint();
21         Thread thread = new Thread(task);
22         thread.start();
23     }
24
25 }
```

```

1  public class ThreadInterruptExample implements Runnable {
2
3      public void run() {
4          for (int i = 1; i <= 10; i++) {
5              System.out.println("This is message #" + i);
6
7              try {
8                  Thread.sleep(2000);
9                  continue;
10             } catch (InterruptedException ex) {
11                 System.out.println("I'm resumed");
12             }
13         }
14     }
15
16     public static void main(String[] args) {
17         Thread t1 = new Thread(new ThreadInterruptExample());
18         t1.start();
19
20         try {
21             Thread.sleep(5000);
22             t1.interrupt();
23
24         } catch (InterruptedException ex) {
25             // do nothing
26         }
27     }
28 }
29

```



*public void interrupt();*  
 întreruperea unui thread poate fi folosită  
 pentru a opri sau a relua execuția acelu  
 thread de către alt thread;

```

1  public class ThreadJoinExample implements Runnable {
2
3      public void run() {
4          for (int i = 1; i <= 10; i++) {
5              System.out.println("This is message #" + i);
6
7              try {
8                  Thread.sleep(2000);
9              } catch (InterruptedException ex) {
10                 System.out.println("I'm about to stop");
11                 return;
12             }
13         }
14     }
15
16     public static void main(String[] args) {
17         Thread t1 = new Thread(new ThreadJoinExample());
18         t1.start();
19
20         try {
21             t1.join();
22
23         } catch (InterruptedException ex) {
24             // do nothing
25         }
26
27         System.out.println("I'm " + Thread.currentThread().getName());
28     }
29 }
30

```

```

1  This is message #1
2  This is message #2
3  This is message #3
4  This is message #4
5  This is message #5
6  This is message #6
7  This is message #7
8  This is message #8
9  This is message #9
10 This is message #10
11 I'm main

```

- *public void join();*
- *public void join(long ms, int ns);*  
 – este folosit în cazul în care thread-ul curent trebuie să aștepte alte thread-uri să-și termine execuția;

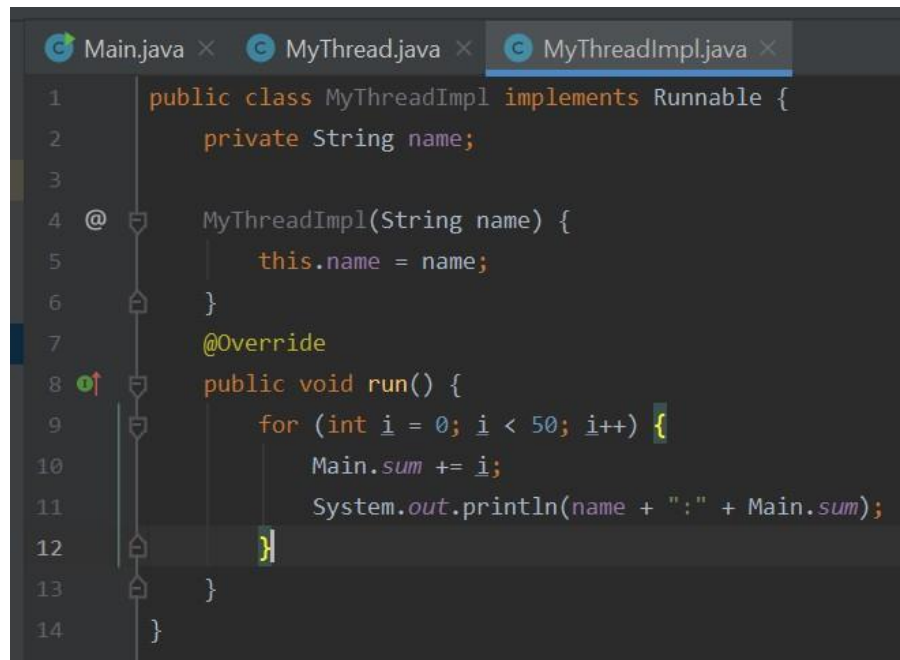


# Programare paralelă-Thread

A screenshot of an IDE window with three tabs: Main.java, MyThread.java (selected), and MyThreadImpl.java. The code in MyThread.java defines a class that extends Thread. It has a private String attribute named 'name'. The constructor 'MyThread(String name)' sets 'this.name = name;'. The 'run()' method contains a for loop from 0 to 49, incrementing 'Main.sum' and printing the thread's name and the current sum.

```
1 public class MyThread extends Thread {
2     private String name;
3
4     MyThread(String name) {
5         this.name = name;
6     }
7     public void run() {
8         for (int i = 0; i < 50; i++) {
9             Main.sum += i;
10            System.out.println(name + ":" + Main.sum);
11        }
12    }
13 }
14 }
```

# Programare paralelă-Runnable



```
1 public class MyThreadImpl implements Runnable {
2     private String name;
3
4     @ MyThreadImpl(String name) {
5         this.name = name;
6     }
7     @Override
8     public void run() {
9         for (int i = 0; i < 50; i++) {
10             Main.sum += i;
11             System.out.println(name + ":" + Main.sum);
12         }
13     }
14 }
```

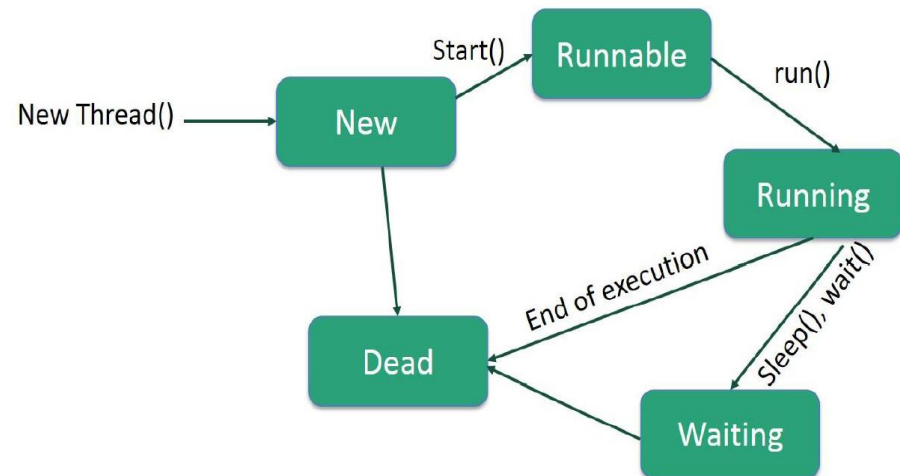
# Programare paralelă-main

```
Main.java × MyThread.java × MyThreadImpl.java ×
1 ▶ public class Main {
2
3     static int sum = 0;
4
5 ▶ public static void main(String[] args) {
6     MyThread thread1 = new MyThread( name: "Thread1");
7     thread1.start();
8     MyThread thread2 = new MyThread( name: "Thread2");
9     thread2.start();
10 }
11 }
```

```
Thread2:0
Thread1:0
Thread2:1
Thread1:2
Thread2:4
Thread1:6
Thread2:9
Thread1:12
Thread2:16
Thread1:20
Thread2:25
Thread1:30
```

- ***Stările unui thread:***

- ***New*** – thread-ul a fost creat, dar nu s-a pornit;
- ***Runnable*** – thread-ul este inițializat și poate fi pornit sau este deja pornit;
- ***Running*** – thread-ul este în etapa de execuție a unui task (starea spre care aspiră toate firele de execuție);
- ***Waiting*** – thread-ul este suspendat temporar și așteaptă un semnal pentru a reîncepe execuția;
- ***Timed Waiting*** – thread-ul este suspendat temporar de către programator, dar la intervale fixe de timp;
- ***Dead*** – un thread intră în această stare când și-a terminat execuția (acesta nu mai poate fi repornit);



- **Grupuri de thread-uri**

- Clasa *java.lang.ThreadGroup***

- tratarea unitară a firelor de execuție care sunt membre în grup;
    - ThreadGroup este o clasă care grupează anumite thread-uri ca o singură unitate și permite efectuarea unor operații pe un grup ca și un întreg (ansamblu) mai bine decât fiecare thread separat;
    - numele grupului trebuie specificat în momentul creării sale;

- **Prioritațile thread-urilor**

- fracțiunea de timp alocat rulării din timpul total de rulare;
- fiecare thread are o valoare a priorității care sugerează programatorului cât de mult trebuie să aibă grijă în cazul în care mai multe thread-uri rulează;
- atunci când se creează un thread nou, acesta are aceeași prioritate cu cel din metoda *main()*;
- prioritatea poate fi setată prin apelarea *Thread.setPriority(int priorityLevel)*;
- există 3 niveluri de prioritate:

*Thread.MIN\_PRIORITY = 1*

*Thread.NORM\_PRIORITY = 5* // valoare implicită

*Thread.MAX\_PRIORITY = 10*

- se poate seta totuși orice valoare de la 1 la 10;

- **Thread-uri Daemon**

- Java definește două tipuri de thread-uri: thread-uri normale (user thread-uri) și thread-uri Daemon (în serviciul altor thread-uri);

- implicit atunci când se creează un thread nou acesta este user thread;

- Java Virtual Machine (JVM) nu își termina execuția dacă există încă user thread-uri care rulează, dar se termină dacă există doar thread-uri Daemon;

- un thread Daemon poate fi creat prin apelul metodei *Thread.setDaemon(true)*, iar starea acestuia poate fi verificată folosind metoda *isDaemon()*;

- **Sincronizarea thread-urilor**

- Posibilitatea accesării simultane de către mai multe fire de execuție a unor resurse (variabile, metode);
- Thread-urile au o memorie comună pe care o împart, astfel intervine conceptul de resource *lock* (*mutex* – *mutual excluson*) pentru accesul secvențial la resurse;
- Într-o aplicație multi-threading există posibilitatea ca mai multe thread-uri să acceseze simultan aceleași date, fapt care conduce la apariția unei stări inconsistente a datelor (corupția datelor);



- Există următoarele situații:

- **deadlock** -> situația în care două sau mai multe fire de execuție se așteaptă reciproc pe termen nedeterminat;

- **starvation** -> situația în care un fir de execuție așteaptă (fără rezultat) accesul la resurse partajate;

-> poate apărea din următoarele motive:



-Thread-urile sunt blocate la infinit deoarece unui thread îi este necesar prea mult timp pentru a executa o secvență de cod sincronizată (ex: operații de intrari/ieșiri)



-Un thread nu primește timp de execuție de la procesor deoarece are o prioritate prea mică în comparație cu alte thread-uri;

- ***livelock*** -> situația în care un fir de execuție reacționează la rularea altui fir de execuție (thread-urile nu progresează datorită faptului că își cedează reciproc execuția);

-> spre deosebire de starea *deadlock*, în această situație thread-urile nu se vor bloca;

-> ex: 2 persoane care doresc să treacă simultan prin același coridor;

- **Monitor (Semafor)**

**Monitor** = un obiect care asigură că o variabilă partajată poate fi accesată într-un moment dat de cel mult un fir de execuție (*semafor*);

-> în cazul în care resursa este accesată, se pune un *lock* pe aceasta, astfel încât să împiedice accesul altor fire de execuție la ea;

-> în momentul în care resursa este eliberată, “*lock-ul*” va fi eliminat pentru a permite accesul altor fire de execuție;

Monitorul este numit și *semafor*, iar programatorii consideră că firul păstrează monitorul pentru acel moment.

Un thread va utiliza un monitor pentru o perioadă limitată de timp.

Dacă monitorul nu este disponibil, firul va fi suspendat într-o stare de așteptare.

Monitorul este o construcție de *sincronizare* care permite thread-urilor să aibă atât excludere reciprocă – *mutual exclusion* (folosind *lock-uri*), cât și *co-operare* (comunicare inter-thread sau cooperare), adică capacitatea de a face thread-urile să aștepte ca anumite condiții să fie adevărate (folosind un mecanism *wait/notify*).

- **Thread Pool**

-> utilizat pentru a porni o mulțime de thread-uri de durată scurtă pentru a utiliza eficient resursele și de a crește astfel performanța;

-> păstrează un număr de thread-uri inactive pregătite de executarea task-urilor necesare (după ce un thread termină de executat un task, aceasta rămâne inactiv în pool și așteaptă să fie selectat pentru a executa noi task-uri);

-> se poate defini un număr limitat de thread-uri în *pool*, util pentru a preveni supraîncărcarea;

## -Tipuri de thread pool-uri:

**\**Catched thread pool*:** păstrează un număr de thread-uri active și creează altele noi dacă este nevoie;

**\**Fixed thread pool*:** limitează numărul maxim de thread-uri concurente. Task-urile adiționale sunt plasate într-o coadă de așteptare;

**\**Single-threaded pool*:** păstrează doar un thread care execută un task la un moment dat;

**\**Fork/join pool*:** un thread pool special care utilizează framework-ul Fork/Join pentru a se folosi de avantajele procesorului care împarte o sarcină mai complexă în mai multe părți mai mici într-un mod recursiv;

- **Executori – introdusi in package-ul concurrent cu Java SE5**

**Executor** = obiect care este responsabil pentru gestionarea thread-urilor și de executarea sarcinilor *Runnable*; A fost extins odata cu evolutia Java.

```
1 Thread t = new Thread(new RunnableTask());  
2 t.start();
```

Se poate înlocui cu:

```
1 Executor executor = anExecutorImplementation;  
2 executor.execute(new RunnableTask1());  
3 executor.execute(new RunnableTask2());
```

- **Există trei interfețe de bază pentru executori:**

\****Executor***: interfață simplă care permite rularea de procese asociate firelor de execuție;  
- are definită metoda *void execute (Runnable command)*;

\****ExecutorService (derivată din Executor)***: adaugă facilități de gestionare a ciclului de viață a proceselor și a executorilor;  
- adaugă metoda *Future<?> submit(Runnable task)* care permite utilizarea obiectului *Future* returnat;

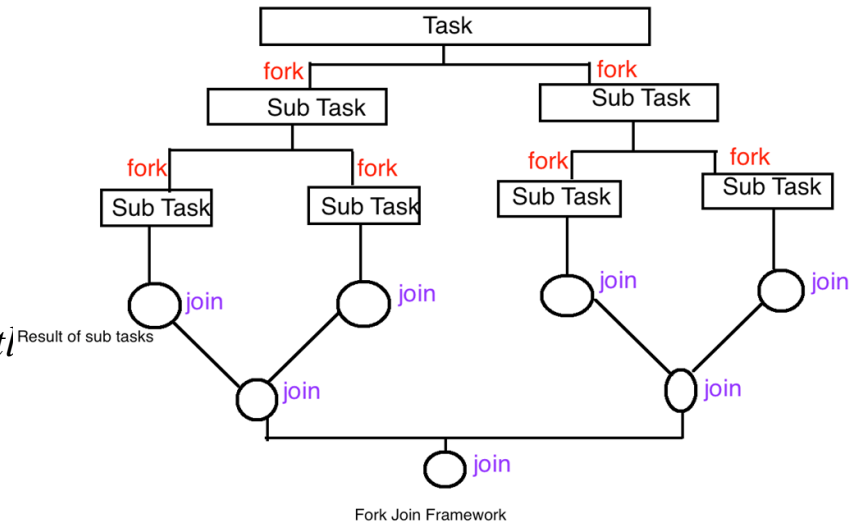
\****ScheduledExecutorService (derivată din Executor Service)***: permite în plus executarea viitoare sau periodică a proceselor;  
- adaugă metoda *<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)* care permite rularea sarcinilor la un moment de timp viitor;

- **Framework-ul *Fork/Join***

**Framework-ul *Fork/Join*** = set de API-uri care permite programatorului să se folosească de avantajele procesării paralele prin intermediul procesoarelor multi-core;

Următorul pseudocode ilustrează principiul de funcționare al framework-ului *Fork/Join*:

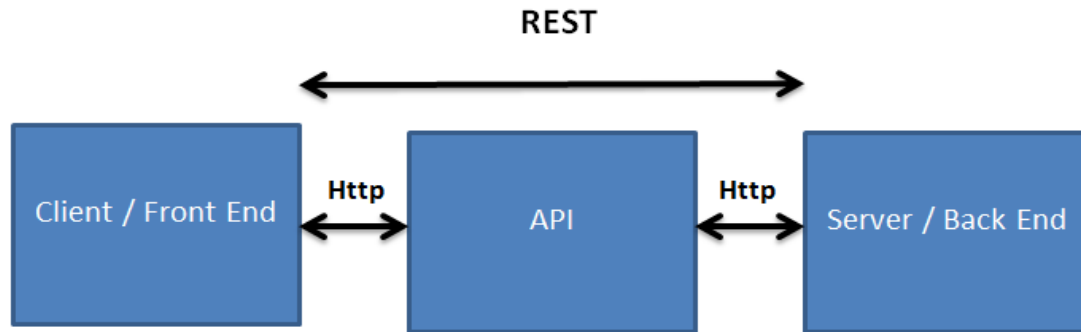
```
if (problemSize < threshold)
    solve problem directly
else {
    break problem into subproblems
    recursively solve each subproblem
    combine the results
}
```





- Framework-ul ***Fork/Join*** utilizat în: procesarea imaginilor, procesarea video, cantități mari de date, etc;
- Pentru a spori performanțele acestui framework, *fork/join* are propriul thread pool care gestionează thread-urile lucrătoare, *forkJoinPool*. Acest pool creează un thread pentru fiecare sub-task supus unei execuții, pentru a se folosi la maxim de procesor, printr-un algoritm de tip *work-stealing*;

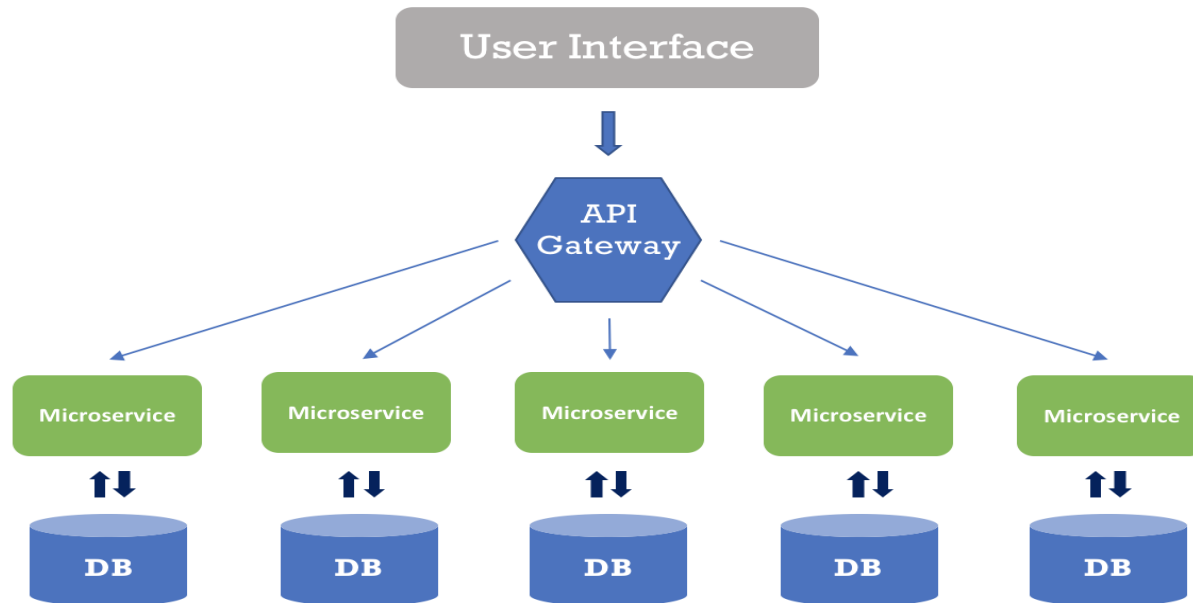
# Multithreading în aplicații web



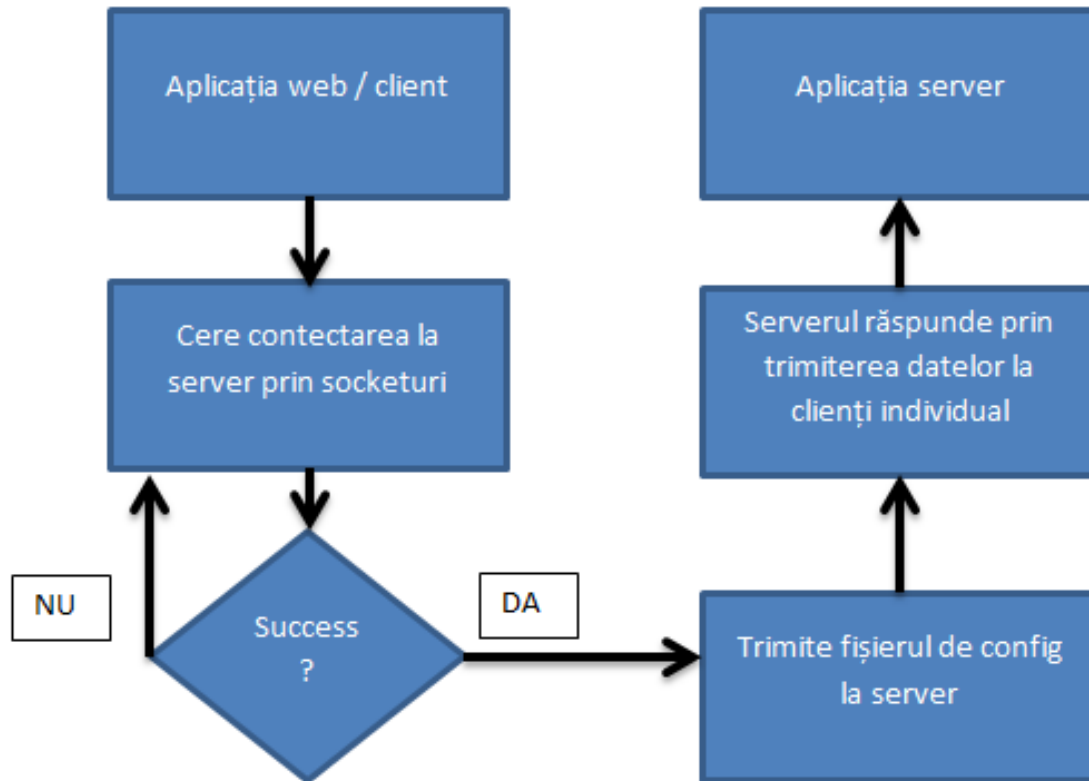
- **HTTP** = Hypertext Transfer Protocol, protocol de comunicare în internet.
- **API** = Application Programming Interface, este o interfață / protocol / “contract” de comunicare între client și server.
- **REST** = Representational State Transfer, un set de constrângeri expuse pentru un contract comun de comunicare (folosește HTTP)

# Multithreading în aplicații web

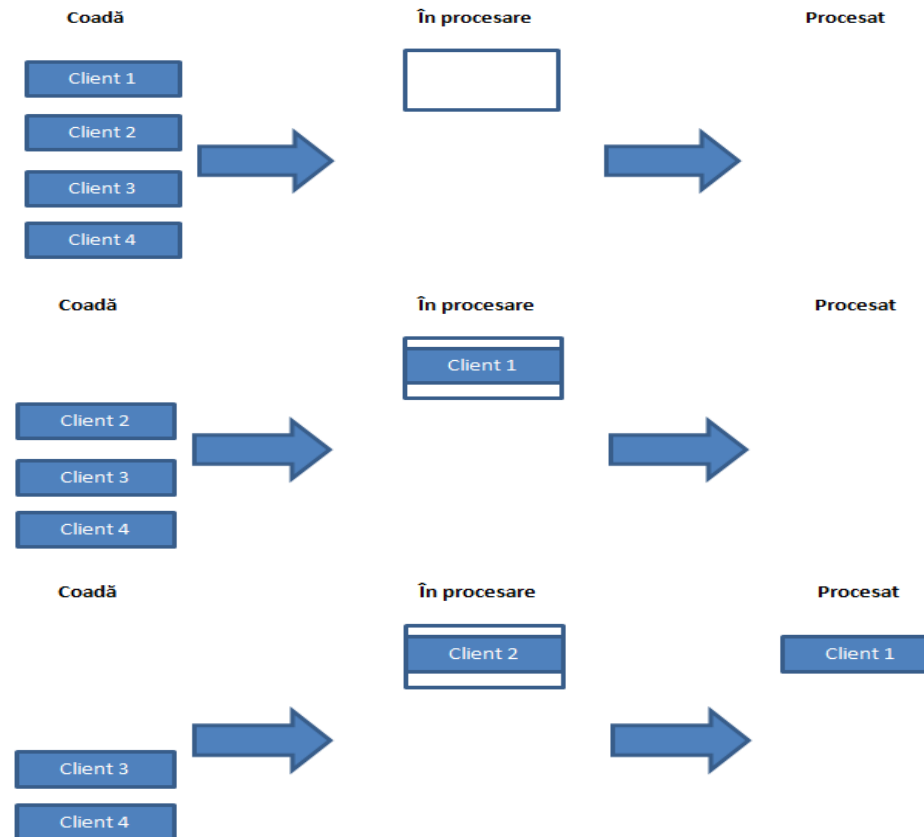
- **Microserviciu** = tehnică de proiectare orientată spre servicii simple care comunică între ele folosind un protocol comun (REST).



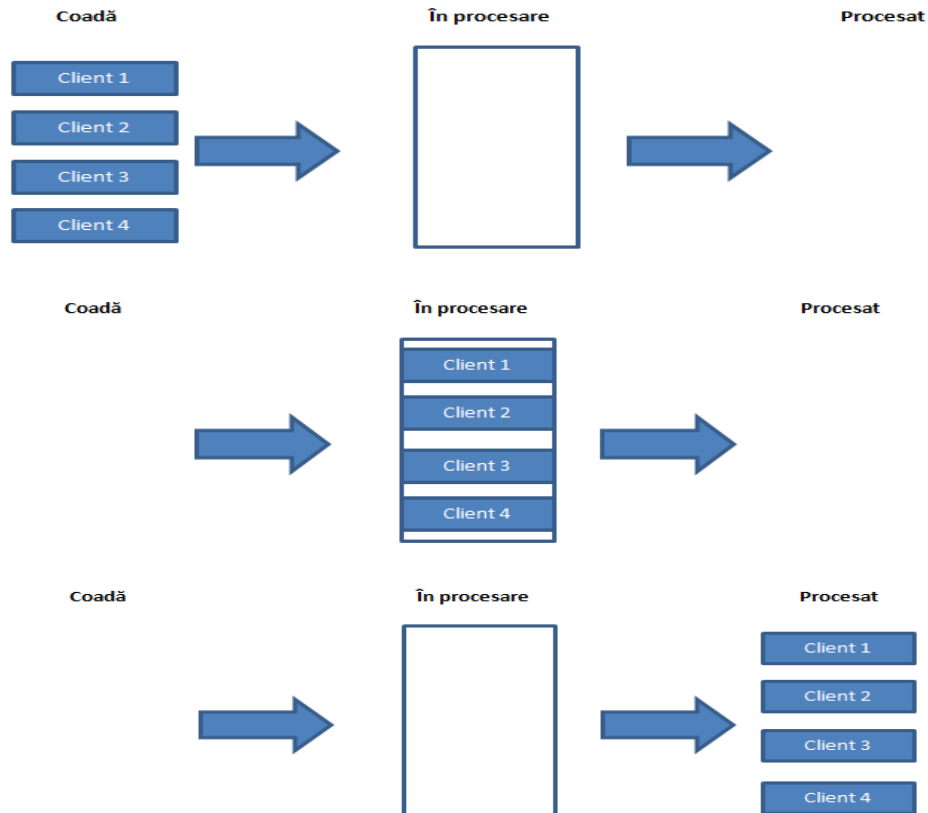
# Aplicația multi-core streaming



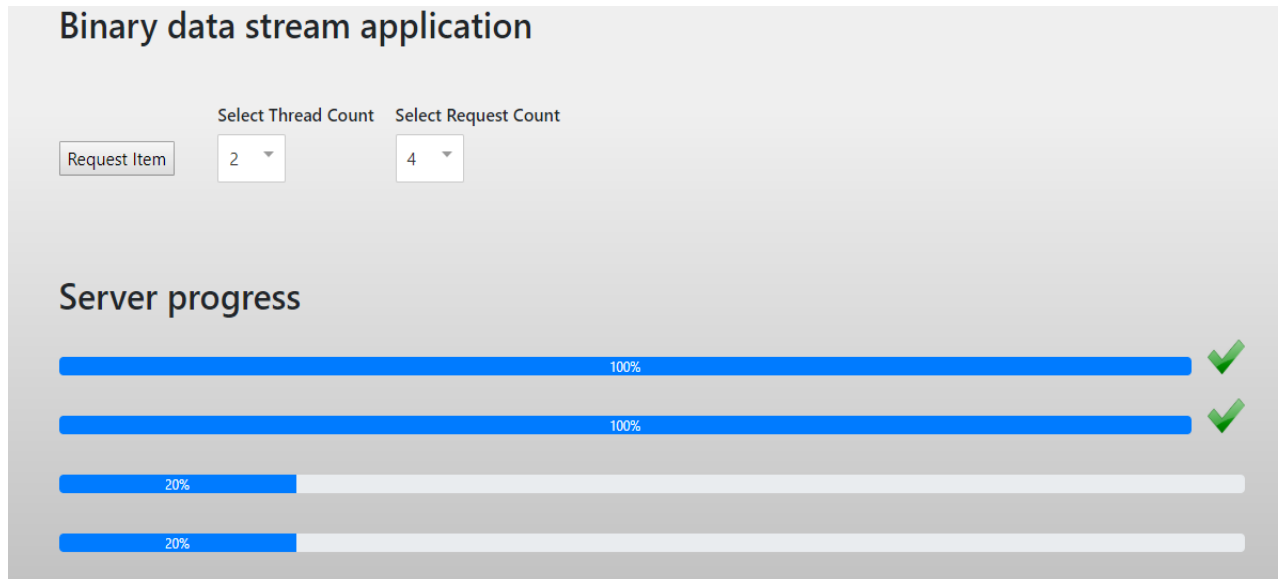
# Aplicația multi core streaming



# Aplicația multi core streaming



# Aplicația multi core streaming



# Aplicația multi core streaming

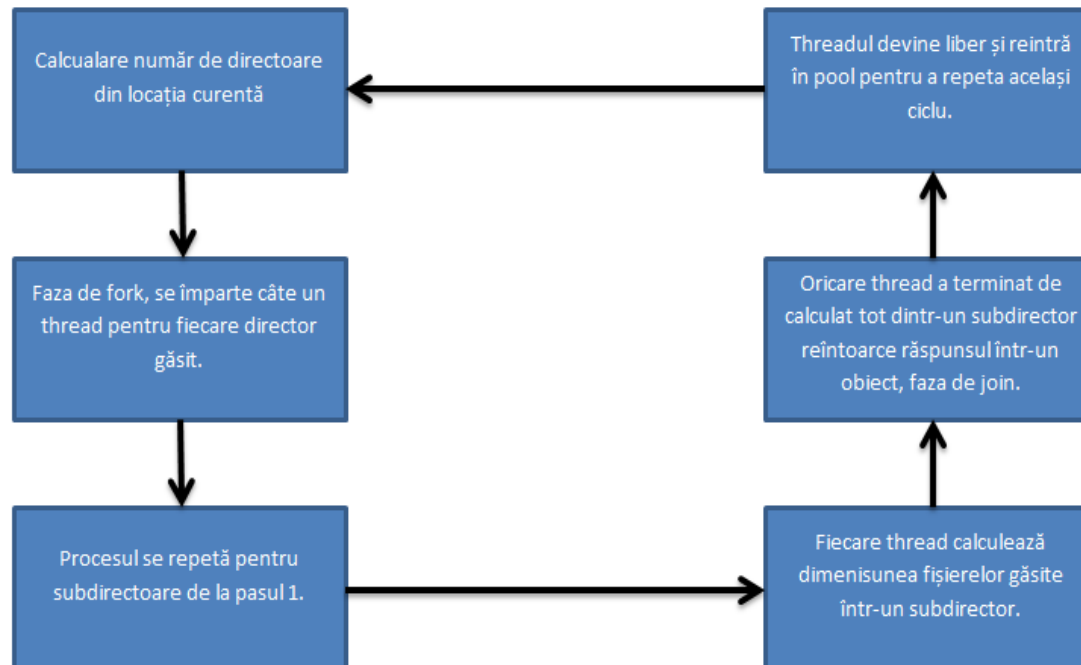
- Rezultate:
- 1 core 4 clienți **7**20 sec
- 2 core 4 clienți **7**10 sec
- 3 core 4 clienți **7**10 sec
- 4 core 4 clienți **7**5 sec



# Aplicația Fork / Join Recursiv

- Microserviciu în Java.
- Împărțirea lucrului (task) între mai multe threaduri (faza de *fork*).
- Executarea procesării de către fiecare thread individul (execuția taskului).
- Adunarea rezultatelor de la fiecare thread (faza de *join*).

# Aplicația Fork / Join Recursiv



# Aplicația Fork / Join Recursiv

Conținut de 4 GB 120% mai rapid

multi core

```
: Size of 'D://BACKUP': 3993188896 bytes (in 4.409 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 3.765 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 3.448 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 3.112 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 3.358 s)
: Average run time is 3.62 s
: Size of 'D://BACKUP': 3993188896 bytes (in 1.834 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 1.685 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 1.495 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 1.606 s)
: Size of 'D://BACKUP': 3993188896 bytes (in 1.584 s)
: Average run time is 1.64 s
: Overall performance difference, 1 thread took 3.62 s, recursive multicore took 1.64 s, 120.53% performance difference
```

Continut de 337 GB 350% mai rapid

multi core

```
: Size of 'F:///': 337159798151 bytes (in 96.633 s)
: Size of 'F:///': 337159798151 bytes (in 8.347 s)
: Size of 'F:///': 337159798151 bytes (in 8.051 s)
: Size of 'F:///': 337159798151 bytes (in 8.038 s)
: Size of 'F:///': 337159798151 bytes (in 7.975 s)
: Average run time is 25.81 s
: Size of 'F:///': 337159798151 bytes (in 6.021 s)
: Size of 'F:///': 337159798151 bytes (in 5.949 s)
: Size of 'F:///': 337159798151 bytes (in 5.184 s)
: Size of 'F:///': 337159798151 bytes (in 5.22 s)
: Size of 'F:///': 337159798151 bytes (in 6.296 s)
: Average run time is 5.73 s
: Overall performance difference, 1 thread took 25.81 s, recursive multicore took 5.73 s, 350.10% performance difference
```

# Aplicația GPU render vs CPU render

- Procesoarele grafice au sute de “mini” core-uri(CUDA, stream processor).
- Calcule simple dar foarte multe (GPU) mai complexe dar mai putine (CPU).
- GPU este folosit în randare video 2D, 3D, realitate virtuală și augmentată dar și în criptografie.
- Un proces foarte des întâlnit pentru măsurarea performanțelor este calcularea de numere prime (100k).

# Aplicația GPU render vs CPU render

```
Device 9847360
  vendor = NVIDIA Corporation
  type:GPU
  maxComputeUnits=5
  maxWorkItemDimensions=3
  maxWorkItemSizes={1024, 1024, 64}
  maxWorkWorkGroupSize=1024
  globalMemSize=2147483648
  localMemSize=49152
-----
Device 462413760
  vendor = Intel(R) Corporation
  type:GPU
  maxComputeUnits=20
  maxWorkItemDimensions=3
  maxWorkItemSizes={512, 512, 512}
  maxWorkWorkGroupSize=512
  globalMemSize=1708759450
  localMemSize=65536
-----
Device 10131040
  vendor = Intel(R) Corporation
  type:CPU
  maxComputeUnits=4
  maxWorkItemDimensions=3
  maxWorkItemSizes={8192, 8192, 8192}
  maxWorkWorkGroupSize=8192
  globalMemSize=8467283968
  localMemSize=32768
```

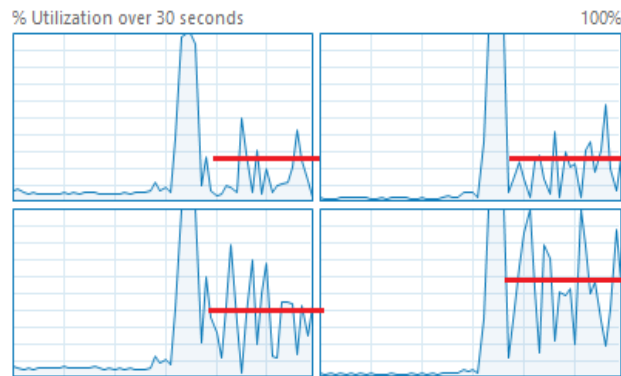
```
CPU single core
time taken: 21428 ms
CPU multi core
time taken: 7190 ms
GPU multi core
time taken: 1346 ms
Relative performance difference 1491.98%
```

Diferența de performanță între CPU single core și GPU este de **1491.98%** iar între CPU multi core și GPU este de **530%**.

Biblioteca folosită pentru programare GPU este openCL (Computing Library)

# Aplicația GPU render vs CPU render

CPU Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz



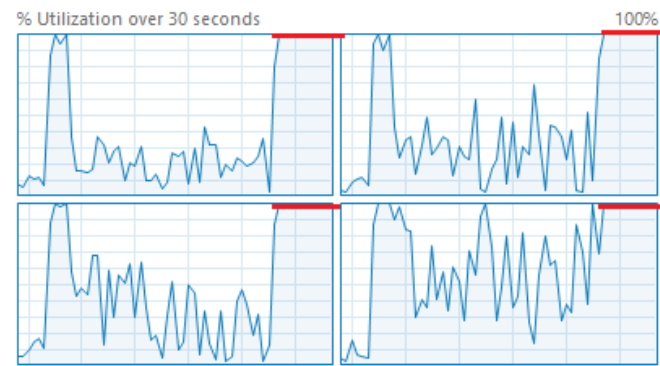
Utilization  
32%

Speed  
3.33 GHz

Maximum speed: 2.80 GHz  
Sockets: 1  
Cores: 2

**Single  
core**

CPU Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz



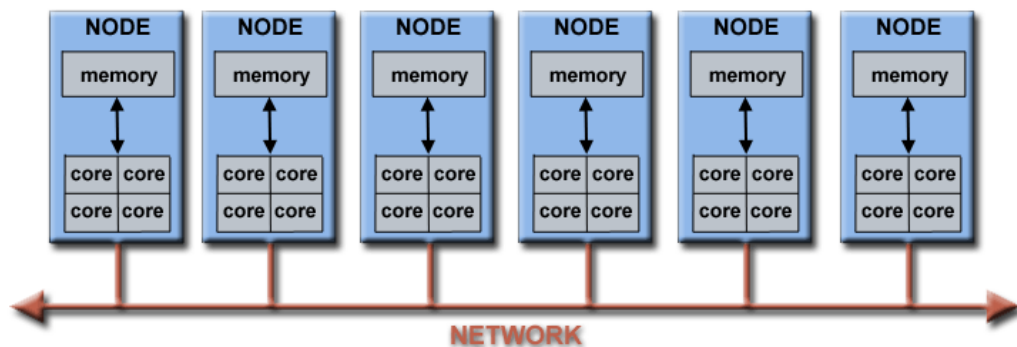
Utilization  
100%

Speed  
3.28 GHz

Maximum speed: 2.80 GHz  
Sockets: 1  
Cores: 2

**Multi  
core**

## 8. Concluzii



- Cele mai utilizate limbaje de programare sunt cele din domeniul server side, capabile să ruleze aplicații pe mai multe core-uri;
- Există numeroase framework-uri în Java dedicate programării paralele și multi-core;
- Sistemele multi-processor/core prezintă și disponibilitate crescută, pentru că se pot defecta unele procesoare/core-uri și sistemul trebuie să funcționeze cu procesoarele/core-urile rămase;