2. Java Concurrency - mecanisme pentru paralelism și concurență

Versiunea JDK 5.0 a constituit un pas major în programarea concurentă, astfel mașina virtuală Java a fost îmbunătățită semnificativ pentru a permite claselor să profite de suportul pentru concurență oferit la nivel hardware prin sisteme multicore. Deasemenea, un set bogat de noi clase au fost adăugate pentru a face mai ușoară dezvoltarea de aplicații concurente. Pachetul *java.util.concurent* aduce un set bine testat și foarte performant de funcții și structuri de date pentru *concurență și paralelism.*

2.1.Obiective

- Studiul structurilor de date și mecanismelor de programare concurentă /paralelă moderne din Java Concurrency
- Implementarea unor aplicații ilustrative cu aceste funcții și structuri de date și analiza performanței.

2.2.Concepte

Îmbunătățirile aduse din perspectiva suportului pentru concurență pot fi structurate în 3 categorii:

a) Modificări la nivelul mașinii virtuale java
Procesoarele moderne oferă suport hardware pentru concurență, în forma unor instrucțiuni *compare-and-swap* (CAS). Înainte de JDK 5.0, singura primitivă în limbajul Java pentru coordonarea accesului între thread-uri era *sincronizarea (synchronized).* Prin expunerea CAS se oferă posibilitatea dezvoltării unor clase Java foarte scalabile pentru aplicații ce solicită o astfel de abordare.

b) Clase utilitare de nivel scăzut
De exemplu clasa *ReentrantLock* oferă funcționalitate asemănătoare cu soluția *synchronized,* dar cu un control mai bun asupra blocării (temporizare lacăte - timed locks, verificare lacăte - lock polling, etc.) și astfel o mai bună scalabilitate.

c) Clase utilitare la nivel înalt
Clase care implementează: monitoare, semafoare, lacăte, bariere, thread-pools și colecții de date cu acces thread-safe. Acestea sunt oferite dezvoltatorilor de aplicații pentru a construi diverse soluții.

2.2.1. Colecții de date pentru mecanismele de concurență

Înainte de JDK 5.0 principalul mecanism pentru asigurarea faptului că o porțiune de cod este thread-safety ,altfel spus accesibilă unui singur thread la un moment dat era primitiva *synchronized*. Pachetul *java.util.concurrent* oferă noi primitive pentru asigurarea acestei proprietăți, precum și o serie de clase utilitare care nu necesită sincronizare adițională în codul aplicației, simplificând astfel codul aplicației.

Framework-ul Collections este un framework flexibil pentru reprezentarea colecțiilor de obiecte, folosind interfețele de bază *Map, List, Set.* Câteva dintre implementări sunt Thread-Safe (*Hashtable, Vector*), celelalte pot fi transformate în thread-safe cu ajutorul colecțiilor, și anume *Collections.synchronizedMap(), Collections.synchronizedList(),Collections.synchronizedSet().*

Pachetul *java.util.concurrent* adaugă noi colecții concurente: *ConcurrentHashMap, CopyOnWriteArrayList* și *CopyOnWriteArraySet*. Scopul acestor clase este să îmbunătățească performanța și scalabilitatea oferită de tipurile de colecții de bază.

Iteratorii s-au schimbat de asemenea în JDK 5.0. Dacă până la versiunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, noii iteratorii oferă o vedere consistentă asupra colecției, chiar dacă aceasta se schimbă în timpul iterării.

Deasemenea,,*CopyOnWriteArrayList* și *CopyOnWriteArraySet* sunt versiuni îmbunătățite ale structurilor Vector și ArrayList. Îmbunătățirile sunt aduse în special la nivelul iterației,astfel, dacă în timpul parcurgerii unui *Vector* sau a unui *ArrayList* colecția este modificată, se va arunca o excepție, iar noile clase rezolvă această problemă.
Se oferă două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.

- *Cozi (Queue)* Există două implementări principale, care determină ordinea în care elementele unei cozi sunt accesate: *ConcurrentLinkedQueue* (acces FIFO) și *PriorityQueue* (acces pe bază de priorități).
- *Cozi cu blocare (BlockingQueue*) Acest tip de cozi sunt folosite atunci când se dorește blocarea unui thread, în situația în care anumite operații pe o coadă nu pot fi executate. Un exemplu ar fi cazul în care consumatorii extrag mai greu din coadă informația decât ea este plasată în coadă de producători. Prin folosirea *BlockingQueue* se blochează automat producătorii până când se eliberează un element din coadă. Implementările interfeței *BlockingQueue* sunt: *LinkedBlockingQueue, PriorityBlockingQueue, ArrayBlockingQueue si SynchronousQueue.*

2.2.2.Thread Pools și Framework-ul Executor

Un mecanism clasic pentru managementul unui grup mare de task-uri este combinarea unei cozi de lucru (work *queue)* cu un set de threaduri (*thread pool)*. Astfel, *work queue* este o coadă de taskuri ce trebuie procesate, iar un *thread pool* este o colecție de thread-uri care extrag sarcini din coada și le execută. Când un *thread worker* termină o sarcină, se întoarce la coadă pentru a vedea dacă mai există sarcini de executat, iar dacă da, extrage sarcina din coada și o execută.

Pachetul *java.util.concurrent* conține un întreg framework pentru manangementul execuției task-urilor care implementează *Runnable.* Un aspect foarte important al folosirii framework-ului *Executor* este faptul că se permite *decuplarea lansării task-urilor de politica de execuție a lor*. Aceasta permite schimbarea politicii de execuție foarte ușor, fără a fi nevoie de modificări majore în cod.

Interfata Executor este foarte simplă:

```
public interface Executor {
        void execute (Runnable command); }
```

Politica de execuție a task-urilor depinde de implementarea de *Executor* aleasă.
Clasa *Executors* oferă diverse metode statice pentru obținerea de instanțe de diferite implementări de executori.

```
Executors.newCachedThreadPool()
Executors.newFixedThreadPool(int n)
Executors.newSingleThreadExecutor()
```

Executorii returnați de primele două metode sunt instanțe ale clasei *ThreadPoolExecutor*, care poate fi intens customizată în funcție de necesități.

Interfața Future .Această interfață permite reprezentarea unui task care poate s-a terminat, care este în execuție sau care încă nu a început să fie executat. Prin intermediul interfeței *Future* se pot anula taskuri care încă nu s-au terminat de executat, se poate verifica dacă un task s-a încheiat sau a fost anulat și se poate aștepta după rezultatul returnat de un task. *FutureTask* este o implementare a interfeței *Future* și are constructori care permit execuție peste o instanță de *Runnable*. Deoarece și *FutureTask* implementează *Runnable*, task-ul obținut poate fi direct transmis către un *Executor*.

2.2.3. Clase pentru sincronizare

O altă categorie de clase folositoare adăugate în *java.util.concurrent* este categoria claselor de sincronizare. Aceste clase coordonează și controlează fluxul execuției pentru unul sau mai multe thread-uri. Exemple de clase de sincronizare: *Semaphore,Mutex, CyclicBarrier, CountdownLatch, și Exchanger* .

Semaphore - implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate. Acesta este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă.

Mutex - un caz special de semafor, cu o singură permisie (permite doar acces exclusiv).

CyclicBarrier- oferă un ajutor de sincronizare: permite unui set de thread-uri să aștepte ca întreg setul să ajungă la o barieră comună.

CountdownLatch - oarecum similar cu *CyclicBarrier* prin faptul că permite coordonarea unui grup de thread-uri. Diferența este că atunci când un thread ajunge la barieră, nu se blochează ci doar decrementează valoarea inițială a lacătului. Este util când o problemă este divizată între mai multe thread-uri, fiecare realizând, o parte. Când un thread termină de rezolvat, decrementează contorul. Thread-ul de control poate verifica dacă lacătul a ajuns la valoarea 0, ceea ce înseamnă că toate thread-urile au terminat execuția.

Exchanger- facilitează schimbul bidirecțional între două thread-uri care cooperează. Este similar unei bariere de tip *CyclicBarrier* cu o valoare inițială de 2, la care se adaugă posibilitatea ca thread-urile să facă schimb de informații când ajung la barieră.

2.2.4.Facilități de nivel scăzut

Limbajul Java are o facilitate integrată de blocare și anume specificatorul *synchronized.* Când un thread achiziționează un monitor, alte thread-uri se vor bloca dacă încearcă să achiziționeaze același monitor, până când primul thread eliberează monitorul. Sincronizarea asigură și că valorile variabilelor modificate de un thread sunt vizibile thread-urilor care preiau accesul aceluiași lock ulterior.

Interfața *Lock* este o generalizare a funcționalității oferite de synchronized, permițând diverse implementări care adaugă o serie de funcționalități noi cele mai importante fiind: *timed waits, interruptible waits, lock polling, multiple condition-wait sets per lock si non-block-structured locking*.

ReentrantLock. Este o implementare a interfeței *Lock*, mult mai scalabilă, cu utilizare recursivă. Oferă viteză îmbunătățită la gestiunea mai multor thread-uri care vor să acceseze aceași resursă.

Conditions. La fel cum interfața *Lock* este generalizarea pentru *synchronized*, interfața *Condition* este o generalizare a metodelor *wait()* si *notify()* din clasa *Object*.

Variabile atomice. Deși sunt foarte rar folosite direct de utilizatori, unele dintre cele mai semnificative clase nou introduse, sunt clasele pentru variabile atomice: (*AtomicInteger, AtomicLong, AtomicRefer*, etc.). Aceste clase expun îmbunătățirile aduse mașinii virtuale, permitand operații atomice de citire-scriere. Aproape toate clasele din *java.util.concurrent* sunt construite peste *ReentrantLock* care la

rândul ei este construită utilizând clasele *atomic-variable*. Așadar, deși sunt transparente pentru majoritatea utilizatorilor, această categorie de clase este cea care aduce principalele îmbunătățiri de scalabilitate oferite de *java.util.concurrent.*

2.2.5. Java Concurrency Framework

Java Concurrency Framework este un framework ce oferă o serie de servicii care permit utilizarea programării concurente în aplicațiile moderne.

Cele mai importante pachete pe care acesta le pune la dispoziție sunt:

- *java.util.concurrent.atomic* - ce oferă o mai mare flexibilitate în utilizarea de lacăte și condiții, în detrimentul sintaxei clasice
- *java.util.concurrent.locks* - clasele din acest pachet extind noțiunea de valoare volatilă, câmpuri și șiruri de elemente cărora, de asemenea, le oferă operații atomice

a)Executori – java.util.concurrent.Executor

Atunci când este necesar să fie rulate mai multe sarcini complexe, în paralel și să se aștepte finalizarea tuturor pentru ca mai apoi să se returneze o valoare, devine destul de dificilă conceperea unui cod bun care să le sincronizeze. Pentru a soluționa această problemă, Java introduce Executor, o interfață ce permite crearea seturilor de thread-uri, susține sincronizarea și execuția lor.

Interfața Executor declară o singură metodă - public void execute (Runnable obj). Pentru a o utiliza este nevoie de:

- implementarea interfeței
- crearea unui obiect ce implementează interfața Runnable :Executor ex = new MyExecutor ();
- apelarea metodei executate ex. execute(unObiectRunnable);

```
import java.util.concurrent.*;
public class MyExecutor implements Executor {
 public void execute(Runnable obj) {
    obj.run(); // execuția va avea loc în threadul apelant
       new Thread(obj).start(); // execuția va avea loc într-un thread distinct }}
```

Dacă este nevoie de generarea și gestionarea unui număr mai mare de thread-uri se poate crea un set de thread-uri (thread-pool). Un set de thread-uri poate fi reprezentat printr-o instanță a clasei ExecutorService. Acesta poate fi de mai multe tipuri :

- *Single Thread Executor* – un set care conține un singur thread; codul se va executa secvențial
- *Fixed Thread Pool* – un set care conține un număr fix de thread-uri; dacă un thread nu este disponibil pentru un task, acesta se pune într-o coadă și așteaptă finalizarea unui alt task
- *Cached Thread Pool* – un set care creează atâtea thread-uri cîte sunt necesare pentru executarea unui task în paralel
- *Scheduled Thread Pool* – un set creat pentru planificarea task-urilor viitoare
- *Single Thread Scheduled Pool* – un set care conține un singur thread utilizat în planificarea task-urilor viitoare.

b)Thread Factory – java.util.concurrent.ThreadFactory Cu ajutorul *Thread Factory* se creează anumite tipuri de thread-uri într-un mod standardizat. Singura metodă disponibilă este n*ewThread* (Runnable r) care se suprascrie în clasele care o implementează, pentru crearea unor thread-uri customizate. Un exemplu de utilizare al acestei interfețe îl gasiți mai jos:

```
import java.util.concurrent.*;
public class MyThreadFactory implements ThreadFactory {
public Thread newThread(Runnable r) {
       return new Thread(r); }
```

```
public static void main(String[] args) {
            MyThreadFactory    mtf    =    new
       MyThreadFactory();
       Thread t = mtf.newThread(new MyThread());
            t.start(); } }

class MyThread extends Thread
{    public void run() {
            System.out.println("Inside: " + this.getName()); } }
```

Futures – java.util.concurrent.Future <V>

```
public class FutureTaskExample {
public static void main(String[] args) {
        MyCallable callable1 = new MyCallable(1000);
       MyCallable callable2 = new MyCallable(2000);
       FutureTask<String> futureTask1 = new FutureTask<String>(callable1);
       FutureTask<String> futureTask2 = new FutureTask<String>(callable2);
       ExecutorService executor = Executors.newFixedThreadPool(2);
       executor.execute(futureTask1);
       executor.execute(futureTask2);
       while(true) {
         try
         {if (futureTask1.isDone() && futureTask2.isDone())
               {System.out.println("Done");
               executor.shutdown();
               return; }
               if(!futureTask1.isDone())
               {System.out.println("FutureTask1 output= "+futureTask1.get());}
               System.out.println("Waiting for FutureTask2 to complete");
               String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);
               if (s != null)
               {System.out.println("FutureTask2 output="+s);  }  }
         catch(InterruptedException | ExecutionException e)
         { e.printStackTrace();  }
         catch(TimeoutException e)
         {
       ……
         }}}}
```

Future reprezintă rezultatul unei acțiuni asincrone. Metodele oferite de această interfață se folosesc pentru a verifica dacă acțiunea *a luat sfârșit,* pentru a *aștepta terminarea* unei acțiuni și pentru a returna rezultatul obținut în urma execuției.

c)Cozi

Coada (FIFO – First In First Out) reprezintă o structură de date asemănătoare unei liste în care primul element introdus va fi și primul extras atunci când se execută o extragere. Începând cu Java 1.7, există mai multe tipuri de cozi implementate și care aparțin tot pachetului java.util.concurrent. Cele mai importante dintre acestea sunt: *AbstractQueue, ArrayBlockQueue, BlockingQueue, ConcurrentLinkedQueue, DelayQueue, e*tc.
Spre exemplu, coada de tipul *ArrayBlockingQueue* care implementează interfața *BlockingQueue* este utilizată în abordarea problemei ” producător-consumator”. În acest caz, coada are la bază o listă de dimensiune fixă și toate elementele cozii trebuie să fie de același tip. Constructorii clasei sunt:

- *ArrayBlockingQueue (int dim)* – creează o coadă care poate conține până la dim înregistrări
- *ArrayBlockingQueue (int dim, boolean fair)* – firele de execuție blocate sunt eliberate în ordinea blocării dacă fair = true și în mod aleator în caz contrar. Metodele acestei clase sunt:
    - *void put (o*) throws InterruptedException – introduce obiectul o în coadă, iar dacă operația nu este posibilă atunci thread-ul care a lansat-o este blocat
    - *boolean offer(o)* – introduce obiectul o în coadă și returnează true; dacă operația nu se poate efectua returnează false fără blocarea thread-ului care a lansat operația

- *peek()* – returnează elementul din vârful cozii
- *poll()* – returnează și șterge elementul din vârful cozii; dacă nu mai există elemente în coadă returnează null
- *take()* throws InterruptedException - returnează și șterge elementul din vârful cozii; dacă nu mai există elemente în coadă blochează thread-ul care a lansat operația
- *void clear()* - șterge conținutul cozii

Un exemplu de implementare a claselor Producer – Consumer devine cu aceste noi construcții:

```
public class Producer implements Runnable{
protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {
	this.queue = queue; }
@Override
public void run() {
	try {
		queue.put("1");
		Thread.sleep(1000);
		queue.put("2");
		Thread.sleep(1000);
		queue.put("3");
	} catch (InterruptedException e) {
		e.printStackTrace();}}}


public class Consumer implements Runnable{
	protected BlockingQueue queue = null;
	public Consumer(BlockingQueue queue)
	{this.queue=queue;}
	@Override
	public void run() {
	try {
		System.out.println(queue.take());
		System.out.println(queue.take());
		System.out.println(queue.take());
	} catch (InterruptedException e) {
		e.printStackTrace();}}}

public class BlockingQueueExample {
public static void main(String[] args) throws Exception {
      BlockingQueue queue = new ArrayBlockingQueue (1024);
      Producer producer = new Producer(queue);
      Consumer consumer = new Consumer(queue);
      new Thread(producer).start();
      new Thread(consumer).start();
      Thread.sleep(4000);}}
```

Programarea multithreading poate deveni uneori dificilă, mai ales când e necesară sincronizarea mai multor fire de execuție ce folosesc aceleași resurse simultan. Din acest motiv au fost introduse mai multe metode de sincronizare prezentate în cele ce urmează.

d) Excluderea reciprocă

Modificatorul *synchronized* atașat unei metode permite unui singur fir de execuție să execute, la un moment dat, o singură metodă dintre cele cu acest modificator, astfel se asociază un mecanism de lacăt fiecărui obiect. Atunci când un thread apelează o metodă synchronized, lacătul se blochează și nu permite accesul unui alt thread decât după eliberarea sa.

În acest fel se asigură excluderea reciprocă a firelor de execuție la resursele / metodele implicate. Modificatorul synchronized poate fi evitat prin utilizarea clasei *ReentrantLock()* ce implementează interfața *java.util.concurrent.Lock*. Aceasta definește următoarele metode:

- *void lock()* – închide lacătul; mai exact, orice thread care încearcă să apeleze o metodă și întâlnește lacătul blocat va fi suspendat.
- *void unlock()* – deschide lacătul
- *boolean tryLock()* – verifică dacă lacătul este blocat sau nu
- *Condition newCondition()* – creează o instanță de tip Condition atașată lacătului.

Sunt prezentate două moduri de incrementare a unei valori partajate de un număr de ori, prima folosind modificatorul synchronized, iar a doua utilizând Lock.

Metoda 1

```
public class Inc {
int n = 0;
void add()
{for (int i = 0; i < 100; i++)
{synchronized(this) n =n+1;}}}
```

Metoda 2

```
public  class
Inc {
int n = 0;
Lock l = new ReentrantLock();
void add()
{l.lock();
try {
for (int i = 0; i < 100; i++)
{n = n+1;}}
finally {
l.unlock();}}}
```

O altă modalitate de implementare a excluderii reciproce o reprezintă semafoarele, reprezentate în Java prin clasa *Semaphore*. Semafoarele se utilizează pentru a limita și controla accesul la o resursă partajată de mai multe threaduri. Înainte să obțină o resursă, un thread trebuie să obțină permisiunea de la semafor – adică validarea că resursa este disponibilă, apoi, când termină de utilizat resursa respectivă, thread-ul se întoarce la semafor pentru a semnala că aceasta este din nou disponibilă.

e)Sincronizarea prin barieră

Sincronizarea barieră presupune existența mai multor procese care execută aceeași secvență de cod care conține o instrucțiune cu rol de barieră. Poate fi vazută ca un set de thread-uri care se așteaptă reciproc să ajungă la un anumit punct comun, o barieră, după care se deblochează și își continuă execuția. Clasa *CyclicBarrier* permite programarea simplă a acestui tip de sincronizare. Se numește ciclic deoarece poate fi reutilizată după ce s-a ajuns la punctul dorit.
Constructorul clasei poate avea două forme: *CyclicBarrier (int num), Cyclic Barrier (int num, Runnable barierAction)* în care num reprezintă numărul firelor de execuție, iar la atingerea acestui prag se execută metoda *run()* din obiectul *barrierAction* transmis.

Cele mai importante dintre metodele acestei clase sunt:
- *int await()* – suspendă thread-ul până la atingerea pragului corespunzător obiectului CyclicBarrier
- *int getNumberWaiting()* – returnează numărul de threaduri suspendate la barieră
- *int getParties()* – returnează valoarea parametrului num
- *void reset()* – reinițializează bariera

Un exemplu de utilizare al clasei CyclicBarrier :

```
 public class TechLead extends Thread{
       CyclicBarrier cyclicBarrier;

       public TechLead(CyclicBarrier cyclicBarrier, String name)
       {super(name);
              this.cyclicBarrier= cyclicBarrier; }
       public void run()
       {
              try {
       Thread.sleep(3000);
       System.out.println(Thread.currentThread().getName()+    "    recruited
developer");
       System.out.println(Thread.currentThread().getName()+   "   waiting   to
complete ...");
              cyclicBarrier.await();
              System.out.println("All     finished     recruting,     "     +
             Thread.currentThread().getName()   +
              " gives offer letter to candidate");
              } catch (Exception e) {
                     e.printStackTrace();} }}


public class HRManager {
public static void main(String[] args) {
    CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
    TechLead techLead1 = new TechLead(cyclicBarrier,"John TL");
    TechLead techLead2 = new TechLead(cyclicBarrier,"Doe TL");
    TechLead techLead3 = new TechLead(cyclicBarrier,"Mark TL");
    techLead1.start();
    techLead2.start();
    techLead3.start();
    System.out.println("No work");}}
```

f)Mecanismul Fork/Join

Framework-ul *Fork/Join,* apărut odată cu versiunea Java 1.7 fiind o implementare a interfeței *ExecutorService* care oferă posibilitatea folosirii la maxim a *multiplelor procesoare disponibile*, pentru îmbunătățirea performanței aplicației create. Se utilizează, în general, pentru task-uri recursive.

Ca și celelalte instanțe ale interfeței *ExecutorService* din pachetul *java.util.concurrent*,ambele *Executor și Fork/join* creează un set de thread-uri pentru distribuirea task-urilor. Noutatea adusă de acest framework este tehnica *work-stealing.* Astfel, thread-urile care execută taskuri extra aplicație pot ”fura” task-uri de la alte thread-uri încă ocupate, pentru a ajuta la terminarea mai rapidă. Atunci când un thread rămâne fără task-uri în coadă, preia task-uri din coada altui thread. Pentru a evita apariția blocajelor, acesta va lua task-uri de la capătul cozii și va lăsa thread-ului gazdă un semnal prin care îl anunță că i-a preluat task-uri. Nucleul acestui framework este reprezentat de clasa *ForkJoinPool* în care apare implementarea algoritmului work-stealing și care poate executa procese ForkJoinTask. Această metodă este recomandată în cazul unor operații complexe.

O aplicabilitate imediată a tehnicii Fork/join este sortarea seturilor foarte mari de date, de exemplu sortarea paralelă a vectorilor. Odată cu versiunea Java 1.8 s-au introdus metode noi în cadrul clasei *java.util.Arrays* care utilizează algoritmul fork/join. Principala metodă introdusă este *parallelSort* care poate fi apelată cu diferite argumente. Algoritmul de sortare este un algoritm de tip merge-sort paralel care împarte vectorul în subvectori care sunt la rândul lor sortați și apoi se compune rezultatul. Atunci când dimensiunea unui subvector atinge un anumit prag, se folosește metoda *Arrays.sort* pentru acel set de date.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

public class ParallelSort {

public static void main(String[] args) {
 List<Integer> list = new ArrayList();
 Random r = new Random();
     for(int x=0; x < 30000000; x++)
     { list.add(r.nextInt(10000));}

     Integer[] array1 = new Integer[list.size()];
     Integer[] array2= new Integer[list.size()];
     array1 = list.toArray(array1);
     array2= list.toArray(array2);

     long start= System.currentTimeMillis();
     Arrays.sort(array1);
     long end = System.currentTimeMillis();
     System.out.println("Sort Time: + " + (end - start));
     start = System.currentTimeMillis();
     Arrays.parallelSort(array2);
     end = System.currentTimeMillis();
     System.out.println("Parallel Sort Time: "+ (end - start))}}
```

h)Streamuri

Prin Stream se înțelege o secvență continuă de elemente care suportă operații agregate secvențiale și paralele, din acest motiv, odată cu Java 8 a fost necesară introducerea unei noțiuni noi - expresiile lambda. O expresie lambda constă:

- dintr-o listă de parametri formali, separaţi prin virgulă şi cuprinşi eventual între paranteze rotunde,
- săgeata direcţională ->
- un corp ce constă dintr-o expresie sau un bloc de instrucţiuni.

Un exemplu de utilizare al expresiilor lambda:

```
System.out.println("=== Sorted Asc ===");
Collections.sort(personList, (Person p1, Person p2)->
p1.getSurname.compareTo(p2.getSurname()));
for(Person p: personList)
   {p.printName();}

System.out.println("=== Sorted Desc ===");
Collections.sort(personList, (Person p1, Person p2)->
p2.getSurname.compareTo(p1.getSurname()));
for(Person p: personList)
   {p.printName();}
```

Atunci când stream-urile sunt executate în paralel, Java partiționează execuția pe mai multe substreamuri. Operațiile agregate iterează prin aceste substream-uri și le procesează în paralel, apoi combină rezultatele. Pentru crearea unui ParallelStream se apelează operația *Collection.parallelStream.* Spre exemplu, în următoarea secțiune se calculează, în paralel, media de vârstă a tuturor bărbaților:

```
Double average= median
.parallelStream()
.filter(p->p.getGender = Person.SEX.MALE)
.mapToInt(Person::getAge)
.average()
.getAsDouble();
```

Tot parallelStream se poate utiliza și la sortarea vectorilor. Pentru aceasta se apelează la metoda *forEachOrdered* ,ca în următorul exemplu :

```
System.out.println("With forEachOrdered:");
    listOfIntegers.parallelStream().forEachOrdered->System.out.println(e + " ");
    System.out.println("");
```

2.2.5. Elemente de analiză de performanță

Programarea paralelă / concurentă este extrem de utilă la execuția cât mai rapidă și eficientă a unor task-uri complexe. Dacă se face o comparație între rularea serială a unei soluții și rularea folosind threaduri, odată cu creșterea complexității se observă eficiența acestora.

Uneori pentru a face alegerea corectă între cele două variante e nevoie de cunoașterea mai detaliată a unor aspecte legate de performanță, date referitoare la *timp de excuție, memorie utilizată*, nuclee etc. Pentru o analiză corectă și detaliată se pot folosi diferite tool-uri existente care oferă toate informațiile necesare. Pentru Java s-au dezvoltat numeroase astfel de tool-uri, care pot fi fie integrate în mediul de lucru (ex NetBeans Profiler, Eclipse Memory Analizer), fie utilizate individual. O listă de astfel de tool-uri și detalii despre cum pot fi descărcate și utilizate se găsește la adresa https://blog.idrsolutions.com/2014/06/java-performance-tuning-tools/.

Pentru afișarea detaliilor despre memoria folosită și despre numărul de thread-uri folosite vom folosi un tool asemănător cu cel de analiză a concurenței din Visual Studio numit *Netbeans Profile*. Acesta este inclus Netbeans însă necesită anumite configurări, astfel :din meniul *Profile* se alege *Advanced Commands->Run Profiler Calibration* și se alege JDK-ul folosit pentru calibrare.

Pentru a rula analizatorul pentru un anumit proiect se setează pentru proiectul respectiv argumentele care ar trebui sa le primească metoda main() prin vectorul de stringuri ‘args’. Pentru a configura Profiler-ul pentru un anumit proiect se selectează proiectul dorit, apoi din meniul - Profile se alege *Profile Project* și se observă statisticile de analiză pentru proiectul dorit.

2.3. Exemple

2.3.1.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului Java Concurency pentru următoarele mecanisme specifice : threadpool, respectiv semafor, exemple pe care le puteți descărca de la adresa *https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/02_Java_concurent.*

2.4. Întrebări teoretice

2.4.1. Explicați și ilustrați aplicabilitatea a două structuri de date thread-safe identificate în Java Concurrency.

2.4.2.Explicați cele două mecanisme de gestionare a threadurilor ThreadFactory și Threadpoolexecutor.

2.4.3. Explicați și exemplificați conceptul de decuplare a lansării taskurilor de politica lor de execuție.

2.4.4. Ce este un Reentrant lock și ce avantaje oferă ?

2.5. Probleme propuse

2.5.1. Să se implementeze mecanismul de sincronizare Producător-Consumator și să se testeze performanța soluției variind tipul de coadă ales și dimensiunea sa, având drept punct de pornire exemplul prezentat în lucrare.

2.5.2. Să se implementeze un server concurent ce deservește un număr nelimitat de clienți folosind FW Executor cu parametrizare adecvată. Se va utiliza analizorul de performanță Netbeans Profiler și se va urmări modificarea timpului de execuție ca urmare a modificării parametrilor din frameworkul Executor.

2.5.3. Să se studieze,testeze și evalueze folosind analizorul de performanță Netbeans Profiler implementările paralele optimizate ale algoritmilor de sortare QuickSort și MergeSort, identificând cele mai adecvate mecanisme și structuri de date din frameworkul JavaConcurency cu ajutorul cărora au fost implementați (sursa algoritmilor :http://heim.ifi.uio.no/~arnem/sorting/). Analiza se va efectua luând drept criterii timpul de execuție și scalabilitatea.

2.6. Miniproiect

Să se implementeze folosind facilitățile moderne de concurență din Java **modelul boss-worker** descris în continuare.

Modelul boss-worker este cel mai potrivit pentru aplicatiile de tipul producător - consumator.
In acest model, un thread este desemnat ca boss, toate celelalte fiind desemnate ca workers. Threadul boss obține sau produce sarcini și le plasează într-o coadă. Threadurile worker preiau sarcinile din coadă și le procesează. Exemple de aplicații în care acest model este recomandat: aplicații care primesc input din surse externe(ex. servere, interfețe cu utilizatorul), algoritmi de căutare, algoritmi de tipul divide et impera, procesarea buclelor în paralel. Deși în general există un singur thread boss, nimic nu impiedică o aplicație sa creeze mai multe threaduri boss. De asemenea, oricare dintre threadurile worker poate acționa ca un thread boss, plasând sarcini în coada de sarcini ,de exemplu ca parte din munca de procesare pe care trebuie să o execute workerul respectiv, el poate să creeze sarcini suplimentare.
Pentru implementarea modelului boss-worker se utilizeaza în general un **"thread-pool".** Astfel thread-ul (thread-urile) boss produce sarcini într-o coadă pe care apoi thread-urile worker le procesează. Un threadpool adresează două probleme:

1. îmbunatățirea performanțelor în cazul în care un număr mare de taskuri sunt executate prin reutilizarea threadurilor worker;
2. limitarea și managementul resurselor utilizate pentru execuția unei colecții de sarcini (taskuri).

Modelul boss-worker în abordarea clasică, anterior JDK 1.5 se implementează utilizînd: două variabile condiționale ,un mutex ,o coadă ,un număr de contoare .Mutexul protejează coada, variabilele condiționale și contoarele. O variabilă conditională este folosită de threadurile worker în așteptare atunci când coada este goală. Cealaltă variabilă condițională este utilizată de threadul boss atunci când coada este plină. Un minimum de 3 contoare este necesar pentru a cunoaște în permanență: C1-numărul de sarcini existente în coadă, C2-numărul de threaduri worker in asteptare C3-numărul de threaduri boss în asteptare

Pentru a plasa noi sarcini în coadă, threadul boss trebuie întâi să preia controlul asupra mutexului. In cazul în care coada nu este plină, threadul boss adaugă sarcini în coadă și semnaleaza acest lucru eventualelor threaduri worker care așteptau. In cazul în care coada este plină, threadul boss așteaptă la o variabilă condițională până când se face loc în coadă pentru noi sarcini. Odată ce sarcinile au fost depuse în coadă, mutexul este eliberat și threadul boss se reîntoarce la munca sa (adica la crearea sau obținerea de noi sarcini). Pentru a prelua sarcini din coadă, un thread worker trebuie întâi să preia controlul asupra mutexului. Dacă există sarcini disponibile, atunci una dintre ele este preluată și acest lucru este semnalat eventualelor threaduri boss care așteptau. Dacă nu exista sarcini in coada (coada este vida), threadul worker asteapta la o variabilă condițională până când apar sarcini în coadă. Odată ce sarcina a fost preluată din coada, mutexul este eliberat și threadul worker începe să proceseze sarcina sa. Sincronizarea este necesară numai atunci când au loc operații cu coada, adică atunci când se pun sau se iau sarcini din coadă. Acesta este singurul loc (adică singura structură de date partajată) unde datele pot fi accesate de mai multe threaduri. In timp ce threadul boss creează sarcini, nici unul dintre threadurile worker nu poate accesa aceasta structură sau datele asociate ei. Odată ce un thread worker și-a preluat sarcina din coadă, nici unul din celelalte threaduri (nici worker, nici boss) nu mai poate accesa acea sarcină sau datele asociate acesteia.

Dimensiunea cozii:poate fi statică sau dinamică. Atunci când coada are o lungime statică sau maximă predefinita, sunt necesare două variabile condiționale (una pentru threadurile worker, cealaltă pentru threadurile boss). Există situații în care lungimea cozii poate fi infinită. In aceste cazuri, este suficientă o singură variabilă

condițională și anume cea pentru threadurile worker în așteptare, deoarece threadurile boss nu se pot bloca, din moment ce coada nu se va umple niciodată. Totuși, cozile statice au avantajul că threadul boss nu supraîncarcă threadurile worker cu mai multă muncă decât acestea pot procesa. O oarecare balansare este asigurată folosind cozi cu lungime statică. Dezavantajul acestei situatii este cel menționat și anume ca threadul boss se poate bloca, fapt ce poate avea diferite impacte asupra utilizatorului sau timpului de răspuns al aplicației.

In modelul boss-worker, fiecare dintre threadurile worker se află într-o buclă continuă în care obțin și proceseaza sarcini. La un moment dat, toate sarcinile se vor fi terminat și threadurile worker ar trebui să se termine și ele. Acestea nu se pot termina pur și simplu când coada este goală, ci trebuie anunțate în mod explicit că trebuie să se termine.

Exista două tehnici de realizare a acestui lucru:

1. sa se creeze o sarcină specială, de "exit", pentru fiecare dintre threadurile worker, și fiecare dintre aceste sarcini să fie plasată în coadă. Când un thread worker primește o sarcină de ieșire (exit task), el se va termina.

   Observație: acest lucru este desigur valabil dacă ordinea de prelucrare a sarcinilor este FIFO, deoarece sarcinile de ieșire trebuie să fie ultimele sarcini depuse de threadul boss în coadă.

2. să se asocieze cozii un flag "exit". Atunci când threadul boss dorește ca threadurile worker să se termine, el seteaza acest flag la valoarea EXIT.

Când threadurile worker încearcă să extragă o sarcină din coadă,în cazul în care coada este vidă și flagul de exit este setat, atunci ele ies din bucla infinită și se termină.

Observație: coada trebuie sa fie vidă. Daca threadurile worker s-ar termina oricând flagul este setat, atunci ele s-ar putea termina înainte ca toate sarcinile să fi fost prelucrate. Odată ce toate threadurile worker au fost instruite să se termine, threadul boss așteaptă terminarea tuturor threadurilor worker (join).

Pentru aplicațiile care folosesc acest model pentru implementarea concurenței, echilibrarea încărcării nu reprezintă o problemă. Aceste aplicații pot folosi modelul boss-worker pentru a împarți sarcinile astfel încât atunci când un thread se blochează, un altul poate continua în locul lui. Acest caz este cel mai des întâlnit atunci cînd modelul boss-worker este folosit pentru servere și interfețe cu utilizatorul, unde cantitatea de sarcini este controlată de evenimente externe.Tthreadul boss acceptă taskuri și le pasează threadurilor worker, iar planificarea balansării nu este o problemă pentru aceste aplicații.

Un alt tip de echilibrare este totusi important în modelul boss-worker: echilibrarea cozii. O aplicație eficienta va implementa cozi care nu produc nici blocarea threadurilor boss, nici blocarea threadurilor worker (mai explicit, coada nu va fi niciodată nici goală, nici plină). Pentru obținerea acestei situații ideale, se poate experimenta în sensul ajustării *numărului de sarcini, a numărului de threaduri worker, a dimensiunii cozii sau a dimensiunii sarcinilor* (acolo unde acest lucru este posibil; există cazuri în care dimensiunea sarcinii este fixată); de exemplu, dacă threadul boss se blochează datorită umplerii cozii, se poate mări numărul de threaduri worker sau se poate mări dimensiunea cozii; dacă threadurile worker se blochează datorită faptului că în coadă nu există sarcini, se poate mări numărul de threaduri boss sau micșora numărul de threaduri worker.

In cazul aplicațiilor care sunt controlate de evenimente externe (de exemplu, input de la utilizator sau de pe rețea), o balansare a cozii nu va fi niciodată posibilă. Aceste aplicații ar trebui să aibă in vedere crearea unui număr dinamic de threaduri boss și threaduri worker. Când volumul de muncă existent este redus, se vor folosi un singur thread boss și câteva threaduri worker; când volumul de muncă sporește, se mărește și numărul de threaduri worker și eventual și cel de threaduri boss.

Pachetul **java.util.concurrent din JDK 1.7** oferă (printre multe alte facilități) metode de implementare a modelului discutat.

Astfel, clasa java.util.concurrent.ThreadPoolExecutor poate fi utilizată pentru crearea unui thread-pool. Următorii parametrii pot fi specificați la instanțierea unui astfel de obiect:

- **corePoolSize** - numărul minim de fire de executie "worker" ce sunt menținute în pool.
- **maximumPoolSize** - numărul maxim de fire de execuție ce pot fi create.
- **keepAliveTime** - când numărul de threaduri din pool este mai mare decat corePoolSize, acest parametru specifică timpul maxim în care firele de execuție în exces vor astepta sarcini înainte de a se termina
- **unit** - unitatea de timp (TimeUnit) pentru parametrul keepAliveTime

- **workQueue** - coada utilizată pentru salvarea sarcinilor ce urmează să fie executate. O sarcină este un obiect ce implementează interfata Runnable și poate fi introdusă în coadă utilizand metoda execute.

Coada trebuie să implementeze interfața BockingQueue<E> Implementări existente pentru o coadă pe care puteți să le utilizati:

ArrayBlockingQueue -coadă de dimensiune fixă ce utilizeaza un vector pentru salvarea sarcinilor
LinkedBlockingQueue -coadă de dimensiune nelimitată implementata printr-o listă simplu inlanțuită (FIFO)
PriorityBlockingQueue -coadă de dimensiune nelimitată în care se poate defini ordinea de inserare a elementelor
SynchronousQueue -coadă fara dimensiune, adaugarea respectiv extragerea elementelor din coada presupunand blocare.

- **threadFactory** - mecanismul de instanțiere a firelor de executie worker
- **handler** - handler utilizat pentru a primi notificari in cazul in care un task nu se poate executa.

Astfel , o sarcină va fi rejectată dacă e îndeplinită una din condițiile :

- mecanismul de introducere de sarcini a fost oprit (shutdown)
- pool-ul de threaduri este saturat (s-a ajuns la limita de maximumPoolSize threaduri și s-a atins limita maximă a cozii de sarcini (dacă se utilizează o coadă de dimensiune maximă fixată))

Schelet de implementare . Instanțierea unui thread-pool ce utilizează o coadă de sarcini de dimensiune infinită, 4 threaduri de bază și stabilește o limită maximă de 20 threaduri ce pot rula.

```
ExecutorService myPool =     new ThreadPoolExecutor(4, 20,
1L,TimeUnit.MILLISECONDS newLinkedBlockingQueue<Runnable>());,
```

Utilizare:

```
while (exista_sarcini){
     [producere sarcini]
     Runnable sarcina = new PoolTask(sarcina);
myPool.execute(sarcina) }
..
   [finish]
   myPool.shutdown();
   try {
   allJoined = myPool.awaitTermination(60L,TimeUnit.SECONDS);
if(!allJoined)
                myPool.shutdownNow()
        } catch (Exception e) {//nothing to do}

Sarcina (task):
class PoolTask implements Runnable{
    Sarcina sarcina;
   PoolTask(Sarcina s){
this.sarcina=s; }
   void run(){
   //procesare sarcina }}
```

2.7. Referințe bibliografice

1. API :https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
2. Tutorial Oracle : http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
3. Tutorial http://tutorials.jenkov.com/java-util-concurrent/index.html
4. Exemple aplicații cu implementare concurentă : http://www.cdac.in/index.aspx?id=ev_hpc_hypack_java_concurrent_programs
5. Java Concurrency essentials, Martin Mois, JDC books,2015 disponibilă la http://enos.itcollege.ee/~jpoial/allalaadimised/reading/Java-Concurrency-Essentials.pdf