

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

*** * ***

Université de Carthage

*** * ***

**Institut National des Sciences
Appliquées et de Technologie**



Compte Rendu N°3

Programmation parallèle avec JAVA

Réalisé par :

Yasmine MHIRI

Amal MTIBAA

Mohamed Wassim RIAHI

GL4 – Groupe1

Année Universitaire : 2018/2019

Exercice 1 :

La configuration logicielle et matérielle de la machine utilisée pour cet exercice:

- Processeur : Inter(R) Core(™) i7
- RAM : 8G
- SE : Windows 10 64bits
- IDE : Eclipse java Neon

On souhaite écrire un programme qui affiche les valeurs des racines carrées de 0 à 10 000. On souhaite paralléliser ce programme en exécutant en même temps plusieurs opérations, chaque thread devra par exemple effectuer le calcul d'une centaine de racine carré.

1)

En utilisant la classe Executors, on crée un ExecutorService qui s'en charge de créer 5 threads chaque un fait le calcul d'une partie (2000 racines carrés)

```
public static void main(String[] args) throws Exception{
    ExecutorService exec= Executors.newCachedThreadPool();

    for(int i=0;i<5;i++){ //5 Threads
        int j=i;
        exec.execute(()->{
            int start=j*2000;
            int end=start+2000;
            for(int k=j+1;k<=end;k++){
                System.out.println(Math.sqrt(k));
            }
        });
    }
}
```

Partie de l'output : Chaque Thread affiche dans l'ordre croissants des racines carrés.

```
70.9859140956852
70.9929573971954
71.0
71.00704190430693
71.01408311032397
71.02112361825881
71.02816342831905
71.03520254071216
71.04224095564554
71.04927867332644
71.05631569396206
71.06335201775947
71.07038764492565
71.07742257566744
71.08445681019164
```

2) Ici, on utilise des Runnable pour effectuer l'affichage, on crée une classe CalculRunnable implémentant la classe Runnable comme suit :

```

public class CalculRunnable implements Runnable {

    private int i;

    public CalculRunnable(int i){
        this.i=i;
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        System.out.println(Math.sqrt(i));
    }

}

```

On crée dans le main un tableau de 10000 CalculRunnables

```

ExecutorService exec1= Executors.newCachedThreadPool();
List<Runnable> runnables=new ArrayList<Runnable>();
for(int i=0;i<=10000;i++){
    runnables.add(new CalculRunnable(i));
}

for(int i=0;i<10000;i++){
    exec1.execute(runnables.get(i));
}

```

Partie de l'Output : l'affichage ne s'effectue pas dans l'ordre des racines carrées croissantes.

```

66.35510530471637
84.38601779915912
75.25290692059676
94.57801012920498
94.58858282054976

```

3) On vise maintenant d'utiliser l'interface Callable et le mécanisme Future pour effectuer une évaluation asynchrone. On crée alors la classe CalculCallable comme suit:

```

import java.util.concurrent.Callable;

public class CalculCallable implements Callable<String> {

    private int i;

    public CalculCallable(int i){
        this.i=i;
    }

    @Override
    public String call() throws Exception {
        // TODO Auto-generated method stub
        return String.valueOf(Math.sqrt(i));
    }

}

```

On crée un tableau de type CalculCallable<string>, la méthode submit(...) fournit une référence aux valeurs future du résultat.

La méthode get() est bloquante

```
ExecutorService exec2= Executors.newCachedThreadPool();
List<Callable<String>> callables=new ArrayList<Callable<String>>();

for(int i=0;i<=10000;i++){
    callables.add(new CalculCallable(i));
}

ArrayList<Future<String>> futures=new ArrayList<Future<String>>();

for(int i=0;i<=10000;i++){
    futures.add(exec2.submit(callables.get(i)));
}

System.out.println("something");

for(int i=0;i<=10000;i++){
    System.out.println(futures.get(i).get()) ;
}
```

Partie de l'Output : on choisit ici d'effectuer l'affichage dans l'ordre des racines carrées croissantes

```
99.9799979995999
99.98499887483122
99.98999949994999
99.99499987499375
100.0
```

Exercice 2 :

La configuration logicielle et matérielle de la machine utilisée pour cet exercice:

- Processeur : Inter(R) Core(™) i7
- RAM : 6G
- SE : Windows 10 64bits
- IDE : Eclipse java photon

- 1) La classe "TaskFJ" étend la classe RecursiveAction et utilise le framework fork/join pour incrémenter les valeurs des cases du tableau array qui est divisé en sous tableaux de tailles inférieures à 1000 pour effectuer le traitement.

```

public class TaskFJ extends RecursiveAction {
    private final int array[];
    private final int start, end;

    public TaskFJ(int array[], int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    protected void compute() {
        if (end - start > 1000) {
            int mid = (start + end) / 2;
            TaskFJ task1 = new TaskFJ(array, start, mid);
            TaskFJ task2 = new TaskFJ(array, mid, end);
            task1.fork();
            task2.fork();
            task1.join();
            task2.join();
        } else {
            for (int i = start; i < end; i++) {
                array[i]++;
                try {
                    TimeUnit.MILLISECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

Dans la classe main, on a déclaré un tableau de taille 100000 et on va calculer le temps d'exécution en utilisant la classe Date:

```

public static void main(String[] args) {
    int array[] = new int[100000];
    TaskFJ taskFJ = new TaskFJ(array, 1, 100000);
    ForkJoinPool pool = new ForkJoinPool();
    Date start = new Date();
    pool.execute(taskFJ);
    pool.shutdown();

    try {
        pool.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    Date end = new Date();
    System.out.printf("Core: Fork/Join: %d\n", (end.getTime() - start.getTime()));
}

```

Après exécution:

```
<terminated> TaskFJ [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (20 mars 2019 à 17:06:32)
```

```
Core: Fork/Join: 37401
```

```
[
```

2) Utilisons maintenant l'Executor pour modifier les valeurs de l'array. L'interface ExecutorService décrit les fonctionnalités d'un service d'exécution de tâches. Elle hérite de l'interface Executor. Et la méthode awaitTermination pour attendre l'achèvement des tâches après une demande d'arrêt ou la fin d'un délai ou l'interruption du thread courant selon ce qui se produira en premier.

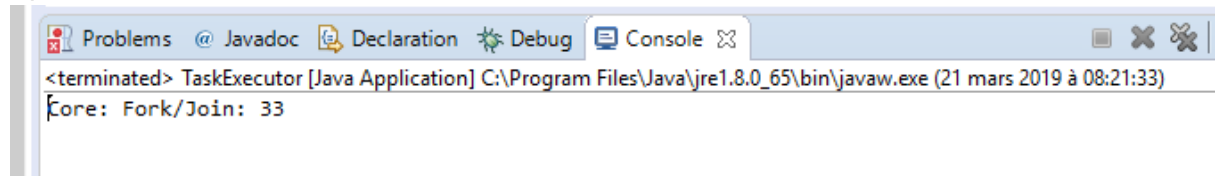
Dans la méthode main(), créons un service d'exécution, doté de 4 threads:

```
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;


public class TaskExecutor{

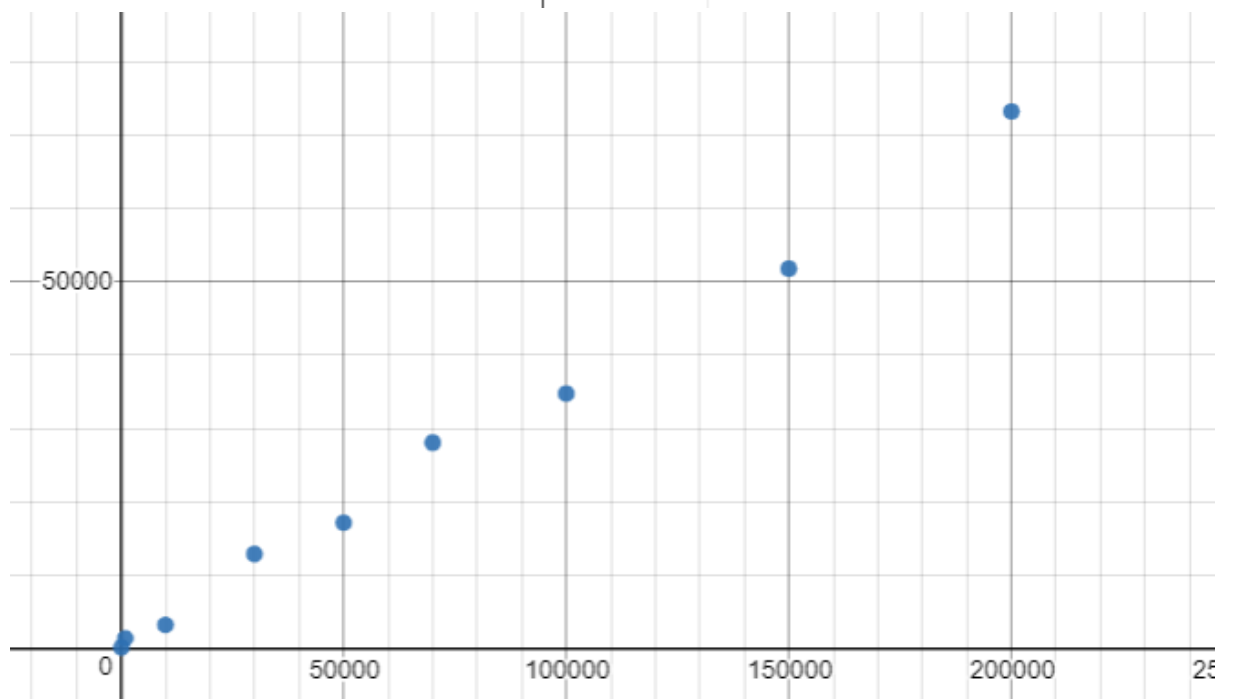
    public static void main(String[] args) {
        int array[] = new int[100000];
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for(int i=0;i<100000;i++)
        {
            int j=i;
            executor.execute(() -> {
                array[j]++;
            });
        }
        executor.shutdown();
        Date start = new Date();
        try {
            executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Date end = new Date();
        System.out.printf("Core: Fork/Join: %d\n", (end.getTime() - start.getTime()));
    }
}
```

Après exécution :




3) - Pour le fork join:

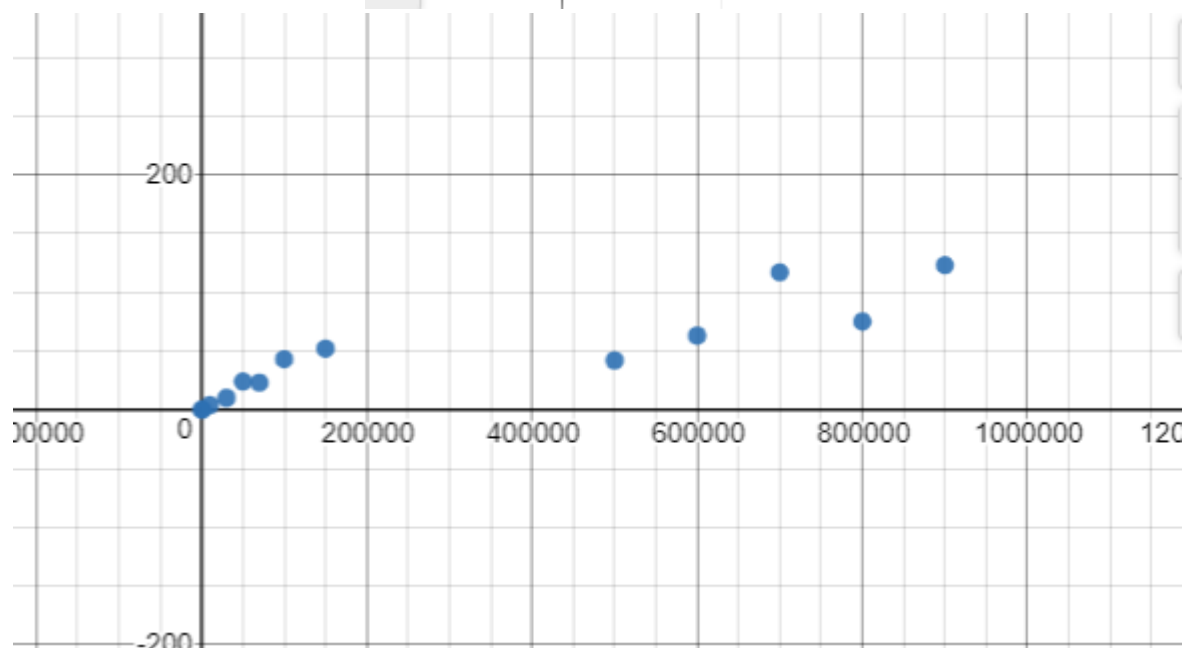
x_1	 y_1
100	112
1000	1374
10000	3206
30000	12872
50000	17142
70000	28071
100000	34761
150000	51812
200000	73270
-----	-----



-> C'est une fonction linéaire.

- Pour l'Executor :

x_1	 y_1
100	0
1000	0
10000	4
30000	10
50000	24
70000	23
100000	43
150000	52
500000	42
600000	63
700000	117
800000	75
900000	123



-> C'est une fonction logarithmique.

4) On remarque que le temps d'exécution en utilisant la classe Executor est meilleur et plus optimal que l'utilisation du framework fork/join.

L'utilisation du framework fork/join facilite la répartition dynamique du travail, en décomposant le traitement en sous-tâches indépendantes. Mais il est lent puisque la décomposition et le fusionnement prennent beaucoup de temps.

L'exécution parallèle des tâches de la Classe Executors a les avantages suivantes :

- Un nombre optimal de thread peut être créé et recyclé.
- Moins de requête dans la file d'attente.
- Minimise la création et la destruction de thread.