ÉCOLE CENTRALE LYON

UE APPRO Applications concurrentes, mobiles et réparties en Java

Rapport BE Prog Concurrente

Auteurs :
M. Valentin VERDON

Encadrants :
M. Alexandre SAÏDI

Version du
7 novembre 2022

Table des matières

1 Présentation de la programmation concurrente

Le principe de la programmation concurrente est de segmenter la tâche à réaliser en plusieurs morceaux qui seront réalisé indépendamment sur différents Threads qui seront exécutés en même temps.
L'objectif de ce rapport est d'utiliser cette logique de programmation dans différents cas de figures : le calcul de π, la réalisation de trie et dans la réalisation du jeu *Game Of Life*.

Pour ne pas alourdir le rapport, nous ne développerons en détails que les deux premiers programmes. Si le lecteur le souhaite, il peut chercher à comprendre en détail le jeu à l'aide du code java (ce dernier est commenté permettant de comprendre les différents éléments du code).

2 Calcul de π

2.1 Méthode de Calcul

Pour calculer π, nous allons utiliser l'intégrale de la loi normale centrée réduit qui a pour expression :

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \tag{1}$$

Dont l'intégrale vaut :

$$\int_0^{\infty} f(x)dx = \frac{1}{2} \tag{2}$$

Nous allons donc segmenter l'intervalle de calcul de l'intégrale et nous associerons ce calcul pour chaque intervalle à un Thread.

2.2 Mise en pratique

Nous créons une classe *Pi* qui contiendra tout le programme.

Dans celle-ci, nous mettons les différents paramètres du calcul de notre intégrale tel que le nombre de Threads, le nombre d'itération et l'intervalle de calcul (nous ne pouvons pas intégrer jusqu'à l'infini).

```
static int Nb_threads = 4;
static int Nb_iterations = 1000000; //itérations par threads
static double resultat_local[] = new double[Nb_threads];
static double max = 10;
static double pas=max/(Nb_iterations*Nb_threads);
```

FIGURE 1 – Variable d'intégration

Nous implémentons la fonction à intégrer dans une fonction à part :

```
public static double f(double x) {
    return Math.exp(-x*x/2);
}
```

FIGURE 2 – Fonction à intégrer

Puis nous créons une classe calcul qui instanciera chaque Thread. C'est une classe qui hérite de *Runnable* dont le constructeur aura un paramètre : le numéro du Thread pour permettre de l'identifier (cela nous permettra de lui associer un

morceau de l'intervalle et de stocker son résultat sans confusion avec les autres Threads).

Nous implémentons la fonction *run* qui contient le code qui sera exécuté par les Threads. Ici, *run* calcul l'intégrale sur le segment associer au Threads de la fonction f avec comme paramètres d'intégrations ceux définies précédemment.

```
private static class Calcul implements Runnable {
    int myNum;

    // pour passer le numéro du Threads en paramètre
    public Calcul(int num) {
        myNum = num;
    }

    public void run(){
        double stockage = 0;
        for (int i=this.myNum*Nb_iterations; i<(this.myNum+1)*Nb_iterations; i++) {// calcul du morceau d'intégrale
            stockage+=pas*(f(pas*i)+f(pas*(i+1)))/2;
        }
        resultat_local[myNum]=stockage; //stockage du morceau d'intégrale
    }
```

FIGURE 3 – Classe Calcul

Enfin, nous implémentons la fonction *main* qui s'exécutera lors de la compilation du programme. Celle-ci va créer un tableau de Threads que l'on remplira en créant nos Threads. Puis nous lancerons le calcul de nos Threads qui iront déposer leurs résultats du calcul de leur morceau d'intégrale dans un tableau. Nous finirons par parcourir ce tableau pour en additionner les différentes composantes ce qui nous donnera π (à un facteur d'intégration près que nous implémentons). Nous calculerons en parallèle la durée du programme pour étudier l'évolution du temps de calcul en fonction du nombre de Threads.

```
public static void main(String[] arges) {
    double valeur_pi=0;

    // pour calculer le temps du programme
    long startTime = System.currentTimeMillis();
    long endTime = System.currentTimeMillis();

    startTime = System.currentTimeMillis();

    Thread Ids[] = new Thread[Nb_threads]; //tableau des threads
    for (int i=0; i< Nb_threads; i++){// création et paramétrage des threads
        Ids[i]= new Thread(new Calcul(i));
        Ids[i].start();
    }

    //Lancement du calcul
    System.out.println("C'est parti");

    for (int i=0; i < Nb_threads; i++) {
        try {
            Ids[i].join();
            valeur_pi+= resultat_local[i];
            System.out.println(resultat_local[i]);
        }
        catch (InterruptedException ie)
        {// Ne devrait pas arriver car on n'appelle pas interrupt()
        }
    }
    //maj valeur pi
    valeur_pi =valeur_pi*valeur_pi*2;


    //affichage du résultat
    endTime = System.currentTimeMillis();
    System.out.printf("La valeur des pi = %f %n", valeur_pi);
    System.out.println(endTime-startTime);
```

FIGURE 4 – Fonction main

2.3 Interprétation des résultats

Pour des essais dans l'intervalle [0,10] avec un nombre d'itérations total constant de 60,000,000 nous obtenons une approximation de π avec un erreur relative de 10^{-12}%.

Pour ce qui est du temps d'exécution, nous avons pu obtenir le graphe suivant :

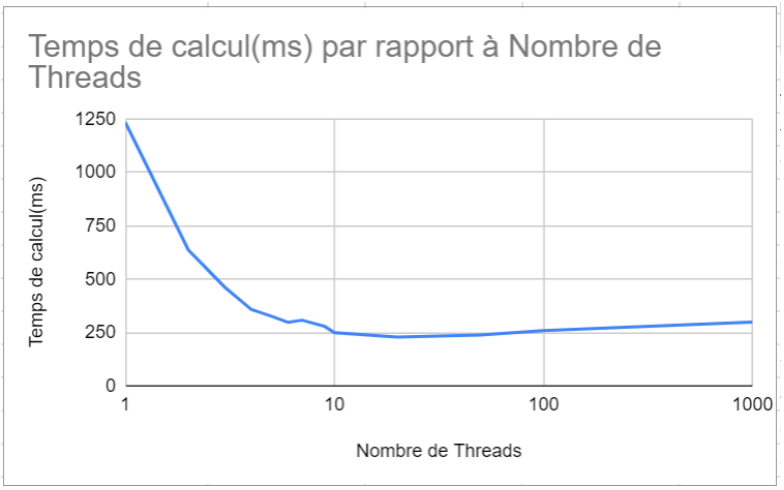

FIGURE 5 – temps de calcul

Nous avons donc un temps d'exécution décroissant avec le nombre de Thread ce qui correspond aux résultats attendus. Cependant, il faut prendre en compte une incertitude de +/- 15 ms car d'autre tâches se réalise en même temps sur l'ordinateur ce qui implique des conditions qui ne sont pas identiques entre les différents essais.

Nous pouvons remarquer qu'à partir de 10 Threads, on obtient un temps d'exécution stable qui tend à s'allonger. En effet, créer des Threads, leurs associer un espace et les faire s'exécuter prends du temps. De plus, l'ordinateur possède un nombre de Threads indépendant limité, ce qui implique qu'à partir d'un certain nombre, plusieurs Threads correspondent pour l'ordinateur à un unique Thread qu'il utilisera plusieurs fois.

3 QuickSort

3.1 Principe

Le tri rapide est un trie assez répandu car il possède une complexité en moyenne de O(n.log(n)). Il a pour principe de prendre en entrée une liste (ici ce sera des tableaux d'une seule dimension), de séparer cette liste en deux sous-liste en fonction d'un élément de la liste, appelé pivot, puis de tri de la même façon ces deux sous listes jusqu'à obtenir des listes d'un voir aucuns éléments. Puis, nous concaténons ces listes en triant par rapport aux premiers éléments. Pour obtenir finalement la liste triée.

Comme en Java nous ne pouvons pas modifier la taille de nos tableaux et que le coup en temps de recopier le tableau dans un tableau de taille supérieur est trop élevé, nous travaillerons directement sur le tableau en entrée.

Ainsi, nous allons associer à nos sous listes des indices de début et de fin qui permettrons de les délimiter dans la liste principale. Pour partitionner la liste, nous allons échanger de place le pivot pour obtenir le résultat souhaité. Nous recommençons l'opération jusqu'à ce que l'indice de début dépasse l'indice de fin ce qui signifie que le morceau de liste est trié.

Voici le code associé à ces opérations :

```
// ----------- Trie Rapide -----------------

//initialisation du tri avec en entrée uniquement le tableau
public static void triRapide(double tableau[]){
int longueur=tableau.length;
triRapide(tableau,0,longueur-1);
}

private static void triRapide(double tableau[],int deb,int fin){
    if(deb<fin){
        int positionPivot=partition(tableau,deb,fin);
        triRapide(tableau,deb,positionPivot-1);
        triRapide(tableau,positionPivot+1,fin);
        }
    }

//On déplace les éléments du tableau pour avoir un tableau trié en fonction du pivot
private static int partition(double tableau[],int deb,int fin){
    int compt=deb;
    double pivot=tableau[deb];

    for(int i=deb+1;i<=fin;i++){
        if (tableau[i]<pivot){
            compt++;//définie la place du pivot pour la fin de la boucle
            //On met l'élément tableau[i] à sa place (peut importe la place du pivot à ce moment)
            echanger(tableau,compt,i);
            }
        }
    //On met le pivot au bon endroit
    echanger(tableau,deb,compt);
    return(compt);
    }

// Fonction pour échanger deux éléments du tableau
public static void echanger(double[] tableau, int compt, int i ) {
    double aux = tableau[compt];
    tableau[compt]=tableau[i];
    tableau[i]=aux;
}
```

FIGURE 6 – tri Rapide

3.2 Utilisation de la programmation concurrente

Pour utiliser la programmation concurrente et optimiser notre trie, nous allons segmenter notre tableau en sous tableaux auxquelles nous associerons un Thread qui aura pour mission de trier ce tableau. Finalement, il faudra parcourir tous ces sous tableaux pour obtenir le tableau final.

Voici la classe *Calcul* qui s'occupe d'implémenter les Threads :

```
// --------------- Parralélisation --------------------
private static class Calcul implements Runnable{
    int deb;
    int fin;
    double[] tab;

    //Contructeur pour passer les indices du Threads
    public Calcul(double[] tab,int deb, int fin) {
        this.deb=deb;
        this.fin=fin;
        this.tab=tab;
    }

    public void run() {
        triRapide(this.tab,this.deb,this.fin);
    }
}
```

FIGURE 7 – Implémentation Threads

Voici la fonction qui permet de concaténer les résultats des Threads :

```
//Regrouper les différents sous tableaux de la parralélisation
public static double [] concatenation(double[] tab, int[] indice) {
    double [] result = new double[tab.length];
    for (int i=0;i<tab.length;i++) {

        //On choisis le premier indice encore valable
        int aux_indice = 0;
        while (indice[aux_indice]>indice[aux_indice+1]) {
            aux_indice+=2;

        }

        //On place le plus petit élément au début du tableau de travail
        double aux_valeur = tab[indice[aux_indice]];
        for (int j=0;j<indice.length;j+=2) {
            if ((aux_valeur>tab[indice[j]])&&(indice[j]<=indice[j+1])) {
                aux_indice= j;
                aux_valeur= tab[indice[j]];
            }
        }
        result[i]=aux_valeur;
        indice[aux_indice]+=1;
    }
    return result;
}
```

FIGURE 8 – Concaténation

La classe *Main* se décompose en plusieurs étapes. Tout d'abord, nous créons deux tableaux aléatoirement de même taille. Puis nous trions le premier à l'aide du tri rapide. Le second sera trié à l'aide de la programmation concurrente. Cela comprend la création des Threads avec leurs indices dans le tableau, le lancement du calcul et la concaténation. Nous calculerons le temps d'exécution des deux programmes ainsi que celui de l'étape de concaténation de la version concurrente.

Voici la classe *Main* :

```java
// ----------------- Main ----------------------
public static void main(String[] arges) {

    // ------------- Préparation des programmes -----------------
    //Variables pour calculer le temps du programme
    long startTime;
    long endTime;
    long auxTime;

    // Création de mon tableau
    double[]monTableau = new double[1000000];
    Random randomNumbers = new Random(); // random number generator
    for(int i=0; i < monTableau.length; i++)
        monTableau[i]= randomNumbers.nextInt(100000);

    //Création du 2nd tableau
    double[]monTableau2 = new double[1000000];
    for(int i=0; i < monTableau.length; i++)
        monTableau2[i]= randomNumbers.nextInt(100000);

    // ----------- Programme sans parralélisme -------------------
    startTime = System.currentTimeMillis();
    triRapide(monTableau);
    endTime = System.currentTimeMillis();
    System.out.println(endTime-startTime);

  //---------------- Avec parralélisme: ---------------------------
    startTime = System.currentTimeMillis();

    //Calcul des indices + initialisation des Threads
    int l = monTableau2.length;
    int pas = (int)l/Nb_threads;
    int deb; int fin;

    int[] indice = new int[Nb_threads*2];
```

(a) *Main* Partie 1

```java
Thread Ids[] = new Thread[Nb_threads]; //tableau des threads
for (int i=0; i< Nb_threads; i++){// création et paramétrage des threads
    if (i< Nb_threads-1) {
        deb = i*pas;
        fin = (i+1)*pas-1;
        indice[i*2]=deb;
        indice[i*2+1]=fin;
    }
    else {
        deb = (Nb_threads-1)*pas;
        fin = l-1;
        indice[2*i]=deb;
        indice[2*i+1]=fin;
    }
    Ids[i]= new Thread(new Calcul(monTableau2,deb,fin));
    Ids[i].start();
}

//Lancement du calcul
for (int i=0; i < Nb_threads; i++) {
    try {
        Ids[i].join();
    }
    catch (InterruptedException ie)
    {// Ne devrait pas arriver car on n'appelle pas interrupt()
    }
}
auxTime=System.currentTimeMillis();
monTableau2 =concatenation(monTableau2,indice);
endTime = System.currentTimeMillis();
System.out.println(auxTime-startTime);
System.out.println(endTime-startTime);
```

(b) *Main* Partie 2

FIGURE 9 – Fonction *Main*

3.3 Exploitation des résultats

Notre programme nous donne dans les deux cas un tableau trié en sortie. Nous allons étudier le temps de compilation pour comparer les deux approches à l'aide de ce graphique :

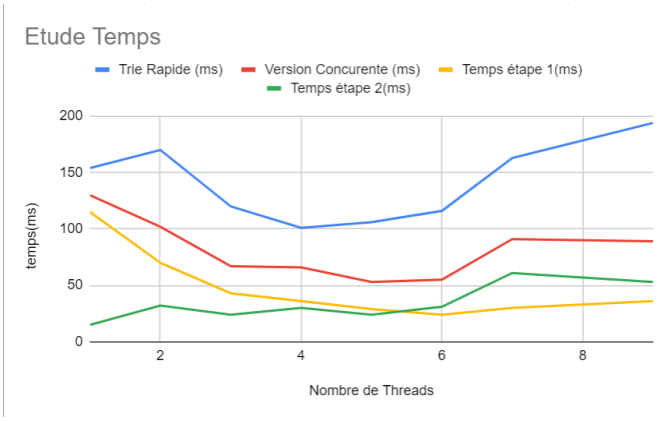

FIGURE 10 – Graphique temps Tri Rapide

Ces tests ont été réalisés avec des tableaux générés aléatoirement de 1,000,000

d'éléments.
On remarque que temps d'exécution du tri rapide simple est assez variable (entre 100 et presque 200ms). Cela s'explique par des configurations plus ou moins déjà "pré-trié" ce qui permet de passer les boucles rapidement. De plus, l'ordinateur effectue d'autre tâches en même temps ce qui impact également le temps d'exécution.

Le tri rapide en programmation concurrente est plus rapide que la version classique (gain de 50% environ). Les meilleurs résultats sont obtenus pour 5 threads. On peut remarquer que coup de l'étape finale est en moyenne égale au tri des sous tableaux. Cette partie du code pourrai être optimisé afin de rendre la version concurrente encore plus performante.

4 Game Of Life

4.1 Règles du jeu

Le *Game Of Life* est un jeu dans lequel les cellules d'un tableau possède deux états : vivant ou mort. À chaque itération, l'état d'une cellule est entièrement déterminé par l'état de ses huit cellules voisines, selon les règles suivantes :

- une cellule morte possédant exactement trois cellules voisines vivantes devient vivante (elle naît)

-une cellule vivante possédant deux ou trois cellules voisines vivantes le reste, sinon elle meurt.

4.2 Réalisation du jeu

Nous avons choisis d'implémenter un jeu qui génère aléatoirement des grilles carrées (dont la taille peut être modifié dans le code) et de voir leur évolution. Nous aurons la possibilité de mettre en pause la simulation, de la relancer et de la réinitialiser. Nous afficherons le temps maximum pour la réalisation d'un étape ainsi que le nombre de threads en action.

Pour implémenter ce jeu, nous avons créé 3 fichiers :
- $Main - GAmeOfLife$ qui contient le programme principale avec le visuel associé
- *CellGrid* qui contient la classe qui crée les grilles avec leurs fonctions associés

- *MajThread* qui s'occupe de la gestion des Threads.

Comme annoncé en début de rapport, nous n'étudierons pas en détails le code associé au jeu mais uniquement la logique l'implémentation.

Dans le programme $Main - GAmeOfLife$, nous mettons tout les élements d'affichages, la taille des cases et le nombre de Threads. Nous créons une grille random. Puis nous associons aux différents boutons leur fonction associé.

Lorsque l'on clique sur le bouton *start*, on crée un Thread qui va lancer le programme lié aux Threads. Celui-ci va créer deux grilles de travailles et séparer la grille principale en sous grilles. Ces sous grilles seront associé à des Threads de travails créé par le premier Thread. Ces Threads de travail s'occupent d'appliquer les règles du jeu et de compléter la grille qui sera afficher lorsque le travail de cette étape sera fini. Ce processus est répété en continue.

4.3 Rendu

Voici la page du jeu :

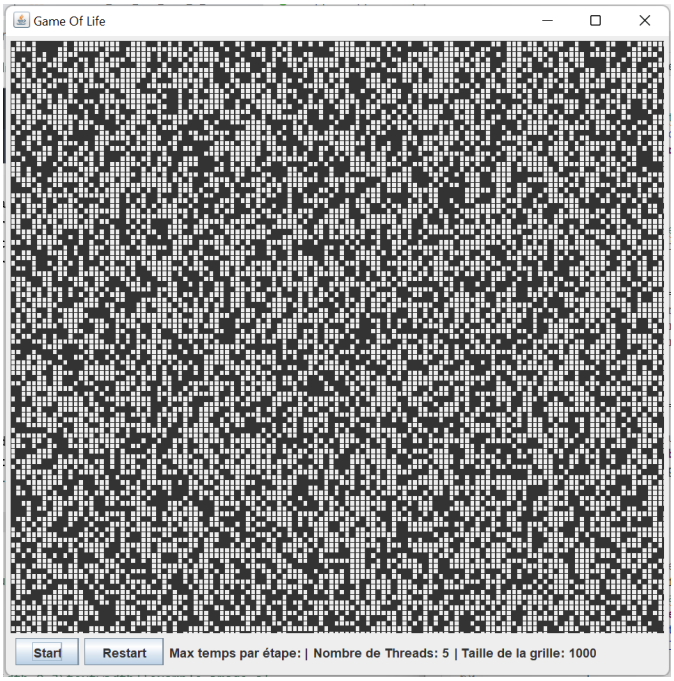

FIGURE 11 – Interface *Game of Life*

Le bouton *Start/Stop* permet de lancer la simulation et de la mettre en pause. Le bouton restart permet de charger une nouvelle grille.

On notera que la grille n'est pas entièrement affichée, la fenêtre est prévue pour une grille de 500 éléments mais pour pouvoir avoir des différences de performances significatives entre les simulations à 1 Threads et à 5 Threads, nous utilisons une grille de 1000 éléments.

Nous aurions pu implémenter de nombreuse fonction supplémentaire comme un système pour se déplacer sur la grille, mettre directement dans la fenêtre le nombre de Thread voire un espace pour entrer une grille personnalisée.

Concernant les performances de notre jeu, nous remarquons que ce sont les premières étapes qui sont les plus gourmandes en ressources car ce sont-elles qui génèrent le plus de changement (les cellules tendent en majorité à mourir sauf pour certaines configurations particulière). C'est pourquoi nous étudions le temps maximal par étape.

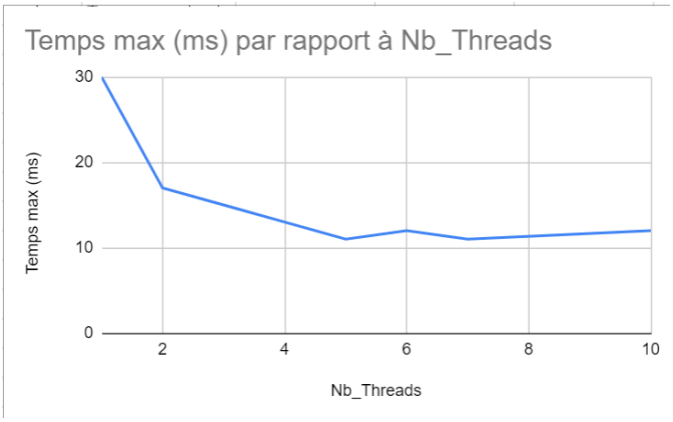

FIGURE 12 – Graphique temps *Game Of Life*

Nous pouvons voir l'intérêt de la programmation concurrente qui permet de diviser par 3 le temps d'exécution du niveau le plus gourmand en ressources. Cependant, il faut bien dimensionner le nombre de Threads. Ici, il est pertinent de mettre 5 Threads. Ce nombre optimal peut varier en fonction de la taille de la grille et des autres tâches à exécuter en même temps ainsi que de la machine que l'on utilise.

5 Bilan

Nous avons vu durant cette étude la puissance de la programmation concurrente. Lorsque nous devons réaliser une tâche gourmande en opérations et que celles-ci sont répétitives, nous pouvons utiliser ce mode de programmation afin d'optimiser nos programmes. Il faut cependant prendre en compte les spécificités du problème et de la machine afin de bien dimensionner le nombre de Threads.