Université de Bourgogne

Rapport de projet de Système distribué

Jeux de la vie en 3D: JAVA RMI

Réalisé par : Abdul-Qadir SANNY

15 Décembre 2022

Table des matières

Chapitre 1

Introduction

Les systèmes distribués sont principalement utilisés pour accélérer les calculs et profiter du parallélisme pour lors de la résolution d'un problème donné. Plusieurs technologie existent pour profiter du paralelisme. Ces technoloogie sont utilisées en fonction de la nature du probleme et des performances recherchées. Lors de notre cours de systeme distribué, nous avons été amené à appliquer les principes de systeme distribué pour la résoltion du probleme le jeux de la vie en 3D dans le but de lancer des simulations en utilisant plusieurs machines afin d'accélérer les traitements et de trouver des configurations stables. Nous avons opté pour JAVA RMI. Dans la suite de ce document nous allons dans un premier temps porté une analyse du probleme, ensuite la specification des pricipale classes et fonction utilisées dans l'implémentation de notre soluton, apres préenter l'architechute logicielle dtaillée de notre application puis une évaluation / comparaison des performances et enfin présenter un jeu de tests.

Chapitre 2

Analyse du sujet

2.1 Le jeux de la vie

Le jeu de la vie, est un automate cellulaire conçu par le mathématicien britannique John Horton Conway en 1970. Ce n'est pas un jeux à proprement dit puisqu'il n'y a pas de joueur. Son évolution est déterminée par son état initial et ne nécessite aucune autre entrée. On interagit avec le jeu de la vie en créant une configuration initiale et en observant comment elle évolue. L'univers du jeux est souvent présenté sous la forme d'une grille en 2 dimensions. Chaque case représente l'état actuelle de la cellule. Les cellules évoluent d'une génération à une autre sur la base d'un certains nombre de règles :

1. Toute cellule vivante ayant moins de deux voisins vivants meurt, comme par sous-population.
2. Toute cellule vivante ayant deux ou trois voisins vivants vit à la génération suivante.
3. Toute cellule vivante ayant plus de trois voisins vivants meurt, comme si elle était surpeuplée.
4. Toute cellule morte ayant exactement trois voisins vivants devient une cellule vivante, comme par reproduction.

La grille initiale constitue la racine du système. La première génération est créée en appliquant simultanément les règles ci-dessus à chaque cellule de la racine, vivante ou morte ; les naissances et les décès se produisent simultanément. Les règles continuent à être appliquées de manière répétée pour créer d'autres générations. Après plusieurs générations, la grille fini toujours par converger vers une configuration dite "stable". Cette configuration se présentation des lors de manière régulière dans les générations qui suivent. Plusieurs configurations ont pu être identifiées et classifiées. On a :

1. Still lifes : Blocks, Loaf etc.
2. Oscillators : Blinker(period 2), Pulsar(period 3), etc.
3. Spaceships : Glider, Light-weight spaceship

2.2 Le jeux de la vie en 3D

Dans notre projet, nous avons implémenté l'algorithme du jeux de la vie en 3D. Dans notre implémentation, nous appliquons exactement les règles appliquées que celle du jeux de la vie 2D. Il faut noter qu'on peut changer certains paramètres. Ici, une cellule a jusqu'à 26 voisins. Ce qui augmente drastiquement le nombre d'opération pour déterminer l'état suivant d'une cellule. C'est pour cela que dans notre projet, nous avons déterminé un algorithme pour paralléliser la simulation. Nous avons opté pour JAVA RMI pour effectuer la parallélisation.

2.3 Généralité sur Java RMI

RMI (Remote Method Invocation) est une technologie fournie à partir du JDK 1.1 pour permettre de mettre en oeuvre facilement des objets distribués.

Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'exécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outil rmic fourni avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

2.4 Structure de données et algorithme

Le but du jeux de la vie est de simuler l'évolution d'une grille de départ jusqu'à retrouver les configurations stables. L'algorithme que nous avons utilisé est très simples.

1. pour chaque cellule de la grille retrouver ces voisins.
2. calculer le nombre de voisin vivants.
3. appliquer la règle établie pour déterminer en fonction de l'état actuel de la cellule et du nombre de voisin vivant l'état suivant de la cellule

4. stocker la nouvelle valeur dans un nouveau tableau pour la prochaine génération.

Cette opération est répétée pour toute les cellules sur plusieurs générations jusqu'à ce qu'on remarque les configurations stables. Afin de paralléliser au mieux notre application, nous avons utilisé JAVA RMI avec le design pattern Bag of task. Ainsi, nous avons un cube (tableau à 3 dimension) qui représente notre environnement contenant des cellules. Ce cube se met à jour après chaque génération. L'état d'une cellule pour la prochaine génération est déterminée dans une fonction qui sera une tache. On a une classe Bag of task pour créer et distribuer les taches (le calcul du prochain état d'une cellule). De ce fait, pour chaque génération, on parcours chaque cellule du cube. Une tache est créée par le Bag of task pour déterminer le prochain état de cette cellule. Un client exécute la tache et le retour est enregistré dans le bag of task. Une fois que toutes les cellules ont été traité pour cette génération, le bag of Task met à jour le cube et l'opération reprend pour la génération suivante. Le traitement est effectué ainsi pendant un nombre de génération qu'on aurait spécifié au début du code.

Chapitre 3

Spécification des classes principales

Nous avons deux classes principales exécutables. La classe **Server** et la classe **Client**. La classe Serveur est utilisée pour démarrer le serveur qui initialise l'environnement et les taches que les client auront auront à effectuer. C'est la classe qui contient l'objet réel BagOfTask. La classe Client est la classe qui sera utilisée par les différentes machines pour se connecter au serveur afin d'exécuter une tache.

3.1 La classe serveur

La classe serveur est le point d'entrée de notre application coté serveur. Cette classe se charge de démarrer le registre RMI, d'instancier et d'enregistrer dans le registre des nom l'objet distribué **BagOfTask** avec les paramètres initiaux tels que les dimensions de la grille, les états initiaux des cellules, le nombre de génération qu'on veut simuler. Une fois démarré, cette classe attends qu'un client se connecte pour lui fournir via le bagOfTask une tache à exécuter

```
        try{
        Registry registry = LocateRegistry.createRegistry (1099);
        initCells(arg);
        Grid grid = new Grid(rows, cols, depths, cells);
        BagOfTaskImpl bot = new BagOfTaskImpl(grid, MAX_GEN);
        String nom="mybagoftask";
        String url = "rmi://" + InetAddress.getLocalHost().getHostAddress()
        registry.bind(nom, bot);
        System.out.println("Enregistrement␣de␣l'objet␣avec␣l'url␣:␣" + url)
    }
        catch (Exception e){System.err.println("Erreur␣:"+e);}
    }
```

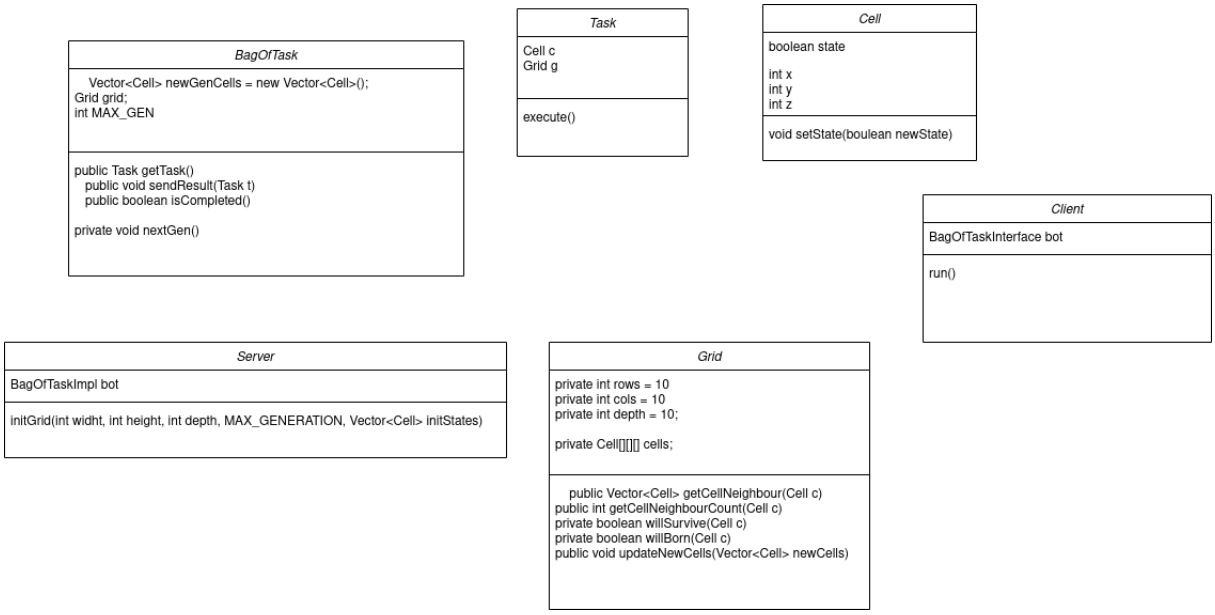

FIGURE 3.1 – Diagramme des différentes classes principales de notre application

3.2 La classe client

La classe client est le point d'entrer pour l'exécution des taches coté client. Cette classe se charge de se connecter dans un premier temps au serveur puis de récupérer une référence sur l'objet distribué distant bagOfTask à partir de son nom. Ensuite, dans une boucle on récupère une nouvelle tache si disponible, on l'exécute et toujours via la référence de l'objet bagOfTask, on retourne le résultat de la tache(c'est à dire en l'occurrence l'état suivant d'une cellule). Le processus se termine lorsqu'il n'y a plus de tache à exécuter. C'est l'objet bagOfTask via la fonction **isCompleted()** qui retourne si toutes les taches ont été terminées.

```
String nom="mybagoftask";
String serveur = InetAddress.getLocalHost().getHostAddress();
if(args.length == 1){
    serveur = args[0];
}
String url = "rmi://" + serveur + "/"+ nom;
System.out.println("Obtention de l'objet distant de l'url : " + url
try{
    BagOfTask bot=(BagOfTask) Naming.lookup(url);
    Task t = bot.getTask();
    while(!bot.isCompleted()){
        while (t != null){
```

```
                t.execute();
                bot.sendResult(t);
                t = bot.getTask();
            }
        }
    }
        catch (Exception e){
        System.err.println("Erreur :"+e);
    }
```

C'est via la fonction **sendResult(Task task)** qu'un client peut envoyer le résultat d'une tache terminée :

```
    public void sendResult(Task task) throws RemoteException {
     TaskImpl t = (TaskImpl)task;
     Cell newCell = new Cell(t.c.x, t.c.y, t.c.z, t.nextState);
     this.newGenCells.add(newCell);

     if (newGenCells.size() == grid.getCols() * grid.getRows() * grid.ge
         grid.updateNewCells(newGenCells);
         System.out.printf("Gen %d termin \n", this.currentGen);
         grid.print();
         endOfGen = true;
         nextGen();
     }
}
```

La fonction **nextGen** réinitialise le bagoftask pour la génération suivante.

3.3 La classe BagOfTask

La classe BagOfTask est la classe qui se charge de créer des taches à distribuer entre les clients. Cette classe contient la grille actuelle(l'ensemble des cellules) et un tableau qui contiendra l'état des cellules de la prochaine génération. Ce tableau est rempli au fur et à mesure qu'une tache est terminée par un client. Grâce à une fonction **getTask()**, un client peut prendre une tache et l'exécuter si disponible. En effet, La fonction **Task getTask()** retourne une nouvelle tache (un objet de type Task) qui a pour but de déterminer l'état prochain d'une cellule. La grille est mise à jours une fois que, pour une certaine génération, toutes les tâches ont été effectuées. Si le nombre

maximale de génération voulu est atteint la fonction **isCompleted()** va retourner **true** et les client n'auront plus à attendre une prochaine tache. Dans le cas contraire, les taches pour la génération suivante sont relancées. Notons que nous avons une interface et une classe d'implémentation pour le BagOfTask. Les client n'ont accès qu'à l'interface.

```
        public Task getTask() throws RemoteException {
        if (this.startTime == -1){
            this.startTime = System.currentTimeMillis();
        }
        if(endOfGen) {
            return null;
        }
        Cell currentCell = this.grid.get(currentI, currentJ, currentK);
        Task t = new TaskImpl(currentCell, this.grid);
        currentI++;
        if (currentI == this.grid.getRows() ){
            currentI = 0;
            currentJ++;
            if (currentJ == this.grid.getCols() ){
                currentJ = 0;
                currentK++;
                if (currentK == this.grid.getDepth() ){
                    currentK = 0;
                    // this generation is complete
                }
            }
        }
        return t;
    }
```

3.4 La classe Task

Cette classe est utilisée uniquement pour exécuter une tache du BagOfTask. Elle contient une référence de la grille et la cellule pour laquelle on souhaite déterminer l'état suivant. La tache à exécuter est de déterminer le nombre de voisin vivants de la cellule puis de déterminer l'état prochain de la cellule.

3.5 Les classes Cell et Grid

La classe Grid représente notre environnement. Il contient principalement un tableau à 3 dimensions de cellules. Ce tableau représente l'état courant de toutes les cellulles. Plusieurs fonctions imprtantes sont présentes dans cette classe.

3.5.1 Vector getCellNeighbor(Cell) et int getCellNeighborCount(Cell)

Ces deux fonctions sont utilisées pour trouver le nombre de voisins vivant d'une cellule de la grille. Les voisins d'une cellules sont celles aux coordonnées ayant les différente combinaisons de x-1, x, x+1, y-1, y, y+1, z-1, z, z+1 (omis x,y,z bien évidemment). Une boucle suffit pour les découvrir et éliminer dans le même temps les cellules qui sont hors de la bordure de notre environnement (grâce à la fonction **boolean isInside(c Cell)**.

```
        public Vector<Cell> getCellNeighbor(Cell c){
        Vector<Cell> neighbor = new Vector<Cell>();
        for (int i = c.x -1; i <= c.x+1; i++) {
            for (int j = c.y -1; j <= c.y+1; j++) {
                for (int k = c.z -1; k <= c.z+1; k++) {
                    if (isInside(i, j, k) && c != this.cells[i][j][k]){
                        neighbor.add(this.cells[i][j][k]);
                    }
                }
            }
        }
        return neighbor;
    };
    }
```

3.5.2 boolean getNextCellState(Cell)

Cette fonction détermine l'état suivant d'une cellule. Deux scénarios possibles :

— si la cellule est vivante, appelle de la fonction **boolean willSurvive(Cell c)** pour savoir si elle restera vivante à la prochaine génération.

— si la cellule est morte, appelle de la fonction **boolean willBorn(Cell c)** pour savoir si elle naîtra à la prochaine génération.

```
        public boolean getNextCellState(Cell c){
```

```
        if (c.isAlive) return willSurvive(c);
        else return willBorn(c);
    }
```

3.5.3 void updateNewCells(Vector¡Cell¿ newCells)

Cette fonction est utilisée pour mettre à jours le tableau (à 3 dimensions) de cellule avec les données de la nouvelle génération calculée préalablement par les machines distantes. C'est cette fonction qui est appelée par le *bagOfTask* une fois que le calcul sur toutes les cellules ont été terminée pour une génération donnée.

```
        public void updateNewCells(Vector<Cell> newCells) {
        for (int c = 0; c < newCells.size(); c++) {
            this.cells[newCells.get(c).x][newCells.get(c).y][newCells.get(c)
        }
    }
```

3.5.4 Affichage

Nous avons une fonction d'affichage qui donne les coordonnées d'une cellule ainsi que son état sous le format

```
(x, y, z, state)
```

. *state = true* si la cellule est vivante et *false* sinon. Étant donnée la taille énorme des données à afficher, nous affichons uniquement les cellules vivantes lors de l'affichage. C'est pour cela que dans les test on ne verra que des cellules avec la valeur *true*

```
public void print(){
    System.out.println("Liste des cellules vivantes : ");
    boolean isEmpty = true;
    for (int i = 0; i < this.rows; i++) {
        for (int j = 0; j< this.cols; j++){
            for (int k = 0; k< this.depth; k++){
                Cell c = get(i, j, k);
                if (c.isAlive){
                    isEmpty = false;
                    System.out.printf("%s\n", c);
                }
```

```
                }
            }
        }

        if(isEmpty){
            System.out.printf("Aucune cellule vivante\n");
        }
    }
```

.

Chapitre 4

Architecture logicielle

Nous avons utilisé comme architecture logicielle le design pattern BagOfTask. Ce design permet de découper un travail en petites tâches indépendantes et de les exécuter de manière parallèle sur plusieurs threads ou sur plusieurs machines. Le "Bag of Tasks" est particulièrement utile lorsqu'il y a beaucoup de tâches à exécuter et que chacune d'elles peut être exécutée de manière indépendante des autres. Ce qui est le cas de notre problématique. Cela permet d'améliorer les performances en utilisant au mieux les ressources disponibles. Dans notre l'implémentation, les tâches représente le calcul de la prochaine étape d'une cellule. Ces tâches sont créer au dur et à mesure qu'un client en fait la demande. Nous n'avons pas un tableau de tâches toutes faites car cela pourrait prendre du temps pour les créer à l'avance pour chaque génération (une tâche pour chaque cellule). Les clients peuvent être vu comme des **workers** qui tournent et exécutent les tâches. Sur une même machine nous pouvons donc avoir plusieurs clients (les workers sont séparés en processus).

Étant donné que nous utilisons JAVA RMI comme echnlgie de paralélsation de tâche, nous avons implémenter le design pattern "Proxy". Le design pattern "Proxy" implémente un objet qui agit comme un substitut pour un autre objet. Dans le cas de RMI, le proxy est un objet qui est exécuté sur l'hôte client et qui agit comme un substitut pour l'objet distant exécuté sur l'hôte serveur. Le proxy envoie les invocations de méthode au serveur et renvoie les résultats au client, en gérant les communications et en cachant la complexité de l'implémentation de l'objet distant.

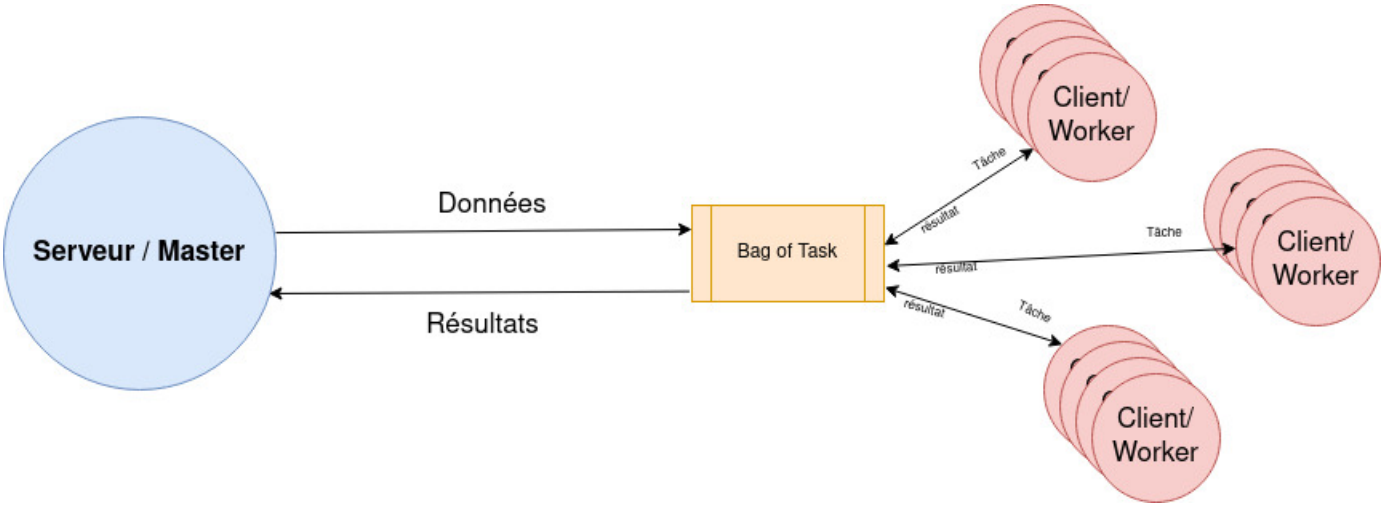

FIGURE 4.1 – Bag of task dans notre systeme

Chapitre 5

Évaluation des performances

La performance de notre application est déterminée par le temps d'exécution de toutes les générations suivant le nombre de machines qui se connectent. Pour déterminer cette métrique, dès lors qu'une machine se connecte au serveur et récupère la première tache nous enregistrons le timestamp de début. Une fois une que toutes les générations ont été déterminées, nous enregistrons le timestamp de fin. La durée du traitement est la différence entre ces deux temps. Nous déterminons ainsi la durée du traitement en faisant varié le nombre de machine qui participent à l'opération.

5.1 Plusieurs processus sur notre machine

Nous avons commencé les expérimentations sur notre propre machine dont les caractéristiques sont les suivantes :

— HP Intel® Core™ i5-8300H CPU @ 2.30GHz × 8

— Ubuntu 20.04.5 LTS 64 bit

— jdk openjdk 11.0.17 2022-10-18.

Nous avons évaluer les performances de notre application avec différentes données. La taille par défaut de notre grilles est de 15 * 15* 15. On simule jusqu'à la génération 5. Les cellules vivantes au départs sont :

(4, 5, 5), (5, 4, 5), (5, 5, 4), (5, 5, 5), (5, 5, 6), (5, 6, 5), (6, 5, 5)

On remarque bien que la tendance est baissière. Le temps d'exécution diminue avec notre le nombre de processus client. Par contre on note que avec 4 processus on a eu un temps de 36912 alors que avec 5 machines on a enregistré 37483. Cette légère augmentation peut-être du au fait que notre machine n' pas pu efficacement repartir les 5 processus couplés avec d'autres tâches de fond.

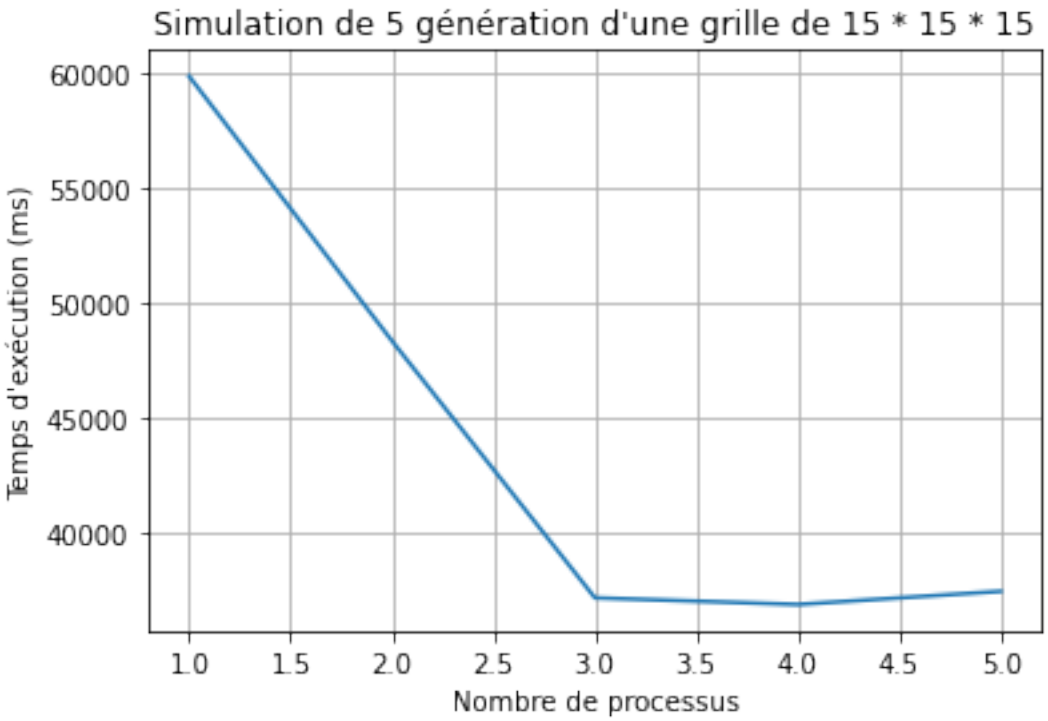

FIGURE 5.1 – Temps de simulation de 5 générations sur notre machine

Nombre de processus	**Temps d'exécution en (ms)**
1	123398
2	105256
3	98856

FIGURE 5.2 – Temps de simulation de 10 générations sur notre machine distribuée entre processus

Nous avons lancer la simulation avec 50 générations mais cela prenais beaucoup trop de temps sur notre machine. Le tableau 5.1 donne le temps après exécution de 10 générations sur trois processus.

5.2 Plusieurs machines

Pour tester les performances avec plusieurs machines, nous avons utilisés les machines de la salle 104 de la faculté. Nous nous sommes connectés à ces machines via VPN avec le client X2Go. Nous avons commencé les expérimentations avec les même données que sur notre machine locale. On obtiens le graphe 5.3 après avoir testé sur 7 machines.

On remarque clairement que les performances augmentent avec le nombre de machine qui participent à l'opération. Notre système est donc bien distribué.

Une comparaison avec les résultats obtenus sur notre machine locale en distribuant les taches

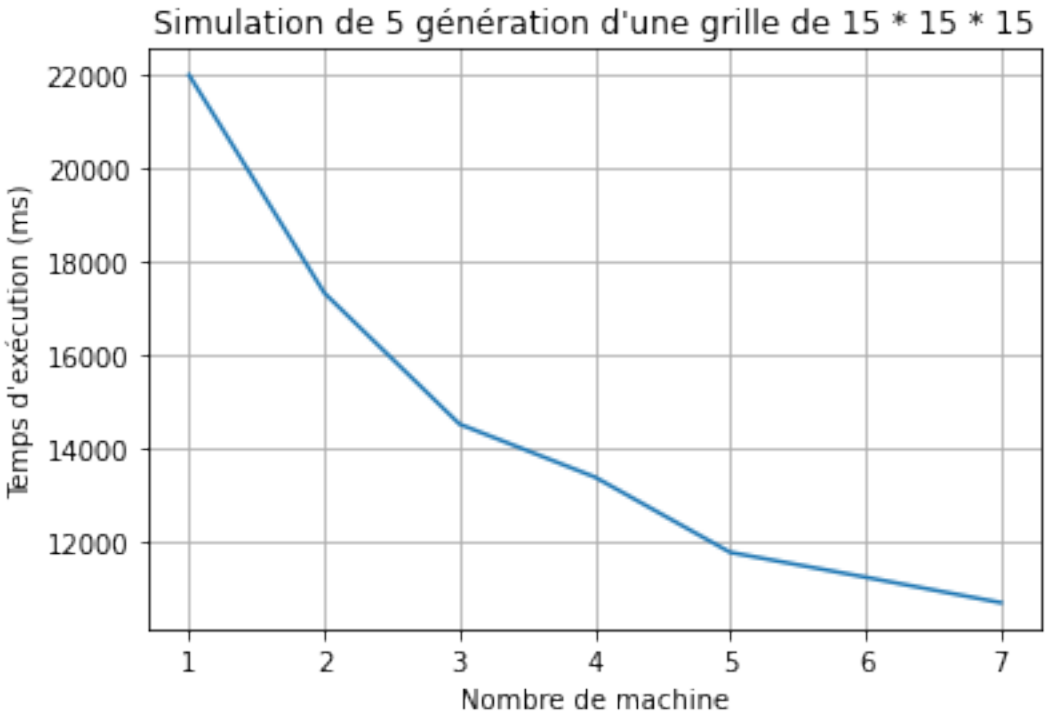

FIGURE 5.3 – Temps de simulation de 5 générations sur 7 machines de la salle 104

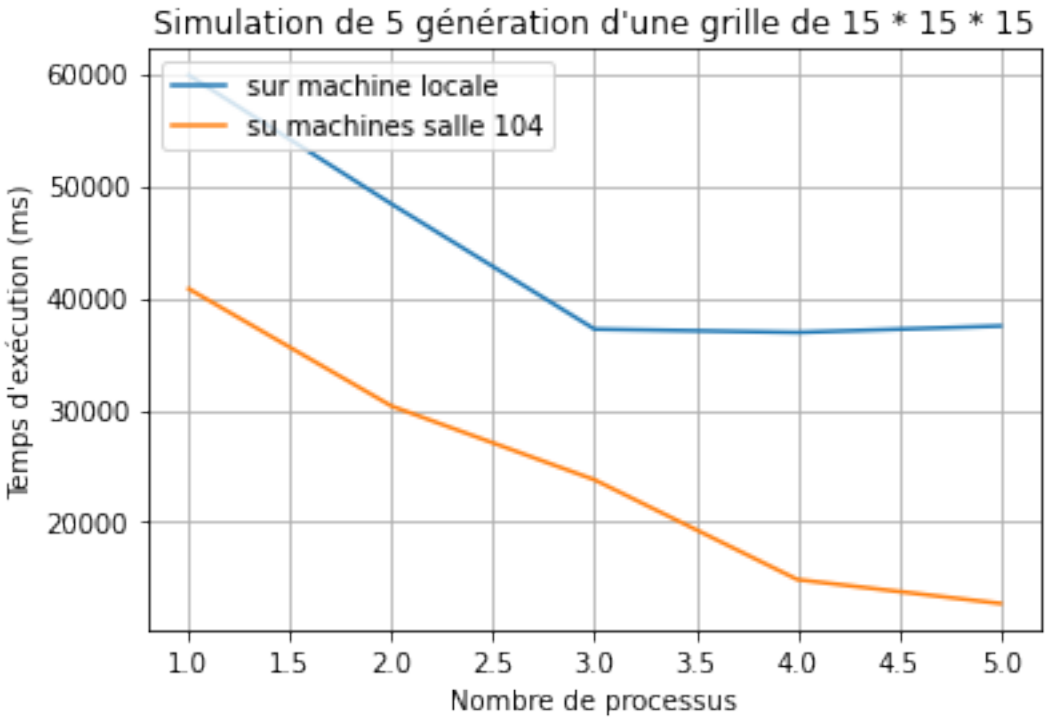

FIGURE 5.4 – Comparaison du temps d'exécution sur 5 processus locale et 5 machines distribuées

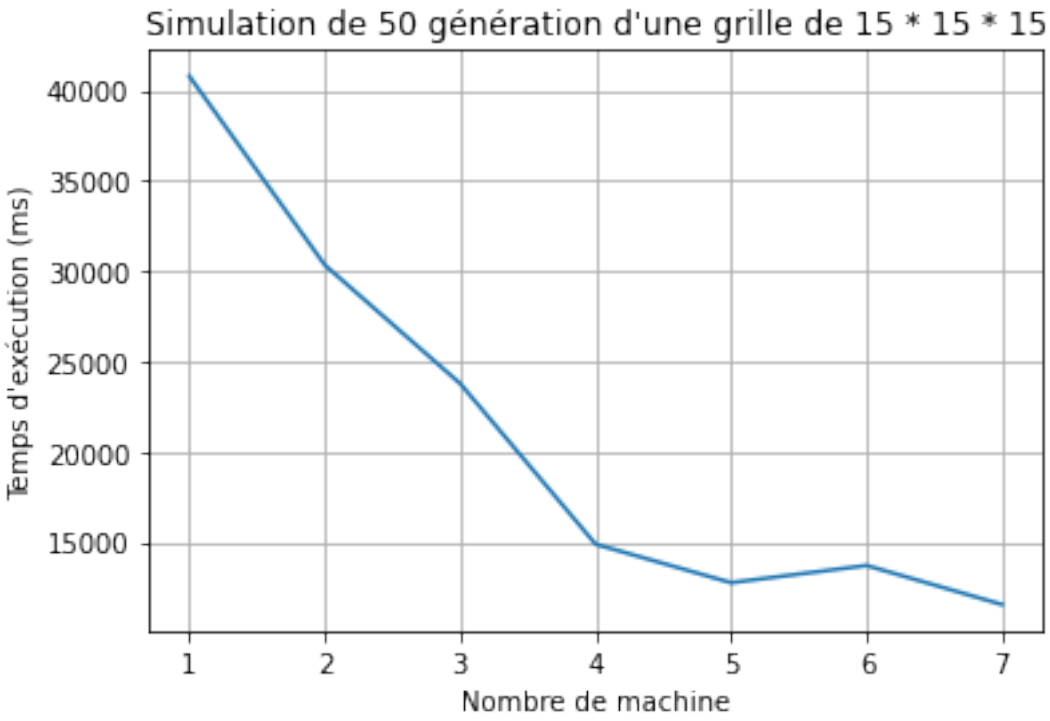

FIGURE 5.5 – Temps de simulation de 50 générations sur 7 machines de la salle 104

entre 5 processus montre que l'utilisation de plusieurs machines différentes est beaucoup plus efficaces que plusieurs processeurs processus sur la même machine. La figure 5.4 illustre bien cela.

Nous avons ensuite tester sur 50 générations avec un nombre de machine allant de 1 à 6 et une machine avec deux processus. On obtient le graphique 5.5.

Comme pour les résultats précédents, même avec une grande quantité de données à traiter la performance augmente avec le nombre de machine en jeux. Par contre on remarque également que lorsqu'on utilise sur la même machine plusieurs processus on note une légère baisse de performance.

Chapitre 6

Exécution du code

Le programme complet et à jours est disponible en acces libre sur mon github à l'adresse https://github.com/qsanny/gameOfLife3D. Pour exécuter notre application du jeux de la vie 3D distribuée avec JAVA RMI, il faut avoir installé une version de java. Nous avons travaillé avec le jdk openjdk 11.0.17 2022-10-18.

1. compiler toutes les classes :

   ```
   javac *.java
   ```

2. exécuter le serveur :

   ```
   java Serveur
   ```

   Ce dernier va exécuter le serveur avec les paramètres par défaut. On peut ajouter en paramètre le chemin vers un fichier de données :

   ```
   java Serveur input.txt
   ```

   Ce fichier doit avoir le format suivant :

   ```
   5
   15 15 15
   4 5 5
   6 5 5
   5 4 5
   5 6 5
   5 5 4
   5 5 6
   ```

   — La première ligne contient le nombre de génération qu'on veut simuler.

— La seconde ligne défini les dimension de notre environnement. Respectivement la largeur, longueur et profondeur
— Les lignes suivantes représentes les coordonnées des cellules vivantes sur le format x y z. Attention une vérification de la cohérence des coordonnées n'est pas faites. Veuillez à ce que les cordonnées fournies appartiennent bien aux dimensions de l'environnement.

3. exécuter les clients sur chaque machine qui doit participer au calcul distribué en mettant en paramètre l'adresse du serveur :

```
java Client 172.31.18.37
```

Si le client s'exécute sur la même machine que le serveur, l'adresse du serveur n'est pas nécessaire

```
java Client
```

Chapitre 7

Jeu de tests

7.1 Sur notre machine

Nous avons testé notre programme sur un certain nombre de génération avec plusieurs processus sur notre machines.

Les images 7.1, 7.2 suivantes montrent l'exécution et les résultat sur notre machine :

7.2 Sur plusieurs de la salle 104

Nous avons ensuite testé notre programme sur un certain nombre de génération avec plusieurs machines de la salle de TP 104 . Les images 7.4, 7.3 suivantes montrent l'exécution et les résultats :

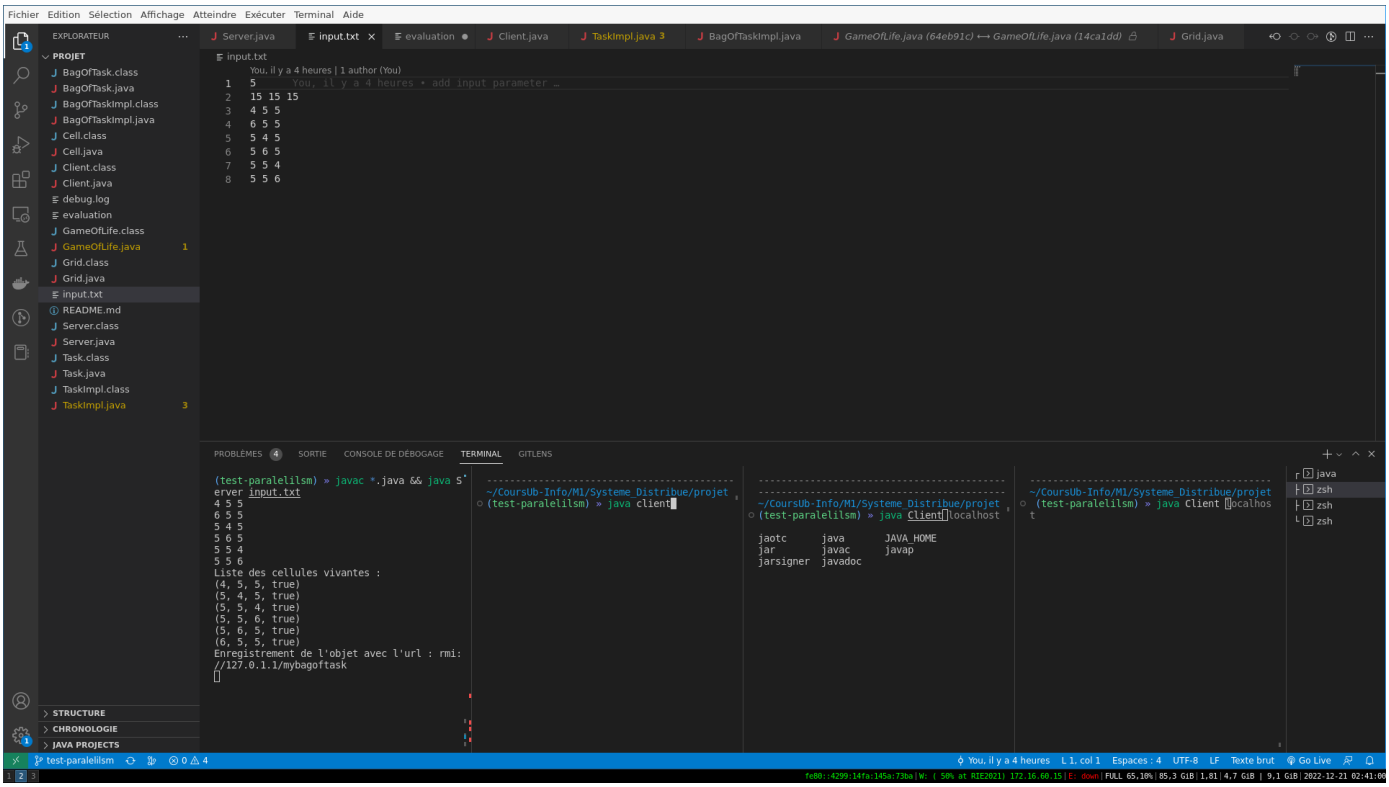

FIGURE 7.1 – Démarrage de l'exécution de notre programme sur notre machine avec 3 processus différents

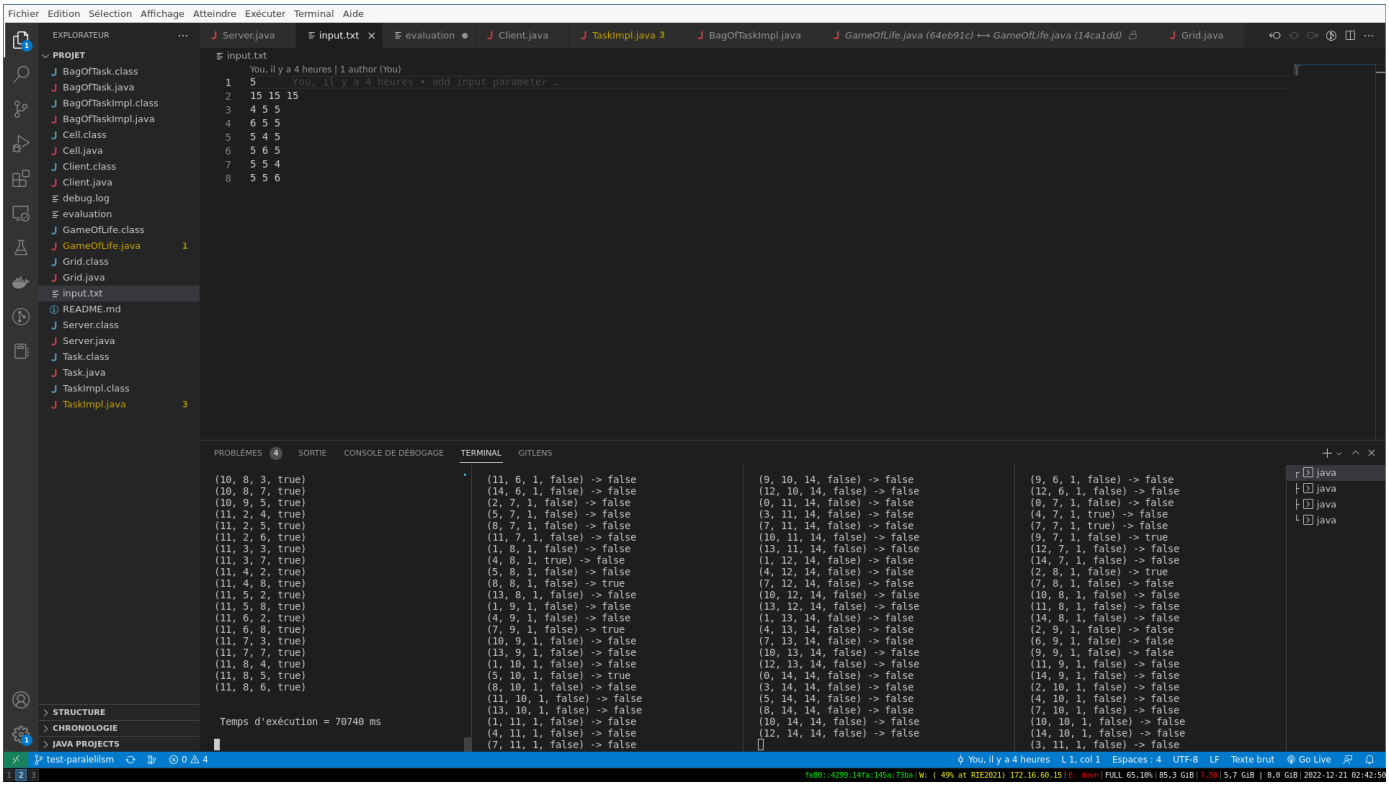

FIGURE 7.2 – Fin de l'exécution de notre programme sur notre machine avec 3 processus différents

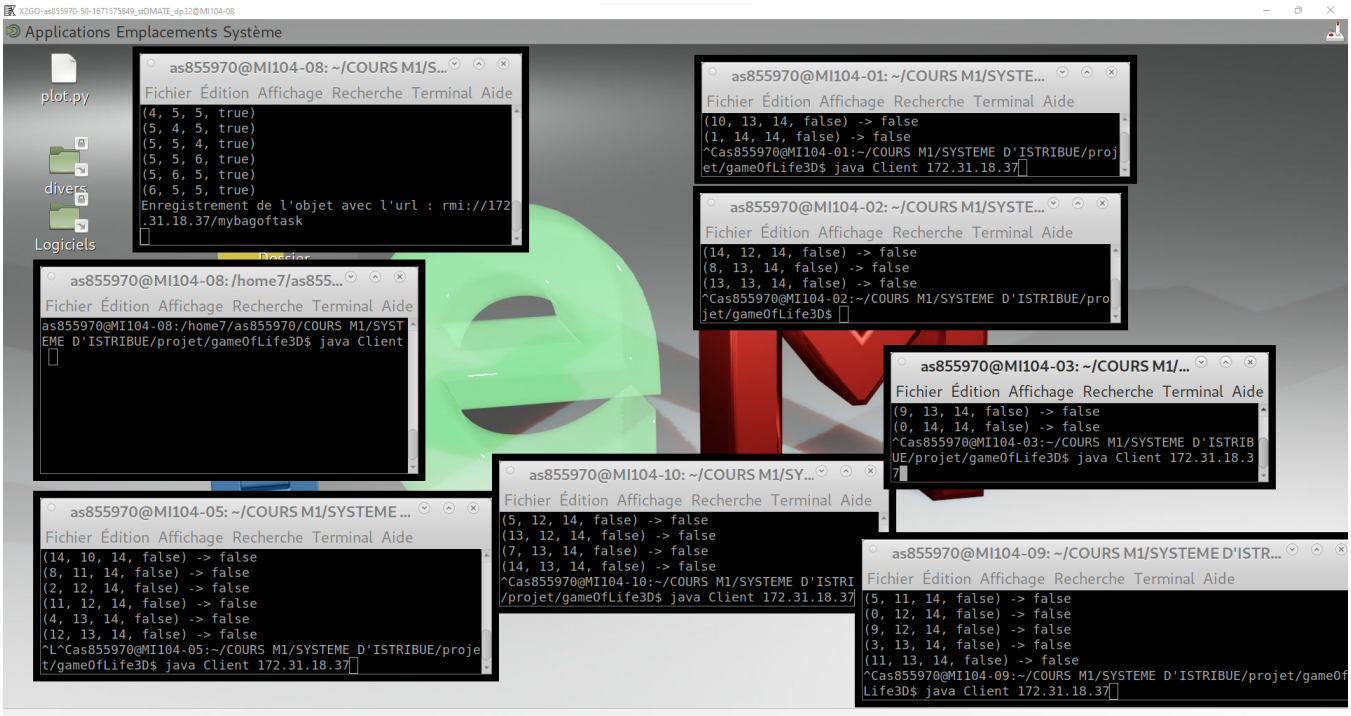

FIGURE 7.3 – Démarrage de l'exécution de notre programme sur 7 machines différentes

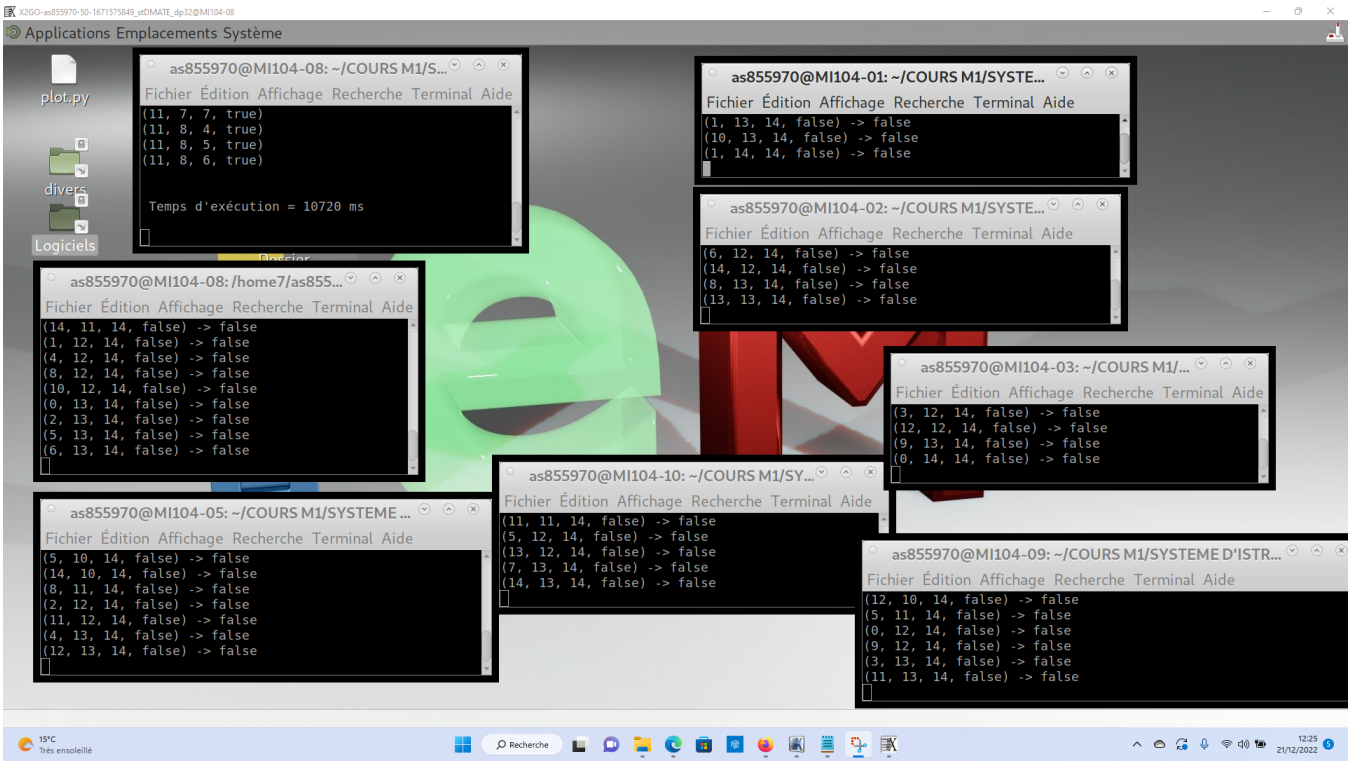

FIGURE 7.4 – Fin d'exécution de notre programme sur 7 machines différentes

Conclusion

Tout au long de ce projet nous avons travaillé à appliquer les notions de systèmes distribués étudier au cours afin d'accélérer l'exécution de la simulation de plusieurs génération du jeux de la vie sur une grille d'espace en 3 dimensions. Après plusieurs test sur notre machine locale avec plusieurs processus puis sur plusieurs machines de la salle 104 de TP, on peut conclure que notre système est bien distribué et on note un gain significatif de performance. Des expérimentations approfondie peuvent encore être menée avec un très grand nombre de machine et de génération a simuler. On pourrait également revoir le découpage des tâches. Au lieu d'envoyer une cellule par tâches, on peut envoyer un certains nombre de cellule à traiter. Cela pourrais augmenter les performances du système. De même, on peut penser à utiliser MPI au lieu de RMI pour la parallélisation des tâches et étudier la performances.

webographie

[1] https://content.wolfram.com/uploads/sites/13/2018/02/01-3-1.pdf. Consulté le 10/12/2022.

[2] https://www.jmdoudoux.fr/java/dej/chap-rmi.htm Consulté le 10/12/2022

[3] https://en.wikipedia.org/wiki/3D$_Life$ 10/12/2022.