

Jeu de la vie

Belli Fabien – Menagé Emmanuel – Gorrieri Cyril – Lestel Guillaume

Table des matières

Jeu de la vie.....	1
Principe.....	3
Les phénomènes intéressants.....	5
Arrêt de la génération.....	5
Génération infini.....	5
Apparition de formes.....	5
Algorithmes de calcul implémentés.....	5
Algorithme linéaire.....	5
Algorithme de calcul.....	6
Inconvénient.....	6
Algorithmes avec threads.....	7
Un thread par case.....	7
Principe.....	7
Section critique.....	7
Fonctionnement.....	7
Utilisation des barrières.....	7
Représentation dans une architecture multi-processeur	8
.....	8
Division de la zone de jeux en 4.....	8
Algorithme.....	8
Division de la zone de jeux en N.....	9
Algorithme des voisin.....	10
Introduction.....	10
Explication par l'exemple (grille 5x5).....	10
Implémentation de la méthode itérative.....	11
Implémentation de la barrière pour un thread par case.....	12
Expérimentations.....	14
Méthode itérative.....	14
Nombre maximum de thread.....	14
Test 1.....	14
Test 2.....	14
Résultats graphiques.....	15
Remarques.....	16
Modélisation FSP.....	16
Programmation.....	16
Diagrammes.....	17
Workers.....	17
Barrière.....	17
Résultats.....	17
Safety.....	17
Comparaison des méthodes.....	19
Méthode itérative.....	19
Méthodes des voisins.....	19
Méthode de la barrière.....	19
Méthode écrivain-rédacteur.....	19
Pour aller plus loin.....	20

Principe

Le jeu de la vie fonctionne sur une grille (un plateau).

Ce jeu contient un nombre fixe de cases, chaque case est soit activée ou désactivée.

Voici par exemple une capture d'écran de la première étape du jeu de la vie :

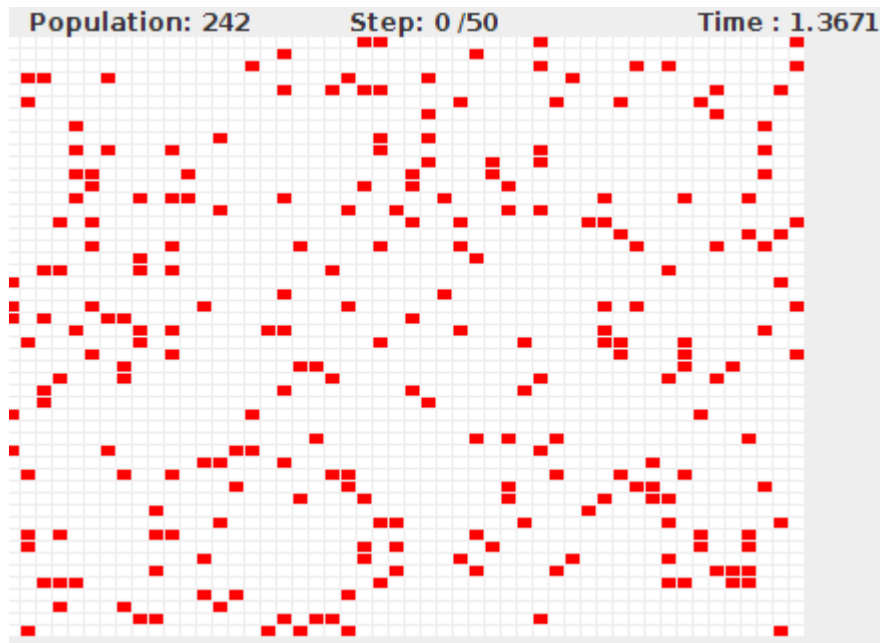


Illustration 1: Capture d'écran du jeu de la vie

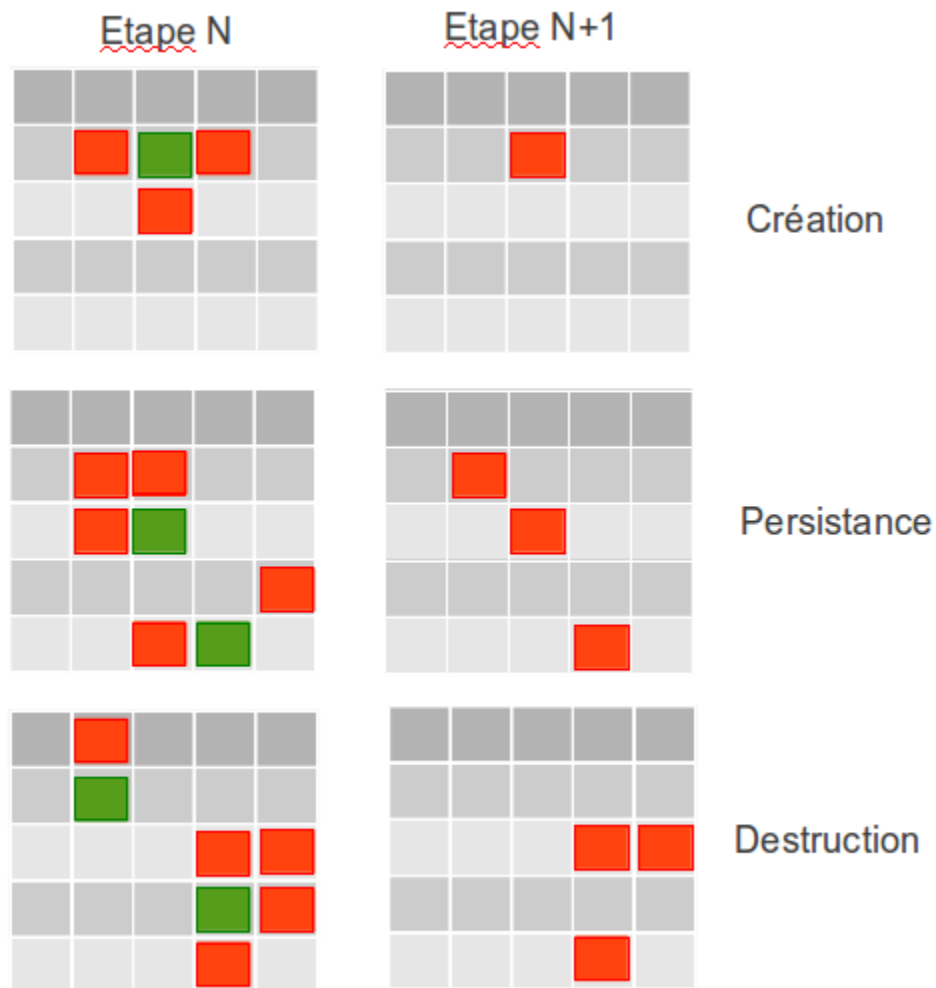
Cette grille peut être remplie au départ de manière aléatoire, ou par des cases choisies par l'utilisateur.

Le jeu n'a pas vraiment de but, mais consiste en une succession d'étapes pouvant éventuellement amener à un état terminal dans lequel les cases ne bougent plus.

À chaque étape, les cases prennent une nouvelle valeur qui est calculée en fonction de ses voisins.

Si le nombre de voisins est de :

- 3 alors la case prend vie et devient active (rouge dans la capture d'écran).
- 2 et la case était active au tour précédent, alors elle reste active au nouveau tour.
- < 2 alors la case est désactivée par famine.
- > 3 alors la case est désactivée par surpopulation.

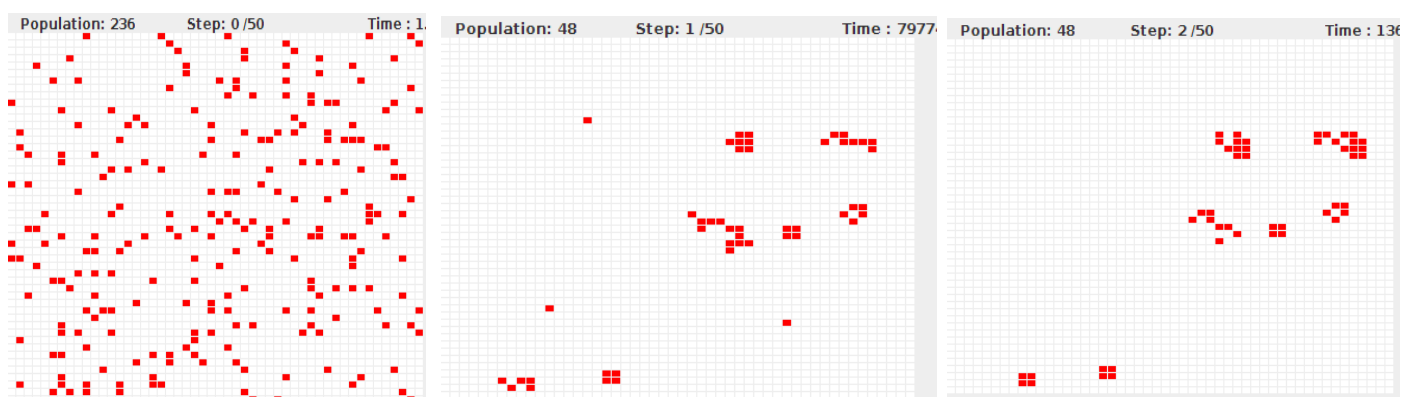


Voici la liste des voisins à vérifier pour connaître la future valeur de la case «ici».

Ainsi les valeurs de « ici » peuvent varier de 0 à 8.

1	2	3
4	ici	5
6	7	8

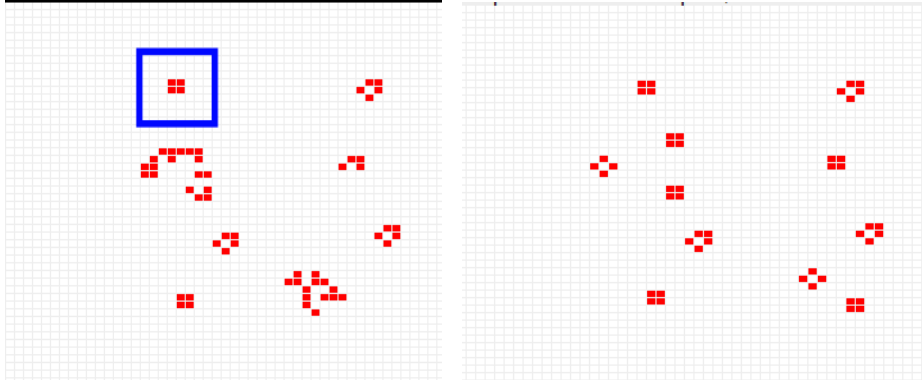
Exemple :



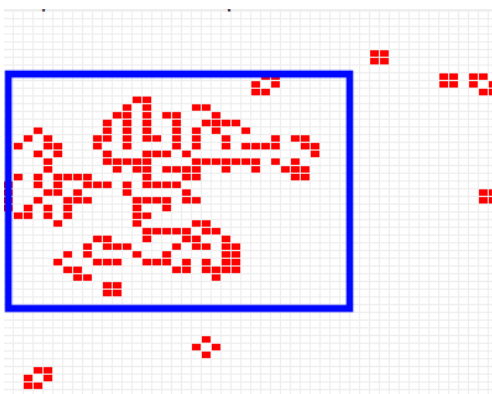
Ci dessus des captures d'écran montrant les 3 premières étapes du jeux de la vie.

Les phénomènes intéressants

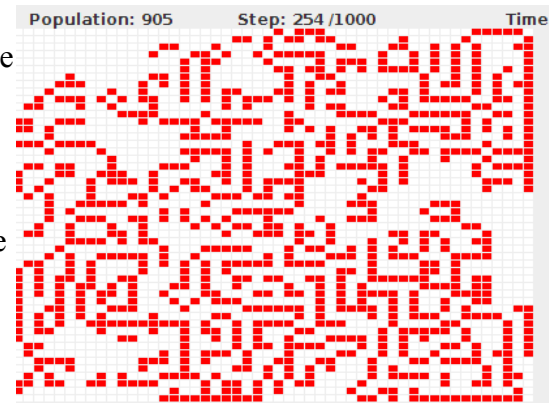
Arrêt de la génération



Génération infini



La partie en bleue ne cesse de grandir continuellement jusqu'à remplir l'intégralité de la grille en formant une sorte de labyrinthe.



Apparition de formes

source : <http://islwww.epfl.ch/biowall/VersionF/ApplicationsF/LifeF.html>

Algorithmes de calcul implémentés

Algorithme linéaire

Cette algorithme parcourt la grille dans son intégralité à chaque étape.

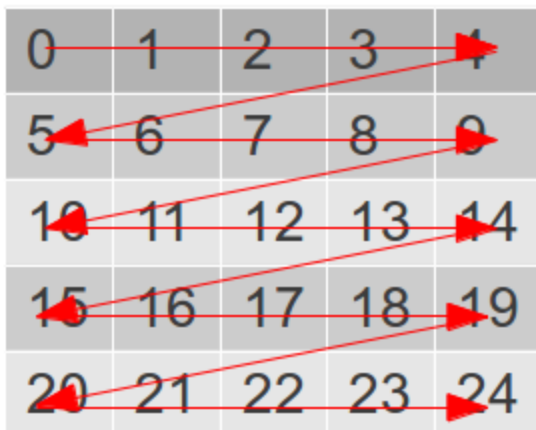
A chaque case analysée, les voisins sont comptés, puis la valeur calculé est sauvegardé dans une nouvelle grille.

Exemple :

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

Ainsi le calcul pour la valeur de la case 0 nécessite de connaître les valeurs des cases 6, 7 et 1.

Dans cette exemple il est nécessaire de faire 30 calculs les uns a la suite des autres dans un seul thread pour chaque étape.

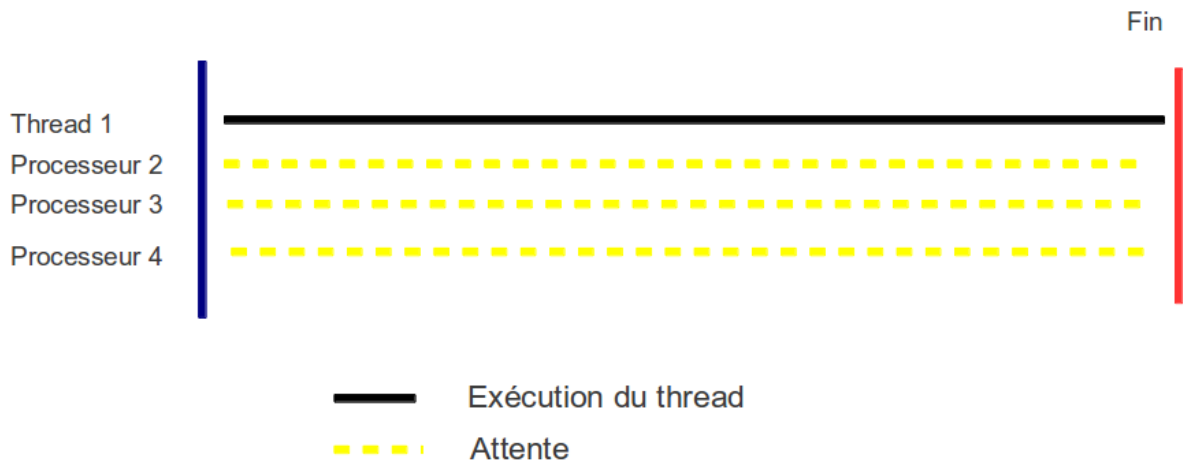


Algorithme de calcul

Le calcul des cases se fait séquentiellement.

Inconvénient

Les calculs se font séquentiellement et l'utilisation des cela crée un perte de temps car on attend d'être arrivé a la fin pour continuer.



On ne peut pas exploiter tous les processeurs de l'ordinateur.

Un seul thread sera utilisé sur un seul processeur.

Algorithmes avec threads

Un thread par case

Principe

Le but étant de faire calculé la prochaine étape de calcul d'une case par un thread.

Section critique

Il n'y a aucune section critique, en effet il n'y a pas de zone partagée. La grille qui est lue n'est pas modifiée mais une autre grille temporaire est créée, dans laquelle on insère les nouvelles données.

Fonctionnement

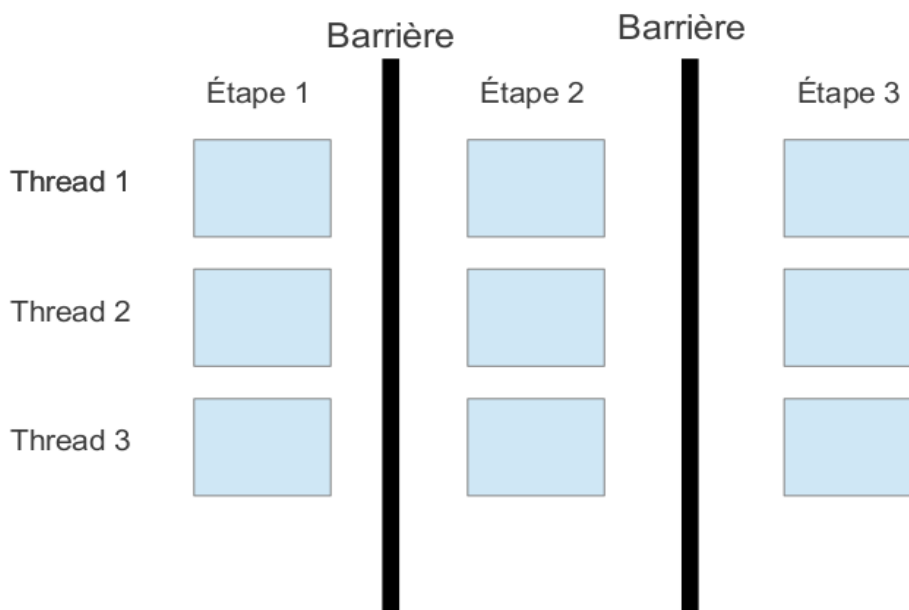
Le problème réside dans le fait que les threads doivent être synchronisés, c'est à dire qu'un thread doit être informé que les valeurs de ses voisins sont à jour et ne correspondent pas aux anciennes valeurs. En effet un thread (une case) serait capable d'être calculé jusqu'à la dernière étape alors que ses cases voisines n'y seraient pas encore.

Il est donc nécessaire de « faire attendre » le thread que tous les autres aient calculés leur nouvelle valeur.

Utilisation des barrières

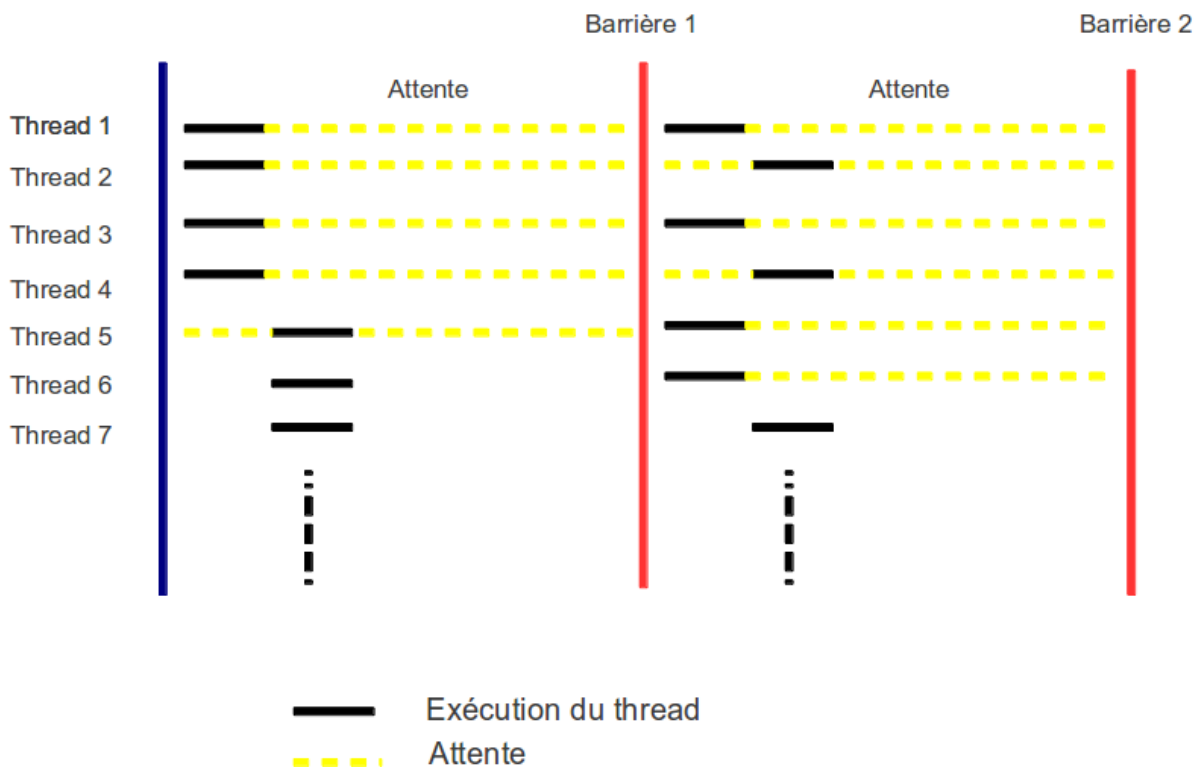
Afin d'informer les threads que tous les calculs ont été effectués, c'est à dire que toutes les cases de la grille ont été calculées, il est nécessaire d'utiliser deux compteurs. Ces deux compteurs sont initialisés à la valeur de la taille de la grille.

Le premier compteur indique que le thread peut commencer le traitement, le second indique qu'il termine le traitement. Ces compteurs forment des barrières que l'on ne peut franchir que lorsque tous les threads ont terminés leur traitement.



Représentation dans une architecture multi-processeur

Les tests ont été effectués sur une architecture 4 processeurs simulés.



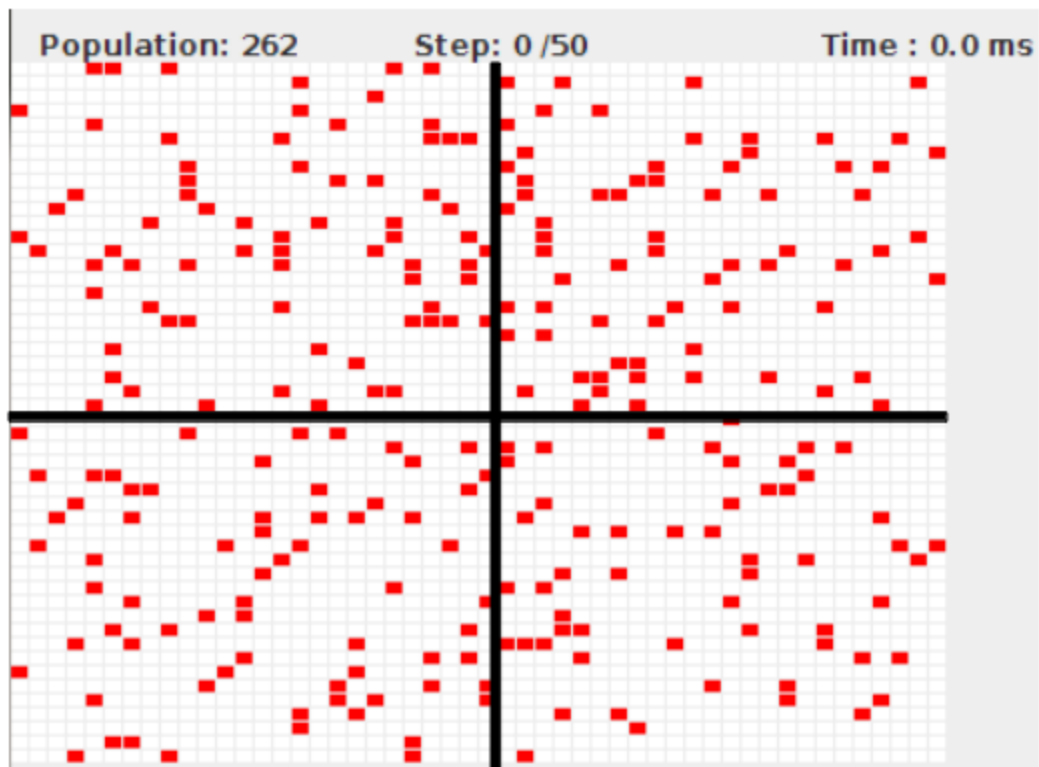
Division de la zone de jeux en 4

Algorithme

La zone de jeux est divisée en 4.

On crée 4 threads qui seront chargés de chaque partie de la zone. Tout ceci est géré par 4 barrières, une pour chaque zone.

Cette méthode est le mélange entre un calcul séquentiel et un calcul par thread. En fait un thread fait un calcul séquentiel sur une partie de la zone de jeux.



Division de la zone de jeux en N

Il est possible de créer un nombre dynamique de thread qui s'occupera d'un section défini du code.

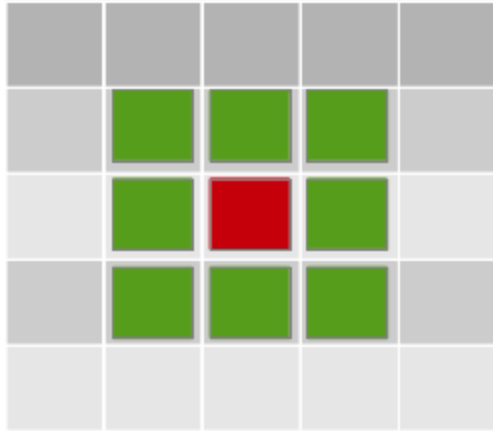
Ainsi si on décide de créer 4 threads, nous nous retrouverons alors dans le cas précédent.

Si on décide de créer autant de thread que de case, alors on se retrouve dans le cas de la barrière défini plus haut.

L'algorithme de synchronisation est celui de la barrière.

Algorithme des voisin

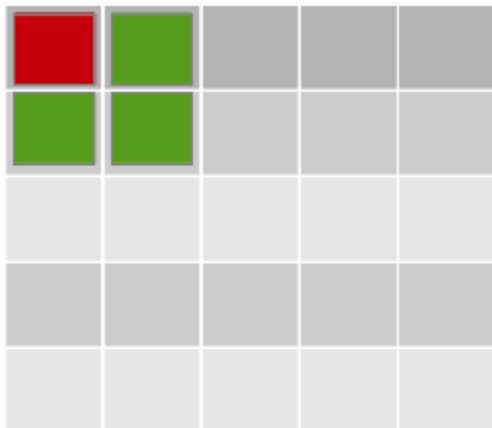
Introduction



Ceci constitue une amélioration de l'algorithme précédent, en effet, pour calculer la valeur de la case rouge, il suffit d'être sûr que les cases vertes (ses voisins), soient calculés. Chaque case est dirigé par un thread.

Ainsi à l'étape N, pour calculer l'étape N+1 de la case rouge, il faut que les valeurs des cases vertes soient connus et qu'il soit possible de dire que la valeur est valable.

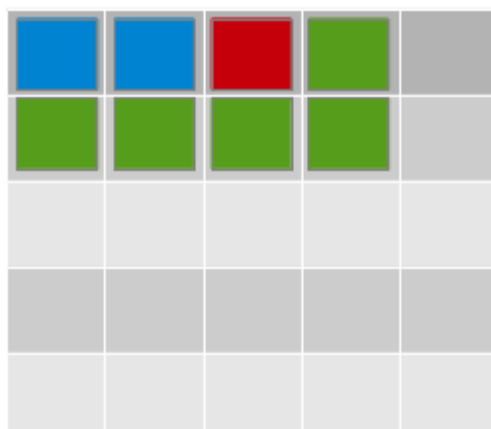
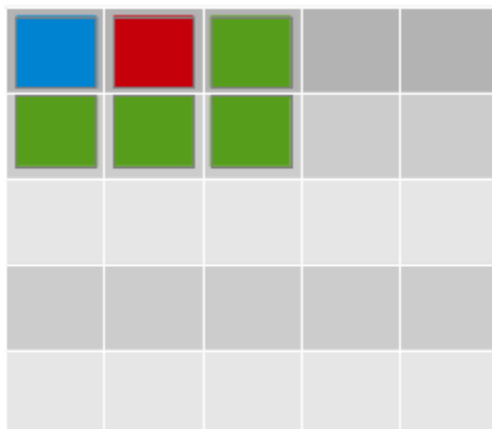
Explication par l'exemple (grille 5x5)



Voici la première itération, on cherche à calculer la valeur de la case rouge.

Mais pour calculer sa valeur, il faut que les valeurs des cases vertes soient connus. Pour calculer ces valeurs il faut que les valeurs de ces cases soient aussi connues.

Légende :
rouge : case en cour de calcul
vert : case l'étape N à connaître
bleue : case à l'étape N+1



Implémentation de la méthode itérative

Pour chaque étape, le calcul de l'état suivant se fait normalement en parcourant toutes les cases de la grille.

Voici le code de la méthode **nextStep()** :

```
public void nextStep(){  
    // Les données sont lues dans une copie de la grille  
    Field tmp = new Field(currentField);  
  
    for(int i = 0; i<tmp.getDepth(); i++){  
        for(int j = 0; j<tmp.getWidth(); j++){  
  
            // Compte le nombre de case adjacente vraie  
            int nbadj = tmp.nbAdjacentTrue(i, j);  
            boolean isAlive;  
  
            if (nbadj == 3 || (nbadj == 2 && tmp.getState(i, j))){  
                isAlive = true;  
            } else {  
                isAlive = false;  
            }  
  
            // Positionne la case courante à la bonne valeur.  
            currentField.place(isAlive, i, j);  
        }  
    }  
}
```

Tout se passe de manière linéaire sans concurrence.

Implémentation de la barrière pour un thread par case

Utilisation de la classe `CyclicBarrier`

```
import java.util.concurrent.CyclicBarrier;
```

Déclarations des barrières :

```
// Barrière annonçant le début du travail des threads
final CyclicBarrier barrierStart;

// Barrière attendant la fin du travail des threads
final CyclicBarrier barrierEnd;
```

Ici il est préférable de créer deux barrière pour permettre au premier thread de commencer. Ces barrières sont **final** pour ne pas pouvoir les modifier après création.

```
barrierStart = new CyclicBarrier(rowNumber * lineNumber + 1, null);
barrierEnd = new CyclicBarrier(rowNumber * lineNumber + 1, new
Runnable())
```

Les barrières sont initialisées au nombre de case de la grille, plus un afficheur qui servira à informer la couche présentation de la modification de la grille.

Ensuite création d'un thread pour chaque case. Ces derniers ne sont pas des threads ordinaire mais des **Worker**.

```
for (int i = 0; i < currentField.getDepth(); i++) {
    for (int j = 0; j < currentField.getWidth(); j++) {
        new Thread(new Worker(i, j)).start();
    }
}
```

Voici la définition minimal de la classe **Worker** :

```
class Worker implements Runnable {
    private int i, j;
    private boolean isAlive;
}
```

Cette classe représente une case et donc un thread, chaque thread est alors caractérisé par une position dans la grille, ainsi que l'état à cette position. Cela permet d'avoir un thread contenant un minimum d'information. Ce qui les rend plus léger et un plus grand nombre peut être créé.

Maintenant que les threads sont créés, il faut calculer leur prochaine valeur, cela est fait par la méthode **nextStep()** :

```
// Calculate the next step of the simulation
public void nextStep() {
    // Bloque la barrière
    barrierStart.await();

    // Attend que le travail des workers soit fini
    barrierEnd.await();
}
```

La méthode **nextStep()** est appelée à chaque itération sur le nombre d'étape dans le main.

De ce fait les barrières sont bloquées et attendent que les **Workers** aient fini leur travail.

De leur coté les **Workers** attendent que les barrières s'ouvrent pour continuer leur travail, leur code est le suivant :

```
public void run() {
    while (true) {
        // Attend l'ordre de commencer
        barrierStart.await();

        // Effectue le calcul de la prochaine étape
        doWork();

        // attend la fin de tous le threads
        barrierEnd.await();
    }
}
```

Ainsi tant que le nombre de **await()** n'est pas égale au nombre de leur initialisation, la barrière ne s'ouvre pas.

Pour ouvrir la barrière il est donc nécessaire d'effectuer :

Nombre de ligne * nombre de colonne + 1 = Nombre de case + 1

await()

Lorsque tous les **await()** sont fait, la barrière s'ouvre et le thread peut alors effectuer son **doWork()**.

Expérimentations

Méthode itérative

La méthode itérative ne semble souffrir de l'augmentation du nombre de case.

Son temps d'exécution s'accroît mais reste inférieure à 20 secondes (grille 1000x1000).

Nombre maximum de thread

Pour un ordinateur donné, et selon son état d'exécution, il ne peut créer qu'un nombre limité de thread.

Test 1

Création de 10 000 thread, ce qui représente une grille de 100x100.

La création des threads prend 19 secondes, mais la grille s'affiche finalement.

L'exécution prendra en moyenne 60 secondes pour 100 étapes.

Test 2

Essayons d'en créer 40 000.

Une erreur apparaît :

```
unable to create new native thread
```

Détail de l'erreur :

```
Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:640)
    at java.awt.EventQueue.initDispatchThread(EventQueue.java:878)
    at
java.awt.EventDispatchThread.run(EventDispatchThread.java:153)
```

On remarque qu'au maximum on peut en créer **33 050**, visualisé grâce à un compteur.

Résultats graphiques

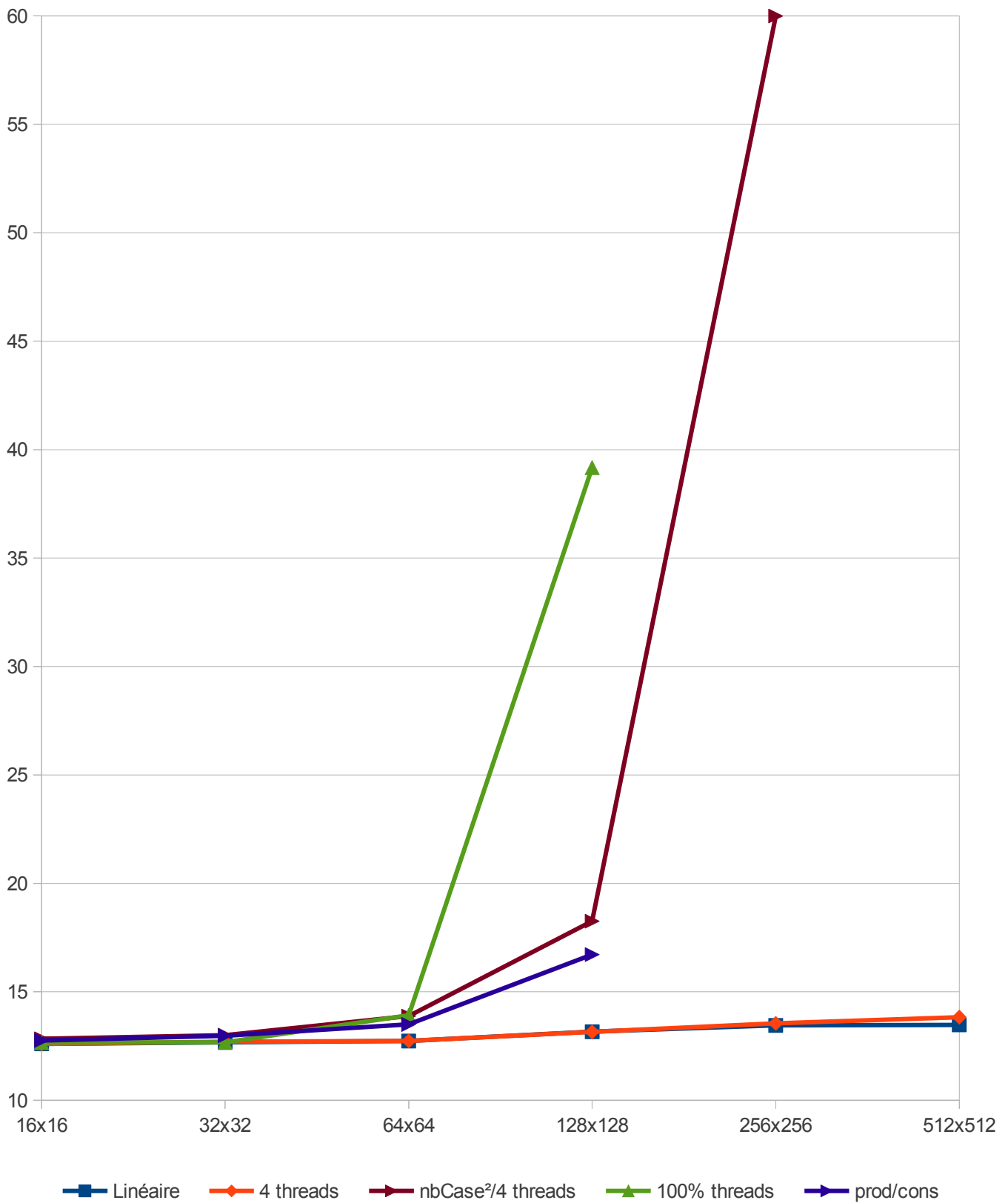


Illustration 2: Graphique de l'évolution du temps (en seconde) d'exécution en fonction de la méthode appliquée pour 50 étapes

Remarques

On remarque que l'algorithme linéaire et contenant 4 threads sont très proche en temps d'exécution, tandis que les algorithmes fortement threadés nécessite des temps beaucoup plus importants.

Modélisation FSP

Programmation

Les **Workers** sont représentés de la manière suivante :

Il attendent que la barrière soit ouverte pour faire les calculs et ensuite ils attendent que la barrière s'ouvre de nouveau pour continuer.

```
WORKER = (open -> doWork -> await -> WORKER).
```

La barrière est représentée de cette façon :

```
BARRIERE = (open -> BARRIERE[0]),  
BARRIERE[c:0..NBCase] = (  
    when (c < NBCase) await -> BARRIERE[c+1]  
    | when (c == NBCase) open -> BARRIERE[0]).
```

Une barrière contient un nombre de case, lorsque le nombre de case courante est inférieur au nombre de case, la barrière attend. Sinon elle s'ouvre.

L'ensemble du processus de jeux est représenté par ces lignes :

```
|| JEUX = (BARRIERE || [p:0..NBThread-1]:WORKER)  
    /{open/[k:0..NBThread-1].open,  
    [l:0..NBThread-1].await/await}.
```

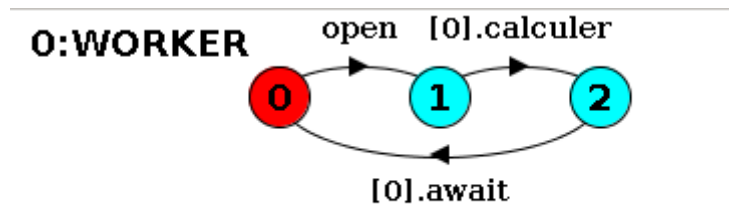
On crée **NBThread** Workers qui sont des threads en parallèle.

Les deux dernières lignes sont très importantes et elles permettent de mettre en relation les **Workers** et la **Barrière**, en effet le **open** de la barrière est renommé en **[0].open** , ... , **[NBThread – 1].open**, de ce fait quand la barrière s'ouvrira, elle ouvrira les open de chacun des processus. De même pour les **await**.

Diagrammes

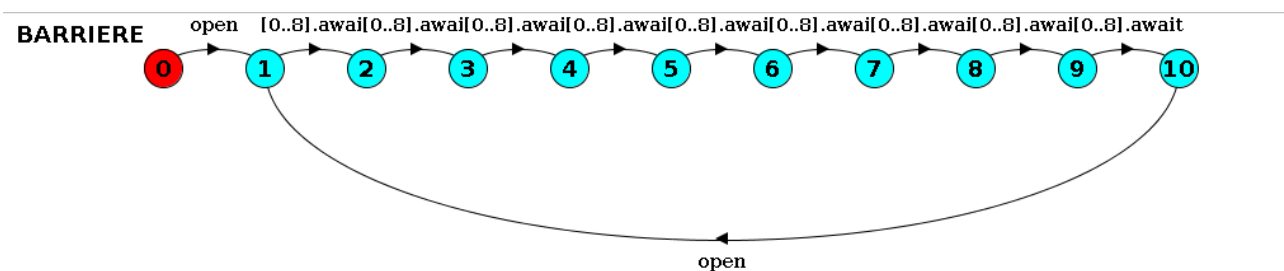
Workers

Voici le travail d'un **Worker** qui ne fait que calculer, mais il doit attendre l'ouverture de la barrière.



Barrière

La barrière effectue une attente constante que tous les **Workers** aient fini leur **calcul**.



Résultats

Safety

La propriété de sécurité est la suivante :

```
property OPEN = (open -> calculer -> await -> OPEN).
```

L'exécution correct du programme est donc de pouvoir ouvrir, calculer puis attendre le prochaine état.

Le **check/safety** permet de vérifier qu'il n'y a pas de **deadlocks**

```
No deadlocks/errors
```

Progress

La propriété progress est la suivante :

```
progress OUVRIE = {open}
```

Le résultat du **check/progress** est le suivant :

```
Progress Check...  
-- States: 10000 Transitions: 56620 Memory used: 82258K  
-- States: 19684 Transitions: 118100 Memory used: 93677K  
No progress violations detected.  
Progress Check in: 123ms
```

Aucune violation de la propriété **progress**, cela signifie que l'état **open** sera toujours accessible dans le futur.

Comparaison des méthodes

Méthode itérative

Pour ce genre d'application et pour un grand nombre de case, la méthode itérative est sans aucun doute la méthode la plus rapide.

Méthodes des voisins

Cette méthode est plus efficace pour les système reparties car elle nécessite peut d'information mais effectue de nombreuses communications.

Méthode de la barrière

Les barrières sont apparemment très lourd à créer.

Méthode écrivain-rédacteur

Pour aller plus loin...

Utilisation des `parallel.foreach`

Il s'agit d'un algorithme de découpage des tâches qui détermine le nombre de thread optimal en fonction des entrées et sorties effectuées et de la charge de calcul.

Cela permet de diminuer par moitié le temps de calcul pour un processeur dualCore sur des temps de calcul importants.

Utilisation :

```
// Utilisation du Parallel.For (avec une lambda expression)
private static void TestParallelFor ()
{
    int nbr = 0;
    Parallel.For(0, 150000, i =>
    {
        // Do something.
        nbr = i * 2;
    });
}
```

Source : <http://webman.developpez.com/articles/dotnet/programmationparallele/>