

# Jeu de la vie

Belli Fabien – Menage Emmanuel – Gorrieri Cyril – Lestel Guillaume



## Table des matières

Jeu de la vie.....	1
Introduction.....	4
Principe.....	4
Les phénomènes intéressants.....	6
Arrêt de la génération.....	6
Génération infinie.....	6
Apparition de formes.....	6
Algorithme linéaire.....	7
Principe.....	7
Inconvénient.....	7
Algorithmes avec threads.....	9
Un thread par case.....	9
Principe.....	9
Section critique.....	9
Fonctionnement.....	9
Utilisation des barrières.....	9
Représentation dans une architecture multi-processeur .....	10
Ressources utilisées.....	11
Algorithme avec un thread par case.....	11
Division de la zone de jeu en 4.....	12
Algorithme.....	12
Division de la zone de jeu en N.....	12
Algorithme des voisins.....	13
Introduction.....	13
Explication par l'exemple (grille 5x5).....	13
Algorithme producteur/consommateur.....	14
Principe.....	14
Avantages.....	14
Inconvénient.....	14
Algorithme producteur/consommateur avec 4 threads.....	14
Principe.....	14
Implémentations.....	15
Implémentation de la méthode itérative.....	15
Implémentation de la barrière pour un thread par case.....	16
Implémentation du producteur/consommateur.....	18
Implémentation.....	18
Expérimentations.....	19
Méthode itérative.....	19
Nombre maximum de thread.....	19
Test 1.....	19
Test 2.....	19
Résultats graphiques.....	20
Valeurs précises.....	22
Remarques.....	22
Modélisation FSP.....	23
Programmation.....	23

Diagrammes.....	24
Workers.....	24
Barrière.....	24
Processus .....	24
Résultats.....	25
Safety.....	25
Progress.....	25
Comparaison des méthodes.....	26
Méthode itérative.....	26
Méthodes des voisins.....	26
Méthode de la barrière.....	26
Méthode producteur/consommateur.....	26
Conclusion.....	27
Pour aller plus loin.....	28
Bibliographie.....	29
Principe.....	29
Articles.....	29
Jeux en ligne.....	29
Jeux téléchargeable.....	29

## Introduction

Le but de ce projet est d'appliquer des algorithmes de synchronisation entre processus au très connu Jeu de la vie.

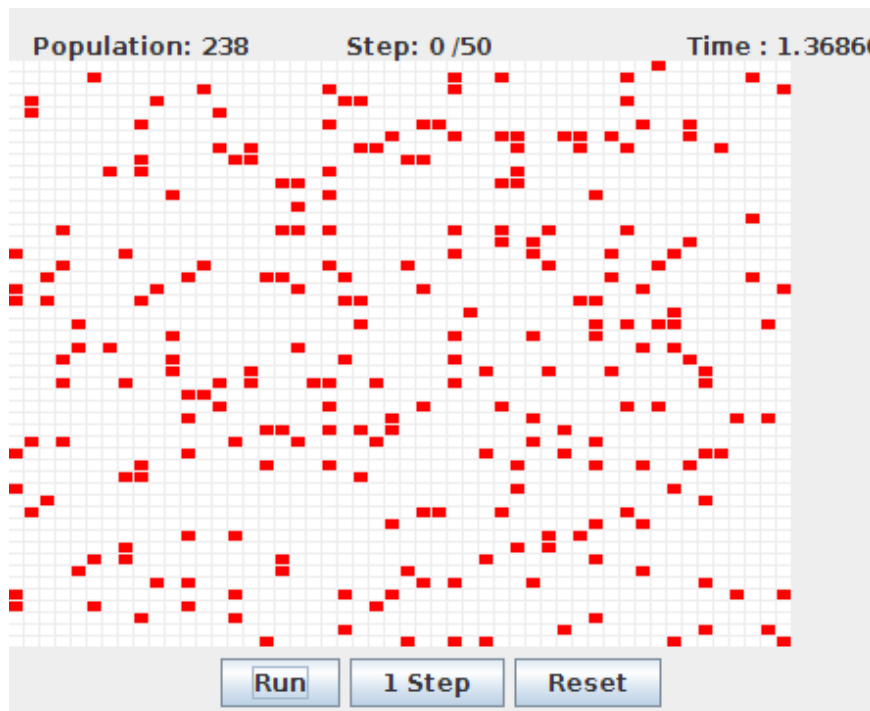
Le but étant de montrer que dans certaines conditions, un algorithme parallèle ne permet pas d'obtenir de meilleurs temps d'exécution qu'un algorithme traditionnel.

## Principe

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

Ce jeu contient un nombre fixé de case, chaque case est soit activée ou désactivée, soit en vie ou non.

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



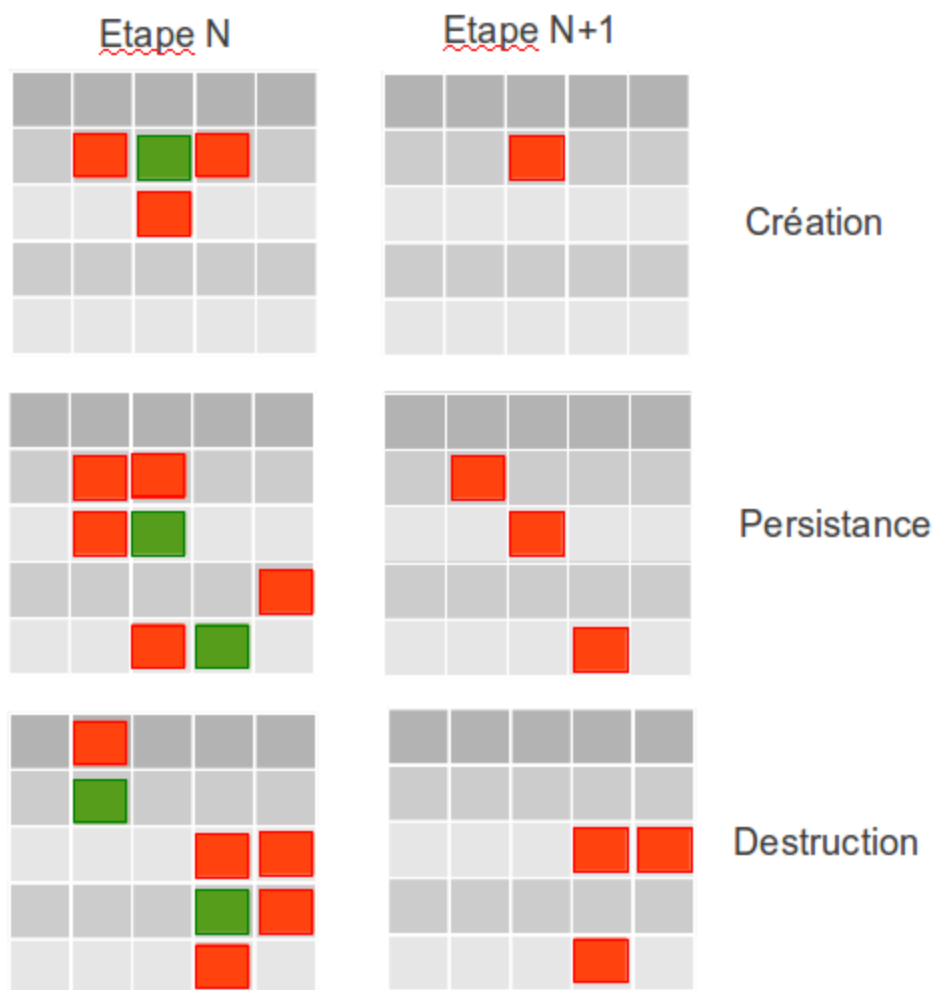
Cette grille peut être remplie au départ de manière aléatoire, ou par des cases choisies par l'utilisateur.

Le jeu n'a pas vraiment de but, mais consiste en une succession d'étape 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 voisin est de :

- 3 alors la case prend vie et devient active (rouge dans la capture d'écran).
- 2 et 3 quand 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.



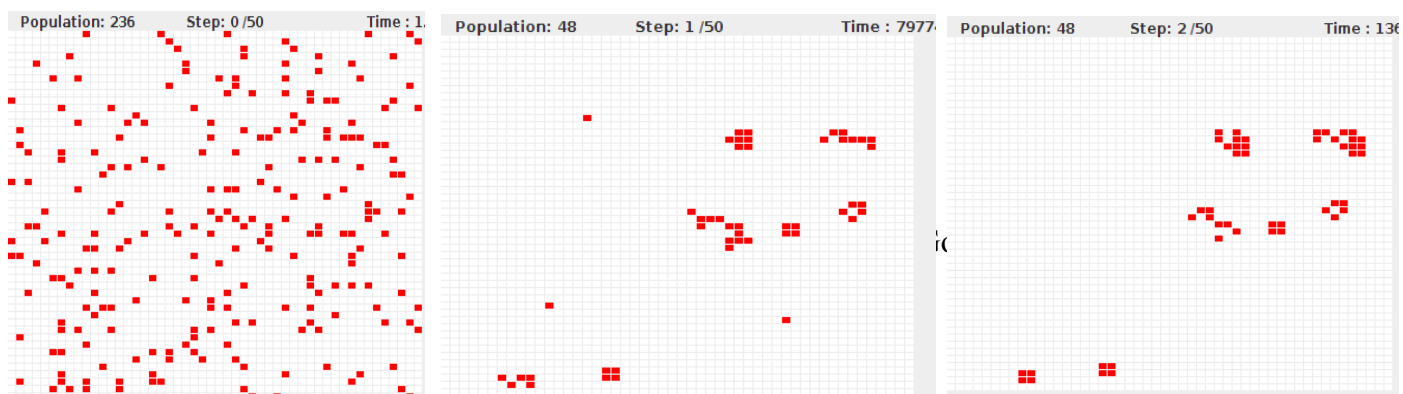
*Illustration 1: Schématisation du cycle du jeu de la vie*

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

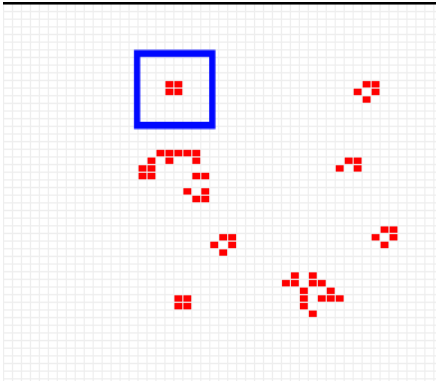
Exemple :



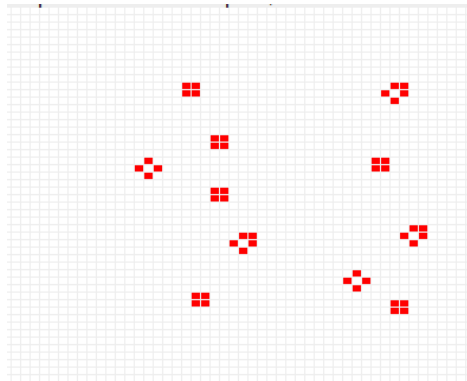
Ci dessus des captures d'écran montrant les 3 premières étapes du jeu de la vie.

## Les phénomènes intéressants

### Arrêt de la génération

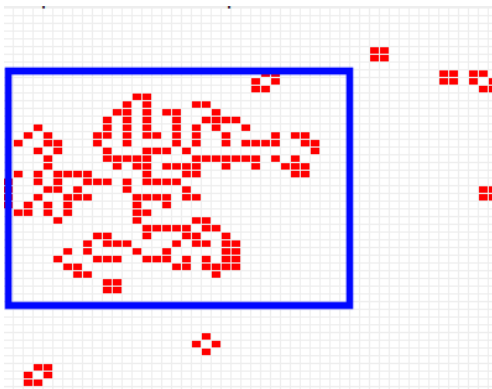


*Illustration 2: Capture d'écran d'un forme statique*



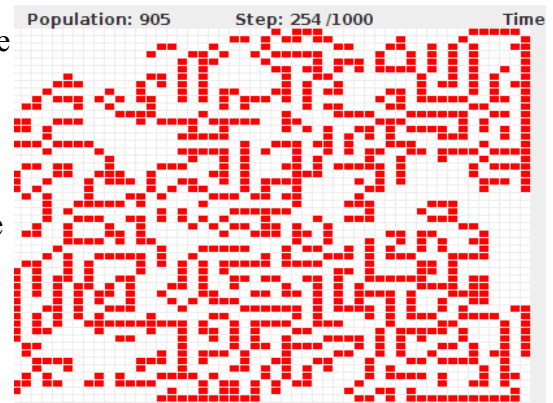
*Illustration 3: Capture d'écran de l'aspect général du jeu*

### Génération infinie



*Illustration 5: Capture d'écran d'une zone de génération infinie*

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



*Illustration 4: Capture d'écran d'une génération infinie*

### Apparition de formes

.

## Algorithme linéaire

### Principe

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

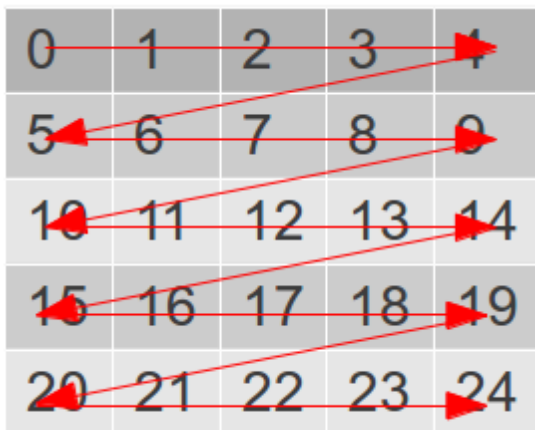
À chaque case analysée, les voisins sont comptés, puis la valeur calculée est sauvegardée dans une nouvelle grille.

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.

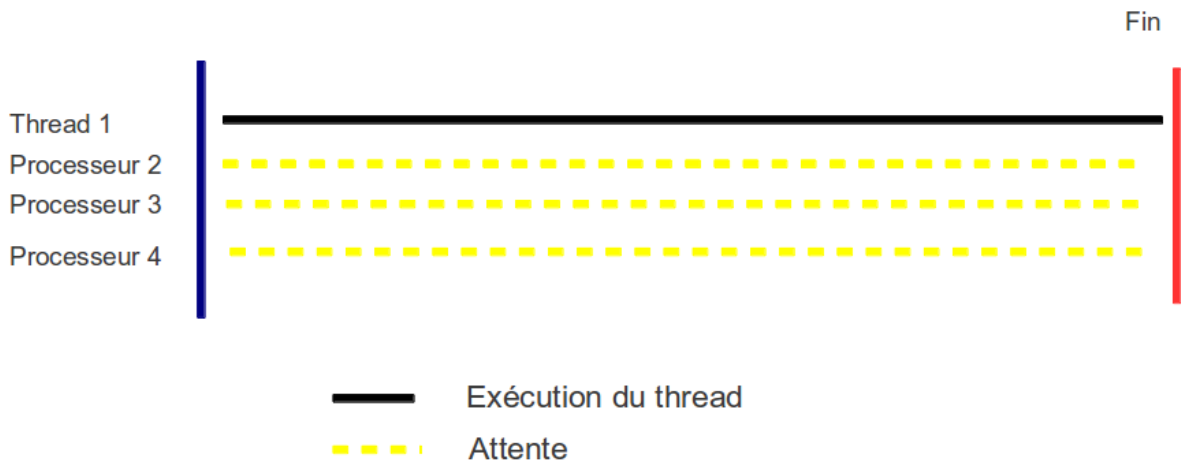
Dans cet exemple il est nécessaire de faire 30 calculs les uns à la suite des autres dans un seul thread pour chaque étape.



Le calcul des cases se fait séquentiellement.

### Inconvénient

Les calculs se font séquentiellement et sont effectués par le même processus.



*Illustration 6: Schéma représentant l'exécution d'un unique processus*

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 calculer 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, où chaque thread a sa case précise à écrire.

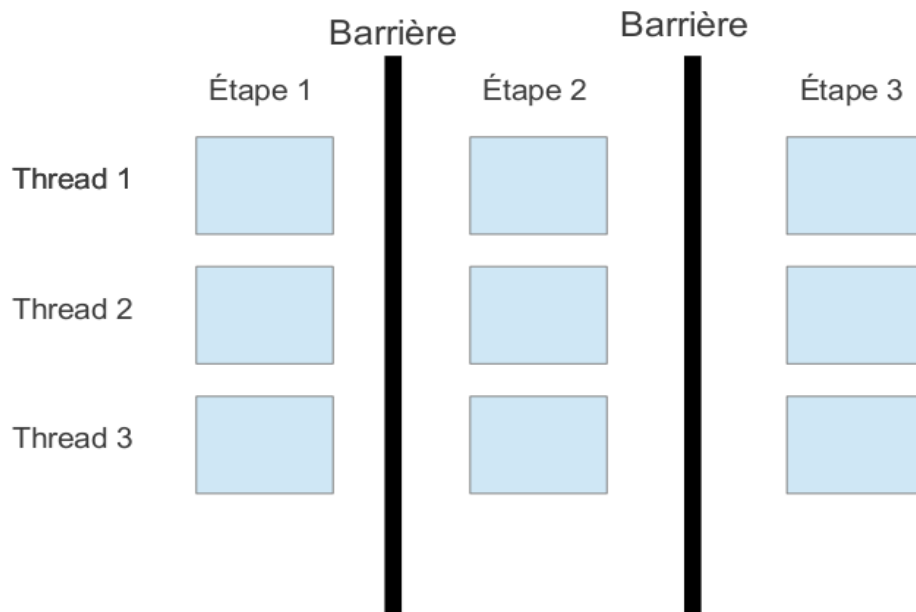
#### Fonctionnement

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) pourrait être calculé jusqu'à la dernière étape alors que ses cases voisines n'y serait pas encore.

Il est donc nécessaire de « faire attendre » le thread jusqu'à ce que tous les autres aient calculé 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.

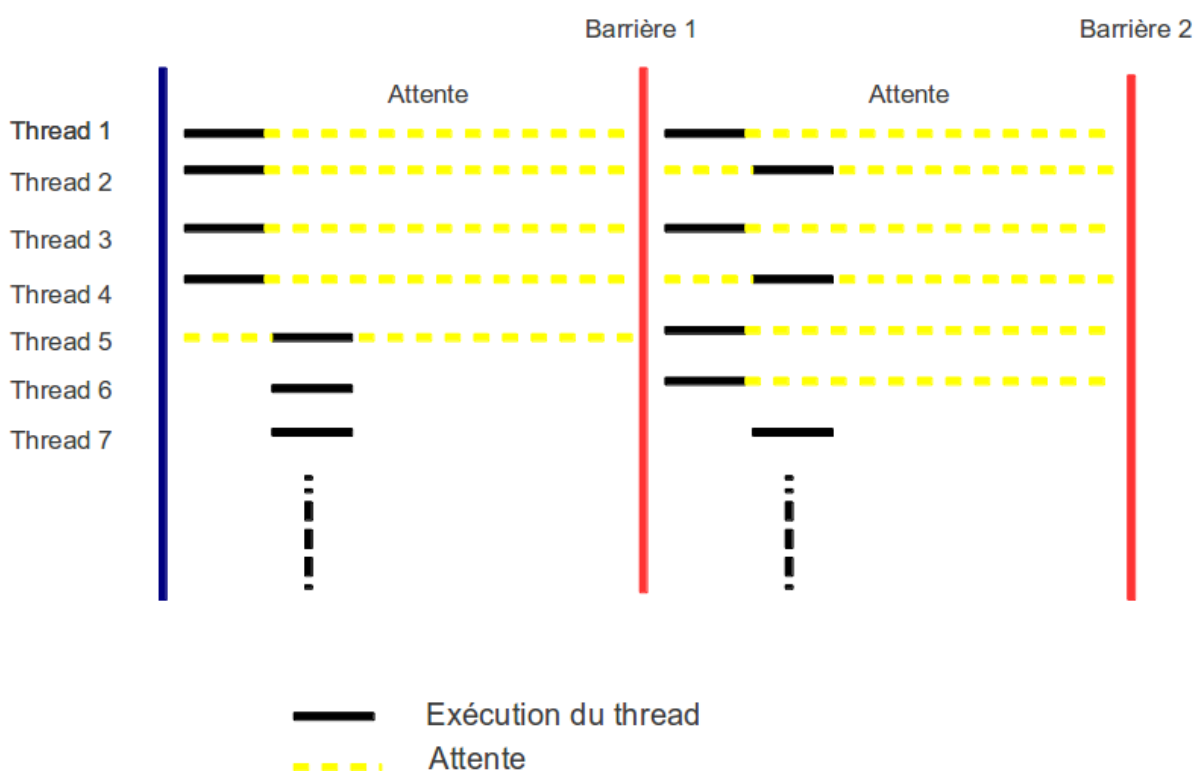


*Illustration 7: Schéma représentant le fonctionnement de la barrière*

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 terminé leur traitement.

## Représentation dans une architecture multi-processeur

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



ne

*Illustration 8: Schéma représentant l'application de la barrière sur plusieurs processus*

Mais il est possible que selon le système d'exploitation, ce système soit tout de même simulé, on obtient alors un pseudo-parralélisme.

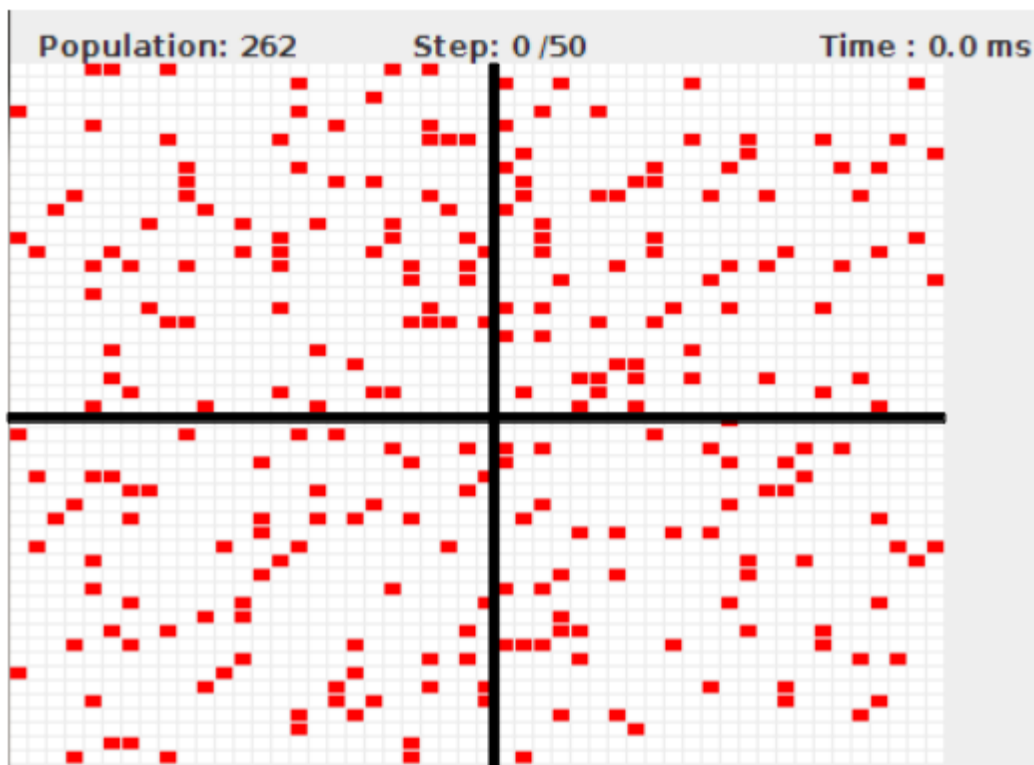
### ***Division de la zone de jeu en 4***

#### **Algorithme**

La zone de jeu 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 jeu.



*Illustration 9: Schéma représentant le découpage de la zone de jeu*

***Division de la zone de jeu en  $N$*** 

Il est possible de créer un nombre dynamique de thread qui s'occupera d'une section définie de la grille.

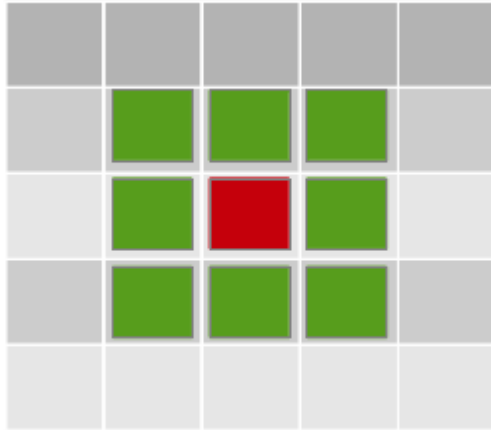
Ainsi si on décide de créer 4 threads, nous nous retrouverons alors dans le cas précédent.

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 voisins

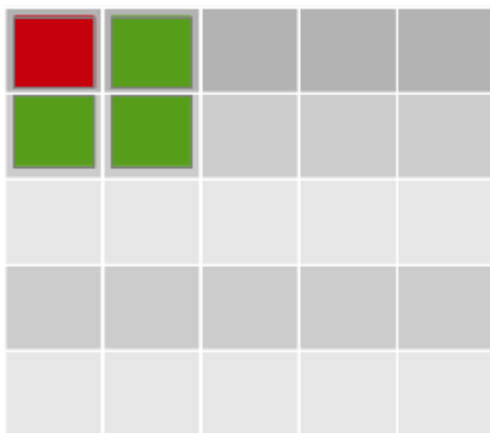
### Introduction



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 calculées. Chaque case est dirigée 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 connues et qu'il soit possible de dire que la valeur est valable.

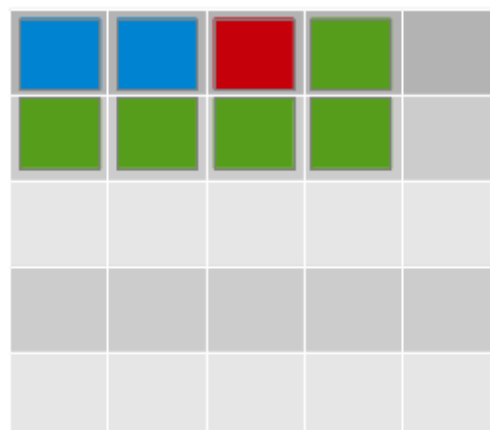
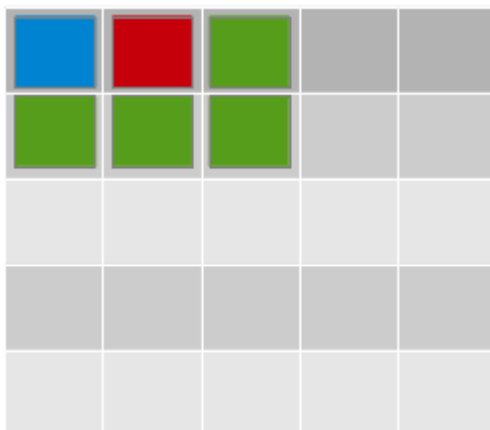
### Explication par l'exemple (grille 5x5)



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 connues. Pour calculer ces valeurs il faut que les valeurs de ces cases soient aussi connues.

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



## ***Algorithme producteur/consommateur***

### **Principe**

Le but étant de déléguer le calcul de chacune des cases à un thread, le processus principal se chargera d'agréger ses valeurs pour mettre à jour la nouvelle grille.

Le thread reçoit le **notify()** uniquement lors du changement d'étape.

Un **sémaphore** permet de gérer l'accès aux valeurs calculées dans les threads et ainsi attendre la fin du calcul des données.

### **Avantages**

Il n'y a pas de barrière, donc la gestion est moins lourde. Les thread s'occupent du calcul, le thread superviseur récupère uniquement les valeurs.

### **Inconvénient**

Pour la lecture séquentielle des valeurs, si la première case est la dernière à être calculée, cela pénalise l'intégralité du jeu.

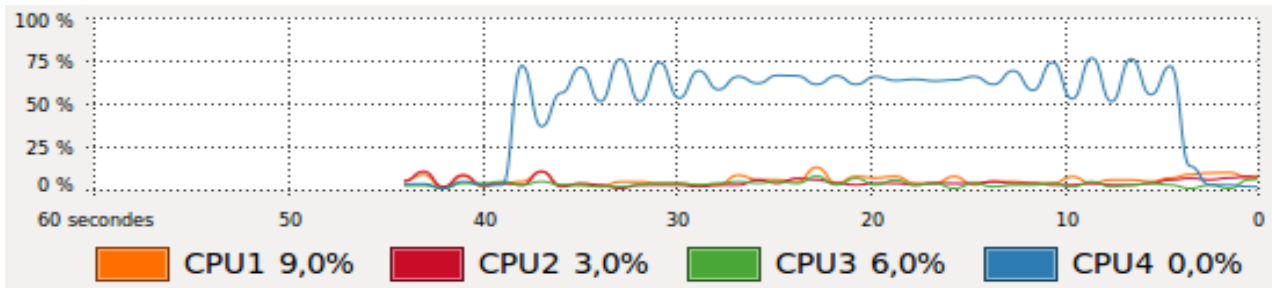
## ***Algorithme producteur/consommateur avec 4 threads***

### **Principe**

Utilisation de l'algorithme des producteurs/consommateurs dans le cas où un thread ne s'occupe plus d'une case mais de plusieurs.

## Ressources utilisées

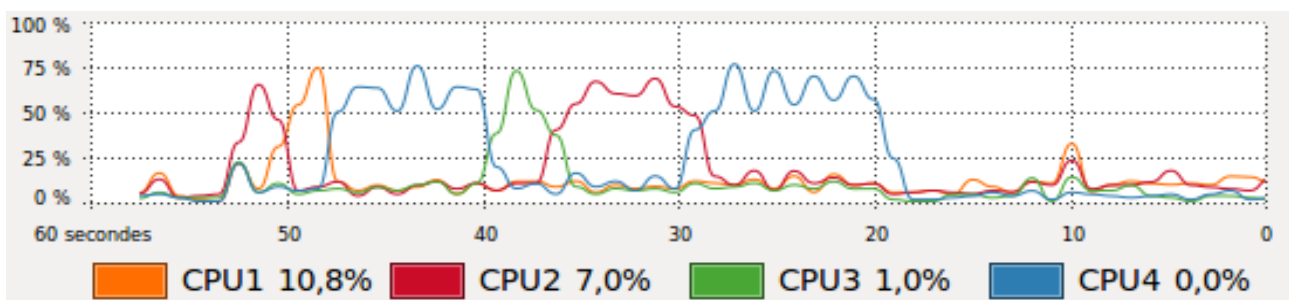
### Linéaire - Grille de 2048x2048



On remarque que l'algorithme linéaire utilise un unique processeur pour effectuer les calculs.

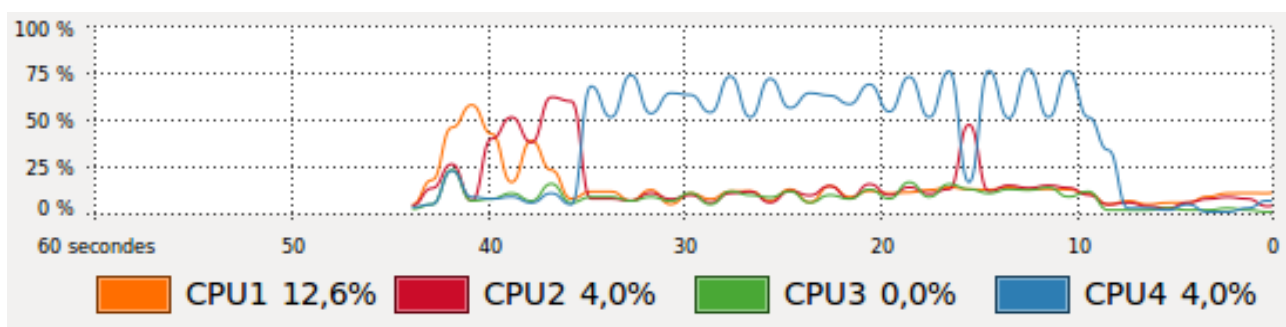
### Parallèle avec 4 processus

#### Grille de 128x128 – Algorithme avec barrière



On remarque que les processeurs sont utilisés à tour de rôle, mais jamais plusieurs à la fois

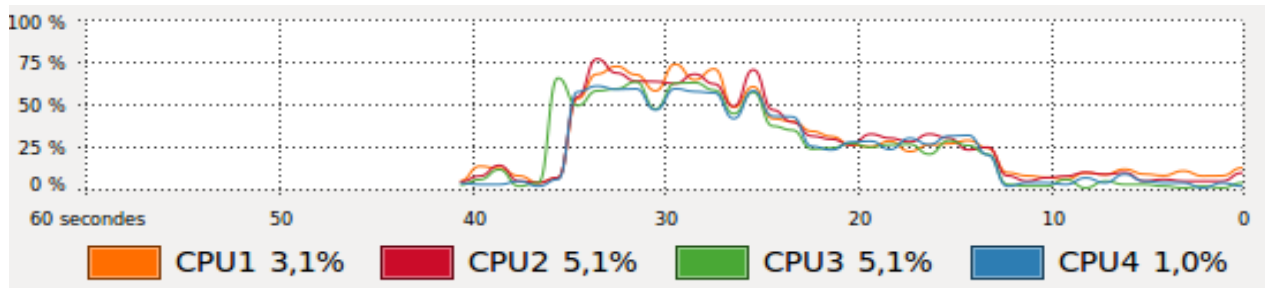
#### Grille 1024x1024 – Algorithme producteur/consommateur



Mêmes remarques que précédemment.

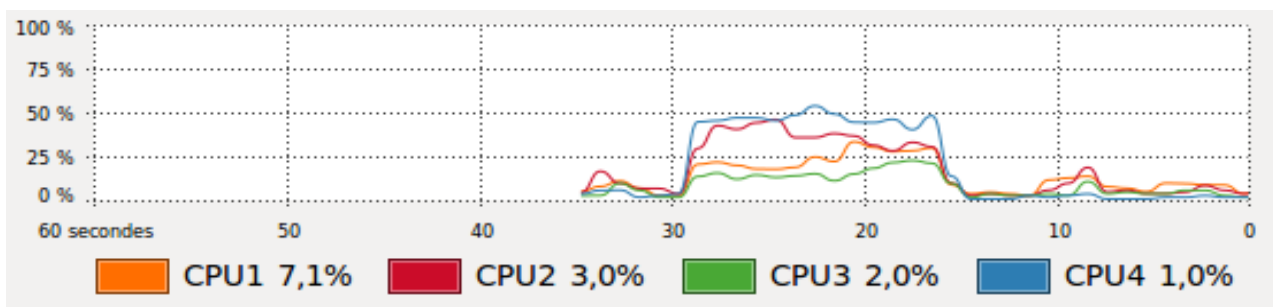
## ***Parallèle avec un processus par case***

### **Grille 128x128 – Algorithme avec barrière**



On remarque que les 4 processeurs sont utilisés pour répartir la grande quantité de thread créée.

### **Grille 128x128 – Algorithme producteur/consommateur**



On remarque que pour la même taille de grille, les processeurs sont moins utilisés que la version avec les barrières. Cela montre bien la lourdeur induite par l'utilisation des barrières.



## Implémentations

### *Implémentation de la méthode itérative*

Pour chaque étape, le calcul de l'état suivant se fait séquentiellement 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 actives  
            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 barrières pour permettre au premier thread de commencer. Ces barrières sont de type **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 minimale 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()** :

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

```
// 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();
}
```

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

De leur côté 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 égal 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 faits, la barrière s'ouvre et le thread peut alors effectuer son **doWork()**.

## ***Implémentation du producteur/consommateur***

### **Implémentation**

La méthode **run()** exécutée par un thread calcul la prochaine valeur et libère le sémaphore.

Le thread est alors mis en attente jusqu'à la lecture effective de la valeur.

```
public void run() {  
    while (true) {  
        value = ... ;  
        synchronized (this) {  
            ready.release();  
            this.wait();  
        }  
    }  
}
```

La méthode **readValue()** du thread permet de récupérer la valeur calculée par ce même thread à condition que le sémaphore permette d'obtenir cette valeur.

Une fois que la valeur est récupérée, le thread est libéré par **notify()** et peut calculer la prochaine valeur. Bien entendu la prochaine valeur est calculée uniquement au prochain tour.

```
public synchronized boolean readValue(Field nextField) {  
    ready.acquire();  
    notify();  
    return tmp;  
}
```

## Expérimentations

### ***Méthode itérative***

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

Son temps d'exécution s'accroît mais reste inférieur 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 threads, 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 :

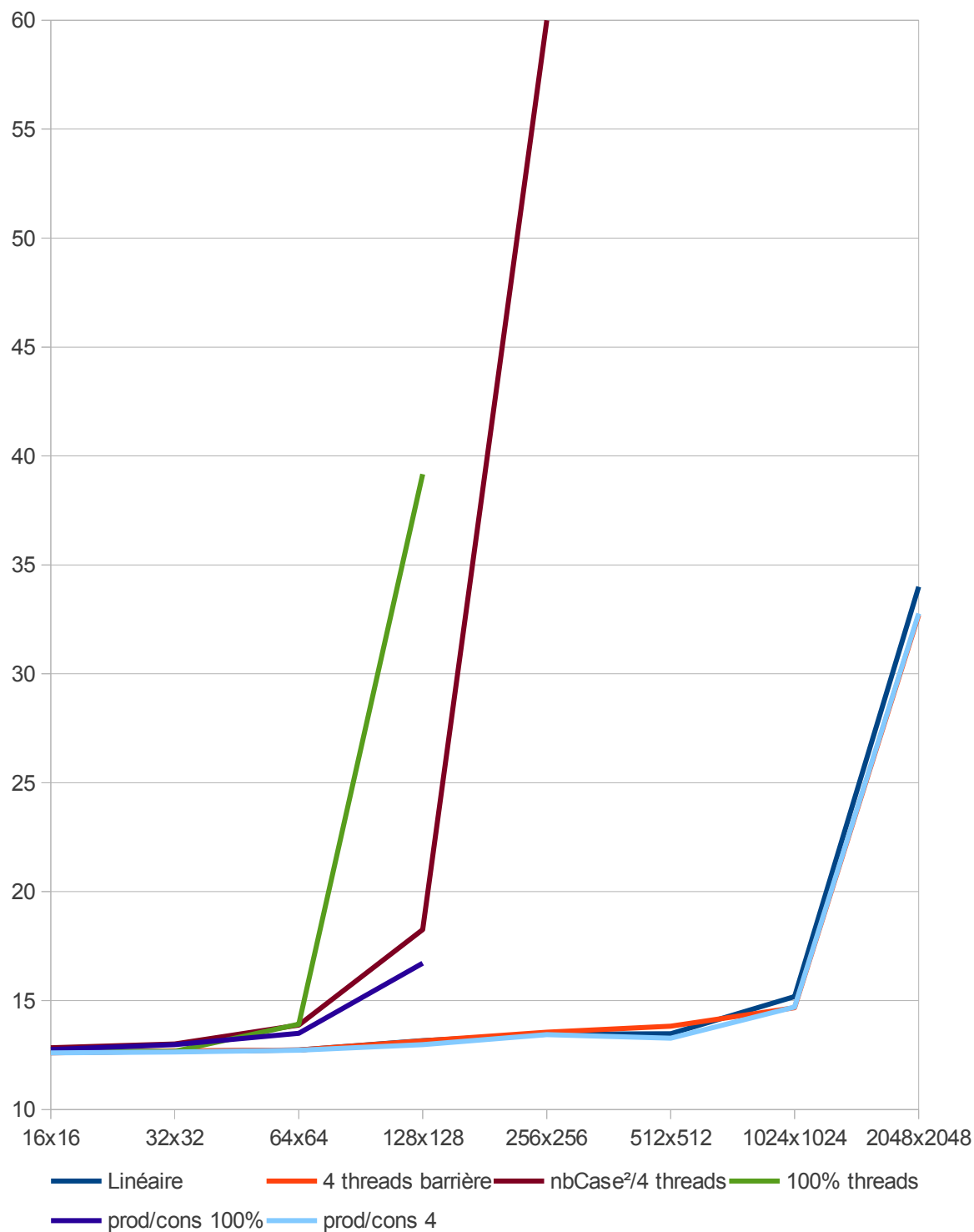
```
unable to create new native thread
```

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 que sur la machine de test, au maximum on peut en créer **33 050**, visualisé grâce à un compteur.

## Résultats graphiques



*Illustration 10: Graphique de l'évolution du temps (en seconde) d'exécution en fonction de la méthode appliquée pour 50 étapes*

## Valeurs précises







	 Linéaire	 4 threads b	 nbCase²/4 t	 100% threa	 prod/cons	 prod/cons
	Valeurs Y	Valeurs Y	Valeurs Y	Valeurs Y	Valeurs Y	Valeurs Y
1	12,6	12,59	12,83	12,66	12,75	12,59
2	12,67	12,69	13	12,67	12,98	12,65
3	12,73	12,73	13,88	13,92	13,5	12,72
4	13,16	13,15	18,26	39,17	16,72	12,97
5	13,45	13,55	59,99			13,43
6	13,48	13,83				13,27
7	15,18	14,67				14,71
8	33,99	32,7				32,766

Illustration 11: Valeurs utilisées pour la génération du graphique

## Remarques

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

Au bout d'un grand nombre de calculs, l'algorithme utilisant les 4 threads permet d'avoir des temps de calcul les plus performants.

## Modélisation FSP

### Programmation

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

Ils 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 courant est inférieur au nombre de case, la barrière attend. Sinon elle s'ouvre.

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

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

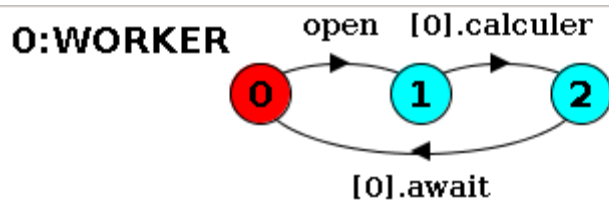
On crée **NBThread** Workers qui sont des threads en parallèle.

Les deux dernières lignes sont très importantes et elles permettent de mettre en relation les **Workers** et la **Barriere**, 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 chaque de 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.

Illustration 12: Automate d'un Worker

### Barrière

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

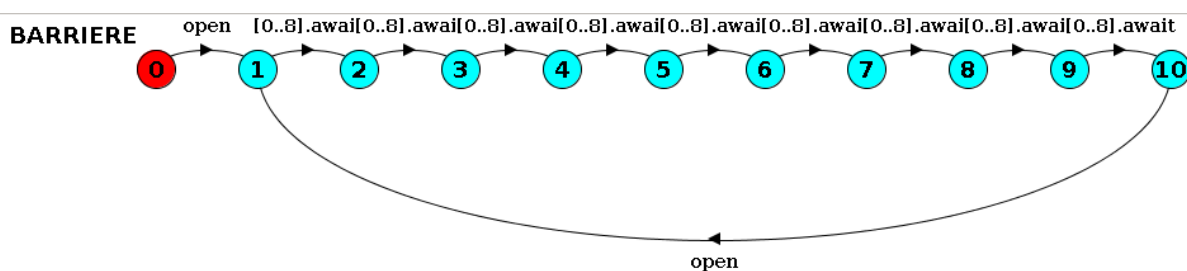


Illustration 13: Automate de la barrière

### Processus

BARRIERE	
0:WORKER	On remarque que pour un grille de taille 3 par 3, c'est-à-dire 9 cases, les 9 workers sont bien créés et agissent en parallèle.
1:WORKER	
2:WORKER	
3:WORKER	
4:WORKER	
5:WORKER	
6:WORKER	
7:WORKER	
8:WORKER	
JEUX	

Illustration 14: Liste des instances créées

## Résultats

### Safety

La propriété de sécurité est la suivante :

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

L'exécution correcte du programme est donc de pouvoir ouvrir, calculer puis attendre le prochain 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 OUVRIR = {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 systèmes repartis car elle nécessite peu d'information mais effectue de nombreuses communications.

### ***Méthode de la barrière***

Les barrières utilisent apparemment un mécanisme très lourd de gestion.

### ***Méthode producteur/consommateur***

La récupération des valeurs se fait par parcours séquentiel de la grille. Dans le cas où l'algorithme de calcul est très rapide, il est plus intéressant d'utiliser une méthode séquentielle. Néanmoins, cet algorithme est intéressant dans le cas où les threads effectuent des calculs de grande taille.

## Conclusion

On remarque qu'une barrière de grande taille nuit à la performance de l'exécution, cela est dû à un algorithme de synchronisation des barrières trop lourd, en tout cas dans le cadre du langage choisi.

Pour que les algorithmes parallèles devient plus performant il faudrait que les calculs qu'ils effectuent soient de plus grande taille, en effet la taille des calculs de nos processus se limite à un comptage du nombre de voisin en vie et l'identification du cas. Le calcul est alors plus rapide que la gestion des barrières.

## Pour aller plus loin...

Nous avons développé le projet en Java, mais d'autres langages offrent des mécanismes dans leur framework pour faciliter l'utilisation des mécanismes de programmation concurrente. Par exemple en C# .NET avec l'utilisation du **parallel.for**.

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 d'une itération de la boucle.

Cela permet dans le meilleur des cas 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;
    });
}
```

## **Bibliographie**

### ***Principe***

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

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

[http://fr.wikipedia.org/wiki/Jeu\\_de\\_la\\_vie](http://fr.wikipedia.org/wiki/Jeu_de_la_vie)

<http://www.dcode.fr/jeu-de-la-vie>

[http://t0m.free.fr/jdlv/jdlv\\_vivant.htm](http://t0m.free.fr/jdlv/jdlv_vivant.htm)

### ***Articles***

<http://www2.lifl.fr/~delahaye/dnalor/Jeudelavie.pdf>

### ***Jeux en ligne***

<http://www.dcode.fr/jeu-de-la-vie>

### ***Jeux téléchargeable***

[http://t0m.free.fr/jdlv/jdlv\\_logiciel.htm#Java](http://t0m.free.fr/jdlv/jdlv_logiciel.htm#Java)