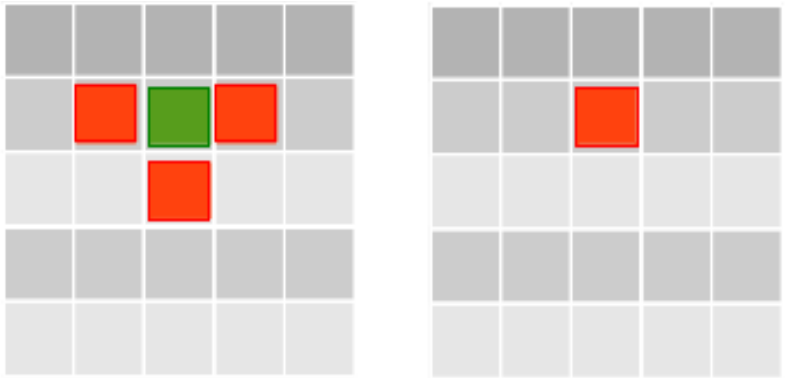
Jeux de la vie

Table des matières

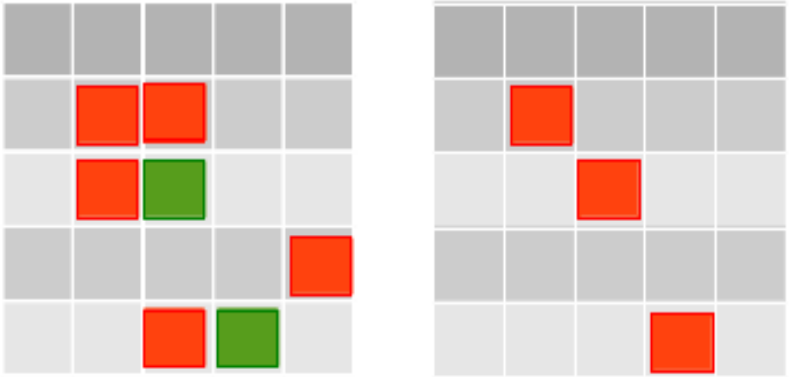
Jeux de la vie

- Grille de taille quelconque

- Calcul du nombre de voisin
  - 3 → création
  - 2 → reste vivante
  - > 3 → surpopulation
  - < 2 → population



Création



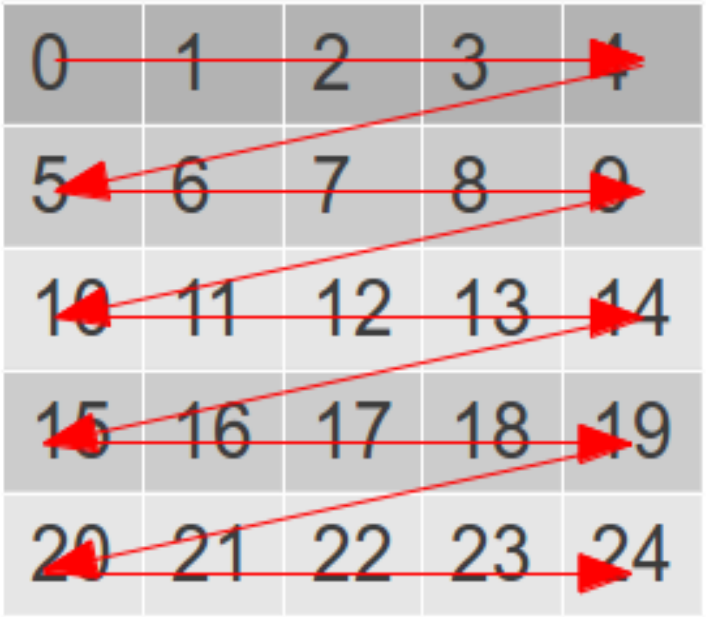
Persistance

Destruction

Algorithme linéaire

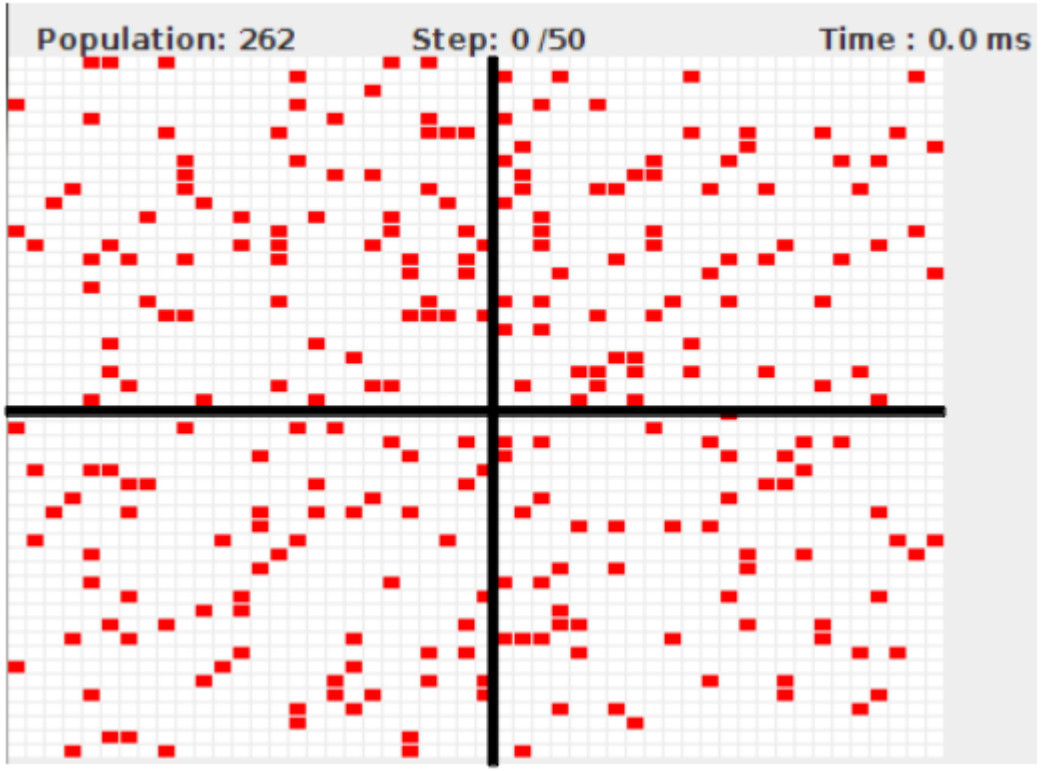
- Algorithme simple

- Efficace pour :
  - Un grand nombre de case
  - Des calculs simples

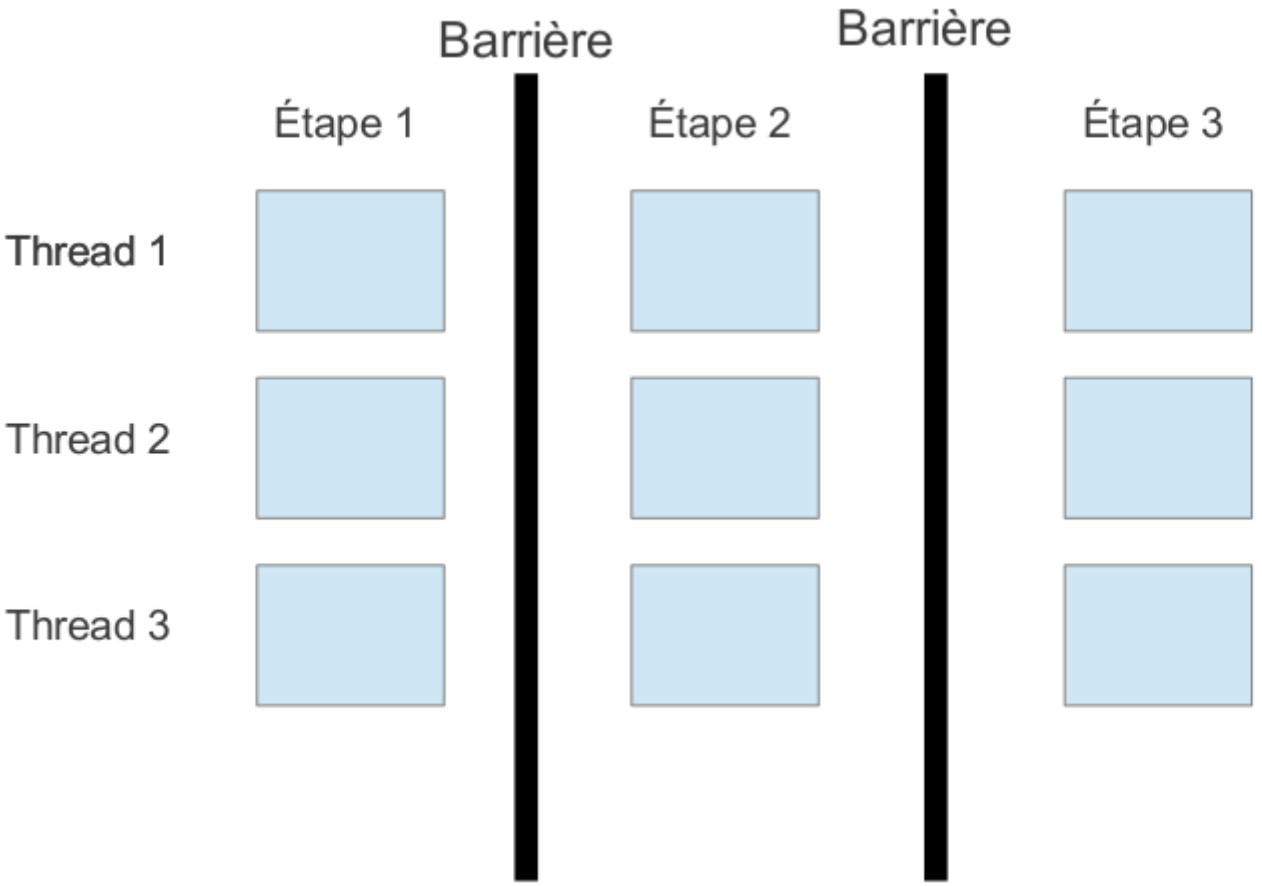


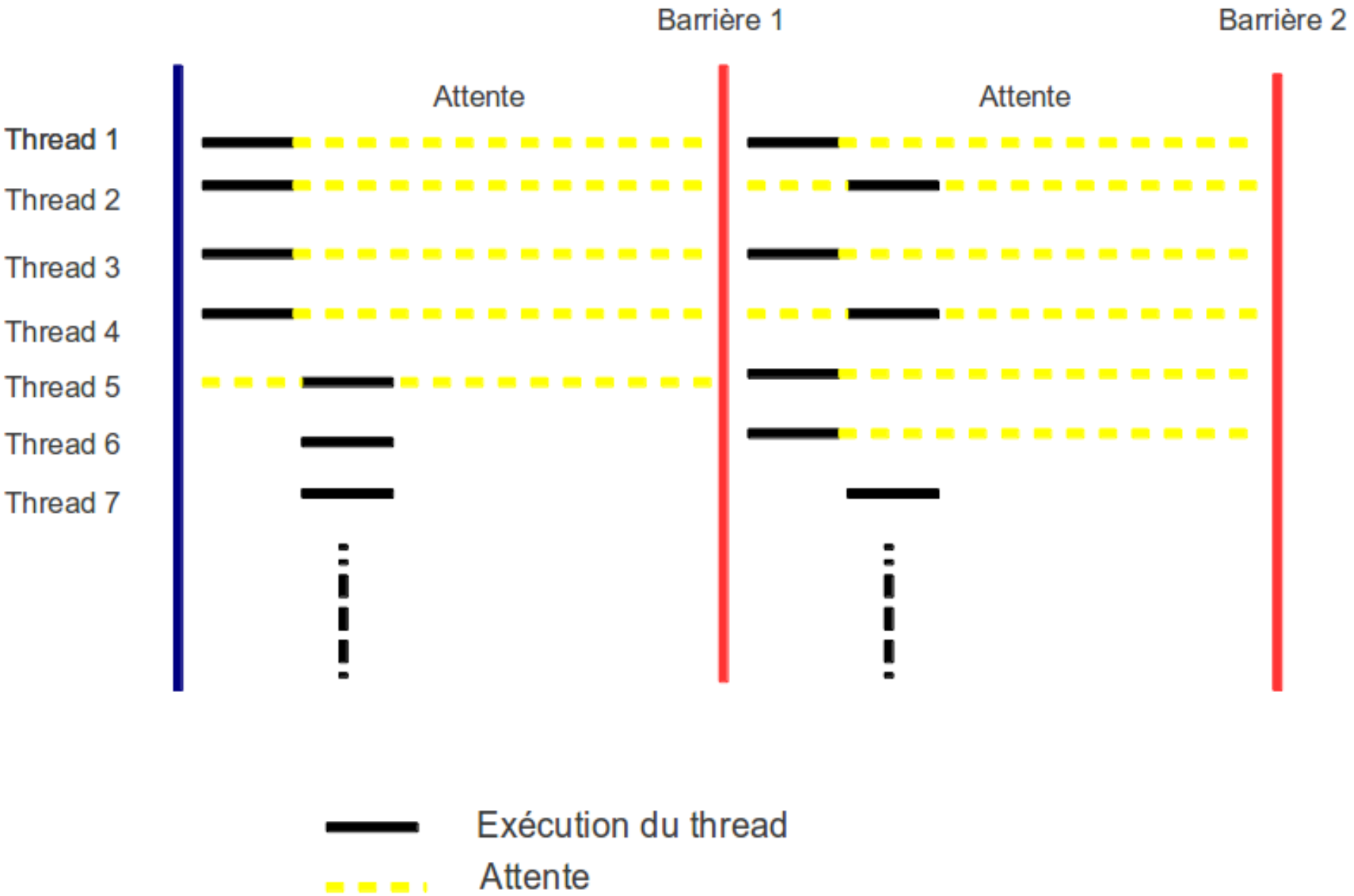
Algorithme parallèle

- Découpage de la zone de jeux en 4.
- Création de 4 threads
- Synchronisé avec des barrières



Les barrières

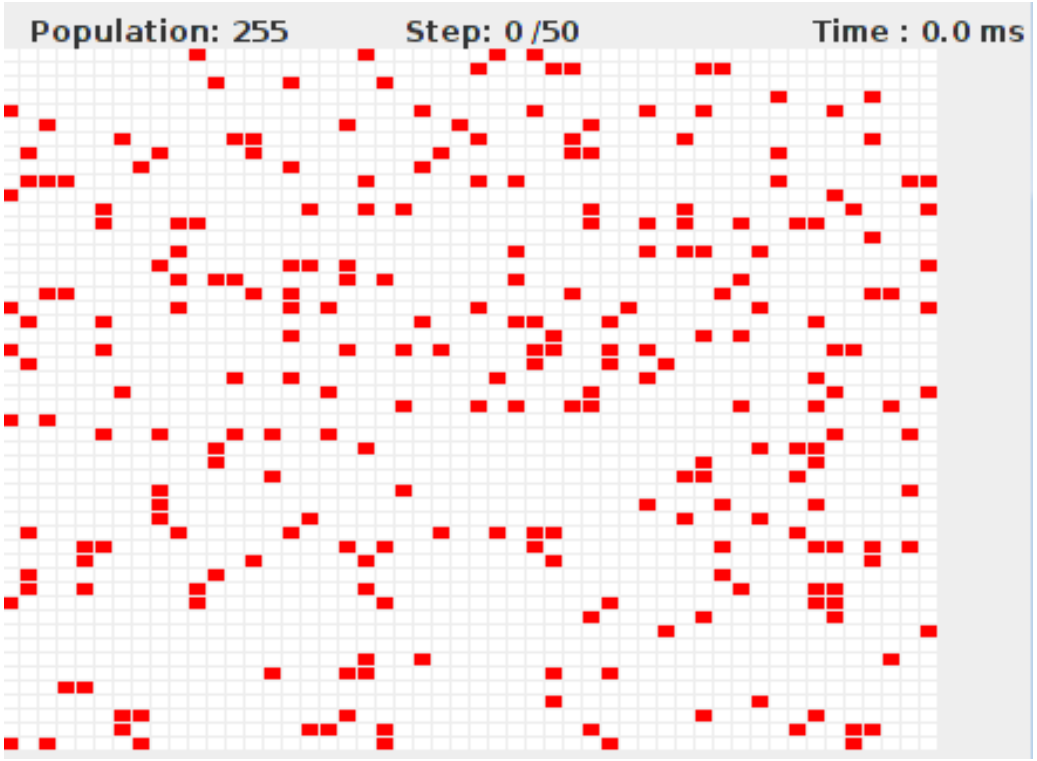



Barrière 1
Barrière 2
Attente
Attente
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Exécution du thread
Attente

Algorithme parallèle

- Un thread par case
- Synchronisé avec

  des barrières
- Initialisation au nombre de case + 1
- Nécessite **nb case + 1** → **await()** pour s'ouvrir



Remarques

- Avantage
  - Simple à utiliser

- Inconvénient
  - Temps de création des barrières très lent.
  - Processus de synchronisation pénalisant

Nombre maximum de thread

Test sur 40 000 cases

33 050 theads au maximum

```
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)
```

Barrière en FSP

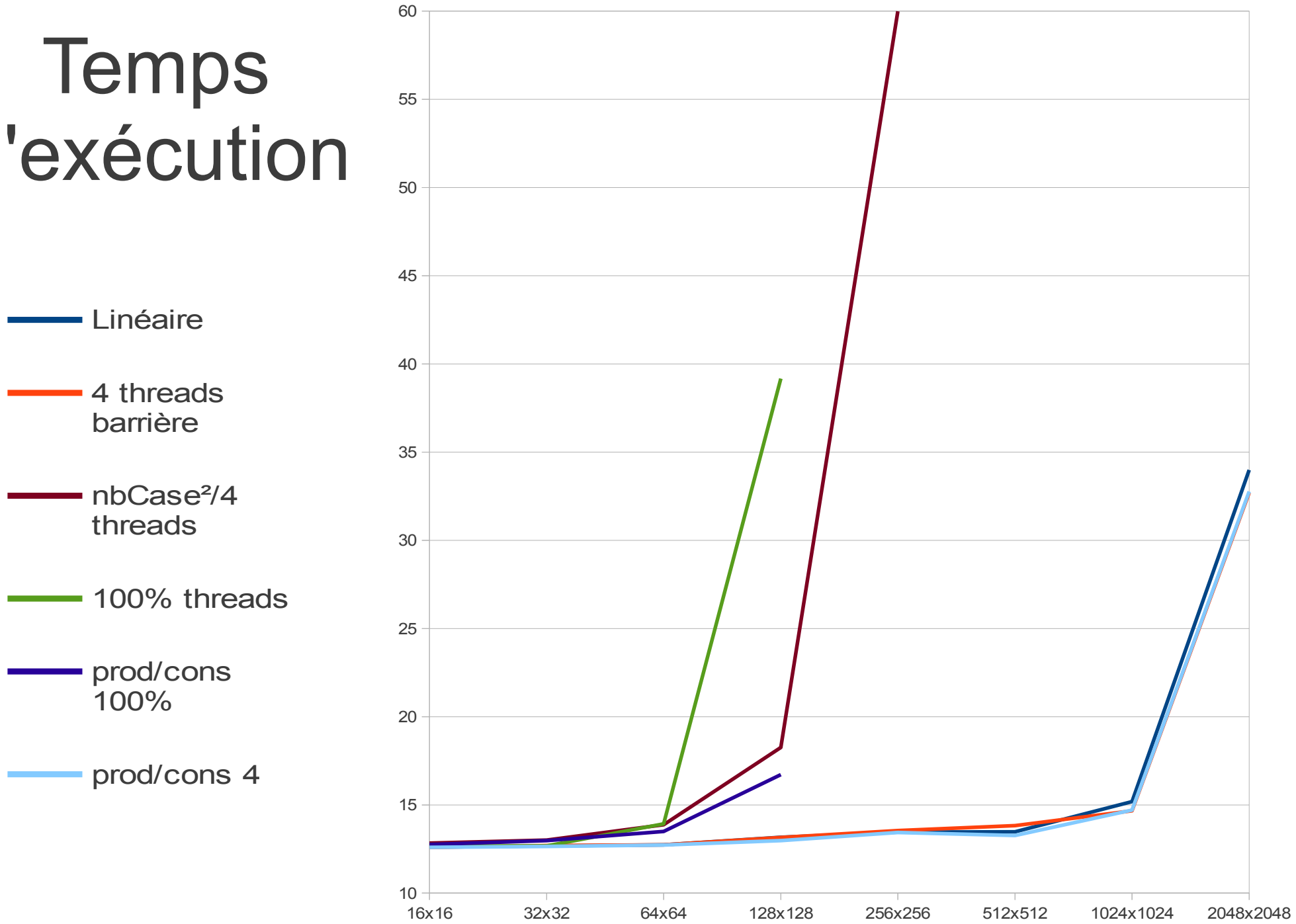
```
BARRIERE = (open -> BARRIERE[0]),
BARRIERE[c:0..NBCase] = (
                          when (c < NBCase) await -> BARRIERE[c+1]
                        | when (c == NBCase) open -> BARRIERE[0]).

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

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

No deadlock.
```

Producteur/consomateur

- Principe
  - Chaque producteur calcul une case
  - Un seul consommateur ; l'afficheur.
  - Moniteur pour la synchronisation

- Avantages
  - Synchronisation efficace

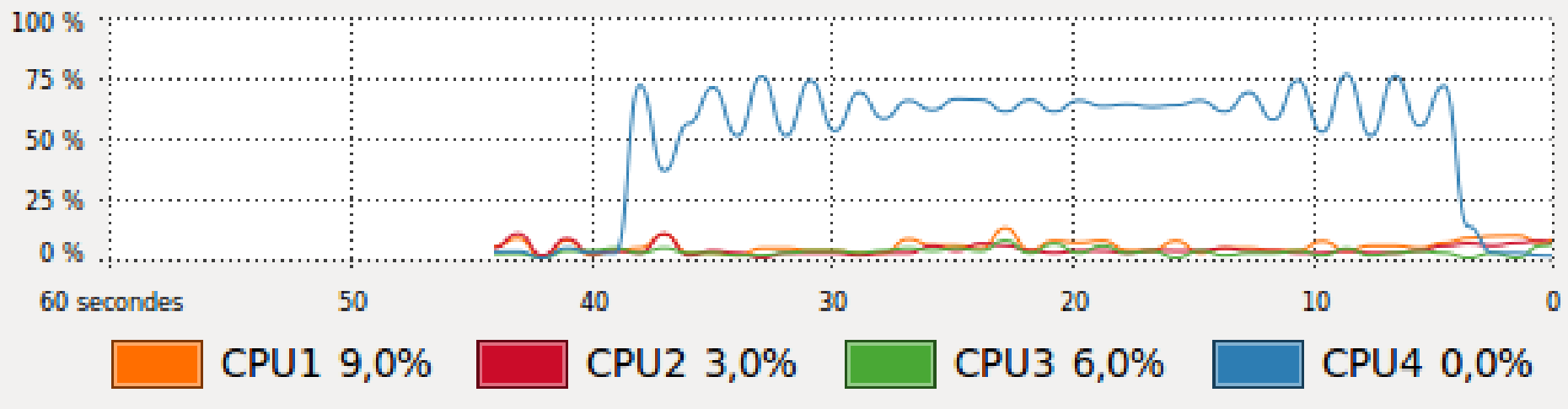
Temps d'exécution



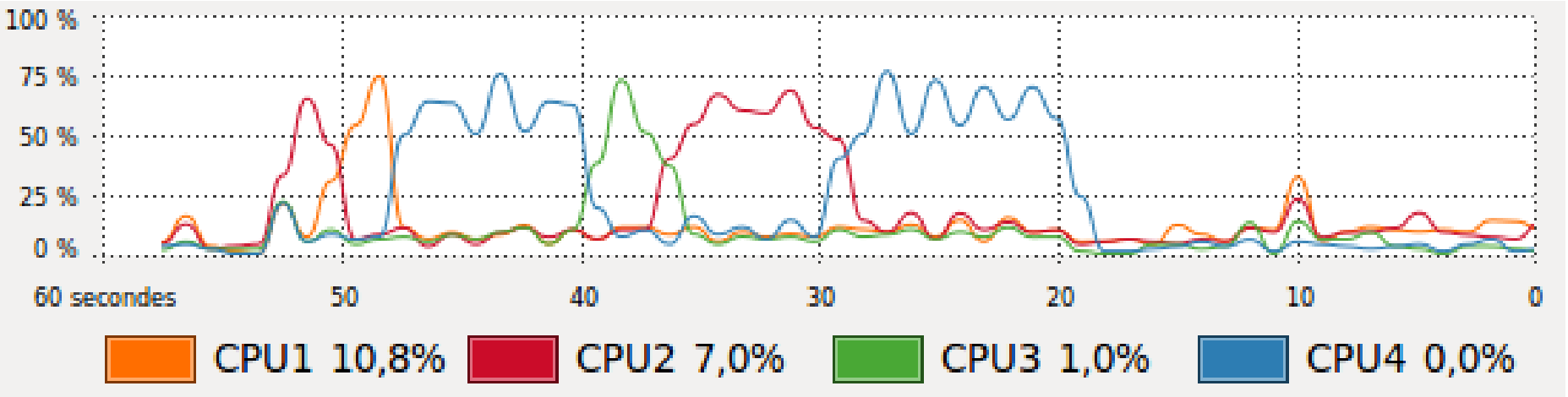
Allocation du processeur

Grille de 2048x2048

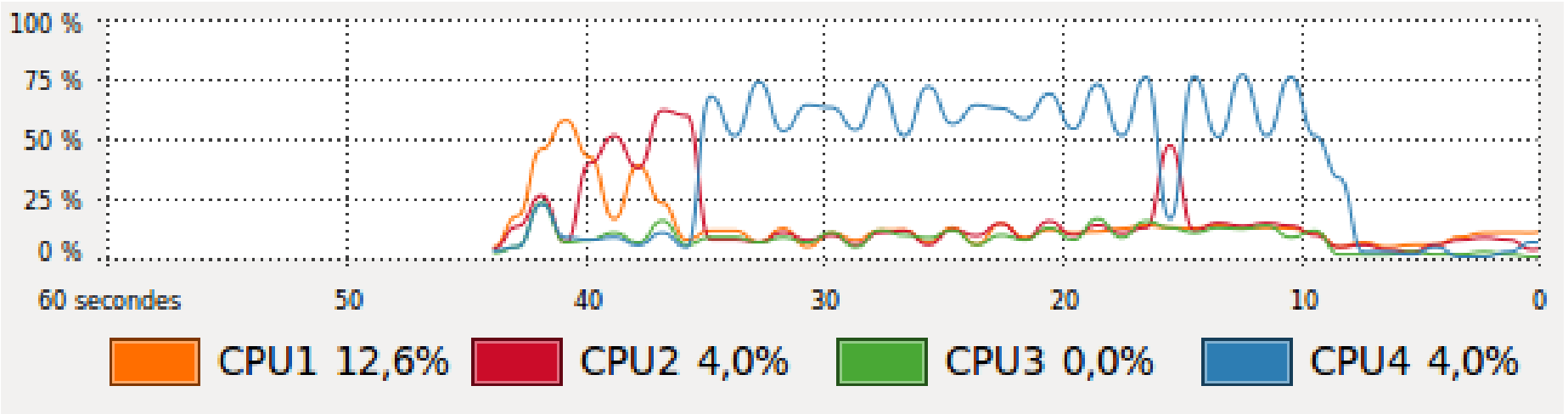
- Algorithme linéaire



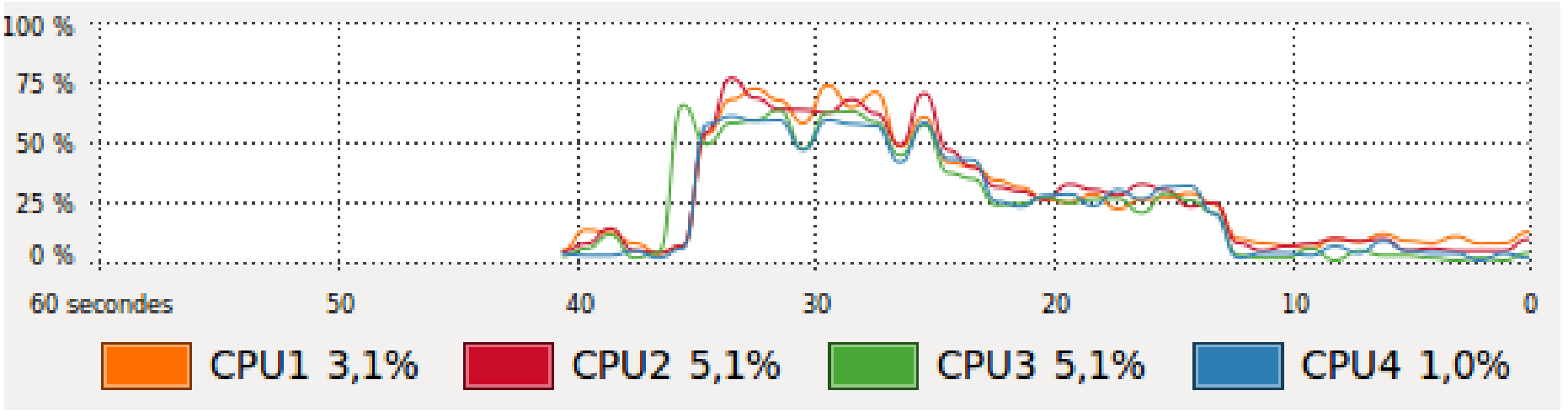
- Algorithme « semi threadé » avec Barriere



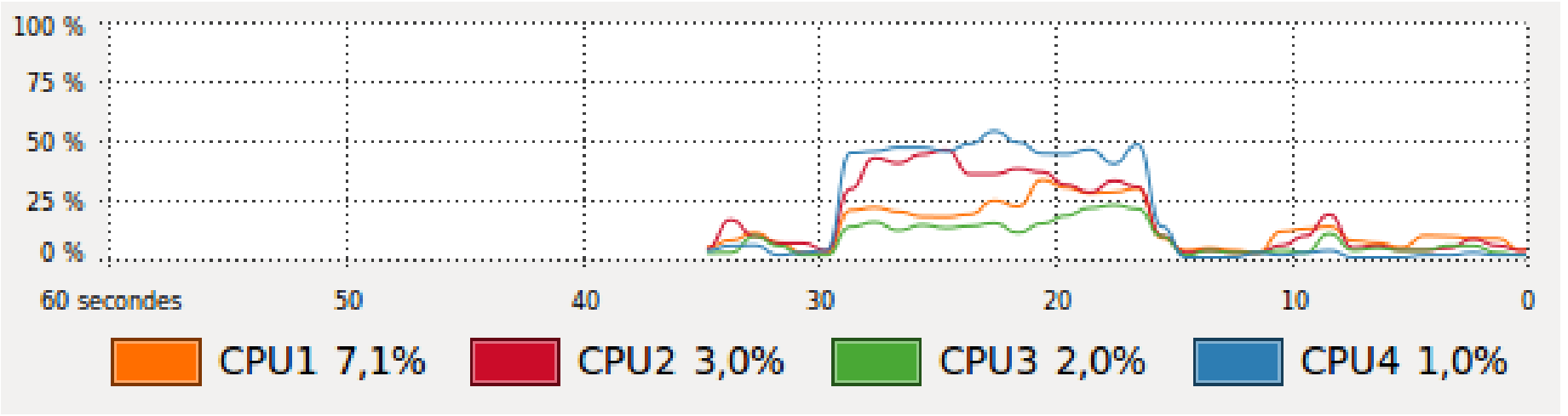
- Algorithme « semi threadé », avec P/C



- Algorithme «100 % threadé » avec Barriere



- Algorithme «100 % threadé », avec P/C



Démonstration

Conclusion

- Un grand nombre de thread n'améliore pas la rapidité des calculs
  - Temps de création important
  - Algorithme de synchronisation lourd
  - Algorithme de calcul léger

- Plus performant pour des « gros » calculs
- Lourdeur des barrières / sémaphore