ROLLIN Antoine
ra408576

09/03/2016
SI4

Programmation Concurrente

TP – Rendu n°3

Introduction

L'objectif de ce projet est de mettre en pratique la manipulation des threads et les principes de synchronisation vus en cours, en réalisant un simulateur qui modélise le phénomène de transfert de chaleur par conduction et l'évolution de la température en tout point d'une plaque chauffée.

Plusieurs versions[1] intégrées dans le même code seront successivement mises en œuvre :

- [0] Simulateur avec traitement itératif de la conduction
- [1] Simulateur avec traitement multi-tâches synchronisé par barrière Posix
- [2] Simulateur avec traitement multi-tâches synchronisé par barrière Posix et variables conditions
- *[3] Simulateur avec traitement multi-tâches synchronisé par barrière Posix et sémaphores*
- *[4] Visualisation du simulateur s'exécutant sur CPU*
- *[5] Visualisation du simulateur s'exécutant sur GPU*

Nous aborderons donc, dans ce document, tous les éléments nécessaires à la configuration et la mise en place des différentes versions, ainsi que leurs résultats en terme de performance.

[1] *Version* : les versions en italique ne sont pas encore implémentées.

Table des matières

1. Informations utiles

1.1. Configuration par défaut

Lancement du programme après compilation avec la configuration par défaut : « ./bin/project »

Si aucun argument n'est donné en ligne de commande, la configuration sera la suivante :

- taille de la matrice : 16x16
- les temps CPU ne sont pas affichés
- les temps utilisateur ne sont pas affichés
- les matrices initiales et finales ne sont pas affichées
- nombre d'itération par étape : 10000
- nombre de thread : 4
- toutes les étapes sont lancées

1.2. Constantes

Dans le fichier project.h, nous pouvons trouver les constantes qui permettent une partie de la configuration du projet :

- H : facteur de propagation pour la formule de Taylor. Nous choisirons un facteur valant 6 pour que chaque cellule de la matrice conserve 4/6 de sa valeur et propage 1/6 pour ses 2 cellules voisines à chaque traitement. Les cellules voisines sont les cellules haut et bas pour le traitement vertical, gauche et droite pour le traitement horizontal.
- TEMP_HOT : valeur de la température du centre de la plaque (par défaut : 128)
- TEMP_COLD : valeur de la température à l'extérieur de la plaque (par défaut : 0)
- REPEAT_NUMBER : nombre de répétition d'une configuration donnée, utile pour le calcul des temps d'exécution (par défaut : 10).

1.3. Gestion des bordures et du centre de la matrice

Les bordures de la matrice peuvent poser un problème lors des traitements verticaux et horizontaux de la conduction de la chaleur.

Par exemple, le traitement de la cellule située en (0,0) nécessiterait la connaissance de la valeur des cellules situées en (-1,0) et (0,-1).

Pour des raisons de simplicité, nous choisirons de simuler ces valeurs à 0. Si nous avions voulu se rapprocher de la réalité, nous aurions choisi une valeur qui correspond à la température ambiante, mais cela aurait eu pour conséquence une conduction supplémentaire de la chaleur par l'extérieur de la plaque.

De plus, la plaque doit être chauffée en permanence, ce qui implique que les valeurs du centre de la matrice doivent être remise à la valeur TEMP_HOT à chaque itération.

1.4. Gestion des configurations

Le programme permet la simulation de différentes versions pour différentes tailles de plaque et différent nombres de thread. Chaque exécution correspond donc à une configuration spécifique qui peut enfreindre certaines conditions fixées par le problème. C'est pourquoi il est nécessaire de vérifier avant l'exécution :

- vérifier que le facteur de taille (-s) de la matrice est compris entre 0 et 9
- vérifier que le facteur du nombre de thread (-t) est compris entre 0 et 5
- vérifier que le nombre de thread généré est inférieur ou égal au nombre de cellule de la matrice

2. Version Itérative (Étape 0)

2.1. Principe

Le principe de la version itérative est de mettre en place le processus de conduction de la plaque (matrice) sans parallélisme. Le principe de conduction se décompose en une suite consécutive de traitements verticaux et horizontaux. Lors de chaque traitement, la nouvelle valeur de chaque cellule est calculée grâce à la formule de Tayor (vue en TD).

Pour des raisons de précision, tous les calculs sont effectués en Float. Aussi l'ordre vertical-horizontal n'influe pas sur la cohérence des données finales (nous aurions pu choisir horizontal-vertical).

2.2. Algorithme

```
INPUT : N (nombre d'itérations), M1 (la matrice principale déjà chauffée), M2 (la matrice secondaire)

Main(N,M1,M2) :

   Pour i allant de 1 à N faire
      Traiter_Verticalement(M1, M2)
      Traiter_Horizontalement(M2, M1)
      Chauffer(M1)
   Fin pour
```

```
INPUT : M1 (la matrice source), M2 (la matrice destination)

Traiter_Verticalement(M1,M2) :

   Pour i allant de 0 à M1.longueur-1 faire
      Pour j allant de 0 à M1.largeur-1 faire
         cellule_cible = M1[i,j]
         voisin_haut = TEMP_FROID
         voisin_bas = TEMP_FROID
         Si j >= 1 alors
            voisin_haut = M1[i,j-1]
         Fin si
         Si j <= M1.largeur alors
            voisin_bas = M1[i,j+1]
         Fin si
         M2[i,j] =
(voisin_haut + (H-2)*cellule_cible + voisin_bas)/H
      Fin pour
   Fin pour
```

3. Version avec Barrière POSIX (Étape 1)

3.1. Principe

En dépit du fait que le traitement vertical et horizontal reste le même, la matrice est maintenant découpée en N régions (où N correspond au nombre de thread). Une région est un carré qui se définit par une coordonnée (x,y) et une longueur.

Chaque région est traitée par un thread. Les synchronisations des threads sont faites *via* une ou plusieurs barrières Posix pour assurer l'ordre des traitements et donc la cohérence des données.

Nous choisirons d'implémenter 2 barrières plutôt qu'une seule pour éviter une synchronisation supplémentaire (entre le processus principal et les threads après le traitement vertical).

L'utilisation de thread est destinée à exploiter toutes les ressources de notre processeur multi-cœurs, et donc à améliorer le temps d'exécution du programme. La section critique est représentée ici par la matrice.

3.2. Découpage de la matrice en régions

Le principe de la découpe consiste à créer une première région qui correspond à la matrice entière, puis à la découper récursivement en 4 jusqu'à atteindre le nombre de région souhaité.

```
INPUT : N (le nombre régions souhaité), S (la taille de la matrice)
OUTPUT : liste (la liste des régions)

Générer_Régions(N,S) :

        liste = Créer_liste_regions_vide()
        liste[0] = Créer_region(0, 0, S)
        nb_region = 1

        Tant que (nb_region < N) faire
                liste = Couper_Régions_En_4(liste)
                nb_region *= 4
        Fin tant que
```

3.3. Processus principal (main)

Le processus principal consiste à :

- Initialiser les barrières
- Lancer les threads pour chaque région
- Attendre la fin des threads
- Détruire les barrières

```
INPUT : N (le nombre de thread), M1 (la matrice principale déjà chauffée), M2 (la matrice secondaire), IT (le nombre d'itérations)

Main (N,M1,M2) :

	regions = Générer_Régions(N, M1.size)
	barrière_a = Init_Barrière(N)
	barrière_b = Init_Barrière(N)

	Pour i allant de 0 à N-1 faire
		Lancer_Thread(regions[i], IT, M1, M2)
	Fin pour

	Attendre_Threads()

	Détruire_Barrière(barrière_a)
	Détruire_Barrière(barrière_b)
```

3.4. Processus thread

Le processus thread consiste consécutivement à :

- Traiter verticalement la matrice principale
- Attendre la fin du traitement vertical de tous les threads
- Traiter horizontalement la nouvelle matrice
- Chauffer la matrice en fonction de la région
- Attendre la fin du traitement horizontal de tous les threads

```
INPUT : R (la région du thread), IT (le nombre d'itérations), M1 (la matrice principale), M2 (la matrice secondaire)

Code Thread (R,IT,M1,M2) :

	Pour i allant de 1 à IT faire
		Traiter_Verticalement(M1, M2)
		Barrière_Attendre(barrière_a)
		Traiter_Horizontalement(M2, M1)
		Chauffer_Matrice()
		Barrière_Attendre(barrière_b)
	Fin pour
```

4. Version avec Barrière POSIX reconstruite (Étape 2)

4.1. Principe

L'objectif de cette version est de reconstruire la barrière Posix utilisée précédemment à l'aide de variables conditions (pthread_cond) et de verrous (pthread_mutex).

Les fonctions qui nous intéressent ici sont barrière_init(), barrière_wait() et barrière_destroy().

Une structure Barrière est donc définie, contenant :

- Un compteur de processus en attente
- Une limite du nombre de processus en attente
- Un verrou
- Une variable condition

4.2. Initialisation de la barrière

L'initialisation de la barrière consiste à initialiser le verrou et la variable condition (avec les fonctions préexistantes), mettre à 0 le compteur de processus en attente et affecter la limite avec la valeur donnée en paramètre.

```
INPUT : N (le nombre limite de thread en attente avant libération)
OUTPUT : B (la barrière initialisée)

Barrière_Init (N) :

        B = Allouer_Mémoire()
        B.compteur = 0
        B.limite = N
        Initialiser_Mutex(B.mutex)
        Initialiser_Cond(B.cond)

        Retourner B
```

4.3. Attente et Libération *via* la barrière

L'attente est accessible par un seul processus à la fois grâce au verrou de la barrière. Lorsqu'un processus fait appel à Barrière_Wait, il incrémente le compteur de processus en attente, se bloque via la variable condition de la barrière si la limite n'est pas atteinte, débloque tous les processus en attente et remet le compteur à 0 sinon.

Note : la fonction *pthread_cond_wait()* utilisée pour l'attente d'un processus déverrouille le verrou passé en paramètre, ce qui permet aux autres threads de faire appel à la fonction, évitant ainsi un blocage.

```
INPUT : B (la barrière utilisée)

Barrière_Wait (B) :

	Bloquer(B.mutex)
	B.compteur += 1

	Si B.compteur == B.limite alors
		B.compteur = 0
		Libéreration()
	Sinon
		Attente()
	Fin si

	Débloquer(B.mutex)
```

4.4. Destruction de la barrière

La destruction de la barrière correspond simplement à la destruction de la variable condition, du verrou et à la restitution de l'espace utilisé par la barrière.

```
INPUT : B (la barrière utilisée)

Barrière_Destroy(B) :

	Détruire_Cond(B.cond)
	Détruire_Mutex(B.mutex)
	Rendre_Espace_Mémoire(B)
```

5. Résultats & Observations

5.1. Comparaison entre la version 0 et la version 1

5.1.1. Évolution des temps d'exécution en fonction du nombre de thread

Voici l'évolution des temps CPU de la version itérative et la version avec barrière Posix en fonction du nombre de thread, pour une matrice de taille [256x256] et 10000 itérations.

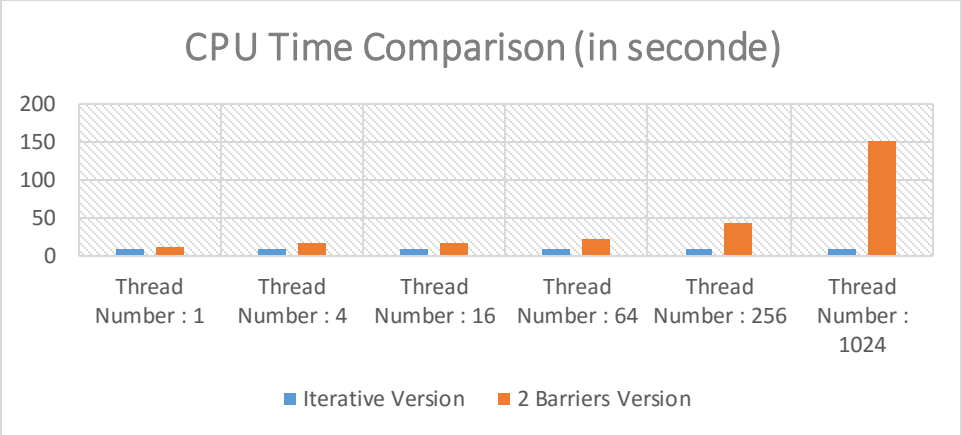

Fig.1 : Comparaison des temps CPU en fonction du nombre de thread (0-1)

Le temps CPU correspond à la somme des temps d'exécution de tous les processeurs. Nous pouvons constater que le programme exploite bien plusieurs processeurs puisque nous observons des temps croissants pour la version avec barrière.

Voici l'évolution des temps utilisateur de la version itérative et la version avec barrière Posix en fonction du nombre de thread, pour une matrice de taille [256x256] et 10000 itérations.

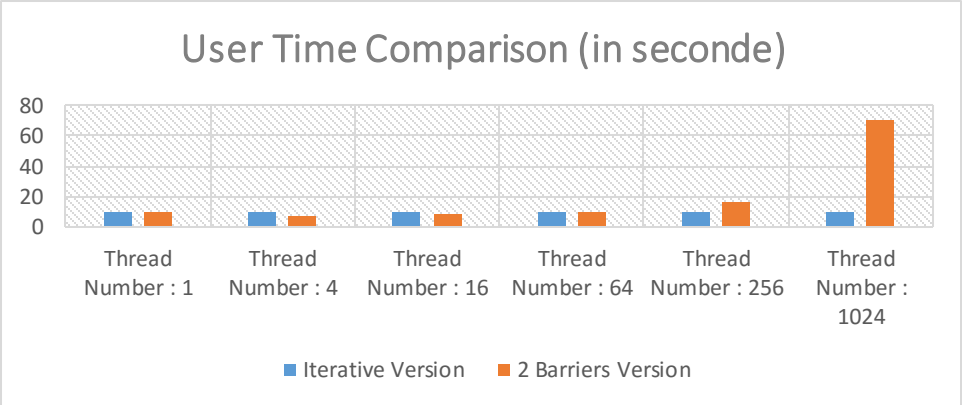

Fig.2 : Comparaison des temps utilisateur en fonction du nombre de thread (0-1)

Le temps utilisateur correspond au temps total d'exécution d'une étape. Nous pouvons observer ici que pour un nombre de thread inférieur ou égal à 64 travaillant sur une matrice de taille [256x256], le temps utilisateur de la version avec barrière est plus petit que celui de la version itérative.

Contrairement à ce que nous pourrions penser, la version threadée n'est donc pas toujours plus rapide que la version non-threadée.

Cela s'explique par le fait qu'en ajoutant des threads, nous complexifions le processus de synchronisation. Lorsqu'il y a N threads, les N threads doivent attendre que chacun ait fini le premier traitement (vertical ou horizontal) pour pouvoir continuer le traitement, ce qui génère une attente qui peut être conséquente quand le nombre de thread est trop grand. Tandis que dans la version itérative, le traitement suit un fil conducteur sans aucun travail de synchronisation et donc sans aucune attente.

Le nombre de thread influe donc sur les performances de temps de la version avec barrière.

5.1.2. Évolution de l'espace mémoire utilisé en fonction du nombre de thread

Voici maintenant l'évolution de l'empreinte mémoire en fonction du nombre de thread sur une matrice de taille [256x256] et 10000 itérations.

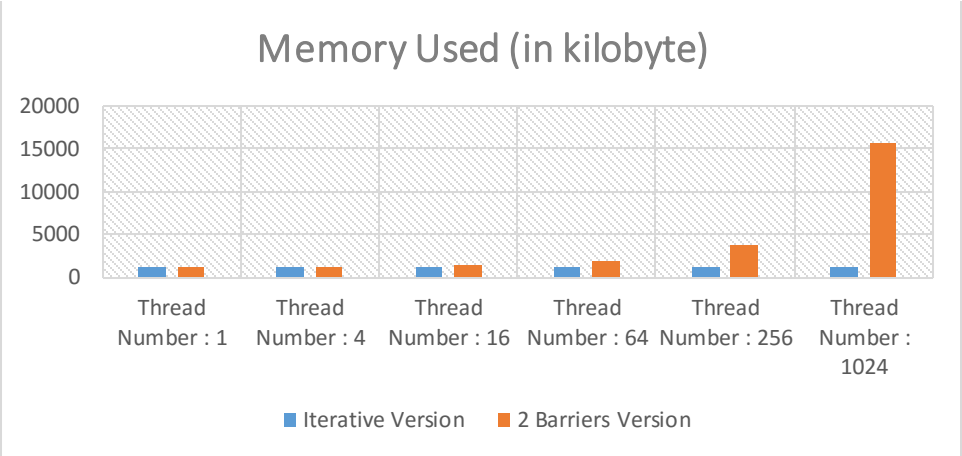

Fig.3 : Comparaison de l'empreinte mémoire en fonction du nombre de thread (0-1)

Nous pouvons constater ici que le nombre de thread est également un facteur de performance en terme d'empreinte mémoire.

5.2. Comparaison entre la version 1 et la version 2

5.2.1. Évolution des temps d'exécution en fonction du nombre de thread

Voici l'évolution du temps CPU en fonction du nombre de thread sur une matrice de taille [256x256] et 10000 itérations, pour la version avec barrière Posix et la version avec barrière reconstruite.

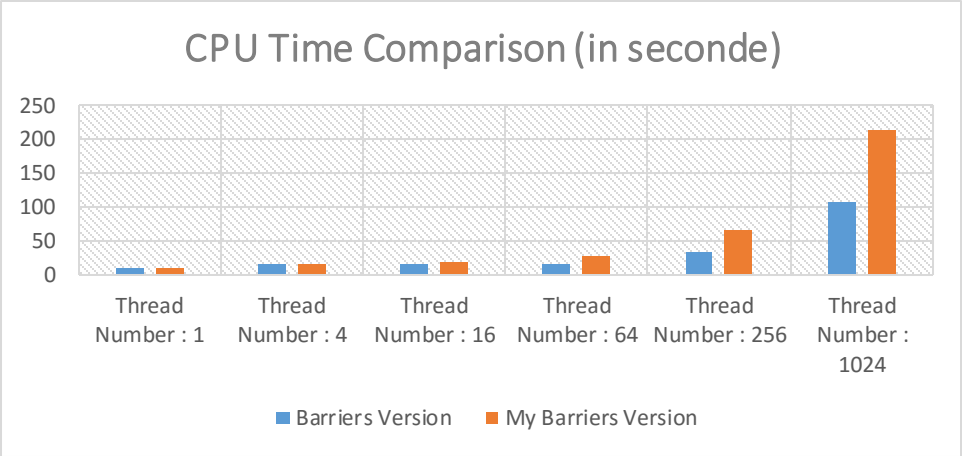

Fig.4 : Comparaison du temps CPU en fonction du nombre de thread (1-2)

Voici maintenant l'évolution du temps utilisateur en fonction du nombre de thread sur une matrice de taille [256x256] et 10000 itérations, pour la version avec barrière Posix et la version avec barrière reconstruite.

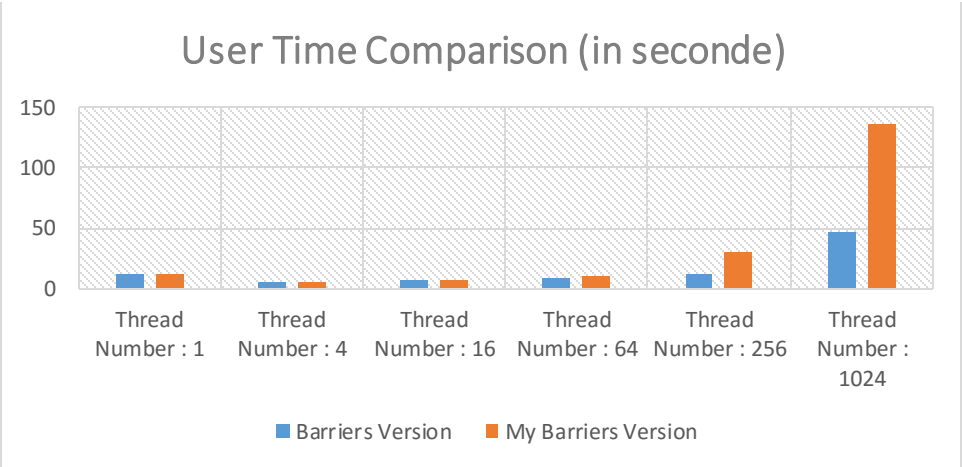

Fig.5 : Comparaison du temps utilisateur en fonction du nombre de thread (1-2)

Nous pouvons donc constater que la version avec barrière reconstruite est toujours plus lente que la version avec barrière Posix, tant pour les temps CPU que pour les temps utilisateurs.

5.2.2. *Évolution de l'espace mémoire utilisé en fonction du nombre de thread*

Voici l'évolution de l'empreinte mémoire en fonction du nombre de thread sur une matrice de taille [256x256] et 10000 itérations, pour la version avec barrière Posix et la version avec barrière reconstruite.

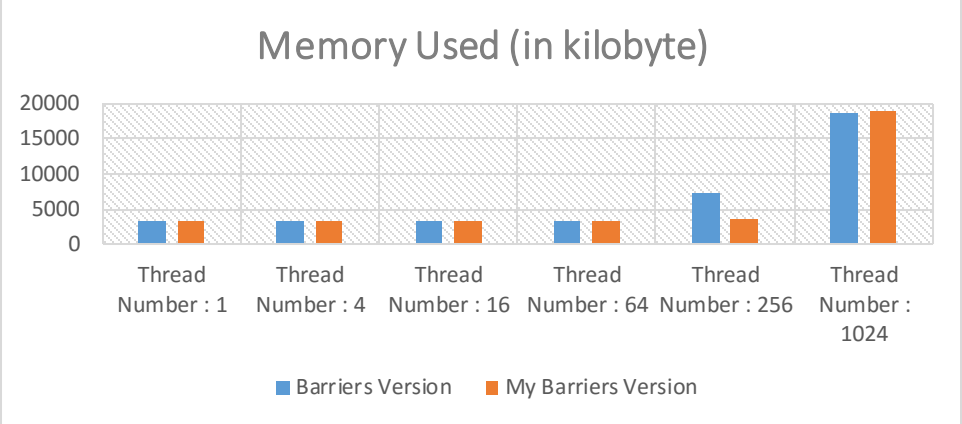

Fig.6 : Comparaison de l'empreinte mémoire en fonction du nombre de thread (1-2)

Bien que nous pouvons observer un gain en espace mémoire pour un nombre de thread égal à 256, les consommations restent égales.

5.2.3. *Évolution de la perte de performance en fonction du nombre de thread*

Voici l'évolution de la perte de performance en fonction du nombre de thread de la version avec barrière reconstruite (les chiffres des graphiques précédents ont été utilisés).

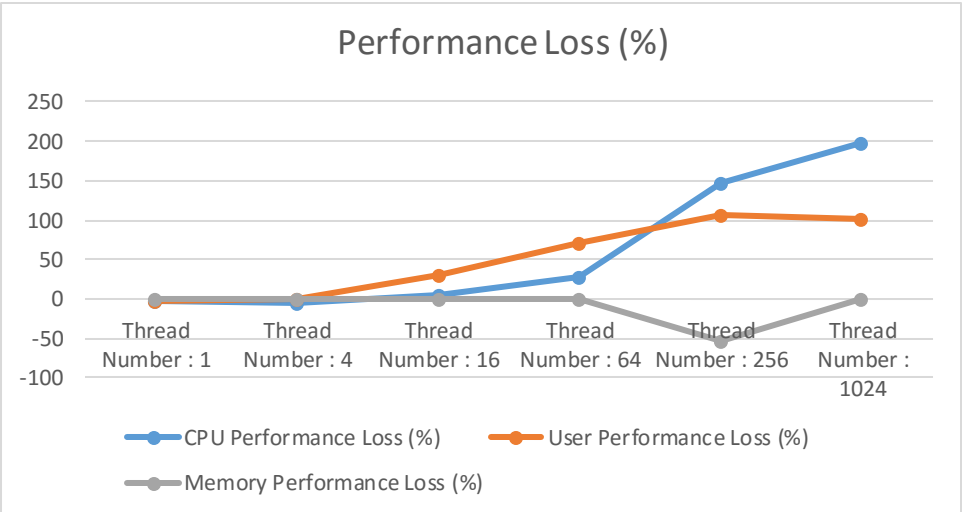

Fig.7 : Évolution de la perte de performance en fonction du nombre de thread

À partir de ce graphique, nous pouvons voir que l'espace mémoire utilisé n'a pas changé, excepté pour une seule valeur du nombre de thread. La modification de la gestion des processus n'a donc pas de réel impact sur l'empreinte mémoire.

Quant aux temps d'exécution, nous pouvons observer une perte de performance croissante , allant jusqu'à des temps 3 fois plus longs.

La version avec barrière reconstruite est donc moins performante au niveau des temps d'exécution. La perte de temps s'observe lors de l'appel à la fonction barrière_wait(), et elle peut être due au fait que :

- La mise en attente des processus est plus lente que dans la fonction Posix
- La libération des processus en attente est plus lente que dans la fonction Posix
- L'utilisation de variable condition et verrou implique une moins bonne exploitation du parallélisme, comparé à la l'utilisation de la barrière Posix.

5.3. Amélioration possible de l'utilisation de la barrière POSIX

Le problème avec la version actuelle réside dans le fait que le premier thread va devoir attendre que le dernier thread ait fini son traitement vertical alors qu'il lui suffirait juste d'attendre que les threads voisins aient fini.

Par exemple, dans un problème de taille [32x32] (1024 cellules) avec 1024 threads, le thread n°1 va devoir attendre la fin du traitement vertical du thread n°1024 pour commencer le traitement horizontal.

Si nous découpons la matrice de la manière suivante, le thread n°1 devrait seulement attendre que les threads 2 et 33 pour commencer le traitement horizontal.

1	2	…	32
33	34	…	64
…	…	…	…
993	994	…	1024

Fig.8 : Matrice découpée en 1024 régions

Une amélioration serait donc possible en ne synchronisant un thread qu'avec ses threads adjacents.