# Programmation concurrente

Version 1.2 : programme avec barrière posix

Aurélien COLOMBET (ca309567)

13 mars 2016

## 1. Introduction

Dans le cadre de ce projet, nous simulons un modèle très simple du phénomène de transfert de chaleur par conduction. Nous partons d'une plaque d'une taille définie (par les paramètres d'exécution du programme), avec une température de base, un contour à température froide fixe, et un centre à température chaude, fixe aussi.

## 2. Hypothèses et calcul des constantes

Notre programme prend en entrée divers paramètres qui lui permettent de définir des données nécessaires à la simulation. Une des données les plus importantes pour commencer est la définition de la taille de la plaque. Nous obtenons le nombre de lignes (et de colonnes) en faisant le calcul $2^{N+4}$ (N compris entre 0 et 9), obtenant ainsi une plaque de taille minimale $2^4 * 2^4$ (= $16 * 16$ ) et de taille maximale $2^{13} * 2^{13}$ (= $8192 * 8192$ ).

A partir de la taille obtenue, nous pouvons calculer les deux constantes utilisées lors de notre programme, nécessaires afin de trouver le centre de la plaque (la partie chauffante). Le simulateur calcule l'emplacement du coin supérieur gauche $2^{N+4-1} - 2^{N+4-4}$ et celui du coin inférieur droit $2^{N+4-1} + 2^{N+4-4}$ . La zone située entre ces deux points est la zone chauffante.

Pour la gestion du contour de la plaque qui doit être à une température froide, on augmente la taille de 2 de la matrice pour avoir la température froide autour de la plaque.

# 3. Algorithme

## a. Description des structures de données

Afin de représenter la plaque, nous avons choisi d'utiliser un tableau à deux dimensions (une matrice). Pour les besoins de notre algorithme, nous utilisons deux tableaux à deux dimensions, un tableau temporaire permettant d'effectuer les calculs intermédiaires, et un tableau contenant le résultat final de chaque itération (ce tableau ne contient que des valeurs correctes à une itération donnée).

## b. Description de l'algorithme itératif

L'algorithme de diffusion de chaleur se déroule en trois étapes, les deux premières étant le calcul. Tout d'abord, nous parcourons la matrice de référence et remplissons la matrice temporaire en effectuant le calcul :

$$(\frac{1}{6} * cellule\ gauche) + (\frac{4}{6} * cellule\ milieu) + (\frac{1}{6} * cellule\ droite)$$

Nous parcourons maintenant la matrice temporaire (sur laquelle nous venons d'écrire), pour écrire sur la matrice de référence avec le même algorithme que précédemment, mais cette fois-ci à la verticale :

$$(\frac{1}{6} * cellule\ dessus) + (\frac{4}{6} * cellule\ milieu) + (\frac{1}{6} * cellule\ dessous)$$

Enfin, nous remplissons à nouveau la zone centrale avec la température chaude de départ (la partie centrale étant la partie chauffante, sa valeur change durant les étapes de diffusion, mais doit redevenir à la température chaude une fois les calculs effectués).

Voici le pseudo-code :

```
// Premiere etape : calcul horizontal
for (i = 1; i < size - 1; i++) {
        for (j = 1; j < size - 1; j++) {
                mat2[i][j] = (mat1[i][j-1]/6.0)
                        + (mat1[i][j+1] / 6.0)
                        + (mat1[i][j]) * 2.0 / 3.0;
        }
}

// Deuxieme etape : calcul vertical
for (i = 1; i < size - 1; i++) {
        for (j = 1; j < size; j++) {
                mat1[i][j] = (mat2[i-1][j] / 6.0)
                        + (mat2[i+1][j] / 6.0)
                        + (mat2[i][j]) * 2.0 / 3.0;
        }
}

fillCenter(mat1, size);//remplit le centre de la matrice par TEMP_CHAUD
```

### c. Description de l'algorithme avec threads (barrière POSIX)

Pour cette étape nous utiliserons la barrière POSIX avec des threads. Nous utiliserons le même algorithme que pour la version itérative mais cette fois-ci nous allons découper notre matrice pour faire les calculs de diffusions dans ces sous matrices par des threads.

L'utilisation de la barrière permet d'attendre que tous les threads avant de continuer. Donc dans notre étape nous initialisons une barrière POSIX pour le calcul horizontal de la diffusion, nous passons en paramètres le nombre de threads que la barrière doit attendre donc le nombre de threads que nous devons créer plus un qui est le thread du main. Nous créons ensuite les threads qui vont faire le calcul sur chaque sous matrices. Nous attendons que toutes les threads aient fini en faisant une attente avec la barrière POSIX. Ensuite, nous détruisons la barrière et nous utilisons le même procédé pour le calcul vertical.

Lorsque le calcul vertical est terminé et que la barrière est détruite, nous faisons une barrière pour remplir le centre de la matrice puis nous détruisons la barrière lorsque le centre est rempli.

Voici le pseudo-code :

```
pthread_barrier_init(barrierHorizontal, nbThreads+1);

for(indThread=0; indThread<nbThreads; indThread++){
      pthread_create (calculSubMatrixHorizontal(indicesSousMatrices));
}

pthread_barrier_wait (barrierHorizontal);
pthread_barrier_destroy(barrierHorizontal);

pthread_barrier_init(barrierVertical, nbThreads+1);

for(indThread=0; indThread<nbThreads; indThread++){
      pthread_create (calculSubMatrixVertical(indicesSousMatrices));
}

pthread_barrier_wait (barrierVertical);
pthread_barrier_destroy(barrierVertical);

pthread_barrier_init(barrierFillCenter, 1);
fillCenter(mat1, size);
pthread_barrier_wait(barrierFillCenter);
pthread_barrier_destroy(barrierFillCenter);
```

# 4. Conclusion

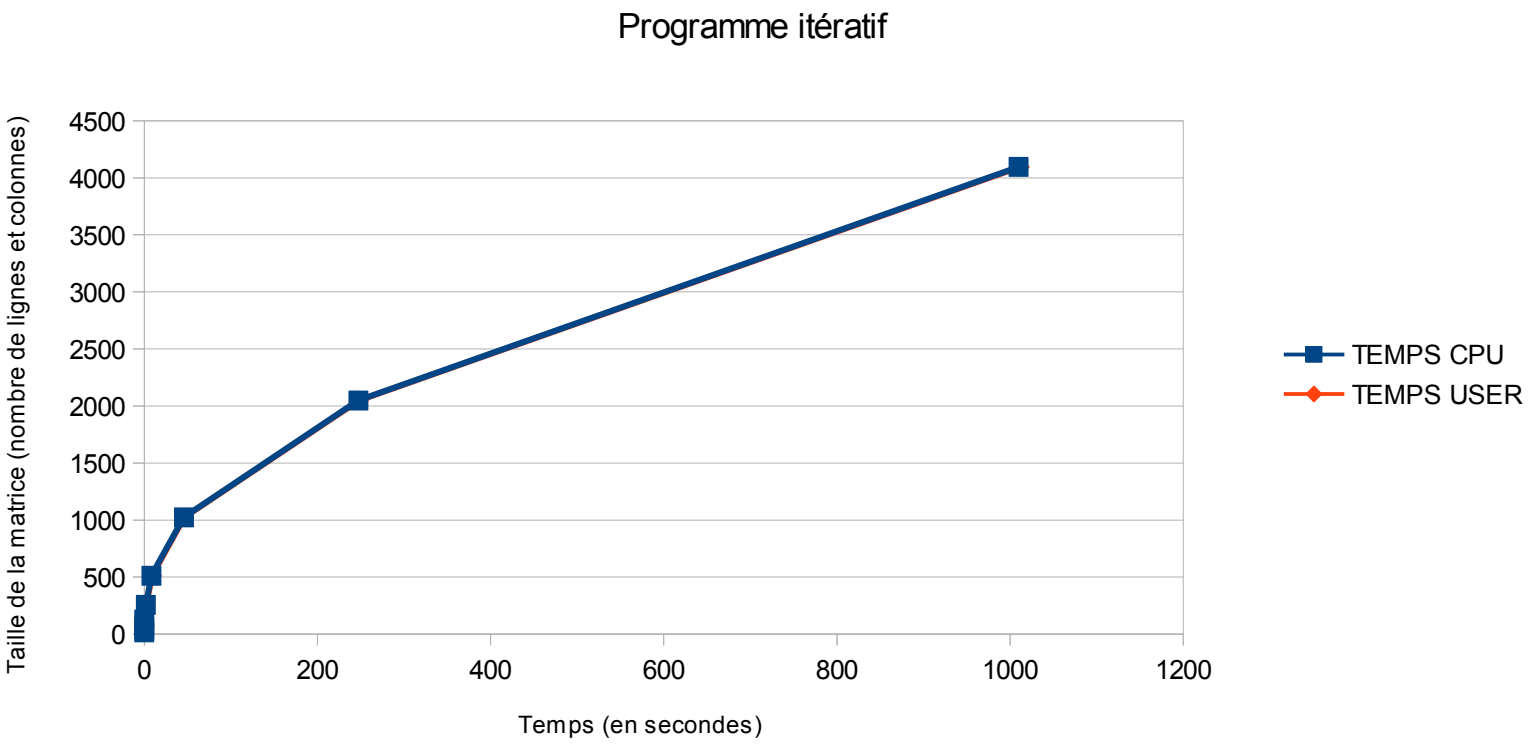
## a. Résultats obtenus

Par défaut, le programme exécute les options : -i 10000 –e 012345 –s 024 –t 13 –m. A notre avancée, cela correspond à effectuer 10 000 itérations (option ‘i’) sur une plaque de 16*16 ( $2^{0+4}$ ), puis sur une plaque de 64*64 ( $2^{2+4}$ ), puis sur une plaque de 256*256 ( $2^{4+4}$ ), et pour chacune des plaques, effectuer dix fois ces itérations pour faire une moyenne de consommation du CPU (option ‘m’).

La consommation du CPU correspond au temps pris pour appliquer l’algorithme, divisé par le nombre de tics d’horloge du CPU par seconde. En lançant notre programme avec les options citées précédemment, nous obtenons un temps de 0 tic d’horloge CPU pour une plaque de taille 16*16, un temps de 1 tic d’horloge CPU pour une plaque de taille 64*64, et un temps de 22 tics d’horloge CPU pour une plaque de taille 256*256.

## b. Résultats obtenus pour l'étape itérative

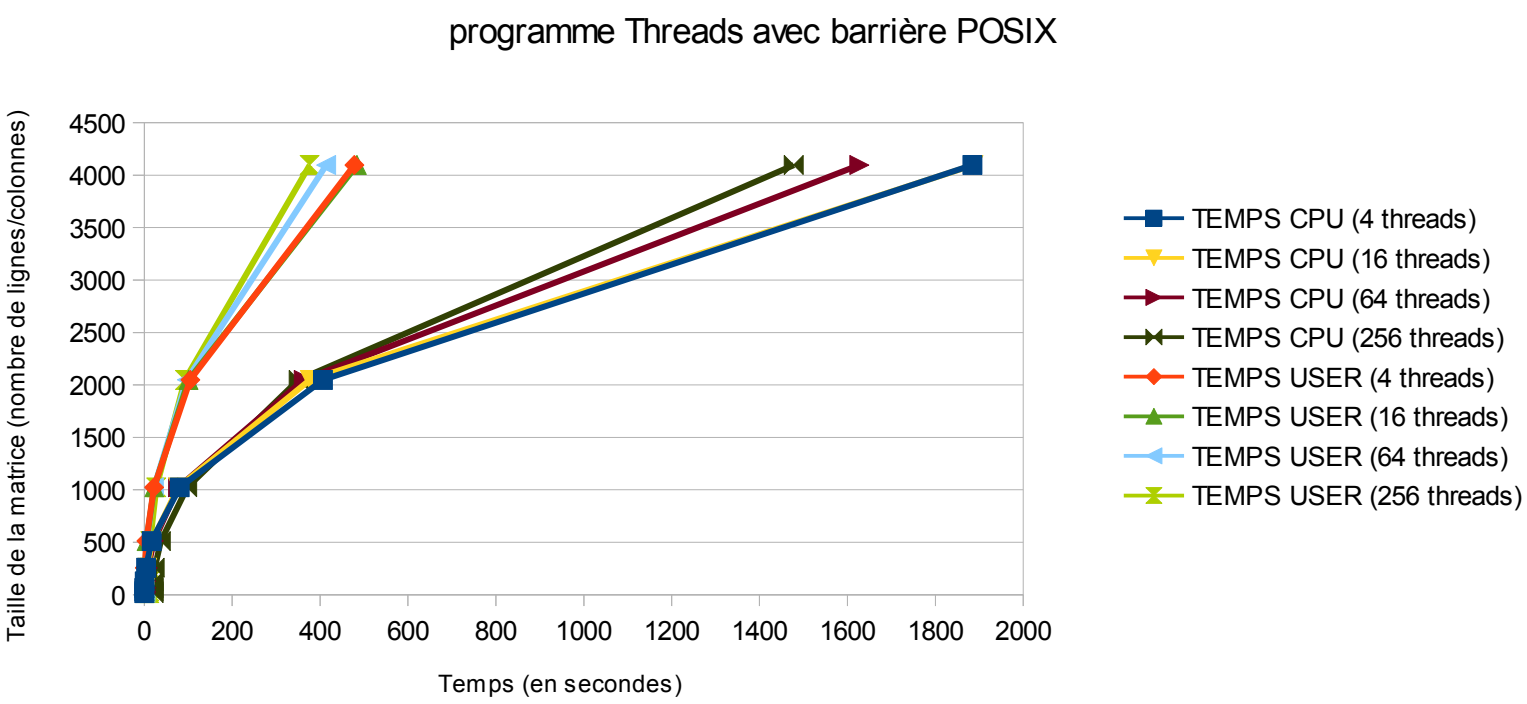
Voici le résultat obtenu pour 1000 itérations avec le programme itératif :




Grâce à ces résultats on voit bien que les courbes de temps CPU et temps utilisateur coïncident. Le temps utilisateur et le temps CPU sont les mêmes quelque soit la taille de la matrice. On voit qu'à chaque fois qu'on multiplie la taille de la matrice par 4, le temps CPU et utilisateur est lui aussi multiplié par 4.

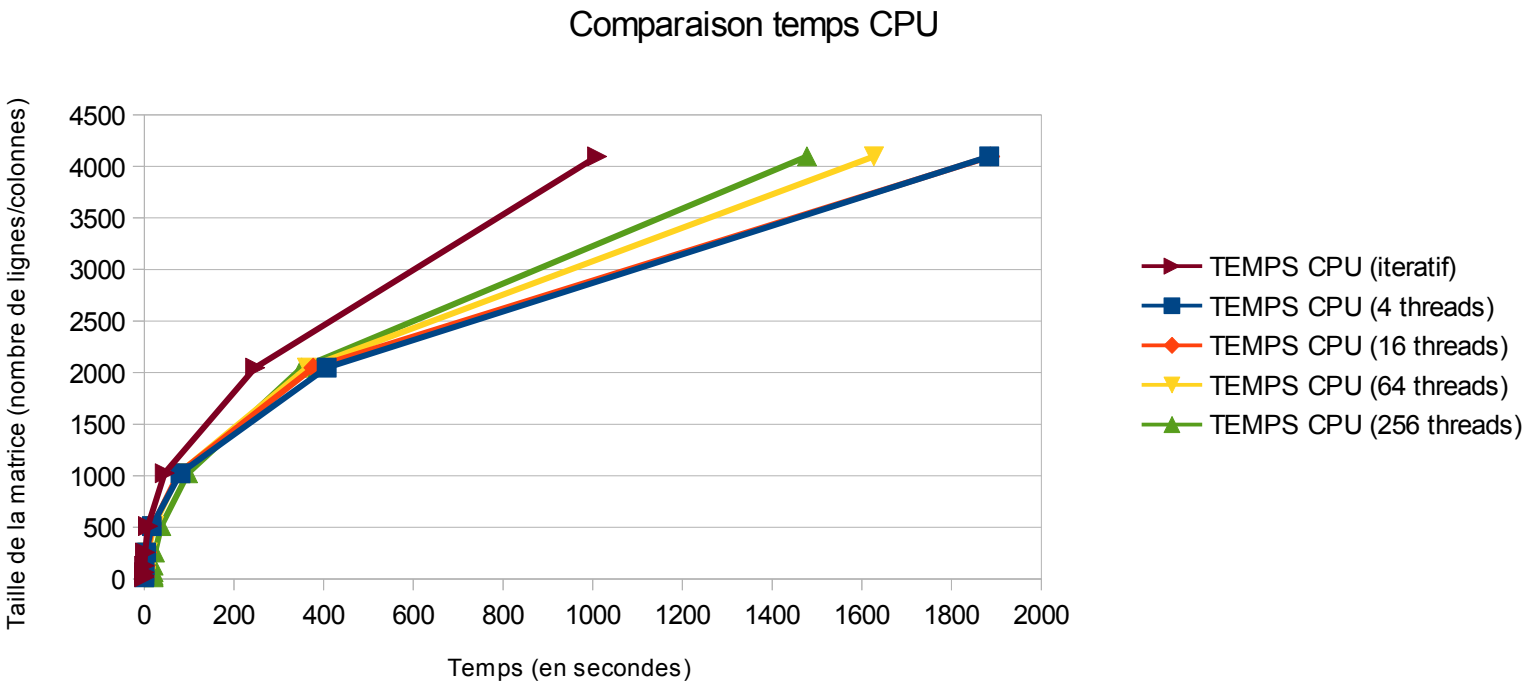
### c. Résultats obtenus pour l'utilisation de la barrière POSIX

Voici le résultat obtenu pour 1000 itérations avec l'utilisation de la barrière POSIX :

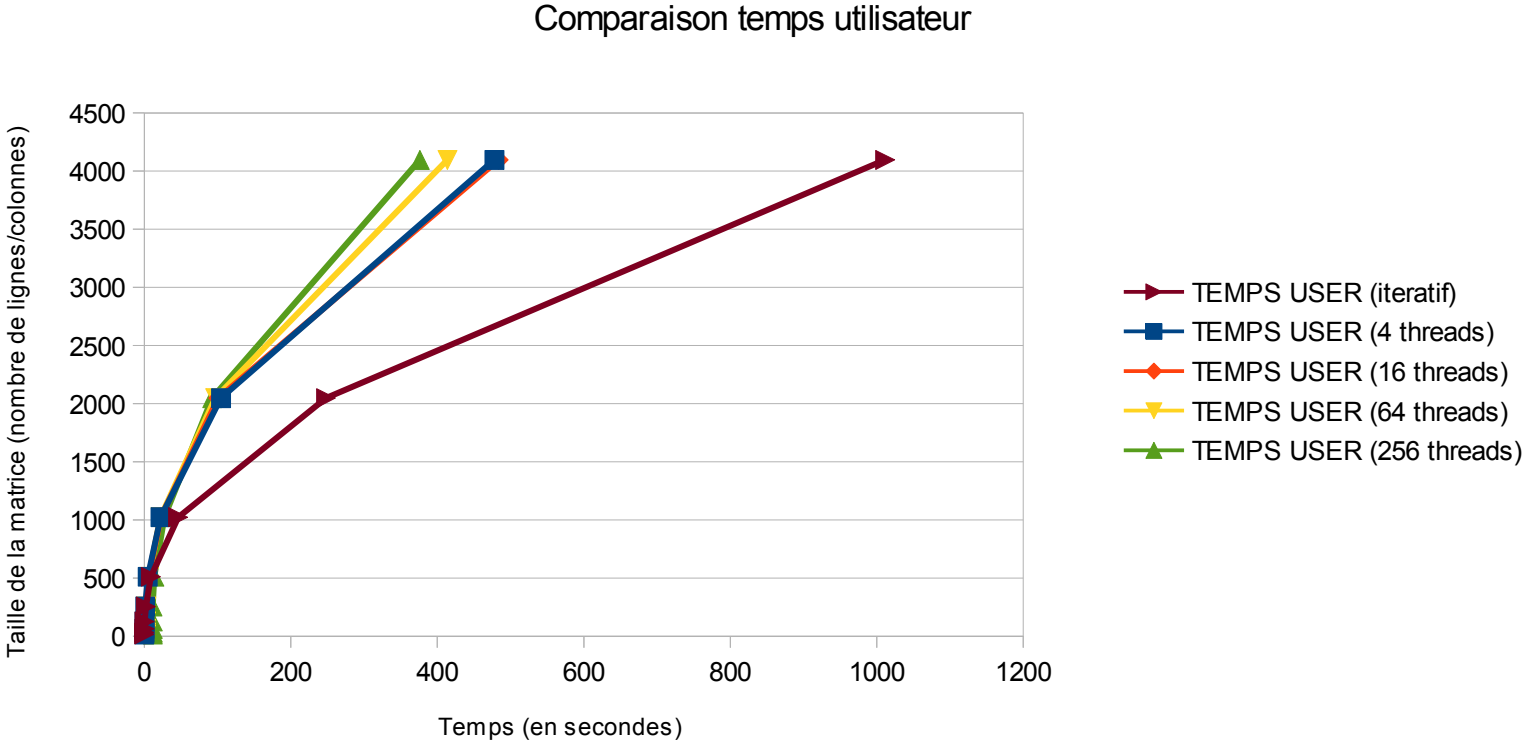



On voit grâce à ce diagramme que quelque soit le nombre de threads, le temps utilisateur est toujours 4 fois moins élevé que le temps CPU. Le nombre de threads utilisé est plus rentable lorsqu'on a des tailles de matrices supérieures à 1024x1024 sinon le temps utilisateur et le temps CPU sont plus élevés que pour le programme itératif (de l'ordre de 20 secondes plus long que le programme itératif pour de petites tailles de matrice).

Comparons maintenant ces résultats avec le programme itératif...




Nous voyons sur ce graphique que le temps CPU est plus élevé lorsque nous utilisons des threads que lorsqu'on calcule avec la méthode itérative (entre 1,5 et 2 fois plus élevé).




Nous voyons sur ce graphique que le temps utilisateur lorsque nous utilisons des threads est moins grand que lorsque nous utilisons la méthode itérative (entre 2 et 2,5 fois moins élevé). En conclusion, nous avons vu que l'utilisation de la barrière à un coût sur le temps CPU mais en contre partie diminue le temps utilisateur de manière non négligeable.