Paris Simon

Rapport

Programmation concurrente et parallèle

Introduction :

L'objectif de ce projet est de simuler la diffusion de chaleur sur une plaque de matière quelconque. Pour modéliser cette plaque nous allons la scinder en plusieurs cellules de dimension égale afin de pouvoir leur attribuer une température. L'objet mathématique choisi sera la matrice. Pour une matrice de taille(NxN) nous créerons un **tableau unidimensionnel** de taille NxN et nous accèderons par exemple à la cellule (2,6) avec l'instruction suivante : tableau[2 * N + 6]. Ce tableau sera constitué d'une **structure** contenant 3 champs : une température courante et la nouvelle suite à la diffusion. Il y aura aussi un champ booléen indiquant si la cellule est constante. En effet, les cellules au centre d'indice i (cf sujet) seront initialement d'une certaine température T et la conserveront tout au long des itérations. De même, pour toutes les cellules sur l'extérieur nous fixons la température à 0. Ces cellules sont rajoutées en plus de notre matrice(NxN) demandée. Toutes les autres cellules seront initialisées à 0 avec le champ constant à 0.

Algorithme :

Voici l'algorithme de diffusion de chaleur que nous avons implémenté :

```
run {
        for (nombre_itération) {
                horizontale
                verticale
                relnit_matrice
}

horizontale {
        for (ligne de 1 à taille_matrice - 1) {
                for (colonne de 1 à taille_matrice - 1) {

                        new_cellule = ( 4 * cellule_courante + cellule_gauche + cellule_droite) / 6
                }
        }
}
```

```
verticale {
        for (ligne de 1 à taille_matrice - 1) {
                for (colonne de 1 à taille_matrice - 1) {

                        new_cellule = ( 4 * cellule_courante + cellule_haut + cellule_bas) / 6
                }
        }
}

relnit_matrice {
        for (ligne de 0 à taille_matrice) {
                for (colonne de 0 à taille_matrice) {
                        if (cellule_courante == cellule_chaude)
                                cellule_courante = TEMPS_CHAUD
                        if (cellule_courante == cellule_froide)
                                cellule_courante = TEMPS_FROID
                }
        }
}
```

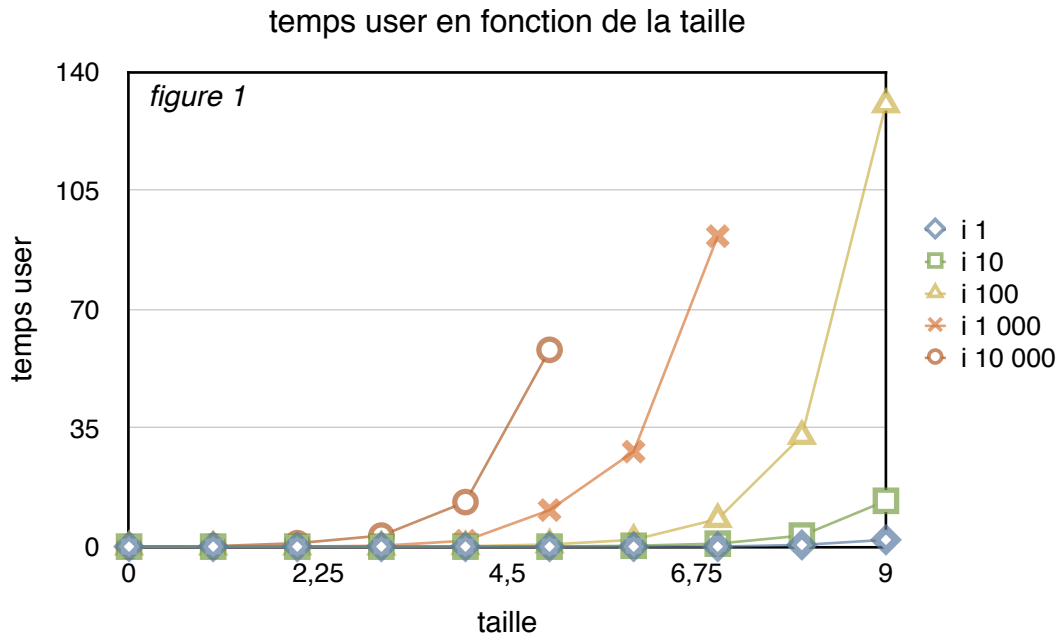
Le **résultat attendu** est une matrice **symétrique** avec les cellules centrales et extérieurs à leurs valeurs initiales. Les autres cellules auront subit les transferts de chaleur successifs et auront donc des valeurs de plus en plus faible tandis qu'elles seront éloignées du centre de la matrice.
Pour l'implémentation avec thread on modifie légèrement le code ci-dessus pour pouvoir parcourir une sous matrice qui sera attribué à chaque thread. On attribut à chaque thread ça sous matrice de la manière décrite ci-dessous. A noter que le '+1' à pour but d'éviter d'attribuer des cellules appartenants aux bords.

```
sous_matrice = taille_matrice / 2^t
for (x de 0 à 2^t) {
        for (y de 0 à 2^t) {
                thread.x = x * sous_matrice + 1
                thread.y = y * sous_matrice + 1
                thread.sousMatrice_taille = sous_matrice
        }
}
```

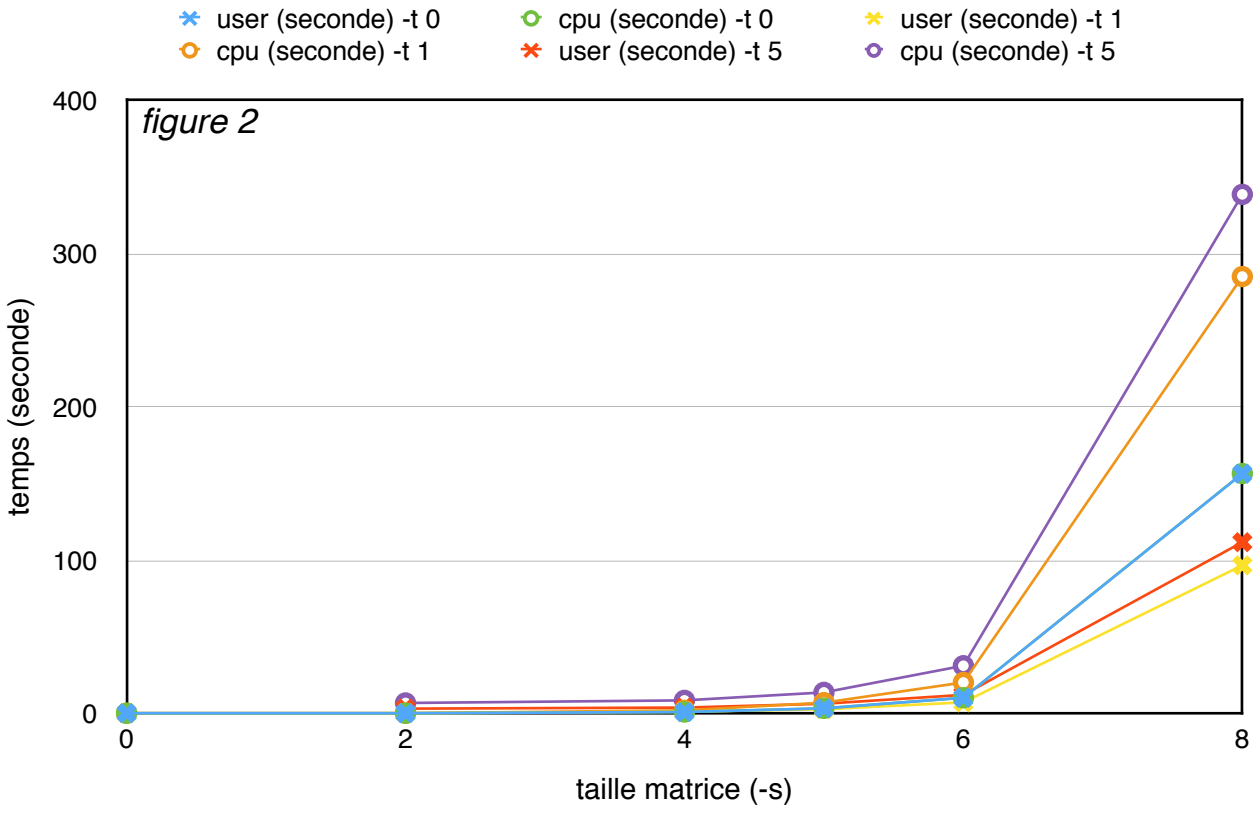
On synchronise les 4^t thread grâce à 3 barrières. Elles protègent les 3 parties suivantes: itération verticale, horizontale et réinitialisation des valeurs de température chaud et froid.

Résultats / interprétations :

Performance de temps :

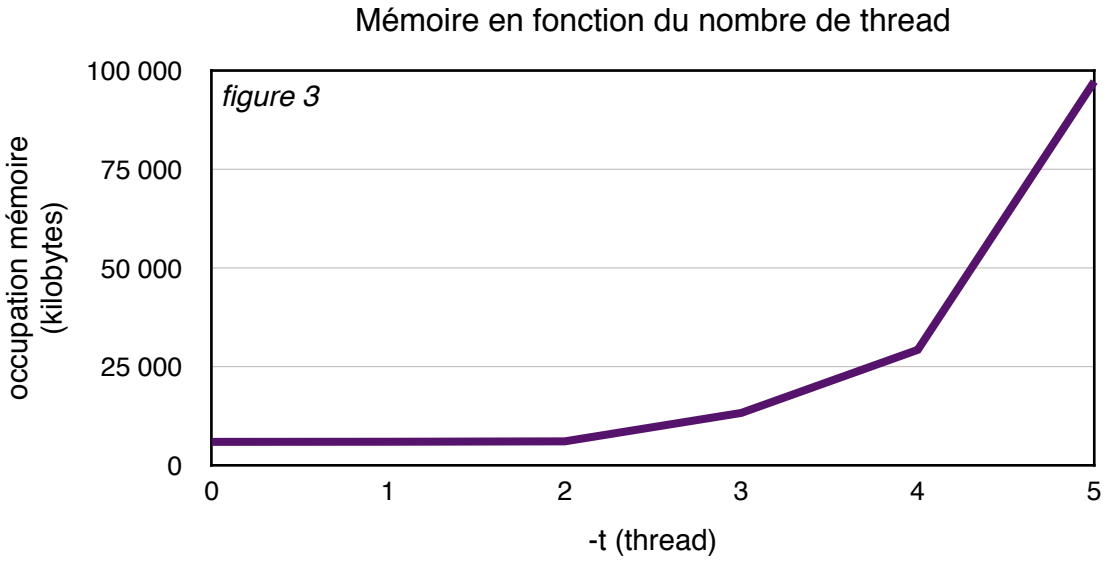


Sur ce graphique nous pouvons voir que l'augmentation de la taille de la matrice augmente le temps de calcul de manière exponentielle. Cela n'a rien d'anormal compte tenu du fait la taille est une puissance de 2 de l'option 's'.



La *figure 2* nous montre que plus l'on augmente la taille de notre matrice, plus l'utilisation de thread réduit la durée d'exécution. C'est en utilisant 4 thread, soit l'option '*-t 1*' que le temps de calcul est optimal. Pour les autres valeurs supérieurs de l'option t, nous avons une légère augmentation du temps de calcul. C'est observation sur le temps d'exécution concerné le temps **user**. Le temps **cpu** lui sera quasiment égal au temps **user** en l'absence de thread, et va augmenter plus le nombre de thread augmentera comme nous pouvons le voir sur la *figure 2.*

Performance de mémoire :



La *figure 3* montre le coup de l'implémentation de thread. Pour obtenir cette courbe, la taille et le nombre d'itération on était fixé. Le nombre de thread étant égal à 4^t cela explique l'augmentation de nouveau exponentiel de l'occupation mémoire.

Enfin ce tableau où la deuxième colonne correspond à la mémoire utilisée pour le processus. La troisième correspond à cette mémoire divisée par la taille en mémoire d'une cellule de notre matrice et la dernière au nombre de cellule effectif (pour s = 0, on a (16+2)*(16+2) = 324. Le '+2' correspond aux cellules extérieures rajoutées qui seront maintenant à une température de 0). Nous constatons que la mémoire utilisée tend à correspondre à l'espace nécessaire aux stockage de toutes les cellules. Ainsi, on peut conclure que la mémoire utilisée est largement déterminé par le nombre de cellules de notre matrice.

s	kilobytes	memo/cell	cell
0	412	34 816	324
1	612	52 224	1 156
2	612	52 224	4 356
3	664	56 320	16 900
5	4 436	377 856	264 196
6	16 036	1 368 064	1 052 676
7	62 116	5 300 224	4 202 500
8	209 716	17 895 424	16 793 604
9	799 728	68 243 456	67 141 636