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 2 champs : une température courante et la nouvelle suite à la diffusion. 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 une certaine constante TEMP_FROID. A noter que ces cellules sont rajoutées en plus de notre matrice(NxN) demandée, on a donc une matrice(N+2xN+2). Toutes les autres cellules seront initialisées 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**) {
            if (cellule_courante == cellule_chaude)
                cellule_courante = TEMPS_CHAUD
            else
                new_cellule = (4 * cellule_courante + cellule_haut + cellule_bas) / 6
        }
    }
}

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 (si la température extérieur est 0) .
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 2 barrières. Elles protègent les 2 parties suivantes: itération verticale et horizontale. 3 barrières étaient utilisées dans le rendu précédent pour protéger la phase de réinitialisation du centre de la plaque au TEMP_CHAUD. Cette phase à était incorporé dans la phase de propagation verticale est nous à donc permit de supprimer la derniers phase, avec sa barrière.

Dans la deuxième étape de ce projet nous avons implémenté nos propres barrières avec les variables conditions Posix, voici l'algorithme :

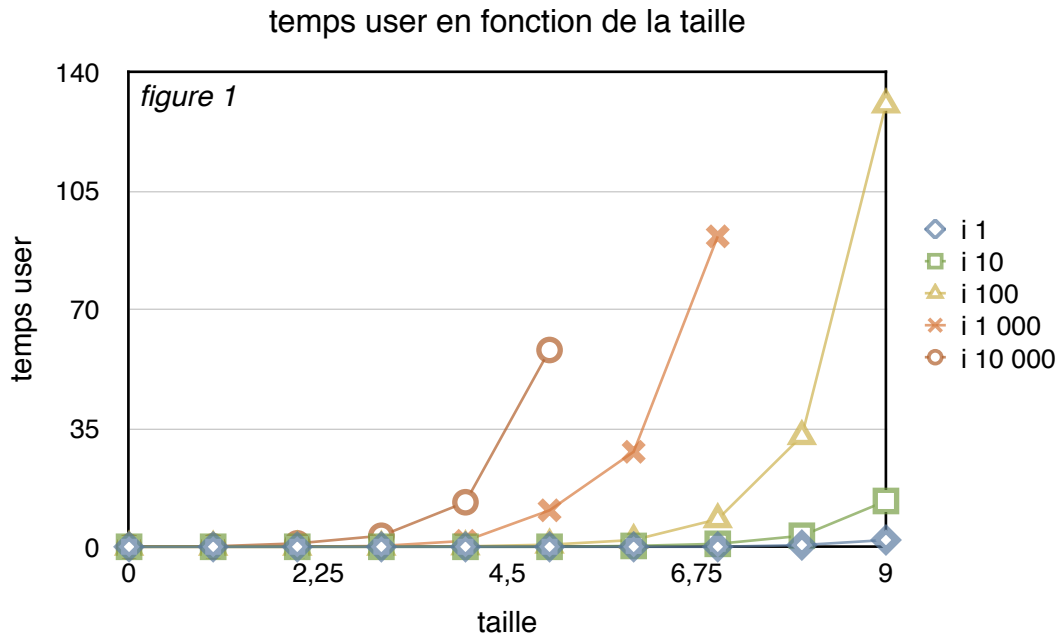
structure **my_barrier** (
    int value
    int max
    pthread_mutex_t mutex   //pointeur
    pthread_cond_t cond   //pointeur
)

```
pthread_barrier_init (my_barrier barrier, int  count) {
        ret = pthread_mutex_init(barrier.mutex, 0)
        if (ret) return ret
        ret = pthread_cond_init(barrier.cond, 0)
        barrier.value = 0
        barrier.max = count
        return ret
}

pthread_barrier_destroy (my_barrier barrier) {
        ret = pthread_cond_destroy(barrier.cond)
        if (ret) return ret
        ret = pthread_mutex_destroy(barrier.mutex)
        if (ret) return ret
        return ret
}

pthread_barrier_wait (my_barrier barrier) {
        ret = pthread_mutex_lock(barrier.mutex)
        if (ret) return ret
        barrier.value = barrier.value + 1
        if (barrier.value == barrier.max) {
                barrier.value = 0
                ret = pthread_cond_broadcast(barrier.cond)
                if (ret) return ret
                ret = pthread_mutex_unlock(barrier.mutex)
                if (ret) return ret
        } else {
                while (0 != barrier.value) {
                        ret = pthread_cond_wait(barrier.cond, barrier.mutex)
                        if (ret) return ret
                }
                ret = pthread_mutex_unlock(barrier.mutex)
                if (ret) return ret
        }
        return ret
}
```

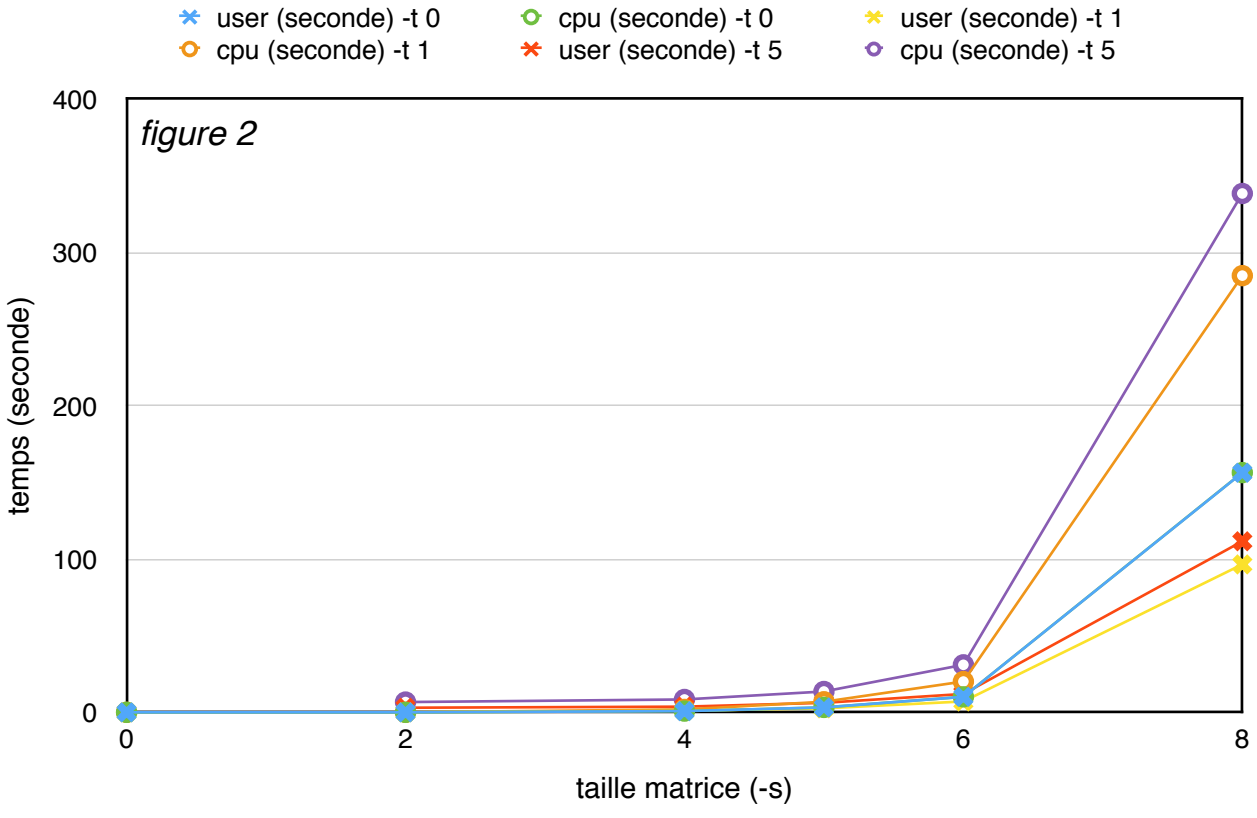
Les fonctions en rouge sont des fonctions de Posix avec lesquelles nous devions composer pour implémenter nos barrières.

Résultats / interprétations :

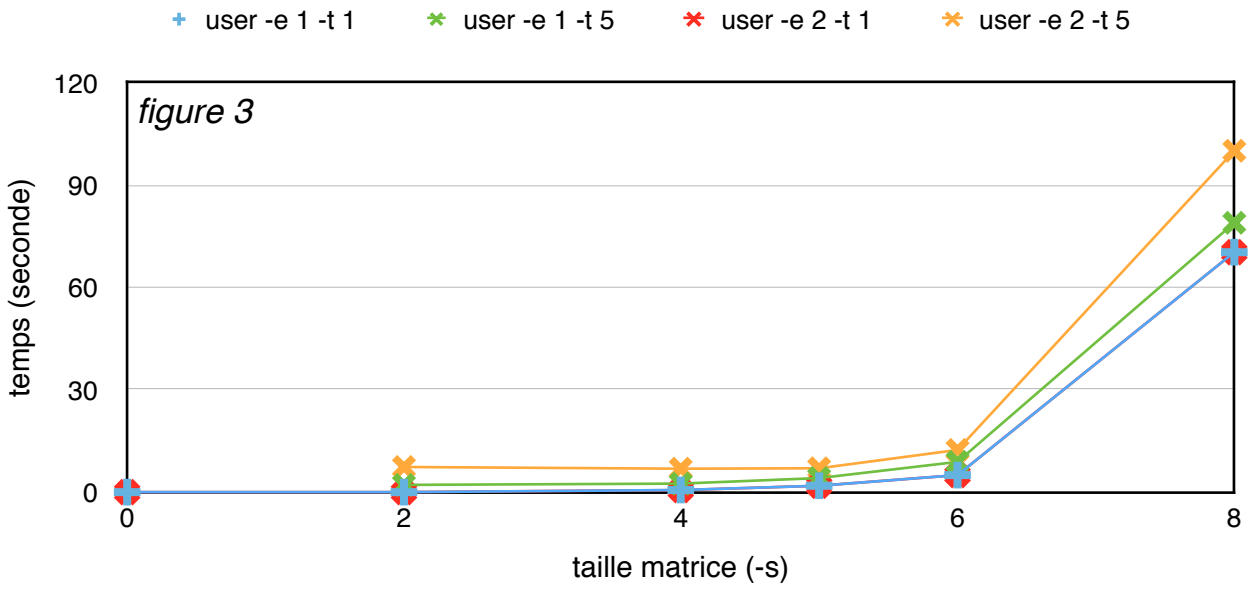
Performance de temps :



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'.

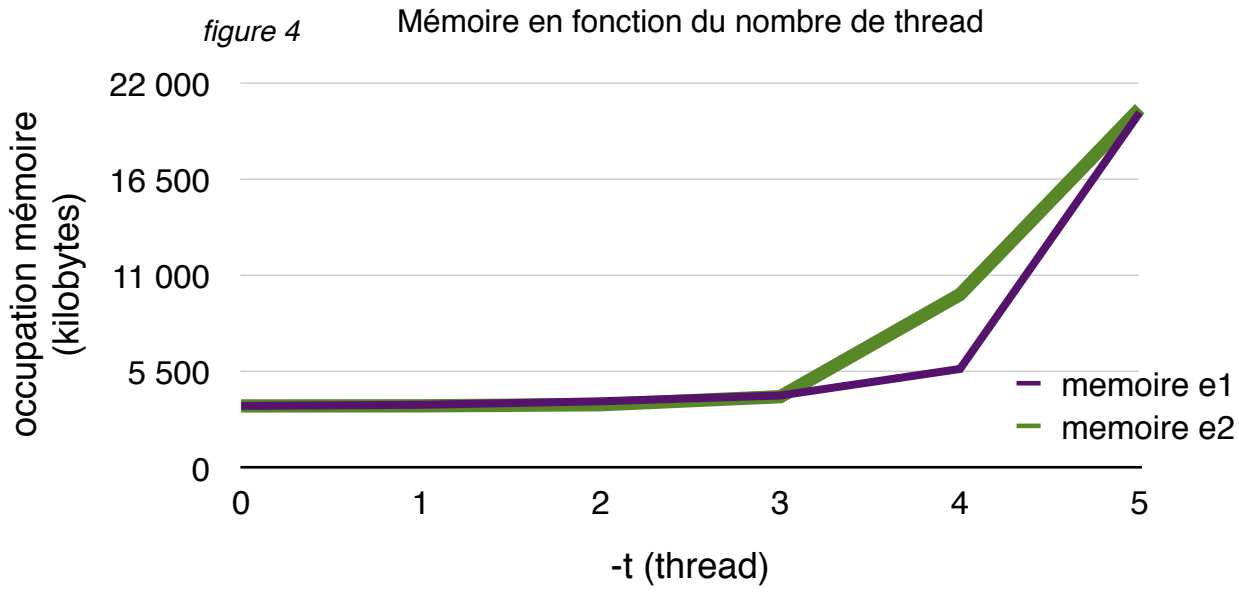


La *figure 2* nous montre que plus l'on augmente la taille de notre matrice, plus l'utilisation de thread réduit la durée d'exécution. C'est en utilisant 4 thread, soit l'option '*-t 1*' que le temps de calcul est optimal. Pour les autres valeurs supérieurs de l'option t, nous avons une légère augmentation 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.*



La *figure 3* est une comparaison des barrières Posix et de nos propre barrières. On peut constater que pour l'option *'-t 1'* les temps d'exécutions avec nos barrières et celles Posix est quasiment similaire. En revanche quand on augmente le nombre de thread, on remarque que notre implémentation de barrière est moins performante. Cela n'a rien de choquant, les barrières Posix implémentent de toutes évidences des optimisations par rapport à notre implémentation relativement simple.

Performance de mémoire :



La *figure 4* 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. On peut voir que les deux implémentation on une utilisation mémoire qui semble tendre vers le même nombre de kilobyte utilisée

Ce tableau concerne l'étape 0, 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

Conclusion :

Nous avons donc pour l'étape 1, remarquer que l'utilisation de thread était optimale pour 4 threads, cela correspond au nombre de coeur de la machine qui a effectuer les calcules. Cela justifie l'utilisation des GPU pour ce genre de calcule, qui sont "simple" et répétitif, on pourra gagner en performance en augmentant le nombre de thread jusqu'à atteindre le nombre de coeur du GPU.
Pour l'implémentation de l'étape 2 on constate que notre implémentation des barrières n'est pas plus performante, du point de vu de la mémoire et du temps de calcul, que celles de Posix. C'était le résultat attendu.