Polytech Nice-Sophia

October 7, 2016

Programmation Concurrente

Plaques Chauffantes

Version 2

Auteur:
Alicia Marin *(ma202240)*

1 Introduction

Dans ce projet, il nous a été demandé de simuler un transfert de chaleur par conduction et de calculer l'évolution de la température en tout point d'un objet à une ou deux dimensions. Le principe est le suivant : nous divisons l'objet (une plaque) en cases de même taille ; le transfert de la température d'une case à l'autre se fait en utilisant la formule de Taylor. Pour simplifier les calculs, le coefficient H de cette formule est fixé à 6.

2 Programme Itératif

Ici sera présenté l'algorithme de calcul mis en place pour déterminer la diffusion de la chaleur dans une grille

2.1 Objets manipulés

Les deux principaux objets manipulés sont des **float**** représentant deux matrices (tableau de **float*** pointants chacuns sur des tableaux de **float**).

- matrix : Correspond à la grille de chaleur de la plaque à l'itération courante.
- matrix2 : Correspond à la grille de chaleur de la plaque à l'itération précédente (pour permettre le calcul de matrix).

À chaque itération, les deux matrices sont interchangées pour réduire le temps de calcul. Sans cela, nous devrions à chaque itération, recopier les valeurs de matrix dans matrix2 (car la matrice précédente est évidement celle qui était la courante de l'itération précédente). Les valeurs qui se trouvaient dans matrix2 (et qui se trouvent donc dans matrix en début d'itération) seront ignorées et remplacées par le résultat de la propagation effectuée.

2.2 Algorithme de propagation

```
//Doing horizontal diffusion;
for x allant de line_2 à line_max - 1 do
    //Computing values for the result's x'th line;
    for y allant de column_2 à column_max-1 do
        matrix2(x,y) ← matrix(x,y) *4;
        matrix2(x,y) ← matrix(x-1,y);
        matrix2(x,y) ← matrix(x+1,y);
        ;
        matrix2(x,y) ← matrix2(x,y)/6;
    end
end
//Doing vertical diffusion;
for x allant de line_1 à line_max - 1 do
    //Computing values for the result's x'th column;
    for y allant de la column_2 à la column_max-1 do
        matrix(x,y) ← matrix2(x,y) *4;
        matrix(x,y) ← matrix2(x,y-1);
        matrix(x,y) ← matrix2(x,y+1);
        ;
        matrix(x,y) ← matrix(x,y)/6;
    end
end
//Maintening hot cells;
for x allant de premiere_cellule_chaude à derniere do
    for y allant de premiere_cellule_chaude à derniere do
        matrix(x,y) = TEMP_CHAUD;
    end
end
```

Algorithm 1: Diffusion2D version 0

2.2.1 Execution

Une étape est constituée d'une diffusion verticale suivie d'une diffusion horizontale. À chaque diffusion, nous parcourons la matrice ligne par ligne, en appliquant à chaque case les valeurs indiquées plus bas.

Pour une diffusion verticale, Nous assignons pour chaque case de la matrice, la valeur suivante :

$$\frac{\textbf{case en haut} + \textbf{4*case courante} + \textbf{case en bas}}{\textbf{6}}$$ de la sous-étape précédente

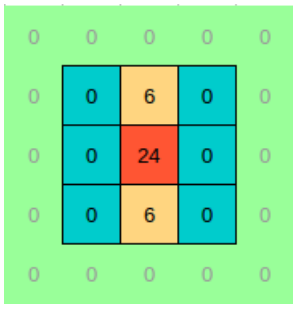

Figure 1: Image représentant la diffusion verticale

Pour une diffusion horizonale, Nous assignons pour chaque case de la matrice, la valeur suivante :

$$\frac{\textbf{case à droite} + \textbf{4*case courante} + \textbf{case à gauche}}{\textbf{6}}$$ de la sous-étape précédente

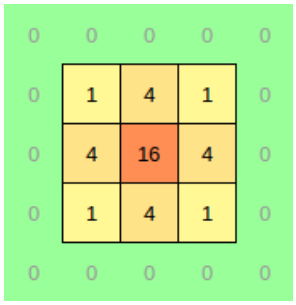

Figure 2: Image représentant la diffusion horizontale

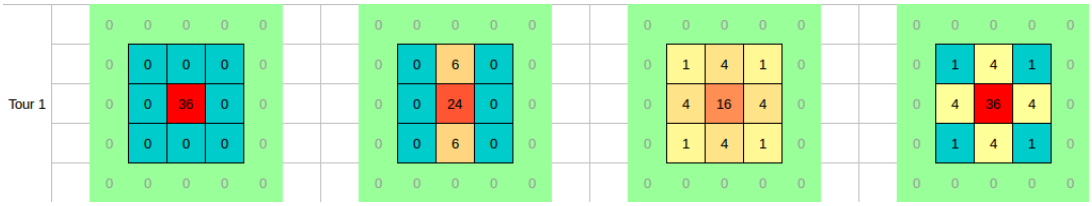

Figure 3: Image représentant la diffusion de la chaleur pendant une étape

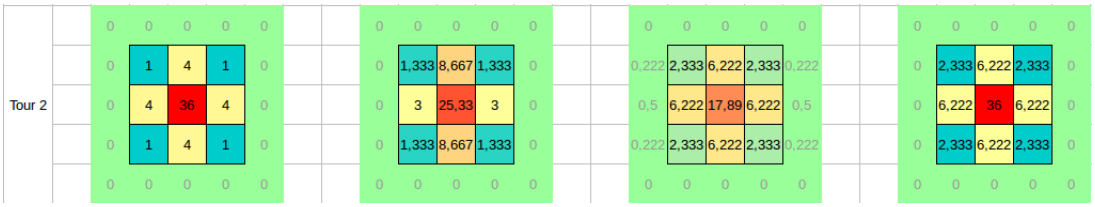

Figure 4: Image représentant la diffusion de la chaleur pendant une étape

En Figure 3 sont représentés les différentes sous-étapes d'une itération. La **première** sous-étape représente la grille initiale de l'itération courante. La **seconde** sous-étape représente la première diffusion, qui se propage à la verticale pour chaque colonne de la grille. La **troisième** sous-étape représente la seconde diffusion, qui se propage à l'horizontale pour chaque ligne de la grille. Enfin la **quatrième** sous-étape représente la grille finale de l'itération courante. Notez que la tuile centrale fait partie de la zone interne, elle est donc remise à la valeur TEMP_CHALEUR à chaque fin d'itération. Respecivement, le contour vert clair représente la zone externe et ne fait pas partie de la dimension du tableau; cette zone reste à la valeur TEMP_FROID à chaque itération.

En Figure 4 sont réprésentées les mêmes sous-étapes, mais de l'itération qui succède celle en Figure 3.

2.3 Synchronisation

Nous appliquons l'Algorithm1 (cf 2.2) pour chaque portion de la matrice, suivant le nombre de threads (découpés en puissance de 4) choisis en option. Les threads parallèles doivent s'attendre entre chaque sous-étapes, en effet si un thread est en avance sur les itérations, cela fausserait les calculs des diffusions. Nous appliquons donc une barrière après la diffusion horizontale

et après la diffusion verticale (et remise à TEMP_CHALEUR de la cellule concernée).
Ce qui fait en tout deux appels de barrière Posix pour une itération.

Diffusion2D;
    Call horizontal diffusion;
    Call barrier_wait;
    Call vertical diffusion;
    Call reset_hot_cells;
    Call barrier_wait;

Algorithm 2: Diffusion2D version 1 & 2

Pour la version 1, nous avons simplement utilisé la fonction barrier_wait de POSIX, pour la verison 2, nous avons implementé notre barrière, décrite dans l'Algorithm 3

```
pthread_mutex_lock(mutex);
count++;
//Si tous les threads ont passé l'étape;
if count >= total_threads then
    count = 0;
    //On debloque tous les threads en wait;
    pthread_cond_broadcast(cond);
    pthread_mutex_unlock(mutex);
end
else
    //S'il faut attendre les autres threads;
    pthread_cond_wait(cond, mutex);
    pthread_mutex_unlock(mutex);
end
```

Algorithm 3: Fonction barrière_wait avec mutex et variables cond

3 Synthèse des mesures

3.1 Version 0 - Séquentiel

Nous avons effectué de nombreuses mesures. Vous trouverez en Figure 5 et 6 les récapitulatifs des mesures.
Chaque ligne représente une instance à X itérations (1000, 2000,...) , et chaque colonnes représentent la taille de la matrice ($2^{(4+\mathbf{0})}, 2^{(4+\mathbf{1})}, ..$).

User	0	1	2	3	4	5	6	7
1000	0,0	0,0	0,1	0,5	2,1	9,3	35,4	141,1
2000	0,0	0,1	0,3	1,1	4,3	17,5	69,1	203,9
3000	0,0	0,1	0,4	1,6	6,8	26,9	106,9	373,8
4000	0,0	0,1	0,5	2,1	9,0	34,5	138,0	416,4
5000	0,0	0,2	0,7	2,6	10,6	43,0	173,2	687,5
6000	0,1	0,2	0,8	3,2	12,7	51,4	207,4	817,1
7000	0,1	0,2	0,9	3,7	14,9	59,9	241,4	909,6
8000	0,1	0,3	1,1	4,3	17,0	68,9	274,9	990,2
9000	0,0	0,2	0,7	2,7	10,8	43,7	277,3	1 103,9
10000	0,0	0,2	0,8	3,0	12,1	48,8	306,3	1 187,3

Figure 5: Tableau récapitulatif des mesures du temps réel obtenues (en seconde)

CPU	0	1	2	3	4	5	6	7
1000	0,0085	0,0333	0,1312	0,5241	2,1158	8,7106	35,0188	139,6293
2000	0,0170	0,0668	0,2647	1,0528	4,2013	17,0513	68,4682	203,1590
3000	0,0254	0,1000	0,3938	1,5926	6,4708	26,3931	105,2992	373,6643
4000	0,0340	0,1334	0,5282	2,1108	8,4531	34,1360	136,7972	416,3822
5000	0,0425	0,1657	0,6564	2,6239	10,5202	42,5787	171,4972	680,7319
6000	0,0511	0,1998	0,7908	3,1644	12,6497	51,0797	204,8821	811,0981
7000	0,0593	0,2318	0,9186	3,6779	14,7093	59,5639	238,9262	903,8555
8000	0,0676	0,2647	1,0516	4,2079	16,8359	68,2401	272,6605	985,3939
9000	0,0436	0,1714	0,6802	2,7173	10,8404	43,7499	277,2736	1 103,9252
10000	0,0486	0,1891	0,7576	3,0203	12,0705	48,8344	306,3587	1 187,3661

Figure 6: Tableau récapitulatif des mesures du temps CPU obtenues (en seconde)

Les temps CPU et réel sont très similaires, du fait qu'aucun autre processus n'était en activité sur notre PC. Le CPU tourne donc en temps réel sur notre programme.
Nous avons illutré les valeurs obtenues dans un graphique en Figure 7. Nous avons fait le choix de n'en illustré qu'un seul par le fait que les deux tableaux sont très similaires.

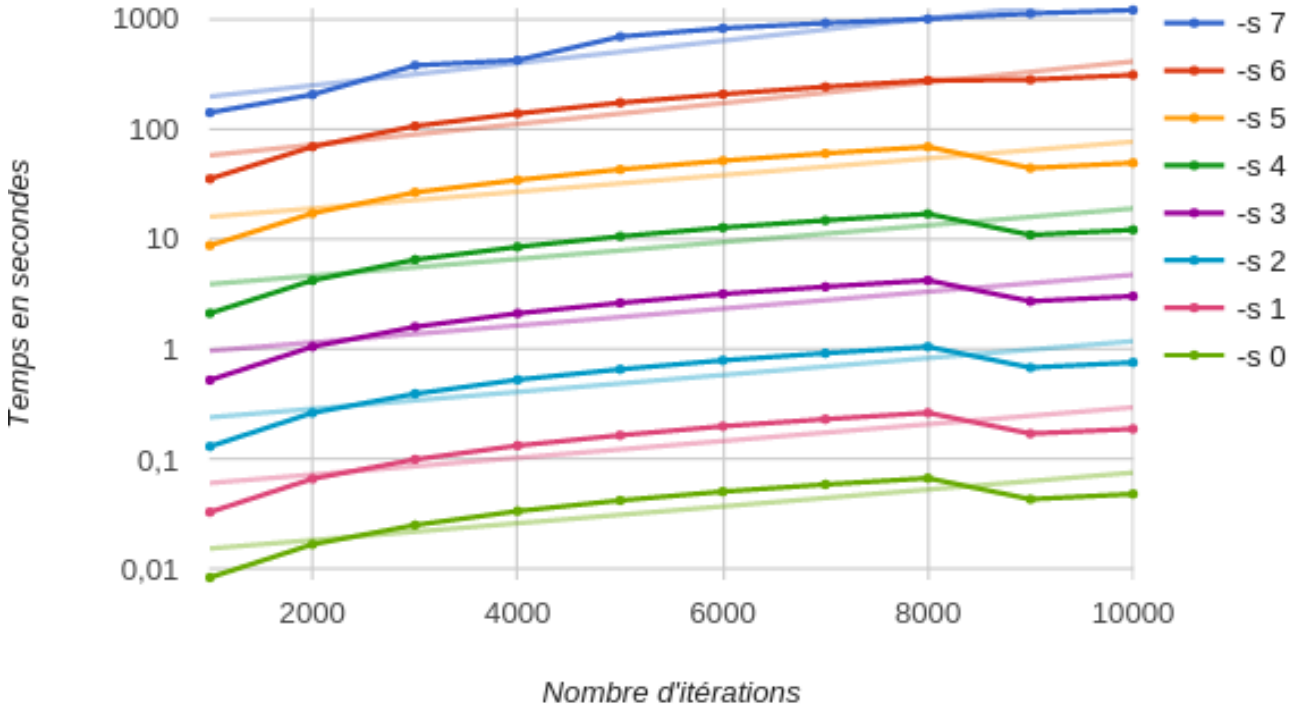

Figure 7: Temps CPU en fonction du nombre d'itérations pour chaque taille de matrice

- Nous pouvons observer en Figure 5, 6 et 7 que les temps CPU et utilisateurs sont sensiblement les mêmes (avec en toute logique le temps CPU inférieur).
- La croissance exponentielle de ces temps en fonction de la taille de la matrice : plus il y a de cases à traiter, plus les itérations sont longues (ce temps est linéaire par rapport au nombre de cellules de la matrice).
- A l'opposé, ces temps croissent linéairement en fonction du nombre d'itérations.

Ces trois faits dénotent bien des caractéristiques d'un CPU : moins fort en parallélisme et meilleur en calcul itératif.

3.2 Version 1 - Barrières Posix

Vous trouverez en Figure 12 et 13 les récapitulatifs des mesures.
Chaque ligne représente le nombre de threads $(4^0, 4^1, 4^2, ...)$
, et chaque colonnes représentent la taille de la matrice $(2^{(4+\mathbf{0})}, 2^{(4+\mathbf{1})}, ...)$.
Nous avons fait les calcules pour 1000 itérations.

nb thread/taille	-s 0	-s 1	-s 2	-s 3	-s 4	-s 5	-s 6
Version 0	0,005	0,023	0,071	0,271	1,041	4,244	17,32
-t 0	0,008	0,019	0,073	0,283	1,119	4,467	18,080
-t 1	0,080	0,057	0,091	0,199	0,664	2,599	10,320
-t 2	0,367	0,380	0,412	0,595	1,481	4,650	13,219
-t 3	1,532	1,625	1,568	1,715	2,439	5,275	14,978
-t 4	6,498	6,452	6,476	6,566	6,960	9,256	18,772

Figure 8: Tableau récapitulatif des mesures du temps réel obtenues (en seconde) Version 1

nb thread/taille	-s 0	-s 1	-s 2	-s 3	-s 4	-s 5	-s 6
Version 0	0,005	0,023	0,071	0,271	1,041	4,244	17,31
-t 0	0,008	0,019	0,073	0,283	1,120	4,475	18,081
-t 1	0,152	0,129	0,217	0,669	2,255	8,817	35,257
-t 2	0,608	0,633	0,788	1,182	3,113	10,055	34,338
-t 3	2,336	2,601	2,563	2,999	4,665	11,259	35,761
-t 4	10,390	10,354	10,273	10,853	12,169	19,562	47,413

Figure 9: Tableau récapitulatif des mesures du temps CPU obtenues (en seconde) Version 1

Nous pouvons remarquer que les temps de calculs en fonction de la taille de la matrice, sont plus importants que dans le programme séquentiel (version 0) pour des petites tailles de matrice (cf ligne 1, Figure 7). Par contre, pour une grande taille (s >2), le programme concurrent devient plus performant.
Cela pourrait s'expliquer par le temps d'initialisation et la gestion des barrières, qui n'est pas très rentable pour une petite matrice. Cependant pour une grande matrice, divisé les charges de travails en plusieurs thread est plus interessant.

Par contre, les résultats montrent que s'il y a trop de threads pour un processus, les performances décroient. En effet, pour une taille de s -6, nous obtenons le résultat 13s pour 4**2 threads, et 18s pour 4**4 threads.

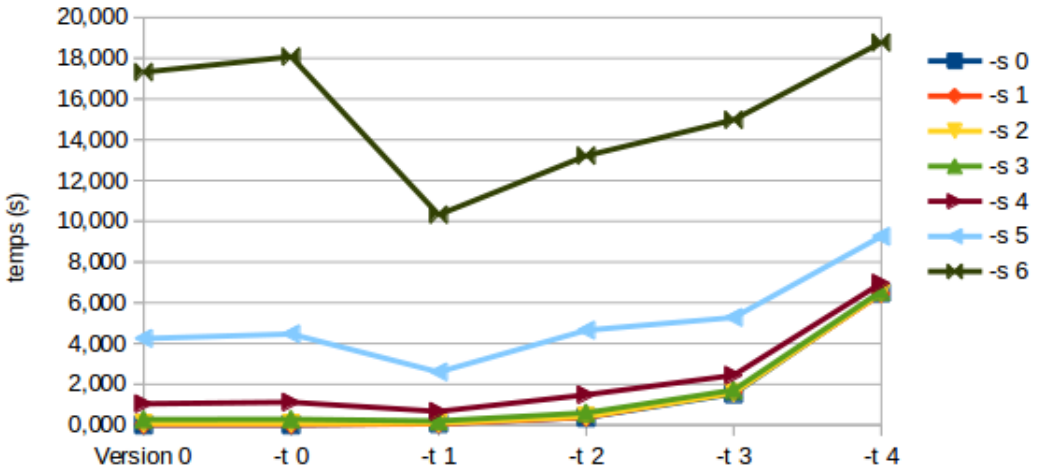

Figure 10: Temps User en fonction de la taille de matrice par option -t

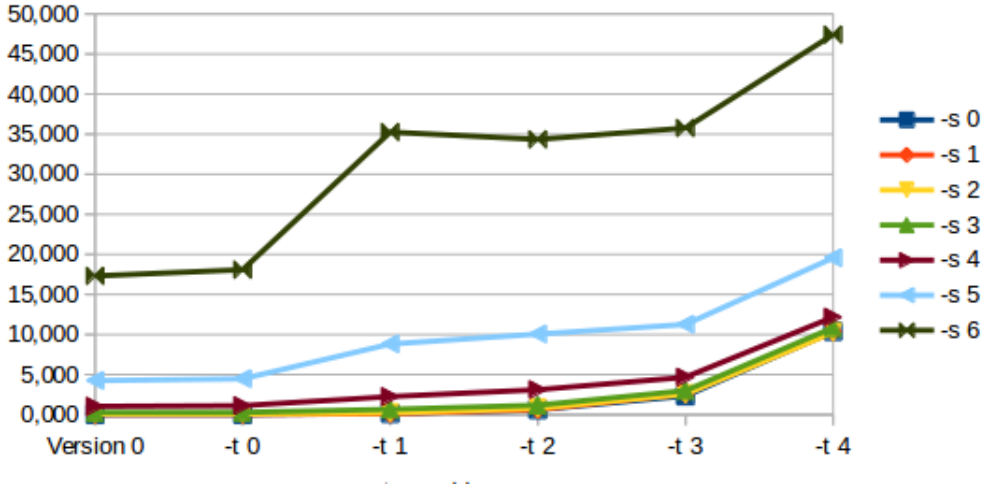

Figure 11: Temps CPU en fonction de la taille de matrice par option -t

Nous remarquons que le temps CPU est plus important que le temps utilisateur. Cela est du aux processus parallèles. Cela génère plus de tour d'horloge.

3.3 Version 2 - Mutex et Variables conditions

Pour la version 2, nous avons remplacé les barrières utilisées dans la Version 1 par des mutex et des variables conditions.
Le principe reste toujours le même mais nous utilisons d'autres fonctions.

Nous avons donc effectué quelques mesures pour juger de la différence.

nb thread/taille	-s 0	-s 1	-s 2	-s 3	-s 4	-s 5	-s 6
Version 0	0,005	0,023	0,071	0,271	1,041	4,244	17,32
-t 0	0,011	0,024	0,074	0,288	1,095	4,542	17,790
-t 1	0,044	0,056	0,079	0,184	0,622	2,688	10,190
-t 2	0,185	0,200	0,173	0,364	0,928	3,566	11,410
-t 3	0,739	0,827	0,695	0,614	1,256	4,124	14,979
-t 4	3,019	3,576	3,691	2,494	2,477	6,272	20,884

Figure 12: Tableau récapitulatif des mesures du temps réel obtenues (en seconde) Version 1

nb thread/taille	-s 0	-s 1	-s 2	-s 3	-s 4	-s 5	-s 6
Version 0	0,005	0,023	0,071	0,271	1,041	4,244	17,31
-t 0	0,011	0,024	0,074	0,288	1,095	4,542	17,780
-t 1	0,067	0,091	0,208	0,621	2,243	8,770	35,880
-t 2	0,262	0,329	0,414	0,926	2,784	10,192	37,397
-t 3	0,986	1,193	1,355	1,672	3,748	12,317	46,446
-t 4	3,805	4,975	5,506	5,617	7,642	19,644	66,380

Figure 13: Tableau récapitulatif des mesures du temps CPU obtenues (en seconde) Version 1

Nous pouvons observer des résultats, comme en version 1, que le temps réel est plus faible pour un nombre de threads entre 4 et 4**3. Il y a peu de différence pour une petite taille de matrice, mais cela est bien mis en valeur pour des grandes matrices. Donc l'utilisation de threads pour des grandes matrices est une bonne façon d'optimiser les calculs.
Par contre, lorsque le nombre de threads est beaucoup trop important par rapport au nombre de coeurs disponible, les résultats deviennent moins bons que pour le programme séquentiel. Cela provient du fait que la matrice est trop découpée, et que le système gaspille du temps à gérer chacune d'elles plutôt que de faire les calculs.

Si le nombre de thread optimal est de 4, cela est dû au fait que les tests ont été effectué sur une machine 4 coeurs.

En ce qui concerne le temps CPU, nous remarquons qu'il est proportionnel au nombre de threads. Cela est tout à fait normal, en effet le nombre de processus parallèle multiplie le nombre de tour d'horloge.

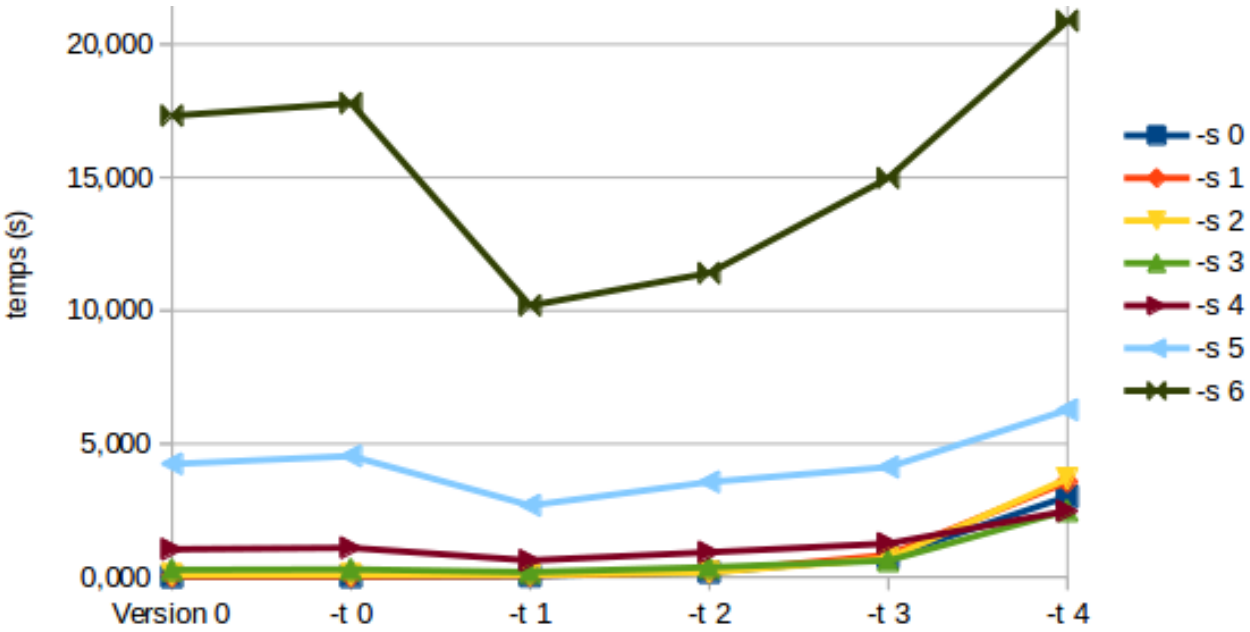

Figure 14: Temps CPU en fonction de la taille de matrice par option -t

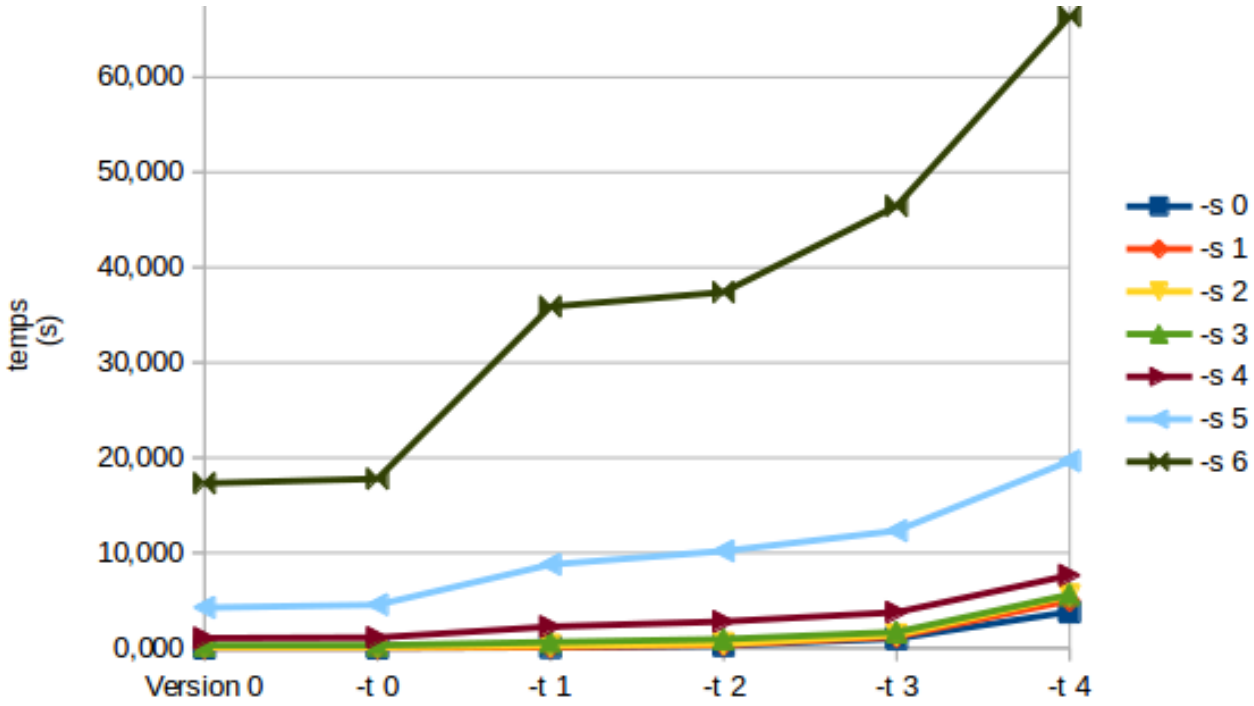

Figure 15: Temps CPU en fonction de la taille de matrice par option -t

Nous avons de plus fait des mesures sur l'empreinte mémoire des executions sur les différentes versions. La version 0 est similaire la ligne -t 0. Les versions 1 et 2 ont les mêmes empreintes.

L'empreinte mémoire commence à varier à partir d'une taille de matrice supérieure à 2**(4+3)

Empreinte	-s 0 1 2	-s 3	-s 4	-s 5	-s 6
-t 0	644	876	1 172	2 756	8 828
-t 1	916	916	1 444	3 024	9 116
-t 2	1 972	1 972	2 500	4 084	10 188
-t 3	5 932	5 932	6 460	8 044	14 148
-t 4	21 772	21 772	22 300	23 884	29 988

Figure 16: Empreinte memoire des executions

3.4 Version 1 VS Version 2

Pour une taille de matrice entre 2**(4) et 2**(4+4), la version 2 sont environ deux fois plus performante. En effet, pour **-s 2 et -t 3** par exemple, nous avons un temps réel de 0,6 s contre 1,5 s dans la version 1.
Les temps sont égaux pour une taille 2**(4+5) et deviennent pire au-delà. Par exemple, pour **-s 6 et -t 4**, nous avons un temps réel de 20s contre 18s dans la version 1.
Nous supposons que cela est dû au fait que comme les barrières POSIX sont implémentées d'une manière plus complète, elles doivent demander beaucoup plus de ressources. Le fait de simplifier en mutex et variables conditions demande moins de travail pour le programme.

4 Conclusion

Pour cette étape du projet, nous avons remplacé les barrières POSIX par des barrières construites à partir de variables conditions et d'exclusions mutuelles. Nous avons pu constater que le programme est plus performant en fonction de la taille de la matrice, allant jusqu'à 2-3 fois plus rapide. Pour la prochaine version, nous utiliserons des semaphores pour construire notre barrière.