0 - Compilation des fichiers.

```
la$ make
gcc     -c -o function.o function.c
gcc     -c -o main2D.o main2D.c
gcc     -c -o main1D.o main1D.c
gcc -g -DLINUX -fopenmp main1D.c function.c -o TP1D
gcc -g -DLINUX -fopenmp main2D.c function.c -o TP2D
```

Réussie !

1 - Equation de la chaleur (1D).

Il s'agit dans un premier temps de créer un code permettant de calculer l'équation de la chaleur sur un objet en 1D, puis de tenter une optimisation de l'exécution de son code en utilisant le parallélisme.

A - Performance sans parallélisme.

Voici la partie du code effectuant le calcul de l'équation de la chaleur.

```
N = (int) (t/dt); // here we compute the number of iterations
for (int k = 0; k<N; k++) // Chaque itération correspond à une itération de temps.
{
  // omp_set_num_threads(4); // A commenter/décommenter pour utiliser le parallélisme.
  // #pragma omp parallel for // idem. Parallélise les boucles sur le nombre de threads défini.
  for (i=0; i < size ; i++)   { // Chaque itération correspond à une itération d'espace.
      if (i==0)
          Tdt[i] = im[i] + dt*(im[i+1]-l1)/2; // gestion des bords.
      else if (i==size)
          Tdt[i] = im[i] + dt*(l2-im[i-1])/2; // gestion des bords.
      else
          Tdt[i] = im[i] + dt*(im[i+1]-im[i-1])/2;
  }

swap = Tdt;
Tdt = im;
im = swap;
}
```

Pour la gestion des bords, on utilise les limites définies comme suit :

```
float l1 = im[0];
float l2 = im[size];
```

Le temps de calcul sans optimisation est obtenu à l'exécution :

```
jason@jason-Aspire-A717-72G:/mec
la$ ./TP1D data.txt res.txt
temps  reel pour 1D : 3.656250
```

J'ai réalisé une moyenne sur 5 exécutions par la suite pour établir le temps d'exécution de référence pour le programme sans optimisation.

Ainsi, on définit T1D* = 3,648 secondes.

B - Performance avec parallélisme.

Les deux lignes commentées plus haut dans le code entre les deux boucles for sont les lignes de code permettant le parallélisme.

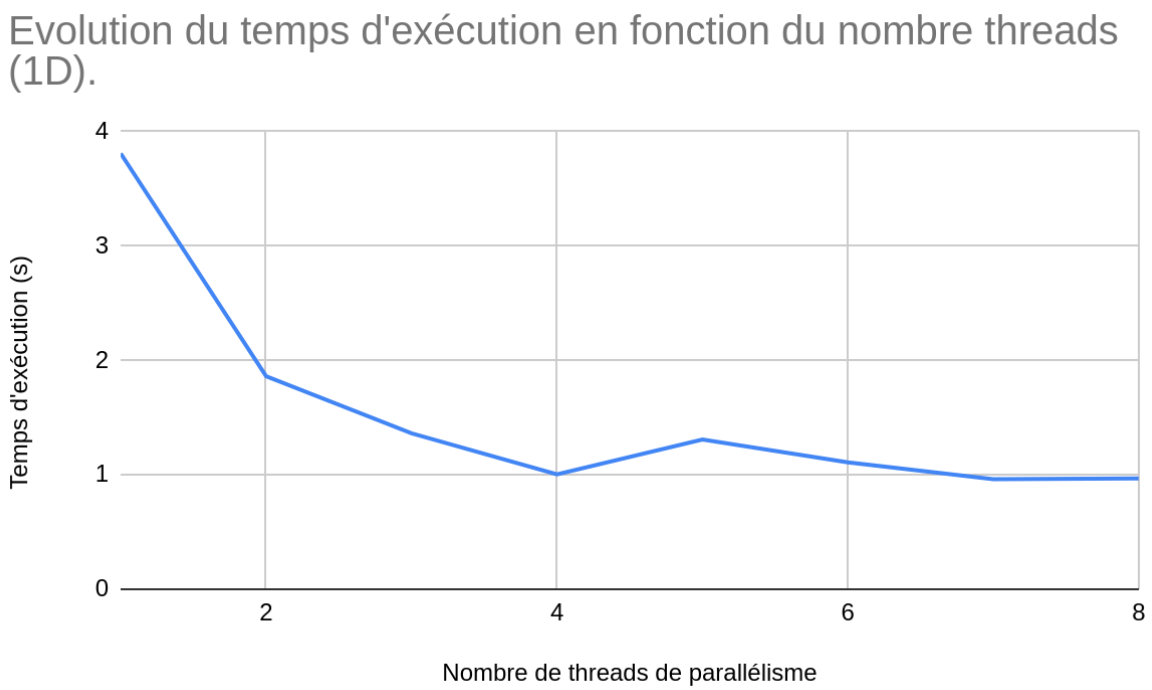
Ici, on se contente de paralléliser les boucles for sur la seule dimension spatiale, puisque sur une itération de temps, les calculs de chaque position sont indépendants.
Comme il n'y a qu'un calcul par boucle, aucune autre optimisation n'est nécessaire.

Ainsi, en lançant le programme sur une machine à 4 cœurs, on peut lancer 4 boucles simultanément à l'aide de 4 threads exécutés simultanément sur chacun un cœur.

Mon ordinateur possède 4 coeurs, mais peut utiliser de l'hyperthreading pour en simuler 8.

On définit sur 5 exécutions T1D1 = 3,809s.

J'ai donc réalisé les mesures de 1 à 8 coeurs utilisés :



Ainsi, on constate une baisse significative du temps d'exécution en passant de 1 à 4 cœurs.
Ce temps repart à la hausse en passant au 5ème cœur, puisque nous passons alors en hyperthreading.

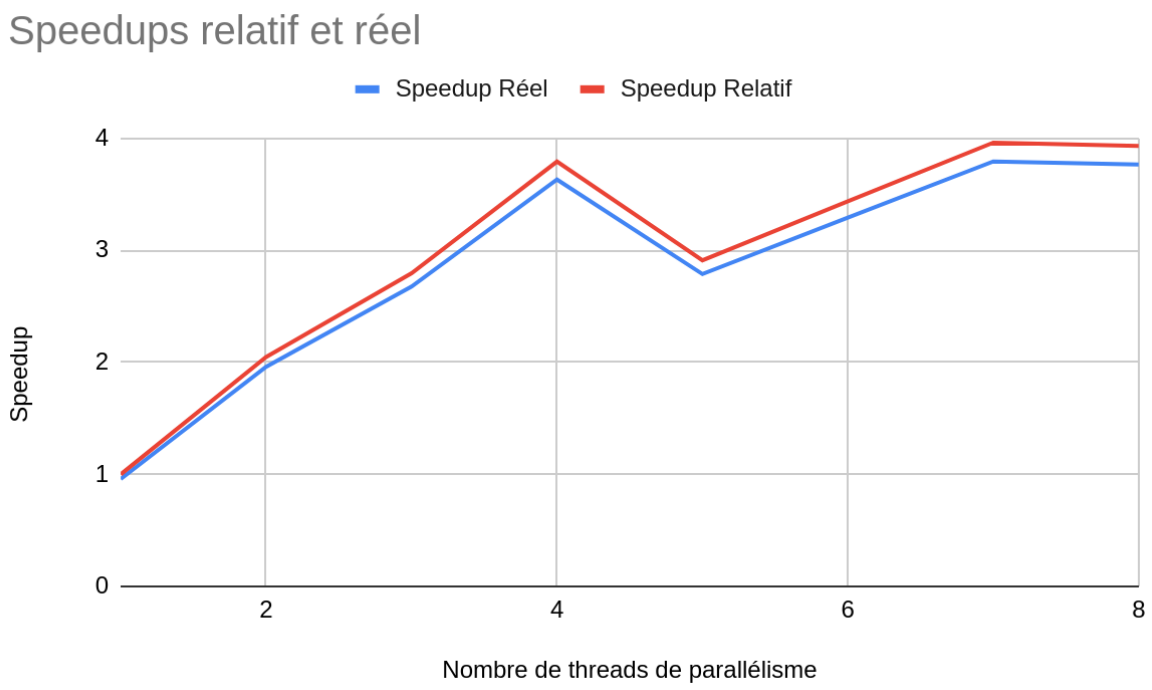
Le temps diminue alors à nouveau jusqu'à 7 cœurs, avant de légèrement augmenter en passant au 8ème.

Utiliser davantage de threads n'apportera pas d'amélioration, puisque il n'y a plus de cœurs disponibles pour éventuellement paralléliser une neuvième boucle.

C - Calcul de speedup.

On définit le speedup réel comme suit :
Sp(p) = Temps d'exécution séquentiel / Temps d'exécution parallèle sur p threads.

et le speedup relatif comme suit :
SRp(p) = Temps d'exécution sur 1 thread / Temps d'exécution sur p threads.



On remarque que le speedup relatif se situe en permanence au-dessus du speedup réel, ce qui indique que le temps d'exécution sur un thread est plus lent que le temps d'exécution séquentiel, sans optimisation.

2- Equation de la chaleur (2D).

Même chose, mais avec des objets en 2D.

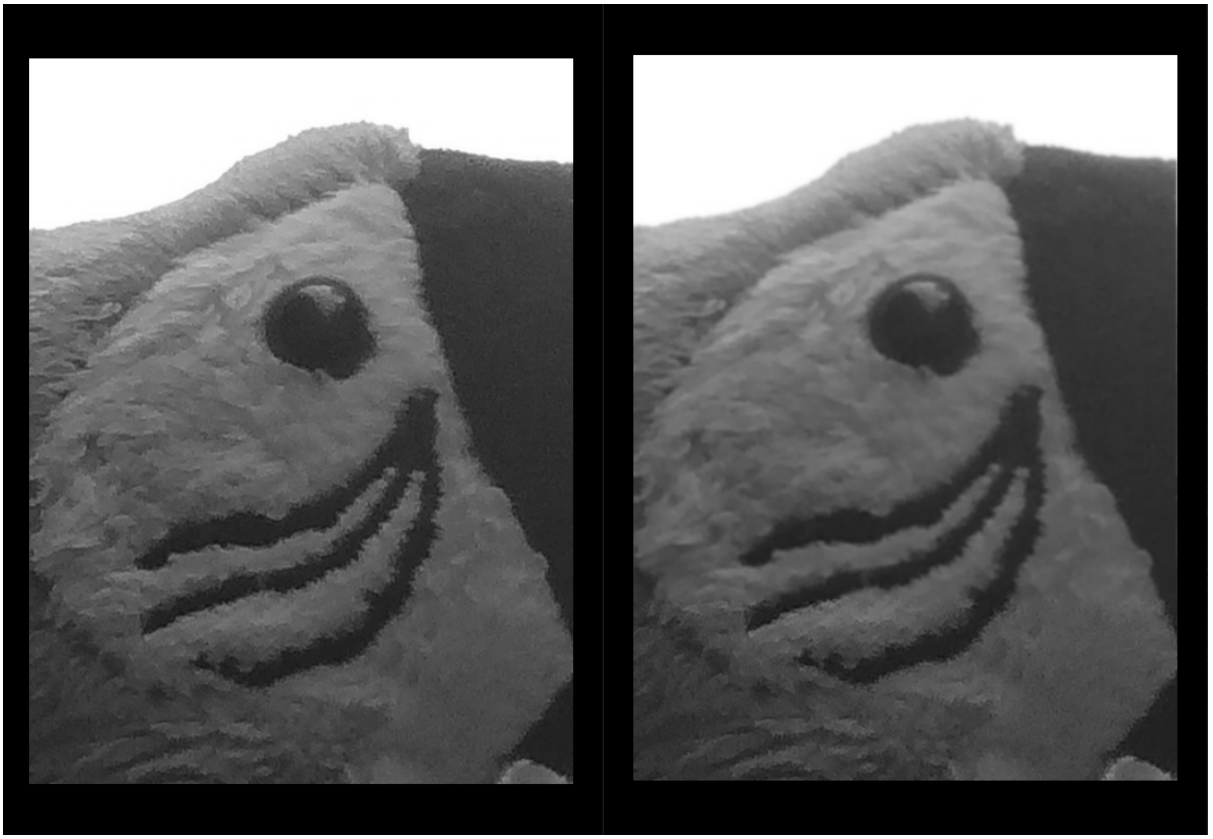
A - Performance sans parallélisme.

Voici le code réalisant le calcul de l'équation de la chaleur sur une image 2D :

```
for (int k = 0; k<N; k++) // Itérations temporelles
{
    // omp_set_num_threads(1);
    // #pragma omp parallel for collapse(2) private(cp, F, d2x, d2y, nn, sn, en, wn)
    for (int i = 0; i < rw ; i++) // Itérations spatiales sur les lignes.
    {
      for (int j = 0; j < cl ; j++) // Itérations spatiales sur les colonnes.
      {
        cp = T [j + i*rw];

        if (i == 0) // Dans le cas où on est sur un bord supérieur.
          nn = cp;
        else
          nn = T[j + (i-1)*rw];

        if (i == (rw-1)) // Dans le cas où on est sur un bord inférieur
          sn = cp;
        else
          sn = T[j + (i+1)*rw];

        if (j == 0) // Dans le cas où on est sur un bord gauche.
          wn = cp;
        else
          wn = T[(j-1) + i*rw];

        if (j == (cl-1)) // Dans le cas où on est sur un bord droit.
          en = cp;
        else
          en = T[(j+1) + i*rw];

        d2x = (en - 2*cp + wn)/(h*h); // Calcul de la composante horizontale.
        d2y = (nn - 2*cp + sn)/(h*h); // Calcul de la composante verticale.

        F = d2x + d2y; // Combinaison.

        Tdt[j + i*rw] = cp + dt*F; // Calcul de la valeur de l'equation de diffusion.
      }
    }

    swap = Tdt;
    Tdt = T;
    T = swap;
  }
```

Il est nécessaire de faire plusieurs tests au sein du calcul pour faire le traitement des bords.

N est défini tel que N = t/dt, où t=1s et dt = 0,001s.
Sur une petite image de test, voici le résultat de l'exécution, avec à gauche, l'image originale, et à droite le résultat, sur lequel on remarque un léger flou :



Voici une exécution de ce code, cette fois sur une image en HD :

La différence est plus subtile compte tenu de la plus haute définition de l’image, mais augmenter le temps sur lequel est calculée l'équation de la chaleur rend le temps d’exécution trop important pour faire beaucoup de mesures.
Voici le temps obtenu :

```
la$ ./TP2D papoch.pgm test2D.pgm
 temps  reel pour 2D 196.675781 :
```

Après une moyenne sur 5 exécutions, on définit T2D* = 196,76s.

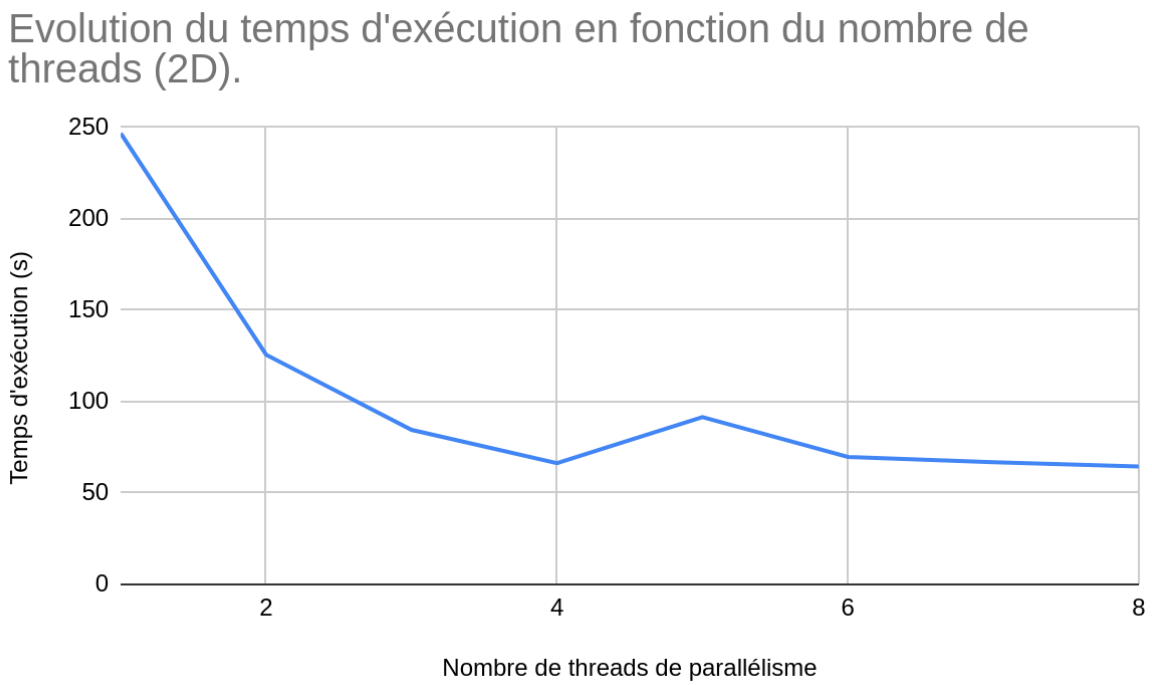
B - Performance en parallélisme.

En décommentant les lignes de code au début de la grande boucle temporelle, on accède au parallélisme d'OpenMP.

Il s'agit, comme précédemment, de paralléliser les boucles spatiales. Ici, on parallélise l'exécution des lignes, puisque on calcule toutes les colonnes pour chaque itération de ligne (d'où l'utilisation du collapse). Il est également nécessaire de privatiser les variables nécessaires au calcul, puisque chaque boucle utilise des variables intermédiaires avant de stocker la valeur finale dans l'image de sortie.

On définit sur une moyenne de 5 exécutions T2D1 = 246,78s.

Voici la courbe des temps d'exécution à différents nombres de threads utilisés.

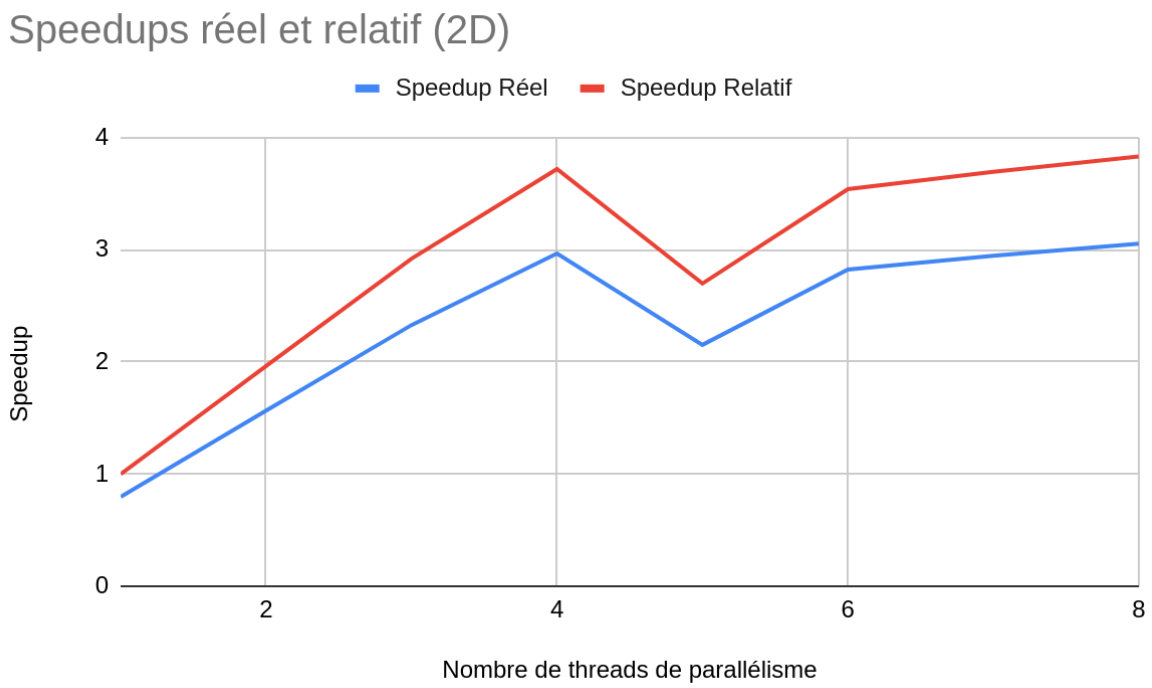


On remarque une évolution similaire par rapport au 2D, avec une amélioration du temps d'exécution jusqu'à 4 coeurs utilisés, une remontée au passage à 5 (hyper threading), puis une redescente, ou le temps d'exécution sur 8 coeurs est similaire au temps d'exécution sur 4.

A noter cependant que la dégression du temps d'exécution est moins marquée à mesure que l'on augmente le nombre de cœurs utilisés.

C - Speedups.

Voici les courbes de speedups obtenues, avec les mêmes formules utilisées plus haut.



On remarque que la différence entre les deux courbes est encore plus flagrante que précédemment, ce qui laisse entendre que l'algorithme n'est pas performant si il n'est exécuté que sur un seul thread.

Le speedup semble être maximum lorsqu'on utilise les 4 cœurs physiques, et s'amenuise quand on passe en hyper threading, conformément aux temps de calculs obtenus.

Conclusion :

On a observé l'efficacité du parallélisme sur le traitement de l'équation de la chaleur aussi bien en 1D qu'en 2D.

On souligne cependant l'inefficacité du parallélisme par rapport au séquentiel dans le cas où seul un seul thread est utilisé.

On notera également la baisse de l'efficacité du parallélisme une fois que l'on passe en hyper threading, c'est-à-dire une fois que l'on dépasse le nombre de cœurs physiques.