DIAS Nicolas 21802856

M1 CHPS 2021/2022

Techniques d'Optimisation de la Parallélisation

Rapport d'optimisation de la simulation

Introduction

Dans ce rapport je vais vous présenter la simulation de tourbillons de Karman qui a été donné comme projet. Je vais vous présenter comment le programme a été debugger et optimiser tout en gardant un résultat juste. Aujourd'hui optimisé un code est très important pour plusieurs chose, dans un premier temps cela permet de gagner du temps et dans un second temps on fait des économies sur la consommation électrique. C'est encore plus important avec des simulations étant donné qu'une simulation peut prendre plusieurs jours voir plusieurs semaines à s'exécuter donc gagner ne serait-ce que 5% de performance est déjà beaucoup.

Outils et bibliothèques utilisé sur le programme :

Pour debugger le programme et l'améliorer différents programme et bibliothèques ont été utilisé :

-GDB : Cet outil permet de gérer les problèmes de compilation. Pour qu'il fonction il faut rajouter l'option « -g » lors de la compilation et il suffit de faire « gdb prog » avec prog le nom du programme à debugger, une fois ceci fait il suffit d'utiliser les options de gdb dont on a besoin. Il faut utiliser GDB qui est un debugger ce qui nous permet, si nécessaire, de trouver les problèmes lors de l'exécution tel que les erreurs segmentations en donnant la fonction dans laquelle c'est arrivé ainsi que la variable qui a causé ce problème. Avec GDB il est également possible de récupérer les fonctions qui ont été appelées par le programme pour bien voir ce qui est arrivé dans le programme avant qu'un problème arrive, cette fonctionnalité peut être pratique pour comprendre pourquoi un programme reste bloqué en nous donnant les dernière fonctions utilisées et donc celle qui bloque. GDB est donc très pratique pour comprendre tout ce qui est arrivé dans le programme et ainsi le debugger.

-Gprof : Cet outil permet de profiler un code, il suffit de rajouter l'option « -pg » dans la compilation puis de lance l'outil en faisant « Gprof prog » prog étant le nom du programme à profiler. Une fois cela fait Gprof donnera différentes informations sur le programme, parmi toutes les informations la plus intéressante est le fait que l'outil nous donne le temps passé dans chaque fonction ce qui nous permet de savoir les fonctions qui prennent trop de temps et donc celles qu'il faut optimiser en priorité.

-Interpol : Cet outil permet d'observer tous les appels MPI effectué par le programme c'est-à-dire tous les envois et toutes les réceptions, quel rang a fait quoi, le temps qu'a pris chaque appel, la quantité de donnée échangé ainsi que d'autres informations. Pour l'utiliser il suffit de télécharger le code sur github, il s'agit d'une bibliothèque d'interposition donc il faut l'utiliser sur le programme puis utiliser l'interface graphique sur le résultat. En ayant accès à toutes ces informations il est plus

facile d'optimiser tous les échanges en pouvant observer les rangs qui sont bloqué trop longtemps sans faire de calcul. Avec cet outil on peut donc voir si des échanges MPI sont « mauvais » et qui peuvent être amélioré.

-MPI qui permet de faire de la communication entre processus et ainsi de paralléliser un code. Cette bibliothèque est utilisée de base dans le programme. Dans ce projet openmpi a été utilisé pour compiler et exécuter le programme.

-Openmp qui permet de créer des threads et donc de paralléliser le programme ou certaines partie du programme.

Première idée sur la scalabilité du programme :

Apres un premier regard sur le programme pour comprendre son fonctionnement on peut se rendre compte que quelques parti sont optimisables principalement certaines boucles et les appels MPI. Le programme semble avoir une scalabilité élevé.

```
for( y = 0 ; y < mesh->height-2 ; y++ )
        MPI_Send( &Mesh_get_col( mesh_to_process, x )[y], DIRECTIONS, MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
```

En effet quand on regarde les échanges MPI il est possible de se rendre compte que les envois et les réceptions ne sont pas optimal du fait que des données sont envoyés une à une au lien d'envoyer des tableaux entier. Ces échanges se font donc dans des boucles ce qui ralentit le programme.

Programme et communications d'origine

Le programme est organisé en différents fichiers, tout d'abord le code est séparé en différents .c qui sont relié à leur header(.h) qui contiennent le code de la simulation, « display.c » est utilisé par les script, « lbm_phys .c » permet de faire les calculs des forces qui s'appliquent aux particules, « lbm_comm.c » gère les communications du programme, « lbm_config .c» s'occupe de la config de base qui doit être utilisé par le programme, « lbm_struct.c » gère les structures de données du programme, « lbm_init .c» permet d'initialiser la configuration de base et « main.c » qui s'occupe d'appeler les fonctions pour un bon fonctionnement du programme, le fichier « usleep.c » ne sert à rien et n'est pas utilisé dans le programme. Des script sont présent afin de vérifier que nos résultats dont juste en créant un gif de la simulation qui a été calculé et enfin un fichier « config.txt » qui permet de modifier les informations de base de la simulation.

Quand on regarde le fonctionnement des communications du programme on voit qu'à chaque itération chaque particule doit envoyer des données à chaque point autour de lui, il doit donc faire 9 envois et 9 réception étant donné que chaque point à 9 voisins direct. En regardant le nombre d'envoi et de réception aux voisins on remarque que 10 envois et réception sont effectués alors que seul 9 suffisent. On peut donc conclure qu'il y'a des communications en trop. On voit également qu'à la fin du programme chaque processus MPI envoi les informations qui doivent être écrite dans un fichier au processus maitre pour qu'il écrive les informations de tout le monde. On peut remarquer que la réception du processus maitre reçoit les informations avec une boucle de réception. On remarque aussi que après chaque appel MPI une barrière est appelé pour que les processus s'attendent, cela ne parait pas nécessaire étant donné que tous les appels sont déjà bloquant et que par conséquent les processus doivent déjà s'attendre entre eux pour avancer. A première vu il parait donc possible de supprimer les barrières. Le programme semble pouvoir être bien optimisé sans

modifier les calculs fait par le programme. En effet si on modifie bien les échanges MPI et que l'on parallélise des boucles les calculs restent exactement les mêmes.

Changement du code :

Pour améliorer le programme il a fallu faire différent changement dans le code afin de le faire fonctionné et d'accélérer son exécution.

Définition multiple :

Tout d'abord lors de la compilation du programme un problème va apparaitre qui nous dit que certaines variables sont définies plusieurs fois, le compilateur nous dit que cela vient du fichier « lbm_phys.h » aux lignes 9,10 et11. Une fois dans le fichier on voit que les variables qui posent problème sont des variables globales. Pour gérer le problème on rajoute le mot clé extern devant chacune des variables pour dire au compilateur qu'elles sont utilisées ailleurs.

Erreur segmentation :

Maintenant on lance le programme et on voit qu'il y a une erreur segmentation. Cette erreur étant dû à un problème de gestion de la mémoire il faut chercher quelle variable cause ce problème. Pour cela il a fallu utiliser un débugger tel que gdb pour nous permettre de suivre ce qu'il s'est passé. En utilisant la fonctionnalité backtrace, il est ainsi possible de voir qu'une variable a posé ce problème.

```
 RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 ) (WH 802 162 )

Thread 1 "lbm" received signal SIGSEGV, Segmentation fault.
0x0000555555556cbf in setup_init_state_global_poiseuille_profile (mesh=0x7fffffffdd00, mesh_type=0x7fffffffdd30,
    mesh_comm=0x7fffffffdd40) at lbm_init.c:85
85                                      Mesh_get_cell(mesh, i, j)[k] = compute_equilibrium_profile(v,density,k);
(gdb) bt
#0  0x0000555555556cbf in setup_init_state_global_poiseuille_profile (mesh=0x7fffffffdd00, mesh_type=0x7fffffffdd30,
    mesh_comm=0x7fffffffdd40) at lbm_init.c:85
#1  0x0000555555556fa8 in setup_init_state (mesh=0x7fffffffdd00, mesh_type=0x7fffffffdd30, mesh_comm=0x7fffffffdd40)
    at lbm_init.c:155
#2  0x000055555555acb in main (argc=1, argv=0x7fffffffdf98) at main.c:159
(gdb) print * mesh_type
$1 = {types = 0x7ffff519e010, width = 802, height = 162}
(gdb) print* mesh_comm
$2 = {x = 0, y = 0, width = 802, height = 162, nb_x = 1, nb_y = 1, right_id = -1, left_id = -1, top_id = -1,
  bottom_id = -1, corner_id = {-1, -1, -1, -1}, requests = {0x7ffff7d2db80 <main_arena>, 0x280, 0xfffffffffffffb0,
    0x290, 0xfffffffffffffb0, 0x27, 0x7ffff7bd97c3 <_int_malloc+3363>, 0x0, 0x0, 0x0, 0x2372e6746897fb00, 0x280,
    0x7ffff7c62e80 <__profil+320>, 0x0, 0x2710, 0x0, 0x2710, 0x7ffff7c62cf0 <__profil_counter>, 0xffffffffffffffff,
    0xffffffffffffffff, 0xffffffffffffffff, 0xffffffffffffffff, 0xffffffffffffffff, 0xffffffffffffffff,
    0xffffffffffffffff, 0xffffffffffffffff, 0xffffffffffffffff, 0xffffffffffffffff, 0x555555555016 <_init+22>,
    0xffffffffffffffff, 0x5555555587bd <__libc_csu_init+77>, 0x7ffff7d322e8 <__exit_funcs_lock>}, buffer = 0x0}
(gdb) print *mesh
$3 = {cells = 0x0, width = 802, height = 162}
(gdb)
```

Avec gdb il est possible d'afficher l'adresse d'un pointeur (print *var) pour voir si celui-ci est NULL et en effet quand on regarde le pointeur des variables qui peuvent poser problème on se rend compte que le pointeur « mesh » pointe vers l'adresse 0 qui correspond à NULL alors que les «mesh_comm » et « mesh_type » sont des pointeurs avec des valeurs, on peut donc en conclure que le problème vient de la variable « mesh » et pas des autres. Il suffit donc de chercher cette variable dans le code afin de régler le problème. On voit que le malloc de cette variable a été mis en commentaire et le pointeur remplacé par NULL, le programme essayant d'utilisé la mémoire soit disant alloué à la variable, une erreur va se produire étant donné que la variable n'a aucune mémoire alloué. Ce

problème est géré en remettant le malloc et en supprimant la mise à NULL du pointeur, en effet la variable aura de nouveaux accès à la mémoire nécessaire pour l'exécution du programme.

Deadlock :

Une fois l'erreur segmentation supprimé le programme peut être lancé de nouveaux, un autre problème se présente tout d'abord, le programme est très lent il faut donc réduire le nombre d'itération pour essayer de le terminé. Mais une fois arrivé à la dernière itération le programme ne se termine pas et reste bloqué, ce problème est généralement dû à un processus qui n'arrive pas à se terminer cela peut venir de différentes choses tel qu'une boucle infini ou une barrière de laquelle le processus reste bloqué. Il faut donc de nouveau utiliser gdb pour comprendre ce qu'il s'est passé. Avec gdb il est possible d'utiliser le backtrace, le backtrace permet de voir toutes les fonctions qui on précédemment utilisé avant le crash ce qui permet de voir de quelle fonction vient le problème et même la variable qui l'a causé. Il suffit ensuite de retrouvé la variable dans le programme pour comprendre ce qu'il se passe. On observe si on lance gdb avec mpi que l'un des processus va jusqu'à « MPI_Finalize » et l'autre est bloqué dans une « MPI_Barrier » et on retrouve la fonction dans laquelle cette barrière est appelé. Pour régler ce problème il y a deux solutions :

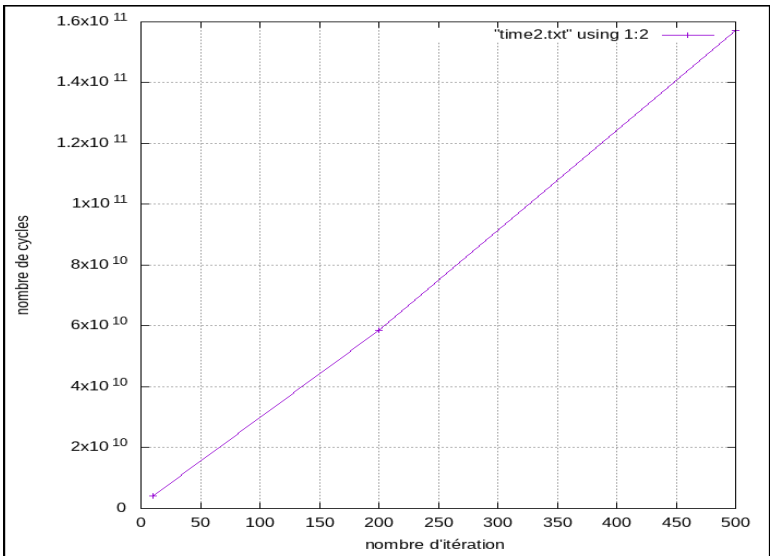
-Soit il faut rajouter une barrière à tout le monde pour que tout le monde s'arrête puis continue.
-La deuxième solution est de tout retirer la barrière pour le processus maitre ainsi il ne s'arrêtera plus.

La deuxième solution est celle que j'ai appliqué car à ce moment du programme le but est uniquement de fermé un fichier et de terminer le programme il n'est donc pas nécessaire que tous les processus s'attendent. Une fois la barrière supprimé le programme fonctionne jusqu'à la fin sans problème car le processus n'est plus bloqué.

Le programme est désormais fonctionnel. Mais si on lance le programme on se rend compte que le programme est très lent, en effet pour pouvoir essayer de le terminé il est nécessaire de réduire le nombre d'itérations vers 10, et avec une mesure du nombre de cycle on trouve que le programme de base prend 1575565106 cycles à s'exécuter.

Sleep :

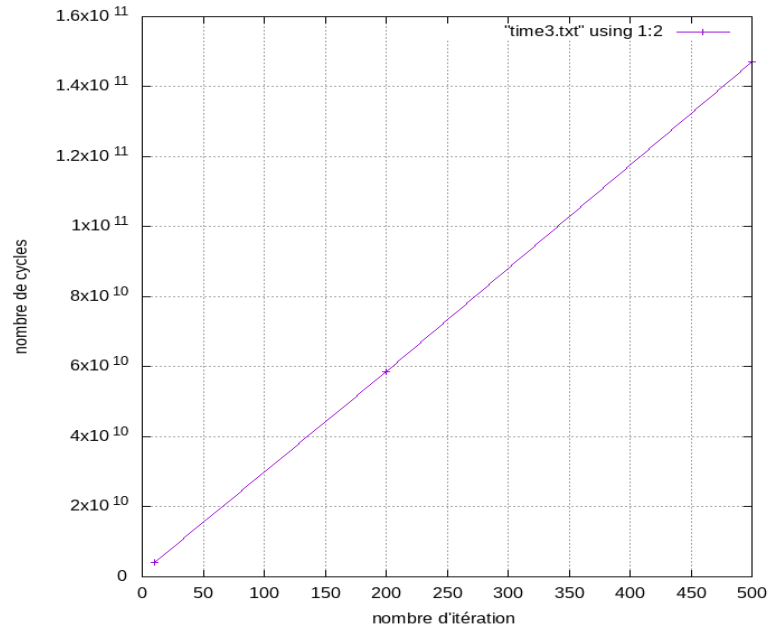
Une fois le programme fonctionnel il faut comprendre pourquoi le programme est si lent. En utilisant gdb il est possible de faire un backtrace c'est-à-dire afficher les fonctions qui sont appelé dans le programme, de cette manière il est possible de se rendre compte qu'un sleep est utilisé plusieurs fois dans le programme, le sleep est appelé autant de fois qu'il y a d'itérations dans le programme, il est donc forcément dans une fonction de la boucles du main. Le sleep est un problème car le programme sera obligé de se mettre en pause pendant une seconde à chaque itérations ce qui ralenti grandement le programme. On trouve donc que le programme appel à chaque itération dans la boucle la fonction « lbm_comm_ghost_exchange » on cherche donc dans cette fonction ou se trouve le sleep. En regardant les fonctions « lbm_comm_sync_ghosts_horizontal » , « lbm_comm_sync_ghosts_vertical » et « lbm_comm_sync_ghosts_diagonale » on voit qu'aucune de ces fonction n'utilise de sleep, il reste néanmoins une fonction a la fin du nom de « FLUSH_INOUT » en cherchant ou trouver cette fonction on voit qu'elle appel en fait un sleep qui a été définie dans un headers. On supprime donc cette fonction qui n'a aucune utilité au programme à part le ralentir.



On voit ici le nombre de cycle pris pour faire 16 itérations, puis 200 et enfin 500, on s'arrête à 500 itération parce que le programme reste lent mais fonctionne malgré tout plus vite que lorsque le sleep est présent. Le programme étant maintenant fonctionnel et s'exécutant à une vitesse convenable il est enfin possible de l'optimiser.

Changement de l'affichage :

Afin d'avoir un affichage plus lisible, l'affichage des itérations a été modifié, pour éviter qu'à chaque itérations on affiche l'itération actuel, on remplace le « \n » du printf par un « \r » ce qui remplace à chaque fois la ligne écrite précédemment. Il se trouve que ce changement a provoqué un changement de performance léger mais présent, en effet le programme a accéléré. L'une des hypothèses qui peut expliquer cette amélioration est qu'avec le « \ r » le printf fonctionnant avec un buffer n'écrit pas à toutes les itérations mais uniquement quand le buffer est plein, il fait par conséquent moins d'affichage



Cette mesure de performance a été faite sur un knl il se trouve que sur un pc portable la différence de performance est encore plus visible.

Utilisation gprof :

Après les quelques modifications déjà apporté au programme on va maintenant utiliser gprof afin de savoir quelle sont les fonctions dans lesquelles le programme passe le plus de temps et donc les fonctions à optimiser.

Suppression des barrières :

Une fois que le programme fonctionne de manière convenable, il faut maintenant essayer de l'optimiser pour cela on utilise gprof qui va nous donner les fonctions dans lesquelles on passe le plus de temps, ces fonctions seront évidemment celle qu'il faut optimiser en priorité. Logiquement pour que le programme gagne en performance il est plus intéressant d'optimisé une fonction qui est appelé plusieurs fois et dans laquelle on passe beaucoup de temps plutôt qu'une fonction qui n'est appelé qu'une seule fois. Il y a plusieurs fonctions qui ressortent, tout d'abord la fonction qui fait tous les envois et toutes les réceptions avec MPI ainsi que les fonctions de calcul des collisions et de propagations de chaque élément de la simulation. Pour commencer j'ai fait le choix de regarder les fonctions d'envoi et de réception MPI. En regardant la fonction « lbm_comm_ghost_exchange » qui s'occupe d'appeler les fonctions qui font les envois et les réceptions, on remarque qu'il y a énormément de « MPI_Barrier ». Les barrières vont obliger les processus à tous s'arrêter à cette barrière jusqu'à ce que tous y soit arrivé. Or dans notre cas comme tous les envois et toutes les réceptions sont bloquantes, les processus vont déjà s'attendre les uns les autres, en effet tant qu'une réception n'est pas faite, l'envoi bloquant va stopper le programme jusqu'à ce que la réception soit faite. Les processus sont donc obligé d'attendre que quelqu'un reçoive leur envoi ou ils vont attendre un envoi pour terminer leur réception, il n'est donc pas nécessaire de rajouter des barrières en plus qui vont ralentir le programme vu que les processus devront faire une étape supplémentaire d'attente qui n'est pas nécessaire. Toutes les barrières dans la fonction « lbm_comm_ghost_exchange » ont donc été supprimées.

Amélioration des envois et réceptions MPI :

Une fois les barrières supprimées il faut regarder comment fonctionnent les échanges MPI qui sont utilisés. On remarque que les envois et réceptions se trouvent à l'intérieur de boucles et que les valeurs qui sont échangées ne sont pas des tableaux en entier mais les valeurs du tableau qui sont envoyé une à une. Le problème étant que avec MPI il est beaucoup plus efficace de faire une seule communication avec beaucoup de données d'un seul coup plutôt que faire plusieurs échanges d'un petit nombre de donnée. Nos échanges seront donc lent dû au fait que l'on fait plus d'envoi que nécessaire.

```
for( y = 0 ; y < mesh->height-2 ; y++ )
        MPI_Send( &Mesh_get_col( mesh_to_process, x )[y], DIRECTIONS, MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
```

On voit bien ici que ce qui est envoyé n'est pas directement le tableau « Mesg_get_col » mais bien les valeurs de chaque case de ce tableau. Il est possible d'optimiser cet envoi en supprimant la boucle et en envoyant le tableau entier à la place pour cela on retire fait de récupérer une case du tableau et on augmente le nombre de données que l'on envoi.

```
MPI_Send( Mesh_get_col( mesh_to_process, x ), DIRECTIONS*(mesh->height-2), MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
```

La taille devient donc DIRECTIONS multiplié par la taille de la boucle qui a été supprimé.

Comme l'envoi de données a été modifié il faut également modifier la réception afin qu'elle corresponde à ce qui est envoyé. La modification est la même, on supprime la boucle on donne le pointeur vers le tableau et on modifie la taille de ce que l'on reçoit.

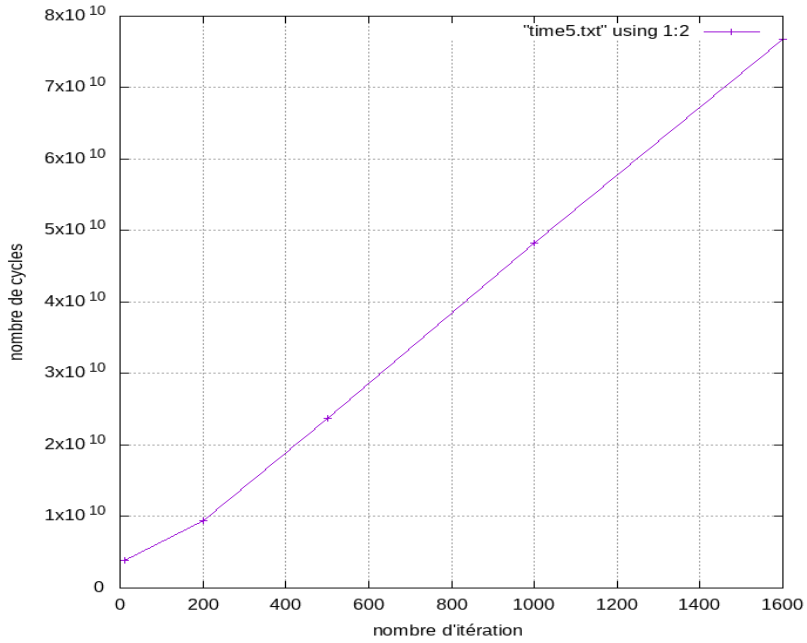
Il y a un problème similaire dans la fonction « lbm_comm_sync_ghosts_vertical »

```
for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
        for ( k = 0 ; k < DIRECTIONS ; k++)
                MPI_Send( &Mesh_get_cell(mesh_to_process, x, y)[k], 1, MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
```

Ici aussi il faut supprimé l'une des boucle mais on supprimera que la boucle intérieur afin d'envoyer le tableau entier. La boucle k sera donc supprimé et la taille du message passera de 1 à DIRECTIONS.

```
for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
        MPI_Send( Mesh_get_cell(mesh_to_process, x, y), DIRECTIONS, MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
```

Il reste malgré tout une boucle qui est plus complexe à supprimé, la réception est modifié de la même manière afin de correspondre à ce qui est envoyé. Les changements apporté sont déjà intéressant car faire un échange de beaucoup de donnée une seule fois est bien plus efficace que de faire plusieurs petits envoi, il en va de même pour la réception.



On voit ici le nombre de cycles que le programme à pris pour s'exécuter sur 16,200,500,1000 et 1600 itérations on voit bien que pour 200 et 500 itérations le nombre de cycles pris par le programme est beaucoup plus faible que précédemment cette optimisation est donc très efficace. Cela est dû au fait que les échanges MPI prennent beaucoup de temps et qu'en retirer une partie pour les faires en une seule fois est bien plus efficace et accélère donc le programme.

Echanges qui ne sont pas nécessaire :

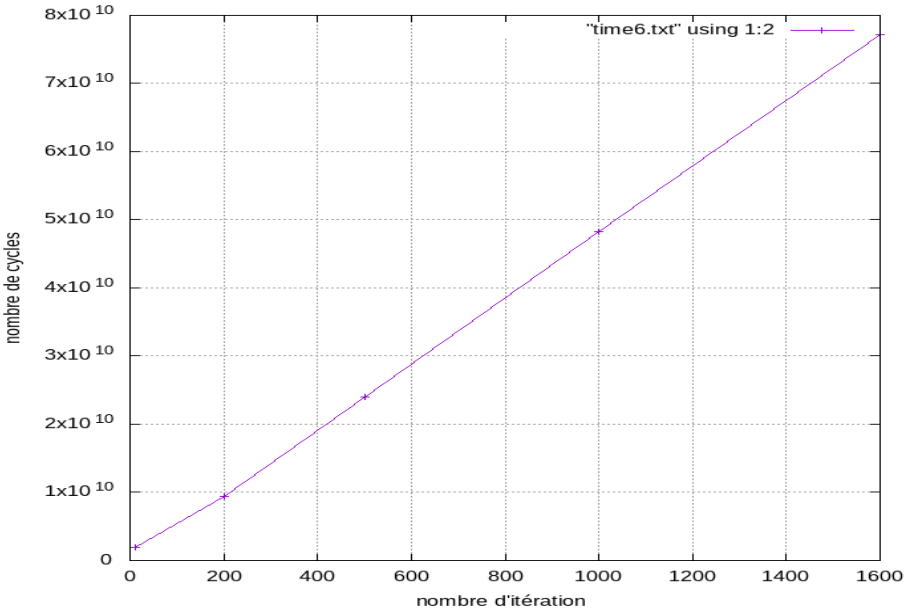
Une fois les échanges de données amélioré il faut se pencher sur le nombre de fonctions appelées pour faire tout les échanges afin de vérifié qu'il n'y ai pas trop de fonctions qui soient appelé. En

regardant de plus près ce qu'il se passe, il est possible de se rendre compte que certaines fonctions sont appelées mais qui ne font rien. En effet les échanges sont fait de manière à ce que chaque particule envoi des données à chacun de ses 9 voisins direct. Comme il faut faire un appel pour un envois et un appel pour un réception, on peut conclure qu'il faut au total 18 appels aux fonctions de communications. Or si on compte bien il y a 20 appel, il y a donc deux appel en trop.

```
// Right to left phase : on reçoit à gauche et on envoie depuis la droite
//lbm_comm_sync_ghosts_horizontal(mesh,mesh_to_process,COMM_SEND,mesh->left_id,1);
//lbm_comm_sync_ghosts_horizontal(mesh,mesh_to_process,COMM_RECV,mesh->right_id,mesh->width - 1);
```

En effet ces appels ligne 311 qui ont été mis en commentaires sont en fait les même que ceux fait en deuxième dans le code à la ligne 253, comme les deux appels sont les mêmes on peut en supprimer l'un des deux car ce sont des appels en trop qui ne sont pas nécessaire au fonctionnement du programme.

Après avoir supprimé cet appel inutile, on utilise l'outil interpol pour observer le fonctionnement des échanges MPI éffectué par le programme. On remarque que toutes les communications qui ont lieu dans ce programme on en tout 2 tag différent. Les tags qui apparaissent sont les tags 3 et 4, en regardant dans le programme à quoi correspond ces tags, on voit que cela correspond au envois de la fonction « lbm_comm_ghosts_vertical », il y a deux tags différents pour l'envoi en haut et l'envoi en bas. On en conclu que les autres fonctions peuvent être supprimé car de toute façon le programme ne les appels pas.



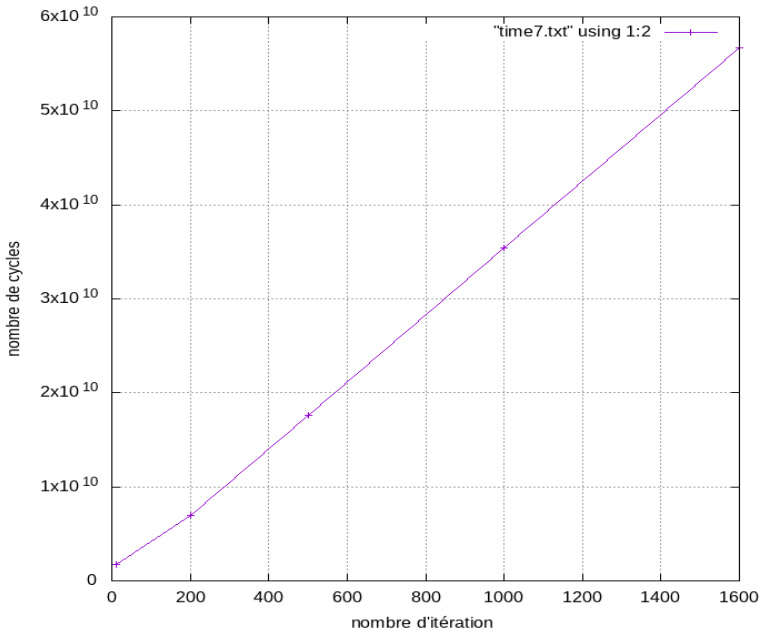
Apres avoir retiré les échanges qui ne font rien on voit que le programme n'a pas accéléré ce qui est logique étant donné que ces appel n'etaient de toute façon pas fait.

Echanges de vertical vers horizontal :

Cette simulation peut être découpé de deux manières différentes, elle peut être découpé verticalement ou horizontalement. De base la simulation est découpé horizontalement

```
//compute splitting
nb_y = lbm_helper_pgcd(comm_size,width);
nb_x = comm_size / nb_y;
//nb_x = lbm_helper_pgcd(comm_size,width);
//nb_y = comm_size / nb_x;
```

En mettant en commentaire les deux premières lignes et en décommentant les deux dernière on peut passer la simulation en découpage vertical, il faut également remplacer le « comm_size,width » de la troisième ligne par « comm_size,height » de cette manière le découpage de la simulation sera modifié. Les envois verticaux étant dans une boucle, il y'en a plus qui sont fait et ils prennent donc plus de temps à se faire en entier. En changeant le découpage de la simulation on peut donc utilisé uniquement les échanges horizontaux qui sont plus efficaces car il n'y a qu'un seul envoi et une seule réception.



Contrairement à l'optimisation précédente on à un gain plutôt conséquent on peut donc en conclure que cette optimisation est éfficace. Cela est dû au fait que lorsque l'on change la séparation de la simulation on ne fait que des échanges horizontaux et plus des échanges verticaux, les échanges horizontaux sont meilleur car il n'y a pas de boucles à l'intérieur et donc il y a moins de données échangées et le programme va plus vite.

Boucles des collisions et de la propagation :

Une fois les échanges MPI amélioré il faut se pencher sur les autres fonctions qui prennent le plus de temps qui sont le calcul des collision et le calcul de la propagation des particules. Ces deux fonctions utilisent des boucles imbriquées il faut donc se demander si les boucles sont implémenter de manière efficaces, en effet en C le stockage se fait en ROW_MAJOR ce qui veux dire que les données sont rangé en ligne dans la mémoire et non pas en colonne comme en Fortran. Pour lire de manière optimiser la lecture en mémoire des données il est donc important de lire le maximum de données possible en ligne et non pas en colonne. Dans notre cas un tableau à deux dimensions rangé en une seule dimension, mais si on regarde la manière de se déplacé à l'intérieur « (i * mesh->height + j) * DIRECTIONS » on comprend que le x est utilisé pour se déplacé dans les colonnes et le y pour se

déplacer sur la ligne, il faut donc incrémenter le y en premier sion souhaite juste récupérer les valeurs du tableau, ce qui évite de chargé une cache line qui nous sert pour plusieurs valeurs et que l'on ne devra pas chargé plusieurs fois.

Pour être plus clair voici un exemple en imaginant une cache line de taille 3:

0	1	2	3	4	5	6	7	8	9

Imaginons ici qu'on veux lire les cases une à une si on fait « (i * mesh->height + j) * DIRECTIONS » et qu'on augment le i en premier on commencer par la case 0 ce qui va chargé en cache line le 0, le1 et le 2 sauf que si on augment i au lieu de lire la case qui contient 1 on va lire la case 2 et avancé de deux en deux, il faudra ensuite revenir en arrière pour lire la case contenant 1 et donc potentiellement rechargé la même cache line que précedemment.

```
for( j = 1 ; j < mesh_in->height - 1 ; j++)
       for( i = 1 ; i < mesh_in->width - 1 ; i++ )
              compute_cell_collision(Mesh_get_cell(mesh_out, i, j),Mesh_get_cell(mesh_in, i, j));
```

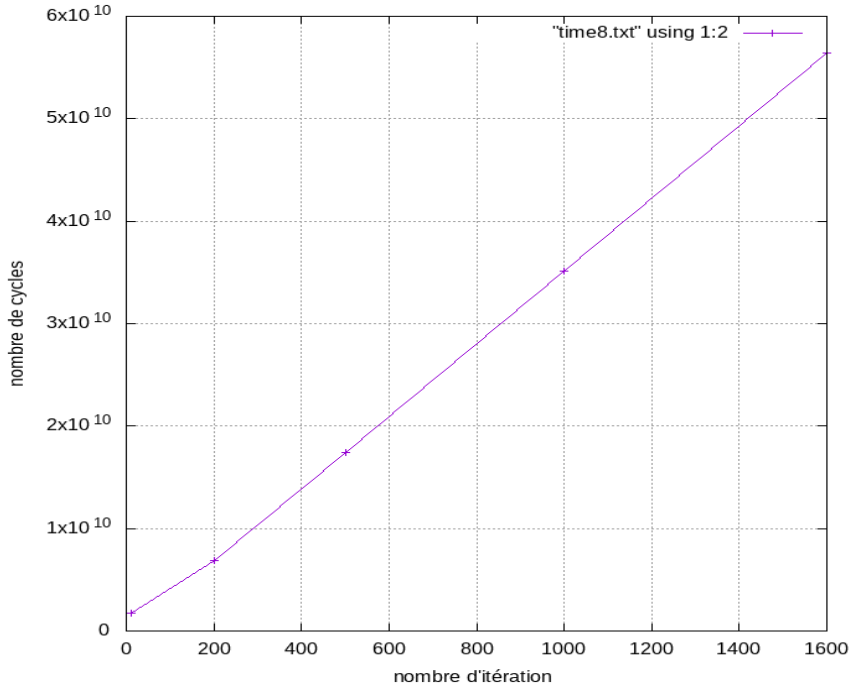
Comme on le voit ici dans le cas de base on modifie le i en premier et apres le y on se déplace donc dans le tableau d'une manière qui n'est pas optimale. Afin d'optimiser cette boucle il suffit donc d'echanger les deux boucles ainsi on incrémentera le j en premier puis le i et le déplacement dans le tableau permettra de faire beaucoup moins de chargement mémoire.

```
for( i = 1 ; i < mesh_in->width - 1 ; i++ )
    for( j = 1 ; j < mesh_in->height - 1 ; j++)
        compute_cell_collision(Mesh_get_cell(mesh_out, i, j),Mesh_get_cell(mesh_in, i, j));
```

Dans le cas de la fonction propagation on fait le même changement et on ne touche pas à la boucle k

```
for ( i = 0 ; i < mesh_out->width; i++)
{
    for ( j = 0 ; j < mesh_out->height ; j++)
    {
        //for all direction
        for ( k  = 0 ; k < DIRECTIONS ; k++)
        {
            //compute destination point
            ii = (i + direction_matrix[k][0]);
            jj = (j + direction_matrix[k][1]);
            //propagate to neighboor nodes
            if ((ii >= 0 && ii < mesh_out->width) && (jj >= 0 && jj < mesh_out->height))
                Mesh_get_cell(mesh_out, ii, jj)[k] = Mesh_get_cell(mesh_in, i, j)[k];
        }
    }
}
```

Pour la même raison que précédemment on inverse les boucles i et j afin de se déplacé de manière optimale dans le tableau et ainsi faire moins de chargement mémoire, la boucle k n'est pas déplacé car le k ne modifie en rien le déplacement en mémoire de notre tableau.



Cette optimisation est efficace par elle-même car la lecture en mémoire est plus optimale mais cette optimisation est également importante car elle va augmenter l'efficacité d'une autre optimisation qui sera faite plus tard qui est d'ajouter le flag –O3 à la compilation ce qui va permettre au compilateur de vectoriser la boucle. L'optimisation avec le flag n'en sera que plus efficace.

Parallélisation des boucles :

Après avoir améliorer le déplacement en mémoire à l'aide des boucles, on se rend compte qu'il est possible d'améliorer leur vitesse d'éxécution. En effet les boucles des fonctions « special_cells », « collision » et « propagation » servent à parcourir toutes les particules de la simulation chaque itération de la boucle ne dépendant pas d'une autre itération, il est donc possible de paralléliser ces boucles. Pour cela on peut utiliser openmp afin de paralléliser les boucles.

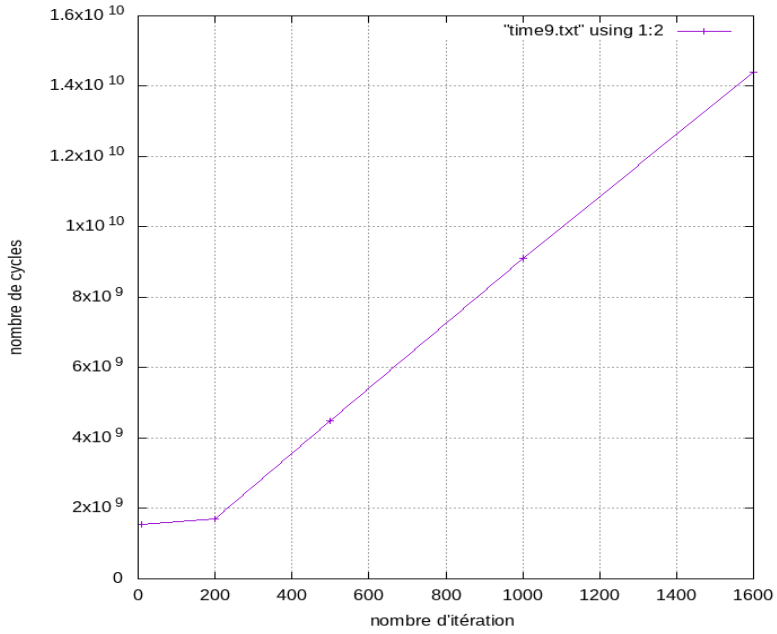
```
#pragma omp parallel
{
    #pragma omp for schedule(static) private(i,j)
    for( i = 1 ; i < mesh_in->width - 1 ; i++ )
        for( j = 1 ; j < mesh_in->height - 1 ; j++)
            compute_cell_collision(Mesh_get_cell(mesh_out, i, j),Mesh_get_cell(mesh_in, i, j));
}
```

Pour utiliser openmp il faut tout d'abord inclure la bibliothèque omp.h, a partir de ce moment on peut utiliser openmp. Afin de paralléliser les boucles on fait tout d'abord un « pragma omp parallel » qui permet de créer une région parallèle, une région parallèle est un endroit du code qui peut être paralléliser avec openmp. Une fois la région parallèle créé on utilise ensuite « #pragma omp for » qui permet de dire qu'il faut paralléliser les boucles for se trouvant sous cette déclaration, on rajoute à la suite un « schedule(static) » qui permet de dire que le nombre de thread que l'on donne au programme divise bien le nombre d'itération de la boucle, les threads auront une séparation égal des itérations de la boucle dans un l'ordre, si on remplace par « schedule(dynamic) » alors le programme s'arrangera pour donner le bon nombre d'it »ration à chaque thread. Il est meilleur de mettre « static » car le programme n'aura pas à redistribuer les itérations à chaque threads mais il ne faut

pas se tromper sur les compte. et enfin on rajoute « private(i,j) » afin que chaque thread possède son propre i et son propre j afin qu'ils ne se dérangent pas les uns les autres. Une fois tout ceci fait il suffit de choisir le nombre de thread que l'on souhaite utiliser pour le programme avec « export OMP_NUM_THREADS=nb_thread » en remplacent nb_thread par le nombre de thread souhaité. La même chose est faite dans les trois fonction à part la fonction « propagation »

```
#pragma omp parallel
{
    #pragma omp for schedule(static) private(i,j,k,ii,jj)
```

En effet cette fonction utilisant plus de variable il faut toutes les mettres en priver pour éviter que les thread se « battent ». Les boucles étant éffectué par plusieurs threads en même temps (chacun faisant un partie) les boucles se termine plus vite que si elles étaient exécuté par un seul thread.



Cette optimisation est très efficace, en effet comme les boucles sont faites en plusieurs petit partie par différents threads, les boucles s'exécutent donc en parallèle et donc plus vite. Ici le code ayant fonctionné sur un knl avec 240 cœurs et mpi ayant été lancé sur 16 processus on en conclu avec un calcul simple que pour utiliser tout les cœurs chaque processus doit utiliser 15 threads.

De entier à entier non signé :

Une optimisation qui peut être faite est de passer le plus de variable possible de entier (signé) à des entier non signé, en effet la gestion de ces deux variable n'est pas la même pour ces deux types et plus d'optimisation sont faite pour les entier non signé. Pour cela on change toutes les valeurs qui ne peuvent pas être négative dans notre cas en non signé. Faire ce changement n'influence en rien le résultat du programme car si les valeurs que l'on passe en non signé ne peuvent pas être négative alors par conséquent les mettre en non signé ne change rien.

Tout d'abord la structure « lbm_comm_t » dans le fichier « lbm_comm.h »

```c
typedef struct lbm_comm_t_s
{
    /** Position de la maille locale dans le maillage global (origine). **/
    size_t x;
    size_t y;
    /** Taille de la maille locale. **/
    size_t width;
    size_t height;
    size_t nb_x;
    size_t nb_y;
    /** Id du voisin de droite, -1 si aucun. **/
    int right_id;
    /** Id du voisin de gauche, -1 si aucun. **/
    int left_id;
    int top_id;
    int bottom_id;
    int corner_id[4];
    /** Requète asynchrone en cours. **/
    MPI_Request requests[32];
    lbm_mesh_cell_t buffer;
} lbm_comm_t;
```

Ici on voit que les valeurs qui ont été passé en entier non signé sont les positions des mailles qui sont forcément positives car sinon elles ne sont pas comprise dans la simulation, la hauteur et la largeur qui sont forcément positive. On ne peux pas changé les positions des voisins en non signé car leurs valeurs doit être égal à -1 si il y'en a pas, les mettre en non signé poserait donc un problème dans le programme.

Dans le fichier « lbm_config.h » la structure « lbm_config_t »

```c
typedef struct lbm_config_s
{
    //discretisation
    size_t iterations;
    size_t width;
    size_t height;
    //obstacle
    double obstacle_r;
    double obstacle_x;
    double obstacle_y;
    //flow parameters
    double inflow_max_velocity;
    double reynolds;
    //derived flow parameters
    double kinetic_viscosity;
    double relax_parameter;
    //results
    const char * output_filename;
    size_t write_interval;
} lbm_config_t;
```

Ici le nombre d'itération devient non signé car on ne peux pas avoir un nombre d'itération négatif, pareil pour la hauteur et la largeur, et enfin l'interal d'écriture dans le fichier est également en non signé car comme le nombre d'itération précédemment cité cela n'a pas de sens d'avoir un nombre d'itération négatif.

Dans le fichier « lbm_struct.h » les structures « Mesh » et « lbm_mesh_type_t »

```c
typedef struct Mesh
{
    /** Cellules du maillages (MESH_WIDTH * MESH_HEIGHT). **/
    lbm_mesh_cell_t cells;
    /** Largeur du maillage local (mailles fantome comprises). **/
    size_t width;
    /** Largeur du maillage local (mailles fantome comprises). **/
    size_t height;
} Mesh;
```

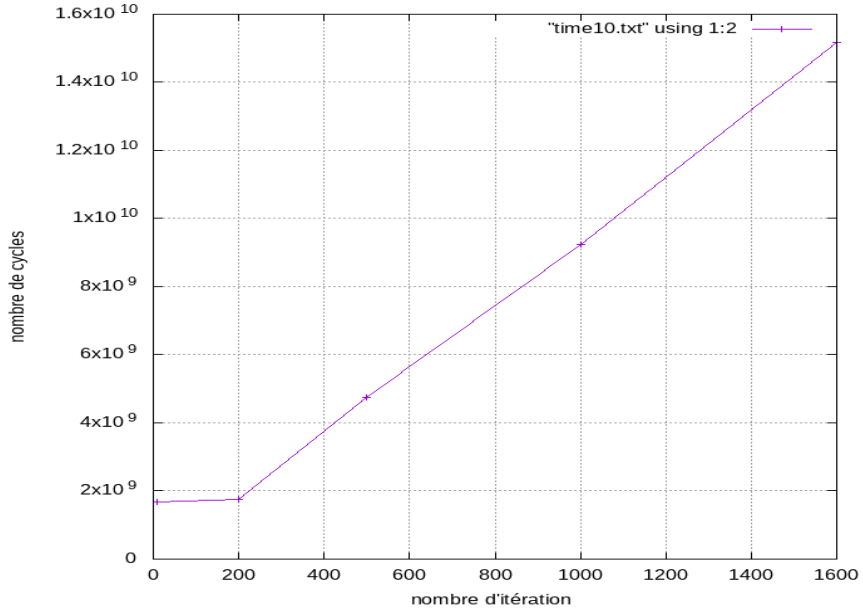
```c
typedef struct lbm_mesh_type_s
{
    /** Type des cellules du maillages (MESH_WIDTH * MESH_HEIGHT). **/
    lbm_cell_type_t * types;
    /** Largeur du maillage local (mailles fantome comprises). **/
    size_t width;
    /** Largeur du maillage local (mailles fantome comprises). **/
    size_t height;
} lbm_mesh_type_t;
```

Dans ces deux structures on modifie la hauteur et la largeur pour les mettre en non signé car comme dit précedemment elles ne peuvent pas être négatives.

Une fois les changement des les .h effectué il faut également changer dans le programme toutes les boucles qui compare un variable à la hauteur et la largeur et passer les variables en non signé également, les printf doivent également être modifié afin d'affiché les bonnes valeurs.

```c
if ((ii >= 0 && ii < mesh_out->width) && (jj >= 0 && jj < mesh_out->height))
```

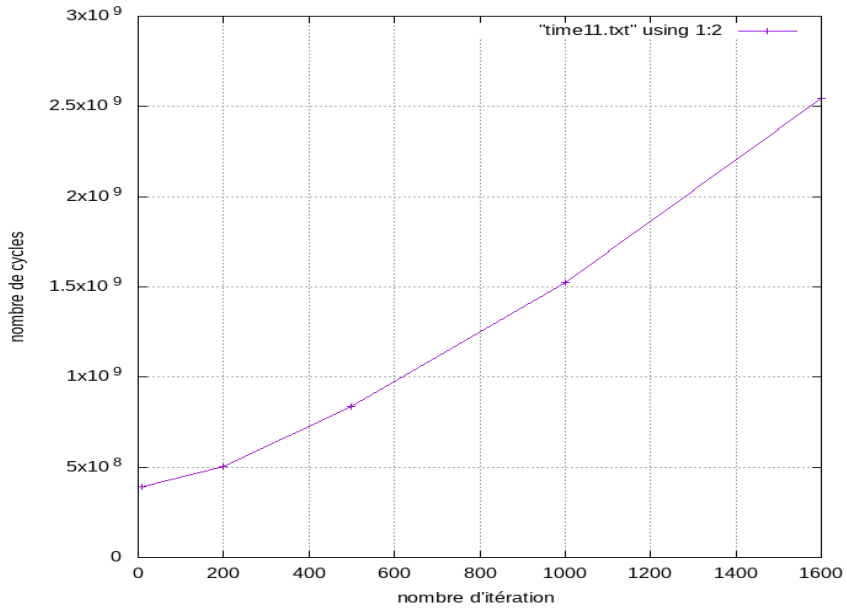
Et enfin dans la fonction « propagation » comme les variables sont devenu des entiers non signé il n'est plus nécessaire de vérifier que les valeurs soient positives car elles le sont forcément ce qui fait des vérification en moins dans le programme, c'est icivimportant car la fonction « propagation » est l'une des fonctions du programme dans laquelle on passe le plus de temps, donc si on fait moins de calcul dedans alors qu'elle est appelé plusieurs fois cela devient intéressant.



Cette optimisation est également très légère mais elle apporte un gain au niveau des cycles d'exécution malgré tout.

Optimisation avec des flags :

Afin d'optimiser encore le programme sans changer le code il est possible d'ajouter des flags lors de la compilation qui permettent au compilateur d'optimiser lui-même le code. Si on rajoute le flags «-O3» on dit au compilateur de beaucoup optimiser le code, avec ce flags il va essayer de vectoriser lui-même les calculs et supprimer certaines parti du programme pas nécessaire comme les boucles qui ne font rien ou des parti de code qui n'on aucun effet. Pour que ce flags soit le plus optimal il faut aider le compilateur en modifiant sois même certaines chose ce qui va l'aider à faire plus d'optimisations comme par exemple en changeant certaines boucles ou en alignant les valeurs en mémoire pour que le compilateur vectorise les calculs. On peut également rajouter le flag « -mavx2 » ce qui permet de dire au compilateur qu'il peux utiliser de l'avx2 pour améliorer le programme si il en a la possibilité. Et enfin on peut également rajouter le flag « -march=native » qui permet de dire au compilateur qu'il peux faire des optimisations en fonctions de la machine sur laquelle le programme fonctionne.



Ici on gagne beaucoup de performance car en laissant le compilateur optimiser lui-même certaines parti du code il devient beaucoup plus efficace car certaines parti code pourront être vectorisé. De plus certaines modification précédente qui n'on pas été efficace d'elle-même on pu améliorer ce optimisation ci. En effet lorsque l'on a echangé l'ordre des boucles cela a probablement aidé les compilateur à vectoriser du fait que la mémoire est mieux lu.

Conclusion :

Suite à toutes les modifications qui ont été effectué on peut se rendre compte des modifications qui sont les plus importantes et celle qui apportent une accéleration qui reste minim par rapport aux autre mais qui fait malgré tout gagné du temps.

-Optimisation les plus efficaces : Les meilleurs optimisations sont parfois les meilleurs, en effet l'une des meilleurs optimisation est de simplement ajouter des flags d'optimisation, cela demande peu de travail pour beaucoup de résultat car avec de bon flags le compilateur est capable de faire beaucoup d'optimisation par lui-même. Les autres optimisations très efficaces étaient de modifier les échanges MPI afin d'en faire moins en envoyant plus de données d'un coup et l'autre optimisation était de paralléliser le code avec openmp. On peut en conclure que paralléliser le programme est une optimisation qui peut être très efficace uniquement si elle est bien faite sinon c'est au risque de perdre des performances.

-Optimisation moins efficace : Beaucoup d'optimisation permetten de gagné quelque performance mais en les accumulant on fini par gagné beaucoup de performance, en plus c'est optimisation peuvent augmenté l'efficacité de d'autre modifications, pour ces optimisations il est necessaire de comprendre le fonctionnement du code pour pouvoir modifié les lectures en mémoires ainsi que certains calculs.