M1 - Techniques d'Optimisation Parallèles

Optimisation d'une simulation

Candice Astier

1er mai 2022

Table des matières

1 Introduction

Ce projet vise à rendre fonctionnel et à optimiser un programme qui simule le parcours d'un fluide dans un tube 2D avec obstacle. Il s'appuie sur la simulation d'une allée de tourbillons de Karman et sur la méthode LBM (Lattice Boltzmann Method). Pour mesurer les optimisations faites, des études de performance ont été développées.

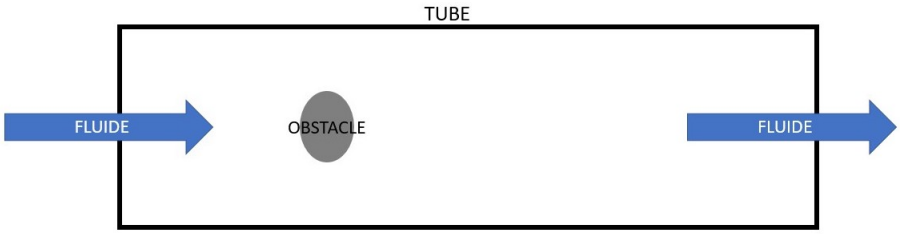

Figure 1 - Schéma simplifié de la simulation

Ce programme est découpé en 5 bibliothèques permettant les calculs sur lesquels un ensemble de modifications ont été effectuées.

- **lbm_config** : ensemble de fonctions décrivant la configuration du problème ;
- **lbm_comm** : ensemble de fonctions gérant toutes les communications entre les processus MPI ;
- **lbm_struct** : ensemble de fonctions gérant les différents types de maillages ;
- **lbm_init** : ensemble de fonctions initialisant toutes les données ;
- **lbm_phys** : ensemble de fonctions effectuant tous les calculs de physique à appliquer sur les particules du fluide.

Afin d'étudier les performances du programme, différents outils ont été utilisés tels que :

- le calcul du temps d'exécution,
- le calcul du nombre de cycles CPU [1],
- l'utilisation de l'outil Valgrind [2],
- l'utilisation de l'outil Perf [3].

L'ensemble des mesures de performances ont été faites sur un ordinateur possédant 4 processus et de 64 octets de cacheline (cf. **8.1 Architecture de la machine**).

2 Étude du code originel

2.1 Structure

L'intérieur du tube a été discrétisé sur un maillage cartésien (avec la hauteur en ordonnée et la largeur du tube en abscisse). Les particules de fluide sont localisées sur des noeuds et possèdent 2 dimensions et 9 directions. Le programme est composé de plusieurs structures décrivant le maillage, les types de cellules (fluide, obstacle...) et leurs positions.

Le maillage est stocké en mémoire sous forme d'un tableau à une dimension, en parcourant le maillage colonne par colonne. Chaque cellule contient les valeurs des 9 directions des particules. Le maillage est donc sous la forme : Mesh[0->8] = cellule n°0, Mesh[9->17] = cellule n°1 ...

Pour pouvoir accélérer le temps d'exécution du programme, il a été parallélisé à l'aide de la bibliothèque MPI (Message Passing Interface). Cela a permis de découper le maillage en plusieurs sous-maillages selon un nombre de processus MPI donné, afin de traiter les différentes parties en parallèle.
Cependant, il existe des cellules communes aux frontières des différentes zones nécessitant un certain nombre de communications entre elles, ainsi que la création de cellules fantômes (cf. **4.5 Communications ROW_MAJOR**) pour stocker les données voisines.

De manière générale, le programme principal (situé dans le fichier main.c) :

1. Un cycle CPU correspond à la durée d'une opération de base. Plus un programme utilisera de cycles CPU, plus son exécution sera longue.
2. Cet outil permet d'avoir un rapport détaillé sur les allocations mémoires effectuées par un programme. Il permet également de détecter (et localiser) les éventuelles fuites de mémoire possible.
3. Partie du code où l'exécution globale du programme passe le plus de temps.

1. Initialise toutes les structures et données, répartit le maillage par processus.
2. Ouvre le fichier de résultats (seul le processus de rang 0).
3. Répète un nombre donné de fois les étapes suivantes :
   (a) Mise à jour des données (densité, vitesse, collisions de particules, déplacement),
   (b) Partage des données nécessaires aux cellules fantômes,
   (c) Écriture dans le fichier de résultats l'ensemble des nouvelles données (seul le processus de rang 0).
4. Fermeture du fichier de résultats (seul le processus de rang 0).
5. Libère la mémoire et arrête les processus MPI.

Une première étude du code d'origine a permis de déterminer que les principaux points de contention de ce programme sont :
— les accès I/O,
— le stockage des données,
— les communications inter-processus MPI.

2.2 Scalabilité

La scalabilité est, ici, la capacité d'un programme à fonctionner normalement lorsque le nombre de processus augmente.
Dans ce programme, l'augmentation du nombre de processus MPI n'influe que sur le nombre de communications entre eux. Il n'influe pas sur le stockage des données et les accès I/O car la taille du maillage est indépendante du nombre de processus ; et les accès I/O ne sont effectués que par le processus MPI maître (processus 0).

Le découpage du maillage en sous-maillages est donné par la formule :

$$nb\_y = PGCD(comm\_size,\ width),\quad nb\_x = \frac{comm\_size}{nb\_y}$$

avec : nb_y le nombre de processus MPI en ordonnée, nb_x en abscisse, $comm\_size$ le nombre total de processus MPI et $width$ la taille du maillage en abscisse.

On peut donc en déduire que :
Si $comm\_size$ est un diviseur de $width$, alors $nb\_y = comm\_size$ et $nb\_x = 1$. Donc, il ne s'agit dans ce cas que d'un découpage horizontal, et ça ne fait qu'augmenter le nombre de lignes de sous-maillage.
Si $comm\_size$ est un nombre premier mais n'est pas un diviseur de $width$, alors $nb\_y = 1$ et $nb\_x = comm\_size$. Donc, il ne s'agit dans ce cas que d'un découpage vertical, et cela ne fait qu'augmenter le nombre de colonnes de sous-maillage.
Dans les autres cas, on ne peut pas déterminer la répartition des processus. On peut seulement dire que dans la plupart des autres cas ce sera un découpage où $nb\_y > 1$ et $nb\_x > 1$; et que dans le cas où la largeur (width) vaut 800, si $comm\_size$ est pair, alors nb_y le sera aussi.

Cependant, $comm\_size$ doit forcement être inférieur ou égal au nombre total de cellules, sinon il y aura des processus qui ne pourront pas avoir accès au maillage (ou possédant un sous-maillage vide de taille 0).
De plus, on peut remarquer que si le nb_y obtenu est supérieur à la hauteur en ordonnée (height), de la même façon que précédemment certains processus n'auront pas de sous-maillage puisqu'il y aura plus de lignes de sous-maillage que de cellules (ce phénomène est contrôlé par une condition plus tard dans le code).

3 Débogage et Nettoyage

Au début de ce projet, les différents problèmes ont été identifiés et ce programme a été rendu fonctionnel. La majeure partie des problèmes ci-dessous ont été identifiés après une lecture en profondeur du code (avant de le tester), les autres ont pu être localisés grâce au débogueur gdb [4] associé à la commande xterm [5] qui permet de créer une fenêtre gdb par processus MPI créé.
De plus, il a été vérifié à la fin de ces corrections et nettoyages que les données obtenues pour 200 frames correspondaient bien avec celles fournies comme références.

4. Débogueur standard du projet GNU.
5. Émulateur de terminal standard.

3.1 Allocation mémoire

Le premier problème se situait dans l'initialisation et la construction des différentes structures, et notamment au niveau du manque d'allocation mémoire pour le maillage.
Comme la mémoire n'était pas allouée, le programme ne pouvait pas stocker les données dans le tableau et donc s'arrêtait sur une erreur de segmentation.
La correction consiste à supprimer les symboles de commentaires devant la fonction d'allocation mémoire ce qui a résolu l'erreur de segmentation.

3.2 Barrière pour les processus MPI

La présence d'une barrière juste avant la fermeture du fichier de résultats empêchait la bonne terminaison des processus MPI, et créait un deadlock [6].
Une barrière MPI permet de synchroniser un ensemble de processus MPI à un endroit donné du programme. L'outil gdb a permis de localiser la barrière responsable du problème et ainsi de trouver qu'elle se situait dans une zone accessible que par un seul processus MPI. Celui-ci attendait donc indéfiniment les autres processus qui ne pouvaient pas accéder à cette barrière.

3.3 Variable globale "extern"

Il a été remarqué lors de la lecture du code que certaines variables globales possédaient l'attribut "extern" et d'autres non (dans la partie physique notamment). Sachant que cet attribut permet d'utiliser une même variable globale dans plusieurs fichiers, il a semblé utile de l'ajouter afin que tous les fichiers et processus MPI puissent l'utiliser correctement.

3.4 Conditions initiales

Certaines parties du code original avaient été mises en commentaire, et notamment une condition permettant l'initialisation des données (dans le fichier lbm_init.c). La remise en fonction de cette condition a permis de donner de la vitesse au fluide lors de la simulation, ainsi que des conditions de départ.

3.5 Fichier usleep.c

Comme le fichier "usleep.c" n'est utilisé dans aucun des programmes fournis, il a semblé judicieux de le supprimer afin de nettoyer le dossier de tous les fichiers inutiles.

3.6 Makefile

Après une lecture du Makefile pour comprendre comment les différents fichiers étaient compilés et quels étaient leurs liens de dépendance, il est apparu que les dernières lignes de ce fichier ne servaient à rien et n'étaient pas appelées lors des différentes compilations car redondantes. Elles ont donc été supprimées.

3.7 Processus multiples

Il a été identifié que si on indiquait un nombre suivant 3,6,7,9,11,12,13... de processus MPI lors de l'exécution, cela créait un décalage des données observé dans l'animation de restitution. Cependant, la source de ce problème n'a pas pu être localisée à temps.

4 Optimisations

Dans une seconde partie, et une fois que le programme est devenu fonctionnel, l'enjeu a été de le rendre le plus optimisé et performant [7] possible.
L'ensemble des mesures de temps d'exécution ont été calculées grâce à la fonction MPI_Wtime() retournant un temps en secondes, représentant le temps d'horloge écoulé depuis un certain temps dans le passé. Dans chacun des cas, cette fonction permet de calculer le temps d'exécution total de la boucle principale dans le fichier main.c (cf. **2.1 Structure**).

6. Phénomène où des processus concurrent s'attendent mutuellement. Un processus peut aussi s'attendre lui même.
7. On définira la performance ici, comme le temps d'exécution le plus rapide possible, ainsi qu'une bonne gestion des données en mémoire.

De plus, les optimisations présentées ci-dessous sont énumérées dans leurs ordres d'ajout/de modification dans le programme. Les temps de références sont donc de plus en plus rapides dans la plupart des parties.

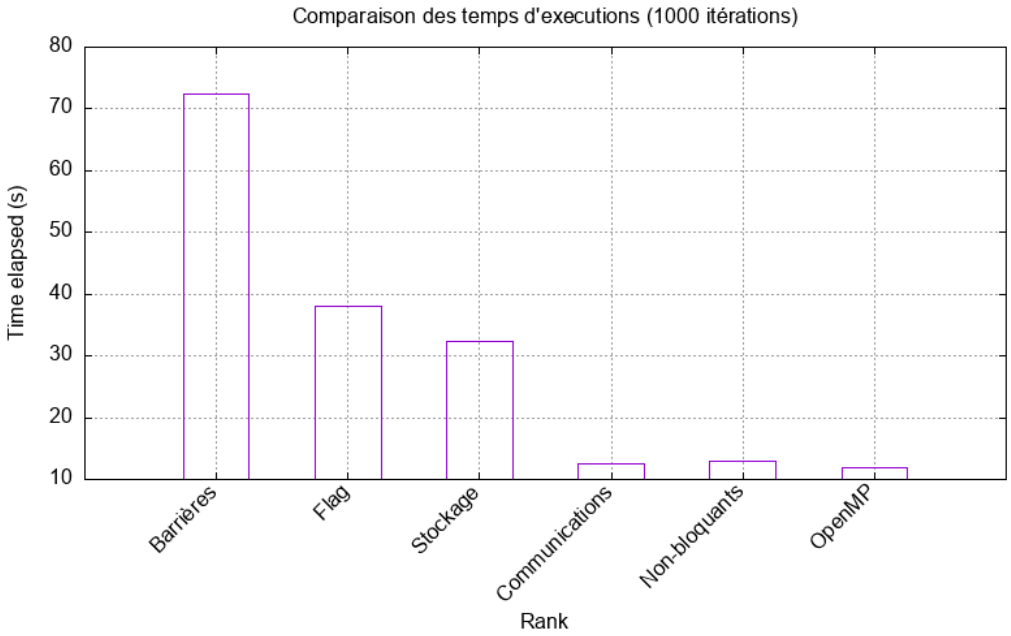

Figure 2 - Schéma d'exemple des communications entre processus MPI (extrait du sujet)

Les valeurs du graphique ci-dessus sont celles détaillées dans les tableaux ci-dessous.

4.1 sleep

La première optimisation majeure a été de localiser et supprimer l'attente de 1 seconde après l'ensemble des communications de cellules fantômes entre processus MPI à chaque itération. Ce "sleep" avait pour conséquence de ralentir considérablement le temps d'exécution total.

Le tableau ci-dessous compare les temps d'exécution de la boucle dans le fichier main.c, pour 4 processus MPI, 100 itérations et une écriture des résultats tous les 5 tours de boucle. On remarque que ce sleep ajoute bien au programme x secondes, avec x le nombre d'itérations.

	Rang 0	Rang 1	Rang 2	Rang 3
Avec sleep	105.765647s	105.765646s	105.765649s	105.765644s
Sans sleep	6.737187s	6.737185s	6.737184s	6.737184s

Figure 3 - Tableau de comparaison des temps d'exécution

Par ailleurs, cette première étude de performance permet de montrer que les processus sont égaux en temps d'exécution, donc qu'il y a une bonne répartition de la charge entre les différents processus dans la boucle. Cela signifie également qu'il n'y a donc pas de processus MPI qui va ralentir tous les autres car plus long.

4.2 Barrières pour les processus MPI

Une lecture du code a permis de révéler un certain nombre de barrières MPI dans la boucle principale entre chacune des sous-étapes (cf. **2.1 Structure**).
Toutes ces barrières imposent un temps d'arrêt pour tous les processus afin de les synchroniser. Elles sont utiles lorsque des processus doivent communiquer entre eux, et doivent pour cela arriver en même temps afin de ne pas utiliser de mauvaises données (par exemple certaines non mises à jour dans les cellules fantômes). Or dans la boucle en question, les fonctions "protégées" par ces barrières ne sont pas dépendantes des valeurs modifiées par les autres processus MPI, on peut donc les enlever sans générer d'erreurs.

Le tableau ci-dessous compare les temps d'exécution de la boucle principale, pour 4 processus MPI, 1000 itérations et une écriture des résultats tous les 5 tours de boucle.

	Rang 0	Rang 1	Rang 2	Rang 3
Avec barrières	74.431382s	74.431376s	74.431379s	74.431379s
Sans barrières	72.412610s	72.412675s	72.412649s	72.412700s

Figure 4 - Tableau de comparaison des temps d'exécution

On remarque qu'il n'y a pas de différences significatives de temps entre les deux versions (les gifs de simulations obtenus sont également identiques), ce qui confirme que les barrières sont ici inutiles car elles n'apportent rien. Cependant, il n'y a aucun gain de temps car tous les processus MPI effectuent les mêmes calculs donc avancent plus ou moins à la même vitesse. S'il y avait eu plus de différences entre les exécutions des processus MPI, ces barrières auraient ralenti l'exécution car certains processus auraient fini leurs calculs bien avant les autres.

4.3 Flag de compilation

Afin d'optimiser un binaire lors de sa création on peut utiliser des "flags de compilation" qui sont des options passés au compilateur afin d'activer un certain nombre d'autres options permettant d'optimiser les performances du code et du binaire, ce que peut faire le compilateur car ayant une vue plus globale du programme. Le flag "-O3" a donc été utilisé, qui est l'un des plus hauts niveaux d'optimisation.

Le tableau ci-dessous compare les temps d'exécution de la boucle dans le fichier main.c, pour 4 processus MPI, 1000 itérations et une écriture des résultats tous les 5 tours de boucle.

	Rang 0	Rang 1	Rang 2	Rang 3
Sans flag	71.384820s	71.384875s	71.384843s	71.384892s
Avec flag	38.058410s	38.058637s	38.058935s	38.059062s

Figure 5 - Tableau de comparaison des temps d'exécution

On remarque que la présence du flag permet de grandement diminuer le temps d'exécution de la boucle. Il y a presque un facteur de diminution de 50% entre les deux versions.

4.4 Stockage ROW_MAJOR

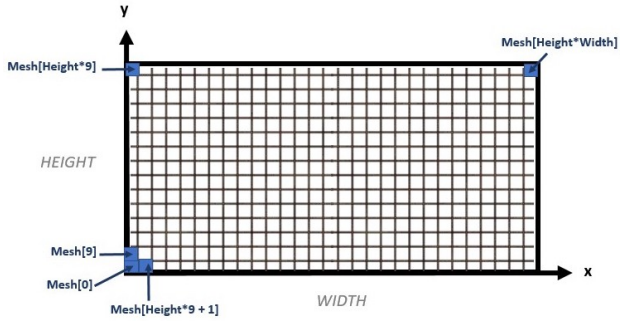

Figure 6 - Stockage COL_MAJOR

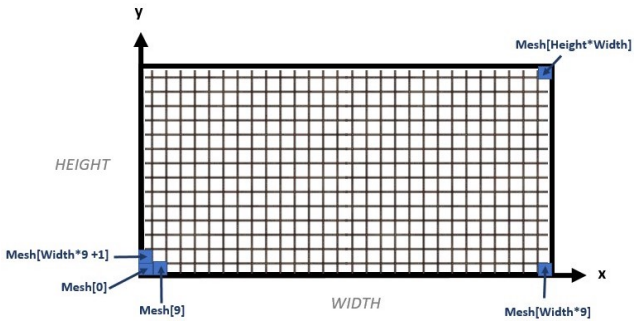

Figure 7 - Stockage ROW_MAJOR

Le stockage d'un tableau de 2 dimensions en ROW_MAJOR est le stockage des cases de ce tableau de manières contiguës en mémoire, en le parcourant ligne par ligne.
La case du tableau de coordonnées $(x; y)$ se trouve donc à la position $[y * WIDTH + x]$ en mémoire (à partir de l'adresse du début du tableau).
Le stockage d'un tableau de 2 dimensions en COL_MAJOR est le stockage des cases de ce tableau de manières contiguës en mémoire, en le parcourant colonne par colonne.
La case du tableau de coordonnées $(x; y)$ se trouve donc à la position $[x * HEIGHT + y]$ en mémoire (à partir de l'adresse du début du tableau).

Comme vu précédemment dans la partie **2.1 Structure**, le stockage original du maillage (et des sous-maillages) est en COL_MAJOR. Donc afin de pouvoir optimiser les communications MPI par la suite, une grosse transformation a été de changer le format de stockage en ROW_MAJOR.

La figure 8 compare les temps d'exécutions de la boucle pour 4 processus MPI, 1000 itérations et une écriture des résultats tous les 5 tours de boucle.

	Rang 0	Rang 1	Rang 2	Rang 3
COL_MAJOR	32.992406s	32.992591s	32.992263s	32.992391s
ROW_MAJOR	32.331395s	32.331393s	32.331396s	32.331393s

Figure 8 - Tableau de comparaison des temps d'exécution

Comme on peut le voir dans le tableau ci-dessus qui compare les temps d'exécution de la boucle principale (pour 4 processus MPI, 1000 itérations et une écriture des résultats tous les 5 tours de boucle), on remarque qu'il n'y a pas de différence entre les deux versions. Ce phénomène est dû au fait que toutes les fonctions parcourent le maillage (ou sous-maillages) toujours de la même façon que ce soit en COL_MAJOR ou ROW_MAJOR. C'est aussi dû au fait que les tableaux sont de petites tailles donc on ne peut pas vraiment observer de différences.

4.5 Communications ROW_MAJOR

Sachant qu'une communication entre processus MPI a besoin entre autres d'une adresse mémoire vers une donnée, d'un nombre de données à transférer et du rang du processus destinataire (si envoi) ou source (si réception). De plus, elle requiert un certain temps d'exécution.

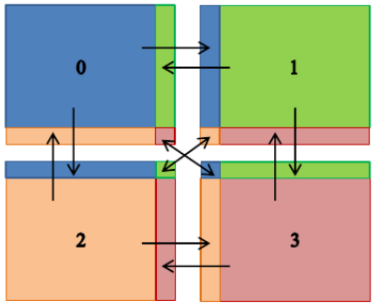

Figure 9 - Schéma d'exemple des communications entre processus MPI (extrait du sujet)

Dans le code d'origine, il y avait deux points de contention au niveau de ces communications :

1. Dans la plupart des cas de partage du maillage (cf. **Figure 9**), chaque processus a besoin des trois types de communications différentes (horizontales, verticales et diagonales) avec leurs cases fantômes associées.
2. Pour chacun des trois types de communications, le code envoyait les données soit cellules par cellules soit cases par cases [8]. Il y avait donc la présence de doubles ou triples boucles, ce qui multipliait les envois de données qui sont pourtant contiguës en mémoire (cf. **4.4 Communications ROW_MAJOR**).

Donc pour résoudre le premier point, le partage du maillage a été modifié et fixé afin de n'avoir que des communications verticales entre les processus. Un processus peut avoir un sous-maillage de hauteur minimum de 1 et maximum HEIGHT (la hauteur totale du maillage) ; il aura toujours une largeur égale à celle du maillage (WIDTH). Une limite de la scalabilité de ce découpage est donc qu'il ne peut y avoir pas plus de processus MPI que de hauteur du maillage.

Pour le second point, comme le maillage est partagé de manière seulement verticale entre les processus et que les données du maillage sont rangées en ROW_MAJOR, il est possible d'envoyer une ligne entière du maillage en un unique envoi : la communication prend en argument l'adresse du début de la ligne comme adresse de départ, et la taille de la ligne comme nombre d'élément à lire en mémoire.
Cela permet alors d'enlever les triples/doubles boucles devenues inutiles, et de réduire considérablement le nombre d'envois.

Le tableau suivant compare les temps d'exécutions de la boucle pour 4 processus MPI, 1000 itérations et une écriture des résultats tous les 5 tours de boucle.

	Rang 0	Rang 1	Rang 2	Rang 3
COL_MAJOR	32.266432s	32.266444s	32.266747s	32.266879s
ROW_MAJOR	12.625798s	12.626029s	12.626032s	12.628599s

Figure 10 - Tableau de comparaison des temps d'exécution

On remarque que le temps d'exécution de la deuxième version est plus faible et qu'il est diminué de environ 50%. Cette modification a donc pu optimiser les performances de ce programme de manière significative.

8. Une cellule contient les 9 directions possible de la particule, tandis qu'une case du tableau qui stocke le maillage ne contient la valeur que d'une seule directions.

4.6 Communications non-bloquantes et recouvrement

Un inconvénient des communications MPI bloquantes sont qu'elles bloquent un processus MPI après un envoi tant que le processus destinataire n'a pas reçu le message. Cela peut donc faire perdre du temps qui aurait pu être consacré à d'autres tâches ne nécessitant pas les données envoyées.
Afin de résoudre ce problème, les communications bloquantes ont donc été échangées par des modifications non-bloquantes afin de pouvoir également appeler un ensemble d'autres fonctions [9] (qui sont alors indépendantes des cellules fantômes) entre les envois et les réceptions. Celles-ci permettent d'envoyer un message sans figer le processus en attente d'une réception. Pour être sûr que le message est bien arrivé, la fonction MPI_Wait() a été utilisée, qui permet aux processus que le message soit bien transféré.
Cette modification a également permis de supprimer les barrières MPI de synchronisation dans les fonctions d'échanges de messages.

La figure suivante compare les temps d'exécution de la boucle principale pour 4 processus MPI, 1000 itérations et avec une écriture toutes les 5 itérations.

	Rang 0	Rang 1	Rang 2	Rang 3
Bloquant	13.525184s	13.525174s	13.526201s	13.526370s
Non-bloquant	12.845000s	12.8465572s	12.846707s	12.849260s

Figure 11 - Tableau de comparaison des temps d'exécution

Comme on peut le voir ci-dessus, le passage de bloquant à non bloquant n'a pas eu d'impact sur les temps d'exécution de la boucle. Cela peut se justifier par le fait qu'il n'y a que 4 processus MPI, donc ça ne fait que 6 communications (envoi-réception) par boucles (ce qui est très peu !) ; et les messages sont envoyés assez rapidement.

4.7 Nettoyage et stockage

Le nettoyage de toutes les données dans les structures inutilisées (buffer, rang de gauche et de droite, énumérations de CORNER ...) a permis de ne pas allouer de la mémoire pour rien, et donc dans un sens d'optimiser le stockage des données. Il n'a cependant pas permis d'accélérer le temps d'exécution de la boucle principale.

Version d'origine	3.185 allocs, 3.185 frees, 2.437.692 bytes allocated
Version actuelle	3.165 allocs, 3.165 frees, 2.436.504 bytes allocated

Figure 12 - Tableau de comparaison des allocations (extrait du rapport fournis par Valgrind)

4.8 OpenMP

Afin d'augmenter le taux de parallélisation pour pouvoir accélérer l'exécution du programme, une programmation hybride MPI+OpenMP a été mise en place et notamment au niveau de la bibliothèque d'initialisations qui a été entièrement modifiée avec des threads OpenMP. Cette programmation hybride permet pour chaque processus d'avoir un nombre de threads attribué afin de pouvoir paralléliser à son échelle. Il s'agit donc d'un mode de parallélisation à fins grains.

On peut donc vérifier dans le tableau ci-dessous (1000 instructions, écriture toutes les 5 instructions, 2 processus MPI ; et 2 threads OpenMP pour la deuxième version, avec une attente passive et une distribution des données statiques) que cela n'a en effet eu aucun impact sur la boucle principale et sur l'écriture de nos données. Cependant toute la partie initialisation des maillages a effectivement pu être accélérée (et donc optimisée).

	Rang 0	Rang 1
Sans OpenMP	11.763931s	11.763905s
Avec OpenMP	11.801775s	11.803319s

Figure 13 - Tableau de comparaison des temps d'exécution

4.9 Effet NUMA

L'effet NUMA est le fait que lors du stockage des données par un processus, celles-ci se retrouvent dans la bande mémoire ou les caches d'un autre coeur NUMA. Cet effet se caractérise par une augmentation du temps de latence de chargement des données car le processus doit aller les chercher ailleurs, ce qui est forcément plus

9. Ce sont notamment les fonctions *special_ cells()* et *collision()*.

long.
Comme il n'y a qu'un seul coeur NUMA sur la machine de tests utilisée, il n'y a pas eu besoin de pallier ce problème. Cependant, pour fixer les noeuds que l'on souhaite uniquement utiliser (lors de la répartition des processus MPI et des threads) et pour indiquer dans quelle mémoire stocker les données, il existe une commande à utiliser lors de l'exécution du programme :

```
$ numactl --cpunodebind=<node list> --membind=<node list>
```

Cette commande permet normalement de ne pas accumuler l'effet NUMA à chaque I/O et donc d'optimiser le programme.
Par ailleurs le découpage du maillage permet également la bonne répartition des données pour ne pas que certains processus chargent des données nécéssaires à un autre processus. Cela aurait également pour effet d'ajouter un temps de récupération des données.

4.10 Structures de données

Une optimisation du stockage des données dans les structures a également été identifiée. Lorsque le programme démarre, les données sont allouées et stockées en mémoire. Cependant selon l'ordre dans lequel elles ont été déclarées, elles seront plus ou moins alignées en mémoire. De plus, une donnée aura une adresse mémoire qui est multiple de sa taille. Un entier (en C) est stocké sur 4 octet, un double sur 8 octets et une adresse sur 1.

```
lbm_config_t
{
        int iterations;
        int width;
        int height;
        double obstacle_r;
        double obstacle_x;
        double obstacle_y;
        double inflow_max_velocity;
        double reynolds;
        double kinetic_viscosity;
        double relax_parameter;
        const char * output_filename;
        int write_interval;
}
```

Par exemple, dans la structure *lbm_config_t* les trois premiers entiers seront bien alignés, tandis que la première adresse accessible pour la donnée *obstacle_r* n'est que 4 octets après *height*. Cela fait donc un espace mémoire (de 4 octets) non utilisé. Pour utiliser avoir le minimum de trous, la donnée *write_interval* a été placée à la suite de *height*.
L'optimisation a donc été de vérifier pour chacune des structures de données si leurs contenus étaient bien alignés en mémoire.

4.11 Cacheline

Lorsqu'une donnée est chargée depuis la mémoire, le processeur charge la donnée ainsi qu'un certain nombre de ses données voisines (alignées en mémoire). Cette quantité est déterminée par la taille de la cacheline (sur la machine de tests utilisé, la cacheline vaut 64 octets). De plus, le coût de lecture dans la mémoire est bien supérieur au coût de lecture directe dans la ligne de cache.
Comme dans la plupart des fonctions il y a une boucle parcourant le maillage (ou sous-maillage) ou l'ensemble des 9 directions d'une cellule ; et que ses données sont bien alignées en mémoire (cf. **4.4 Stockage ROW_MAJOR**), le processus MPI ne va chercher en mémoire des informations que toutes les tailles de cachelines.
Le maillage est composé de valeurs de type double, donc stockées sur 8 octets, ce qui signifie qu'une cacheline peut contenir au plus 8 cases du tableau à la fois. Elle est donc remplie de données pertinentes à chaque tour de boucle, et ne contient aucun trou.

Si les données n'avaient pas rempli la ligne de cache (par exemple, si le maillage était composé de moins de directions ou de dimensions), une optimisation de stockage aurait pu être de regrouper les données ou de les espacer, en ajoutant des données vides, afin de pouvoir charger les données dans les lignes de cache de manière optimisée.

4.12 Autres

Dans cette partie, il est présenté un ensemble d'autres optimisations qui pourront être explorées par la suite.

4.12.1 Compilateur

Chaque compilateur est plus ou moins adapté à des types de codes en particulier.
Le programme pourra notamment être testé avec le compilateur *icc* qui est plus conseillé pour des gros calculs. Ce compilateur permettrait alors d'accélérer les temps de calculs et donc d'accélérer les temps d'exécution.

4.12.2 MPI_File

Une utilisation des fonctions MPI I/O qui permettent à tous les processus MPI d'écrire dans un même fichier de manière parallèle et organisée, sans corrompre les données.

4.12.3 Vectorisation

L'outil Perf a permis de déterminer les fonctions de calculs les plus utilisées dans le programme, et plus particulièrement leurs instructions [10] limitantes.

```
Samples: 2M of event 'cycles', Event count (approx.): 906418123548
Overhead  Command  Shared Object           Symbol
  17.06%  lbm      mca_btl_vader.so        [.] mca_btl_vader_component_progress
  14.57%  lbm      lbm                     [.] get_cell_velocity
  12.51%  lbm      lbm                     [.] get_vect_norme_2
  12.32%  lbm      lbm                     [.] propagation
  11.63%  lbm      lbm                     [.] compute_equilibrium_profile
   6.44%  lbm      lbm                     [.] Mesh_get_cell
   6.29%  lbm      lbm                     [.] compute_cell_collision
   4.70%  lbm      lbm                     [.] get_cell_density
   3.14%  lbm      libopen-pal.so.40.10.1  [.] opal_progress
   1.29%  lbm      mca_pml_ob1.so          [.] mca_pml_ob1_recv_req_start
   0.82%  lbm      libopen-pal.so.40.10.1  [.] opal_timer_linux_get_cycles_sys_timer
   0.70%  lbm      mca_pml_ob1.so          [.] append_frag_to_list
   0.68%  lbm      mca_pml_ob1.so          [.] opal_free_list_wait.constprop.9
   0.56%  lbm      mca_pml_ob1.so          [.] mca_pml_ob1_recv_frag_callback_match
   0.42%  lbm      mca_pml_ob1.so          [.] mca_pml_ob1_recv
```

Figure 14 - extrait du rapport d'analyse de Perf

```
Samples: 2M of event 'cycles', 4000 Hz, Event count (approx.): 906418123548
get_cell_velocity  /users/user2204/original/lbm [Percent: local period]
  2.29 |       movsd   (%rax),%xmm1
  0.52 |       mov     -0x8(%rbp),%eax
  1.65 |       cltq
  0.45 |       lea     0x0(,%rax,8),%rdx
  1.70 |       mov     -0x20(%rbp),%rax
  0.42 |       add     %rdx,%rax
  2.00 |       movsd   (%rax),%xmm2
  0.46 |       mov     -0x4(%rbp),%eax
  1.71 |       cltq
  8.81 |       mov     -0x8(%rbp),%edx
  1.69 |       movslq  %edx,%rdx
  0.45 |       add     %rdx,%rdx
  1.99 |       add     %rdx,%rax
  3.66 |       lea     0x0(,%rax,8),%rdx
  1.71 |       lea     direction_matrix,%rax
  8.39 |       movsd   (%rdx,%rax,1),%xmm0
 16.21 |       mulsd   %xmm2,%xmm0
  0.45 |       mov     -0x4(%rbp),%eax
  1.70 |       cltq
  0.43 |       lea     0x0(,%rax,8),%rdx
  1.68 |       mov     -0x18(%rbp),%rax
  0.43 |       add     %rdx,%rax
  7.77 |       addsd   %xmm1,%xmm0
  8.16 |       movsd   %xmm0,(%rax)
       |     for ( k = 0 ; k < DIRECTIONS ; k++)
  1.70 |       addl    $0x1,-0x8(%rbp)
  0.48 |10f:   cmpl    $0x8,-0x8(%rbp)
  1.89 |     ↑ jle     96
```

Figure 15 - extrait du rapport d'analyse de Perf pour la fonction *get_cell_velocity()*

D'après une première étude rapide de ce rapport, on remarque que parmi ces instructions il n'y a pas d'instructions vectorielles.

10. Une instruction est une opération la plus simple possible (ex : addition, multiplication, récupération de données,...) pouvant être traduite directement en instruction machine.

La figure 15 présente donc le taux d'occurence de certaines instructions en code assembleur. Les instructions choisies [11] se regroupent en trois catégories :

— des instructions "scalaires" : traitent les données une par une ;
— des instructions "quad-words" : traitent les données 2 par 2 (ou 4 par 4, selon la taille des données) ;
— des instructions "packed" : traitent les données par blocs ;

Les deux dernières catégories traduisent donc une vectorisation des instructions et des données.

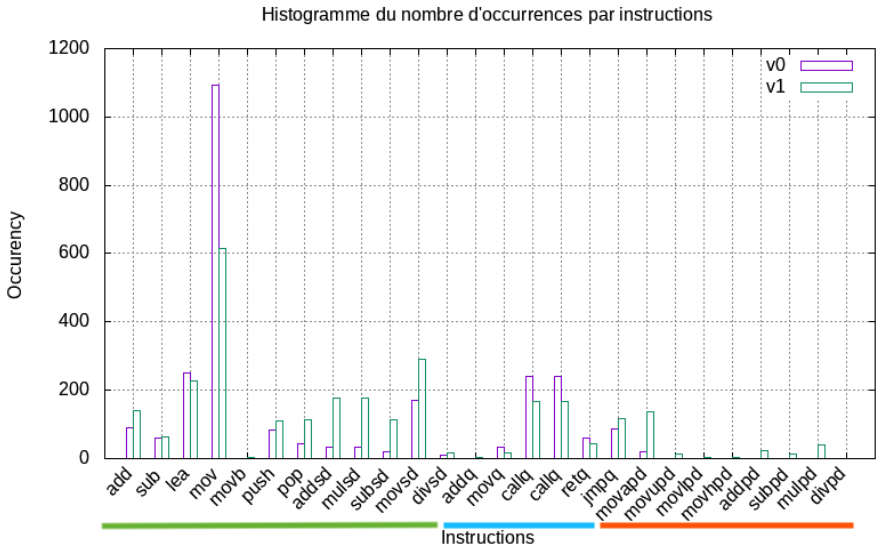

Figure 16
avec : $v0$ la version d'origine et $v1$ la version la plus optimisée

On en déduit donc que même si une partie des optimisations déjà effectuées ont permis d'augmenter la part d'instructions vectorielles, elles ont également augmenté le nombre d'instructions scalaires. Il y a donc besoin d'augmenter la vectorisation des calculs dans les boucles du programme, afin de pouvoir réduire le temps d'exécution.

5 Points importants

Pour optimiser correctement un programme parallèle MPI, les points importants sont donc :

1. La répartition des données entre les processus MPI : ce point définit majoritairement les communications entre processus MPI ; puisque selon la répartition il y aura plus ou moins besoin de cases fantômes et donc plus ou moins de communications inter-processus ;
2. Le stockage des données (dans notre cas en ROW_MAJOR) : pour accéder aux données le plus rapidement possible ;
3. Le recouvrement des communications si possible : pour gagner le plus de temps d'exécution possible ;
4. Une implémentation hybride MPI+OpenMP : afin de paralléliser à tous les niveaux.

6 Conclusion

Pour conclure, le programme de simulation a d'abord été débogué puis optimisé. Les optimisations sont donc de deux types :

— une optimisation du stockage et de la gestion de la mémoire,
— une optimisation des temps d'exécution.

Elles reposent notamment sur une réorganisation des données et sur une réorganisation des communications pour les réduire au maximum. Une autre optimisation importante a été le passage à une programmation hybride MPI+OpenMP. Tout cela a donc permis de résoudre au mieux les différents points de contention identifiés.
Cette analyse doit ensuite se poursuivre avec les différents points abordés dans la section **4.12 Autres**.

11. Les instructions présentées ont été choisies arbitrairement après une rapide étude du code assembleur généré.

7 Références

Références

[1] **TOP**
Supportsdecours

[2] **Options That Control Optimization**
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] **x86 and amd64 instruction reference**
https://www.felixcloutier.com/x86/index.html

[4] **Programation hybrides MPI+OpenMP**
https://grenoblecalcul.imag.fr/IMG/pdf/form_hybride_print.pdf

[5] **MPI**
https://www.hpctoday.com/hpc-labs/optimiser-les-implementations-mpi-1ere-partie/

8 Annexes

8.1 Architecture de la machine

Grâce à la commande "lstopo", il a été généré un schéma de la structure machine utilisée pour les tests de performances.

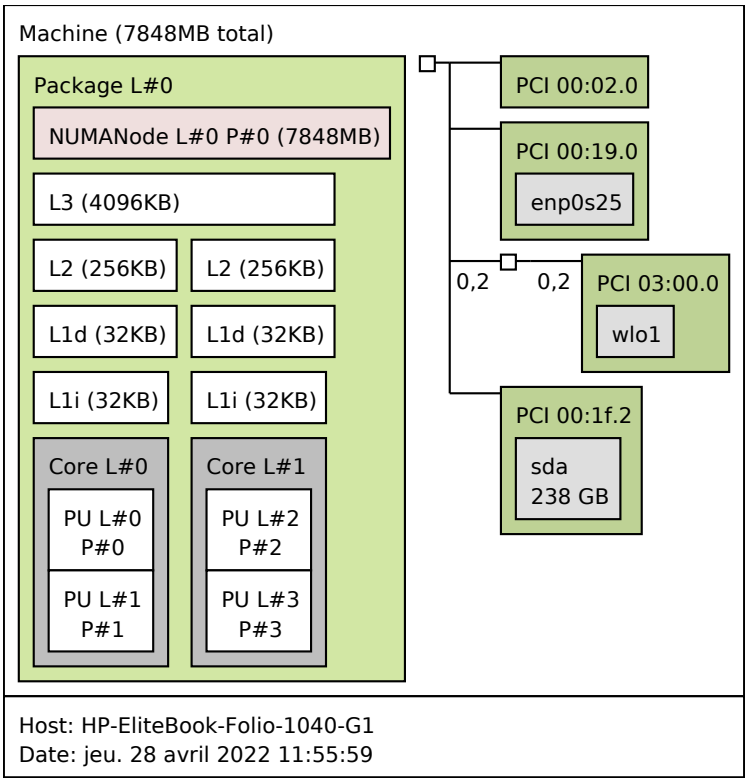

Figure 17 - structure de la machine de tests