Master M1 Calcul Haute Performance Simulation (CHPS)

Projet Techniques d'optimisation:

Optimisation d'une simulation Numérique

Aicha Maaoui

Avril 2022

Abstract

Le projet "Optimisation d'une simulation Numérique" est associé au cours de techniques d'optimisation de la parallélisation, dans lequel une optimisation d'une simulation numérique d'une allée de tourbillons de Karman (à partir d'un code fourni) est faite afin de la rendre la plus scalable possible.

Ce projet a pour buts:

- Identification et correction des bugs du code d'origine avec les techniques de débogage (Chapitre 1),
- Description et évaluation du code d'origine (Chapitre 2),
- Optimisation (Chapitre 3),
- Conclusion sur les règles conduites (Chapitre 4).

Contents

List of Tables

Listings

Chapter 1

Identification & Correction des Bugs du code d'Origine

On se propose dans un premier lieu de compiler et exécuter le code d'origine de la simulation numérique fourni dans le cadre du projet, afin d'identifier les problèmes et de les corriger.

1.1 Bug 1: Faute de Segmentation

La compilation et l'exécution du code de base sur un processus est possible à l'aide des commandes "**make**" et "**./lbm**" dans un terminal Linux.

1.1.1 Description de la faute de segmentation

L'exécution du code aboutit au résultat illustré dans listing (1.1). Le nombre d'itérations a été réduit pour des tests plus rapides (De 16000 itérations à 100 itérations).

Listing 1.1: Faute de Segmentation lors de l'exécution du code d'origine.

```
1 aicha@aicha-Vostro-3500:/.../simu_simple_LBM$ ./lbm
2 ========== CONFIG ==========
3 iterations          = 100
4 width               = 800
5 height              = 160
6 obstacle_r          = 17.000000
7 obstacle_x          = 161.000000
8 obstacle_y          = 83.000000
9 reynolds            = 100.000000
```

```
reynolds              = 100.000000
inflow_max_velocity   = 0.100000
output_filename       = resultat.raw
write_interval        = 10
------------ Derived parameters --------------
kinetic_viscosity     = 0.034000
relax_parameter       = 1.661130
=================================================
RANK 0 (LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) (POSITION 0 0)
     (WH 802 162 )
[aicha-Vostro-3500:17282] *** Process received signal ***
[aicha-Vostro-3500:17282] Signal: Segmentation fault (11)
[aicha-Vostro-3500:17282] Signal code: Address not mapped (1)
[aicha-Vostro-3500:17282] Failing at address: (nil)
[aicha-Vostro-3500:17282] [0] /lib/x86_64-linux-gnu/libc.so.6(+0x430c0)
    [0x7fdf10a170c0]
[aicha-Vostro-3500:17282] [1] ./lbm(+0x2b55)[0x55db72ceeb55]
[aicha-Vostro-3500:17282] [2] ./lbm(+0x2e32)[0x55db72ceee32]
[aicha-Vostro-3500:17282] [3] ./lbm(+0x19ef)[0x55db72ced9ef]
[aicha-Vostro-3500:17282] [4] /lib/x86_64-linux-gnu/libc.so.6 (__libc_start_main
    +0xf3)[0x7fdf109f80b3]
[aicha-Vostro-3500:172Description et evaluation du code d origine82 ] [5]
    ./lbm(+0x138e)[0x55db72ced38e]
[aicha-Vostro-3500:17282] *** End of error message ***
Segmentation fault (core dumped)
```

Cette exécution mène a une faute de segmentation (Segmentation fault) du programme, suite à une adresse nulle. Ceci est mis en évidence dans listing (1.1) particulièrement par des lignes 20, 21, 22 et 30 (en rouge).

1.1.2 Identification de la faute de segmentation avec GDB

GDB est utilisé avec la flag de compilation **-g**, qui est déjà implementée dans le **Makefile: CFLAGS=-Wall -g**. Cette flag permet d'utiliser les informations de débogage supplémentaires pour indiquer le source des bugs.

La démarche d'exécution du code d'origine dans GDB est décrite sous une forme réduite dans listing (1.2).

Listing 1.2: Exécution du code d'origine dans GDB.

```
1  aicha@aicha-Vostro-3500:/ ... /simu_simple_LBM$ gdb ./lbm
2  GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
3  Copyright (C) 2020 Free Software Foundation, Inc.
4  ...
5  Reading symbols from ./lbm...
6  (gdb) run
7  Starting program: /media/aicha/DATA/Technique_Optimisation/TOP_LBM/simu_simple_
       LBM/lbm
8  [Thread debugging using libthread_db enabled]
9  Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
10 [Detaching after fork from child process 17305]
11 [New Thread 0x7ffff7841700 (LWP 17310)]
12 [New Thread 0x7ffff6ea5700 (LWP 17311)]
13 ================== CONFIG ==================
14                       ...
15 ------------ Derived parameters ------------
16                       ...
17 ============================================
18  RANK 0 (LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) (POSITION 0 0)
        (WH 802 162 )
19
20 Thread 1 "lbm" received signal SIGSEGV, Segmentation fault.
21 0x0000555555556b55 in setup_init_state_global_poiseuille_profile (mesh=0
       x7fffffffdc10, mesh_type=0x7fffffffdc40, mesh_comm=0x7fffffffdc50) at lbm_init
       .c:85
22 85      Mesh_get_cell(mesh, i, j)[k] = compute_equilibrium_profile(v,density,k);
```

D'après listing (1.2), on constate que le problème menant au premier bug figure dans la ligne 85, "*Mesh_get_cell (mesh, i, j)[k] = compute_equilibrium_profile(v,density,k)*", dans le code source *lbm_init.c*. Pour savoir la source exacte du problème, on peut imprimer des lignes à partir du code source de *Mesh_get_cell()*.

Listing 1.3: Impression des lignes à partir du code source de *Mesh_get_cell()*.

```
1  (gdb) list Mesh_get_cell
2  110 /******************* FUNCTION *********************/
3  111 /**
4  112  * Fonction a utiliser pour recuperer une cellule du maillage en fonction de
       ses coordonnees.
5  113 **/
6  114 static inline lbm_mesh_cell_t Mesh_get_cell( const Mesh *mesh, int x, int y)
7  115 {
8  116  return &mesh->cells[ (x * mesh->height + y) * DIRECTIONS ];
9  117 }
10 118
11 119 /******************* FUNCTION *********************/
```

D'après listing (1.3), on voit que mesh est un pointeur, qu'on peut afficher son contenu comme montré dans listing (1.4).

Listing 1.4: Impression des lignes à partir du code source de *Mesh_get_cell()*.

```
1  (gdb) print mesh
2  $1 = (Mesh *) 0x7fffffffdc10
3  (gdb) print *mesh
4  $2 = {cells = 0x0, width = 802, height = 162}
```

On peut remarquer d'après la ligne 4 dans listing (1.4) que "***mesh*** $->$ ***cells***" contient une adresse nulle. On en déduit que l'initialisation de mesh n'est pas faite correctement. De plus, on peut afficher la pile d'appel avec ***backtrace*** comme illustré dans listing (1.5).

Listing 1.5: Impression des lignes à partir du code source de *Mesh_get_cell()*.

```
1  (gdb) backtrace
2  0 0x0000555555556b55 in setup_init_state_global_poiseuille_profile (mesh=0
       x7fffffffdc10, mesh_type=0x7fffffffdc40, mesh_comm=0x7fffffffdc50) at lbm_init
       .c:85
3  1 0x0000555555556e32 in setup_init_state (mesh=0x7fffffffdc10, mesh_type=0
       x7fffffffdc40, mesh_comm=0x7fffffffdc50) at lbm_init.c:156
4  2 0x00005555555559ef in main (argc=1, argv=0x7fffffffdea8) at main.c:160
```

La ligne 4 de listing (1.5) indique que *mesh* est appelée dans *main.c* depuis la ligne 160. Le code source *lbm_struct.c* montre que dans la routine *Mesh_init*, *mesh*$->$*cells* n'est pas alouée,

mais initialisée à *NULL* dans la ligne 20 de *lbm_struct.c*.

1.1.3 Correction de la faute de segmentation

L'allocation mémoire de $mesh->cells$ est commentée dans le fichier *lbm_struct.c* (ligne 19 dans *lbm_struct.c* et representée par la ligne 5 dans listing (1.6)). Donc la correction consiste à commenter la ligne 20 dans *lbm_struct.c* (representée par la ligne 6 dans listing (1.6)), où $mesh->cells$ est initialisée à *NULL* et décommenter la ligne où elle est allouée.

Listing 1.6: Correction de la faute de segmentation.

```
1  void Mesh_init( Mesh * mesh, int width,  int height )
2  {
3   ...
4   //alloc cells memory
5   mesh->cells = malloc( width * height  * DIRECTIONS * sizeof( double ) );
6   //mesh->cells = NULL;
7
8   //errors
9   if( mesh->cells == NULL )
10  {
11   perror( "malloc" );
12   abort();
13  }
14 }
```

De plus, on remarque qu'il y a un bout de code juste au dessous de l'allocation de *mesh* (lignes $23-27$ dans *lbm_struct.c* et representées par les lignes $9-13$ dans listing (1.6)) qui sert à vérifier que $mesh->cells$ est différente de NULL.

Après correction de la faute de segmentation, le programme s'exécute et affiche le progrès de calcul en fonction du nombre d'itérations.

1.2 Bug 2: Deadlock

Dans cette section, on se propose d'exécuter le code avec deux processus MPI en utilisant la commande ***mpirun -np 2 ./lbm***.

1.2.1 Identification du deadlock

Après exécution du 100 itérations de code avec deux processus MPI, cette exécution reste bloquée à la dernière itération, comme le montre la figure (1.1). Donc à priori il y a un bug après à la dernière itération ou pendant la terminaison du programme.

```
Progress [   95 /   100]
Progress [   96 /   100]
Progress [   97 /   100]
Progress [   98 /   100]
Progress [   99 /   100]
```

Figure 1.1: Blocage de l'exécution du code à la dernière itération.

Pour déboguer le code, on va utiliser une technique de débogage, gérée par l'utilisateur, qui est *fprintf on stderr (no bufferisation)*. Puisque on ne sait pas encore l'endroit exact du bug, un *fprintf* est mis juste à la fin de la boucle itérative, et un autre après la fermeture du fichier output, comme le montre listing (1.7). Ces deux *fprintf* affichent l'identifiant du processus.

Listing 1.7: Ajout de *fprintf* pour localiser le bug.

```
1  //time steps
2  for ( i = 1 ; i < ITERATIONS ; i++ )
3  {
4      ...
5      //save step
6      if ( i % WRITE_STEP_INTERVAL == 0 && lbm_gbl_config.output_filename != NULL )
7          save_frame_all_domain(fp, &mesh, &temp_render );
8      //First fprintf on stderr
9      fprintf(stderr, "rank %d: Before end of loop \n", rank);
10 } //end of loop
11
12 if( rank == RANK_MASTER && fp != NULL)
13 {
14     close_file(fp);
15 }
16 //second fprintf on stderr
17 fprintf(stderr, "rank %d: After closing the file \n", rank);
```

L'ajout de ces deux sorties donne l'affichage illustré dans la figure (1.2.a). On remarque que la sortie de *fprintf* ajoutée n'est pas cohérente avec la sortie du progress puisque cette dernière

est faite avec *printf* qui utilise probablement un buffer. Afin de rendre la sortie cohérente, le progress a été aussi afficher avec *fprintf on stderr*.

La figure (1.2.b) montre l'affichage cohérent à la fin du calcul. On remarque que les deux processus sortent de la boucle itérative, mais seulement le processus 1 passe l'étape de la fermeture du fichier d'output. La fermeture du fichier output est faite grâce à la routine *close_file* qui est exécuté uniquement par le *RANK_MASTER*, c'est à dire par le rang 0 dans notre cas (comme défini dans *lmb_comm.h*). En analysant la routine *close_file*, on remarque la présence d'un *MPI_Barrier*, ce qui oblige l'attente de tous les processus. La présence de cette commande ici n'est pas cohérente puisque *close_file* est exécuté seulement par un seul processus. Et voila !

```
Progress [   11 /   100]
rank 0: Before end of loop
Progress [   12 /   100]
rank 1: Before end of loop
rank 0: Before end of loop
rank 1: Before end of loop
rank 0: Before end of loop
Progress [   13 /   100]
Progress [   14 /   100]
rank 1: Before end of loop
rank 0: Before end of loop
rank 1: Before end of loop
rank 0: Before end of loop
```

(a) Présence d'incohérence avec la sortie du progress.

```
Progress [   99 /   100]
rank 1: Before end of loop
rank 0: Before end of loop
rank 1: Before end of loop
rank 1: After closing the file
```

(b) Mise en évidence du bug du Deadlock.

Figure 1.2: Affichage de sorties ajoutées pour le débogage du Deadlock.

1.2.2 Correction du bug du Deadblock

Le problème étant la présence d'un *MPI_Barrier* dans la routine *close_file* dans *main.c*, la solution consiste à commenter la ligne 63 de *main.c* (representée dans listing (1.8) par la ligne 2).

Listing 1.8: Correction du bug du Deadlock.

```
1 void close_file(FILE* fp){
2  //MPI_Barrier(MPI_COMM_WORLD);
3  fclose(fp); //close file
4 }
```

```
Progress [   98 /   100]
rank 1: Before end of loop
rank 0: Before end of loop
Progress [   99 /   100]
rank 1: Before end of loop
rank 1: After closing the file
rank 0: Before end of loop
rank 0: After closing the file
aicha@aicha-Vostro-3500:/media/aicha/DATA/Technique_Optimisation/TOP_LBM/simu_simple_LBM$
```

Figure 1.3: Affichage de sorties ajoutées après correction du bug du Deadlock.

Remarque:

Il est également possible d'utiliser d'autres techniques de débogage pour identifier ce bug, tel que *GDB*. Ce dernier permet d'avoir une fenêtre de débogage par processus MPI en utilisant *xterm -e* .

1.3 Conclusion:

Au cours de ce premier chapitre, on a pu identifier et corriger les bugs qui empêchaient la bonne exécution du code en séquentiel ou en parallèle. Cette étape est cruciale pour la suite de travail, puisqu'il faut déjà un code valide pour pouvoir ensuite l'optimiser. Les bugs rencontrés étaient de nature différente et aussi produisaient des messages d'erreur ou des comportements différents. La faute de segmentation produisait un message d'erreur qui ne donne pas beaucoup d'informations sur sa nature à première vue. Pour cela, on a préféré de passer par un outil de débogage (gdb) qui permet petit à petit de résoudre le problème. Concernant le deuxième bug, la sortie par défaut du programme nous a donné déjà une idée de l'emplacement du problème dans le code et l'ajout de *fprintf* était suffisant pour préciser le bug. Donc on peut conclure des outils de débogage différents peuvent être utilisés en fonction du problème rencontrer.

Chapter 2

Description & Evaluation du Code

Après l'identification des bugs qui empêchaient l'exécution du programme dans le chapitre 1, ce chapitre est consacré à la description et l'évaluation du code d'origine.

2.1 Structure du programme

La compréhension de la structure du programme est principalement faite par la lecture du code. La fonction *main* dans *main.c* donne un aperçu global du déroulement de l'exécution. Le flowchart dans la figure (2.1) décrit les différentes étapes du calcul. Dans un premier temps, on ne considère pas les fonctions liées à l'échange de données pour l'exécution parallèle (tout ce qui est lié à MPI). Les fonctions liées à l'écriture des fichiers output ne figurent non plus dans ce flowchart pour raison de simplicité.

Le programme commence par charger la configuration avec les différentes caractéristiques du domaine. Le domaine est décomposé en sous-domaines pour le calcul parallèle, ainsi chaque processus continue le calcul uniquement sur le sous-domaine qui lui est associé. Ensuite, la mémoire est allouée pour chaque sous-domaine et les variables de calcul sont initialisées. Après avoir commencé la boucle itérative où à chaque itération les étapes suivantes sont exécutées: Application des conditions conditions particulières, calcul des collisions et propagation des particules. A chaque intervalle d'itérations, le programme écrit un output. Après la boucle d'itération, le programme se termine par la fermeture du fichier d'output et la libération de la mémoire allouée.

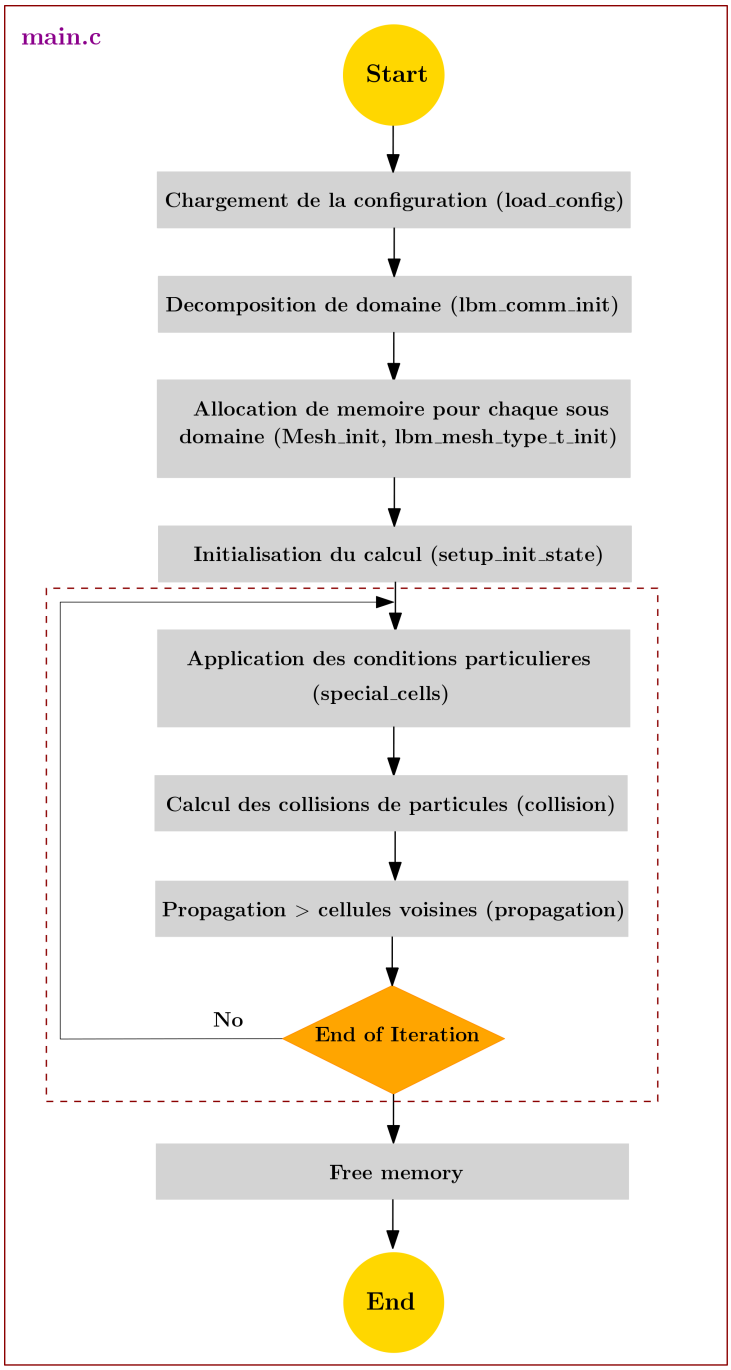

Figure 2.1: Flowchart du calcul sans prise en compte des communications MPI.

2.2 Description de la décomposition du domaine

La décomposition du domaine mérite une attention particulière. Sa compréhension détaillée est nécessaire pour maîtriser la parallélisation du programme. Cette étape est effectuée par la routine *lbm_comm_init* dans *lbm_comm.c*. D'abord, le domaine est découpé verticalement et horizontalement. D'après la structure générale de la routine, on peut comprendre que la direction horizontale est suivant l'axe des x et la direction verticale est suivant l'axe des y.

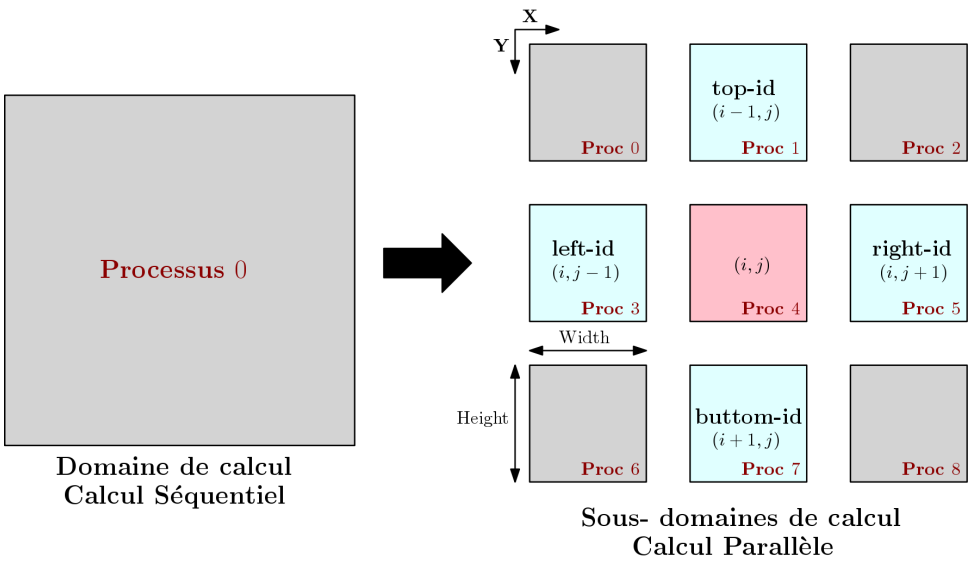

Figure 2.2: Décomposition du domaine en 9 sous-domaines.

La figure (2.2) illustre un exemple de décomposition en 9 sous-domaines, 3 colonnes et 3 lignes. On trouve deux implémentations du découpage fournies dans le code, dont une est commentée, comme le montre le listing (2.1). L'implémentation commentée est celle cohérente puisque le nombre de colonnes, *nb_x*, doit être en fonction de la taille du domaine dans la direction horizontale (*width*), alors que le nombre de ligne, *nb_y*, doit être en fonction de la taille du domaine dans la direction verticale (*height*).

Malgré que l'exécution du programme ne produit pas d'erreur avec l'implémentation incohérente, l'implémentation cohérente a été choisie et décommentée. Le nombre de lignes est donc d'abord calculé comme le plus grand diviseur commun entre la hauteur du domaine, *height*, et le nombre de proc disponibles. Ensuite le nombre de colonnes est déduit à partir du nombre de lignes. Donc le découpage se fait principalement horizontalement si la hauteur du domaine est un multiplicateur du nombre des procs. Cette méthode ne garantit pas de trouver une solution de découpage, d'où le test de vérification. Chaque processeur lui est attribué un sous-domaine comme le montre la figure (2.2).

Listing 2.1: Implémentation cohérente et incohérente du decoupage dans le code.

```
1 //Version incoherente
2 nb_y = lbm_helper_pgcd(comm_size,width);
3 nb_x = comm_size/nb_y;
4 // Version coherente
5 //nb_y = lbm_helper_pgcd(comm_size,height);
6 //nb_x = comm_size / nb_y;
```

2.2.1 Exemple de résultat

Après la modification du décomposition du domaine, la bonne exécution du code a été vérifiée. La figure (2.3) montre le résultat après 100 itérations. L'animation gif montre bien la propagation des particules suite à la réflection avec l'obstacle circulaire après l'initialisation du calcul. La validation du résultat a été vérifié avec le test de checksum comme le montre la figure (2.4) (Le premier nombre étant en hexadécimal et le deuxième en décimal).

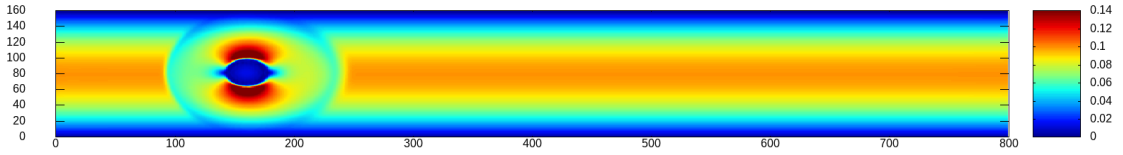

Figure 2.3: Propagation des particules suite à la réflection avec l'obstacle circulaire, étape 99/100.

```
aicha@aicha-Vostro-3500:/media/aicha/DATA/Technique_Optimisation/TOP_LBM/simu_simple_LBM$ ./display --checksum resultat.raw 1
21511 - 136465
aicha@aicha-Vostro-3500:/media/aicha/DATA/Technique_Optimisation/TOP_LBM/simu_simple_LBM$ ./display --checksum resultat.raw 2
21511 - 136465
aicha@aicha-Vostro-3500:/media/aicha/DATA/Technique_Optimisation/TOP_LBM/simu_simple_LBM$ ./display --checksum resultat.raw 4
21504 - 136452
```

Figure 2.4: Validation des résultats avec *checksum*.

2.3 Première évaluation de la scalabilité du code

Au premier lieu, on a évalué la scalabilité forte du code d'origine, qui consiste à garder la même quantité globale de travail, i.e., fixer le nombre d'itérations à **1000 itérations** et garder la même taille du domaine **(width = 800 cells, Height = 160 cells)** tout en variant le nombre du processus.

Idéalement, en diminuant la quantité de travail par unité de calcul, on obtient un speed-up proportionnel au nombre de processus. La variation du nombre de procs de 1 à 36 (avec un pas doublé à chaque fois) aboutit aux résultats de performance (temps de calcul) illustrés dans la figure (2.5).

L'écart anormal entre le temps réel et le temps utilisateur à chaque fois qu'on répartit le travail sur un nombre de procs donné montre qu'il y a une anomalie dans le code d'origine, qui sera mise en évidence en analysant les communications MPI dans la section suivante.

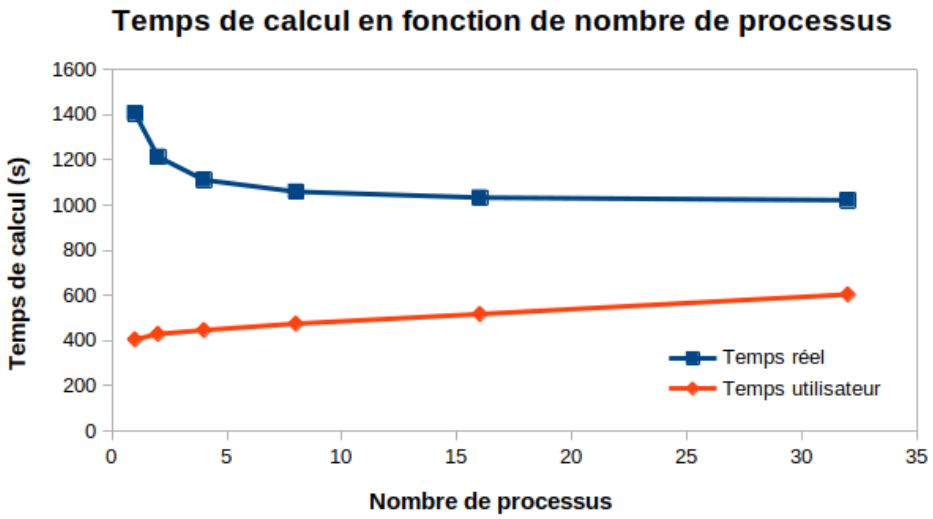

Figure 2.5: Temps de calcul réel et utilisateur en fonction de nombre de processus.

2.4 Description de communication MPI

On peut remarquer plusieurs parties dans le code qui sont dédiés aux communications entre les processeurs. Dès le découpage du domaine on peut remarquer la reservation de la mémoire pour des cellules fantômes supplémentaires. Les cellules fantômes servent à faire la liaison de données entre un sous-domaine avec ces voisins comme le montre la figure (2.6).

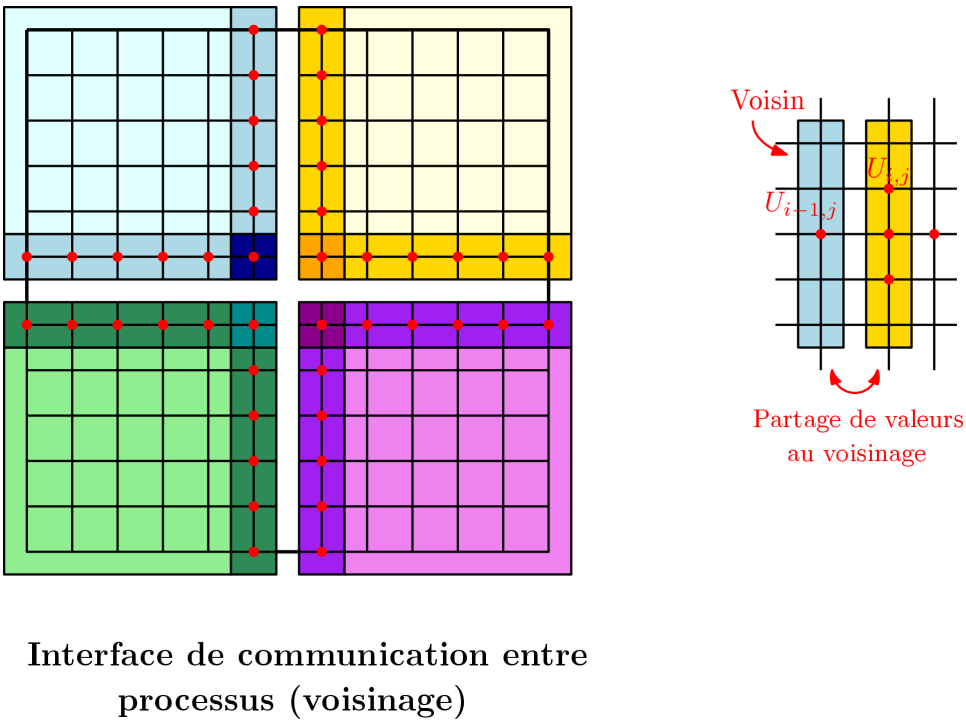

Figure 2.6: Schéma du calcul avec des communications bloquantes.

La mémoire a été aussi allouée pour un buffer qui sera utilisé éventuellement pour des communications asynchrone. La boucle itérative dans la fonction *main.c* contient plusieurs commandes *MPI_Barrier* comme le montre la figure (2.7).

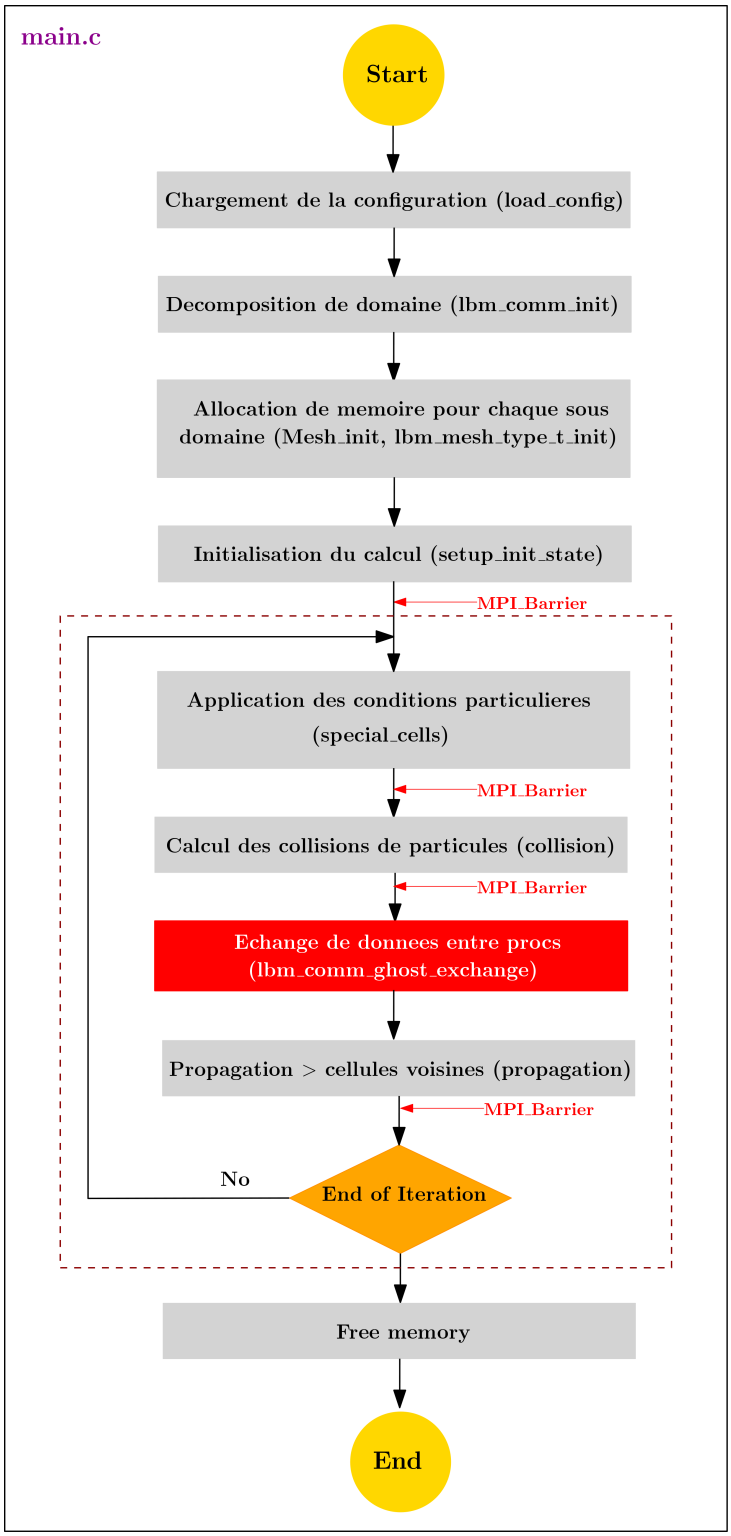

Figure 2.7: Flowchart du calcul avec prise en compte des communications MPI.

La présence de certaines de ces commandes semble injustifiée à priori, et ça doit faire l'objet

d'investigation dans le chapitre 3. Les communications MPI se trouvent principalement dans deux routines implémentées dans *lbm_comm.c*:

save_frame_all_domain

Puisque l'écriture du fichier output doit s'effectuer par un seul proc, tous les procs envoie leurs données au rang 0 avec une communication point à point bloquante (*MPI_Send* et *MPI_Recv*).

lbm_comm_ghost_exchange:

Cette routine assure l'échange de données entre les cellules fantômes. L'échange se fait toujours par communication point par point bloquante. L'appel de *MPI_Send* et *MPI_Recv* se fait par des routines qui adaptent l'envoie de rangée de cellule verticale (*lbm_comm_sync_ghosts_vertical*), horizontale (*lbm_comm_sync_ghosts_horizontal*), ou les cellules diagonales (*lbm_comm_sync_ghosts_diagonal*). On peut remarquer un *MPI_Barrier* après chaque appel de ces routines intermédiaires. La présence exhaustive de cette commande est aussi à investiguer dans le chapitre 3.
On remarque aussi que deux échanges sont dupliqués:

- La communication $CORNER\_BOTTOM\_LEFT \rightarrow CORNER\_TOP\_RIGHT$.
- La communication right to left phase.

En analysant la routine *lbm_comm_sync_ghosts_horizontal*, on peut remarquer que l'envoie des données se fait cellule par cellule, ce qui peut sûrement être optimisé. L'analyse de la routine *lbm_comm_sync_ghosts_vertical* montre que l'envoie des données se fait même direction par direction. On remarque aussi deux dans cette routine:

- Une incohérence dans la taille des données envoyées et reçues. Puisque on envoie un seul double, il est attendu de recevoir aussi un seul double, non pas le nombre total de directions.
- Le parcours des cellules suivant la direction horizontale se fait de $x = 1$ à $x < mesh\_to\_process \rightarrow width - 2$, ce qui fait $width - 3$. C'est cohérent de commencer à $x = 1$ puisque on souhaite éviter la cellule du coin. Cependant, on doit aller jusqu'à $x < mesh\_to\_process \rightarrow width - 1$.

On a préféré rendre le code plus cohérent comme le montre le listing (2.2). L'analyse des échanges horizontales et verticales montre que l'accès en mémoire au message à envoyer est plus

performant avec les échanges horizontales. Donc, on peut conclure il y a intérêt à favoriser le découpage vertical.

Listing 2.2: Routine *lbm_comm_sync_ghosts_vertical* de *lbm_comm.c*.

```
1  void lbm_comm_sync_ghosts_vertical( lbm_comm_t * mesh, Mesh *mesh_to_process,
       lbm_comm_type_t comm_type, int target_rank, int y )
2  {
3   ...
4   switch (comm_type)
5   {
6    case COMM_SEND:
7     //for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
8     for ( x = 1 ; x < mesh_to_process->width - 1 ; x++)
9      for ( k = 0 ; k < DIRECTIONS ; k++)
10      MPI_Send( &Mesh_get_cell(mesh_to_process, x, y)[k], 1, MPI_DOUBLE,
            target_rank, 0, MPI_COMM_WORLD);
11    break;
12   case COMM_RECV:
13    //for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
14    for ( x = 1 ; x < mesh_to_process->width - 1 ; x++)
15     for ( k = 0 ; k < DIRECTIONS ; k++)
16      // MPI_Recv( &Mesh_get_cell(mesh_to_process, x, y)[k], DIRECTIONS,
            MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD,&status);
17      MPI_Recv( &Mesh_get_cell(mesh_to_process, x, y)[k], 1, MPI_DOUBLE,
            target_rank, 0, MPI_COMM_WORLD,&status);
18    break;
19   default:
20    fatal("Unknown type of communication.");
21  }
22 }
```

A la fin de *lbm_comm_ghost_exchange*, la routine *FLUSH_INOUT* et le commentaire qui lui est associé attirent l'attention. Ici le code est censé traiter uniquement la communication entre les procs et normalement elle ne doit pas avoir de rapport avec les entrées/sorties. Cette routine est définie en tant que macro dans *lbm_config.h*. Elle revient à concaténer les lettres s,l,e,e,p en utilisant l'opérateur pour exécuter la routine *sleep(1)*. C'est bien caché !!! C'est très probablement le source de décalage entre le temps réel et le temps utilisateur pendant la première évaluation de scalabilité. **So it was VERY important to remove *FLUSH_INOUT*.**

2.5 Deuxième évaluation de la scalabilité du code

2.5.1 Scalabilité forte

Après avoir commenté la ligne de *FLUSH_INOUT()* dans *lbm_comm.c*, le temps réel devient quasiment égal au quotient du temps utilisateur sur le nombre de processus. On refait l'étude de la scalabilité forte évaluée dans la section 2.3 comme le montre la figure (2.8). Le speed-up est définie comme étant le bénéfice d'un calcul parallèle par rapport au même algorithme sur un unique processus.

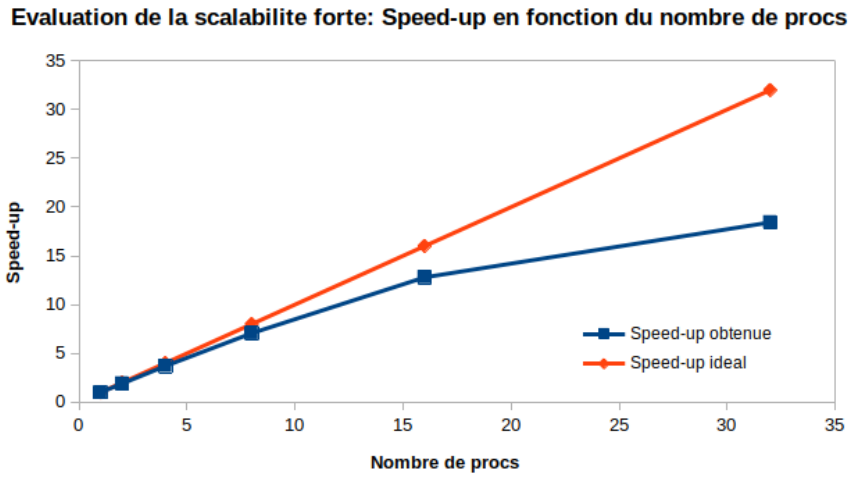

Figure 2.8: Analyse de la scalabilité forte du code d'origine, taille du domaine (en cells) = 800×160, cas de 1000 itérations.

2.5.2 Scalabilité faible

La scalabilité faible consiste à garder la même quantité de travail par unités de calcul. Pour évaluer la scalabilité faible, on fixe le nombre d'itérations à 1000, ainsi que le *Height* du taille du domaine à 160 *cells* et on varie le *width* en variant le nombre de processus, soit 250 *cells* par processus. Soit le tableau (2.1) récapitulatif de la taille du domaine en fonction du nombre de processus.

Nombre de proc	Width (cells)	Height (cells)	Nombre de proc	Width (cells)
1	250	160	4	1000
2	500	160	16	4000
8	2000	160	-	-

Table 2.1: Taille du domaine en fonction du nombre de processus.

La variation du speed-up dans le cas de la scalabilité faible en fonction du nombre de proc est illustrée dans la figure (2.9).

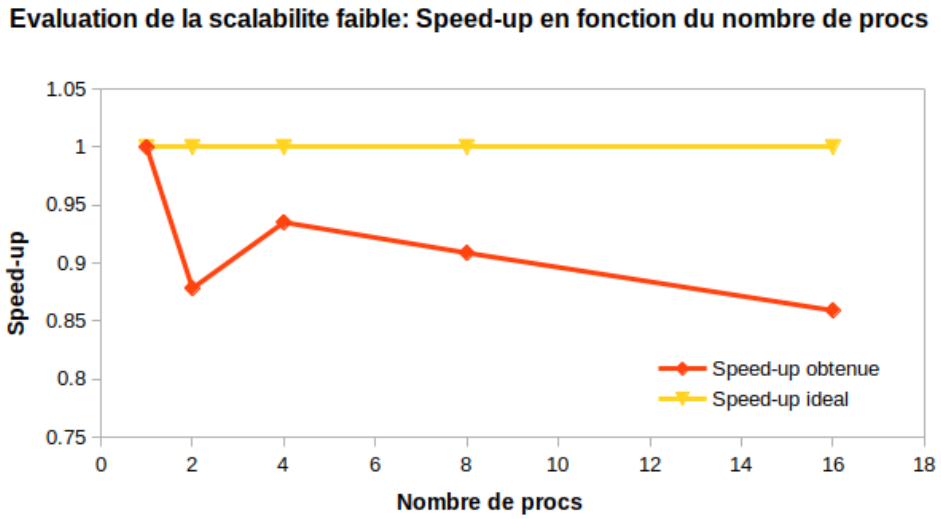

Figure 2.9: Analyse de la scalabilité faible du code d'origine, taille du domaine (en cells) = $[250 : 4000] \times 160$, cas de 1000 itérations.

D'après les figures (2.8) et (2.9), on peut conclure que:

- **Scalabilite forte:** le code d'origine suit bien le speed-up ideal jusqu'a 4 processus, ensuite la performance commence à se dégrader. A 32 processus, le code a perdu environ 49% par rapport au speed-up ideal,
- **Scalabilite faible:** le code d'origine n'est pas faiblement scalable. Cette dernière a tendance à se dégrader de plus en plus tout en augmentant le nombre de processus. Néanmoins il faut garder en tête que la taille du domaine a été augmentée dans la largeur, alors que l'augmentation du domaine d'autres manières peut donner d'autres résultats.

2.6 Profilage de code

Le profilage de code consiste à analyser son exécution, afin de mesurer son comportement lors de l'exécution (fonctions appelées et temps passé, etc.)

On se propose dans cette section de déterminer les fonctions ayants le plus grand temps d'exécution en utilisant *perf*, *"Perf Wiki"* 2020 et *Gprof*, *"GPROF Tutorial"* 2012. La taille du domaine a été fixée à 800 *cells* $\times$ 160 *cells* pour un nombre d'itérations de 1600 et un interval d'écriture égal à 50.

2.6.1 Profilage de code avec Perf

Le profilage de code d'origine est possible avec *perf* en ajoutant la flag de compilation *-g* dans le *Makefile* et en utilisant les commandes présentes dans listing (2.3).

Listing 2.3: Commandes bash pour le profilage de code avec *perf*.

```
1 $ sudo perf record ./lbm
2 $ sudo perf report
```

On obtient ainsi une liste des fonctions classées par coût, comme illustré dans la figure (2.10).

```
Samples: 284K of event 'cycles', Event count (approx.): 289282324180
Overhead  Command   Shared Object                  Symbol
  22,90%  lbm       lbm                            [.] propagation
  19,89%  lbm       lbm                            [.] get_vect_norme_2
  17,17%  lbm       lbm                            [.] get_cell_velocity
  13,17%  lbm       lbm                            [.] compute_equilibrium_profile
  10,59%  lbm       lbm                            [.] compute_cell_collision
   8,76%  lbm       lbm                            [.] Mesh_get_cell
   5,57%  lbm       lbm                            [.] get_cell_density
   0,55%  lbm       lbm                            [.] collision
   0,42%  lbm       lbm                            [.] special_cells
   0,31%  lbm       lbm                            [.] lbm_cell_type_t_get_cell
   0,11%  lbm       lbm                            [.] compute_bounce_back
   0,04%  lbm       lbm                            [.] save_frame
   0,03%  lbm       [kernel.kallsyms]              [k] acpi_os_read_port
   0,02%  orted     [kernel.kallsyms]              [k] pci_conf1_read
   0,02%  lbm       [kernel.kallsyms]              [k] native_queued_spin_lock_slowpath
   0,02%  lbm       lbm                            [.] Mesh_get_cell
   0,02%  lbm       [kernel.kallsyms]              [k] native_read_msr
   0,02%  lbm       libm-2.31.so                   [.] __sqrt_finite@GLIBC_2.15
   0,02%  lbm       lbm                            [.] setup_init_state_global_poiseuille_profile
   0,01%  lbm       lbm                            [.] helper_compute_poiseuille
   0,01%  lbm       [kernel.kallsyms]              [k] native_write_msr
   0,01%  lbm       [kernel.kallsyms]              [k] sync_regs
   0,01%  lbm       [kernel.kallsyms]              [k] perf_event_task_tick
   0,01%  lbm       [kernel.kallsyms]              [k] update_curr
   0,01%  orted     [kernel.kallsyms]              [k] delay_tsc
   0,01%  lbm       [kernel.kallsyms]              [k] native_write_msr_safe
   0,01%  lbm       [kernel.kallsyms]              [k] timekeeping_advance
   0,01%  lbm       [kernel.kallsyms]              [k] _raw_spin_lock_irqsave
   0,01%  lbm       [kernel.kallsyms]              [k] cpuacct_account_field
   0,01%  lbm       [kernel.kallsyms]              [k] timerqueue_add
   0,00%  lbm       lbm                            [.] lbm_cell_type_t_get_cell
   0,00%  lbm       libmpi.so.40.20.3              [.] PMPI_Barrier
```

Figure 2.10: Profilage du code d'origine avec *Perf*.

D'après la figure (2.10), la routine *propagation* passe plus de temps à s'exécuter, suivie par *get_vect_norme_2*. Les pourcentages du temps indiquées en rouge montrent les fonctions critiques qui nécessitent plus de temps d'exécution.

2.6.2 Profilage du code avec Gprof

Le profilage de code d'origine est possible avec *Gprof* en ajoutant la flag de compilation *-pg* dans le *Makefile* et en utilisant les commandes présentes dans listing (2.4).

Listing 2.4: Commandes bash pour le profilage de code avec *Gprof*.

```
$ ./lbm
$ gprof lbm gmon.out > analysis.txt
$ code analysis.txt
```

On obtient ainsi une liste des fonctions classées par coût (en mode exclusif), comme illustré dans la figure (2.11).

analysis.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 22.79     16.51    16.51 208768000     0.00     0.00  get_cell_velocity
 14.90     27.31    10.80 3692927008     0.00     0.00  get_vect_norme_2
 14.64     37.92    10.61     1599     6.64    10.41  propagation
 13.48     47.69     9.77 1844415504     0.00     0.00  compute_equilibrium_profile
 13.28     57.32     9.63 204672000     0.00     0.00  compute_cell_collision
  9.26     64.03     6.71 4131772827     0.00     0.00  Mesh_get_cell
  7.21     69.26     5.22 208768000     0.00     0.00  get_cell_density
  2.62     71.16     1.90 204672000     0.00     0.00  lbm_cell_type_t_get_cell
  1.03     71.90     0.75                             main
  0.44     72.22     0.32     1599     0.20     1.47  special_cells
  0.21     72.37     0.15     1599     0.09    32.70  collision
  0.18     72.50     0.13  1440699     0.00     0.00  compute_bounce_back
  0.03     72.52     0.02       32     0.63    14.33  save_frame
  0.03     72.54     0.02        2    10.01    23.04  setup_init_state_global_poiseuille_profile
```

Figure 2.11: Profilage du code d'origine avec *Gprof*.

D'après la figure (2.11), on remarque qu'en utilisant *Gprof*, la routine *get_cell_velocity* consomme plus de temps en s'exécutant, suivie par la routine *get_vect_norme_2*. Les fonctions ayant le plus grand temps d'exécution sont classées en ordre décroissant dans *Gprof*, comme le montre la figure (2.11).

La comparaison de résultats de profilage du code avec *Gprof* et *Perf* est expliquée par le fait que *Gprof* donne le temps passés des fonctions en mode exclusif, différemment à *Perf* qui est en mode inclusif.

2.6.3 Débogage mémoire avec Valgrind

L'évaluation des fuites mémoires est possible avec *Valgrind* en ajoutant la flag de compilation *-g* dans le *Makefile* et en utilisant la commande présente dans listing (2.5).

Listing 2.5: Commande bash pour le débogage mémoire avec Valgrind.

```
$ valgrind --tool=memcheck --leak-check=full ./lbm
```

L'option *–leak-check=full* permet de voir des détails sur la fuite mémoire du code d'origine. En exécutant le programme avec Valgrind comme montré dans listing (2.5), on obtient un résumé du tas (Heap) et des fuites mémoires (leak) comme illustré dans la figure (2.12).

```
==32919==
==32919== HEAP SUMMARY:
==32919==     in use at exit: 93,389 bytes in 85 blocks
==32919==   total heap usage: 26,084 allocs, 25,999 frees, 33,989,750 bytes allocated
```

(a) HEAP Summary.

```
==32919== LEAK SUMMARY:
==32919==    definitely lost: 8,222 bytes in 36 blocks
==32919==    indirectly lost: 599 bytes in 20 blocks
==32919==      possibly lost: 0 bytes in 0 blocks
==32919==    still reachable: 84,568 bytes in 29 blocks
==32919==         suppressed: 0 bytes in 0 blocks
```

(b) LEAK Summary.

Figure 2.12: Débogage mémoire avec *Valgrind*.

Après analyse des résultats de débogage mémoire avec *Valgrind*, on peut constater que la taille de la fuite mémoire reste relativement limitée (de l'ordre de *koctet*). Cette fuite mémoire n'est pas due à l'allocation des variables utilisées pour le calcul et les communications.

2.7 Conclusion

L'analyse et le profilage du code sont des étapes nécessaires pour évaluer ses performances et avoir des idées afin de l'optimiser. Au cours du deuxième chapitre, on a pu analyser le code d'origine, la méthode de décomposion de domaine, ainsi que les différentes communications *MPI* ce qui a permis de repérer quelques faiblesses et incohérences. Cette analyse a mis en évidence entre autre la présence de communications dupliqués, la présence excessive de *MPI_Barier*, et surtout la présence d'un temps d'attente d'une seconde à chaque itération de calcul. Ensuite, l'étude de la scalabilité forte a montré qu'on obtient seulement environ 50% du speed-up idéal avec 32 processeurs (évaluation sans le sleep). Le profilage du code avec Gprof et Perf a permis de repérer les anomalies et les routines ayant le plus de temps d'exécution.

Chapter 3

Optimisation

3.1 Flags de compilation

On se propose dans cette partie de diminuer le temps d'exécution en ajoutant des flags de compilation pour le compilateur *gcc*, *"Using the GNU Compiler Collection (GCC)"* 2022.

La taille du domaine (en cells) est fixée à 800×160 pour un nombre d'itérations égal à 1000. Le calcul étant séquentiel, l'exécution du code est faite sur un seul processus.

Le tableau (3.1) regroupe les flags de compilation ajoutés au fichier *Makefile*, dans *CFLAGS*.

Flag	**Description**	**Temps Exec.**
Default (gcc -Wall)	-	405.49 sec
-DNDEBUG	Desactive les asserts dans le code (une fois valide)	401.58 sec
-O2	Recommandé par Intel pour utilisation générale	102.34 sec
-O3	Optimisation (calcul intensif en virgule flottante)	58.19 sec
-Ofast	-O3 plus quelques extras	56.03 sec
-march=native -msse4.2	activer les extensions SSE	50 sec

Table 3.1: Ajout de flags de compilation pour optimiser le temps de calcul.

Le speed-up suite à l'ajout des flags de compilation est illustré dans la figure (3.1).

Dans la suite, les flags de compilations utilisés sont donnés dans listing (3.1).

Listing 3.1: Flags de compilation utilisés dans *Makefile*.

```
1 CFLAGS=-Wall -DNDEBUG -Ofast -march=native -msse4.2
```

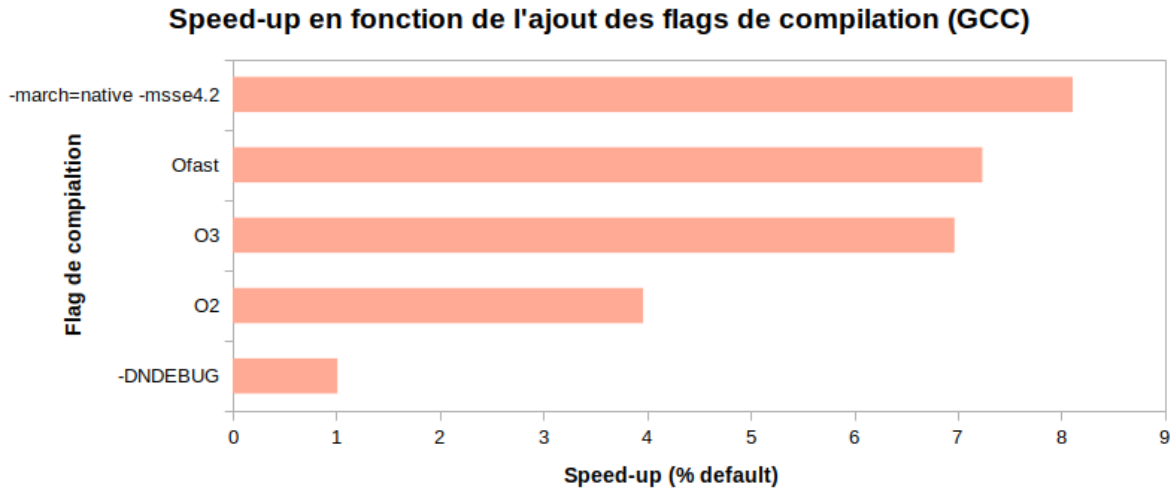

Figure 3.1: Speed-up obtenu lors de l'ajout des options d'optimisation (Flags de compilation), domaine = 800×160 et nombre d'itérations = 1000.

Dans la suite, on considère une taille fixe de domaine (en cells) évaluée à 1592×5000. Le nombre d'itérations étant 300, on se propose dans la suite d'essayer des optimisations au niveau communications MPI afin de réduire de plus en plus le temps de calcul sur un nombre de processus égal à 64. Les dimensions du domaine et le nombre de processus choisi mènent à un découpage 8×8, ce qui garantit d'avoir tous les types de communications.

3.2 Suppression des duplications de communication

Suite à la description des communications MPI fournie dans le chapite 2 section 2.4, plusieurs points d'amélioration ont été identifiés. On commence ici par supprimer les communications dupliquées ($CORNER\_BOTTOM\_LEFT \rightarrow CORNER\_TOP\_RIGHT$ et right to left phase). Afin que cette modification ait un effet sur les résultats, il est logiquement nécessaire d'avoir à la fois un découpage vertical et horizontal. La figure (3.2) montre le temps d'exécution avant et après la suppression des échanges dupliqués. Le gain est très faible (environ 1%) et rentre dans l'incertitude de la mesure du temps d'exécution. Donc, cette modification n'a pas d'effet sur les performances du cas test considéré. L'utilisation d'un domaine plus grand et/ou l'utilisation de plus de processus peut mener à des différences plus importantes.

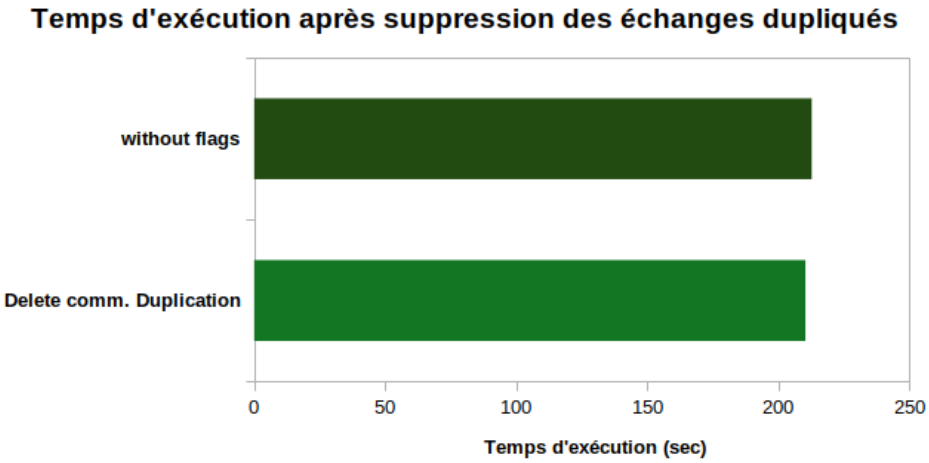

Figure 3.2: Temps d'exécution avant et après la suppression des échanges dupliqués.

3.3 Suppression des MPI_Barrier

On investigue maintenant la pertinence et l'effet de nombreux *MPI_Barrier* dans les routines *main* et *lbm_comm_ghost_exchange*. D'abord, il n'y a pas de justificatif de la présence de cette commande dans la boucle itérative entre des étapes de calcul où les communications MPI n'interviennent pas. Ainsi un premier test consiste à supprimer toutes les *MPI_Barrier* dans la routine *main*, sauf celle qui précède l'appel de la routine *lbm_comm_ghost_exchange*. L'échange des données entre les sous-domaines est assuré par des communications bloquantes et il n'est pas nécessaire de les synchroniser. On effectue donc un deuxième test qui consiste à supprimer en plus tous les *MPI_Barrier* dans la routine *lbm_comm_ghost_exchange*.

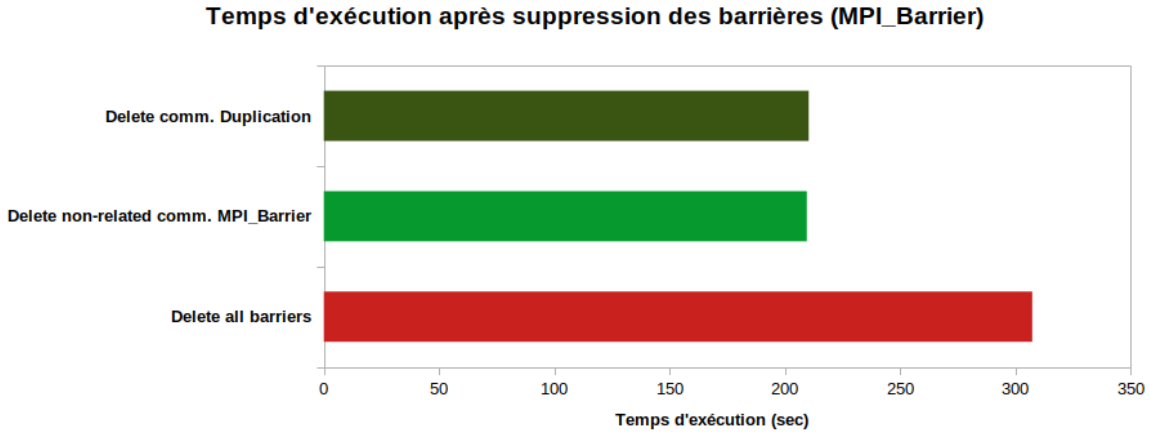

Figure 3.3: Temps d'exécution avant et après suppression des de barrières.

La figure (3.3) montre l'effet de ces deux tests sur le temps d'exécution. Le premier test n'a quasiment aucun effet sur les performances. La suppression de *MPI_Barrier* entre les

échanges a par contre un effet négatif et le temps d'exécution augmente d'environ 46%. On a l'impression que la synchronisation des échanges bloquants entre de tous les processus avec *MPI_Barrier* donne un meilleur ordonnancement des communications. De plus, l'implémention jusqu'à maintenant comporte un grand nombre de communications non optimisé qui avoir de l'influence sur l'ordenancement. Uniquement la première modification a été retenue pour la suite de l'étude avec les communications bloquantes.

3.4 Diminution des Send et Recv

L'analyse des communications dans la section a souligné que l'échange des données se fait cellule par cellule, voir même direction par direction. Donc il est important de réduire ces échanges et envoyer les données par bloc. Les données de calcul de chaque sous-domaine sont stockées dans la variable *Mesh→ cell*, qui est une liste de double. L'organisation de ces des cellules dans cette liste se fait colonne par colonne le montre la figure (3.4).

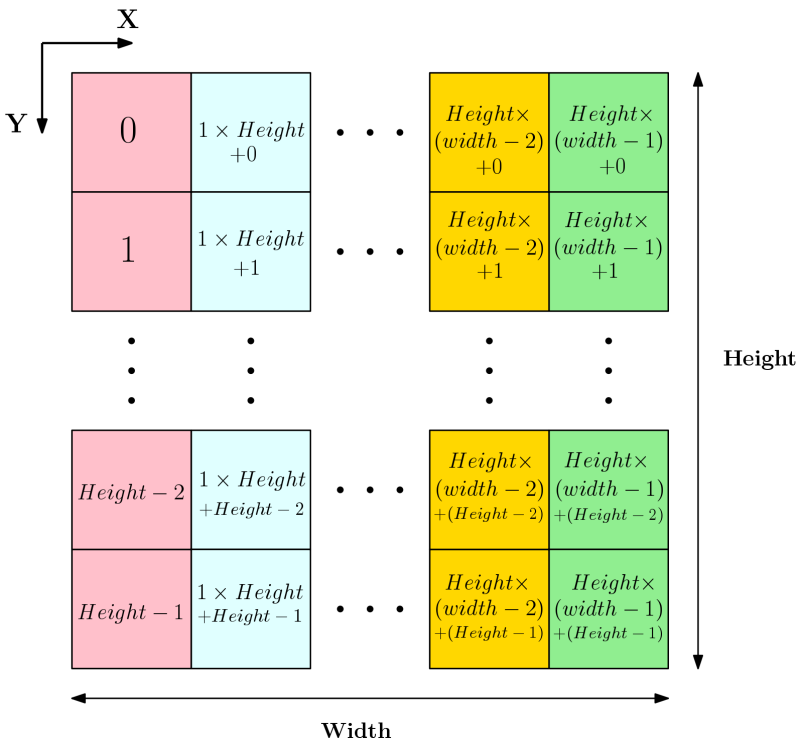

Figure 3.4: Alignement des cellules d'un sous-domaine dans la mémoire.

Deux getters facilitent l'accès à cette liste: *Mesh_get_cell* qui permet de récupérer une cellule du maillage en fonction de ces coordonnées, et *Mech_get_col* qui permet de récupérer la première cellule d'une colonne du maillage. Ainsi, il est possible de regrouper l'envoie des données comme suit:

- **Pour les échanges horizontaux:** L'envoie d'une colonne entière est directe puisqu'il suffit de donner l'adresse de la première cellule de la colonne et sa taille en mémoire (nombre de cellules × DIRECTIONS).
- **Pour les échanges verticaux:** Dans ce cas on a besoin d'envoyer une ligne de cellules qui ne sont pas alignées dans la mémoire. On utilise alors un buffer qui regroupe dans une première étape ces cellules pour un seul envoi. C'est l'opération inverse qui est faite lors de la réception, on récupère tous dans un buffer ensuite on répartit sur les cellules. Le listing (3.2) montre l'implémentation de ces deux manipulations. La déclaration et l'allocation en mémoire du buffer a déjà été fournie dans le code d'origine. Ici on peut se permettre d'utiliser le même buffer pour toutes les communications, puisque elles sont bloquantes.

Listing 3.2: Flags de compilation utilisés dans *Makefile*.

```
1 //Version coherente (horizontal)
2 /* nb_y = lbm_helper_pgcd(comm_size,height);
3 nb_x = comm_size/nb_y;*/
4
5 // Version verticale (decoupage)
6 nb_x = lbm_helper_pgcd(comm_size,width);
7 nb_y = comm_size / nb_x;
```

Cette réduction de communications permet de gagner environ 6% du temps d'exécution, comme le montre la figure (3.5).

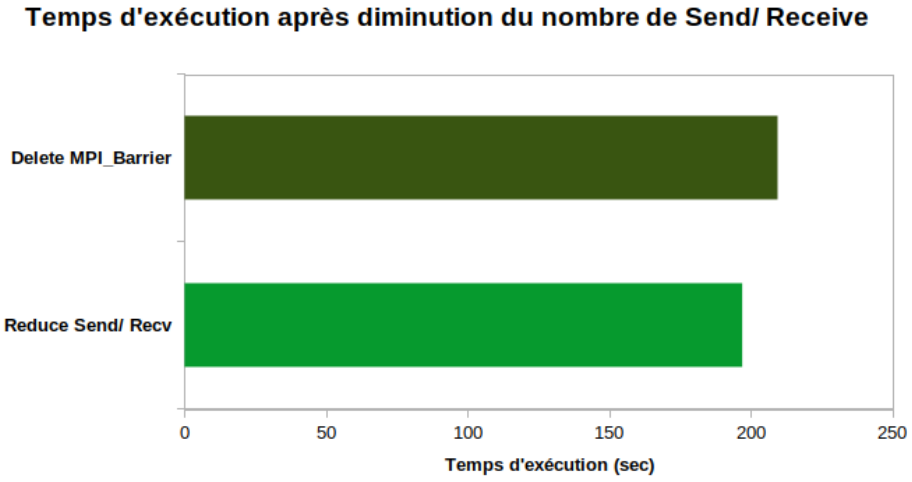

Figure 3.5: Temps d'exécution avant et après diminution du nombre de Send/ Receive.

3.5 Communication paire-impaire

Les commandes *MPI_Send* et *MPI_Recv* sont organisées de la même manière pour tous les processeurs. Puisque ces commandes sont bloquantes, la réception des données du voisin de gauche par exemple est conditionnée par la terminaison de l'envoie des données vers le voisin de droite. Ceci crée une chaîne de dépendance de communication et réduit la performance. Afin d'éviter ce problème, la solution classique consiste à alterner les commandes d'envoi et de réception entre les processeurs pair et impaire, comme montré dans listing (3.3).

Listing 3.3: Communication paire-impaire dans la routine *lbm_comm_ghost_exchange*.

```
1 void lbm_comm_ghost_exchange(lbm_comm_t * mesh, Mesh *mesh_to_process )
2 {
3  ...
4
5  if (rank % 2 == 0){
6   // on envoie a gauche puis on re oit a droite
7  } else {
8      // on re oit a droite puis on envoie a gauche
9  }
```

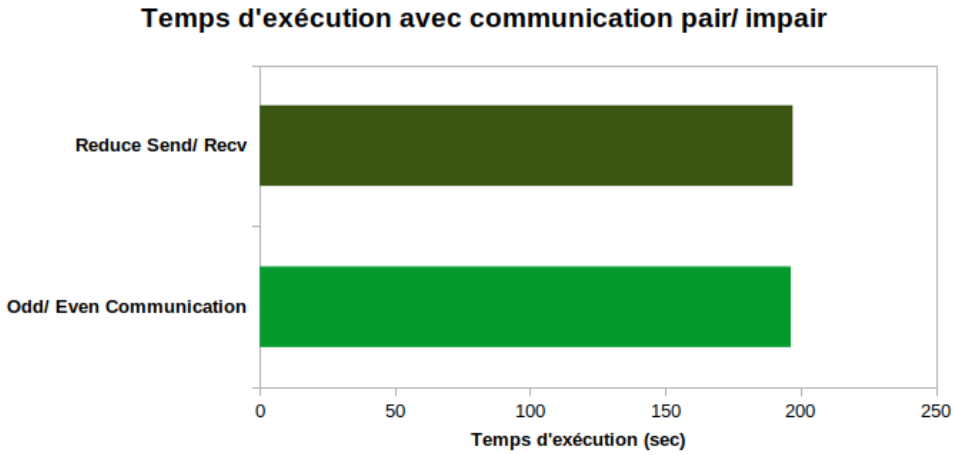

Figure 3.6: Temps d'exécution avant et après communication paire-impaire.

La figure (3.6) montre le temps d'exécution après l'implémentation de cette solution. Dans ce cas, cette solution n'a pas d'influence sur la performance. Il faut garder en tête que le découpage de ce cas test est 8×8. Un domaine plus grand avec plus de sous-domaines peut plus mettre en évidence le gain suite à la réduction du nombre des échanges et les échanges pair/impair.

3.6 Communication non-bloquante

Jusqu'à maintenant, les échanges reposent sur des communications bloquantes avec la présence de *MPI_Barrier* après chaque couple envoi/réception. La réduction du nombre des échanges ou la mise en place des communications pair/impair ont permis un gain très limité des performances. Il est intéressant de tester la performance avec des communications non bloquantes en utilisant *MPI_Isend* et *MPI_Irecv*. Cette approche a l'avantage de permettre le recouvrement, i.e. passer à d'autres instructions (autres communications ou calcul interne) pendant le temps d'attente d'une communication. Les routines de communication ont été dupliquées et adaptées pour cette approche (avec *async* au lieu de *sync* dans leurs noms). Toutes les *MPI_Barrier* ont été supprimées et la commande *MPI_Waitall* a été ajoutée à la fin des communications. Deux points d'implémentation méritent d'être détaillés:

- **Suivie des requêtes:** Les commandes *MPI_Isend* et *MPI_Irecv* demandent en argument une requête, de type *MPI_request*. C'est grâce à cette requête qu'on peut demander la terminaison de la communication avec *MPI_Waitall*. Au maximum, chaque processeur doit faire 16 communications: 8 envois (avec le haut, bas, droite, gauche et 4 coins) et autant de réceptions. Dans la structure utilisée pour le découpage du domaine (*lbm_comm_t*) on déclare alors un tableau de 16 requêtes (et non pas 32). Cependant, certains sous-domaines se trouvent au bord du domaine et ne nécessitent pas toutes les 16 communications. Afin de connaitre le nombre des communications et attribuer une requête à chacune, un entier (*id_req*) a été ajouté dans la structure *lbm_comm_t* et utilisé comme indice du tableau des requêtes. Cet indice est initialisé à 0 au début des communications, ensuite il est incrémenté après chaque utilisation pour l'utiliser à nouveau pour la communication suivante. Enfin, grâce à cet indice, la commande *MPI_Waitall* peut connaître le nombre total de communications à attendre.

- **L'utilisation du buffer:** Comme expliqué dans section 3.3, un buffer est utilisé pour l'échange des données par bloc pour les communications verticales. Ce buffer peut être utilisé (au maximum) pour 4 communications: envoie/réception avec le voisin du haut, et de même avec le voisin du bas. L'utilisation de communication non bloquante peut poser problème puisqu'il est possible de réutiliser le buffer alors qu'il contient encore des données non envoyées. Pour éviter ce problème, 4 buffers ont été déclarés est alloués de la même manière que précédemment. Des flags sont utilisés pour connaître quel buffer à

utiliser pour quelle communication. Le listing (3.4) illustre une partie de l'implémentation de la routine de communication verticale asynchrone.

Listing 3.4: Partie d'implémentation de la routine de communication verticale asynchrone.

```
1  void lbm_comm_async_ghosts_vertical( lbm_comm_t * mesh, Mesh *mesh_to_process,
       lbm_comm_type_t comm_type, int target_rank, int y, int line_pos)
2  {
3      ...
4   if(comm_type==COMM_SEND) {
5     if(line_pos == TOP_LINE_SEND) {
6      for ( x = 1 ; x < mesh_to_process->width - 1 ; x++)
7       memcpy(&mesh->buffer_upper_send[(x-1)*DIRECTIONS], Mesh_get_cell(
            mesh_to_process, x, y), sizeof(double) * DIRECTIONS);
8      MPI_Isend( &mesh->buffer_upper_send[0], DIRECTIONS * (mesh_to_process->width
           - 2), MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD, &mesh->requests[mesh->
           id_req]);
9      mesh->id_req++;
10    }
11    else if(line_pos == BOTTOM_LINE_SEND) {
12              ...
13    }
14  }
15  else if(comm_type==COMM_RECV) {
16  ...
17  }
18 }
```

La figure (3.7) compare le temps d'exécution obtenu avec les communications bloquantes ou non-bloquantes. Les communications non-bloquantes permettent de gagner environ 4% de performance. C'est-à-dire que les processeurs ont pu économiser une partie du temps d'attente pour entamer d'autres communications.

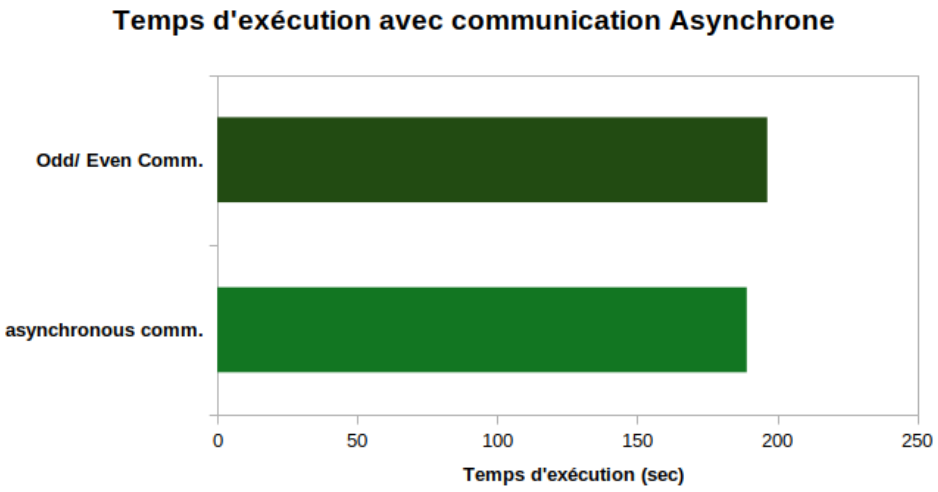

Figure 3.7: Temps d'exécution avant et après la communication asynchrone.

3.7 Méthode de découpage du domaine

On s'est intéressé dans les sections 3.2 jusqu'à 3.6 à l'optimisation de l'utilisation de MPI pour l'échange des données. La méthode de découpage du domaine peut avoir aussi de l'influence sur les performances. Comme expliqué dans la section 2.2, le nombre de découpes horizontales est d'abord calculé comme le plus grand diviseur en commun (pgcd) entre la hauteur du domaine et le nombre de processeurs disponibles (voir listing (2.1)). Dans un calcul standard, souvent le nombre de processeur est exprimé en puissance de 2 et les dimensions du domaines sont arrondi à la centaine, voir au millier. Donc il est très probable que le pgcd soit grand et que le découpage soit favorisé dans le sens horizontal, ce qui favorise les communications verticales. Alors que, comme détaillé dans la section 3.4, et à cause de l'alignement des cellules dans la mémoire, l'échange des données dans les communications verticales n'est pas direct est nécessite une étape supplémentaire de parcours et regroupement dans un buffer. Afin d'éviter cette étape, on a intérêt à favoriser les communications horizontales en favorisant les découpes verticales. Pour ce faire, il suffit de calculer le nombre de découpes verticales d'abord comme le pgcd du nombre de processeurs et la largeur du domaine comme le montre le listing (3.5).

Listing 3.5: Découpage de domaine.

```
1 // Version verticale (decoupage)
2  nb_x = lbm_helper_pgcd(comm_size,width);
3  nb_y = comm_size / nb_x;
4
5 //Horizontal
```

```
6  // nb_y = lbm_helper_pgcd(comm_size, height);
7  // nb_x = comm_size / nb_x;
```

Le test de cette modification dans le cas considéré depuis la section 3.2 n'aurait pas de sens puisqu'il a été dimensionné pour avoir toutes les types de communication avec un découpage 8×8. Dans cette section on considérer un autre dimensionnement qui va plutôt accentuer l'effet recherché: 4480×1792 avec toujours 300 itérations et 64 processeurs. La hauteur et la largeur du domaine sont toutes les deux des multiples de 64. Ainsi, tout le domaine sera découpé soit totalement horizontalement soit totalement verticalement en fonction de la méthode appliquée. La figure (3.8) compare le temps d'exécution avec les deux méthodes.

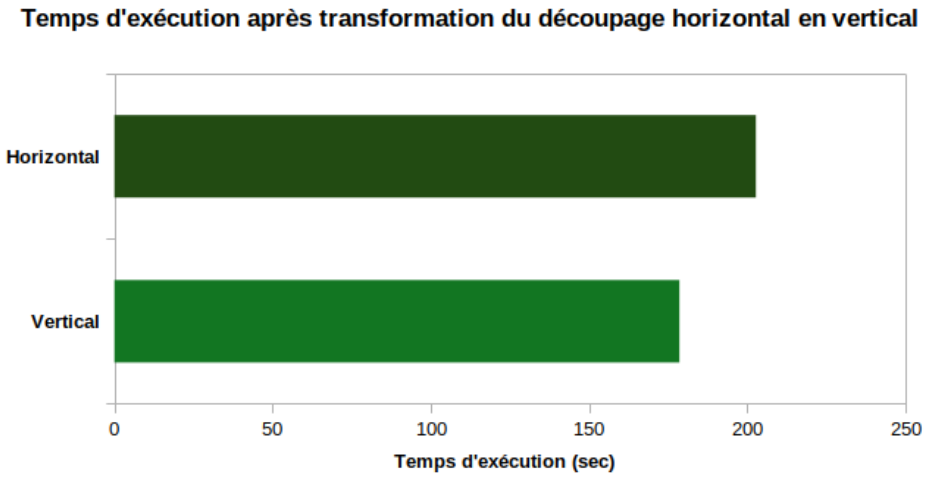

Figure 3.8: Temps d'exécution avant et après le découpage du domaine en vertical.

Comme attendue, le découpage vertical est plus performant avec un gain d'environ 12% par rapport au découpage horizontal. On rappelle bien sûr que le cas considéré a été dimensionné pour accentuer cet effet. Un autre point important à considérer lors du découpage du domaine dans des cas pratique c'est l'équilibrage de calculs entre les les processeurs (ou load balancing). Ce point n'a pas été traité dans ce projet puisque l'équilibrage est déjà parfait.

3.8 Correction de l'ordre des boucles

Le profilage du code avec *Perf* (section 2.6.1) et *Gprof* (section 2.6.2) montrent que les routines qui coûtent le plus en temps cpu sont: *propagation*, *get_vect_norme_2*, *get_cell_velocity*, *compute_equilibruim_profile* et *compute_cell_collision*. Toutes ces routines sont implémentées dans le fichier *lbm_phys.c*. Donc il serait intéressant d'essayer d'optimiser aussi cette partie du code.

On peut remarquer les routines *collision* et *propagation* la présence d'une double boucle for qui parcourt le maillage du sous-domaine horizontalement puis verticalement. Cependant, comme expliqué dans la section 3.4, les cellules sont alignées dans la mémoire verticalement (voir figure 3.4). Donc on a intérêt à inverser l'ordre du parcours pour bénéficier d'un accès plus facile à la mémoire, comme le montre le listing (3.6). En effet, les cellules voisines ont plus de chances d'être déjà chargées dans la mémoire cache la plus proche à l'unité de calcul. A partir de cette section, on revient au dimensionnement du cas considéré depuis la section 3.2.

Listing 3.6: Ordonnancement de boucles dans *lbm_phys.c*.

```
1  void collision(Mesh * mesh_out,const Mesh * mesh_in)
2  {
3   ...
4
5   //loop on all inner cells
6   //for( j = 1 ; j < mesh_in->height - 1 ; j++  )
7   for( i = 1 ; i < mesh_in->width - 1 ; i++  )
8    //for( i = 1 ; i < mesh_in->width - 1 ; i++)
9    for( j = 1 ; j < mesh_in->height - 1 ; j++)
10     ...
11 }
```

La figure (3.9) montre l'impact de cette modification sur le temps d'exécution. Le gain est d'environ 11%.

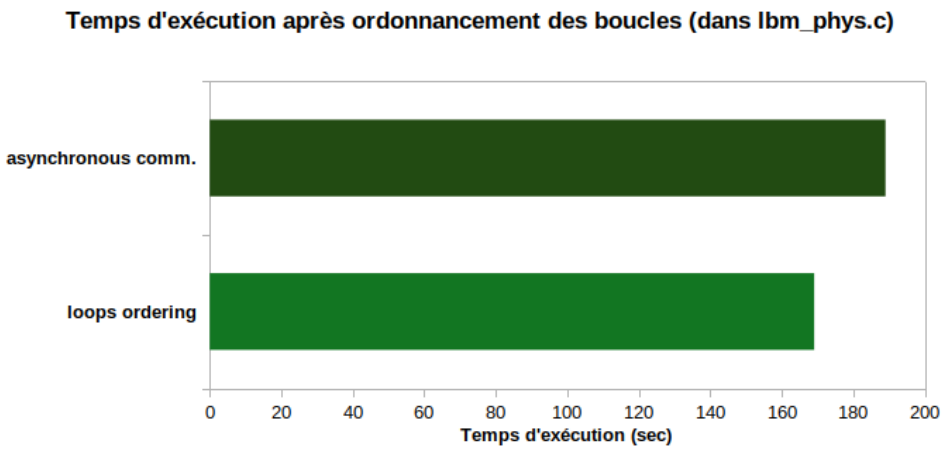

Figure 3.9: Temps d'exécution avant et après ordonnancement de boucles dans *lbm_phys.c*.

3.9 Vectorization et alignement de mémoire

La vectorisation consiste à modifier l'exécution d'une seule opération sur un seul thread à plusieurs opérations qui s'effectuent de manière simultanée. Au début, les boucles avec des opérations d'addition ont été vectorisé avec la directive *pragma omp simd*, comme illustré dans listing (3.7), ensuite la mémoire contenant les données a été alignée en utilisant l'attribut dans listing (3.8).

Listing 3.7: Vectorization de boucles dans *lbm_phys.c*.

```
1  /* ******************* HEADERS ******************** */
2  #include <omp.h>
3  /* ******************* FUNCTION ******************** */
4  //Vectorizer 1 ere boucle
5  double get_vect_norme_2(const Vector vect1,const Vector vect2)
6  {
7   //vars
8   ...
9
10  #pragma omp simd reduction(+:res)//vectorize loop
11  //loop on dimensions
12  for ( k = 0 ; k < DIMENSIONS ; k++)
13   res += vect1[k] * vect2[k];
14  return res;
15 }
16
17 //Vectorizer 2eme boucle
18  for ( d = 0 ; d < DIMENSIONS ; d++)
19  {
20   ...
21
22   #pragma omp simd reduction(+:v[d])//vectorize loop
23   //sum all directions
24   for ( k = 0 ; k < DIRECTIONS ; k++)
25    v[d] += cell[k] * direction_matrix[k][d];
26   ...
27  }
```

Listing 3.8: Alignement de mémoire dans *lbm_struct.h*.

```
1 /* ********************* TYPEDEF ******************** */
2 typedef double __attribute__((aligned(16))) Vector[DIMENSIONS];
```

Les temps d'exécution de deux cas: (i) en utilisant la vectorisation et l'alignement mémoire, et (ii) en utilisant la flag de compilation (option d'optimisation) *-O3*, sont illustrés dans la figure (3.10).

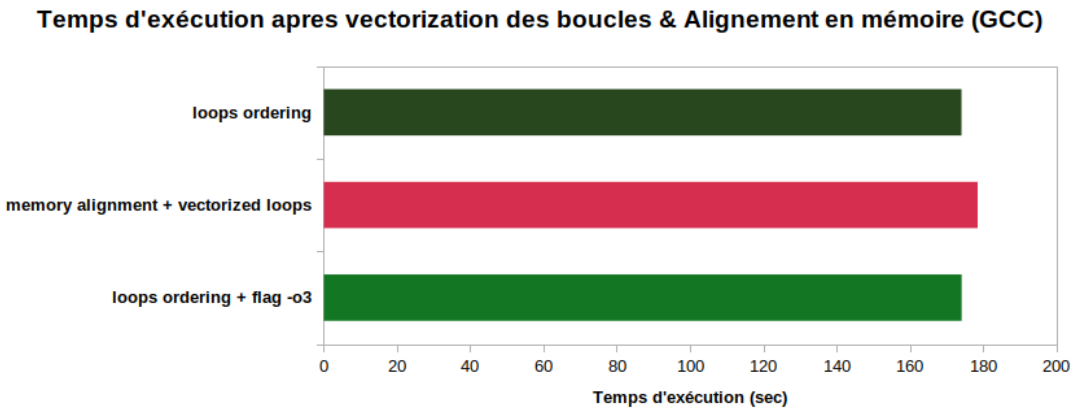

Figure 3.10: Temps d'exécution avant et après la vectorization des boucles et l'alignement mémoire.

En effet, la vectorisation et l'alignement mémoire ne semblent pas avoir un effet sur le code exécuté. Alors, qu'il y a une amélioration du temps d'exécution en supprimant les lignes ajoutées dans listing (3.7) dont le but est de vectoriser le code et d'aligner les données. Ceci s'explique par:

- Il faut activer au moins *-O2* dans le *Makefile* pour que *-fopenmp* fonctionne, et pour les performances en général (sinon c'est inutile),
- Les registres SIMD peuvent être utilisés sans OpenMP et en utilisant l'option *-O3* car, selon la documentation GCC, ils incluent l'indicateur *-ftree-vectorize*.

On se contente dans la suite d'utiliser la flag de compilation *-O3* dans le *Makefile* et de ne pas considérer les changement faites dans cette section.

3.10 OpenMP

Le programme contient initialement du *MPI* (mémoire distribuée). L'ajout de l'*OpenMP* (mémoire partagée) rend le code hybride. On peut donc travailler à la fois avec un système à

mémoire partagée et distribuée.

Pour se faire, on choisit d'ajouter *#pragma omp parallel* devant la boucle dans la routine *special_cells*, comme montré dans listing (3.9). Ce choix est justifié par la non-existence de dépendance inter-itération. De plus, on a des variables qui sont locales à la routine, et qu'on peut les inclure dans *private* de *#pragma omp parallel*.

Listing 3.9: Alignement de mémoire dans *lbm_struct.h*.

```
1  void special_cells(Mesh * mesh, lbm_mesh_type_t * mesh_type, const lbm_comm_t *
       mesh_comm)
2  {
3   ...
4
5   #pragma omp parallel for private (i,j)
6   //loop on all inner cells
7   for( i = 1 ; i < mesh->width - 1 ; i++ )
8   {
9    for( j = 1 ; j < mesh->height - 1 ; j++)
10   {
11    switch (*( lbm_cell_type_t_get_cell( mesh_type , i, j) ))
12    {
13     ...
14    }
15   }
16  }
17 }
```

Pour utiliser *OpenMp*, il est nécessaire d'ajouter *-fopenmp* dans le *Makefile*. L'exécution de ce programme pour un nombre de cores égal à 64 et un nombre de threads par core égal à 4 est possible à l'aide des commandes dans listing (3.10).

Listing 3.10: Alignement de mémoire dans *lbm_struct.h*.

```
1  $ make all
2  $ export OMP_NUM_THREADS=4
3  $ mpirun -np 64 ./lbm
```

Le temps d'exécution suite à l'ajout de l'*OpenMp* est illustré dans la figure (3.11).

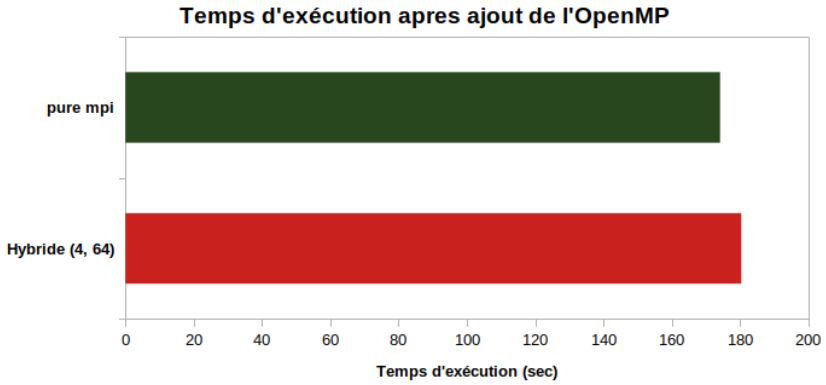

Figure 3.11: Temps d'exécution avant et après la vectorization des boucles et l'alignement mémoire.

Cette première tentative d'hybridation en ajoutant de l'*OpenMp* n'apporte pas d'amélioration comme le montre la figure (3.11). Dans la suite, on ne va pas considérer cette approche car on a perdu au niveau performance. Cependant, il est nécessaire de bien choisir les régions parallèles pour que l'hybridation apporte un gain.

3.11 Evaluation de scalabilité

On se propose dans cette partie de réévaluer la scalabilité forte du nouveau code afin de vérifier s'il est fortement scalable ou non (sans prendre en compte de l'effet de flags de compilation).

Pour se faire, on revient au dimensionnement du problème déjà évoqué dans la première étude de scalabilité (section 2.5.1). Soit une taille de domaine égale à 800×160 pour un nombre d'itérations égal à 1000. Les résultats de speed-up sont ainsi illustrés dans la figure (3.12).

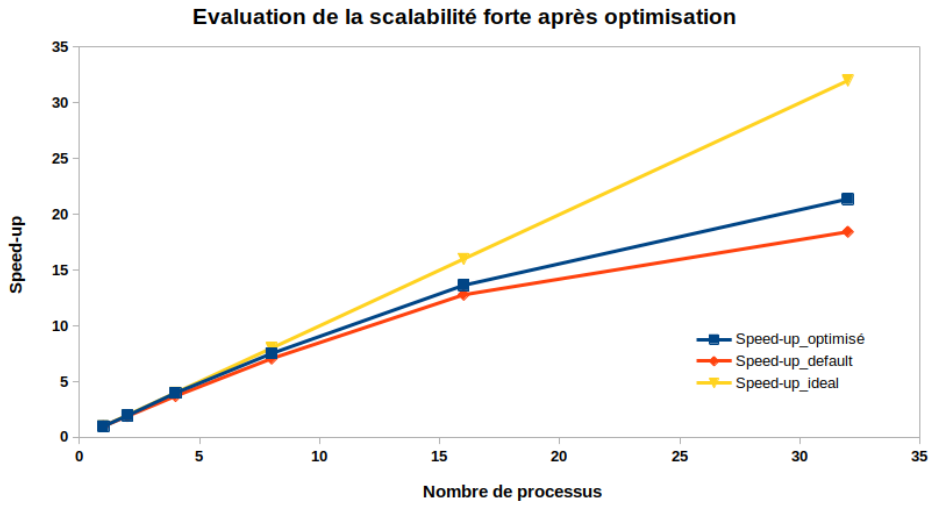

Figure 3.12: Evaluation de scalabilité après optimisation.

On en déduit ainsi que la scalabilité s'est améliorée par rapport a celle d'origine.

Chapter 4

Conclusion (Règles conduites)

Au cours de ce projet, on a pris en main un code de simulation numérique d'allée de tourbillons de von Karman par la technique LBM (Lattice Boltzmann Method). Le code d'origine a été volontairement bugué et désoptimisé. L'objectif du projet est d'abord de faire fonctionner le code, ensuite l'évaluer et l'optimiser afin de le rendre le plus scalable possible.

La première étape consistait à identifier et à corriger les bugs pour avoir une version fonctionnelle. Les bugs rencontrés étaient de natures différentes (problème d'allocation de mémoire et blocage de processus parallèle) et produisait des comportement différents (respectivement erreur de segmentation et attente infini). Afin de résoudre ces problèmes, on a utilisé des techniques de débogage variées (gdb et *fprintf*). Grâce à ces techniques, on a pu identifier et corriger rapidement des bugs sans avoir lu au préalable tout le code. De plus, des techniques différentes ont été utilisées en fonction du problème rencontré.

Après avoir un code valide, et avant de commencer l'optimisation, on a évalué les performances du code d'origine et analysé sa structure. Cette étape a permis de comprendre les détails d'implémentation et les faiblesses sur lesquelles on peut se concentrer dans la partie optimisation. Les échanges entre les sous-domaines reposent sur des communications bloquantes avec des synchronisations supplémentaires en utilisant des *MPI_Barrier*. L'analyse des communications a permis de repérer plusieurs points à améliorer: des échanges dupliqués et la présence excessive des communications à cause d'échanges de données non regroupées. On a pu aussi repérer la présence d'un temps d'attente d'une seconde (*sleep(1)*) à chaque itération de calcul, ce qui réduit fortement les performances du code. Les performances ont été ensuite évaluées sans le sleep.

L'étude de la scalabilité forte a montré qu'on obtient seulement environ 50% du speed-up idéal avec 32 processeurs. Les profilages avec Gprof et Perf montrent que les routines qui coûtent le plus en temps cpu se trouvent dans le fichier *lbm_phys.c*.

La partie optimisation commence par l'ajout des flags d'optimisation pour le compilateur. Cette modification a permis d'avoir un speed-up de 8 par rapport à la version initiale. L'échange des données par blocs a permis de gagner environ 6% du temps d'exécution. L'utilisation de communication non-bloquante à donner un gain supplémentaire d'environ 4%. L'influence du découpage du domaine sur la performance des communications a été aussi étudiée, et on a mis en évidence qu'un découpage adapté peut avoir de l'influence sur les performances. L'ordre des boucles utilisées pendant le calcul a aussi de l'influence non négligeable sur les performances.Des tentatives de vectorisation des boucles et leur parallélisation avec OpenMP ont été menées sans aboutir à des améliorations.

Les différentes modifications apportées au cours de ce projet ont permis d'avoir un speed-up de 15% par rapport à la version initiale avec le sleep(1) (508.63 *sec*), comme illustré dans la figure (4.1). On peut aussi constater une légère amélioration de la scalabilité forte. Il est encore possible de pousser l'optimisation plus loin avec une implémentation efficace de la parallélisation hybride *MPI/OpenMP*.

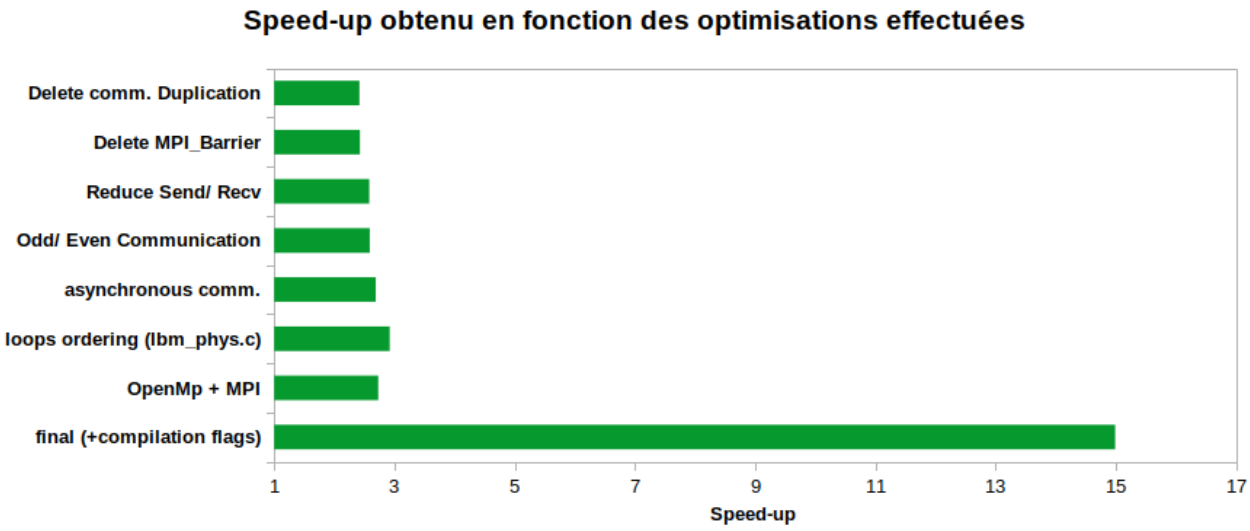

Figure 4.1: Speed-up obtenu en fonction des optimisations effectuées.

Ansi les règles tirées sont les suivantes:

- Il faut avoir une version fonctionnelle du code avant de commencer l'optimisation.
- Les technique de débogages permettent d'identifier et de résoudre les problèmes sans la lecture préalable du code.
- Des techniques de débogages différentes peuvent être utilisés en fonction du problème rencontré.
- L'analyse des communications, la compréhension de l'alignement des données dans la mémoire et le profilage du code sont des étapes nécessaires pour identifier les points à améliorer dans le code.
- L'ajout des flags d'optimisation pour le compilateur permet rapidement d'avoir un gain considérable.
- Il faut essayer de regrouper au maximum les données lors des échanges entre les processeurs.
- La méthode de découpage de domaine a une influence sur les performance des communications MPI.
- Réduire au maximum les communications bloquantes et les barrières MPI en faisant attention au comportement réel lors de l'exécution.
- Le gain de performance apporté par une modification peut dépendre du taille du domaine considéré, du nombre des processeurs, et aussi des caractéristiques de machine. Donc, il faut bien choisir le cas test pour l'évaluation de l'optimisation.

References

"GPROF Tutorial" (2012). URL: https://www.thegeekstuff.com/2012/08/gprof-tutorial/.

"Perf Wiki" (2020). URL: https://perf.wiki.kernel.org/index.php/Main_Page.

"Using the GNU Compiler Collection (GCC)" (2022). URL: https://gcc.gnu.org/onlinedocs/gcc/.

Appendix A

Description de la machine utilisée

Les analyses de performance ont été faites à l'aide du cluster *OB1-Exascale Computing Research cluster, knl01*, qui ont les caractéristiques montrées dans le tableau (A.1).

Nom du modele	Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
Nombre de processuers	64
Nombre de threads	256
Nombre de threads par core	4
Mémoire totale	96 GB

Table A.1: Caractéristiques de la machine utilisée.

Remarque: Le code associé à ce rapport est déposé dans le dépot git **TOP_LBM**. Le code SSH de ce dépot est le suivant:

git@github.com:Chaichas/TOP_LBM.git