

Optimisation d'un code de simulation : LBM

Auteurs :

M. Sébastien DUBOIS

Professeur :

Table des matières

1	Description de l'algorithme LBM	3
2	Description du matériel à disposition	4
2.1	Description matérielle	4
2.2	Description logicielle	4
3	Outils de mesure en HPC	6
3.1	Outils de debug	6
3.2	Outils de monitoring	6
3.3	Outils de benchmarking	7
3.4	Outils de profiling / prise de trace	7
3.4.1	MAQAO	8
3.4.2	CallGrind/Kcachegrind et PERF	8
3.4.3	TAU	8
3.4.4	Scalasca	8
4	Correction de bogue	9
4.1	Problème de compilation	9
4.2	Première exécution : erreur de segmentation mémoire	9
4.3	lenteur excessive du code	11
4.4	État des lieux et profiling	12
5	Optimisations MPI	13
5.1	Barrières	13
5.2	Nombre d'appels MPI	15
5.3	Partitionnement	16
5.4	Résultat final	17
5.5	Binding	18
6	Optimisation OpenMP	19
7	Optimisation séquentielle	20
7.1	Boucles imbriquées	20

	7.2	Vectorisation	21
8		Mesure de performances et Scalabilité	22
	8.1	Validation des codes	22
	8.2	Étude de scalabilite des codes	23
	8.3	Mesures sur le probleme initial	24
	8.4	Mesures sur un problème plus grand	27
	8.5	Mesures sur le cluster Ruche	30
9		Conclusion	32

1 Description de l'algorithme LBM

L'algorithme d'étude se découpe globalement en trois phases :

- une phase de calcul arithmétique, dans laquelle les résolutions des équations sont faites ;
- une phase d'envoi de données entre les domaines ;
- une phase optionnelle de sauvegarde des données de la simulation qui s'intercale à l'intérieure de la phase de calcul arithmétique.

Globalement, la complexité en envoi est de l'ordre de $\mathcal{C}^{\text{MPI}} = \mathcal{O}(n)$, avec n la taille caractéristique du problème 2D. La complexité en calcul est de l'ordre de $\mathcal{C}^{\text{MPI}} = \mathcal{O}(n^2)$. Ainsi, on peut déjà faire l'hypothèse suivante :

- lorsque le problème est de petite taille, la complexité n'a que peu d'importance puisque elle est définie à un facteur de multiplication près - et c'est ce multiplicateur qui conditionne le poids de chacune des deux parties. Le temps alloué aux communications peuvent rester majoritaires ;
- lorsque le problème devient très grand, la complexité gagne en importance dans l'estimation de la répartition du temps et vraisemblablement, les parties calculatoires deviendront majoritaires.

Le problème de base a été donné avec des dimensions 800×160 , ce qui reste un petit problème. Les communications MPI devraient jouer un rôle essentiel. Des essais de plus grande dimension seront opérés afin de vérifier ces précédentes hypothèses.

2 Description du matériel à disposition

Les aspects matériels et logiciels jouent un rôle essentiel en HPC. La définition du matériel passe par la description de la micro-architecture utilisée, du nombre de coeurs et - dans un contexte HPC - du nombre de noeuds ainsi que du réseau d'interconnexion entre ceux-ci. La définition du logiciel passe par la description du système d'exploitation utilisé - ou dans un contexte HPC, de la description des microkernels propres aux noeuds.

2.1 Description matérielle

Plusieurs systèmes ont été utilisés lors de l'exécution de ce projet. Notamment, des machines personnelles orientées *consumers*, telles que :

- Un ordinateur portable
- un ordinateur fixe
- une workstation

En tant qu'étudiant en classe de CHPS, nous avons aussi accès à deux serveurs nous permettant de faire des expérimentations HPC, avec :

- Des accès à des serveurs KNL/KNM (non interconnectés)
- des accès au cluster de calcul RUCHE dédié à l'Université Paris-Saclay

Voici brièvement un descriptif des matériels utilisés :

	laptop	desktop	workstation	Ruche
CPU	Intel 7700HQ	AMD Ryzen 9 3900X	AMD 3970x@4.0GHz	Intel Xeon Gold 6230
arch.	KabyLake	ZEN2	ZEN2	
AVX	AVX2	AVX2	AVX2	AVX512
NUMA	1	1	4 - L3 level	2 per MB
RAM	16Go@2400	16Go@3600	64Go@3600	6 channels
OS	Arch@5.17.4	Arch@5.17.4	Arch@5.17.4	CentOS7

2.2 Description logicielle

Le serveur RUCHE nous permet d'obtenir via SLURM de *jobs* de une heure sur 20 noeuds de calcul chacun compose de 20 coeurs. Pour des raisons d'allocation de taches, environ 8 noeuds seront utilisés en pratique afin d'avoir une allocation des ressources immédiates. Sur RUCHE, les programmes utiles - tels que GCC, OPENMPI, SCALASCA sont chargeables via la `commandmodule`. Cependant, un environnement `spack` a été installé sur toutes les machines personnelles et sur le cluster afin d'utiliser les mêmes programmes parfois indisponibles sur `module` de RUCHE, comme TAU, les divers implantations MPI tels que MPC et les compilateurs gcc récents. Les scripts développés utilisent des commandes `spack` et il peut être nécessaire pour l'utilisateur d'adapter les scripts en fonction de son environnement - notamment dans le cas de doublons d'installation. Voici des exemples de modules installes via `spack` :

- des compilateurs, tels que gcc@11.3.0, gcc-fortran ;
- des bibliothèques d'implémentation MPI tels que openmpi@4.1.3, mpich@4.0.2, intel-mpi, intel-oneapi-mpi, mpcframework@4.1.0 (finalement pas testée par faute de temps) ;

- des logiciels de prises de trace tels que scalasca, tau (scalep); valgrind, igprof, likwid, strace.

Le programme de compilation peut jouer un rôle essentiel au niveau des performances, pouvant résulter à un gain de performance conséquent. Tous les compilateurs sont dans leurs dernières versions stables, installés depuis les mêmes commandes SPACK. Bien-sur, cela ne veut pas dire que la configuration des compilateurs seront identiques sur les machines - rappelons que par exemple la bibliothèque MKL privilégient arbitrairement les architectures Intel par exemple.

3 Outils de mesure en HPC

3.1 Outils de debug

Plutôt que d'utiliser des affichages consoles avec des `printf`, il est largement plus commode - et obligatoire en HPC - d'utiliser des débogueurs pour résoudre des problèmes d'exécution. Les débogueurs permettent de d'obtenir notamment les informations suivantes :

- une explication brève de la raison du crash ;
- la ligne exécutée au moment du crash, grâce aux commandes de `backtrace`. En principe, on peut aussi obtenir la ligne d'exécution courante et de contrôler par la touche *entrée* la poursuite de l'exécution. Cela sera utile pour la suite ;
- les valeurs des variables au moment du bug - ou à un instant précise - permettant d'explorer l'état de la mémoire ;
- les informations peuvent être obtenues pour chaque thread. Il est en revanche plus fastidieux d'obtenir les informations multi-process.

Les débogueurs les plus populaires sont `gdb`, `lldb` mais il en existe d'autres comme `cgdb` - qui utilise `gdb`, et d'autres spécifiques au HPC multi-noeuds basées aussi sur le projet `gdb`. Il est à noter que nous avons eu un projet de classe de développement d'un débogueur et que nous possédons chacun notre propre débogueur séquentiel.

Remarque : Il est nécessaire d'utiliser des drapeaux de compilation comme `-g`, `-g3` permettant d'insérer dans le ELF les symboles de debug permettant d'introduire des informations relatives au code source.

3.2 Outils de monitoring

Le monitoring sur une machine permet de contrôler l'état de la machine selon principalement :

- la mémoire RAM utilisée et disponible ;
- l'utilisation du CPU actuelle par coeur ;
- l'espace disque disponible ;
- l'utilisation du réseau ;
- l'arbre des processeurs et threads exécutés au sein de la machine.

Cela permet de vérifier aisément de vérifier qu'un code s'exécute convenablement (*load balancing*, utilisation RAM à l'exécution), mais aussi de stopper aisément l'exécution d'un programme incontrôlable. Sur le cluster RUCHE et KNL, cela permet aussi de visualiser l'utilisation cpu opérée par d'autres utilisateurs, ce qui est susceptible d'altérer les performances de l'exécution. Les principaux outils sont :

- `top`, `htop` ;
- `bashtop`.

Lors du processeur d'expérimentation, les codes ont été exécutés à cote d'une visualisation `bashtop` afin de contrôler du bon paramétrage du nombre de noeuds et du nombre de threads notamment. En effet, certaines implémentations MPI empêchant de base d'exécuter plusieurs threads au sein d'un même rang.

3.3 Outils de benchmarking

L'exécution d'outils de benchmarking permet de valider par exemple la stabilité d'une machine (micro-benchmarks) et d'en comprendre ses performances. Il est facile de coder ses propres benchmarks bien qu'il en existe de très nombreux.

Les benchmarks d'INTEL - les IMB - sont des benchmarks MPI permettant de visualiser les performances des directives au sein d'un système physique. Son exécution permet à la fois de visualiser les performances propres à un système HPC, mais aussi à une implémentations MPI spécifique. Ces benchmarks ne correspondent généralement pas à un cas d'utilisation réelle, mais je m'en sers notamment pour mesurer la latence minimale d'une communication au sein d'un réseau, ainsi que du débit maximale entre les différents noeuds. Cela permet de stresser le réseau afin de vérifier qu'il n'y ait pas des problèmes de performances

J'ai développé quelques microbenchmarks me permettant de comprendre l'impact de différentes optimisations sur une machine particulières, notamment :

- des benchmarks MPI personnels me permettent de tracer une carte des débits maximaux et des latences entre les différents noeuds, permettant ainsi de capter une asymétrie propre à la topologie du réseau d'interconnexion ;
- des benchmarks de calcul séquentiel permettant de comparer les performances du compilateur avec des optimisations manuelles des codes. Notamment, le déroulage manuel des boucles ainsi que l'utilisation des *intrinsics* de codes AVX2/AVX512 ;
- des benchmarks d'instruction permettant de mesurer le throughput des instructions classiques en simulation, comme les additions, multiplications, fonctions cos, exp, $\sqrt{}$, ;
- des benchmarks de `rdtsc` permettant de comparer le compteur classique `rdtsc` avec un compteur amélioré `fenced rdtsc`, très utile pour des microbenchmarks.

*Note : Ces outils existent déjà, et dans des versions plus propres et professionnels. Cependant, le fait de les concevoir soi-même permet de maîtriser exactement les raisons d'un potentiel gain - notamment au sujet du déroulage de boucles et des *intrinsic*.*

3.4 Outils de profiling / prise de trace

Les outils de profilings sont nombreux : certains sont spécialisés sur un type de profiling particulier. Par exemple, le logiciel MAQAO privilégie l'étude de la capacité de vectorisation du code, alors que TAU favorise l'étude du parallélisme. Au cours de ce travail, de nombreux logiciels de profiling ont été utilisés - à des fins de découverte - mais aussi et surtout parce que chaque profiler a une caractéristique particulièrement intéressante. Voici la liste des profilers installés et testés au cours du projet :

- des profilers adaptés pour le séquentiel tels que MAQAO, callgrind/Kcachegrind, perf ;
- des profilers adaptés pour le parallélisme, tels que TAU, SCALASCA par l'intermédiaire de ScoreP, Pprof,
- des logiciels de timeline tels que Vampir et Jumpshot - Vampir étant limité à 8 rangs en version d'essai.

3.4.1 MAQAO

3.4.2 CallGrind/Kcachegrind et Perf

L'avantage de ce premier groupe de programmes (fournis avec VALGRIND) est de pouvoir visualiser le temps passe de chaque ligne du code source, avec une correspondance avec le code assembleur. Une représentation graphique permet de représenter le coût temporel de chaque fonction de façon général, ce qui facilite la démarche d'optimisation. Ce logiciel a été utilisé pour un profiling séquentiel en vue d'améliorer le code source, mais aussi en multithread/multinoeud afin de comprendre quels parties séquentielles réduisent la scalabilité.

Un code est profile avec Callgrind à l'aide du script `bash/valgrind.sh`.

PERF est aussi utilisé au cours de l'étude et propose un affichage console très intéressant avec le poids associé aux instructions en elles mêmes, utile pour, par exemple localiser des problèmes d'accès mémoire ou une mauvaise sélection des instructions par le compilateur.

3.4.3 TAU

le logiciel TAU permet par interposition de profiler et tracer les appels MPI afin par exemple d'en évaluer le coût ou de reconnaître des schémas de `MPI_Barrier` inutiles. Grâce à une suite de logiciels de conversion, les traces sont convertibles pour les logiciels JUMPSHOT et VAMPIR. Ces logiciels ont été utilisés principalement pour aider aux optimisations liées aux appels MPI et à reconnaître un très mauvais schéma de communication. Son utilisation se fait avec `bash/tau.sh`

3.4.4 Scalasca

La suite scalasca concurrente à TAU permet par instrumentation du code à la compilation de tracer les événements au cours du temps afin de récupérer des informations similaires qu'avec TAU. Les deux suites d'outils sont très vastes et il a été impossible de tout découvrir à temps. Notamment, j'ai trouvé la surcouche SCALASCA très lourde à l'exécution mais cela provient sûrement d'un mauvais paramétrage de l'instrumentation. Il doit être possible d'alléger la surcouche en précisant un nombre restreint d'options à activer. Scalasca permet aussi de visualiser des traces sur timeline. Les traces peuvent être lancées avec `bash/scalasca.sh`

4 Correction de bogue

4.1 Problème de compilation

Le projet initial comporte des problèmes dès la compilation. Il peut arriver parfois que le projet récupéré ne compile pas directement lorsque la configuration utilisée n'est pas exactement la même que celle du développeur insouciant - du fait des versions de compilateurs par exemple. Il est alors parfois nécessaire de se plonger dans le code afin de résoudre les problèmes initiaux. Ici en l'occurrence, la compilation est empêchée du fait d'un signal de double définition au sein du fichier `lbm_phys.h` liées aux entités `opposite_of`, `equil_weight` et `direction_matrix`. La commande `find . -exec grep "opposite_of" '' -print` permet de remarquer effectivement que les entités sont redéfinies ailleurs. Il est alors nécessaire d'ajouter le mot clé `extern` avant la définition de chacun de ces entités dans le fichier *header* correspondant. La compilation est possible une fois ces mots clés ajoutés.

4.2 Première exécution : erreur de segmentation mémoire

La première exécution en séquentielle (`./lbm`) ou multi-noeuds (`mpirun -np 2 ./lbm`) aboutit à une erreur de segmentation mémoire indiquée par le message d'erreur [2] 636078 `segmentation fault (core dumped) ./lbm`. Une erreur de segmentation mémoire est lorsque le programme demande un accès à une adresse mémoire pour laquelle il n'a pas les droits. Le noyau du système d'exploitation stoppe alors le programme afin d'assurer l'intégrité du système. La stratégie de débogage commence par l'exécution du programme au sein d'un débogueur en séquentiel afin de déterminer la raison du signal d'erreur. J'ai personnellement utilisé `cgdb` pour son interface améliorée par rapport à `gdb`. Voici la démarche suivie aillant permis la résolution de ce bogue :

- premièrement, l'exécution du programme au sein du débogueur par la commande `cgdb ./lbm`;
- ensuite, le lancement de l'exécution par la commande `run` au sein de l'interface ;
- le programme est exécuté jusqu'au signal d'erreur en montrant la ligne exécutée au moment du signal. Il est à noter que cette ligne ne peut être obtenue par le débogueur que si le programme a été compilé avec les symboles de débogage comme `-g`. Afin de permettre la visualisation de l'embriquement des appels de fonctions au moment du bogue, il est possible d'utiliser la commande `backtrace`.

Voici les résultats obtenus au sein du débogueur :

```
1 81|     for ( k = 0 ; k < DIRECTIONS ; k++)
2 82|     {
3 83|         //compute equilibr.
4 84|         v[0] = helper_compute_poiseuille(j + mesh_comm->y,MESH_HEIGH...
5 85|-----> Mesh_get_cell(mesh, i, j)[k] = compute_equilibrium_profile(v,...
6 86|         //mark as standard fluid
7 87|         *( lbm_cell_type_t_get_cell( mesh_type , i, j) ) = CELL_FUILD..
8 88|         //this is a try to init the fluide with null speed except on ...
9 89|         //if (i > 1)
10 90|         //      Mesh_get_cell(mesh, i, j)[k] = equil_weight[k];
```

```
11 91|    }
```

Listing 1 – ligne du bogue

```
1 Thread 1 "lbm" received signal SIGSEGV, Segmentation fault.
2 0x000055555555569f7 in setup_init_state_global_poiseuille_profile
3 (mesh=0x7fffffff6a40, mesh_type=0x7fffffff6a70, mesh_comm=0
4 x7fffffff6a80) at lbm_init.c:85
5 85|     Mesh_get_cell(mesh, i, j)[k] = compute_equilibrium_profile(v,density,k
6 );
7 (gdb) bt
8 #0 0x000055555555569f7 in setup_init_state_global_poiseuille_profile
9 (mesh=0x7fffffff6a40, mesh_type=0x7fffffff6a70,
10 mesh_comm=0x7fffffff6a80) at lbm_init.c:85
11 #1 0x00005555555556cce in setup_init_state
12 (mesh=0x7fffffff6a40, mesh_type=0x7fffffff6a70, mesh_comm=0x7fffffff6a80)
13 at lbm_init.c:155
14 #2 0x0000555555555584d in main
15 (argc=1, argv=0x7fffffff6c78) at main.c:159
```

Listing 2 – backtrace

Il est possible de récupérer les valeurs des variables dans le contexte, comme par exemple les valeurs de `i`, `j`, `k`. Pour essayer de déterminer le bogue, nous allons passer en revue les valeurs accédées au moment du bogue :

```
1 (gdb) print i
2 $18 = 0
3 (gdb) print j
4 $19 = 0
5 (gdb) print k
6 $20 = 0
7 (gdb) print v
8 $21 = {-0.0025315454293738382, 0}
9 (gdb) print density
10 $22 = 1
11 (gdb) print k
12 $23 = 0
13 (gdb) print mesh
14 $24 = (Mesh *) 0x7fffffff6a40
15 (gdb) print mesh->cells
16 $25 = (lbm_mesh_cell_t) 0x0
```

Listing 3 – valeurs

Cela permet de mettre en évidence le fait que le pointeur `mesh->cells` ne pointe vers aucune adresse valide. En effet, en explorant dans le code, on remarque que la ligne d'allocation mémoire de `mesh->cells` a été mise en commentaire. Il suffit simplement de retirer le commentaire afin de résoudre le bogue :

```
1 mesh->cells = NULL;
2 mesh->cells = malloc(width * height * DIRECTIONS * sizeof(double));
```

Listing 4 – correctif d'allocation

et le code fonctionne sans problème de segmentation mémoire.

4.3 lenteur excessive du code

On se rend compte assez rapidement que, quel que soit la taille du problème, du nombre de rangs MPI, une itération prend environ une seconde. Il est alors nécessaire de profiler le code à l'exécution afin de déterminer dans quelle fonction le programme passe le plus de temps.

Il est alors venu à l'idée d'utiliser des profilers tels que **perf**, **gprof**, **callgrind**/**Kcachegrind** mais ils ne permettent pas d'introduire les temps des appels systèmes au sein des rapports à ma connaissance. Ainsi, il a été impossible de déterminer la raison de la lenteur. Une possibilité est alors d'utiliser un tracer. On peut utiliser - mais cette démarche est peu propre - un débogueur comme **cgdb** permettant d'afficher la ligne de code exécutée au moment d'une sortie d'exécution par **Ctrl+C**. Voici le résultat obtenu :

```
1 #0 0x00007ffff7a69a55 in clock_nanosleep@GLIBC_2.2.5 () from /usr/lib/libc.so.6
2 #1 0x00007ffff7a6e717 in nanosleep () from /usr/lib/libc.so.6
3 #2 0x00007ffff7a6e64e in sleep () from /usr/lib/libc.so.6
4 #3 0x00005555555557b32 in lbm_comm_ghost_exchange
5 mesh=0x7fffffff3ab0, mesh_to_process=0x7fffffff3a80) at lbm_comm.c:319
6 #4 0x0000555555555962 in main
7 (argc=1, argv=0x7fffffff3d28) at main.c:189
```

Listing 5 – présence d'un temps de pause dans le code - **cgdb**

Aussi, une exécution au sein de **strace** - un tracer - affiche :

```
1 clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7fff13c45770) = 0
```

Listing 6 – appel système par **strace**

Le coupable a été trouvé : il existe au sein du programme un appel système à **clock_nanosleep** d'une durée d'une seconde, présent d'après **cgdb** à la ligne 319 dans **lbm_comm.c**. La commande associée à cette ligne est **FLUSH_INOUT()** ; ce qui est à première vue très étrange. Une simple recherche **find . -exec grep "FLUSH_INOUT" -print** permet de remarquer la présence de **define __FLUSH_INOUT__ concat(s,l,e,e,p)(1)** au sein de **lbm_config.h**. Sympa !

Une fois retiré - malgré le commentaire - le code est bien plus rapide et la checksum inchangée. Ce temps d'attente imposé est absolument inutile puisque le code de base utilise des appels MPI bloquants. Lorsque des appels non-bloquants seront utilisés, il est largement plus logique d'utiliser des appels **MPI_Wait**.

Nous avons dorénavant un code fonctionnel, et relativement rapide. Il reste maintenant à l'optimiser selon plusieurs axes :

- du point de vue MPI, en améliorant la façon dont est découpé le problème, et la façon dont les appels sont opérés entre les noeuds ;
- du point de vue du parallélisme inter-noeuds, en ajoutant un paradigme hybride MPI/openMP ;
- du point de vue séquentielle, en améliorant l'exécution des fonctions élémentaires et des kernels de calcul. Cela passe notamment par l'amélioration des accès mémoires ;
- du point de vue de la vectorisation. Les processeurs de tests sont munis d'instructions AVX2 ou AVX512 permettant sur des flottants double précision d'opérer 4 ou 8 opérations simultanés. Le gain potentiel peut être conséquent.

La démarche d'amélioration sera détaillée dans les parties ci-dessous selon l'ordre dans lesquelles je les aies implémentées. Il est possible de présenter les résultats une fois les optimisations faites mais ce n'est pas la démarche qui sera prise. Grâce à la compilation conditionnelle, les optimisations seront *a posteriori* désactivées afin de comparer la non présence d'une optimisation par rapport au code le plus optimise. Cela permet de comparer plus rigoureusement l'effet de chacune des optimisations en utilisant une base identique.

4.4 État des lieux et profiling

Afin d'optimiser le code, des outils de *profiling* sont utilisés afin de comprendre les points chauds du code. Il est par exemple préférable d'optimiser un morceau de code qui prend 90 du temps d'exécution plutôt qu'une partie qui en prend 1. Des outils permettent de déterminer le temps alloué à chaque fonction - et même à chaque ligne - ce qui permet de renforcer la démarche d'optimisation. Aussi, des outils propres au MPI permettent de visualiser les appels MPI sur une timeline et de comprendre les limites du code. La démarche d'optimisation a consisté à jongler entre ces deux approches de *profiling*. Il est vite apparu que la première limite est liée aux appels MPI, ainsi, les optimisations MPI sont présentées en premier lieu.

5 Optimisations MPI

La décomposition de domaine consiste au découpage d'un domaine rectangulaire en sous-domaines rectangulaires de taille égale. Des noeuds fantômes (*ghost nodes*) permettent de faire la liaison d'information entre les différents domaines. Des appels MPI sont nécessaires pour propager l'information d'un domaine a l'autre.

La première étape fut de tester la scalabilité du code. Voici une représentation de la scalabilité MPI du code sur les différents systèmes disponibles :

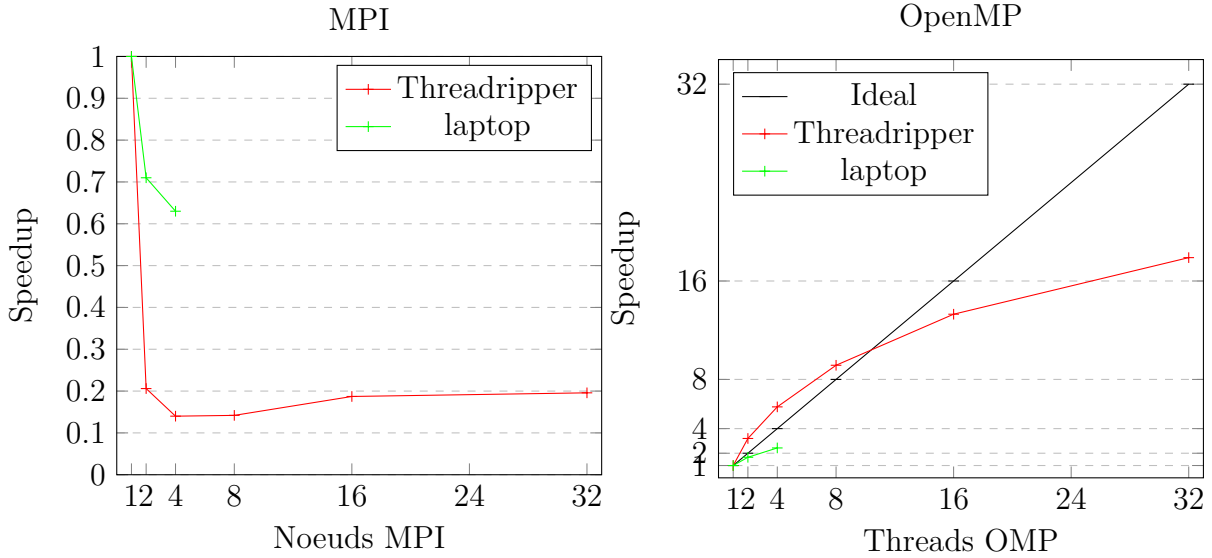


FIGURE 1 – Scalabilité forte en MPI et OpenMP avec le code de base sur un problème de configuration 800×160 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

Il est remarquable que l'exécution sur plusieurs noeuds est plus lent que sur un seul. Le code de base souffre d'un problème de scalabilité conséquente et il est primordial de résoudre ce problème en premier lieu.

La première hypothèse faite est que les envois MPI sont mal opérés. Il faut alors scrupuleusement lire le code associé aux échanges MPI `lvn_comm.c`. Afin de comprendre le problème facilement, il a été utilisé le logiciel de profiling MPI `tau` qui permet de récupérer les traces des appels MPI. Il sera ensuite possible de visualiser la *timeline* par `Vampir` ou `Jumpshot`.

Note : Il existe un programme concurrent appelé SCALASCA qui sera aussi utilisé) titre de comparaison.

5.1 Barrières

Une première phase de trace a été effectuée avec TAU et visualisée sur JUMPSHOT et VAMPIR. Voici les résultats obtenus :

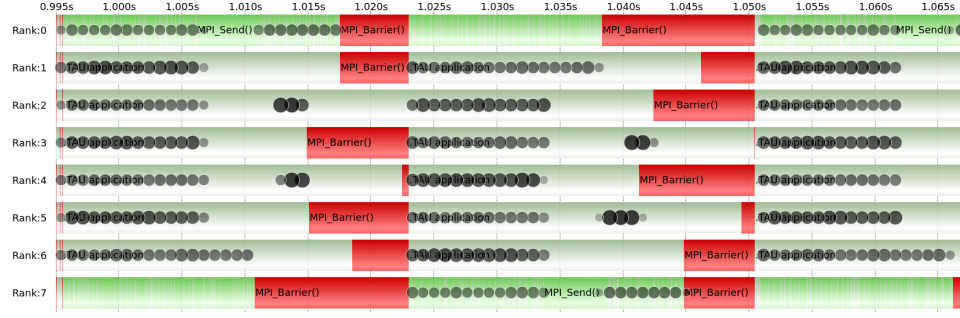


FIGURE 2 – Schéma de communication MPI entre 8 noeuds dans un contexte MPI pour quelques itérations. Les points blancs représentent des groupes d’envois rassemblés pour simplifier la visualisation. Les Barres rouges correspondent aux latences passées dans les barrières MPI. Aucun calcul n’est fait dans ces zones rouges.

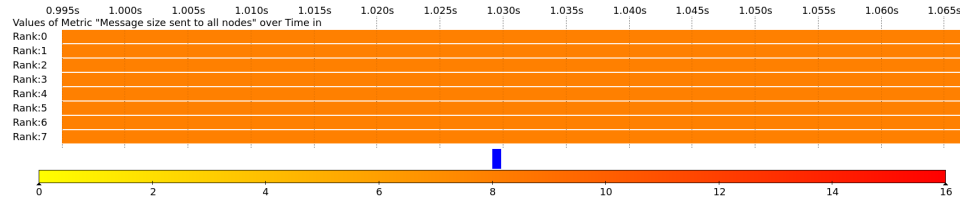


FIGURE 3 – Représentation graphique de la taille des messages envoyés pour le même problème et le même intervalle de temps. On remarque que la conception initiale du code effectue des envois de données de 64 bytes pour chacune des communications.

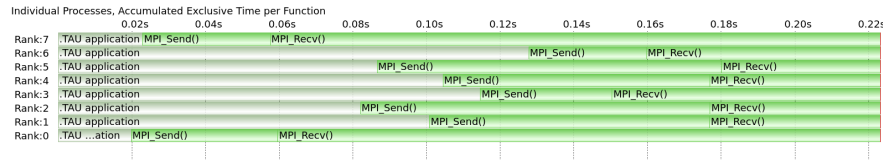


FIGURE 4 – Représentation par rang du temps total accumulé alloué à chaque transaction MPI. Les rangs 0 et 7 passent significativement plus de temps dans les phases de réception.

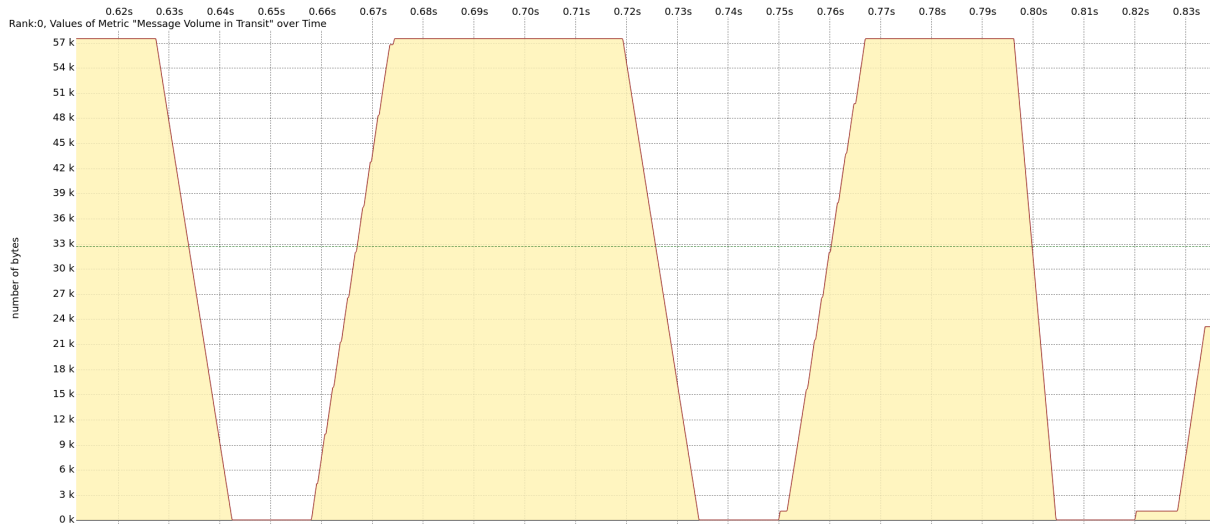


FIGURE 5 – Représentation du débit moyen des transactions sur quelques itérations.

Il y a dans le code original des barrières `MPI_Barrier` après chaque phase d'envoi. les barrières bloquantes permettent sur le principe à bloquer chaque rang tant qu'il y a au moins un rang qui n'a pas atteint la barrière. Les barrières sont placées entre des phases d'envoi et de réception opérés par des appels MPI bloquants. Ainsi, les barrières n'ont aucun intérêt dans le cas présent. Ainsi, il est propose de supprimer les barrières avec le drapeau de compilation `CCBARRIER="-DNOBARRIER"`. Les barrières liées à la gestion de fichier sont gardées.

Le gain reste relativement faible et un goulot d'étranglement plus important se trouve ailleurs.

Note : On remarque aussi la présence de parties de codes inutiles concernant `lbm_comm_sync_ghosts_hor` dont certains appels sont fais deux fois. Ces parties de code sont désactivées par le flag de compilation `-DNOUSELESS`

5.2 Nombre d'appels MPI

L'outil de timeline d'appels MPI permet d'observer qu'il y a de très nombreux appels MPI pour chaque itération. Voici une représentation du nombre d'appels pour chaque noeuds pour une itération seulement :

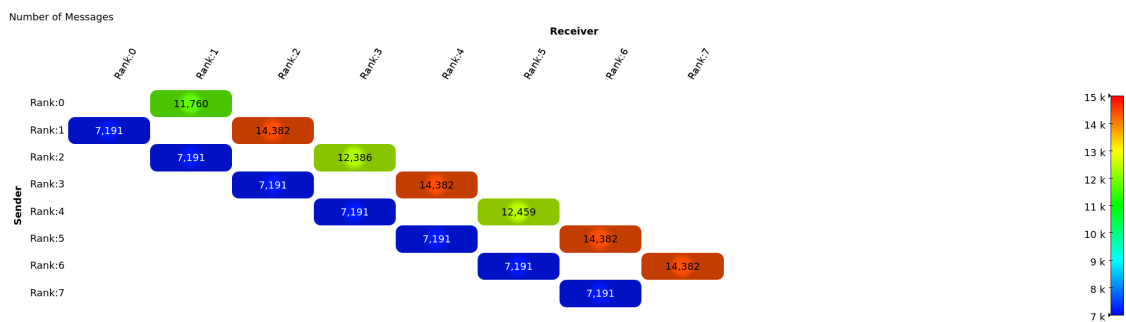


FIGURE 6 – Représentation matricielle du nombre de messages reçus et envoyés pour chaque couple *sender/reciever* sur la même plage temporelle que les figures précédentes. De très nombreux messages sont envoyés pour chaque itération - de l'ordre du millier.

Les messages envoyés sont des scalaires de taille 8 bytes (flottant double précision). Or, des benchmarks peuvent montrer que ce type d'envoi souffre de problèmes de débits car le coût de les surcouches *software* et *hardware* permettant la communication entre les noeuds deviennent prédominantes. Voici les résultats de débit et de latence en fonction de la taille d'envoi que j'avais mis au point avant le début du projet :

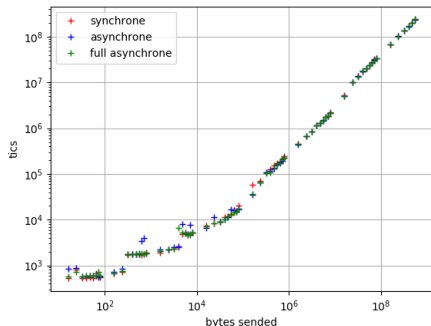


FIGURE 7 – bandwidth

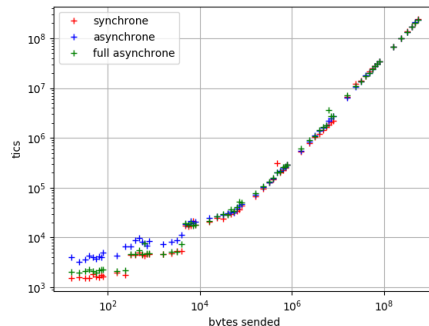


FIGURE 8 – latency

FIGURE 9 – Débit et latence d'envoi entre deux rangs MPI (Architecture ZEN2)

Une solution simple est de factoriser les appels en arrangeant les données de sorte à faire le minimum d'envois. Dans le cas d'un partitionnement vertical cela passe par une simple modification de la taille d'envoi puisque les données dans le tableau sont continues en mémoire. Cependant, dans le cas de base - le cas d'un partitionnement horizontal - les données ne sont pas continues en mémoire. Il est possible de stocker temporairement les données dans un tableau *buffer* afin de pouvoir factoriser les envois en un seul.

On note que cette solution permet d'énormes gains, et le code est enfin relativement scalable. Le temps alloué aux envois MPI deviennent minoritaires par rapport au coût associé au calcul arithmétique. Cependant, il est encore possible d'optimiser les envois.

5.3 Partitionnement

Dans le cas du code de base, un calcul de *pgcd* favorise un partitionnement à découpage horizontal. Puisque la configuration de la simulation est un rectangle dans la longueur x , ce découpage aboutit à des tailles de messages importants. Il est possible de revoir le partitionneur en favorisant alors un découpage vertical induisant des tailles de transferts réduits. En plus, le *buffer* de factorisation devient inutile ce qui augmente d'autant la rapidité du transfert. Encore mieux, il est possible de spécifier explicitement les nombres de blocs dans chaque direction afin d'améliorer la scalabilité faible selon chacune des directions.

*Note : Il est possible de spécifier le sens de partitionnement par le drapeau **CCSPLIT** à **-DVERTICAL** ou **-DHORIZONTAL**.*

5.4 Résultat final

A l'aide d'une trace TAU accompagné de PPROF, on peut aisément récupérer des statistiques concernant la partie MPI. Voici son résultat pour le code de base ainsi que pour le code optimisé :

```

1
2 FUNCTION SUMMARY (total):
3
4 %Time      Exclusive      Inclusive      #Call      #Subrs      Inclusive      Name
5           msec          total msec          #Call      #Subrs      usec / call
6
7 100.0      58,588      1:28.723      8      1.62087E+06      11090474      .TAU application
8 29.6      26,230      26,230      19120      0      1372      MPI_Barrier()
9 2.7      2,355      2,355      800008      0      3      MPI_Send() [THROTTLED]
10 1.5      1,364      1,364      800008      0      2      MPI_Recv() [THROTTLED]
11 0.2      158      158      8      0      19807      MPI_Init()
12 0.0      24      25      8      16      3165      MPI_Finalize()
13 0.0      0.937      0.937      1656      0      1      MPI_Comm_rank()
14 0.0      0.936      0.936      16      0      58      pthread_join
15 0.0      0.229      0.229      24      0      10      MPI_Reduce()
16 0.0      0.047      0.047      40      0      1      MPI_Comm_size()

```

Listing 7 – PPROF/TAU - code MPI de base - 200 itérations

```

1
2 FUNCTION SUMMARY (total):
3
4 %Time      Exclusive      Inclusive      #Call      #Subrs      Inclusive      Name
5           msec          total msec          #Call      #Subrs      usec / call
6
7 100.0      1,190      1,933      8      7380      241737      .TAU application
8 15.1      291      292      8      16      36524      MPI_Finalize()
9 9.8      189      189      2814      0      67      MPI_Send()
10 8.2      159      159      8      0      19906      MPI_Init()
11 3.9      74      74      16      0      4657      MPI_Barrier()
12 1.1      21      21      2814      0      8      MPI_Recv()
13 0.3      5      5      1656      0      3      MPI_Comm_rank()
14 0.1      1      1      16      0      72      pthread_join
15 0.0      0.348      0.348      24      0      14      MPI_Reduce()
16 0.0      0.097      0.097      40      0      2      MPI_Comm_size()

```

Listing 8 – PPROF/TAU - code MPI optimisé - 200 itérations

Voici les remarques principales :

- On a pu très fortement réduire le nombre d'appels MPI, notamment le `MPI_Send`, `MPI_Recv`, `MPI_Barrier`. Notons qu'il reste des barrières liées à la gestion de fichiers, dont la partie pourrait aussi s'optimiser. ;
- Sur le code de base, il est clair que la majorité du temps est alloué aux appels MPI. Sur le code optimisé, la part devient beaucoup plus faible, avec des temps alloués à la surcouche MPI extrêmement réduite ;
- Notons que, du fait de la lourdeur du logiciel de trace - liée à un très grand nombre d'appels MPI suivis - les mesures sont altérées par des phases de `flush` et les résultats ne sont à regarder qu'à titre indicatif.

La phase d'optimisation est donc un succès. Il reste cependant un axe d'amélioration : utiliser un partitionneur qui décompose les domaines en carré. En effet, cela permet d'être plus robuste à d'autres types de configurations - au hasard lorsque le domaine d'étude est très haut sur la direction y ... Aussi, il serait nécessaire de travailler sur le recouvrement calcul/communication en utilisant des appels non-bloquants.

Les résultats précédents utilisent tous les deux les mêmes optimisations séquentielles et sont exécutées sur 8 rangs avec un seul thread OPENMP.

5.5 Binding

Le *Binding* consiste à arranger les processus ou les threads à des coeurs particuliers au sein d'un processeur voire d'un cluster - notamment lorsque les performances des communications sont asymétriques. En effet, la hiérarchie mémoire et les réseaux d'interconnexions ont tendance à adopter des topologies hétérogènes afin d'améliorer les accès mémoires au déficit d'accès mémoires topologiquement éloignés. Par exemple, les processeurs d'architecture Zen 2 possèdent des caches L3 pour chaque groupe de CCD et non pas un cache unifié. Cela permet d'améliorer significativement les performances lorsque la localité mémoire du programme est bien gérée - mais peut potentiellement les réduire dans le cas contraire. Aussi, dans un cluster, les noeuds sont rarement tous connectés entre eux et des latences supplémentaires peuvent subvenir.

Dans le cadre du cpu, il existe des programmes de *binding* comme `taskset` ou `numactl`. Des expériences ont été faites avec par exemple `-bind-by core/machine/l3cache` ou une alternative `-map-to`. Il faut adapter le *binding* afin d'améliorer la bande passante. Par exemple, dans le cas d'un programme MPI non-hybride, il est préférable d'allouer les processus sur le même cache L3 afin d'accélérer les accès. Cela dit, le binding est plus délicat que cela à manier : dans le cas explicite précédemment, il peut mener à un manque de cache mémoire : des gros problèmes préféreraient alors un partage des processus sur chacun des caches L3.

Voici un rapide comparatif de performances entre différents bindings sur la version la plus optimisée :

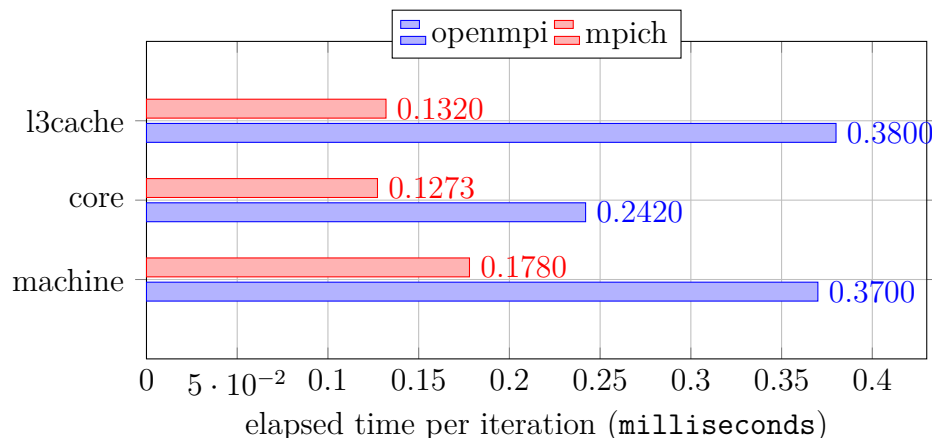


FIGURE 10 – Effet du binding MPI sur machine THREADRIPPER en fonction de l'implémentation MPI (`-np 32, OMP_NUM_THREADS=1`)

Cela nous permet de comprendre qu'il faut utiliser les bons arguments lors de l'appel `mpirun` afin d'exploiter au mieux les performances de l'implémentation. Au sujet de la différence de performance entre les deux implémentations, elle peut provenir :

- De flags de compilations différents au moment de la compilation de chacune des implémentations. En désactivant des fonctionnalités à la compilation sur chacune des implémentations, on peut les rendre plus ou moins efficaces ;
- des paramètres fournis à `mpirun`. EN effet, les deux implémentations n'utilisent pas les mêmes paramètres par défaut. Il faudrait tous les préciser pour assurer une comparaison juste.

6 Optimisation OpenMP

OPENMP est une plateforme multi-threading permettant de paralléliser facilement un code selon un paradigme de mémoire partagée. La librairie convertit des morceaux de code par des threads `pthread`s. Cette librairie est particulièrement adaptée pour le calcul scientifique en ce qu'elle permet d'approcher les performances maximales avec un faible temps de développement et de débogage. Il est souvent inutile d'utiliser des `pthread`s pour certains codes.

La démarche de parallélisation suit le raisonnement :

- Sans processus MPI, profiler le code de sorte à déterminer les fonctions critiques. Les boucles imbriquées sont souvent les entités qui nécessitent le plus de calcul ;
- ajouter des pragmas OpenMP au niveau des zones critique parallélisables - les boucles - afin d'accélérer significativement l'exécution du code ;
- étudier les performances en fonction du nombre de threads appliqués par `export OMP_NUM_THREADS=<N>`

Le code présente deux boucles facilement parallélisables au sein de `lbm_phys.c` dans les fonctions de résolution des équations physiques. Il s'agit de boucles imbriquées qui effectuent chacune les mêmes tâches sans branchement. Ainsi, les paramètres de configuration les plus optimaux pour ces boucles sont `pragma omp for schedule(static)`, `static` signifiant que la quantité de travail allouée à chaque thread est déterminé à la compilation. Il existe des alternatives comme `dynamic` et `guided` mais qui alourdissent la gestion des thread alors que la répartition du travail entre les threads est relativement efficace. Aussi, afin d'assurer un découpage du travail efficace, il est nécessaire d'appliquer le parallélisme au niveau de la boucle la plus externe plutôt qu'interne afin d'améliorer la granularité. D'ailleurs, cela montre un certain avantage lorsque les deux boucles sont inversées pour améliorer les accès mémoires.

A posteriori, je pense que `schedule(static)` peut poser des problèmes de répartition sur des problèmes de très grande taille.

Des résultats de performance sont disponibles dans la partie **scalabilité**.

7 Optimisation séquentielle

Traditionnellement, l'optimisation séquentielle se fait **avant** l'optimisation parallèle. Cependant, le code est offert avec un paradigme de parallélisation déjà existant, ce qui remet en question le processus traditionnel d'optimisation.

L'optimisation séquentielle consiste en l'optimisation des calculs effectués sans tenir compte du multiprocessing ou du multithreading. Par exemple, il s'agit de :

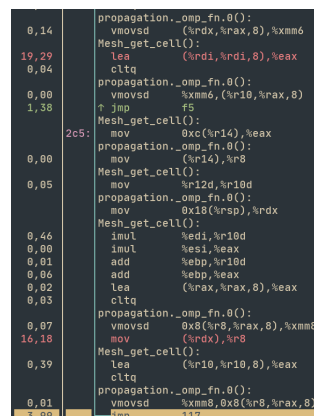
des méthodes de déroulage de boucle ; améliorer les accès mémoires, notamment dans des boucles imbriquées ; jouer sur les directives de compilation telles que `-Ofast`, `-mavx2`, `-funroll-loops` ; vectoriser proprement le code le cas échéant.

Encore une fois, les outils de *profiling* sont utiles afin de repérer les parties nécessitant une amélioration :

PERF et KCACHEGRIND permettent de connaître les zones critiques du code ; MAQAO permet en particulier d'améliorer la vectorisation du code.

7.1 Boucles imbriquées

On reconnaît grâce à PERF la présence d'une boucle coûteuse en temps :



0,14	propagation._omp_fn.0():
	vmovsd (%rdx,%rax,8),%xmm6
	Mesh_get_cell():
19,29	lea (%rdi,%rdi,8),%eax
0,04	cltq
	propagation._omp_fn.0():
0,00	vmovsd %xmm6,(%r10,%rax,8)
1,38	jmp %r5
	Mesh_get_cell():
	mov 0xc(%r14),%eax
	propagation._omp_fn.0():
0,00	mov (%r14),%r8
	Mesh_get_cell():
0,05	mov %r12d,%r10d
	propagation._omp_fn.0():
	mov 0x18(%rsp),%rdx
	Mesh_get_cell():
0,46	imul %edi,%r10d
0,00	imul %esi,%eax
0,01	add %ebp,%r10d
0,06	add %ebp,%eax
0,02	lea (%rax,%rax,8),%eax
0,03	cltq
0,07	propagation._omp_fn.0():
	vmovsd 0x8(%r8,%rax,8),%xmm8
16,18	mov (%rdx),%r8
	Mesh_get_cell():
0,39	lea (%r10,%r10,8),%eax
	cltq
	propagation._omp_fn.0():
0,01	vmovsd %xmm8,0x8(%r8,%rax,8)
3,99	jmp 117

FIGURE 11 – Rapport PERF sur une boucle du code

L'accès `Meshh_get_cells` concentre la plupart du temps de l'exécution de la fonction. On remarque que les données envoyées dans la fonction `Mesh_get_cells` ne sont pas continues en mémoire à cause des ordres d'indexation. On propose alors d'inverser l'ordre des boucles - tout simplement - afin de profiter de la même *cache-line* sur plusieurs itérations. Voici les différences de rapport - seul le flag `DOPTIMIZEDLOOP` a été ajouté :

```

0.00 #if STANDARDLOOP
0.00 #pragma omp parallel for schedule(static)
0.00 for (int j = 0; j < mesh_out->height; j++) {
0.00     19 call(s) to 'omp_get_thread_num' (libgomp.so.1.0.0; parallel.c)
0.00     19 call(s) to 'omp_get_num_threads' (libgomp.so.1.0.0; parallel.c)
1.05     for (int i = 0; i < mesh_out->width; i++) {
        // for all direction
        // for all direction
        for (int k = 0; k < DIRECTIONS; k++) {
            // compute destination point
            ii = (i + direction_matrix[k][0]);
            jj = (j + direction_matrix[k][1]);
            // propagate to neighbor nodes
            if ((ii >= 0 && ii < mesh_out->width) &&
                (jj >= 0 && jj < mesh_out->height)) {
                Jump 22 107 184 of 22 186 376 times to lbm_struct.h:116 [propagation_omp_fn.0]
                Jump 3 078 of 2 468 556 times to lbm_phys.c:357
14.14         Mesh_get_cell(mesh_out, ii, jj) = Mesh_get_cell(mesh_in, i, j[k]);
                Jump 2 450 259 times to lbm_phys.c:357
                Jump 4 921 874 times to lbm_phys.c:361
                Jump 14 735 051 times to lbm_phys.c:365
            }
        }
    }
}

```

```

0.00 #elif OPTIMIZEDLOOP
0.00 #pragma omp parallel for schedule(static)
0.00     19 call(s) to 'omp_get_thread_num' (libgomp.so.1.0.0; parallel.c)
0.00     19 call(s) to 'omp_get_num_threads' (libgomp.so.1.0.0; parallel.c)
0.69     for (int i = 0; i < mesh_out->width; i++) {
        for (int j = 0; j < mesh_out->height; j++) {
            Jump 2 453 318 of 2 468 556 times to lbm_phys.c:381
            // for all direction
            for (int k = 0; k < DIRECTIONS; k++) {
                // compute destination point
                ii = (i + direction_matrix[k][0]);
                jj = (j + direction_matrix[k][1]);
                // propagate to neighbor nodes
                if ((ii >= 0 && ii < mesh_out->width) &&
                    (jj >= 0 && jj < mesh_out->height)) {
                    Jump 22 107 184 of 22 189 454 times to lbm_struct.h:116 [propagation_omp_fn.0]
                    Jump 3 078 of 2 468 556 times to lbm_phys.c:374
                    Jump 3 078 of 2 468 556 times to lbm_phys.c:379
10.22         Mesh_get_cell(mesh_out, ii, jj) = Mesh_get_cell(mesh_in, i, j[k]);
                    Jump 2 450 259 times to lbm_phys.c:374
                    Jump 4 930 956 times to lbm_phys.c:379
                    Jump 4 921 874 times to lbm_phys.c:381
                    Jump 9 804 095 times to lbm_phys.c:382
                }
            }
        }
    }
}

```

FIGURE 12

Il faut répéter ce processus impérativement sur chaque portion sensible. Chaque amélioration fera émerger une nouvelle zone critique. L'exécution du profiler avec OpenMP permet la mise en évidence les points chauds du code. Généralement, une partie séquentielle émergera, qu'il faudra songer soit à paralléliser, soit à optimiser finement le cas échéant.

7.2 Vectorisation

Les processeurs modernes sont composés d'instructions vectorielles de type SIMD. Notamment, les processeurs X86 possèdent les instructions SSE et AVX et les processeurs ARM les instructions Neon et sve. Les instructions de type SIMD consistent à effectuer une même opération sur plusieurs données à la fois, préchargées dans des registres spécifiques. Par exemple, la somme d'un vecteur avec un autre s'effectue très bien en vectorisation lorsque les données sont contenues continues en mémoire.

En général, on laisse le compilateur générer du code vectoriel, à l'aide des drapeaux `-mavx2` ou `-ftree-loop-vectorize` par exemple. Cependant, la génération automatique de code vectorielle ne se passe pas toujours très bien, et parfois une très légère modification au sein du code permet au compilateur de le générer convenablement. Une alternative est d'utiliser les directives OpenMP `simd` ou les intrinsics - qui appellent des codes assembleurs correspondants.

Note : En général, le compilateur INTEL génère un meilleur code vectorisé que ses concurrents.

8 Mesure de performances et Scalabilité

8.1 Validation des codes

Les modifications apportées au code sont susceptibles de modifier les résultats de la simulation selon deux principales raisons :

- Par ajout d'un bogue d'implantation. Par exemple, un message MPI qui n'est plus envoyé au bon rang, une mauvaise gestion des variables privées et publiques en OpenMP ;
- Par des effets d'optimisations altérant la précision du résultat - sans bug particulier d'implémentation. La précision machine étant limitée - même en flottant 64 bits - l'ordre des opérations ainsi que les instructions CPU utilisées peuvent altérer la précision du résultat. Voici diverses raisons de perte de précision au sein d'un code de calcul :
 - la non commutativité des opérations élémentaires. La norme IEEE754 - mais aussi les autres - induit le fait que, du fait de la précision machine, des opérations tels que $(B + C) + DB + (C + D)$ en général. Une modification simpliste du code peut mener à la modification de l'ordre des opérations - et ainsi altérer le résultat ;
 - l'utilisation d'instructions processeurs particuliers. En effet - prenons l'exemple de l'architecture **x86** - les processeurs possèdent parfois pour la même opération arithmétique plusieurs instructions différentes. Notamment, les processeurs **x86** proposent le jeu d'instruction **x87** permettant d'effectuer des opérations comme la somme sur 80 bits, mais en même temps des instructions **AVX** limitée à des données 64 bits. Sans spécification particulière au niveau de la compilation, l'utilisation d'un type d'instruction n'est pas explicitement définie et le compilateur peut mener à l'utilisation de la première ou de la seconde en fonction de la version. Cela est susceptible d'altérer la précision du résultat. Note : la vectorisation peut parfois faire perdre beaucoup de précision à cause d'un changement dans l'ordre des opérations (cf non commutativité des opérations).

Pour ces raisons, nous avons à disposition un programme permettant de valider le résultat via checksum. La checksum est très permissive - en ce qu'elle permet une altération des résultats par des optimisations agressives - mais permet tout-de-même la mise en évidence de bogues d'implémentation.

Le contrôle des optimisations a été mis en place à l'aide d'un script `bash bash/validation.sh` afin de contrôler les optimisations au long du développement :

- Différentes combinaisons d'optimisation sont compilées à l'aide de la compilation conditionnelle mise en place au sein du code ;
- ces programmes sont exécutés avec différents nombres de noeuds et de threads - en particulier 1 noeud et 1 thread comme base de validation ;
- en tant que simulation chaotique, plus le nombre d'itérations est important, plus la divergence entre une variante et une autre est probable. Il est nécessaire d'utiliser un nombre relativement important d'itérations afin que la simulation puisse mettre en exergue les erreurs d'implémentation.
- on contrôle alors que les checksums sont identiques sur toutes les combinaisons. Il faut prendre garde d'éviter l'explosion combinatoire et sélectionner judicieusement les variantes à tester. Aussi, le `sleep` est absent dans tous les codes du script de validation.

En **conclusion**, le script de validation a confirmé que toutes les combinaisons de variantes et de découpage rang/thread sont valides - du moins jusqu'à l'itération 5000. Puisque la simulation est chaotique, il est pratiquement impossible d'avoir la même checksum à ce niveau d'avancement de la simulation s'il y avait un problème d'implémentation.

Note : la checksum fourni n'a pas été utilisée car elle ne donne pas de solution assez loin. Il est nécessaire d'avoir une checksum allant jusqu'à un nombre d'itération de l'ordre de 600 pour que l'information de vibration initiale se répande sur tout le domaine d'étude. Ainsi, la checksum valide est générée à partir du code de base déboguée, après avoir été comparée à la checksum fournie - et cela sur un seul rang MPI.

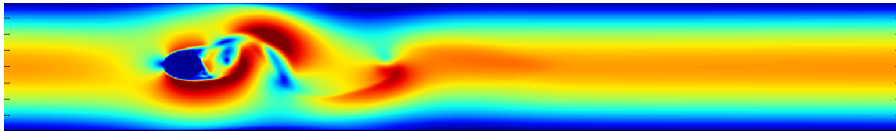


FIGURE 13 – Résultat de la simulation avec `reynolds=1000` pour une dimension 800x160

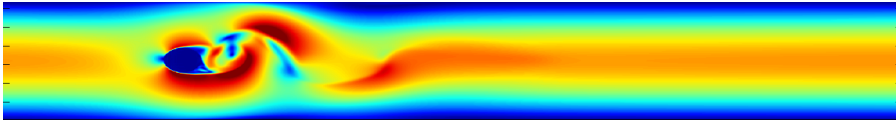


FIGURE 14 – Résultat de la simulation avec `reynolds=1000` pour une dimension 1600x320

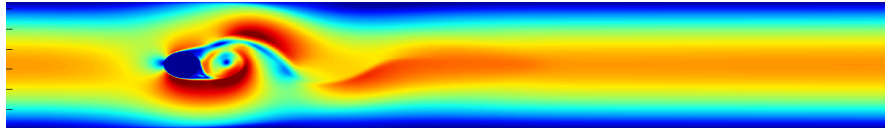


FIGURE 15 – Résultat de la simulation avec `reynolds=1000` pour une dimension 3200x640

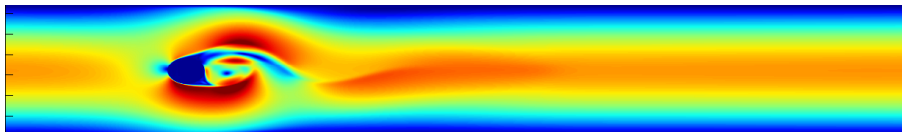


FIGURE 16 – Résultat de la simulation avec `reynolds=1000` pour une dimension 6400x1280

Visuellement, le raffinement semble adoucir la solution mais le phénomène physique est conservé. L'implémentation optimale est semble-t-il bien valide !

Note : Le pas de temps est divisé par deux à chaque raffinement de cellule. La dernière simulation nécessite des dizaines de milliers d'itérations !

8.2 Étude de scalabilité des codes

Afin de ne pas alourdir le document, l'étude de scalabilité est principalement étudiée entre la version de base et la version la plus optimisée. Un script `bash/scaling.sh` permet d'extraire

les informations de scalabilité faible et forte sur les codes de base (sans sleep) et optimisés. Plusieurs précisions sont à fournir :

La scalabilité d'une application HPC correspond à la capacité d'accélérer les calculs d'autant que le nombre d'entités de calculs augmente. On discerne généralement deux types de scalabilités :

- La scalabilité forte correspond à un cas de scalabilité dans un contexte de taille de problème constant. Ainsi, son étude se résume à étudier les performances de l'application en fonction du nombre de noeuds et de coeurs pour un problème donné ;
- la scalabilité faible correspond au cas de scalabilité pour lequel la taille du problème augmente avec le nombre d'entités de calcul. Ainsi, son étude se résume à étudier les performances en augmentant d'autant la taille de problèmes qu'est augmenté le nombre de noeuds/coeurs.

La configuration utilisée pour l'étude de scalabilité est celle du fichier initial `config.txt`. La direction d'augmentation de la taille de domaine pour l'étude de scalabilité faible est selon la longueur - afin de ne pas alourdir le document - mais il est aussi judicieux d'étudier la scalabilité selon la largeur, voire selon les deux directions. Cette dernière proposition nécessiterait d'avoir un matériel digne de son nom puisqu'il faudrait par rigueur quadrupler les ressources à chaque augmentation de taille de problème. Aussi, il est à noter que les résultats de scalabilité dépendent de la taille du problème du fait de problématiques de vitesses de mémoire. Voici la démarche d'étude de scalabilité ;

- Un script `bash/scaling.sh` est développé afin de lancer de nombreux tests automatiquement ;
- le script lance des mesures de temps sur le code initial - sans le `sleep` - et sur le code le plus optimisé grâce au processus de compilation conditionnelle ;
- les mesures sont effectuées pour différentes configurations de noeuds/threads openMP. Il est à savoir que, sur des machines monosockets, seul un seul thread est lancé dans le contexte MPI et seul un rang est lancé dans un contexte OpenMP. En revanche, sur machine hpc, les variables sont changées pour charger au maximum la machine.
- Les mesures de temps proviennent de compteurs `fenced_rdtsc`. Les sauvegardes de fichiers sont désactivées. L'étude n'exploite pas le simultaneous threading.

8.3 Mesures sur le probleme initial

Voici une compilation des résultats obtenus par l'intermédiaire du script de test. Les mesures ont été faites avec la configuration de base et sans enregistrement des données sur le disque - cette partie étant de fréquence purement arbitraire. Les mesures suivantes exposent les résultats pour le code de **base** puis **optimise** en scalabilité **faible** et **forte** sur plusieurs machines :

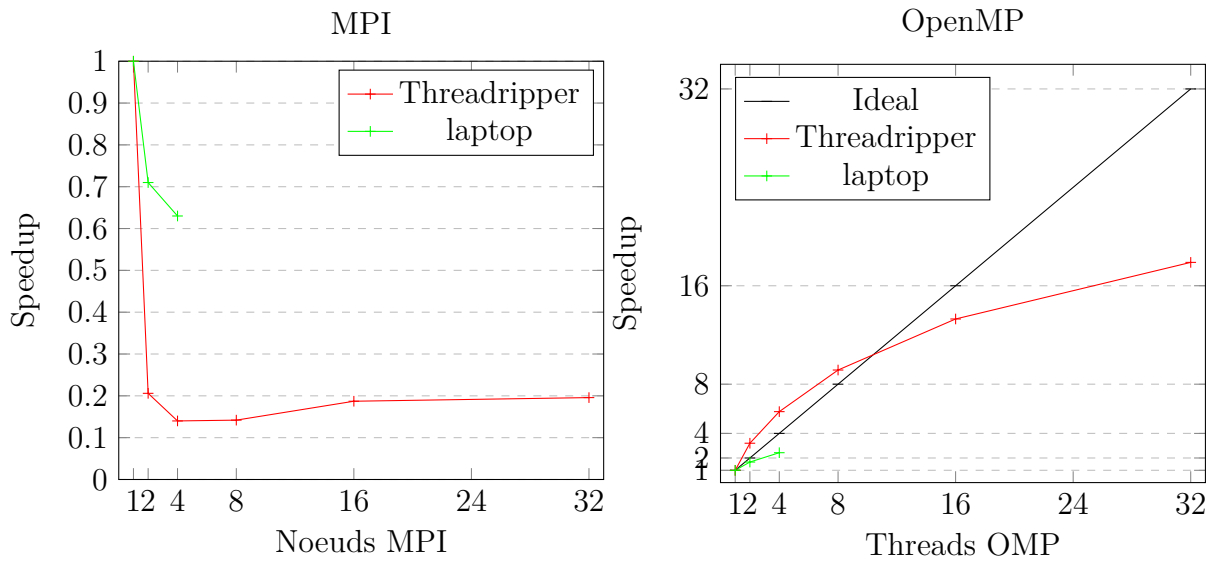


FIGURE 17 – Scalabilité forte en MPI et OpenMP avec le code de base sur un problème de configuration 800×160 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

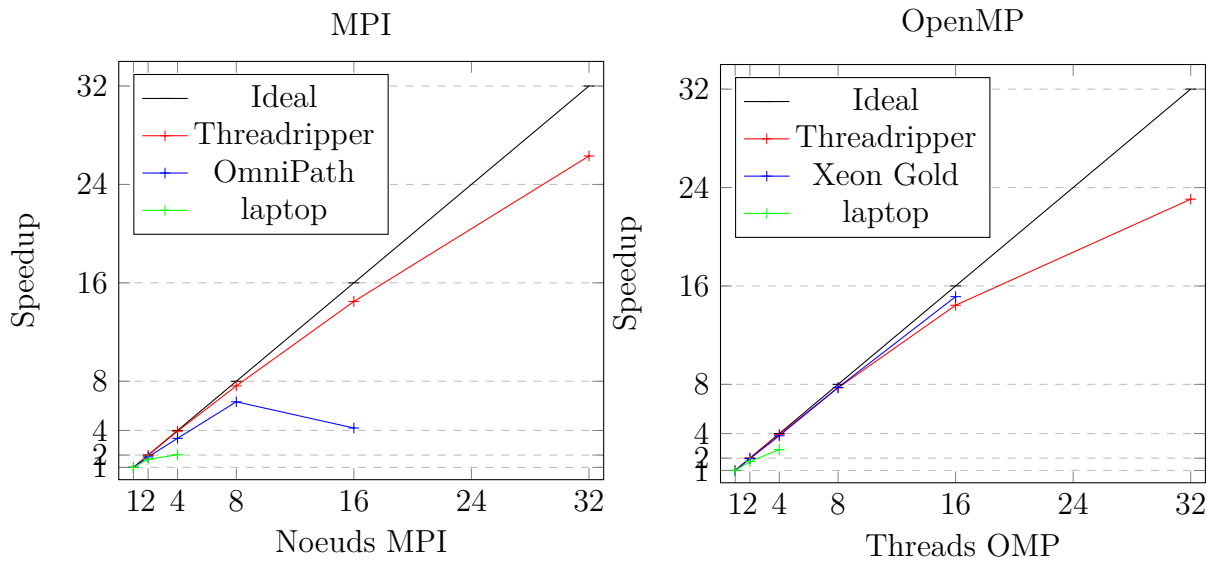


FIGURE 18 – Scalabilité forte en MPI et OpenMP avec le code optimisé sur un problème de configuration 800×160 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

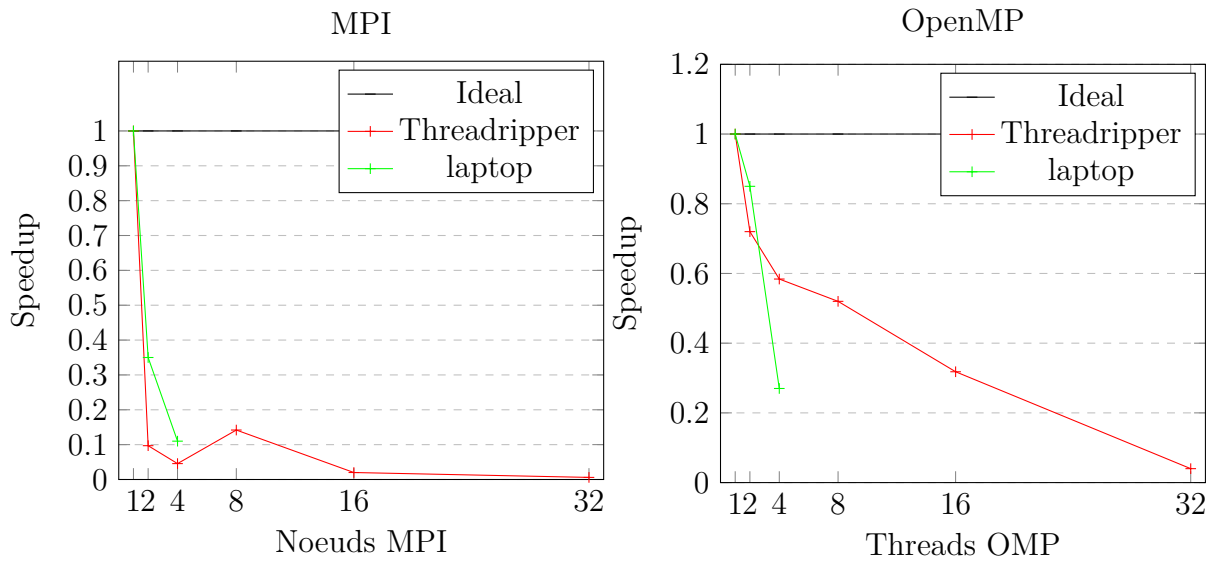


FIGURE 19 – Scalabilité faible selon x en MPI et OpenMP avec le code de base sur un problème de configuration initialement 800×160 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

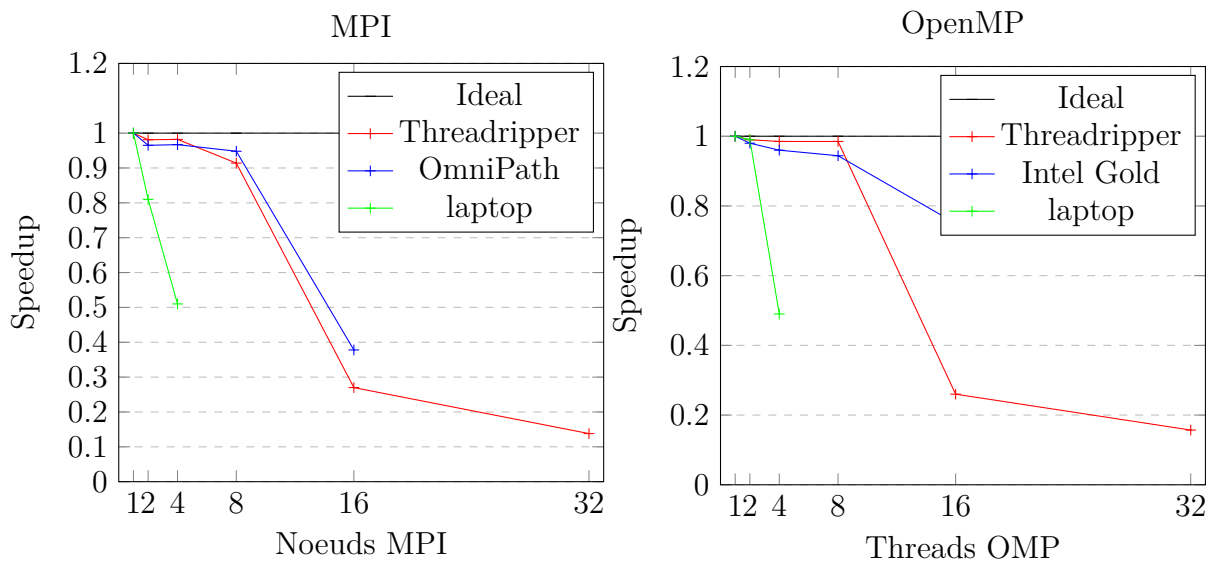


FIGURE 20 – Scalabilité faible selon x en MPI et OpenMP avec le code optimisé sur un problème de configuration 800×160 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

- concernant le code de base, il est clair que le code de base n'a aucune capacité de scalabilité MPI. Cela provient bien-sûr des grandes latences propres à chaque envoi MPI composé uniquement que d'un flottant. Pourtant, le code a une réelle capacité de scalabilité, en témoigne les résultats en OpenMP. On note cependant un résultat étrange sur la machine THREADRIPPER malgré de nombreuses vérifications de configuration. De toutes les manières, le passage à l'échelle nécessite nécessairement une refonte de la partie MPI!

- en revanche, le code optimisé fait preuve de très bons résultats. La scalabilité forte est presque parfaite bien que les performances en OpenMP semblent réduites. Cela provient probablement de phénomènes de type *false-sharing* qui peut encore s'améliorer. La scalabilité faible laisse penser que pour des problèmes plus grands sur des machines non-distribuées, l'empreinte mémoire du problème est tel que chaque accès mémoire est plus lent.

Il est nécessaire de préciser les performances en monothread et monoprocess afin de pouvoir comparer les performances d'une machine à une autre :

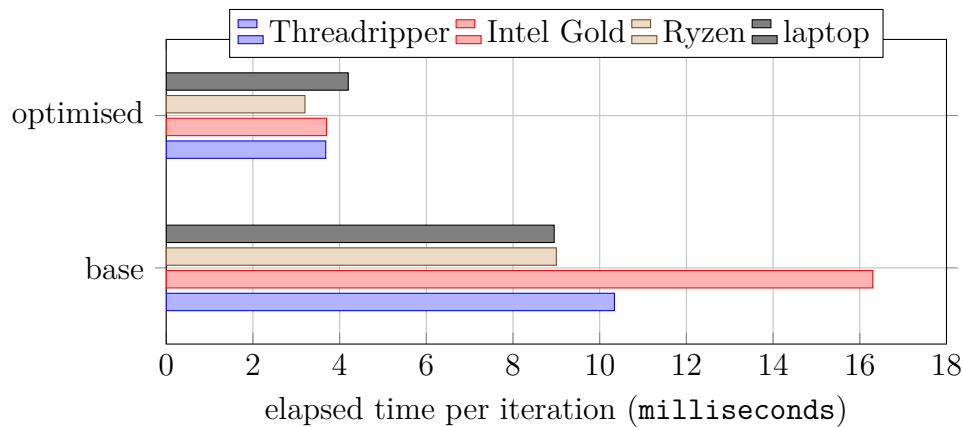


FIGURE 21 – Performance en séquentiel (single core)

8.4 Mesures sur un problème plus grand

Il est intéressant de faire les mesures sur des problèmes plus grands - le vrai intérêt du HPC - afin de mettre en exergue les problèmes induits par la complexité et le coût mémoire de la simulation. On propose dorénavant d'utiliser un problème de taille 8000×1600 contre 800×160 auparavant. L'empreinte mémoire est 100 fois plus grande et l'empreinte MPI est 10 fois plus grande. On s'attend à avoir une baisse des performances du fait de l'empreinte mémoire :

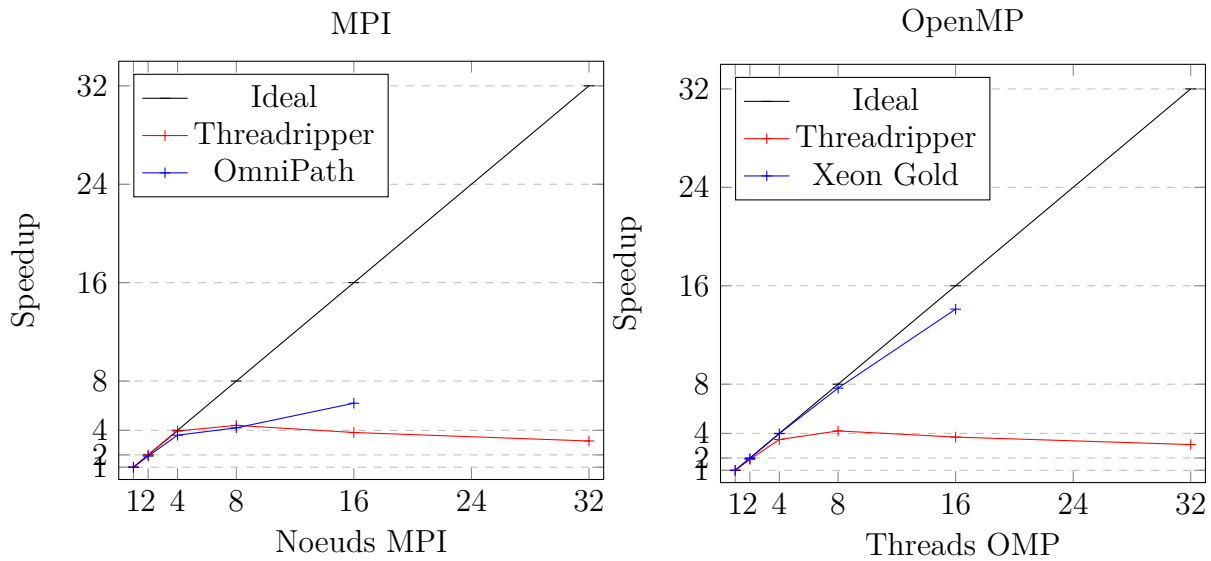


FIGURE 22 – Scalabilité forte en MPI et OpenMP avec le code optimisé sur un problème de configuration 8000×1600 . Les études sont faites en thread unique (MPI) ou rang unique (OpenMP).

Le plateau dont souffre le système THREADRIPPER provient fort probablement de problèmes de bande passante avec la mémoire RAM. En effet, la plateforme permet d'avoir une RAM en *quad channels*, alors que la plateforme INTEL permet d'avoir 6 *channels* de mémoire. Cela permet d'augmenter la bande passante - et donc les performances - dans le cas de programmes étant *memory bound*.

Il est nécessaire de préciser les performances en monothread et monoprocess afin de pouvoir comparer les performances d'une machine à une autre :

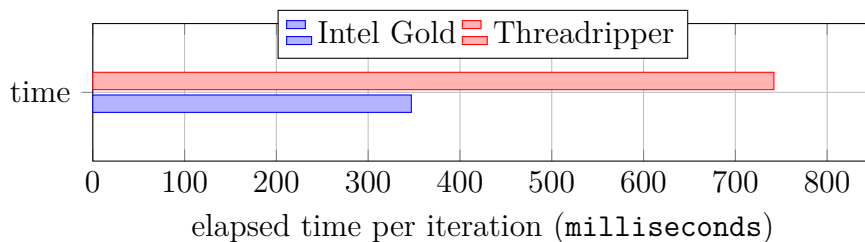


FIGURE 23 – Performance en séquentiel (single core)

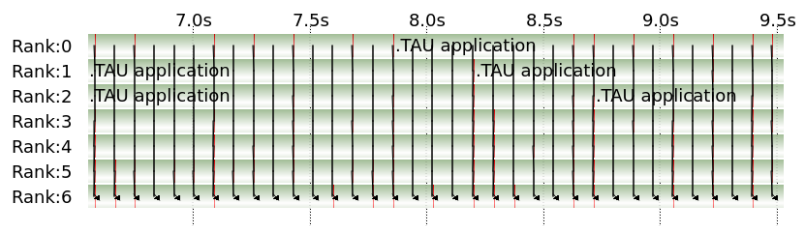


FIGURE 24 – Timeline de trace MPI pour un problème de grande taille sur quelques dizaines d'itérations

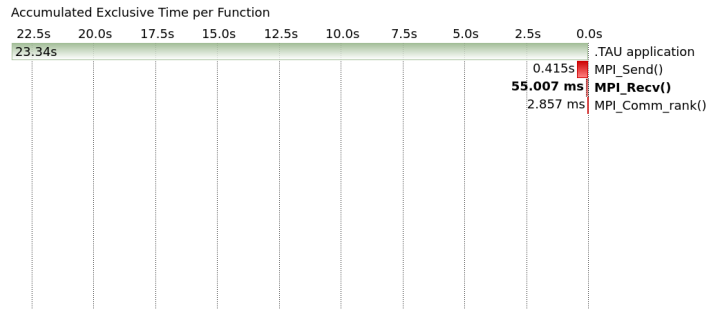


FIGURE 25 – Répartition des temps au cours de l'exécution sur une fenêtre de plusieurs dizaines d'itérations - 8 rangs

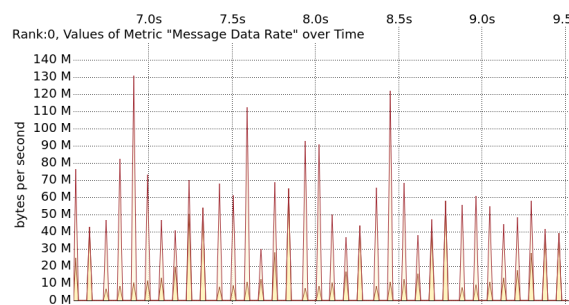


FIGURE 26 – Représentation du débit moyen dans le même cas d'usage que précédemment

Pour des problèmes de grande taille, les performances semblent maintenant limitées par le débit de la RAM (*memory bound*). EN général, les très grands problèmes sont *memory bound* car la complexité des envois MPI est souvent linéaire. Dans le cas d'un problème *memory bound*, les optimisations MPI et séquentielles ne servent plus à grand chose et il faut parfois repenser les optimisations.

Voici un rapport ScoreP final présentant les points chauds lors de l'exécution du grand problème sur machine THREADRIPPER :

1	flt	type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
2		ALL	3,100,681,255	1,106,353,913	355.24	100.0	0.32	ALL
3		USR	3,100,672,926	1,106,351,721	298.62	84.1	0.27	USR
4		OMP	3,753	810	6.22	1.8	7678.74	OMP
5		MPI	3,065	572	33.50	9.4	58565.29	MPI
6		COM	2,080	800	16.90	4.8	21128.56	COM
7		SCOREP	41	10	0.00	0.0	24.69	SCOREP
8								
9		USR	1,733,562,688	637,954,880	92.13	25.9	0.14	get_vect_norme_2
10		USR	825,181,344	317,377,440	135.82	38.2	0.43	compute_equilibrium_profile
11		USR	158,080,000	32,000,000	5.11	1.4	0.16	get_cell_velocity
12		USR	158,080,000	32,000,000	5.21	1.5	0.16	get_cell_density
13		USR	150,885,072	58,032,720	9.07	2.6	0.16	helper_compute_poiseuille
14		USR	74,880,000	28,800,000	49.71	14.0	1.73	compute_cell_collision
15		USR	2,428,452	185,337	0.03	0.0	0.16	compute_bounce_back
16		USR	1,872	720	0.00	0.0	0.17	lbm_comm_sync_ghosts_diagonal
17		MPI	1,159	171	18.60	5.2	108784.04	MPI_Send
18		MPI	1,098	171	1.06	0.3	6217.82	MPI_Recv
19		USR	936	360	0.00	0.0	0.19	lbm_comm_sync_ghosts_vertical
20		COM	936	360	0.00	0.0	0.59	lbm_comm_sync_ghosts_horizontal
21		OMP	783	90	0.00	0.0	0.83	!\$omp parallel @lbm_phys.c:312
22		OMP	783	90	0.00	0.0	0.59	!\$omp parallel @lbm_phys.c:357
23		OMP	783	90	0.00	0.0	1.10	!\$omp parallel @lbm_phys.c:399
24		MPI	364	140	0.00	0.0	0.80	MPI_Comm_rank
25		USR	260	10	1.43	0.4	143037.59	save_frame
26		COM	234	90	0.00	0.0	21.34	propagation
27		COM	234	90	0.00	0.0	6.19	collision
28		COM	234	90	0.00	0.0	17.26	special_cells
29		COM	234	90	0.00	0.0	3.72	lbm_comm_ghost_exchange
30		OMP	234	90	0.07	0.0	730.71	!\$omp for @lbm_phys.c:312
31		OMP	234	90	0.00	0.0	1.44	!\$omp implicit barrier @lbm_phys.c
32	:330	OMP	234	90	4.06	1.1	45151.77	!\$omp for @lbm_phys.c:357
33	:363	OMP	234	90	0.00	0.0	1.61	!\$omp implicit barrier @lbm_phys.c
34		OMP	234	90	2.09	0.6	23211.67	!\$omp for @lbm_phys.c:399
35	:414	OMP	234	90	0.00	0.0	8.94	!\$omp implicit barrier @lbm_phys.c
36		USR	208	80	0.00	0.0	0.19	helper_get_rank_id
37		MPI	204	30	0.00	0.0	94.52	MPI_Reduce
38		MPI	136	20	13.64	3.8	681964.17	MPI_Barrier
39		USR	78	30	0.07	0.0	2452.78	Mesh_release
40		USR	78	30	0.00	0.0	5.47	Mesh_init
41		COM	52	20	0.00	0.0	0.99	setup_init_state
42		USR	52	20	0.02	0.0	1117.13	setup_init_state_border
43		COM	52	20	16.90	4.8	844870.42	
44		USR	52	20	0.00	0.0	192.88	setup_init_state_circle_obstacle
45		MPI	52	20	0.00	0.0	0.83	MPI_Comm_size
46		SCOREP	41	10	0.00	0.0	24.69	lbm
47		COM	26	10	0.00	0.0	53.31	main
48		USR	26	1	0.00	0.0	36.85	close_file
49		USR	26	1	0.00	0.0	11.39	open_output_file
50		USR	26	1	0.00	0.0	2.92	write_file_header
51		USR	26	10	0.00	0.0	138.31	lbm_mesh_type_t_release
52		USR	26	10	0.00	0.0	4.42	lbm_mesh_type_t_init
53		COM	26	10	0.00	0.0	16.70	save_frame_all_domain
54		USR	26	10	0.00	0.0	0.83	lbm_comm_release
55		COM	26	10	0.00	0.0	2.16	lbm_comm_init
56		COM	26	10	0.00	0.0	11.56	lbm_comm_print
57		USR	26	10	0.00	0.0	0.31	lbm_helper_pgcd
58		USR	26	1	0.00	0.0	0.42	print_config
59		USR	26	10	0.00	0.0	19.46	load_config
60		USR	26	10	0.00	0.0	0.50	update_derived_parameter
61		USR	26	10	0.00	0.0	1.26	setup_default_values
62		MPI	26	10	0.01	0.0	888.53	MPI_Finalize
63		MPI	26	10	0.18	0.1	18289.48	MPI_Init

Listing 9 – SCOREP/SCALASCA - code MPI de base - 10 itérations

Il est clair que la partie MPI est minoritaire et il est peu utile d’optimiser la partie MPI par des appels asynchrones dans le cas présent - comme supposé en hypothèse. Cependant, il y a deux fonctions très souvent appelées qui compensent la majorité du temps d’exécution. Ce sont ses parties qu’il faudra améliorer pour gagner en performances. Il faudra songer a de la vectorisation et a une meilleure parallélisation OpenMP.

8.5 Mesures sur le cluster Ruche

Afin de tester le code sur un véritable environnement HPC, il a été fait des mesures de scalabilité en poussant au maximum les performances - c’est adire en étudiant :

- étudiant par rapport au nombre de noeuds mais avec un nombre de threads OpenMP maximal (`OMP_NUM_THREADS=16`) ;
- étudiant par rapport au nombre de threads OpenMP mais avec un nombre de noeuds maximal (`srun -n 16 ./lbm`).

Les mesures ont été faites sur une allocation 20×20 . Ce surplus permet d'éviter des grosses pertes de performance lorsque une autre personne travaille sur la même machine, les allocations étant distribuées et non privées.

On remarque que la scalabilité du code MPI devient inquiet ante au niveau de 16 noeuds, alors qu'elle est très convenable dans le cas du parallélisme à mémoire partagée. Ce problème peut provenir d'un goulot d'étranglement au niveau des communications MPI. Je n'ai hélas à ce moment la pas pris le soin de vérifier les latences des communications.

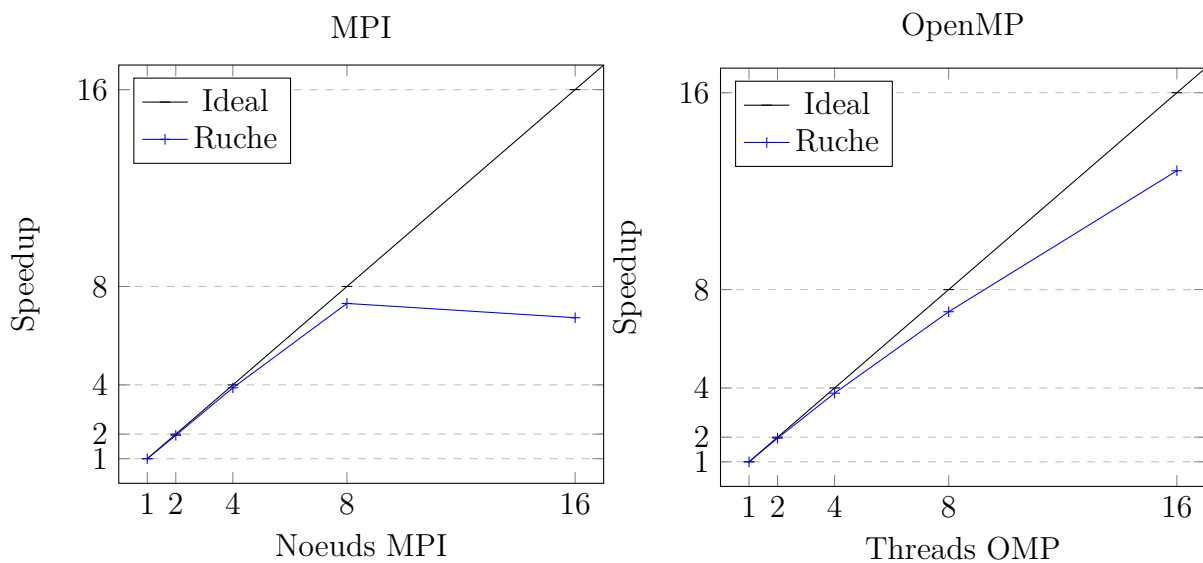


FIGURE 27 – Scalabilité forte en MPI et OpenMP avec le code optimisé sur un problème de configuration 8000×1600 . Les études sont faites De sorte à charger au maximum la machine. Notamment, l'étude MPI est effectuée avec 16 threads par noeuds et l'étude OpenMP s'effectue avec 16 noeuds.

9 Conclusion

La simulation LBM fait partie des types d'algorithmes réputées pour être fortement parallélisables et nous avons pu le démontrer à travers cet exercice. Les paradigmes de mémoire partagée (MPI) et mémoire distribuées (OMP) ont été exploitées. D'un temps initial de $11ms$ par itérations, nous sommes parvenus à $0.12ms$ par itérations sur plate-forme THREADRIPPER et à $0.08ms$ sur une machine HPC avec le problème de base. Il reste néanmoins des axes d'amélioration, notamment par une vectorisation manuelle et de meilleurs déroulages des boucles critiques.

Finalement, la bonne utilisation des outils de profilage et de traces permettent d'améliorer aisément les performances à l'exécution d'un programme. SCALASCA permet une compréhension rapide des points chauds d'un programme selon les paradigmes multiprocess et multiprocesses. Des affichages en Timeline semblent fortement aider pour comprendre comment améliorer les ordres d'appel MPI.