Optimisation du projet LBM

1) Introduction

Ce document a pour but de fournir un exemple de rapport d'optimisation du projet LBM (Lattice Boltzman Method) donné aux étudiants et parallélisé en MPI. La version fournie aux étudiants étant volontairement boguée, la première section traitera de l'utilisation de GDB pour corriger le problème. Le but est de trouver rapidement les erreurs sans avoir à lire au préalable tout le code.

2) Débogage de la faute de segmentation

On commence par compiler le programme et le lancer tel quel :

```
user@localhost: make
user@localhost: ./lbm
================== CONFIG ==================
iterations           = 16000
…
============================================
RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 )
(WH 802 162 )
*** Process received signal ***
```

Le programme se termine par une faute de segmentation liée à une mauvaise adresse. Afin de trouver la source du problème rapidement, nous allons lancer le programme dans GDB, mais avant cela, on s'assure que le programme est bien compilé avec l'option « *-g* » afin que le programme contienne des informations utiles au débogueur, dans le fichier *Makefile* :

```
CFLAGS=-Wall -g
```

On recompile si nécessaire et on lance le programme dans GDB :

```
user@localhost: gdb ./lbm
```

GDB offre alors un invite commande dans lequel on lance l'exécution à l'aide de la commande *run* :

```
(gdb) run
```

Le programme se lance et plante à nouveau. Mais cette fois, GDB vous affiche des informations supplémentaires et vous rend la main :

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004031d6 in setup_init_state_global_poiseuille_profile
         (mesh=0x7fffffffd220, mesh_type=0x7fffffffd1f0,
          mesh_comm=0x7fffffffd0b0) at lbm_init.c:85
          Mesh_get_cell(mesh, i, j)[k] = compute_equilibrium_profile(v,density,k);
```

GDB nous affiche la ligne posant problème, dans *lbm_init.c* ligne 85. Sur la ligne concernée, *mesh* est la seule structure complexe contenant des pointeurs. On peu alors afficher son contenu sans recompiler :

```
(gdb) print mesh
$1 = 0x7fffffffd220
(gdb) print *mesh
$2 = {cells = 0x0, width = 802, height = 162}
```

Ici, on constate immédiatement que le champ, *mesh->cells* contient une adresse nulle (0x0), qui est pourtant utilisée à l'intérieur de la fonction *Mesh_get_cell()*. Il y a donc certainement un problème à l'initialisation de l'objet *mesh*.

Le code source de *Mesh_get_cell()* peut être affiché directement dans GDB :

```
(gdb) list Mesh_get_cell
114 static inline lbm_mesh_cell_t Mesh_get_cell( const Mesh *mesh, int x, int y)
115 {
116     return &mesh->cells[ (x * mesh->height + y) * DIRECTIONS ];
117 }
```

Afin de savoir d'où provient *mesh* on peut afficher la pile d'appels :

```
(gdb) backtrace
#0  0x00000000004031d6 in setup_init_state_global_poiseuille_profile
    (mesh=0x7fffffffd220, mesh_type=0x7fffffffd1f0, mesh_comm=0x7fffffffd0b0)
    at lbm_init.c:85
#1  0x0000000000403491 in setup_init_state (mesh=0x7fffffffd220,
    mesh_type=0x7fffffffd1f0, mesh_comm=0x7fffffffd0b0) at lbm_init.c:155
#2  0x0000000000402003 in main (argc=1, argv=0x7fffffffd338) at main.c:141
```

En remontant la pile, on constate que *mesh* est transmit depuis la ligne 141 de la fonction main. À partir de là, on va lire les sources et remonter au problème pour le corriger. On constate qu'effectivement *mesh* n'est pas alloué, mais initialisé à NULL dans la fonction *Mesh_init()* dans *lbm_struct.c*.

3) Débogage du dead-lock

La version fournie se bloque en fin d'exécution juste après avoir calculé le dernier pas de temps si on utilise plusieurs processus. Afin de voir apparaitre le problème rapidement on modifie le fichier de configuration (*config.txt*) pour ne calculer que 50 pas de temps.

```
iterations = 50
```

On lance avec deux processus MPI :

```
user@localhost: mpirun –np 2 ./lbm
```

Pour trouver la ligne fautive on lance le programme dans GBD, mais cette fois, nous avons deux processus à surveiller et non plus un seul. Une astuce consiste à lancer chaque processus dans un terminal séparé :

```
user@localhost: mpirun –np 2 xterm –e gdb –command=gdb-script ./lbm
```

La commande se compose de :

- *mpirun -np 2* : lancement habituel des deux processus MPI.
- *xterm* -e : plutôt que lancer notre programme directement on lance un terminal (*xterm*) qui exécute immédiatement la commande qui suit l'option –e, pour nous il s'agit de GDB.
- *gdb --command=gdb-script* : chaque terminal va lancer GDB. Afin de ne pas avoir à lancer nous même « *run* » sur chacun des terminaux on donne un script s'en chargeant via « --command=gdb-script ».
- Le fichier *gdb-script* doit contenir au minimum la commande « *run* » pour GDB.
- *./lbm* : on souhaite déboguer ce programme.

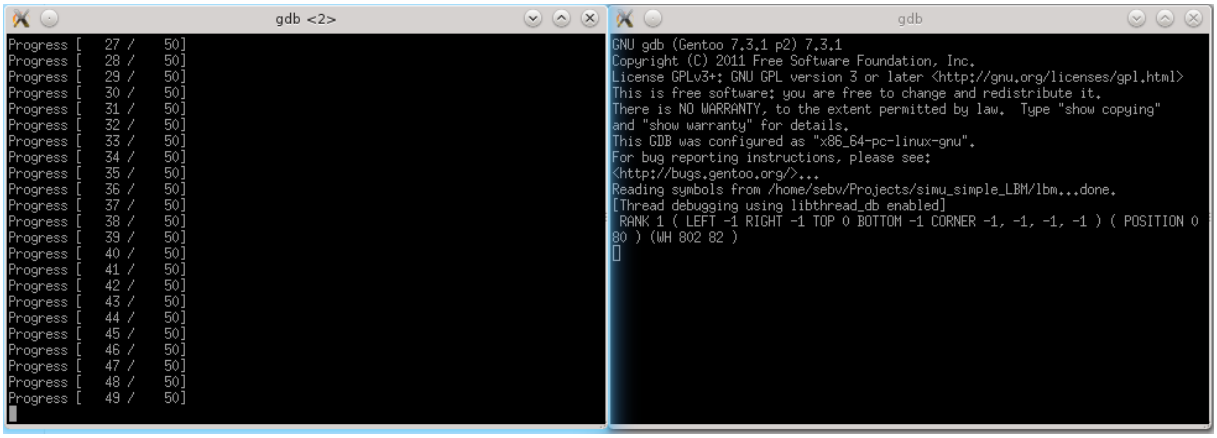


Figure 1 : Débogage avec GDB et XTerm des deux processus MPI.

Maintenant que le processus est bloqué on peut l'interrompre grâce à là combinaison de touche « CTRL + C » sur une des deux fenêtres. GDB nous rend alors la main. En exécutant un *backtrace* :

```
(gdb) backtrace
…
#7  0x00007ffff78fab46 in PMPI_Barrier () from /usr/lib/libmpi.so.0
#8  0x0000000000402148 in main (argc=1, argv=0x7fffffffd028) at main.c:18
```

On se rend compte que l'un des deux processus (celui affichant la progression) est bloqué sur une barrière MPI dans *main.c* ligne 18. En s'y rendant, on constate rapidement que la barrière est appelée uniquement sur le nœud 0 d'où le blocage.

REGLE N°1 : Avant de commencer à optimiser, il faut déjà avoir un programme qui marche.

OUTIL : Concernant le débogage, ici on a utilisé GDB en mode console, il existe bien sur des interfaces pour ça. Pour MPI on trouve notamment DDT (http://www.allinea.com/), ou TotalView (http://www.roguewave.com/).

4) Description de la machine de test

Afin de pouvoir reproduire les résultats, il est toujours bon de noter les caractéristiques de la machine utilisée pour les tests. Ceci permet surtout d'avoir bien en tête l'environnement dans lequel on travaille.

	Station de travail	Cluster
Processeurs	Intel Core i7 2600K @ 3.40 Ghz	Intel Xeon X5560 @ 2.80Ghz
Coeurs / processeurs	4	4
Nombre de processeurs par noeuds	1	2
Hyperthreading	Oui	Non
Noeuds NUMA	1	1
OS	Gentoo – Kernel 3.2.1	Bull Redhat 6
Compilateur	GCC 4.5.3	ICC 11.1
MPI	OpenMPI 1.4.3	Bull OpenMPI 1.1.14.1
Noeuds	1	100

5) Option de compilation

Avant tout, assurez-vous des options de compilation qui sont utilisées. Dans le cadre de notre projet, c'est justement l'occasion de se rendre compte qu'aucune option d'optimisation n'est passée au compilateur.

Lorsque le code contient des assertions (fonction *assert()* de *assert.h*) il est préférable d'ajouter la définition de la macro NDEBUG qui désactive ces dernières, toutefois dans notre cas il n'y en a pas assez pour que cela ait un impact significatif :

```
CFLAGS=-Wall -DNDEBUG
```

Avec ICC, le mode de compilation par défaut est équivalent à –O2, mais avec GCC on est par défaut en mode –O0, soit aucune optimisation. Comme nous travaillons avec GCC, il est préférable d'utiliser au minimum –O2 ou –O3.

```
CFLAGS=-Wall -DNDEBUG -O3
```

Il peut également être utile de préciser l'architecture cible afin de permettre à GCC d'utiliser toutes les optimisations associées :

```
CFLAGS=-Wall -DNDEBUG -O3 -march=core2 -msse4.2
```

La figure 2 montre justement les gains en séquentiel que l'on obtient sur la station de travail avec ces différentes options pour un problème 800 * 160 – 2000. Les temps sont ici mesurés à l'aide de la commande time. Les sorties sont désactivées pour cette mesure.

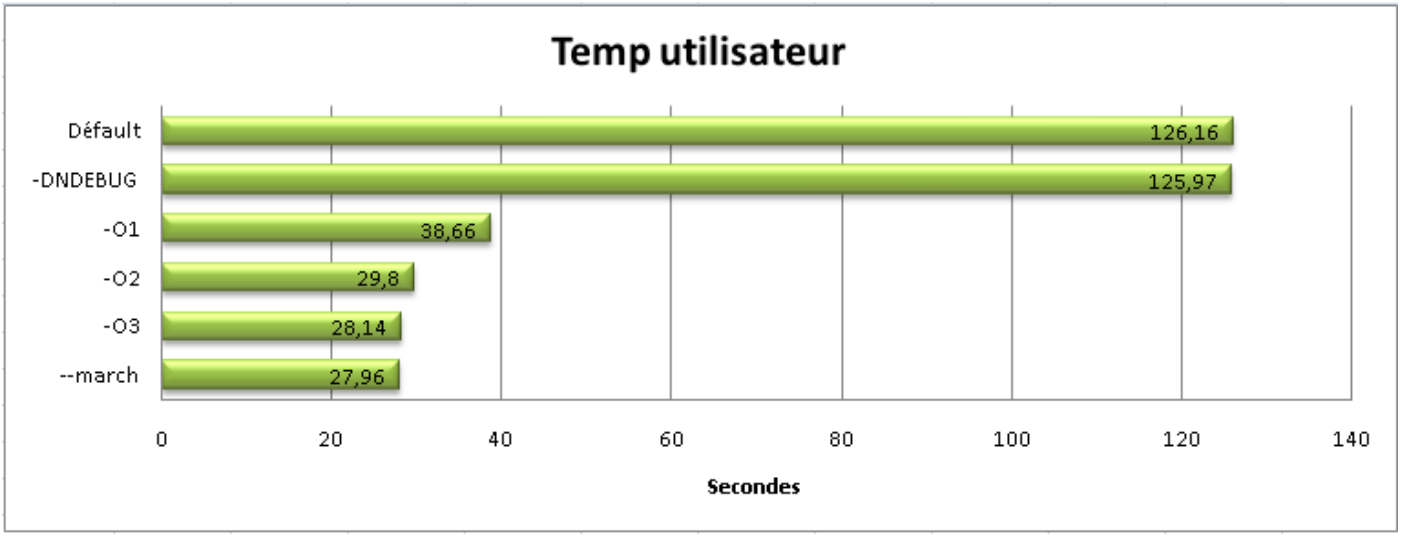


*Figure 2 : Temps utilisateur pour différentes options d'optimisation pour un problème de taille 800 * 160 et 2000 pas de temps sur la station de travail.*

Avec ces options de compilation, on a déjà une accélération d'un facteur 4.5 sur la version séquentielle.

REGLE N°2 : Le compilateur sait faire beaucoup d'optimiser du code, mais il faut lui demander de le faire.

INFO : Il existe beaucoup d'options que l'on trouvera dans le manuel du compilateur. Attention toutefois à bien comprendre leur impact, car certaines options ne garantissent un résultat juste que sous certaines conditions. C'est par exemple le cas de l'option --fstrict-aliasing.

ATTENTION : dans les dernières versions de GCC, -O3 implique --fstrict-aliasing, cette option peut provoquer des résultats invalides avec certains types d'utilisation des pointeurs. Si -O2 est en général sans risque -O3 demande tout même certaines précautions.

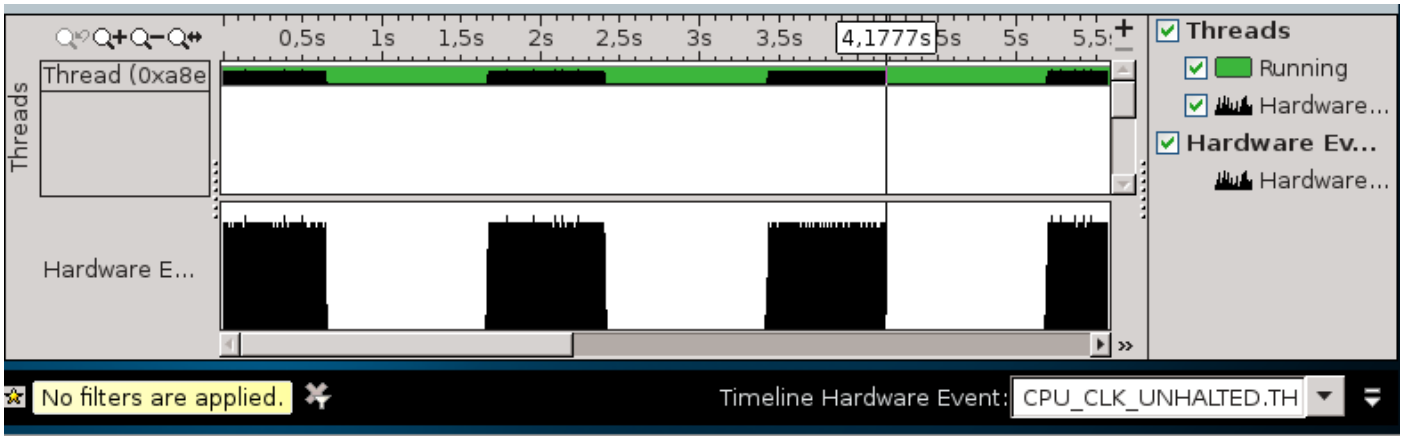
SPEED-UP : 4.5

6) Temps d'attente : sleep()

Lors des mesures de performance précédentes on remarque un écart anormal entre le temps réel d'exécution et le temps d'occupation du processus (temps utilisateur) que l'on obtient à l'aide de la commande *time* :

```
user@localhost: time ./lbm
…
real    1m10.140s
user    0m30.011s
sys     0m0.047s
```

Visuellement, il y une attente à intervalle régulier (tous les 50 pas de temps), ce qui se confirme par une analyse à l'aide *VTune Amplifier* :



Le processus est régulièrement mis en pause avec une durée de l'ordre de la seconde. Soit on comprend tout de suite que le problème vient de l'étape de sauvegarde qui est effectivement appelée tous les 50 pas de temps. Soit on décide de faire une analyse à l'aveugle, nous détaillerons cette deuxième approche considérant qu'il est trop complexe d'entrer tout de suite dans le code pour trouver le problème.

Des attentes de ce type ne peuvent qu'avoir lieu suite à un appel système, on peut donc les tracer à l'aide de la commande « *strace* » :

```
user@localhost: strace ./lbm
```

On remarque alors que l'on attend systématiquement suite aux appels :

```
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {0x7f417de6d750, [CHLD], SA_RESTORER|SA_RESTART,
0x7f417d2bbad0}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({1, 0}, 0x7fffe8b39600)       = 0
```

On peut vérifier l'hypothèse en filtrant uniquement les *nanosleep* :

```
user@localhost: strace -e nanosleep ./lbm
```

Clairement ils coïncident avec les ralentissements, il ne reste alors plus qu'à le tracer via GDB pour trouver le fautif, l'appel à *nanosleep* a toute les chance d'être lié à un appel à *sleep()* :

```
user@localhost: gdb ./lbm
(gdb) break sleep

(gdb) run
Breakpoint 1, 0x00007ffff68e8480 in sleep () from /lib64/libc.so.6
```

```
(gdb) backtrace
#0  0x00007ffff68e8480 in sleep () from /lib64/libc.so.6
#1  0x000000000040571c in save_frame_all_domain (fp=0x80f0c0,
    source_mesh=0x7fffffffd1f0, temp=0x7fffffffd1d0) at lbm_comm.c:359
#2  0x0000000000401e2b in main (argc=1, argv=0x7fffffffd338) at main.c:145
```

En allant voir le code, il s'avère que *FLUSH_IO*() est en fait une macro vers *sleep*(1). Ce genre d'attente n'a aucune justification ici. Elle peut donc être supprimée sans risque. Les IO sont synchrones, et même si elles étaient asynchrones, un *sleep* ne serait pas la solution pour s'assurer de l'écriture des données.

Expérimentalement, on vérifie bien que les temps d'exécution et d'occupation du CPU redeviennent similaires.

INFO : L'introduction du *sleep* est ici très artificielle, mais il arrive que de tels appels soient oubliés dans les codes suite à des phases de débogage, un simple « *grep sleep* » sur le code est parfois très efficace pour les trouver !

SPEED-UP : 1 seconde par sauvegarde, soit un speed-up de **2.3** avec la configuration par défaut en séquentielle.

7) Scalabilité d'origine

Jusqu'à maintenant nous avons préparé le terrain en réglant les problèmes de base sur les performances séquentielles. Il est maintenant temps de passer aux performances parallèles. Pour cela, on commence par une évaluation de la scalabilité du code fourni.

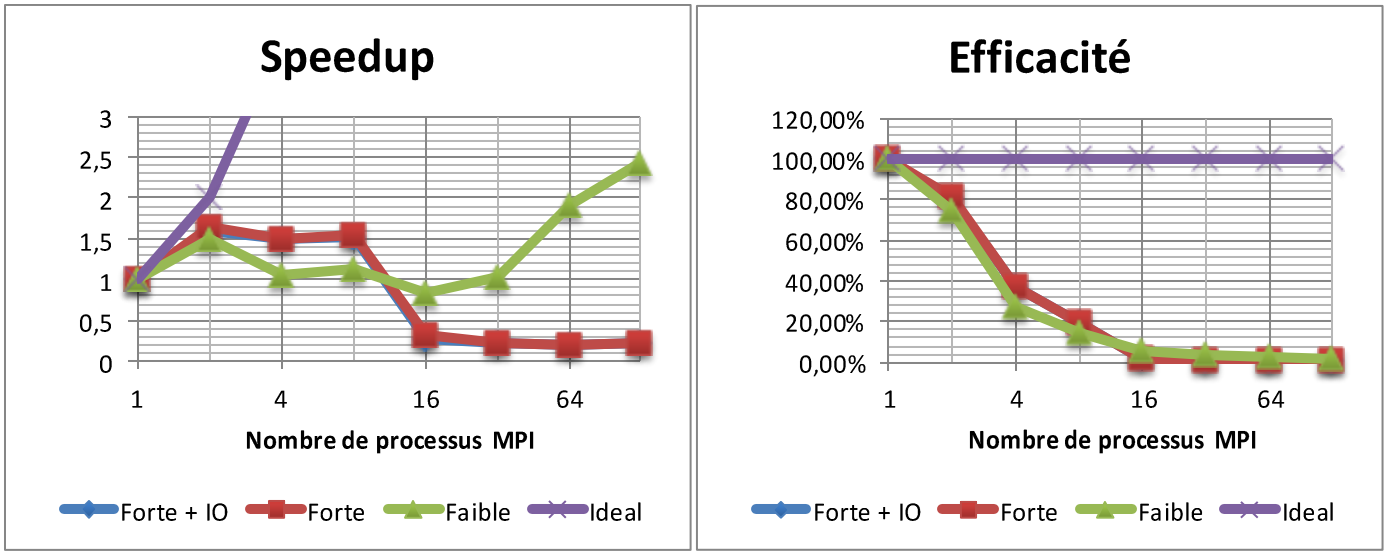


Figure X : Scalabilité faible (nb=1, 800x320 – 2000) et forte (1600 x 320 – 1000).

Clairement le programme n'est pas scalable, les performances ont même tendance à se réduire lorsqu'on grossit le problème au point que le test donne une scalabilité faible moins bonne que la forte avec un effondrement complet à partir de 4 cœurs.

8) Première analyse des communications

Pour commencer, on utilise *Scalasca* pour faire une première évaluation des communications MPI de l'application. Pour cela, on modifie le *Makefile* de la façon suivante et on recompile :

```
MPICC=scalasca –instrument mpicc
```

On lance une mesure avec un problème 800 x 160 – 200 sur deux cœurs via la commande :

```
user@localhost: scalasca –analyse mpirun –np 2 ./lbm
```

On peut alors regarder les résultats avec :

```
user@localhost: scalasca –examine ./epik_lbm_2_sum
```

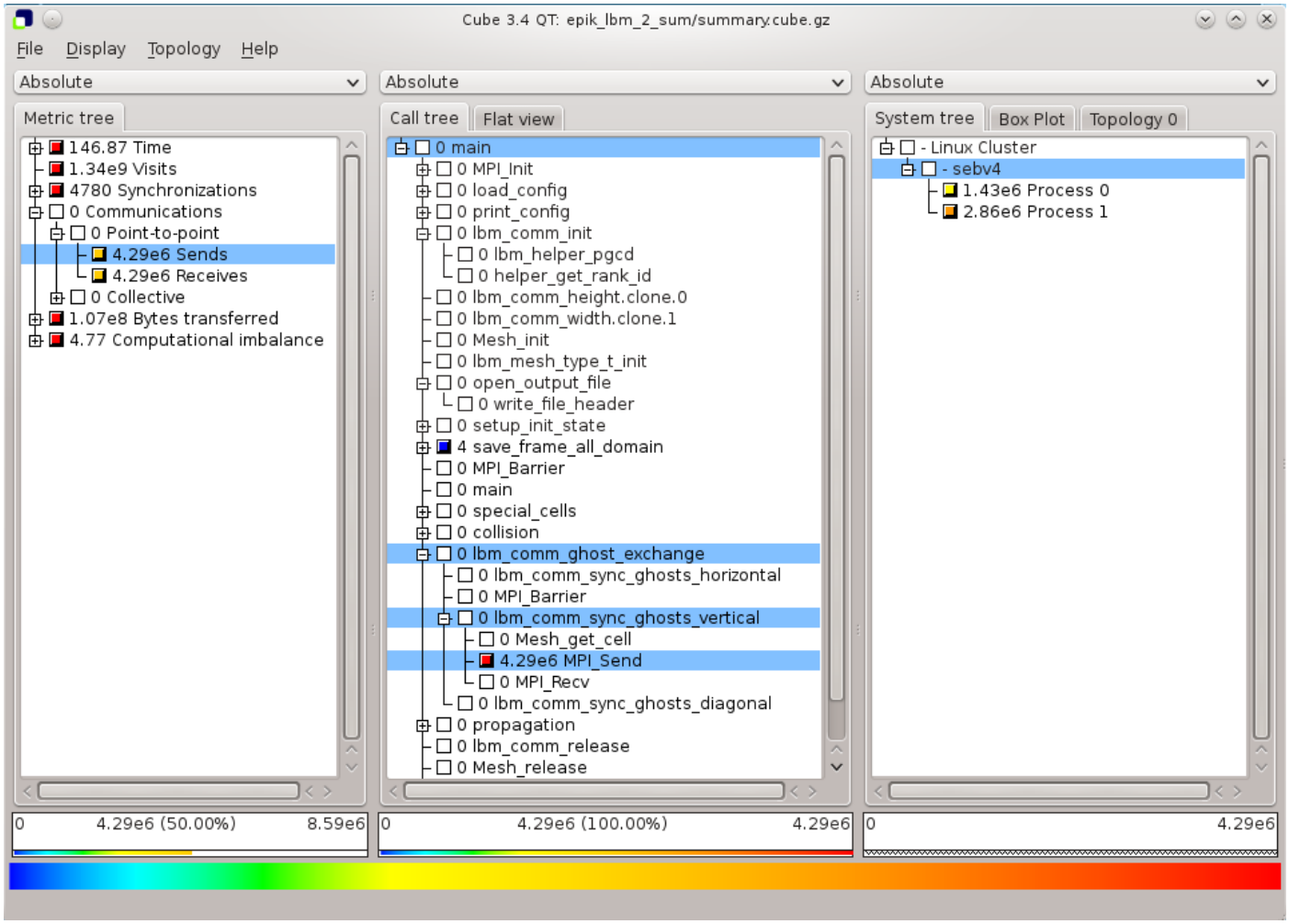


Figure X : Profile Scalasca pour un problème 800 x 160 – 200 sur deux processus MPI.

On peut faire quelques observations :

1. De manière relativement logique, les communications se concentrent sur les deux fonctions *save_frame_all_domain*() et *lbm_comm_ghost_exchange()*. Pour les 200 pas de temps, on a effectué 4 millions de communications ce qui semble totalement déraisonnable.
2. *Scalasca* nous annonce 107 Mo échangés, avec 4 millions de communications on a une moyenne de 100 octets par communication, ce qui est vraiment peu.
3. Côté synchronisation on retrouve près de 4700 barrières réparties entre *main()* et *lbm_ghost_exchange()*, soit près de 20 par pas de temps, ce qui semble à nouveau déraisonnable.

En terme de communication, on a 200 pas de temps, on a coupé en deux le maillage à calculer. Comme chaque nœud du maillage dépend des valeurs des 8 voisins, on peut s'attendre en ordre de grandeur à 10 * 200 * 2 = 4000 communications au grand maximum, avec 4 millions observés il y a visiblement un problème quelque part.

OUTIL : Scalasca est un logiciel libre de profiling MPI téléchargeable à l'adresse : http://www.scalasca.org/.

À partir de ces observations, on va pouvoir commencer à rentrer dans le code pour tenter de comprendre les différents problèmes de performances de l'application. Non visible sur la capture d'écran, Scalasca donne aussi la fraction de temps consommée par les différentes fonctions.

9) Description du schéma de communication

Général :

Nous l'avons vu précédemment, les communications touchent essentiellement les fonctions :

- *main()*
- *lbm_ghost_exchange()*
- *save_frame_all_domain()*

La suite donne le schéma pratique des communications obtenues en lisant le code de ces trois fonctions.

Pour main :

Au niveau de main, les communications sont :

1. Appels à *save_frame_all_domain()*
2. En boucle pour chaque pas de temps
    a. Calcule via *special_cells()* suivie d'un *MPI_Barrier*.
    b. Calcule des collisions suivies d'un *MPI_Barrier*.
    c. Échange des mailles fantômes
    d. Calcule des propagations suivies d'un *MPI_Barrier*.
    e. Appels à save_frame_all_domain()
3. MPI_Barrier avant de fermer le fichier.

On peut déjà remarquer l'utilisation intensive de MPI_Barrier qui reste à justifier.

Échange des mailles fantômes :

Principal point de communication, on trouve la séquence :

1. Échange vers la droite suivie d'une barrière.
2. Échange vers la gauche suivie d'une barrière.
3. Échange vers le bas suivi d'une barrière.
4. Échange vers le haut suivi d'une barrière.
5. Échange vers l'angle haut gauche suivi d'une barrière.
6. Échange vers l'angle bas gauche suivi d'une barrière.
7. Échange vers l'angle haut droit suivi d'une barrière.
8. ~~*Échange vers l'angle bas gauche suivi d'une barrière.*~~
9. Échange vers l'angle bas droit suivi d'une barrière.
10. ~~*Échange vers la gauche suivie d'une barrière.*~~

Même remarque : il reste à montrer que les barrières sont utiles étant donné leur nombre. Autre point, on voit immédiatement apparaitre une duplication inutile de certains échanges (8 et 10). Ces échanges sont tous basés sur des appels synchrones, il faudra donc vérifier la formation éventuelle de chaines de dépendances.

Échanges dans la fonction de sauvegarde :

Pour la sauvegarde, le nœud maitre reçoit tour à tour les portions de maillage des différents nœuds et les écrits tour à tour, on peut noter ainsi sur le nœud maitre :

1. Réception du sous-domaine du nœud 1
2. Écriture du domaine
3. Réception du sous-domaine du nœud 2
4. Écriture du domaine.
5. …

On a une croissance linéaire du temps de sauvegarde avec le nombre de cœurs, cette fonction est donc totalement non scalable. Dans le cadre de ce cours, les étudiants avaient toutefois pour consigne d'ignorer cette fonction, car une modification des IO est rapidement couteuse en temps.

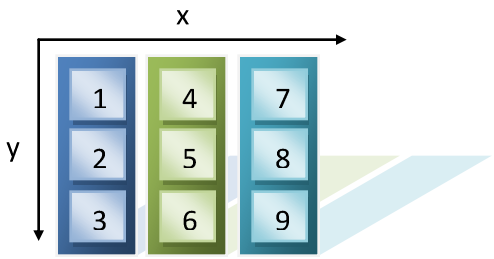
Représentation mémoire du sous-domaine

Lors des échanges on voit à plusieurs reprises l'usage de la fonction *Mesh_get_cell()*, cette dernière définit l'organisation mémoire du maillage avec une représentation contiguë suivant y :

```
static inline lbm_mesh_cell_t Mesh_get_cell( const Mesh *mesh, int x, int y)
{
      return &mesh->cells[ (x * mesh->height + y) * DIRECTIONS ];
}
```

Remarque : Cette fonction toute simple est appelée un très grand nombre de fois puisqu'au cœur de presque toutes les boucles de calculs, on vérifie donc bien qu'elle est définie de façon à pouvoir être inliné. Pour cela, elle doit être implémentée dans un fichier d'en-tête de sorte que le compilateur ait accès au code au moment de l'utilisation. Ici, c'est bien le cas puisqu'elle se trouve dans *lbm_struct.h*.

Ce qui donne sur un schéma :



Le parcours se fait de manière continue selon l'axe Y et discontinue suivant X.

INFO : L'inlining permet de gagner en performance en évitant d'effectuer un appel de fonction si cette dernière est trop courte pour en valoir la peine. Dans ce cas, le compilateur remplacera directement l'appel par le code de la fonction. Ceci permet de gagner en performance tout en maintenant un code propre et découpé. Attention toutefois à ne pas tomber dans le piège du tout inliné, ceci peut être contre-productif

INFO : Pour les performances, la représentation mémoire des données est souvent la clé, il est donc très important d'avoir en tête la représentation choisie pour bien comprendre les problèmes de performances d'une application.

10) Étape 1 : Réduction du nombre de Send/Recv

Dans la section précédente, nous avons vu que l'application effectuait un nombre trop important de petites communications, notamment au sein de la fonction *lbm_comm_sync_vertical*, si on se rend dans les sources on trouve :

```
case COMM_SEND:
      for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
            for ( k = 0 ; k < DIRECTIONS ; k++)
                  MPI_Send( &Mesh_get_cell(mesh_to_process, x, y)[k], 1, MPI_DOUBLE,…);
      break;
case COMM_RECV:
      for ( x = 1 ; x < mesh_to_process->width - 2 ; x++)
            for ( k = 0 ; k < DIRECTIONS ; k++)
                  MPI_Recv( &Mesh_get_cell(mesh_to_process, x, y)[k], 1,MPI_DOUBLE,…);
      break;
```

Ici on remarque effectivement une communication pour chaque élément (un double) du tableau *mesh_to_process*. Il serait préférable de tous les envoyer en un coup ce qui réduira grandement le nombre de communications.

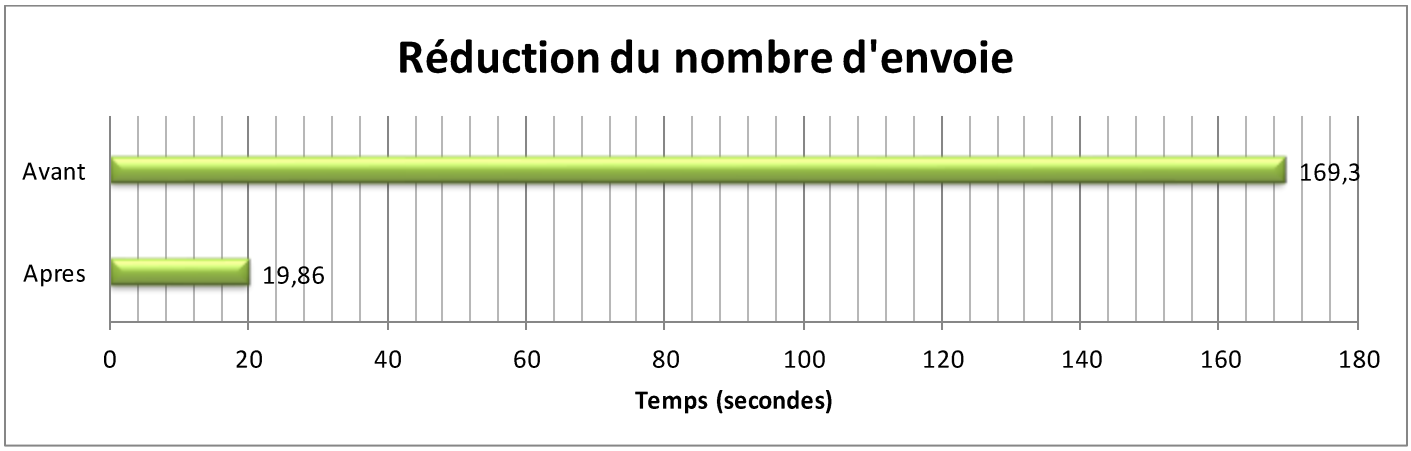
Dans *lbm_comm_sync_horizontal()* on transmet une bande complète le long de Y. Comme cette dernière est contiguë en mémoire, elle peut être utilisée directement, le code précédent devient donc :

```
case COMM_SEND:
      MPI_Send( Mesh_get_cell( mesh_to_process, x, 1 ), (mesh->height-2) * DIRECTIONS,
                               MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
      break;
case COMM_RECV:
      MPI_Recv( Mesh_get_cell( mesh_to_process, x, 1 ), (mesh->height-2) * DIRECTIONS,
                                MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD,&status);
      break;
```

Pour lbm_comm_sync_vertical() les données à transmettre ne sont plus contigües on doit donc passer par un buffer temporaire, on introduit dans le code les fonctions *helper_copy_line_to_buffer()* et *helper_copy_line_from_buffer()* pour obtenir :

```
case COMM_SEND:
      helper_copy_line_to_buffer(mesh_to_process,mesh->buffer,y);
      MPI_Send( mesh->buffer, (mesh->width-2) * DIRECTIONS, MPI_DOUBLE, …);
      break;
case COMM_RECV:
      MPI_Recv( mesh->buffer, (mesh->width-2) * DIRECTIONS, MPI_DOUBLE, …);
      helper_copy_line_from_buffer(mesh_to_process,mesh->buffer,y);
      break;
```

Avec ces modifications, sur un problème 6400 x 2560 – 200 sur 128 cœurs on obtient les performances :



SPEED-UP : 8.5 pour 128 processus et pour un problème 6400 x 2560 - 200.

11) Suppression des échanges dupliqués

Nous l'avons vu lors de la description du schéma de communication, deux des communications sont dédoublées sans aucune raison, nous pouvons donc les supprimer sans risque.

On supprime donc ligne 303 de *lbm_comm.c* :

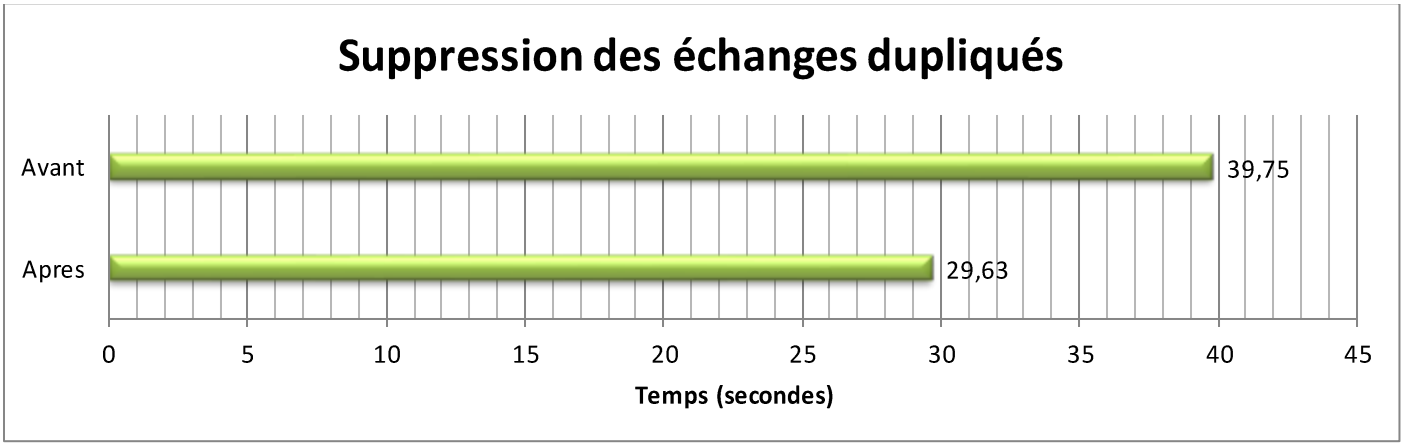
```
//bottom left
lbm_comm_sync_ghosts_diagonal(mesh,mesh_to_process,COMM_SEND,
      mesh->corner_id[CORNER_BOTTOM_LEFT],1,mesh->height - 2);
```

```
lbm_comm_sync_ghosts_diagonal(mesh,mesh_to_process,COMM_RECV,
      mesh->corner_id[CORNER_TOP_RIGHT],mesh->width - 1,0);

//prevend comm mixing to avoid bugs
MPI_Barrier(MPI_COMM_WORLD);
```

Et ligne 327 :

```
//prevend comm mixing to avoid bugs
MPI_Barrier(MPI_COMM_WORLD);

// Right to left phase : on reçois en haut et on envoir depuis le bas
lbm_comm_sync_ghosts_vertical(mesh,mesh_to_process,COMM_SEND,mesh->top_id,1);
lbm_comm_sync_ghosts_vertical(mesh,mesh_to_process,COMM_RECV,mesh->bottom_id,mesh-
>height - 1);
```

Pour les performances, on obtient sur un problème 6400 x 2560 - 400 :



Remarque, par manque de temps, la mesure "avant" est donnée en extrapolant le temps précédent, considérant une évolution linéaire du temps de calcul en fonction du nombre de pas de temps.

SPEED-UP : 1.3 pour 128 processus et pour un problème 6400 x 2560 - 400.

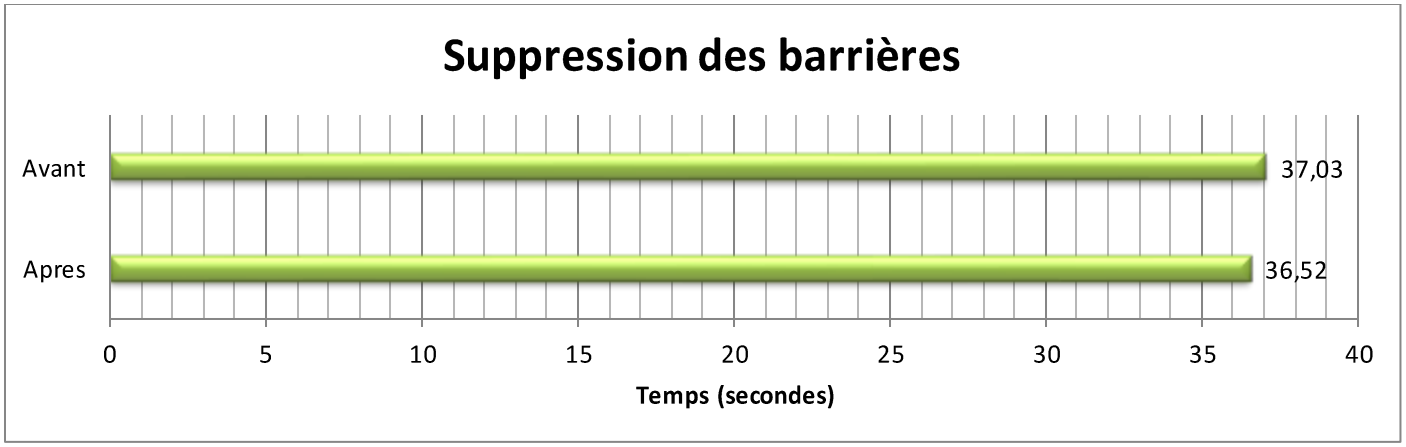
12) Suppression des barrières

Lors de l'analyse avec Scalasca ou la lecture du code, nous avons remarqué la présence de nombreuses barrières MPI qui n'ont aucune justification. Dans main, certaines sont placées entre des fonctions de calcules qui n'impliquent aucune communication. Il n'y a donc aucune raison d'attendre si on utilise pas de segment de mémoire partagé (ce qui est notre cas). Entre les communications, de même, une fois les données reçues depuis un voisin, il n'y a aucune raison d'attendre les autres communications avant de pouvoir effectuer les calcules locaux.

Quand au commentaire, si une barrière permet d'éviter un bogue c'est d'abord et avant tout parce que l'application est mal codée et qu'elle ne respecte pas un standard, s'il est justifié c'est qu'il y a avant tout quelque chose à corriger.

```
//prevend comm mixing to avoid bugs
MPI_Barrier(MPI_COMM_WORLD);
```

En terme de temps, sur un problème 6400 x 2560 - 500 on obtient :



Remarque, par manque de temps, la mesure "avant" est donnée en extrapolant le temps précédent, considérant une évolution linéaire du temps de calcul en fonction du nombre de pas de temps.

Ici on remarque un gain très faible, dans certaines situations il peut également arriver que la suppression des barrières réduise les performances. Toutefois, ces dernières deviennent un réel endicape pour une utilisation à plus large échelle il convient donc de les éviter autant que possible.

SPEED-UP : 1.01 pour 128 processus et pour un problème 6400 x 2560 - 500.

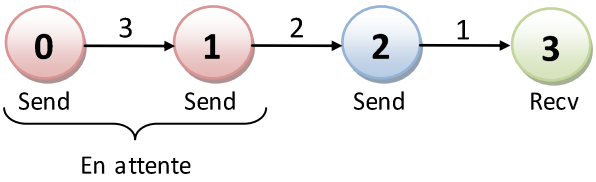
13) Communication pair-impair

Lors de la lecture du code, on remarque que les fonctions de communications utilisées sont toutes synchrones (MPI_Send, MPI_Revc). Dans ce cas, chaque étape d'échange doit attendre qu'une communication se termine avant d'entamer la suivante.

D'autre part, on remarque dans *lbm_comm_ghost_exchange()* que tous les processus effectuent leur communication dans le même ordre. Or on sait que ce type d'échange introduit une chaine de dépendance. En effet, tel quel on a par exemple pour le premier échange :

1. Tous les processus attendent sur MPI_Send() sauf le dernier.
2. Le dernier effectue son MPI_Recv() et débloque le processus associé.
3. Ce processus va à son tour débloquer son voisin
4. Et ainsi de suite.

Problème, la chaine de dépendance se rallonge quand on ajoute des processus, au bout d'un moment les temps d'attente deviennent un problème. Le processus 0 lui a une tendance à attendre N fois le temps d'une communication, N étant le nombre de processus sur une direction donnée. On rappelle qu'un supercalculateur actuel (début 2012), N peut à l'extrême être de l'ordre de 100 000, voire 500 000.

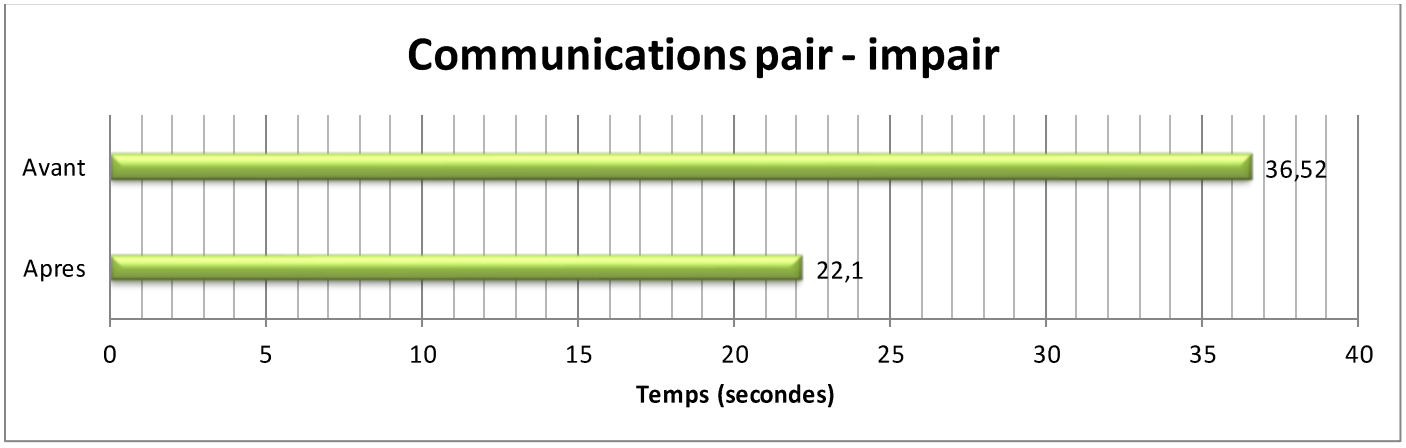


On corrige le problème en modifiant *lbm_comm_ghost_exchange()* de sorte à avoir :

```
if (rank % 2 == 0)
{
      //communications avec SEND, puis RECV
      ....
```

```
} else {
      //communications avec RECV, puis SEND
      ....
}
```

D'un point de vue performance, toujours sur 128 cœurs avec un problème 6400x2560 - 500 on obtient :



Remarque, par manque de temps, la mesure "avant" est donnée en extrapolant le temps précédent, considérant une évolution linéaire du temps de calcul en fonction du nombre de pas de temps.

On remarque un gain important avec cette modification et il ne faut pas oublier que les bénéfices vont ici de manière croissante avec le nombre de cœurs utilisés. A 100 000 cœurs, une telle erreur peut réduire drastiquement les performances de l'application.

SPEED-UP : 1.6 pour 128 processus et pour un problème 6400 x 2560 - 500.

14) Correction d'ordre des boucles (optimisation séquentielle)

A ce niveau on peut remarquer avec Scalasca que l'on passe beaucoup de temps dans les fonctions de calcul de *lbm_phys.c*. En y regardant de plus près, on remarque que certaines effectuent les doubles boucles sous la forme :

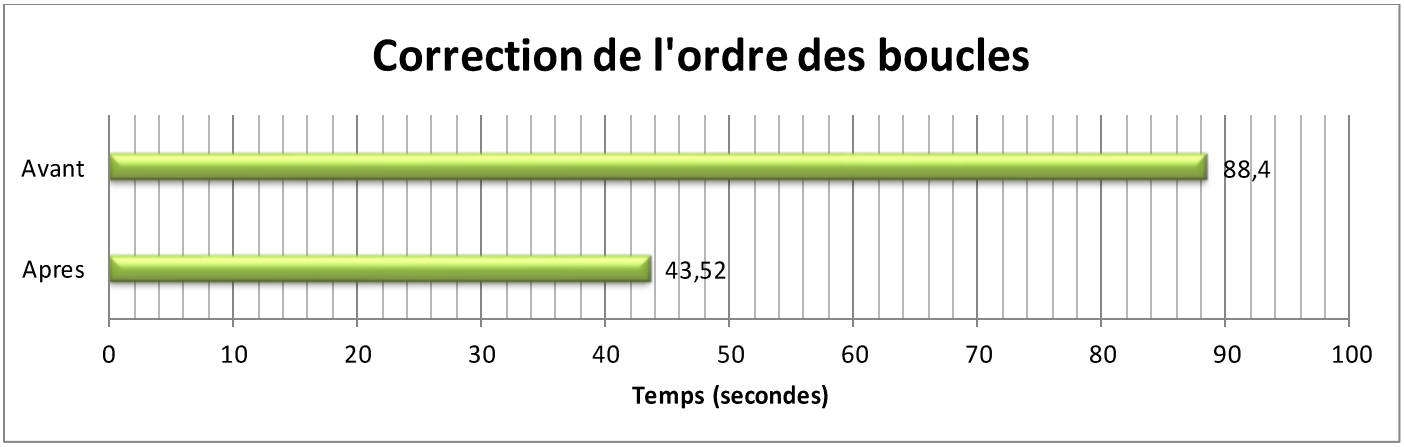
```
for( j = 1 ; j < mesh_in->height - 1 ; j++)
           for( i = 1 ; i < mesh_in->width - 1 ; i++ )
                 .... (Mesh_get_cell(mesh_out, i, j)....
```

Mais parfois comme :

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

Comme toutes ces fonctions manipulent le même tableau, il y en a forcément qui effectuent des accès mémoire non contigus. On rappelle que précédemment nous avons vu que *Mesh_get_cell()* organisait les données de manière contiguë suivant Y, donc suivant le second paramètre de la fonction. Pour nos boucles il s'agit de j. Pour tirer tout le potentiel du processeur il est donc largement préférable d'effectuer la boucle suivant *j* à l'intérieur de la boucle sur *i*. On doit donc retenir la deuxième solution. Toutes les boucles de ce fichier doivent donc être corrigées pour suivre ce pattern d'accès (inversion des deux lignes for).

On obtient les performances sur un problème 6400x2560 - 2000 toujours sur 128 cœurs :



SPEED-UP : 2 pour 128 processus et pour un problème 6400 x 2560 - 2000.

INFO : Attention aux accès mémoires, ils sont très importants pour la performance, inverser la priorité d'accès génère une mauvaise utilisation des caches et empêche la vectorisation. Dans ce genre de cas simple, l'inversion des boucles permet de gagner beaucoup de temps sans effort.

15) Boucles OpenMP, première tentative

Au vu des modifications précédentes, on passe un certain temps dans des boucles de calcules relativement simple, on peut donc se demander s'il ne serait pas possible de les paralléliser en mode thread en utilisant OpenMP pour éliminer les communications MPI intranœud. On vérifie avant tout qu'il n'y ai pas de dépendance inter-itération et pas d'utilisation de variables globales. Dans notre cas, rien de bloquant pour une telle parallélisation.

Prenons comme exemple la fonction *special_cells()* ligne 283 dans *lbm_phys.c*. On ajoute avant la boucle for la plus externe :

```
int i,j;
#pragma omp parallel for private (i,j)
for( i = 1 ; i < mesh->width - 1 ; i++ )
            for( j = 1 ; j < mesh->height - 1 ; j++)
```

On prend soin de bien vérifier toutes les variables de la fonction qui doivent être rendues privées en les marquant avec la directive *private()*. La même procédure s'applique aux autres fonctions de calcul et d'initialisation.

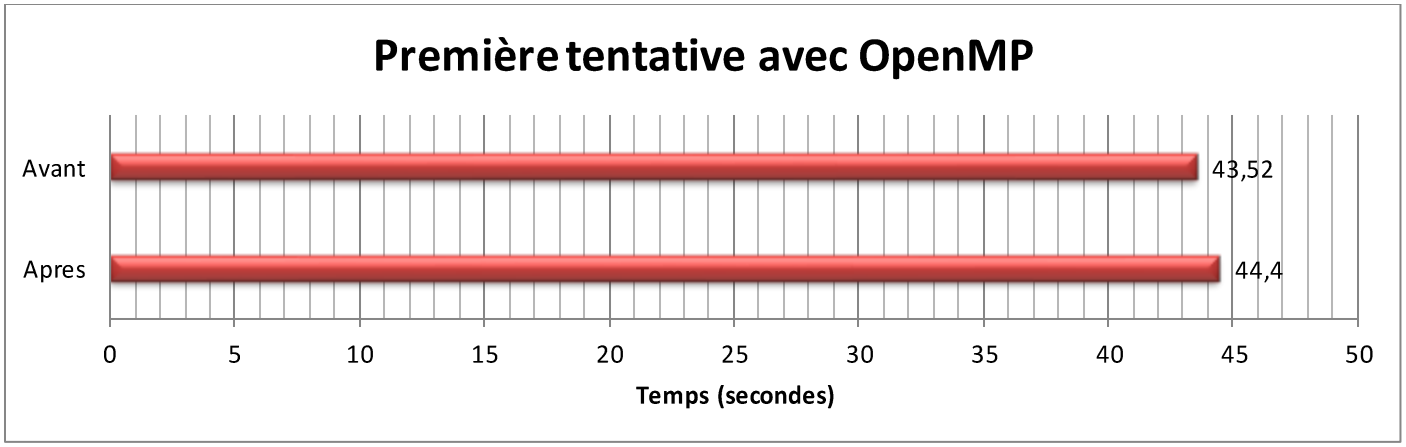
Pour la compilation on doit modifier le fichier *Makefile* de la façon suivante :

```
#flags
CFLAGS=-Wall -DNDEBUG -O3 -march=core2 -march=core2 -msse4.2 -fopenmp
LDFLAGS=-lm -fopenmp
```

En terme de lancement sur le cluster, on procède maintenant de la manière suivante pour utiliser les 8 cœurs de chaque nœud via OpenMP (pour 128 cœurs) :

```
OMP_NUM_THREADS=8 salloc -c 8 -n 16 -N 16 ./lbm config.txt
```

En terme de performance a obtient sur 6400x2560 - 2000 toujours sur 128 cœurs :



Ici on a perdu en performance, on ne retiendra donc pas cette modification pour les tests qui suivent. Les tests OpenMP seront poussés plus loin par la suite. On voit que mélanger OpenMP et MPI n'est pas forcément trivial.

16) Communication asynchrone.

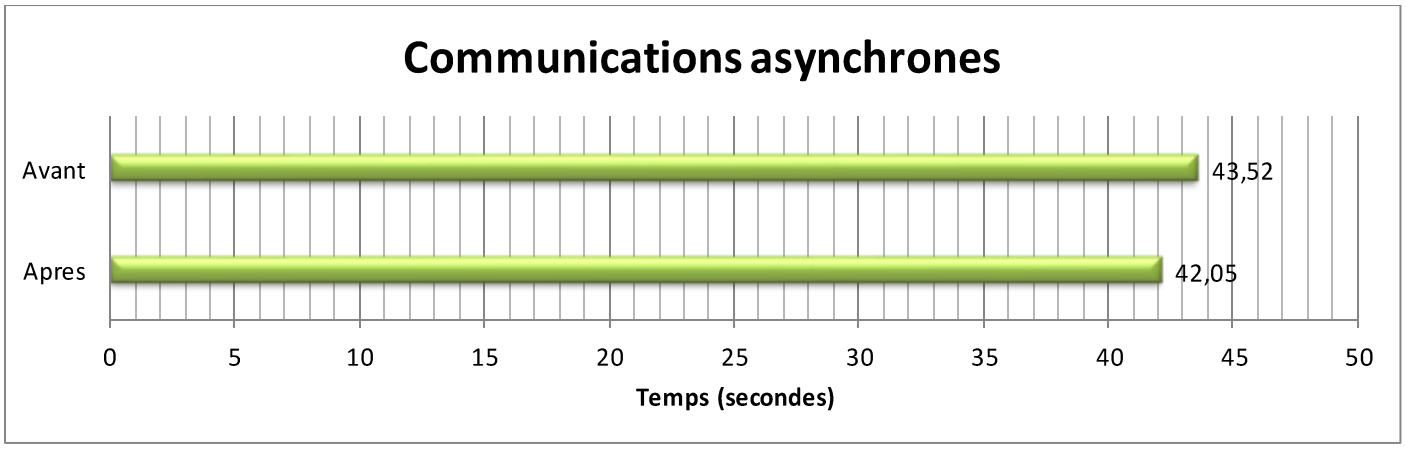
En ce qui concerne les communications, nous avons dans un premier temps supprimé les chaines de dépendance dans les communications. Mais nous pouvons allez plus loin en remplaçant les communications synchrones par des communications asynchrones. En gros, remplacer des *MPI_Send()* par *MPI_ISend()* et *MPI_Recv()* par *MPI_IRevc()*. Dans un premier temps, nous enverrons toutes les communications pour les attendre en bloc via un *MPI_Waitall()*.

Cette optimisation demande quelques modifications du code, car les buffers de copie temporaires de *lbm_comm_sync_ghosts_vertical()* doivent être différents pour les mailles fantômes du haut et du bas du sous-domaine. On doit donc allouer deux buffers séparés dans *lbm_comm_init()*.

Un buffer doit également être ajouté pour agréger tous les *MPI_Request_t* à fournir lors de l'appel à *MPI_Waitall()*.

L'avantage de cette modification est de permettre de lancer les 8 communications en même temps au lieu de les faires une à une ce qui peut laisser plus de marge de manœuvre au runtime pour l'ordonnancement de ces communications.

En terme de performance, toujours sur le problème 6400x2560 - 2000 sur 128 cœurs (sans OpenMP), on obtient :



Le gain ici est mineur, nous avons déjà levé le gros du problème avec les communications pair-impair. En tout cas pour le nombre de processeurs utilisé pour les tests.

17) Communication asynchrone et recouvrement

Pour l'instant la fonction de communication se contente de lancer tous les *MPI_ISend()* et *MPI_IRecv()* pour les attendre en bloc immédiatement après. Il serait préférable de réaliser des calculs pendant que les communications sont en cours avant de les attendre.

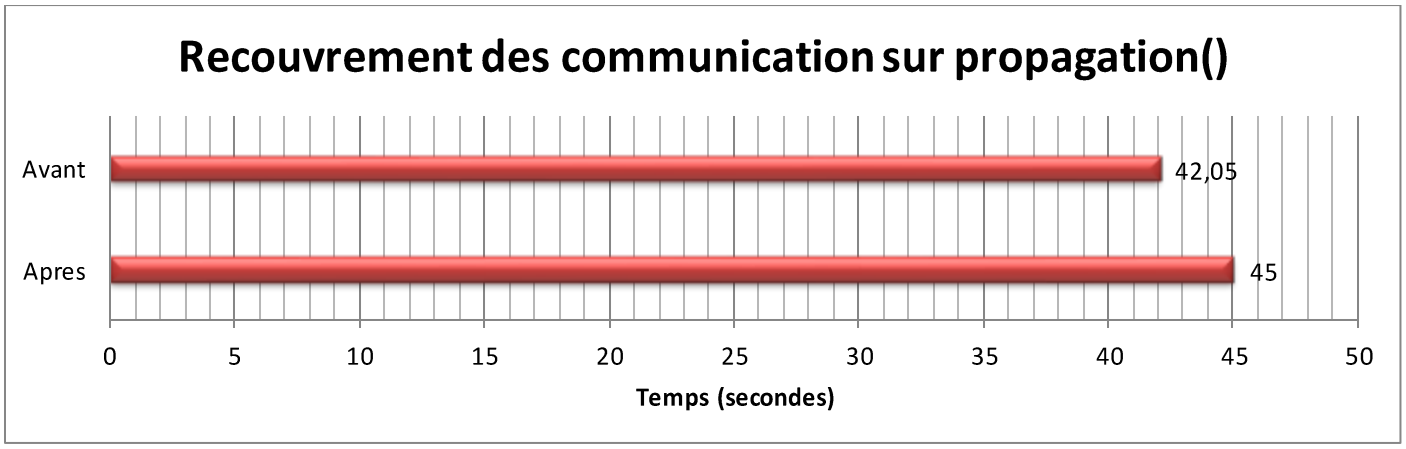
On remarque que la fonction d'envoi est suivie en terme de calcul par la fonction *propagation()*. Dans cette dernière, nous pouvons effectuer tous les calculs sauf ceux des mailles directement en contact avec les mailles fantômes, on peut donc couper cette fonction en deux :

1. propagation_inner() pour effectuer les parties centrales indépendantes des mailles fantômes.
2. propagation_outter() pour effectuer les calcule sur les bords une fois que les mailles fantômes ont été reçues.

On modifie donc la fonction main de façon à avoir :

```
propagation( &mesh, &temp);
propagation_inner(&mesh,&temp);
lbm_comm_ghost_wait_finish( &mesh_comm, &temp );
propagation_border(&mesh,&temp);
```

En terme de performance toujours sur le problème 6400x2560 - 2000 sur 128 cœurs (sans OpenMP), on obtient :



Ici on perd en performance, ce qui est encore actuellement hélas souvent le cas. En effet les communications MPI ont besoin du processeur pour faire progresser les communications, si nous le monopolisons pour du calcul nous ralentissons ces dernières. Ce problème se règle sur certains calculateurs tels que les Blue Gene qui réservent un ou des cœurs pour les communications.

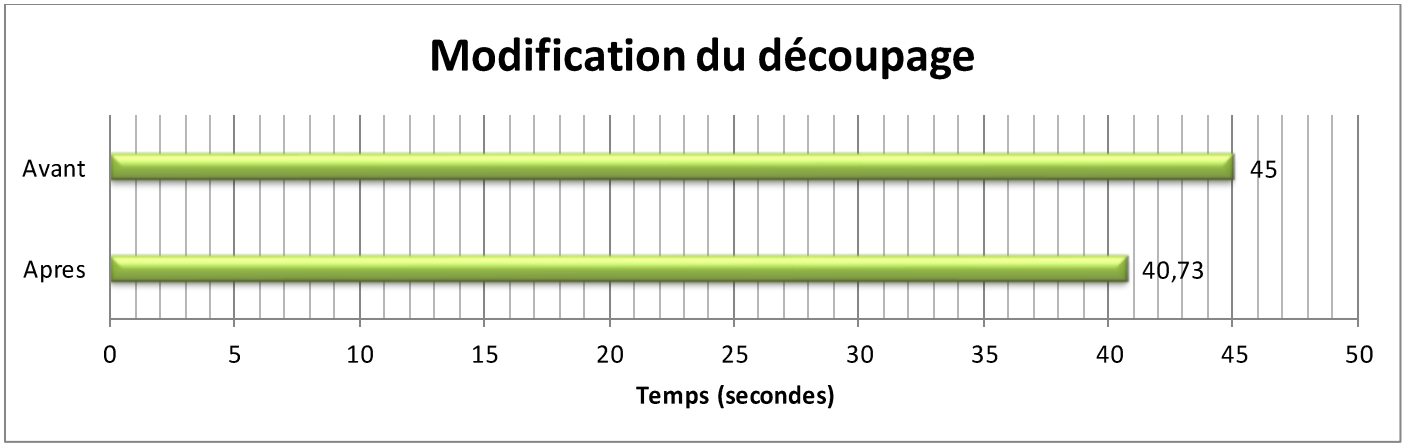
18) Modification du découpage

En regardant de plus près le découpage des communications on se rend compte qu'elle la répartition des sous-domaines se fait en priorité suivante Y, hors un tel découpage favorise l'utilisation de *lbm_comm_sync_ghosts_vertical()*, c'est-à-dire les communications qui nécessitent un buffer intermédiaire. On peut remarquer qu'il est préférable d'utiliser un découpage selon X pour réduire ces copies.

Au niveau du code, on modifie dans lbm_comm_init() :

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

En terme de performance toujours sur le problème 6400x2560 - 2000 sur 128 cœurs (sans OpenMP), on obtient :

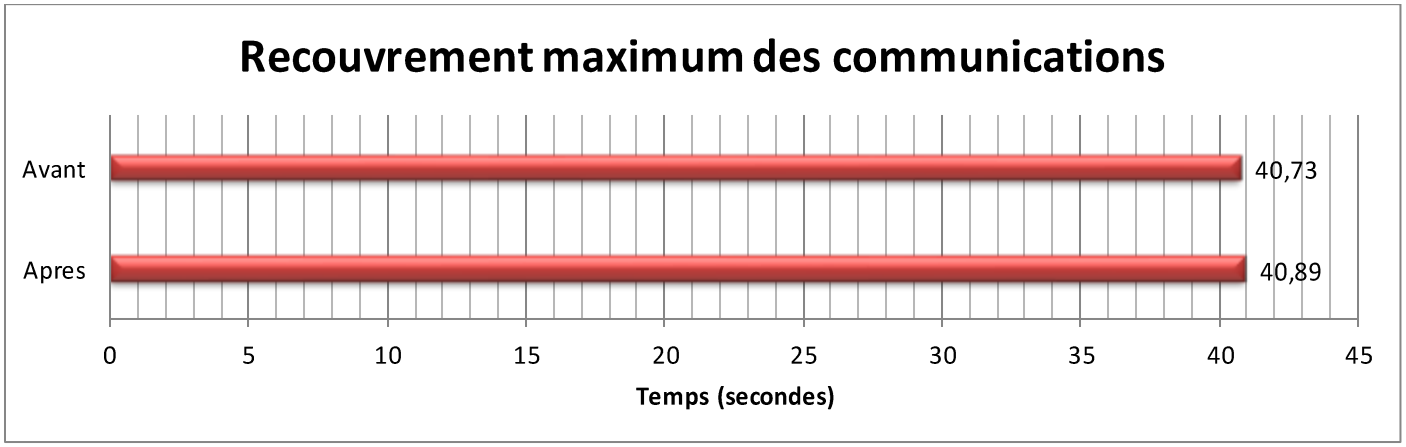


19) Recouvrement maximum des communications asynchrones

Précédemment, nous avions mis en place du recouvrement des communications par la fonction *propagation()*, en pratique il s'avère que nous pouvons appliquer la même procédure sur *collision()* et *special_cells()*. Les modifications sont faites de sorte que la boucle principale dans *main()* devienne :

```
special_cells( &mesh, &mesh_type, &mesh_comm);
special_cells_border( &mesh, &mesh_type, &mesh_comm);
collision( &temp, &mesh);
collision_border( &temp, &mesh);
lbm_comm_ghost_exchange( &mesh_comm, &temp );
special_cells_inner( &mesh, &mesh_type, &mesh_comm);
collision_inner( &temp, &mesh);
```

On obtient alors les performances :



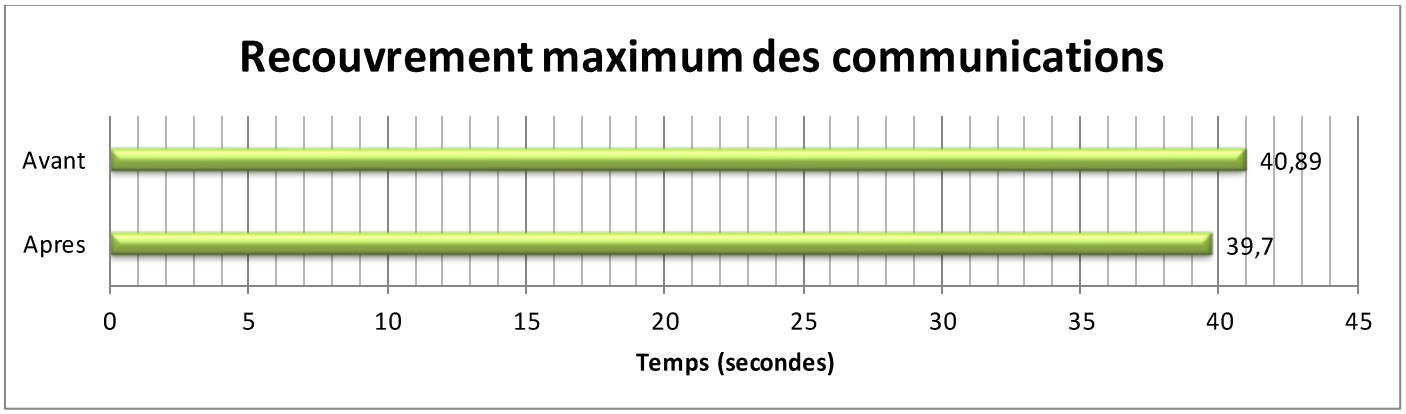
Encore une fois pas de gains significatif pour le recouvrement. Il faudrait tester avec MPC pour voir. En tout cas, la dégradation sur 128 cœurs n'est pas très importante et il est possible que disposer de ce recouvrement soit intéressant pour un nombre plus important de cœurs. On garde donc cette modification.

20) OpenMP, une seule section parallèle.

Lors de la première tentative d'utilisation d'OpenMP, nous avions ajouté des sections parallèles au niveau des boucles. Or en faisant cela, nous forçons des synchronisations a la fin de chacune des fonctions concernée puisque nous entrons, sortons, en permanence des sections parallèles.

Une amélioration consiste à remonter l'utilisation de *#pragma omp parallel* au niveau du main, et ne n'utiliser plus que des *#pragma omp* for dans les fonctions appelées. Ceci impose de protéger les sections séquentielles de main. On protège donc les sections critiquent par des couples *#pragma omp barrier* et *#pragma omp single*.

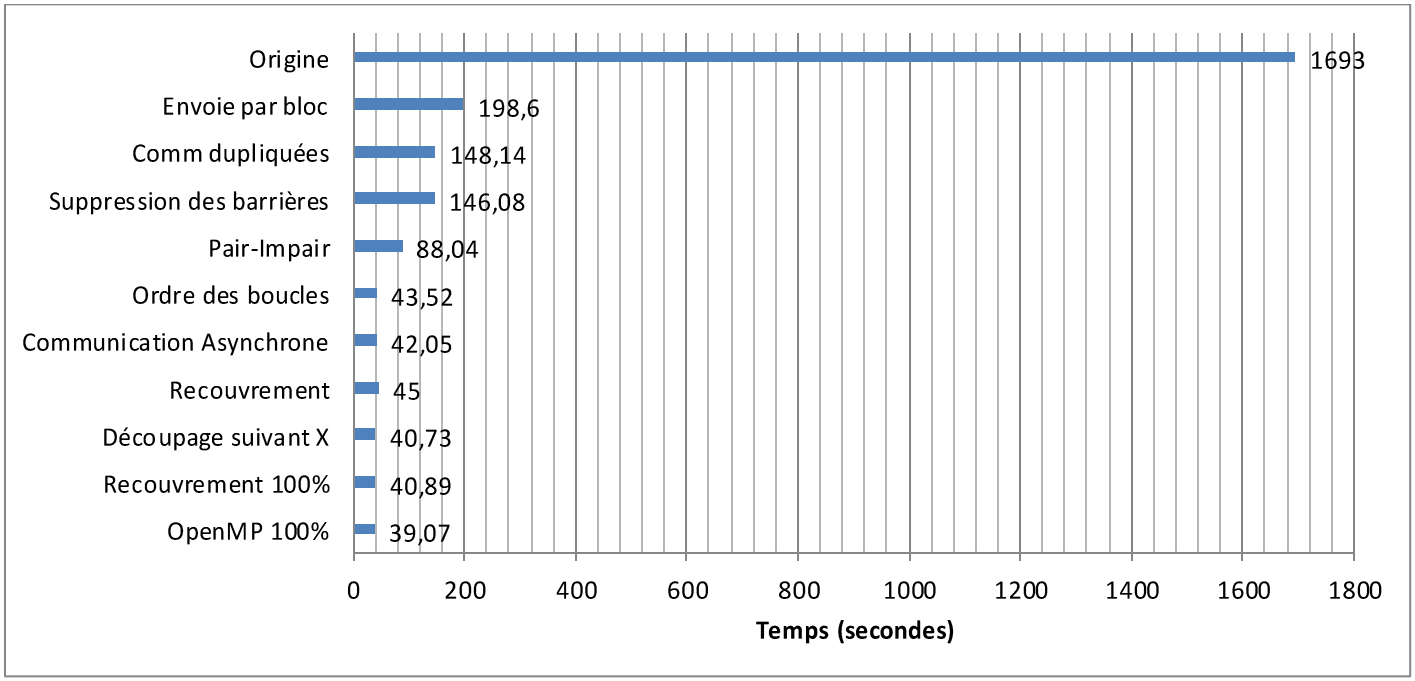
On obtient alors les performances :



[À corriger : boulette dans l'implémentation, une des barrières est inutile avant le wait, à confirmer].

21) Résumé des optimisations

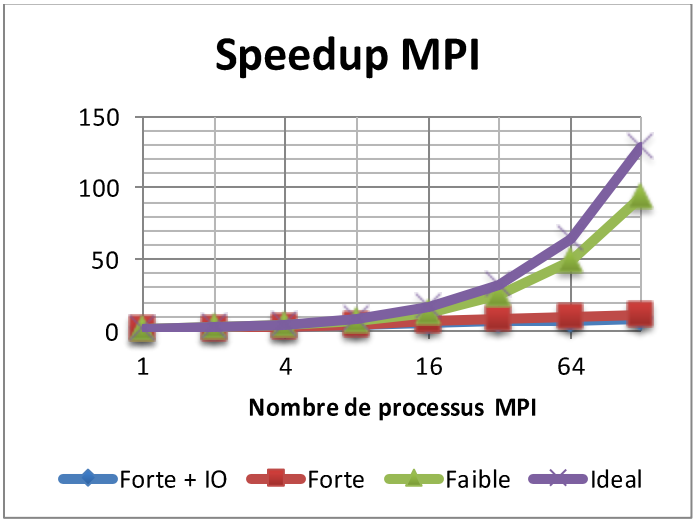
Si on rassemble toutes les optimisations sur un même graphique on obtient pour un problème 6400x2560 - 2000 sur 128 cœurs :

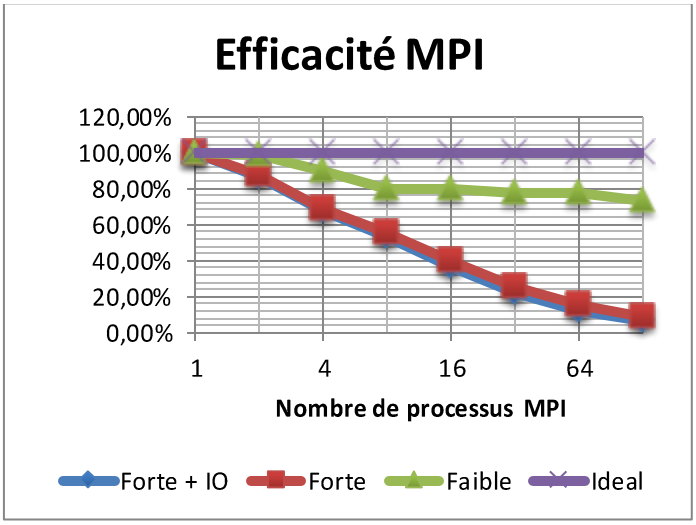


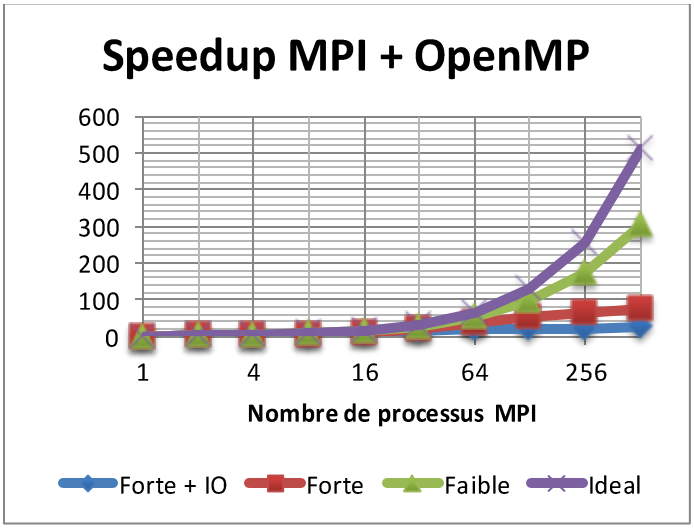
Pour ce problème, le speedup total est de **43** auquel il faudrait également ajouter le facteur lié à l'utilisation de O2/O3 comparé au mode O0 utilisé par défaut avec GCC.

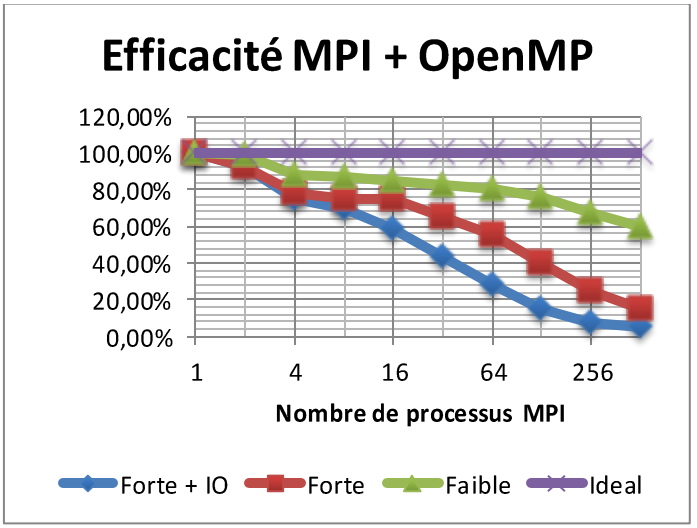
22) Scalabilité finale.

Au final, on peut obtenir en faisant la même analyse qu'au départ. On rappel qu'au départ, on avait une efficacité de 20% au mieux sur 16 cœurs :









23) Quelques règles de conduite

Au vu de l'analyse précédente, on peut donner quelques règles de conduite pour l'optimisation d'une application :

1. Vérifier que l'application fonctionne, on n'optimise pas tant que ça ne marche pas.
2. Si on ne l'a pas écrite soit même, commencer par se familiariser avec l'application.
3. Faire attention aux paramètres de compilation.
4. Choisir une taille de problème et évaluer les performances (scalabilité…)
5. Lorsque c'est possible, ne pas hésiter à utiliser des outils pour repérer les problèmes de performances.
6. Bien regarder ce que le code fait plutôt que ce que la documentation dit qu'il fait.
7. Faire attention aux dépendances avec les communications bloquantes.
8. Attention à la représentation mémoire et aux accès aux données.
9. Traquer les communications globales (barrières, réduction…) et les réduire au strict minimum.
10. Même si on a pas forcément gagné dans notre exemple, il peut être bénéfique de fournir des possibilités de recouvrement des communications.
11. Bien évaluer l'impact de chaque optimisation, même si les gains dépendent de l'ordre dans lequel on les applique.

En terme de modification du code, il est certainement préférable d'insister sur le fait de garder un code maintenable, quitte à ne pas appliquer certaines optimisations conduisant à un code trop illisible.