Université Versailles-Saint-Quentin

Paris-Saclay

Spécialité :
Calcul Haute Performance, Simulation

Module :
Technique d'optimisation de la parallélisation

Rapport de Projet :
Seismic core - 3D stencil optimization

Présenté par :
- Sofiane ARHAB.

Encadré par :
- Gabriel Dos Santos.
- Hugo Taboada.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

1 Introduction

Ce projet a pour objectif l'optimisation d'un code de calcul réalisant un stencil de 16^e ordre à 49 points et 3 axes représentant un coeur sismique. Le stencil est une itération de Jacobi utilisant 3 matrices de 3^e ordre (Matrices 3D) stockant des valeurs à virgule flottante sur 8 octet.

Le but sera d'utiliser toutes (lorsque cela est possibles) les techniques d'optimisation vu en cours afin de rendre le code aussi rapide que possible sans aucune restriction, un script de comparaison de la vitesse d'exécution avec la version de base et de la vérification de la validité des résultats est fournie par notre encadrant, ce script sera utilisé a chaque tentative d'optimisation pour garantir l'intégrité des résultats à chaque étape.

Toutes les modifications apportées au code seront versionnisé dans ce dépôt Github[1], ce rapport sera également disponible sur la plate-forme collaborative Overleaf[2].

Enfin l'utilisation des outils vu en TDs sera particulièrement important dans le cadre de ce projet, ceux-ci représentent le nerf de la guerre, on retrouve les outils de débbuguage GDB et Valgrind, ainsi que les outils de profiling tel que Perf, Gprof et MAQAO.

2 Présentation du problème

Comme on le disait le code jouet fourni calcule un stencil d'ordre 16, 49 points, à 3 axes, comme dans la figure 1. Les dimensions du stencil sont définis au runtime, via un fichier de configuration (par défaut config.txt). Le calcul du stencil est une itération de Jacobi qui utilise trois tensors d'ordre 3 stockant des valeurs à virgule flottante double précision.

• A est le tensor d'entrée, initialisé avec une valeur de 1,0 pour les cellules centrales et une valeur de 0,0 pour les cellules fantômes.

• B est un tensor d'entrée constant, initialisé comme suit :

$$B_{x,y,z} = sin(z \times cos(x + 0.311) \times cos(y + 0.817) + 0.613)$$
$$\forall x \in [0, dim_x), \forall y \in [0, dim_y), \forall z \in [0, dim_z)$$

• C est le tensor de sortie, avec tous ses coefficients initialisés à 0,0.

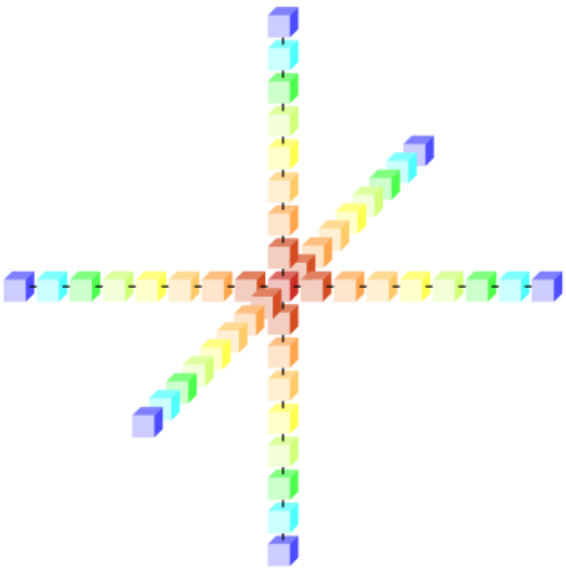


FIGURE 1.1 – Représentation visuelle du stencil

3 Architecture cible

Le hardware joue un rôle important dans l'optimisation des performances d'un code de calcul ainsi que la version des compilateurs utilisés et les différente implémentations, pour ce projet nous nous baserons sur cette architecture cible :

Modèle	Coeurs	Fréquence(GHz)	L1(KiB)	L2(MiB)	L3(MiB)	Cache line (o)
AMD Ryzen 7 6800H	8	1.6 - 4.7	256	4	16	64

TABLE 1.1 – Architecture cible (x86_64)

Nous utiliserons les version des compilateurs et de la librairie MPI suivant :

Compilateur	Version
GCC	11.4.0
MPI	mpich-3.4.1

TABLE 1.2 – Versions compilateur et MPI

4 Protocole de mesure

- Brancher le laptop à la prise d'alimentation.
- Fermer toutes application tierces, les autre processus indésirable et couper le réseau.
- Tuer l'interface graphique `Ctrl + ALt + F3` (tty3).
- Définir le Gouverneur du CPU en mode "performance" avec la commande `sudo cpupower frequency-set -g performance`.
- Tout au long du rapport, les résultats présenter seront sur un problème de dimensions `100x100x100`, sauf si précisé autre.
- Les optimisations seront cumulatives , chaque nouvelle optimisation se base sur la précédente.

— la métrique utilisé pour mesurer les performance sera le temps d'exécution en secondes, on se baseras sur le script python qui nous a été fourni afin d'évaluer le speedup générale toujours par rapport à la version de base, les temps d'exécution présenté dans les tableaux seront toujours la médiane de 10 itérations.

— On testeras les flags d'optimisation suivant `-O2 -O3 -Ofast`

— `-O2` : en plus des flags d'optimisation de `-O1` , effectue toutes les optimisation qui ne requiert pas de compromis espace/vitesse.

— `-O3` : en plus des flags d'optimisation de `-O2` , effectue plusieurs autre optimisations sur les boucles en les déroulant, les intervertissant etc..

— `-Ofast` : en plus des flags d'optimisation de `-O3`, applique des optimisation mathématique brutale comme réordonner les opérations flottantes ce qui peut s'avérer plus rapide mais ne respecte pas le standard `IEE-754`, notamment le flag `-ffast-maths` qui effectue des approximation sur les flottant pour aller plus vite, ce qui provoque une perte de précision.

5 Présentation des outils

Comme dis en introduction pour optimiser efficacement, et justifier chaque décision on se base sur les outils qui sont mis a notre disposition, durant ce projet on se baseras surtout sur les suivant :

— **GDB** [3] : GNU Debugger, également appelé GDB, est le débogueur standard du projet GNU. Il est portable sur de nombreux systèmes type Unix et fonctionne pour plusieurs langages de programmation, comme le C.

— **Valgrind** [4] : Valgrind est un framework d'instrumentation pour créer des outils d'analyse dynamique. Il existe des outils Valgrind qui peuvent détecter automatiquement de nombreux bugs de gestion de la mémoire et de threading, et profiler les programmes en détail.

— **Gprof** [5] : Outil de profillage de GNU, gprof permet de déterminer quelles parties d'un programme prennent le plus de temps d'exécution.

— **Perf** [6] : perf est un outil puissant il peut instrumenter les compteurs de performances du processeur, les points de trace, les kprobes et les uprobes (traçage dynamique). Il est capable d'effectuer un profilage léger. Il est également inclus dans le noyau Linux, sous tools/perf, et est fréquemment mis à jour et amélioré.

— **MAQAO** [7] : MAQAO (Modular Assembly Quality Analyser and Optimizer) est un framework d'analyse et d'optimisation des performances fonctionnant au niveau binaire en mettant l'accent sur les performances de base. Son objectif principal est de guider les développeurs d'applications tout au long du processus d'optimisation grâce à des rapports synthétiques et des astuces.

— **Strace** [8] : C'est un outil de débogage sous Linux pour surveiller les appels système utilisés par un programme, et tous les signaux qu'il reçoit.

Chapitre 2

1 Débbuguage

Avant de se lancer dans l'optimisation, nous devons débugguer le code , en effet celui-ci nous a été fournie avec plusieurs bugs par défaut et il nous reviens de les corriger et d'expliquer la démarche entreprise pour identifier et corriger chaque erreur.

1. Inclusion de la bibliothèque MPI : la première erreur rencontrée lors de la première tentative de compilation est que le programme ne trouve pas le fichier d'en-tête de la bibliothèque MPI :

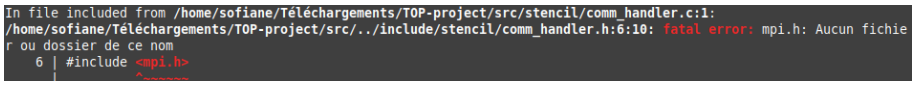


FIGURE 2.1 – Erreur bibliothèque MPI

   On met à jour le `CMakeLists.txt` interne en rajoutant l'instruction CMake `include_directories(${MPI_INCLUDE_PATH})` et en ajoutant cette variable dans l'instruction `target_include_directories()`

2. Erreurs d'allocations : La bibliothèque MPI étant reconnu, la compilation s'effectue sans souci, à l'exécution on a tout d'abord un warning concernant le chemin du fichier de config, on rectifie ce dernier dans le `main.c`, puis on un message d'erreur sur la sortie d'erreur standard `stderr`, indiquant que l'allocation de mesh de dim X a échouer, on lance alors `GDB` en recompilant avec les symboles de debug on met un breakpoint au niveau de la fonction d'allocation `mesh_new` puis on `run`, arrivé à la fonction on exécute instruction par instruction, on print l'adresse de `cells` pour vérifier que celle-ci n'est pas `NULL` et que l'allocation a réussi, puis on arrive à la condition `(NULL != cells)` qui vaut 1 et qui explique l'arrêt du programme,

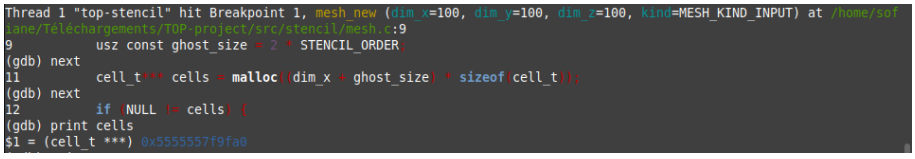


FIGURE 2.2 – Condition erroné

   on corrige donc cette condition, on relance l'erreur disparaît mais cette fois on a un segfault, gdb indique que l'erreur est survenue au moment d'accéder à l'adresse `mesh->cells[i][j][k].kind` l'adresse printé ne semble pas être valide

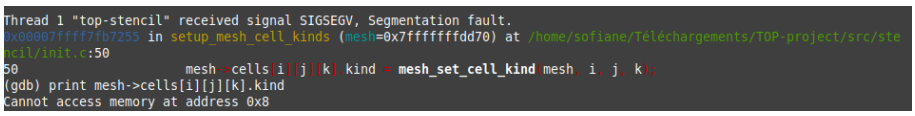


FIGURE 2.3 – Segmentation Fault

on retourne donc à notre fonction d'allocation et constate le problème à la 3^e boucle d'allocation , l'appel à malloc est commented_out et remplacé par `NULL` on décommente l'appel à la malloc et on corrige la condition similaire à l'étape précédente.

3. Variable non initialisé : Le code compile et s'exécute normalement il est cependant très lent, on passe le code sur valgrind afin de repérer d'autre éventuelle problèmes

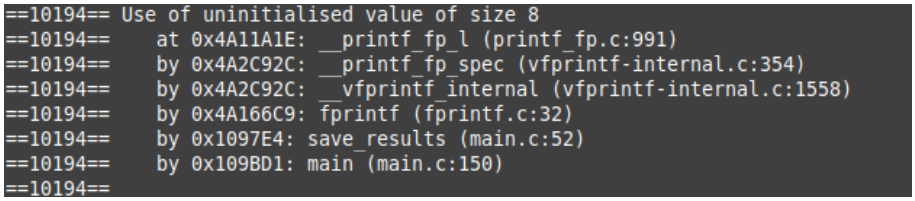


FIGURE 2.4 – Variable non initialisé dans fprintf

Valgrind nous informe a plusieurs reprise de l'utilisation d'une varaiable non initialisé de taille 8 byte dans l'appel `fprintf` de la fonction `save_results`, on remarque également que dans la sortie toute une colonne de donnée vaut 0, on identifie cette variable comme étant `glob_elapsed_s`, qui est le buffer de réception d'une communication collective MPI, et on se rend compte du problème l'appel à `Allreduce()` n'est pas effectué par tous les processus a cause d'une condition, on supprime alors cette condition et le code semble s'exécuter sans bugs à présent, on peux passer à l'optimisation.

2 Scalabilité

Nous allons à présent mesurer la scalabilité forte et faible de notre programme corrigé, à noté que nous avons effectué ces mesures avec la première optimisation de la section suivante car celle-ci corrige un ralentissage artificiel du code qui n'est pas représentatif de la réele capacité de calcul de celui-ci, nous allons commencé par évalué la forte scalabilité, autrement dis la capacité du code a réduire son temps d'exécution par deux a mesure que l'on double le nombre de processus MPI, à noté que l'on a eu recours au flag `--oversubscribe` pour 16 processus pour un problème de taille `100x100x100`.

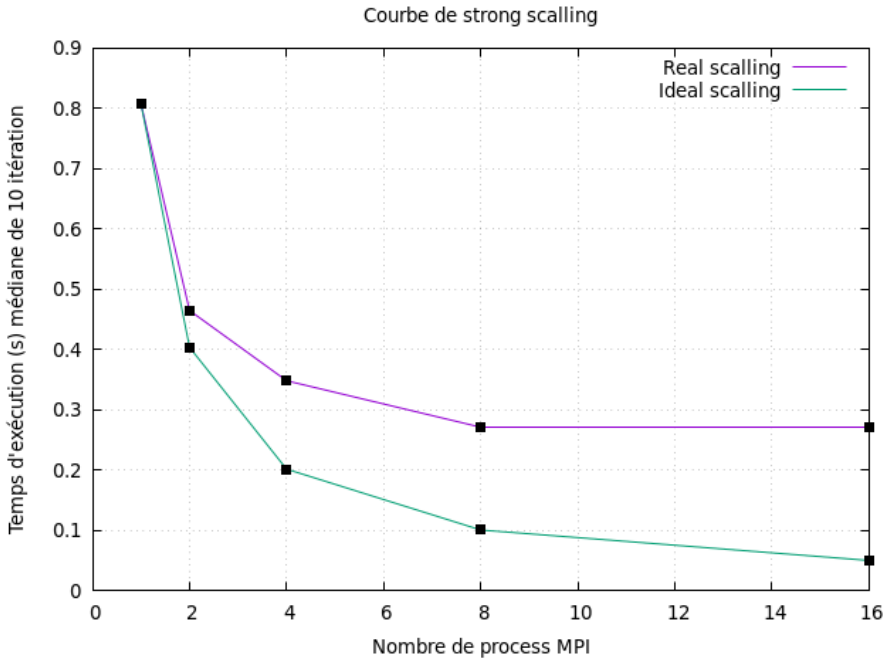


FIGURE 2.5 – Strong scalling

On voit que le programme scale assez mal et qu'il en dessous de ce qu'un scalling ideal serait pour ce cas. Nous allons à présent évaluer le scalabilité faible du programme , en d'autre terme sa capacité a maintenir un temps d'exécution constant a mesure que l'on double les dimensions du problème mais également le nombre de processus.

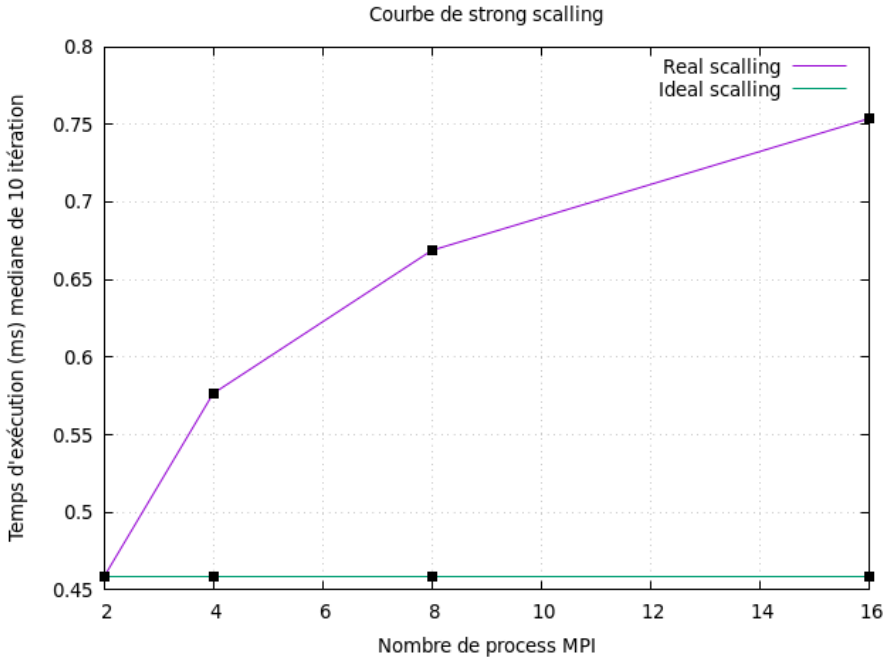


FIGURE 2.6 – Weak scalling

La aussi on constate que le programme scale mal, on prend le double de temps en passant de 1 à 2 processus et 5 fois plus de temps en passant de 2 à 4.

3 Optimisations séquentiel

Maintenant que nous avons corriger tout les bugs apparent , on peux passer à l'optimisation du code, cette étape passeras d'abord par la relecture attentive de ce dernier et l'utilisation des outils de profilages

3.1 Appel sleep caché

Dans le fichier d'en-tête `mesh.h` notre attention est attiré vers une macro bizarre `__builtin_sync_proc(_)` qui fait appelle à `catof(p,l,e,a,s,e)(1)` qui est une autre macro définie dans `types.h` comme `catof(a, b, c, d, e, f) e##b##c##c##a`, on comprend que cette macro ré-ordonne les lettres qui lui sont passé en paramètre selon un schéma donné , ce qui a pour résultat de transformer `please` en `sleep`, et de lancer l'appel système `sleep(1)` et de passé 1 seconde à attendre à chaque itération , le tout camouflé dans une fonction `MPI_Syncall_callback` qui est a son tour appelé dans `comm_handler_ghost_exchange`.

```
// Need to synchronize all remaining in-flight communications before exiting
// BS !
// MPI_Syncall(MPI_COMM_WORLD);
```

Figure 2.7 – Appel sleep caché

Pour confirmer tout cela, on utilise l'outil de diagnostique `strace` pour monitorer les appel système à sleep , cela se fait à l'aide de la commande
`strace -e trace=clock_nanosleep ./top-stencil` .

```
RANK 0:
  COORDS:     0,0,0
  LOCAL DIMS: 100,100,100
      -   -
   -  *   -
   -  -
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
****************************************
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffdd6c9f1b0) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=0, tv_nsec=100000}, NULL) = 0
+++ exited with 0 +++
sofiane@sofiane-ASUS-TUF-Gaming-A15-FA507RR-TUF507RR:~/TOP_PROJECT_GRP/TOP-project/build$
```

Figure 2.8 – Sortie de strace sur le programme

On constate effectivement, plusieurs appel à `clock_nanosleep` durant l'exécution du programme. On retire donc l'appel de fonction caché.

3.2 Cache Blocking

Le calcul de stencil est une opération pour laquelle le cache blocking peut aider a améliorer la localité spatiale et temporel, tout d'abord nous allons déterminer de taux de cache miss avec
`perf stat -e cache-references,cache-misses,L1-dcache-load-misses,L1-dcache-load`
pour la version de base corrigé du code

```
Performance counter stats for './top-stencil':

     2 578 798 452      cache-references
     1 575 853 564      cache-misses              #   61,108 % of all cache refs
     1 002 483 230      L1-dcache-load-misses     #    3,00% of all L1-dcache accesses
    33 381 070 075      L1-dcache-load
```

FIGURE 2.9 – Taux de cache miss générale et du L1

On constate un taux de miss assez élevé sur l'ensemble des caches, et `3%` lorsque l'on s'intéresse au L1, uniquement, pour implémenter notre version cache-blocké on va s'aider de la petite macro suivante `#define min(a, b) ((a) < (b) ? (a) : (b))`retournant le minimum de deux valeurs, on utilise une macro pour éviter les appels de fonctions supplémentaire même si le compilateur aurait très certainement 'inliné' la fonction, l'intérêt de cette macro minimum sera d'éviter de devoir retoucher aux indices à l'intérieur de la boucle interne.

On passe donc à l'implémention de la version cache-blocking de la fonction `solve_jacobi`, on utilsera un bloc de dimension 3, dont les facteur de blocking seront pour le cas 100x100x100 `X=32 , Y=32 , Z=32`, ces facteurs ont été déterminé empiriquement (try and retry) et ne sont sûrement pas les plus adaptés, le choix du/des facteurs de blocking étant un sujet de recherche a part entière, nous allons pas tenter de trouver les meilleurs valeurs possible ici.

```
Performance counter stats for './top-stencil':

     2 403 559 556      cache-references
     1 317 957 067      cache-misses              #   54,834 % of all cache refs
       969 686 594      L1-dcache-load-misses     #    2,91% of all L1-dcache accesses
    33 335 974 880      L1-dcache-load
```

FIGURE 2.10 – Taux de cache miss générale et du L1 avec cache-blocking

On constate néanmoins une réduction du taux de miss générale, et une très faible réduction du taux de miss du cache L1.

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.572648916	64.81
-O3	0.546772728	72.06
-Ofast	0.102260642	816.25

TABLE 2.1 – Résultats - Cache Blocking

3.3 Passage en Structure of Array

Etant donné que dans le calcul du stencil au sein de la fonction `solve_jacobi()` on accède uniquement au champ `f64 value` de la struct `cell_s`, il serait bénéfique d'avoir toutes les `value` contiguë en mémoire, pour ce faire on doit changer la struct qui de base est une Array of structure en une Structure of Array. En passant le code sur l'outil lprof de maqao, ce dernier nous incite également a essayer de passer de AOS en SOA. On modifie donc la struct `cell_s` et la struct `cell_t` tel que :

```c
typedef struct cell_s {
    f64 value;
    cell_kind_t kind;
} cell_t;
typedef struct mesh_s {
    usz dim_x;
    usz dim_y;
    usz dim_z;
    cell_t*** cells;
    mesh_kind_t kind;
} mesh_t;
```

```c
typedef struct cell_s {
    f64* value;
    cell_kind_t* kind;
} cell_t;
typedef struct mesh_s {
    usz dim_x;
    usz dim_y;
    usz dim_z;
    cell_t cells;
    mesh_kind_t kind;
} mesh_t;
```

On se débarasse du triple pointeur sur `cells` et on passe a un pointeur 1D , le passage en SOA nécessite également d'adapter toutes les fonctions pour prendre en compte la nouvelle méthode d'accès aux données, nous ne détaillerons pas toutes les modifications apporté a chaque fonction.

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.528911403	78.01
-O3	0.527922318	78.30
-Ofast	0.07173216	1210.86

TABLE 2.2 – Résultats - Structure of Array

Légère amélioration sur l'ensemble des flags , le passage en SOA est plus impactant aux niveau des accés mémoires, nous allons donc analyser cette aspect.

```
Performance counter stats for './top-stencil':

     2 023 631 136      cache-references
     1 293 653 608      cache-misses              #   63,927 % of all cache refs
       762 760 178      L1-dcache-load-misses     #    3,76% of all L1-dcache accesses
    20 262 549 374      L1-dcache-load

       7,779347785 seconds time elapsed

       5,544896000 seconds user
       0,032767000 seconds sys
```

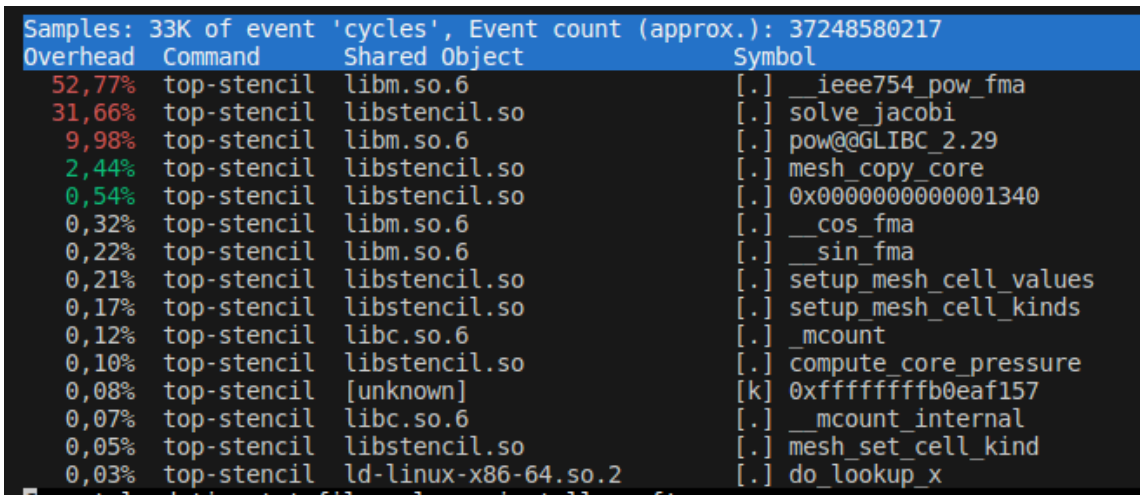
FIGURE 2.11 – Taux de cache miss en SOA.

Bien qu'à première vu le taux de cache-miss semble avoir augmenté au niveau du cache L1 en réalité, le nombre de cache line transféré depuis la RAM a considérablement baissé (d'environ $13 * 10^9$) ce qui sous entend une amélioration de la localité spatiale et le nombre de miss en lui même à également baissé , ce qui sous entend une amélioration de la localité temporelle.

3.4 Remplacement de pow()

3.4.1 Première approche

En analysant le code avec l'outil de profilage `perf`, on apprend que l'on passe plus de la moitié de tout le temps d'exécution dans la fonction `pow()` de la `glibc`, c'est le plus gros hotspot du programme.



```
Samples: 33K of event 'cycles', Event count (approx.): 37248580217
Overhead  Command      Shared Object              Symbol
  52,77%  top-stencil  libm.so.6                 [.] __ieee754_pow_fma
  31,66%  top-stencil  libstencil.so             [.] solve_jacobi
   9,98%  top-stencil  libm.so.6                 [.] pow@@GLIBC_2.29
   2,44%  top-stencil  libstencil.so             [.] mesh_copy_core
   0,54%  top-stencil  libstencil.so             [.] 0x0000000000001340
   0,32%  top-stencil  libm.so.6                 [.] __cos_fma
   0,22%  top-stencil  libm.so.6                 [.] __sin_fma
   0,21%  top-stencil  libstencil.so             [.] setup_mesh_cell_values
   0,17%  top-stencil  libstencil.so             [.] setup_mesh_cell_kinds
   0,12%  top-stencil  libc.so.6                 [.] _mcount
   0,10%  top-stencil  libstencil.so             [.] compute_core_pressure
   0,08%  top-stencil  [unknown]                 [k] 0xffffffffb0eaf157
   0,07%  top-stencil  libc.so.6                 [.] __mcount_internal
   0,05%  top-stencil  libstencil.so             [.] mesh_set_cell_kind
   0,03%  top-stencil  ld-linux-x86-64.so.2      [.] do_lookup_x
```

FIGURE 2.12 – Sortie de perf.

La fonction est appelé plusieurs fois dans la boucle interne de la fonction `solve_jacobi()`, cela est très couteux d'une part car c'est un appel de fonction et d'autre part car l'algorithme utilisé est très générale permettant également de calculer les puissances négatives et factionnaires, ce qui rajoute un surcoût de traitement qui n'est pas adapté a notre cas d'autant plus qu'on ne calcul que les puissance base 17. La première approche pour optimiser cette fonction est d'essayer de changer d'algorithme, on utiliseras l'algorithme d'exponentiation rapide avec manipulation de bit ayant pour base fixe 17 étant donné que c'est l'unique cas traité.

```
double power_of_17(usz exponent) {
if (exponent == 0) return 1.0;
if (exponent == 1) return 17.0;
double result = 1.0;
double base = 17.0;
while (exponent > 0) {
    if (exponent % 2 == 1) { result *= base; }
    base *= base;
    exponent >>= 1;     }
return result;
}
```

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.28027076	235,47
-O3	0.284879562	229.30
-Ofast	0.2760721375	241.02

TABLE 2.3 – Résultats - Exponentiation rapide

On constate un speed up significatif en changeant d'algorithme, on remarque aussi qu'en `-Ofast` les performances se dégradent par rapport à la dernière mesure, les optimisations mathématiques aggresivent du flag `-Ofast` ne pouvant sûrement pas être appliqué entièrement sur notre fonction `powers_of_17`

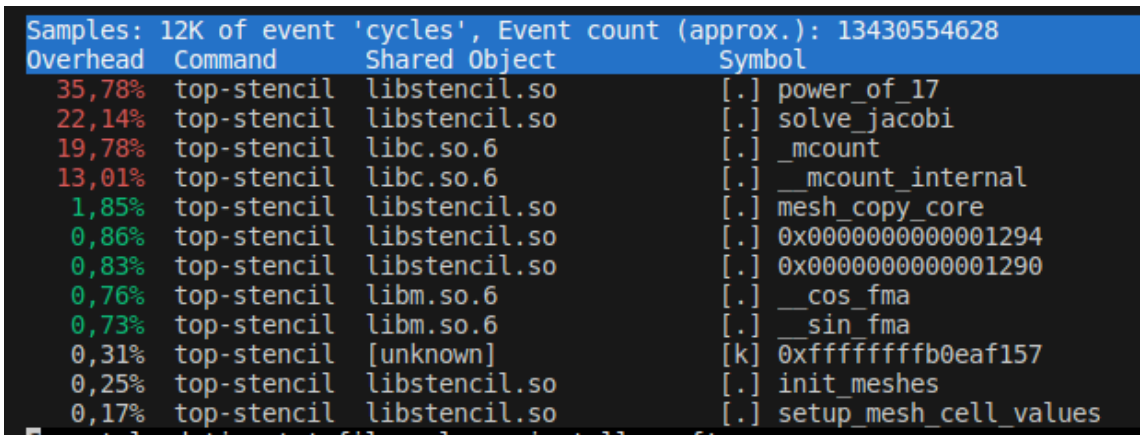


FIGURE 2.13 – Sortie de perf avec remplacement de pow

On passe environ 17% de temps en moins à calculer les puissances.

3.4.2 Seconde approche

La deuxième approche consiste a mettre en place une Look Up table, en effet on remarque que les puissances calculé sont redondantes , on re calcule bien trop souvent une puissance que l'on avait déjà calculé dans les itérations précédente , la mise en place d'une LUT permettras de réduire drastiquement le nombre d'appel et de calculs inutile, on rempliras cette table une fois en utilisant notre fonction précédente.
Dans la fonction `solve_jacobi` on rempli un tableau de taille `STENCIL_ORDER` qui représenteras notre LUT

```
f64 powers[STENCIL_ORDER];
for (usz o = 0; o < STENCIL_ORDER; ++o) {
    powers[o] = power_of_17((f64)o);
}
```

Puis on accéderas a ce tableau pour récupérer les puissance dont on a besoin durant le calcul en accédant à la case `o` en O(1).

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.034087956	2637.89
-O3	0.035626971	2540.17
-Ofast	0.035729909	2517.61

TABLE 2.4 – Résultats - Look Up Table

Notre plus grand speed up jusqu'à présent, on atteint la barre des 2500% sur l'ensemble des flags, même constat pour `-Ofast` a partir de maintenant il n'y auras plus un énorme écart avec les deux niveau d'optimisation précédent.

```
Samples: 3K of event 'cycles', Event count (approx.): 3396784022
Overhead  Command      Shared Object               Symbol
  70,73%  top-stencil  libstencil.so              [.] solve_jacobi
   6,11%  top-stencil  libstencil.so              [.] mesh_copy_core
   3,72%  top-stencil  libm.so.6                  [.] __cos_fma
   2,37%  top-stencil  libm.so.6                  [.] __sin_fma
   1,06%  top-stencil  libc.so.6                  [.] _mcount
   0,97%  top-stencil  libc.so.6                  [.] __mcount_internal
   0,75%  top-stencil  libstencil.so              [.] init_meshes
   0,66%  top-stencil  libstencil.so              [.] setup_mesh_cell_values
   0,40%  orted        [unknown]                  [k] 0xffffffffaa584b42
   0,36%  orted        [unknown]                  [k] 0xffffffffaaaff812
   0,35%  top-stencil  [unknown]                  [k] 0xffffffffaa4af157
   0,31%  top-stencil  libstencil.so              [.] mesh_set_cell_kind
Cannot load tips.txt file, please install perf!
```

FIGURE 2.14 – Sortie de perf avec Look Up Table

En repassant sur perf , on constate que le temps passé dans notre fonction `power_of_17` est devenu si négligeable qu'il n'apparaît plus dans le rapport de profilage de perf, c'est désormais `solve_jacobi()` qui consomme la majeure partie du temps d'exécution.

```
 Performance counter stats for './top-stencil':

       502 593 986      L1-dcache-load-misses     #   22,87% of all L1-dcache accesses
     2 197 927 807      L1-dcache-load
       895 552 187      cache-references
       437 132 851      cache-misses              #   48,812 % of all cache refs

       3,011893835 seconds time elapsed

       0,591003000 seconds user
       0,049594000 seconds sys
```

FIGURE 2.15 – Taux de cache miss avec LUT

En examinant à nouveau nos accès aux caches , on remarque que la aussi on a considérablement réduit le nombre de cache line chargé depuis la RAM vers le L1 on passe de 20 milliard d'accès au L1 à 2.2 milliard seulement soit presque 10 fois moins tout en réduisant les miss de $\approx$ 200 millions, bien que la sortie nous indique 22.87% de miss cela reste moins que lors de la dernière mesure, ce qui démontre la aussi une meilleure localité les données

3.5 Réduction des accès et des calculs dans la fonction solve jacobi

La fonction `solve_jacobi` contient énormément d'accès et de calculs redondant dans sa logique, par la suite nous allons essayer de réorganiser les boucles (I,K,J) vers (I,J,K) pour optimiser les accés mémoires et favoriser la vectorisation, de pré-calculer tout ce qui peut l'être à l'extérieur afin de réduire au maximum les calculs d'utiliser des variables temporaires etc..

```
for (usz o = 1; o <= STENCIL_ORDER; ++o) {
    usz idx_plus_o = (i + o) * dim_yz + j * dim_z + k;
    usz idx_minus_o = (i - o) * dim_yz + j * dim_z + k;
    sum += (A->cells.value[idx_plus_o] * B->cells.value[idx_plus_o] +
    A->cells.value[idx_minus_o] * B->cells.value[idx_minus_o]) * powers[o - 1];

        usz idx_plus_o_dimz = i * dim_yz + (j + o) * dim_z + k;
        usz idx_minus_o_dimz = i * dim_yz + (j - o) * dim_z + k;
        sum += (A->cells.value[idx_plus_o_dimz] * B->cells.value[idx_plus_o_dimz] +
            A->cells.value[idx_minus_o_dimz] * B->cells.value[idx_minus_o_dimz]) * pow
```

```
            usz idx_plus_o_dimyz = i * dim_yz + j * dim_z + (k + o);
            usz idx_minus_o_dimyz = i * dim_yz + j * dim_z + (k - o);
            sum += (A->cells.value[idx_plus_o_dimyz] * B->cells.value[idx_plus_o_dimyz] +
                A->cells.value[idx_minus_o_dimyz] * B->cells.value[idx_minus_o_dimyz]) * p
        }
        C->cells.value[idx] = sum;
```

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.030177494	3018.25
-O3	0.028838588	3160.01
-Ofast	0.031977424	2755.85

TABLE 2.5 – Résultats - Optimisations divers de la fonction solve

On constate un speed up conséquent, uniquement en réarrangeant les boucles, les pattern d'accès mémoire et en précalculant un maximum.

```
Iteration #10/10
 Performance counter stats for './top-stencil':

       151 472 213      L1-dcache-load-misses     #    6,92% of all L1-dcache accesses
     2 190 145 246      L1-dcache-load
       286 614 755      cache-references
       152 669 602      cache-misses              #   53,266 % of all cache refs

       2,891725454 seconds time elapsed

       0,492393000 seconds user
       0,024826000 seconds sys
```

FIGURE 2.16 – Taux de cache miss après optimisation des accès de solve jacobi

Le nombre de ligne de caches rappatrié depuis la RAM vers le L1 reste constant comparé à la dernière mesure , le taux de miss sur le L1 quant à lui a encore diminué , l'échelle étant quasi la même comparé à la dernière mesure on peut dire qu'on a réduit d'environ `16%` le taux de miss, ce qui indique une amélioration de la localité temporelle.

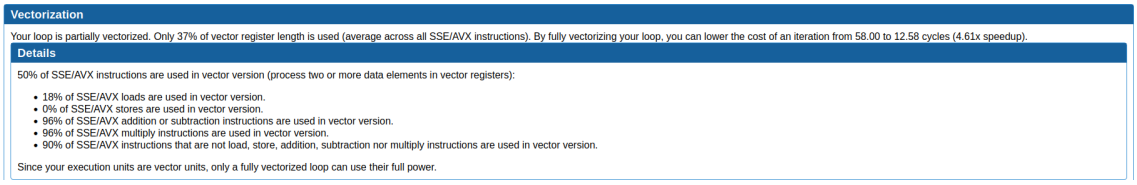


FIGURE 2.17 – Rapport CQA de MAQAO

En utilisant le Code Quality Analyzer de Maqao sur notre fonction solve jacobi, celui-ci nous indique que nos modification ont favoriser la vectorisation partielle de la boucle interne automatiquement par le compilateur , seulement `37%` de la taille des registres SSE (Streaming SIMD Extensions) ont été utilisé, soit les registres `xmm`, MAQAO nous indique également qu'en vectorisant complètement on pourrait atteindre un speedUp de 4.61x , et réduire le coup d'une itération de 58.00 à 12.58 cycles.

4 Optimisations parallèle

Nous allons maintenant tenter d'optimiser d'exécution parallèle du code, a noté que cela ne sous-entend pas que nous avons fais le tour des possibles optimisations séquentielle, ou que cette dernière version séquentielle bénéficient du speed up max théorique. Cependant dans le cadre de ce projet et avec les contraintes de temps dont nous disposons , nous nous contenteront de ce qui a été fait jusqu'à présent pour le coté séquentiel. Tout d'abord nous allons faire le point sur le comportement de notre dernière version en parallèle. Nous allons changer les dimensions du problème,, on passe désormais à du 128x128x128, il est donc nécessaire de refaire les mesure de la dernière version en séquentielle avant.

Flag	Temps d'exécution (s)	Speed-Up (%)
-O2	0.137290811	1186.59
-O3	0.137236529	1186.35
-Ofast	0.11956343	1378.36

TABLE 2.6 – Résultats version précédente - 128x128x128

Nous allons maintenant étudier le comportement en parallèle

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.107835220	1534.46
4	0.209661152	739.86
8	0.238706261	639.10
16	0.319274723	440.18

TABLE 2.7 – Résultats - avec flag -O2

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.107321727	1526.46
4	0.214935148	712.16
8	0.253589427	592.74
16	0.314629808	460.95

TABLE 2.8 – Résultats - avec flag -O3

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.106848104	1554.14
4	0.207585965	747.36
8	0.248481398	606.73
16	0.332747534	432.81

TABLE 2.9 – Résultats - avec flag -Ofast

Etudions la scalabilité,

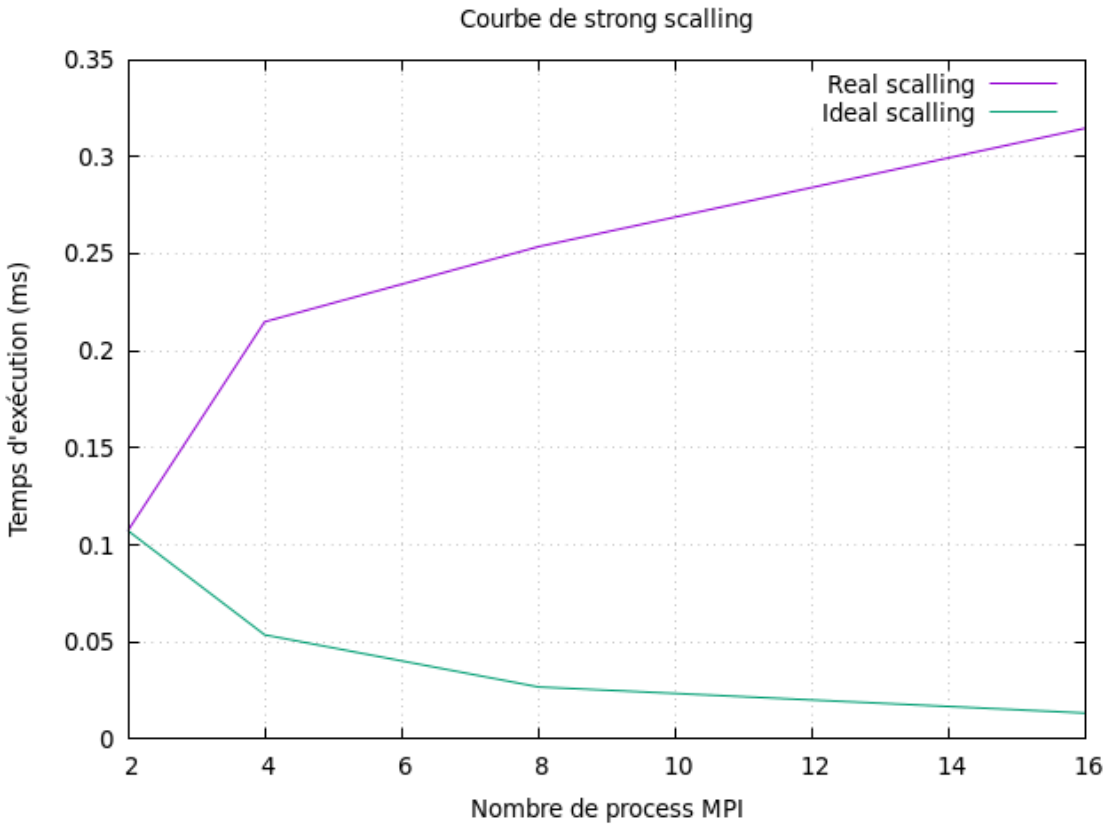


FIGURE 2.18 – Courbe de strong scalling

Cette version du code n'est absolument pas strong scalled au vue de la courbe ci dessus, on arrive même a faire plus lent a mesure que l'on double notre nombre de processus

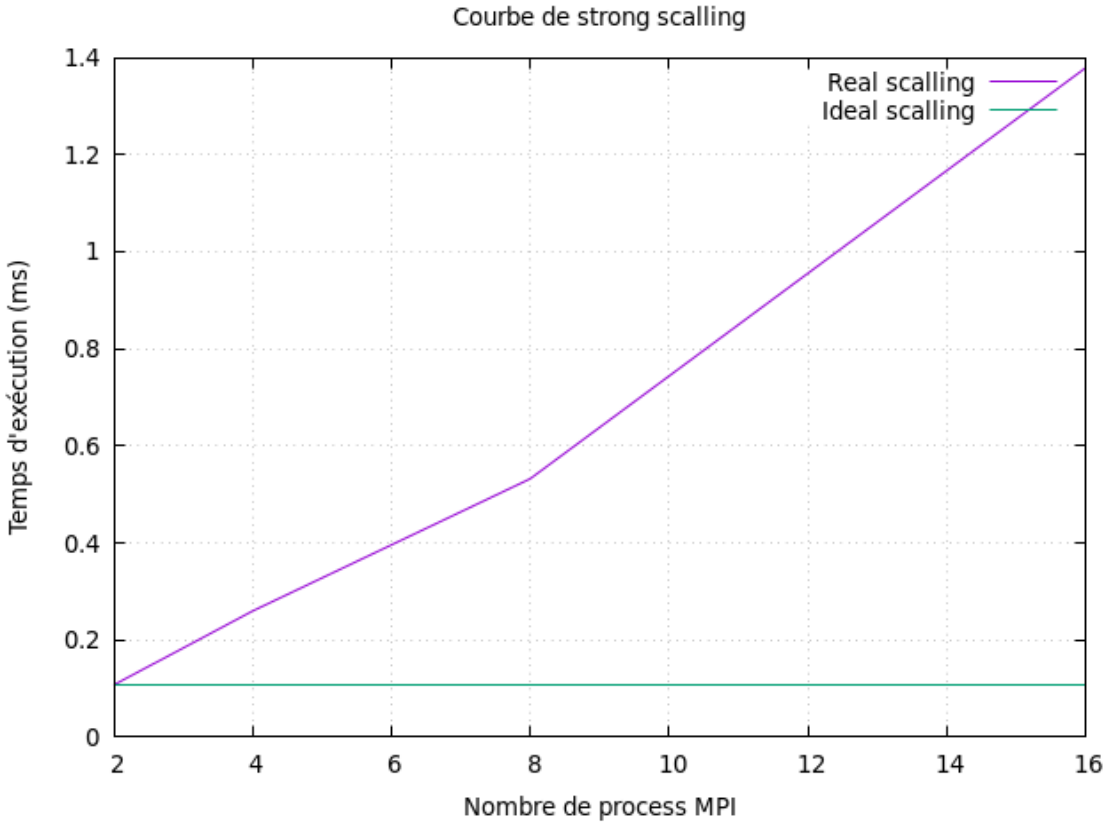


FIGURE 2.19 – Courbe de weak scalling

La aussi on constate que cette version n'est pas du tout weak scalable , on observe le même comportement que lors de notre première étude.

4.0.1 Nettoyage des communications MPI

Notre première optimisation , consisteras a réécrire les routines d'échanges de mailles fantômes, en effet actuellement les échanges se font un élément à la fois ce qui engendre une sur utilisation de communications SEND/RECV , nous allons donc réécrire ces échanges, en faisant en sorte d'envoyer un buffer contenant toutes les mailles en un seul Send/Recv etc e pour les trois fonction `ghost_exchange_front_back()`, `ghost_exchange_left_right()` et `ghost_exchange_top_bottom()`

```
static void ghost_exchange_left_right(
    comm_handler_t const* self, mesh_t* mesh, comm_kind_t comm_kind, i32 target, usz x
) {
    if (target < 0) {
        return;
    }
    int block_size = STENCIL_ORDER * mesh->dim_y * mesh->dim_z;
    usz idx = x_start * mesh->dim_y * mesh->dim_z;
    switch (comm_kind) {
        case COMM_KIND_SEND_OP:
            MPI_Send(
                &mesh->cells.value[idx], block_size, MPI_DOUBLE, target, 0, MPI_COMM_W
            );
            break;
        case COMM_KIND_RECV_OP:
            MPI_Recv(
                &mesh->cells.value[idx],
                block_size,
                MPI_DOUBLE,
                target,
                0,
                MPI_COMM_WORLD,
                MPI_STATUS_IGNORE
            );
            break;
        default:
            __builtin_unreachable();
    }
}
```

Voici comment nous avons modifier `ghost exchange_left_right()`, on calcul l'index du block correspondant aux mailles a envoyé et on envoie ce bloc directement en un seul Send, la même logique a été suivie pour les deux autres fonctions.

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.022692908	4046
4	0.017994636	5044.07
8	Erreur	Erreur

TABLE 2.10 – Résultats - Un seul Send/Recv - 100x100x100

On constate une amélioration d'environ `1000%` chaque fois que l'on double le nombre de processus , la mention erreur pour 8 processus signifie que pour 8 processus et plus les résultats divergent beaucoup trop, après plusieurs expérimentation sur la taille du problème, nous somme arrivée à la conclusion qu'il y'a un problème avec la décomposition actuelle du domaine qui fait que lorsque un processus possèdent un sous-problème de taille différent, autrement dis si la charge de travail n'est pas parfaitement distribué entre les processus, les résultats divergent de la solution de référence.

Voici les résultats pour un problème de tailles `128x128x128` :

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.081223853	2059.31
4	0.081223853	2812.70
8	0.095110138	1800.14
16	0.103290275	1610.46

TABLE 2.11 – Résultats - Un seul Send/Recv - 128x128x128

4.1 Décomposition du domaine

On sait que la décomposition du domaine actuelle n'est pas la plus optimale, car les résultats divergent si la charge de travail n'est pas parfaitement équilibrée, la décomposition est définie dans la fonction le fichier `comm_handler.c`, dans la fonction `comm_handler_new`.

```
comm_handler_t comm_handler_new(u32 rank, u32 comm_size, usz dim_x, usz dim_y, usz

// Compute splitting 100,100,50
u32 const nb_z = gcd(comm_size, (u32)(dim_x * dim_y));
u32 const nb_y = gcd(comm_size / nb_z , (u32)dim_z);
u32 const nb_x = (comm_size / nb_z) / nb_y;

// Compute current rank position
/../

// Setup size
/../

// Setup position
/../

// Compute neighboor nodes IDs
/../
/../
```

Les variables `nb_z nb_y nb_x` représente la taille du sous domaine attribué a chaque processus , la logique actuelle chercheras a découper la taille de la dernière dimensions par 2 jusqu'à atteindre 6 , avant de passer au découpage de la seconde dimension.

```
comm_handler_t comm_handler_new(u32 rank, u32 comm_size, usz dim_x, usz dim_y, usz
// Compute splitting 50,100,100
u32 const nb_x = gcd(comm_size, (u32)(dim_z * dim_y)) ;
u32 const nb_y = gcd(comm_size / nb_x, (u32)dim_x ) ;
u32 const nb_z = (comm_size / nb_x) / nb_y ;
```

Dans notre nouvelle décomposition au lieu de découper systématiquement la dernière dimensions, on le fait sur la première `nb_x`, comme indiqué ci-dessus. Avec cette nouvelle décompositions nous obtenons ces résultas :

Nb. Process	Temps d'exécution (s)	Speed-Up (%)
2	0.016554044	5853.45
4	0.011554243	7918.34
8	Erreur	Erreur

TABLE 2.12 – Résultats - Décomposition en X - 100x100x100

5 OpenMP

Nous allons maintenant essayer de paralléliser certaines de boucles à l'aide de OpenMP, nous avons décider de paralléliser les boucles suivantes :

1. Dans `init.c` au niveau de la fonction `setup_mesh_cell_values()` qui s'occupe de l'initialisation, on emploie également la clause `schedule(static,2` afin de distribuer des chunk de taille 2.

```
#pragma omp parallel for schedule(static,2)
for (usz i = 0; i < mesh->dim_x; ++i) {
    for (usz j = 0; j < mesh->dim_y; ++j) {
        for (usz k = 0; k < mesh->dim_z; ++k) {
                /../
            }
        }
    }
}
```

2. Dans `solve.c` au niveau de la fonction `solve_jacobi` qui s'occupe de du calcul du stencil, on utilise la même clause avec la même taille de chunk.

```
    /../
#pragma omp parallel for schedule(static,2)
for (usz kk = STENCIL_ORDER; kk < dim_z_minus_ST; kk += BLOCK_SIZE_Z) {
    for (usz jj = STENCIL_ORDER; jj < dim_y_minus_ST; jj += BLOCK_SIZE_Y) {
        for (usz ii = STENCIL_ORDER; ii < dim_x_minus_ST; ii += BLOCK_SIZE_X) {

            for (usz i = ii; i < min(dim_x_minus_ST, ii + BLOCK_SIZE_X); ++i) {
                for (usz j = jj; j < min(dim_y_minus_ST, jj + BLOCK_SIZE_Y); ++j)
                    for (usz k = kk; k < min(dim_z_minus_ST, kk + BLOCK_SIZE_Z);
                        /../
                        }
                        C->cells.value[idx] = sum;
                    }
                }
            }
        }
    }
}
```

Si nous avons choisie ces boucles particulièrement, c'est en partie en raison de la first touch policy, bien que nos mesures ne soit pas dans une architecture NUMA, le fait de paralléliser ces boucles garantit qu'on n'expérimenteras pas d'effet NUMA.

Maintenant que nous avons introduit les threads OpenMp, nous devons trouver un certain équilibre entre le nombre de processus MPI et le nombre de threads OpenMP, voici le résultats de nos expérimentation :

Nb threads OMP	Nb process MPI	SpeedUp(%)	Temps d'exécution (s)
2	2	6017.91	0.014982026
2	4	10356.87	0.009344502
4	2	6040.20	0.015135573
4	4	7431.77	0.011133282
8	2	5752.32	0.015954329
8	4	2752.25	0.032533933

TABLE 2.13 – Résultat - schedule(static,1)

Nb threads OMP	Nb process MPI	SpeedUp(%)	Temps d'exécution (s)
2	2	5712.66	0.015836513
2	4	8063.00	0.011550126
4	2	5772.01	0.015740388
4	4	6268.31	0.013900512
8	2	5741.66	0.015573765
8	4	2883.70	0.030558891

TABLE 2.14 – Résultat - schedule(static,2)

Nb threads OMP	Nb process MPI	SpeedUp(%)	Temps d'exécution (s)
2	2	5744.91	0.016151469
2	4	7359.31	0.012195704
4	2	5687.00	0.015963125
4	4	6347.21	0.015256395
8	2	5690.25	0.015962444
8	4	2776.93	0.034618453

TABLE 2.15 – Résultat - schedule(static,4)

En faisant varier la taille du chunk distribué en round-robin, on constate que dans les 3 cas, on obtient toujours le meilleure résultats avec le combo 2 threads OMP et 4 Process MPI, on note également un nouveau pic à `10356.87%` avec une taille de chunk de 1 et le combo précédent.

6 Dernière étude de scalabilité

Nous avons fais le tour des optimisations que nous avons implémentée , on va maintenat réaliser une dernière étude de la scalabilité sur notre version finale sur un problème de taille 128x128x128

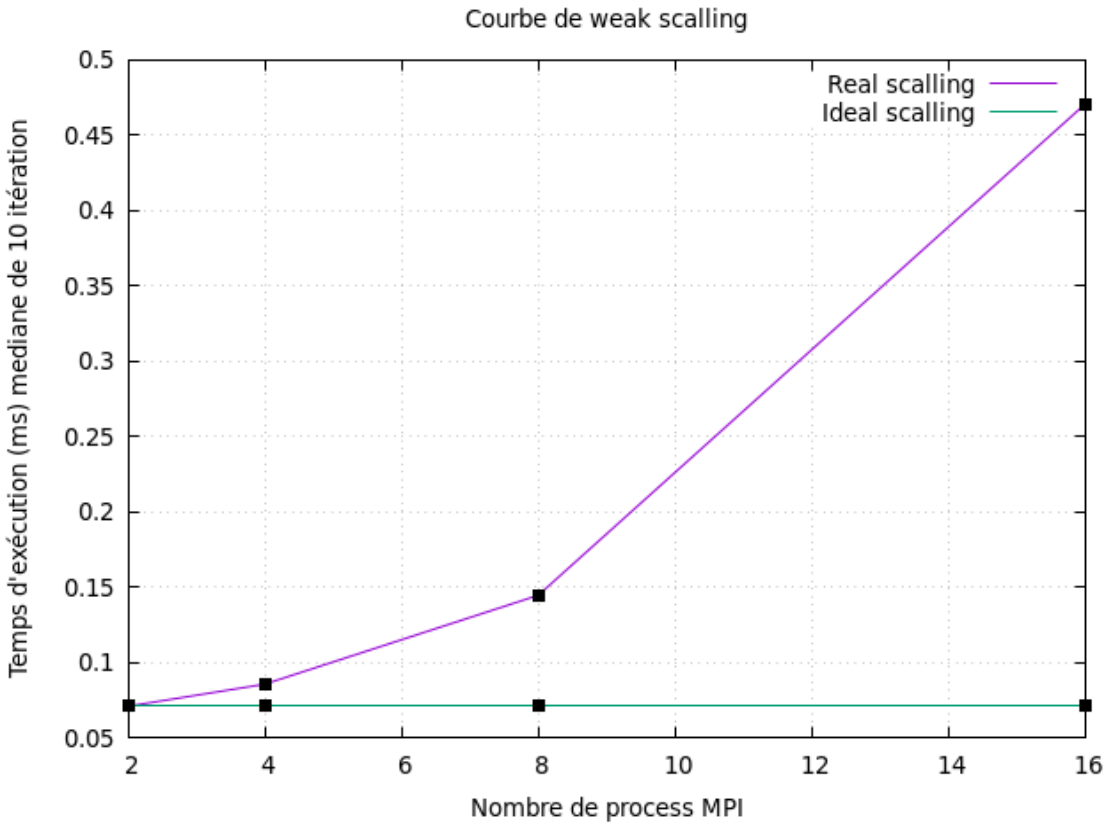


FIGURE 2.20 – Courbe de weak scalling

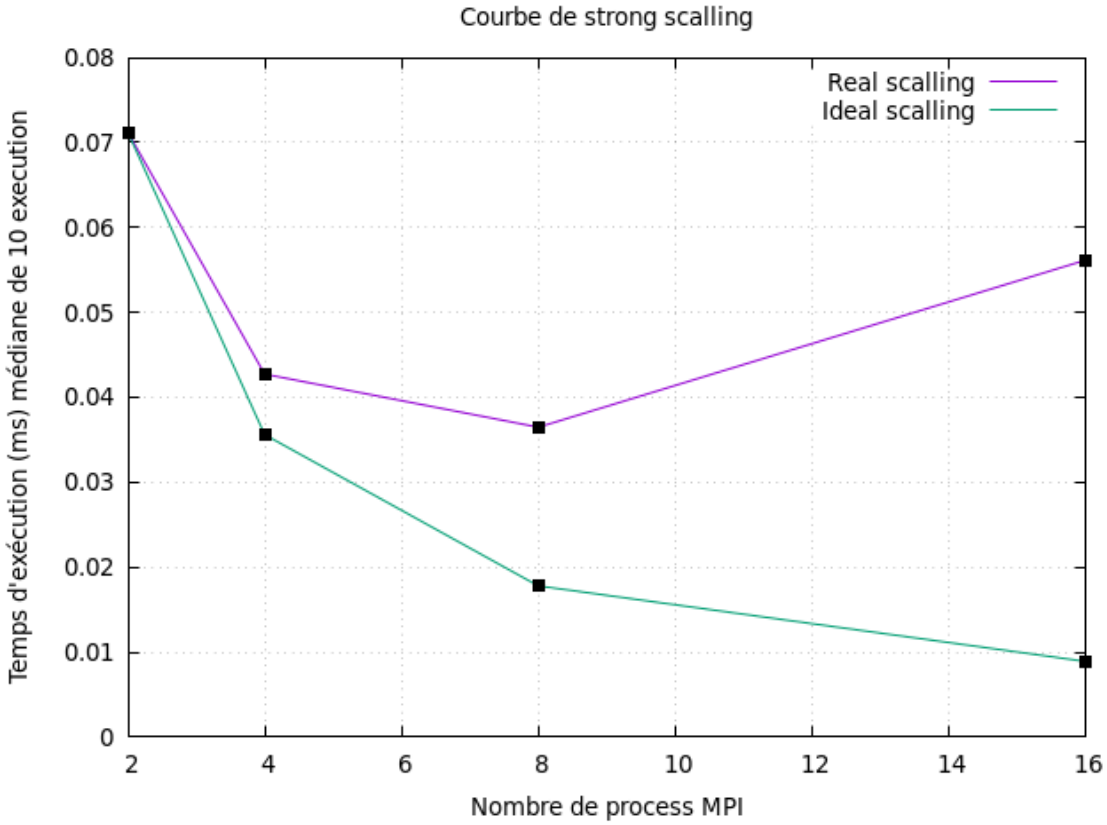


FIGURE 2.21 – Courbe de strong scalling

Notre version finale n'est donc ni strong scalled ni weak scalled.

7 Conclusion

Ce projet nous auras permis de nous familiariser avec les techniques d'optimisation et d'optimisation de la parallélisation de manière approfondie, de prendre en mains tout l'arsenal d'outils de profilage et de debbugage afin de mieux comprendre le comportement d'un programme, ce projet a été une expérience très enrichissante, le code jouet fournie nous offrait toutes les liberté d'expérimentation avec les techniques vu en cours et en TDs.

Bibliographie

[1] Depôt github du projet. https://github.com/gethubryma/TOP_PROJECT_GRP.

[2] Rapport overleaf latek. https://fr.overleaf.com/read/fdtbbcdwbkjc#f89556.

[3] Gnu debugger. https://fr.wikipedia.org/wiki/GNU_Debugger.

[4] Valgrind. https://valgrind.org/.

[5] Gnu profiler man page. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[6] Linux perf man page. https://perf.wiki.kernel.org/index.php/Main_Page.

[7] Modular assembly quality analyzer and optimizer. https://www.maqao.org/.

[8] strace linux syscall tracer. https://strace.io/.