# Examens final – Automne 2009 :

## Parallélisation hybride en mémoire partagée et distribuée

# Introduction

Au cours de notre session, nous avons vu qu'il existe différents types de programmation parallèle. L'exercice que nous allons traiter nous permet d'aborder la programmation hybride mêlant les concepts de mémoire partagée et distribuée. Pour ce faire, nous disposons d'un **programme séquentiel dont l'objectif est de déterminer la répartition de la chaleur dans un bloc solide cubique** sur lequel a été appliquée différentes sources chauffantes. Au cours de cette introduction, nous allons présenter l'organisation de ce programme afin d'identifier les parties parallélisables. Ce dernier est donc composé de quatre fonctions :

- **La fonction principale,** qui alloue en mémoire la taille nécessaire à la représentation du bloc cubique et qui fait appel à une fonction d'initialisation des valeurs de la chaleur pour chacun des points du bloc. Ensuite elle fait appel à la fonction "gauss_seidel" qui réalise le calcul approximatif de la répartition de la chaleur. Pour obtenir la meilleur approximation possible, cette fonction effectue autant de passage que nécessaire jusqu'à ce que la valeur de l'erreur sur le calcul dépasse un certain plafond ou que le nombre maximal de passage soit atteint. Une fois la répartition de la chaleur déterminée, la fonction principale appelle la fonction "write" qui va générer des images représentant cette répartition en différentes tranches du bloc.
- **La fonction d'initialisation**, parcours chaque point du bloc et initialise leur valeur de la chaleur. Des valeurs particulières sont données pour chaque face du cube (points extrêmes) et une valeur de base pour ses points internes qui seront modifiés par "gauss_seidel".
- **La fonction "gauss_seidel"**, est une méthode itérative qui permet d'obtenir une approximation de la chaleur comme dit précédemment. Celle-ci parcours également chaque point du bloc et calcul la valeur de la chaleur en ce point en fonction de ses voisins sur les différents axes "x", "y" et "z". La fonction réalise avant tout une copie de l'ancienne valeur avant le calcul et détermine la différence avec la nouvelle. La valeur maximale des différences est finalement retournée par la fonction.
- **La fonction "write"**, parcours dans un premier temps chaque point du bloc pour déterminer les valeurs maximales et minimales. Ces deux dernières sont utilisées afin d'ajuster la couleur des autres points du cube lors de la génération de l'image d'une tranche pour une valeur "x" donnée.

A la lecture de cette description, nous nous apercevons que tout le problème réside dans le parcours du bloc et le calcul de la chaleur par la fonction "gauss_seidel". La structure itérative de chaque fonction du programme nous permettra de paralléliser ce dernier et d'essayer d'en tirer un avantage maximum.

# 1- Parallélisation MPI

## *1a. Approche*

Comme nous avons pu le remarquer dans la précédente description, l'**ordre global** du programme est et **restera séquentiel** avec les trois principales phases qui sont l'initialisation, le calcul et la génération d'image.

Cependant, chacune de ces fonctions possède une structure itérative qu'il sera possible de paralléliser. La partie la plus importante du programme reste le calcul des valeurs de chaleur qui est réalisé un certain nombre de fois, borné par le nombre de passes ou selon un taux d'erreur global.

La parallélisation selon l'interface MPI utilise le concept de mémoire distribuée. Selon ce principe, il sera possible de **répartir les calculs de la fonction "gauss_seidel"** en utilisant un partitionnement de données **pour que celle-ci ne traite qu'une portion du volume totale**. Un avantage de cette solution est que si la taille de ce volume augmente, il sera possible de la répartir sur plusieurs nœuds de calcul.

## *1b. Partitionnement*

L'espace dans lequel le programme évolue est un cube solide. De plus, la méthode de résolution selon "gauss_seidel" représente un maillage structuré en 3 dimensions où chaque point possède une "connexion" avec ses voisins pour le calcul de sa nouvelle valeur. Afin d'implémenter cette approche de **partitionnement de données** et d'équilibrer la charge de calcul sur les processeurs ou différents nœuds, nous allons **décomposer le volume du bloc en P "tranches" de hauteur N/P selon l'axe des Z**, N étant la taille unitaire du bloc et P le nombre de "process" MPI qui seront exécutés.

Cependant, cette répartition présente un problème. Pour calculer la nouvelle valeur de la chaleur d'un point, la méthode "gauss_seidel" a besoin d'accéder à la valeur de la chaleur de ses voisins. Si il n'y a aucun problème pour les voisins des axes "x" et "y", nous aurons un problème en ce qui concerne les voisins de l'axe "z" pour les "couches" extrêmes frontières de chaque "tranche". En effet, les voisins seront présents dans les "tranches" traitées par d'autres "process". Pour résoudre ce problème, nous allons utiliser la **méthode additive des points fantômes selon Jacobi**. Ainsi, chaque "proccess" MPI possède un tranche de hauteur N/P + le nombre de "couche fantôme" nécessaire. A chaque itération, après le calcul de la fonction de "gauss_seidel", les "process" se synchroniseront afin de s'échanger les valeurs de leurs couches fantômes. De ce fait, lors de la prochaine itération, les valeurs des couches extrêmes de la tranche N/P seront correctement calculées.

## *1c. Communications*

Comme nous venons de le voir, nous devons effectuer une **synchronisation** entre les différents "process" MPI qui traiteront chacun une partie du volume principale. De plus, une fois qu'ils auront tous achevé leurs calculs, **il sera nécessaire de rassembler ces valeurs** en un unique bloc pour ensuite

procéder à la génération des images.

Au cours du dernier laboratoire qui nous a permis de nous familiariser avec l'API MPI, nous avons pu remarquer qu'il est préférable d'employer la méthode d'envoi de message "Bsend". En effet, celle-ci étant "non bloquante", elle est favorable à l'émission de messages contenant beaucoup de données. Le fait d'utiliser la méthode "Bsend" permet au "process" de réaliser les deux envois consécutifs sans perdre de temps et de se mettre en attente des messages de ses voisins. Le reste de l'émission sera effectuée en tâche de fond pendant le temps d'attente. Ainsi c'est cette méthode que nous emploierons pour la **synchronisation des points fantômes**. Pour ce faire, nous devrons également déclarer et allouer un espace mémoire tampon permettant l'envoi des messages. De manière générale, chaque "process " devra envoyer une "couche" de points à ses deux "process" voisins, c'est pourquoi le "buffer" devra pouvoir accueillir au minimum ces deux "couches" de données, le temps que le premier message soit complètement émis.

En ce qui concerne la fonction qui aura la responsabilité de **rassembler les valeurs finales** calculées, elle pourra utiliser la **méthode standard "Send"** puisque si cette dernière est bloquante, il n'y aura aucun effet sur le temps d'exécution étant donné que le "process" en question n'aura plus aucune tâche à réaliser après cet envoi.

A noter que lors de la synchronisation des couches fantômes, **il faudra également veiller à la synchronisation et à la réduction de la valeur d'erreur** retournée par chaque appel de "gauss_seidel" puisque cette valeur intervient en tant que condition de fin d'itération. Cette synchronisation s'effectuera par l'intermédiaire de la méthode "Allreduce" qui permet de réaliser la réduction maximale nécessaire et de diffuser cette valeur à chaque "process". "Allreduce" agit comme une barrière implicite.

En ce qui concerne la **réception des messages**, nous utilisons la méthode standard bloquante "Recv", car aucun "process" MPI ne peut continuer si il n'a pu se synchroniser et recevoir les valeurs des couches fantômes.

## *1d. Problèmes rencontrés*

La principale difficulté dans cette étape de parallélisation avec MPI fut la gestion des points fantômes. En effet, dans chaque "process" MPI, nous ne disposons que d'un pointeur sur le volume représentant la tranche du bloc à traiter. Ainsi, lors de la génération des messages à transmettre aux voisins, il est nécessaire d'ajuster les décalages du pointeurs afin d'accéder aux bonnes données.

De plus, il faut noter qu'il y a principalement deux types de tranches, celle disposant de deux couches de points fantômes et celle des "process" 0 et size-1, qui sont les tranches extrêmes.

Il fallait également prendre en compte la possibilité d'avoir une valeur de "size" ou un nombre de "process" impaire et également traiter le cas où nous n'avons qu'un seul "process" MPI. Dans ce dernier cas, il n'est pas nécessaire de générer les séquences de communication.

# 2- Parallélisation Open MP

## *2a. Approche*

Dans un second temps, nous avons utilisé le langage OpenMP qui est fondé sur le principe de parallélisation en mémoire partagée. En utilisant les directives associées, il est très facile de **paralléliser le traitement interne de nos fonctions et surtout les boucles itératives**. Nous pouvons remarquer qu'il nous aurait été possible de paralléliser les fonctions d'initialisation et "write" (la recherche des extremums ainsi que la génération d'image ). Cependant, dans le reste de notre exercice, nous essayons de comparer les différents types de programmation. Ainsi, si ces fonctions étaient parallélisées à l'aide d'OpenMP et pas avec MPI, cela fausserait notre interprétation des temps de résolutions.

## *2b. Partitionnement*

Encore une fois, nous privilégions le **partitionnement et la réparation de données**. En effet, la fonction "gauss_seidel" repose sur le parcours itératif de chaque points du volume, soit une triple boucle imbriquée. La décomposition classique de ce genre d'instruction permet de répartir l'intervalle d'itération de la première boucle entre les différents threads qui s'occuperont chacun d'une "sous tranche".

## *2c. Directive(s) utilisée(s)*

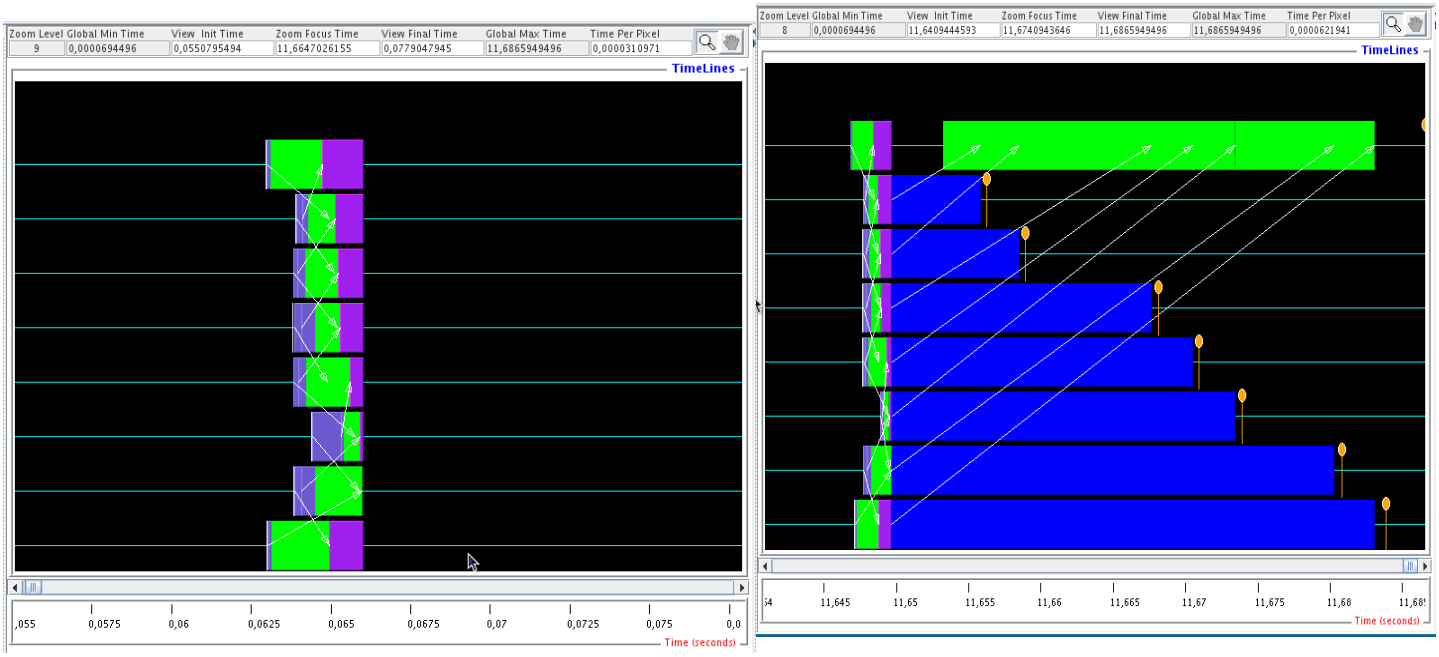
La fonction "gauss_seidel" comporte principalement une partie de traitement itérative. Ainsi nous avons utilisé la directive **"#pragma omp parallel for"**. Encore une fois, chaque thread OpenMP traitera une partie de la tranche associée au "process" MPI en question. La répartition entre les différents threads est réalisée automatiquement par la directive. **A noter qu'il est nécessaire de réaliser une réduction de la variable privée "err"** afin de retourner sa valeur maximal au "process" appelant. Pour ce faire, nous utilisons un tableau de dimension "Nthreads*8" afin d'éviter un problème de faux partage des données en cache. Chaque thread OpenMP ira inscrire sa valeur "privée" de "err" à l'indice associée à son numéro de thread (num_thread *8). A la sortie de la section parallèle de la boucle "for" selon "z", il ne reste plus qu'un seul thread qui parcourt le tableau et réalise la réduction.

## *2d. Problèmes rencontrés*

La parallélisation selon OpenMP fut dans sa globalité plutôt facile. En effet, nous avions déjà eu l'occasion d'effectuer ce genre de parallélisation au cours de notre laboratoire n°2.

# 3- Communication MPI

## *3a. Diagrammes*




## *3b. Explication des séquences*

Voici les séquences de communication effectuées dans le programme parallélisé selon MPI. La **capture de gauche** représente les échanges au cours de la **synchronisation des points fantômes**. Comme nous pouvons l'observer, une fois leur travail achevé, les "process" envoient leur points fantômes à leurs voisins puis se mettent en attente de leurs valeurs respectives. **L'utilisation de "Bsend" nous permet de réduire le temps globale de l'échange** car le reste de l'envoi s'effectue pendant le temps d'attente. Ces communications sont plutôt efficaces lorsque chaque "process" termine le calcul selon "gauss_seidel" en même temps que les autres. Dans le cas où un thread met plus de temps (voir annexes) le gain n'est pas forcément significatif. De plus, la méthode "Allreduce", en violet à la fin de la communication, agit bien comme une barrière et permet la synchronisation totale des "process".

En ce qui concerne les communications lors du **rassemblement des tranches en un bloc final**, nous pouvons voir que le "process" 0 attends les valeurs des autres "process" dans leur ordre de numérotation. Le fait d'utiliser la **méthode standard "Send" ne pose pas de problèmes** car seul le "process" 0 effectuera encore des traitements dans la suite du programme. Ainsi, le temps d'exécution du programme ne dépend que de la séquence de réception, qui elle doit être bloquante. Utiliser la méthode standard "Send" nous permet de n'utiliser qu'un "buffer" de taille inférieur à N/P pour la synchronisation des points fantômes.

# 4- Performances MPI

## *4a. Mesure du temps d'exécution en secondes*

Code séquentiel :

| Real | User | Sys |
|---|---|---|
| 64,34 | 60,72 | 0,02 |

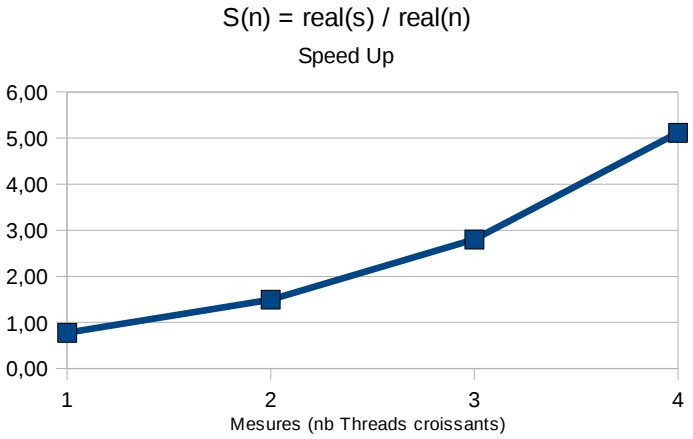
Exécution sur la machine l4712-20

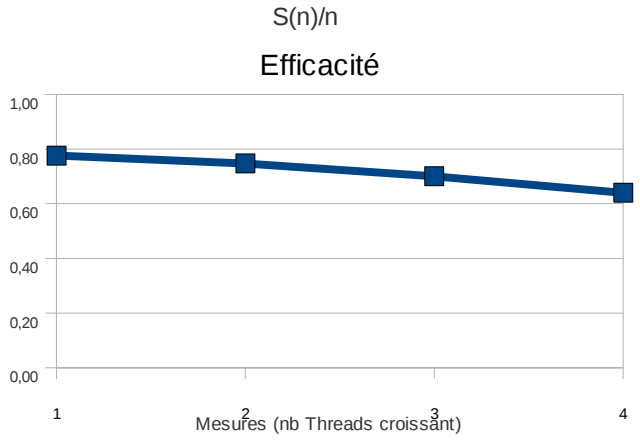
Pour un nombre de "process" MPI variable :

| Threads | Real | User | Sys |
|---|---|---|---|
| 1 | 82,95 | 78,47 | 0,05 |
| 2 | 43,06 | 79,35 | 0,11 |
| 4 | 22,97 | 82,74 | 0,18 |
| 8 | 12,58 | 90,23 | 0,37 |

Les résultats affichés ont été calculés sur une moyenne de quatre exécutions.

## *4b-c. Courbe représentative du Speedup : S(n),*



### *4d. Facteurs limitant*

Lorsque l'on observe la courbe de "Speed Up", l'allure nous autorise à croire que nous pouvons augmenter le nombre de "process" MPI sans problèmes. Cependant, lorsque que nous nous concentrons sur l**a courbe d'efficacité**, nous pouvons remarquer que celle-ci **diminue**. **Cela est principalement du au temps et surtout au nombre de communications** qui sont nécessaires pour la synchronisation des points fantômes. En effet, cette synchronisation à lieu à chaque itération de calcul, après chaque exécution de la fonction "gauss_seidel". Mais **avec l'augmentation du nombre de "process" MPI, le nombre de messages augmente également, alors que la quantité de données échangée entre chaque "process" diminue**. Ainsi, l'avantage de l'utilisation de la méthode "Bsend" se perd progressivement.

Le fait d'avoir exécuté le programme parallélisé avec MPI sur un seul "host" pour les mesures ne pose aucun problème. En effet, des test ont été réalisés en distribuant le calcul sur plusieurs nœuds et nous obtenons le même ordre de grandeur en ce qui concerne le temps d'exécution (à savoir que dans ce cas les latences des communications réseau doivent être prises en compte).

## 5- Performances OpenMP

### *5a. Mesure du temps d'exécution*

Code séquentiel :

| Real | User | Sys |
|---|---|---|
| 64,34 | 60,72 | 0,02 |

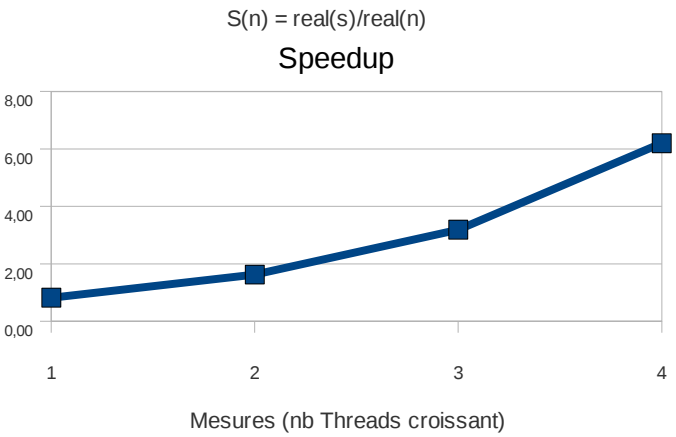
Exécution sur la machine l4712-20

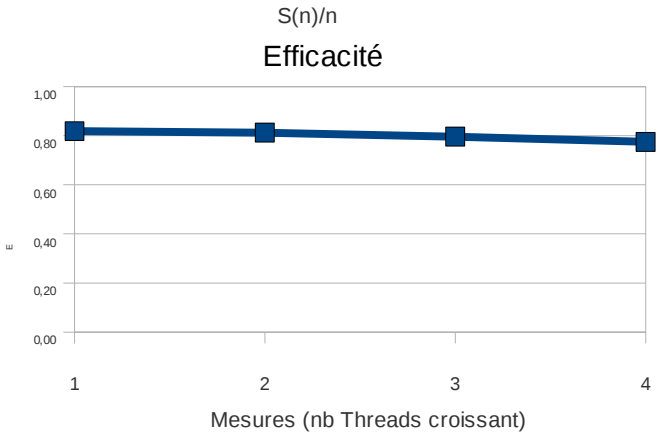
Pour un nombre de threads OpenMP variable :

| Threads | Real | User | Sys |
|---|---|---|---|
| 1 | 78,59 | 78,49 | 0,03 |
| 2 | 39,58 | 78,98 | 0,04 |
| 4 | 20,21 | 80,45 | 0,06 |
| 8 | 10,38 | 82,23 | 0,10 |

Les résultats affichés ont été calculés sur une moyenne de quatre exécutions sur la machine l4712-16.

### *5b-c. Courbe représentative du Speedup : S(n) et de l'efficacité : S(n)/n*




### *5d. Facteurs limitant*

En ce qui concerne la parallélisation de la fonction "gauss_seidel" à l'aide d'OpenMP, nous pouvons voir que l**a courbe d'efficacité est plus constante que celle de MPI**. La valeur pour 1,2, 4 ou 8 thread est toujours de l'ordre de 0,8. Ainsi, tout porte à croire qu'il n'y a aucun facteur limitant à ce niveau, si ce n'est la réduction concernant la variable "err". Cette réduction a été implémentée en utilisant un tableau qui doit être parcouru linéairement par un seul thread en fin de calcul. Ainsi, plus le nombre de thread OpenMP est important, plus la réduction sera longue.

## 6- Performances Hybride

### *6a. Mesure du temps d'exécution*

Code séquentiel :

| Real | User | Sys |
|---|---|---|
| 64,34 | 60,72 | 0,02 |

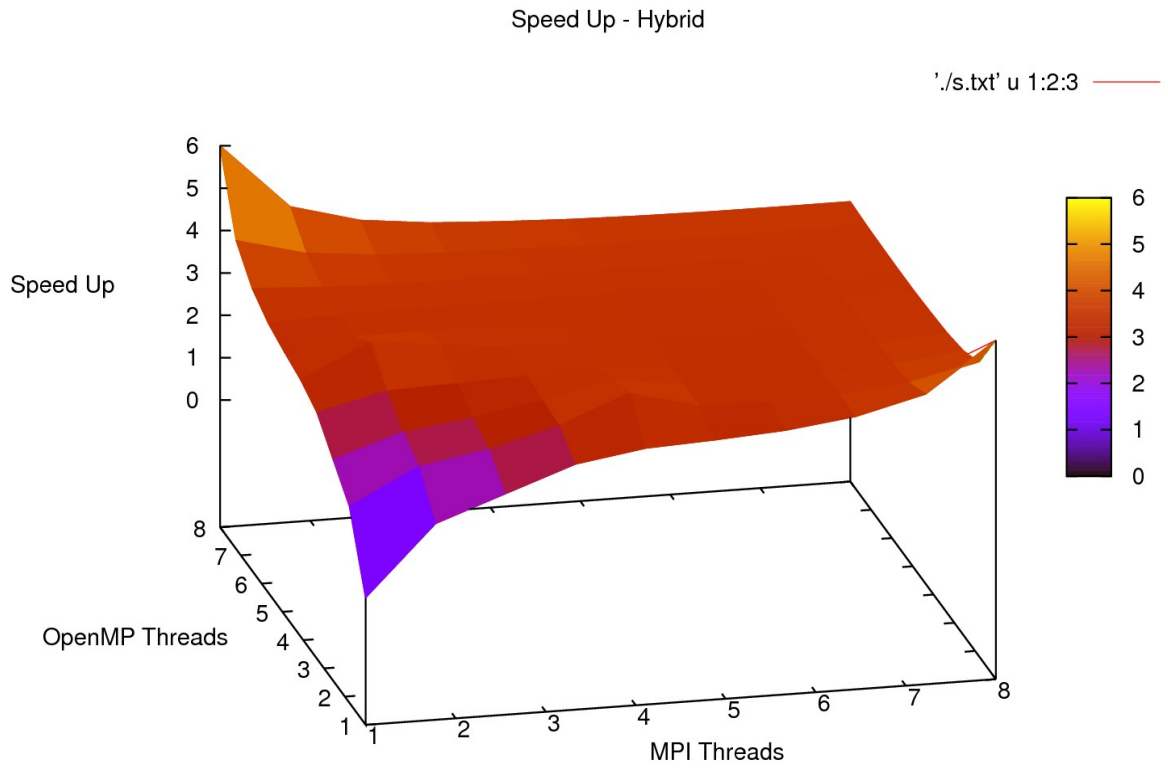
Exécution sur la machine l4712-20

Pour un nombre de threads MPI et OpenMP variable :

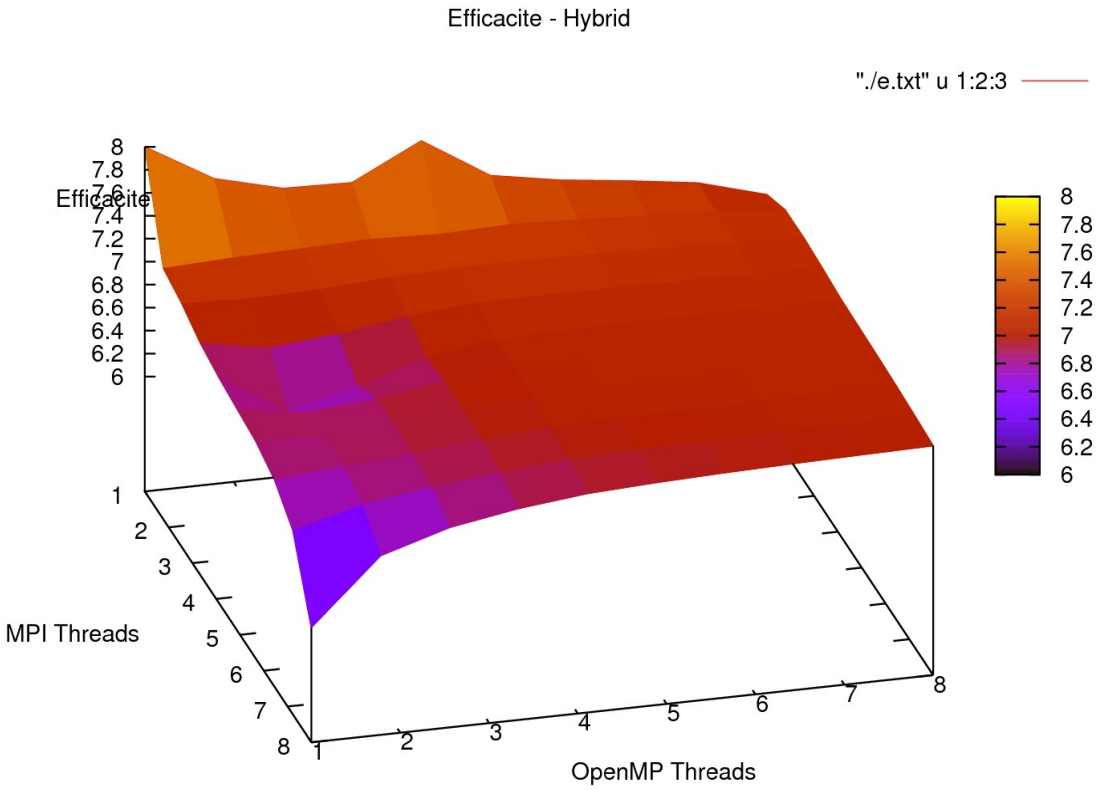
| Threads MPI – OMP | Real | User | Sys |
|---|---|---|---|
| 1 – 1 | 78,84 | 78,74 | 0,03 |
| 1 – 2 | 39,55 | 78,91 | 0,04 |
| 2 – 1 | 40,69 | 79,07 | 0,10 |
| 1 – 4 | 20,19 | 80,35 | 0,06 |
| 2 – 2 | 21,48 | 81,34 | 0,12 |
| 4 – 1 | 21,23 | 82,22 | 0,18 |
| 1 – 8 | 10,35 | 82,03 | 0,10 |
| 2 – 4 | 11,54 | 83,46 | 0,16 |
| 4 – 2 | 11,73 | 86,75 | 0,23 |
| 8 – 1 | 12,80 | 90,52 | 0,39 |

Les résultats affichés ont été calculés sur une moyenne de quatre exécutions sur la machine l4712-16.

## *6b. Courbe représentative du Speedup : S(n)*



## *6c. Courbe représentative de l'efficacité : S(n)/n*




## Note :

- Les axes des graphiques de "Speed Up" et d'efficacité ont été inversés afin d'avoir un meilleur aperçu de l'ensemble de la surface.
- Les valeurs de l'efficacité ont été multipliées par un facteur 10 afin d'obtenir le graphique de surface.

## *6d. Facteurs limitant*

A la lecture de la courbe de surface du "Speed Up", nous pouvons remarquer que nous obtenons des valeurs maximales lorsque le programme s'exécute soit entièrement en OpenMP, soit entièrement en MPI.

Pour ce qui est de l'efficacité, nous pouvons remarquer que la tendance à la diminution avec l'augmentation du nombre de "process" MPI est confirmée. Nous obtenons cependant un pic d'efficacité lorsque nous sommes en configuration avec 2 "process" MPI et 4 threads OpenMP.

# Conclusion

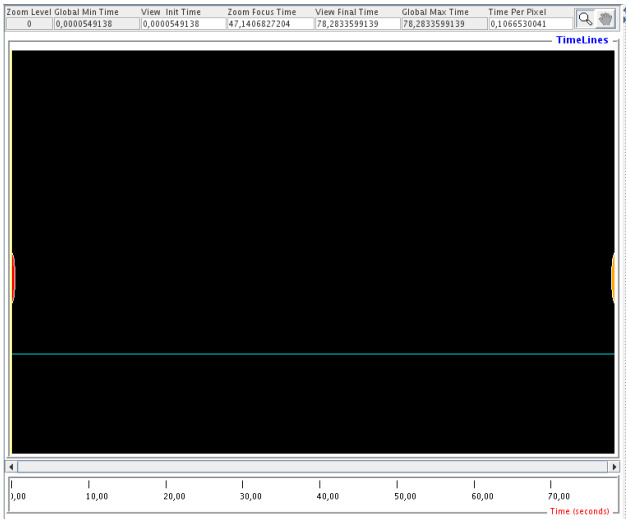
Au cours de cet exercice, nous avons procédé à la parallélisation d'un programme de calcul de répartition de chaleur dans un cube solide utilisant la méthode itérative de "Gauss-Seidel". Dans un premier temps, nous avons réparti le volume total du cube en plusieurs tranches, traitée chacune par un "process" MPI. Nous avons conservé un état synchronisé à l'aide de messages et de communications de données. Dans un second temps, nous avons parallélisé le traitement interne de la fonction "Gauss_seidel" de manière à partager les itérations de calculs internes à une "tranche".

Pris séparément, dans chacun des cas, nous obtenions des résultats assez satisfaisant sur le temps de résolutions du programme. Lors de l'utilisation en mode hybride, nous avions également obtenu des temps sensiblement comparables et de même ordre de grandeur que lors des exécutions complète MPI ou OpenMP.

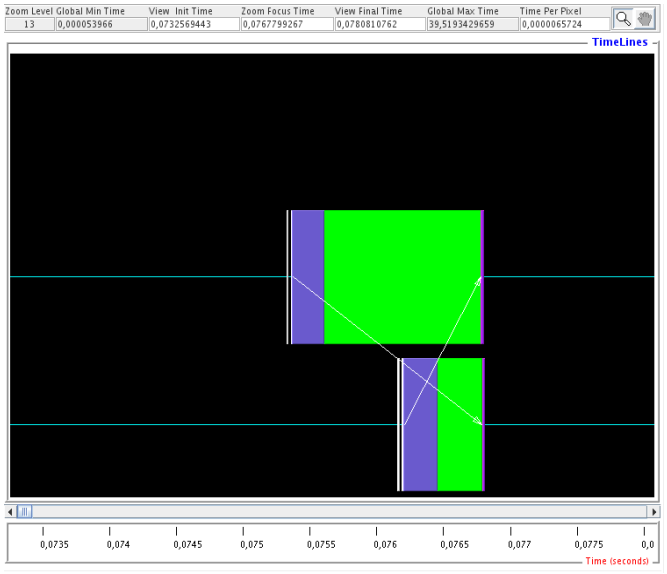
Cependant, nous pouvons ajouter que de part leur concept de mémoire distribuée pour l'un et partagée pour l'autre, il semble préférable d'utiliser les "process" MPI pour répartir le volume total du bloc sur différents nœuds de calcul lorsque celui-ci est important. En ce qui concerne le nombre de thread OpenMP, ce dernier peut être maximal pour chaque nœud. En suivant ce principe, nous devrions obtenir des résultats assez satisfaisant. Dans certains de nos test, nous atteignons un temps de 7s pour une exécution répartie sur 2 nœuds avec une taille par défaut de 128 et un temps de 26,18s pour 8 nœuds et une taille de 256 (temps séquentiel : 535,55s). Ce denier cas nous offre une accélération de 20,46 mais seulement une efficacité de 0,3. Encore une fois, le facteur limitant est le passage de messages et la latence des communications réseau pour ce genre de programme qui ne possède pas beaucoup de partie parallélisable significatives et un nombre importants d'échanges.

# Annexes

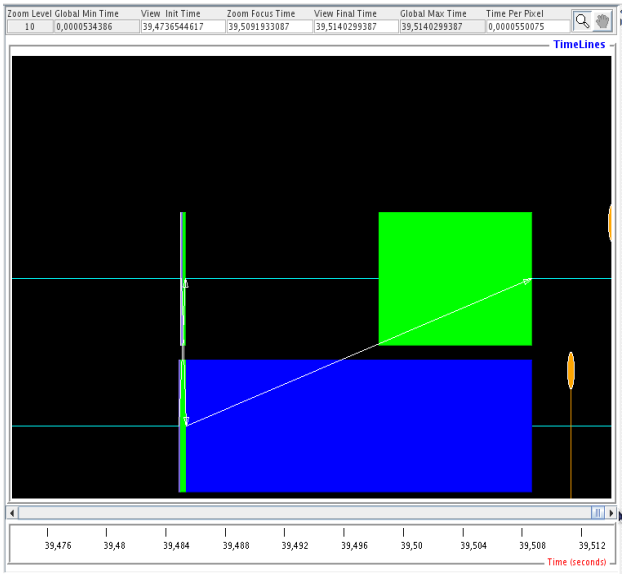
## *Captures de communications MPI*




*Communications avec 1 "process" MPI*

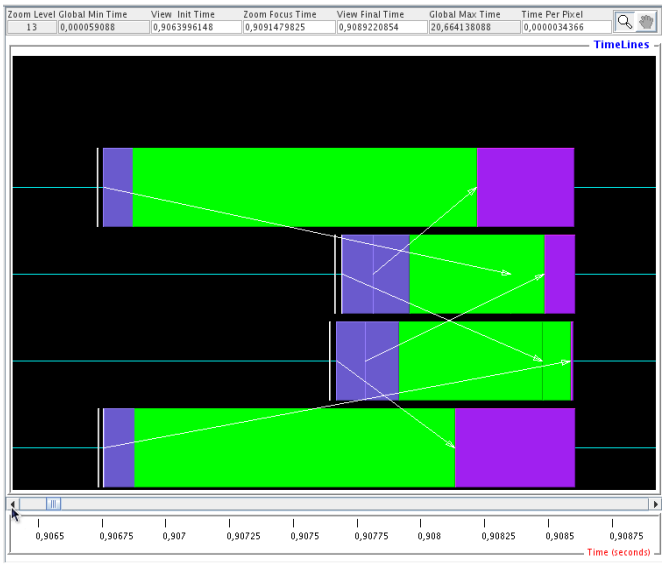



*synchronisation des points fantômes*

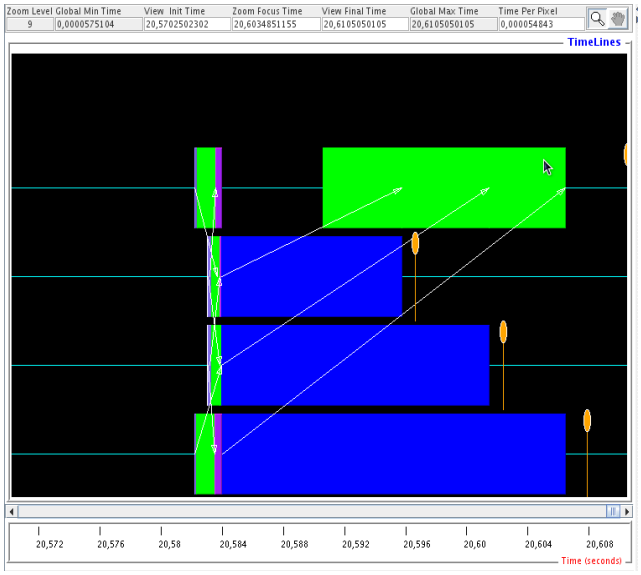



*rassemblement des tranches*

*Communications avec 2 "process" MPI*

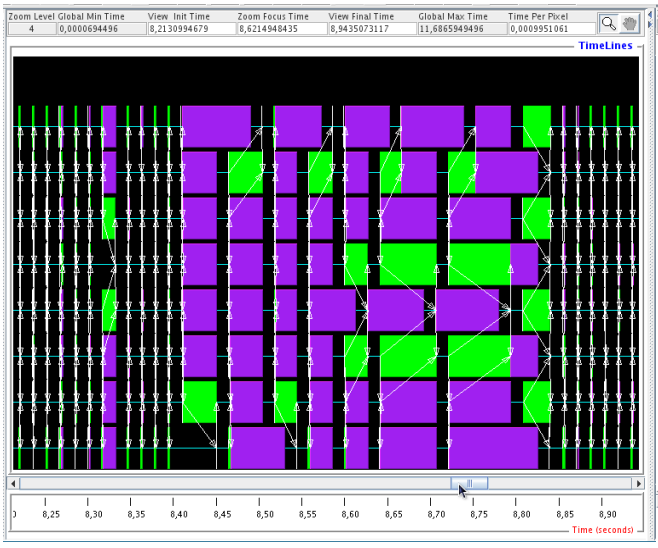


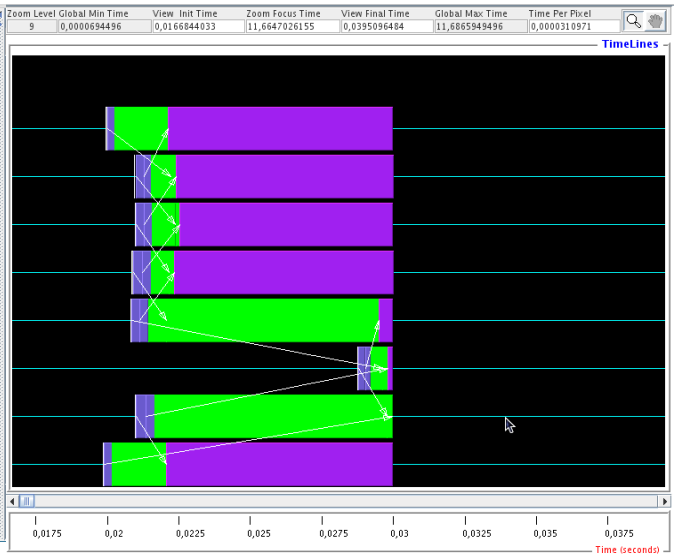

*synchronisation des points fantômes*




*rassemblement des tranches*

*Communications avec 4 "process" MPI*

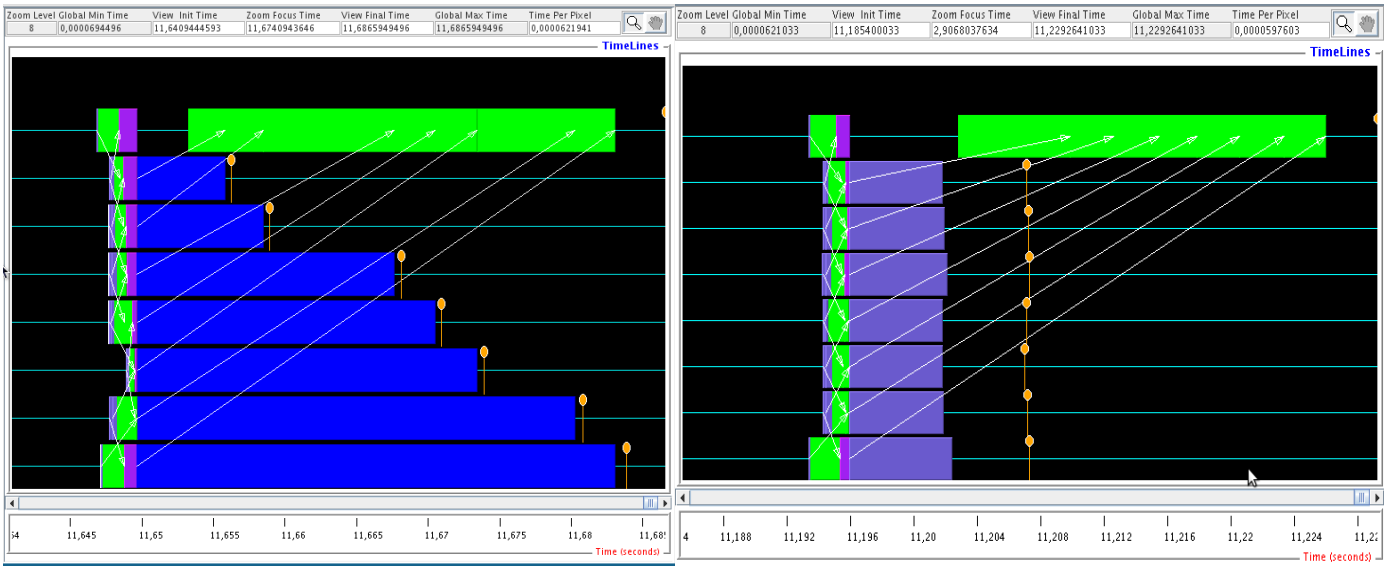



*Communications avec 8 "process" MPI*

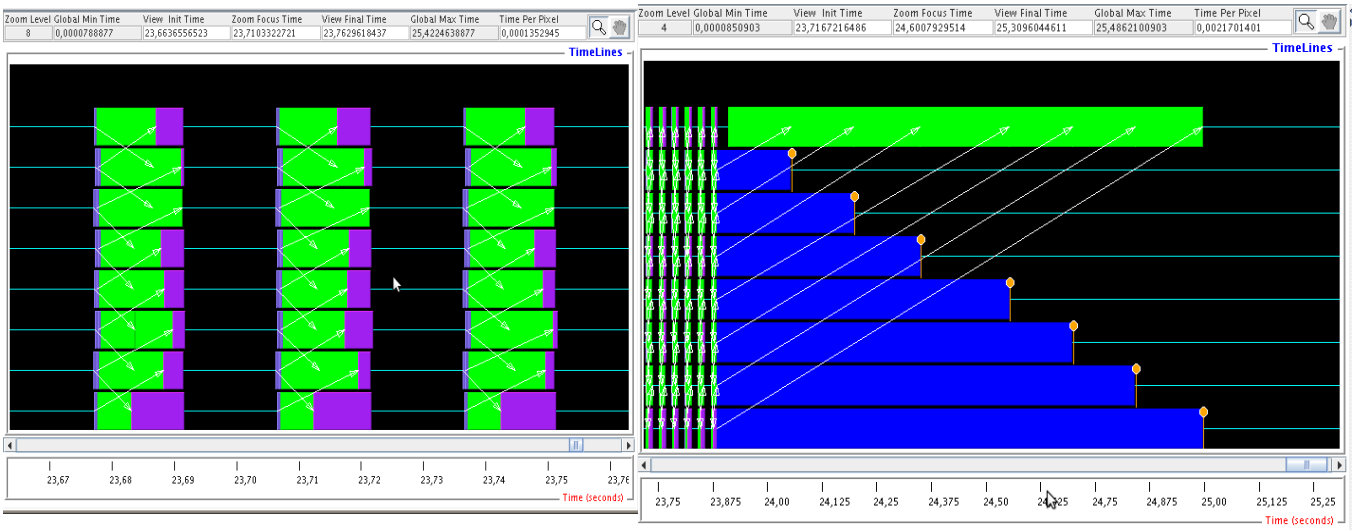
*Capture de gauche : Synchronisation de points fantômes avec un "process" en retard.*

*Capture de droite : Aperçu de plusieurs itérations de "gauss_seidel".*




*Communications avec 8 "process" MPI*

*Communications lors du rassemblement des tranches en un bloc final. A gauche les communications utilisant la méthode standard "Send" et à droite la méthode non bloquante "Bsend".*




*Communications avec 8 "process" MPI*

*Communications lors de la synchronisation et du rassemblement des tranches avec une taille de bloc de 256 et une exécutions répartie sur 8 nœuds de calcul. Nous pouvons remarquer des temps de communication légèrement plus long , dû aux interfaces réseau.*

## Code hybrid.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "omp.h"

#define ERROR             0.035
#define ITERATION   1000

// Maximum Open MP threads (for reduction of "err" in Gauss_seidel)
#define MAX_THREADS       256

//#define PRGDEBUG
#ifdef PRGDEBUG
  #define DBG(x) printf x
#else
  #define DBG(x)
#endif


void initialize(double *volume, long size, int comm_rank, int comm_size, long step);
double gauss_seidel(double *volume, long size, int comm_rank, int comm_size, long step);
void write(double *volume, long size);

void swap_ghost(double *volume, long size, int comm_rank, int comm_size, long step, void
*buffer);
void gathering(double *volume, double *cube, long size, int comm_rank, int comm_size, long
step, long rest);


int main(int argc, char **argv){
  double *volume = NULL, *cube = NULL;
  double error, my_error;
  long iteration, size;
  long step, rest;

  int comm_rank, comm_size;
  int thread_num, num_threads;
  void* buffer;
  int taille;

  if (argc >= 2) {
    size = atol(argv[1]);
  } else {
    size = 128;
  }

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);


  // 0 - Prepare Bsend's buffer
  MPI_Pack_size( (int)(size*size*2), MPI_DOUBLE, MPI_COMM_WORLD, &taille);
  taille += (2*MPI_BSEND_OVERHEAD);
```

```
  DBG(("8 - Thread %d : taille : %d\n", comm_rank, taille));

  buffer = (char *)malloc(taille);
  if( NULL == buffer ){
    printf("pb dans l'allocation de buffer dans Swap\n");
    exit(-1);
  }
  MPI_Buffer_attach(buffer, taille);


  // 1 - Data interval for each MPI Thread
  step = (size/comm_size);
  rest = size % comm_size;  // case if "size" or "comm_size" are odd

  if( 1 != comm_size ){
    if( comm_rank == 0 ){
      step +=1;
    }
    else if( comm_rank == comm_size -1 ){
      step += (1 + rest);
    }
    else {
      step +=2;
    }
  }
  DBG(("2 - Thread %d : step : %ld ! \n", comm_rank, step));


  // 2 - Bloc dynamic allocation for curent MPI Thread
  volume = (double *) malloc(size * size * step * sizeof(double));
  if( NULL == volume ){
    printf("2 - Thread %d : Erreur allocation de volume ! \n", comm_rank);
    exit(-1);
  }


  // 3 - Initilization
  initialize(volume, size, comm_rank, comm_size, step);
  if( comm_rank == 0 ){
    printf("Initialisation - done\n");
  }

  // 4 - Computation
  DBG(("Thread %d - start computation\n", comm_rank));
  iteration = 0;

  if( 1 != comm_size ){
    // More than One MPI Thread
    do {
      my_error = gauss_seidel(volume, size, comm_rank, comm_size, step);
      DBG(("Thread %d - out from GAUSS\n", comm_rank));
      iteration++;

      swap_ghost(volume, size, comm_rank, comm_size, step, buffer);

      MPI_Allreduce(&my_error, &error, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
      DBG(("Thread %d - iteration %ld error %f\n", comm_rank, iteration, error));
    } while ((iteration < ITERATION) && (error > ERROR));
  }else {
```

```
    // Only One MPI Thread
    printf("1 MPI Thread : Gauss_seidel\n");
    do {
      error = gauss_seidel(volume, size, comm_rank, comm_size, step);
      DBG(("Thread %d - out from GAUSS\n", comm_rank));
      iteration++;
      DBG(("Thread %d - iteration %ld error %f\n", comm_rank, iteration, error));
    } while ((iteration < ITERATION) && (error > ERROR));
  }
  DBG(("Thread %d - out from computation\n", comm_rank));

  if( comm_rank == 0 ){
    printf("Gauss_Seidel - done - iteration %ld error %f\n", iteration, error);
  }


  if( 1 == comm_size ){
    printf("1 MPI Thread : write\n");
    write(volume, size);
  }
  else{
    // 5 - Gathering volumes
    if( 0 == comm_rank ){
      cube = (double *) malloc(size * size * size * sizeof(double));
      if( NULL == cube ){
        printf("2 - Thread %d : Erreur cube == NULL ! \n", comm_rank);
        exit(-1);
      }
    }

    gathering(volume, cube, size, comm_rank, comm_size, step, rest);
    //MPI_Barrier(MPI_COMM_WORLD);
    DBG(("7 - Thread %d : After gathering barrier! \n", comm_rank));

    // 6 - Write pictures
    if( 0 == comm_rank ){
      if( NULL == cube ){
        printf("2 - Thread %d : Erreur cube == NULL ! \n", comm_rank);
        exit(-1);
      }
      DBG(("7 - Thread %d : will write! \n", comm_rank));
      write(cube, size);
      DBG(("7 - Thread %d : has written! \n", comm_rank));

      free(cube);
    }
  }

  free(volume);
  MPI_Finalize();
  return 0;
}
```

```
void initialize(double *volume, long size, int comm_rank, int comm_size, long step){
  long x, y, z;

  for (z = 0; z < step; z++) {
    for (y = 0; y < size; y++) {
      for (x = 0; x < size; x++) {
        if ((x == 0) || (x == (size - 1))) {
          volume[(z * size + y) * size + x] = 373.15;
        } else if (y == 0) {
          volume[(z * size + y) * size + x] = 323.15;
        } else if (y == (size - 1)) {
          volume[(z * size + y) * size + x] = 298.15;
        }

        // First bloc
        else if( comm_rank == 0 && z == 0) {
          volume[(z * size + y) * size + x] = 273.15;
        }

        // last bloc
        else if( comm_rank == (comm_size -1) && z == (step - 1)) {
          volume[(z * size + y) * size + x] = 273.15 + (double) x / (double) size * 100.0;
        }

        else {
          volume[(z * size + y) * size + x] = 295.15;
        }
      }
    }

    // DEBUG ONLY
    if( comm_rank == 0 && z == 0) {
      DBG(("layer 0 initialized\n"));
    }
    else if( comm_rank == (comm_size -1) && z == (step - 1)) {
      DBG(("layer %d initialized\n", size -1));
    }
  }
}


double gauss_seidel(double *volume, long size, int comm_rank, int comm_size, long step){
  long x, y, z;
  double old, new, err = 0.0;
  int currentThread, num_threads;
  double terr[MAX_THREADS] = {0.0};

  // One Z line / Thread Open MP
  #pragma omp parallel for shared(volume, size, comm_rank, comm_size, num_threads, terr)
firstprivate(step) private(z, y, x, currentThread, old, new)
  for (z = 1; z < (step - 1); z++) {
    currentThread = omp_get_thread_num();
    num_threads = omp_get_num_threads();
    DBG(("process %d of %d, thread %d of %d\n", comm_rank, comm_size, currentThread,
num_threads));
    for (y = 1; y < (size - 1); y++) {
      for (x = 1; x < (size - 1); x++) {
        // Sav previous value
        old = volume[(z * size + y) * size + x];
```

```
        // Compute new one
        new = (    volume[((z - 1) * size + y) * size + x] +
           volume[((z + 1) * size + y) * size + x] +
           volume[(z * size + (y - 1)) * size + x] +
           volume[(z * size + (y + 1)) * size + x] +
           volume[(z * size + y) * size + (x - 1)] +
           volume[(z * size + y) * size + (x + 1)]) / 6;
        volume[(z * size + y) * size + x] = new;

        // Compute error (currentThread*8 in order to avoid false sharing)
        if (fabs(new - old) > terr[currentThread*8]) {
           terr[currentThread*8] = fabs(new - old);
        }
      }
    }
  }

  // Error Reduction in sequential mode
  //num_threads = omp_get_num_threads();
  DBG(("num_thread = %d\n", num_threads));

  for (currentThread = 0; currentThread < num_threads; ++currentThread) {
    if (terr[currentThread*8] > err){
     err = terr[currentThread*8];
     DBG(("thread %d : err = %f\n", currentThread, err));
    }
  }

  DBG(("process %d of %d, thread %d of %d : error = %f\n", comm_rank, comm_size,
currentThread, num_threads, err));

  return err;
}


void write(double *volume, long size){
  FILE *fp;
  char name[32];
  double minimum, maximum, unit;
  long x, y, z;
  minimum = INFINITY;
  maximum = 0.0;

  printf("6 - Write call\n");

  // Search for Max and Min
  for (z = 0; z < size; z++) {
    for (y = 0; y < size; y++) {
      for (x = 0; x < size; x++) {
        if (volume[(z * size + y) * size + x] < minimum) {
          minimum = volume[(z * size + y) * size + x];
        }

        if (volume[(z * size + y) * size + x] > maximum) {
          maximum = volume[(z * size + y) * size + x];
        }
      }
    }
  }
```

```
  printf("6 - Max = %ld Min = %ld \n", maximum, minimum);

  // WRITTING
  for (x = (size / 8); x < size; x += (size / 4)) {
    sprintf(name, "volume_%04ld.ppm", x);
    fp = fopen(name, "wb");
    fprintf(fp, "P6 %ld %ld 255\n", size, size);

    for (z = 0; z < size; z++) {
      for (y = 0; y < size; y++) {
        unit = (volume[(z * size + y) * size + x] - minimum) / (maximum - minimum);

        fputc(255.0 * (1.0 - cos(unit * M_PI_2) * cos(unit * M_PI_2)), fp);
        fputc(255.0 * sin(unit * M_PI) * sin(unit * M_PI), fp);
        fputc(255.0 * cos(unit * M_PI_2) * cos(unit * M_PI_2), fp);
      }
    }
    fclose(fp);
  }
}


void gathering(double *volume, double *cube, long size, int comm_rank, int comm_size, long
step, long rest) {
  int count;
  long i;
  MPI_Status status;

  DBG(("5 - Thread %d : Gathering! \n", comm_rank));

  // Step without ghost points layers
  if( ((comm_size-1) == comm_rank) || (0 == comm_rank) ){
    --step;
  }
  else {
    step -= 2;
  }

  // Compute
  if( 0 == comm_rank ){
    // Gathering
    if( NULL == cube ){
      printf("Thread %d : Erreur allocation de cube ! \n", comm_rank);
      exit(-1);
    }

    // Thread 0
    for( i=0; i< (size*size*step); ++i ){
      cube[i] = volume[i];
    }

    // Others Threads
    for(i = 1; i< (comm_size - 1); ++i){
      MPI_Recv( (cube + i*size*size*step), (int)(size*size*step), MPI_DOUBLE ,i ,0,
MPI_COMM_WORLD, &status);

      MPI_Get_elements(&status, MPI_DOUBLE, &count);
      DBG(("5 - Thread %d : Recieved %d values from thread %d ! \n", comm_rank, count,
i ));
```

```
      DBG(("5 - should get %d ! \n\n",  (int)(size*size*step) ));
    }

    // Last Thread
    MPI_Recv( cube + i*size*size*(step), (int)(size*size*(step +rest)) , MPI_DOUBLE
,i,0,MPI_COMM_WORLD, &status);
    DBG(("5 - Thread %d : Recieved %d values from thread %d ! \n", comm_rank, count, i ));
    DBG(("5 - should get %d ! \n\n",  (int)(size*size*(step+rest)) ));
  }
  else {
    // Send compute volume without ghost points Layer
    DBG(("5 - Thread %d : send %ld values ! \n", comm_rank, size*size*step ));
    MPI_Send( (volume + size*size), (int)(size*size*step), MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
  }
  DBG(("5 - Thread %d : done \n", comm_rank));
}


void swap_ghost(double *volume, long size, int comm_rank, int comm_size, long step, void
*buffer){
  MPI_Status status;

  DBG(("8 - Thread %d - in Swap\n", comm_rank));

  if( (comm_size -1) != comm_rank ){
    // Send penultimate layer
    DBG(("8 - Thread %d : Sent Layer step -2 to thread %d ! \n", comm_rank, comm_rank +1));
    MPI_Bsend( (volume + (size*size*(step-2))), (int)(size*size), MPI_DOUBLE, comm_rank +1,
0, MPI_COMM_WORLD);
    DBG(("8 - Thread %d : after send ! \n", comm_rank));
  }

  if(0 != comm_rank){
    // Send second layer
    DBG(("8 - Thread %d : Sent Layer 2 to thread %d ! \n", comm_rank, comm_rank -1));
    MPI_Bsend( (volume + size*size), (int)(size*size), MPI_DOUBLE, comm_rank -1, 0,
MPI_COMM_WORLD);
    DBG(("8 - Thread %d : after send ! \n", comm_rank));

    // Receive First layer
    DBG(("8 - Thread %d : Waiting for step -2 from thread %d ! \n", comm_rank, comm_rank
-1));
    MPI_Recv( volume, (int)(size*size) , MPI_DOUBLE , comm_rank -1,0,MPI_COMM_WORLD,
&status);
    DBG(("8 - Thread %d : Received step -2 from thread %d ! \n", comm_rank, comm_rank -1));
  }

  if( (comm_size -1) != comm_rank ){
    // Receiv Last layer
    DBG(("8 - Thread %d : Waiting for Layer 2 from thread %d ! \n", comm_rank, comm_rank
+1));
    MPI_Recv( volume + (size*size*(step-1)), (int)(size*size) , MPI_DOUBLE , comm_rank
+1,0,MPI_COMM_WORLD, &status);
    DBG(("8 - Thread %d : Received Layer 2 from thread %d ! \n", comm_rank, comm_rank +1));
  }
}
```