TP 3: Solving the 2D Heat Equation with MPI

Michel Donnet

November 18, 2023

Contents

1 Explication du code

Mon code est composé d'un Makefile, d'un main.cpp contenant la fonction principale, d'un fichier grid.cpp et son header grid.h contenant les signatures des fonctions présentes dans grid.cpp, et d'un fichier writer.cpp avec son header writer.h contenant la signature de la fonction définie dans writer.cpp.

1.1 grid.cpp

Dans ce fichier se trouvent les fonctions utiles pour créer une grille 2D, convertir une grille 2D en grille 1D, convertir une grille 1D en grille 2D, afficher une grille, et enfin une dernière fonction appelée put_into_2D prenant en paramètre une grille 2D et une grille 1D, avec le nombre de colonnes que représente cette grille 1D, le rang du processeur qui désire insérer la grille 1D dans la grille 2D, et la taille de la grille 2D. Cette fonction permet de prendre une grille une dimension représentant une ou plusieurs colonnes d'une grille 2D, et de les insérer dans une grille 2D suivant le rang du processeur...

1.2 main.cpp

Dans le main, on commence par initialiser l'environnement MPI par la commande

```
MPI_Init(NULL, NULL);
```

Puis on collecte l'information du nombre de processeurs utilisé et du rang du processeur actuel

```
// Get rank of processor
int rank = 0;
MPI_Comm_rang(MPI_COMM_WORLD, &rank);
// Get total number of processors
int world_size = 0;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

Puis, si on est le processeur 0, on prend le temps à l'état initial grâce à l'appel à la fonction $MPI\_Wtime()$ qui nous donne le temps en seconde lors de son appel, et à la fin de notre code, on prend le temps et on compare avec le temps initial pour obtenir le temps d'exécution de notre code:

```
// Begining of the code
double start = 0;
if (rank == 0) {
    start = MPI_Wtime();
}
...
// End of the code
if (rank == 0) {
    double end = MPI_Wtime();
```

```
    cout << end - start << endl;
}
```

Ensuite, on prend notre grille 2D, on vérifie que son nombre de colonnes est divisible par le nombre de processeurs utilisés dans notre problème par:

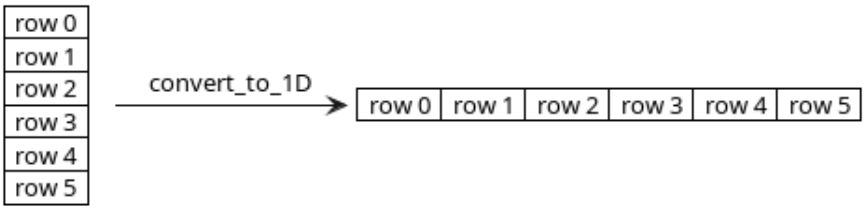
```
if (size % world_size != 0) return -1;
```

Puis on crée une grille 2D, et on la converti en grille 1D grâce aux fonctions définies dans grid.cpp.

```
vector<vector<double>> grid = create_grid(size);
vector<double> grid_line = convert_to_1D(grid, size);
```

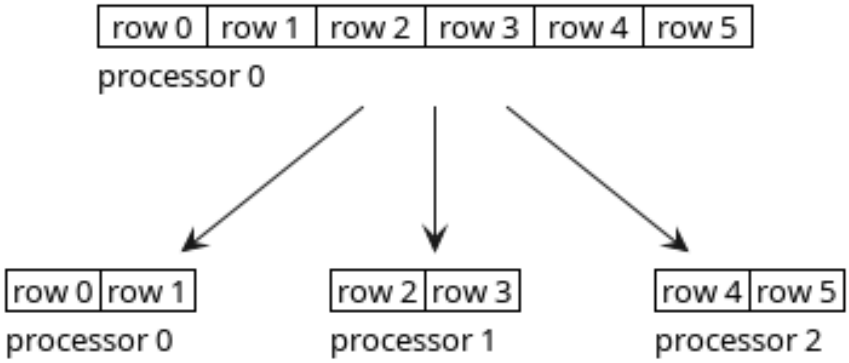
Voici notre grille créée. Ici, pour mes représentations, je vais estimer que les processeurs vont travailler avec notre grille découpée en lignes, car comme notre grille est carrée, travailler avec des lignes est exactement la même chose que travailler avec des colonnes.

Ce qui nous donne la grille précédente en ligne:



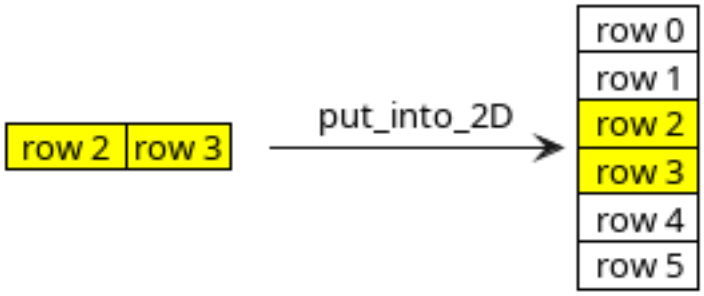
Enfin, on va découper notre grille linéaire en N grilles linéaires plus petites en utilisant la primitive Scatter de MPI qui prend en paramètre un vecteur 1 dimension et qui va le diviser en N vecteurs de dimension donnée dans MPI, avec N le nombre de processeurs utilisés. Cela nous donne:

```
MPI_Scatter(
    grid_line.data(),       // Data to split
    nb_columns * size,      // Size of a split
    MPI_DOUBLE,             // Data type
    recvbuf.data(),         // Destination of a split
    nb_columns * size,      // Size of destination
    MPI_DOUBLE,             // Data type of destination
    0,                      // Rank of processor which split datas
    MPI_COMM_WORLD);        // Communicator between processors
```



Ensuite, on va prendre la donnée reçue et on va l'insérer dans une grille 2D précédement créée, car ce sera plus facile de calculer l'équation de chaleur sur une grille 2D. En effet, dans l'énoncé, un code fonctionnant sur une grille 2D a été donné.

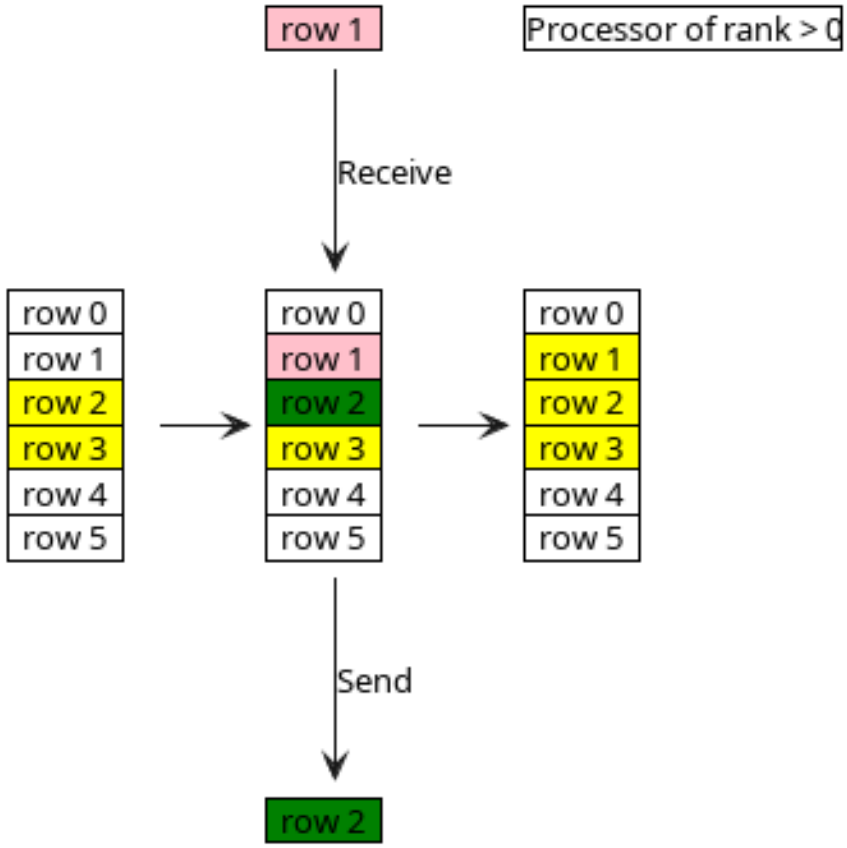
```
grid = put_into_2D(grid, recvbuf, nb_columns, rank, size);
```



Si on n'est pas le premier processeur, on envoie notre première ligne au processeur de rang inférieur, et on reçoit de ce processeur sa dernière ligne.

```
if (rank > 0) {
        MPI_Isend(
            grid[rank * nb_columns].data(),      // Data to send
            size,                                // Size of data
            MPI_DOUBLE,                          // Type of data
            rank - 1,                            // Rank of processor which receive data
            0,                                   // Tag
            MPI_COMM_WORLD,                      // Communicator
            &top_request                         // Communication request
        );
        MPI_Irecv(
            grid[rank * nb_columns - 1].data(), // buffer to receive data
```

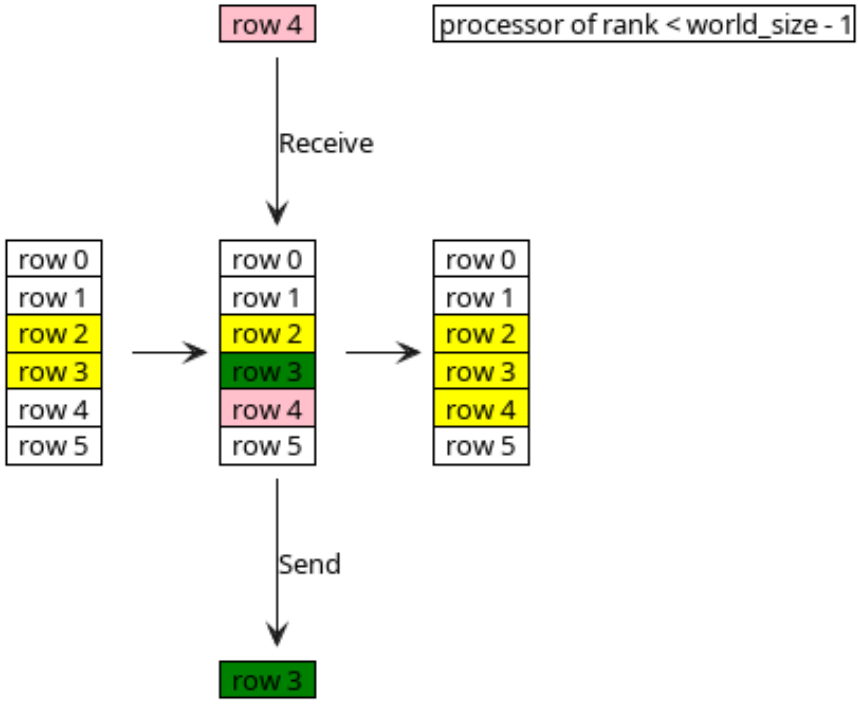
```
            size,                                   // Size of buffer
            MPI_DOUBLE,                             // Type of data
            rank - 1,                               // Rank of processor which send data
            0,                                      // Tag
            MPI_COMM_WORLD,                         // Communicator
            &top_request                            // Communication request
        );
}
```



Si on n'est pas le dernier processeur, on envoie notre dernière ligne au processeur de rang supérieur, et on reçoit de ce processeur sa première ligne.

```
if (rank < world_size - 1) {
        MPI_Isend(
            grid[(rank + 1) * nb_columns - 1].data(),   // Data to send
            size,                                       // Size of data
            MPI_DOUBLE,                                 // Type of data
            rank + 1,                                   // Rank of processor which receive
```

```
            0,                                          // Tag
            MPI_COMM_WORLD,                             // Communicator
            &bottom_request                             // Communication request
        );
        MPI_Irecv(
            grid[(rank + 1) * nb_columns].data(),       // buffer to receive data
            size,                                       // Size of buffer
            MPI_DOUBLE,                                 // Type of data
            rank + 1,                                   // Rank of processor which send data
            0,                                          // Tag
            MPI_COMM_WORLD,                             // Communicator
            &bottom_request);                             // Communication request
}
```



Ensuite, on attend que les échanges soient finis par ces lignes de code:

```
if (rank > 0) {
    MPI_Wait(&top_request, MPI_STATUS_IGNORE);
}
if (rank < world_size - 1) {
    MPI_Wait(&bottom_request, MPI_STATUS_IGNORE);
}
```

Enfin, on calcule notre équation de chaleur, grâce au code donné en annexe, et on met dans un vecteur ligne le résultat afin de préparer la réunion des données (car il faut que les données soient continues en mémoire pour que le Gather et le Scatter marchent...). J'ai simplement fait en sorte que chacun des processeurs calcule uniquement les lignes qui lui ont été attribué:

```
for (int i=rank*nb_columns;i<rank*nb_columns + nb_columns;i++)
```

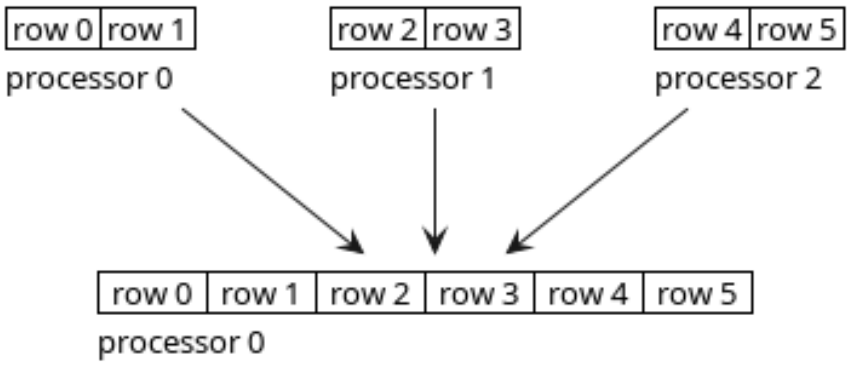
Pour ne pas calculer les bords, j'ai ajouté une condition demandant de ne pas calculer l'équation si on se trouve sur le bord de la grille.

```
if (i == 0 || j == 0 || j == size - 1 || i == size - 1) recvbuf[n] = grid[i][j];
```

Enfin, maintenant que chacun des processeur a calculé sa part de l'équation de chaleur et stocké le tout dans une grille 1 dimension, on réuni tous les calculs effectués dans un vecteur ligne grâce à un Gather. Cela s'exprime par cette ligne de code:

```
MPI_Gather(
    recvbuf.data(),          // Buffer of data to merge
    nb_columns * size,       // Size of buffer
    MPI_DOUBLE,              // Type of data
    grid_line.data(),        // Destination of all datas
    nb_columns * size,       // Size of a split
    MPI_DOUBLE,              // Type of data
    0,                       // Root processor which collect data
    MPI_COMM_WORLD           // Communicator
);
```

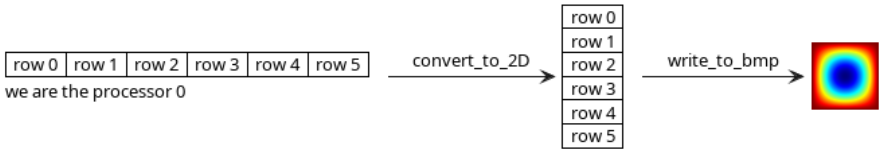
Ce qui, graphiquement, nous donne:



Et maintenant que nous avons récupéré toutes les données sous forme d'un vecteur ligne (toutes les données sont récupérées dans le processeur 0), on peut transformer ce vecteur ligne en une grille 2D grâce à la fonction *convert_to_2D* définie dans le fichier grid.cpp si on est le processeur 0, et ensuite créer une

image bmp à partir des données contenues dans la grille.

```
grid = convert_to_2D(grid_line, grid, grid.size());
write_to_bmp(
    size, // Size of the grid
    grid, // The grid to use
    time, // The iteration (just used to name the file of the output image)
    *min_element(grid_line.begin(), grid_line.end()), // Get min of the grid
    *max_element(grid_line.begin(), grid_line.end())  // Get max of the grid
);
```



1.3 Makefile

Voici à quoi ressemble mon Makefile:

```
CC = mpic++
CFLAGS = -g -Wall -c
OBJS = grid.o writer.o
.PHONY = clean cleanbmp

tp3: main.cpp $(OBJS)
    $(CC) $(OBJS) main.cpp -o tp3

grid.o: grid.cpp
    $(CC) $(CFLAGS) grid.cpp -o grid.o

writer.o: writer.cpp
    $(CC) $(CFLAGS) writer.cpp -o writer.o

clean:
    rm $(OBJS) tp3

cleanbmp:
    rm T_*

cleanoutput:
    rm ./err/* ./out/*
```

J'ai ajouté une méthode cleanbmp pour supprimer toutes les images que j'ai généré par mon programme, ainsi qu'une méthode cleanoutput pour supprimer tous les fichiers que j'ai créé dans baobab.

1.4 Run.sh

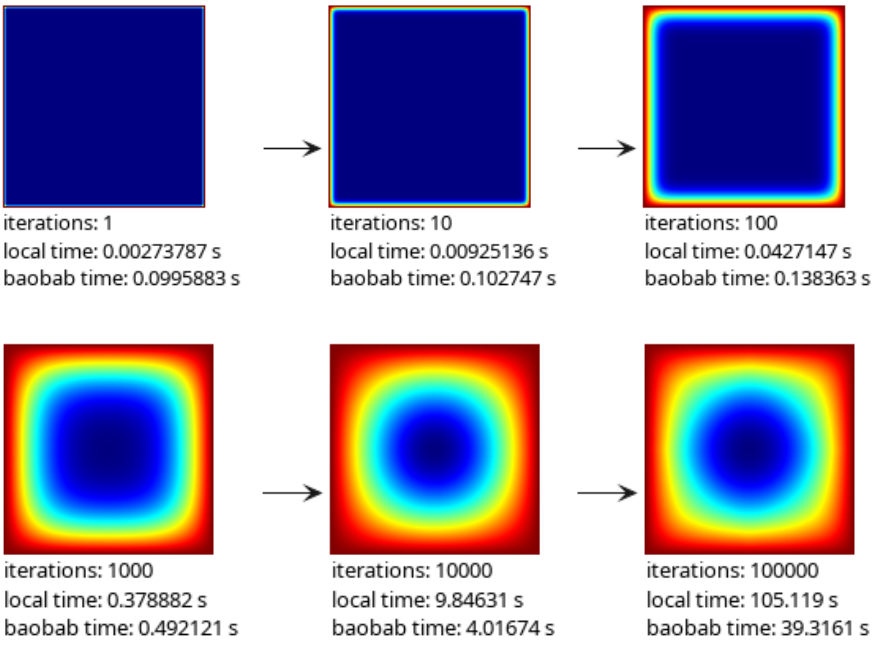
Voici le contenu de mon script run.sh. Pour changer le nombre de processeurs utilisés, il faut changer le –ntasks.

```
#!/bin/sh
#SBATCH --job-name Michel_TP2           # Permit us to find easily our job
#SBATCH --output ./out/Michel_TP2-out.o%j     # Outputs will be written here
#SBATCH --error ./err/Michel_TP2-err.e%j      # Errors will be written here
#SBATCH ----ntasks 1                    # Number of processors used
#SBATCH --partition debug-cpu           # Partition to use
#SBATCH --time 15:00                    # Maximum time execution

# Load modules for compiling and run program
module load foss
module load CUDA

echo $SLURM_NODELIST

# Compile program
make
# Run program. If the parameter is not given to execute an exercise, we print an error
echo "64x64 10^0 to 10^5 iterations"
srun --mpi=pmi2 ./tp3 64 1
srun --mpi=pmi2 ./tp3 64 10
srun --mpi=pmi2 ./tp3 64 100
srun --mpi=pmi2 ./tp3 64 1000
srun --mpi=pmi2 ./tp3 64 10000
srun --mpi=pmi2 ./tp3 64 100000
echo "128x128 10^0 to 10^5 iterations"
srun --mpi=pmi2 ./tp3 128 1
srun --mpi=pmi2 ./tp3 128 10
srun --mpi=pmi2 ./tp3 128 100
srun --mpi=pmi2 ./tp3 128 1000
srun --mpi=pmi2 ./tp3 128 10000
srun --mpi=pmi2 ./tp3 128 100000
echo "256x256 10^0 to 10^5 iterations"
srun --mpi=pmi2 ./tp3 256 1
srun --mpi=pmi2 ./tp3 256 10
srun --mpi=pmi2 ./tp3 256 100
srun --mpi=pmi2 ./tp3 256 1000
srun --mpi=pmi2 ./tp3 256 10000
srun --mpi=pmi2 ./tp3 256 100000
```

Ainsi, j'exécute mon programme pour des grilles de taille différente et pour un nombre d'itérations différent. Les 'echo' permettent de se retrouver dans la lecture du fichier d'output.

2 Tests et discussion sur les résultats obtenus

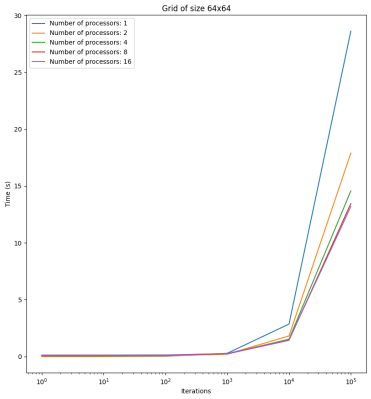
Tout d'abord, voici les résultats obtenus pour $10^0, 10^1, 10^2, 10^3, 10^4$ et 10^5 itérations pour une grille de taille 128×128. J'ai ici donné le temps d'exécution sur baobab et sur ma machine, pour 4 processeurs:

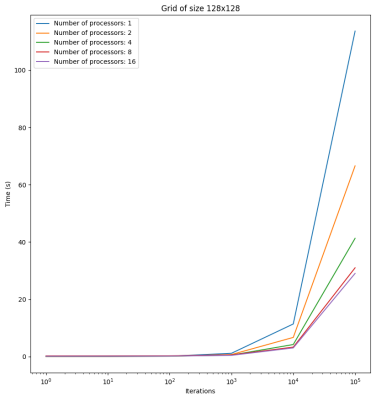


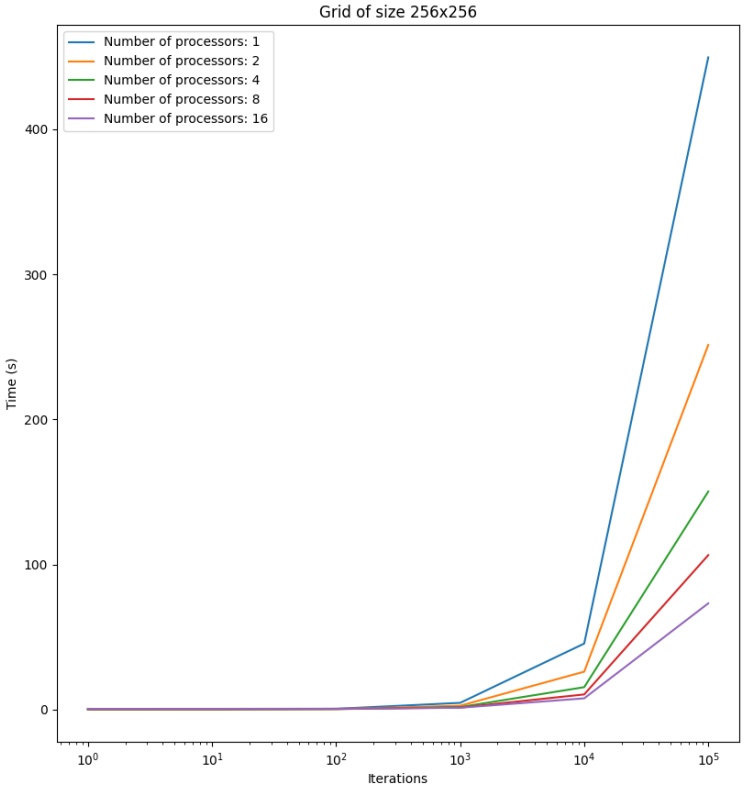
En observant les résultats, nous pouvons remarquer que le temps d'exécution sur ma machine est plus rapide que le temps d'exécution sur baobab pour un nombre d'itérations inférieur ou égal à 1000. Je pense que cela est dû à la communication entre les processeurs: sur baobab, les processeurs ne sont pas forcément sur le chip, tandis que dans mon ordinateur, les processeurs sont sur un même chip, donc la communication inter processeurs est très rapide, contrairement à baobab. Par contre, baobab possède des processeurs plus puissants que les miens. En effet, mes processeurs doivent avoir une fréquence d'horloge de 2 GHz tandis que sur baobab, on doit plutôt être à 4 GHz. C'est pourquoi, pour un gros calcul où les temps de communication deviennent négligeable, baobab est plus rapide, car il est plus lent que mon ordinateur pour communiquer entre les processeurs, mais plus rapide pour exécuter des calculs.

2.1 Exécution sur baobab

J'ai exécuté mon programme sur baobab en variant plusieurs paramètes (taille de la grille, itérations, nombre de processeurs...) et créé des graphes avec python grâce aux données obtenues. J'ai exécuté run.sh pour un nombre différent de processeurs


Grid of size 64x64
Number of processors: 1
Number of processors: 2
Number of processors: 4
Number of processors: 8
Number of processors: 16
Time (s)
Iterations

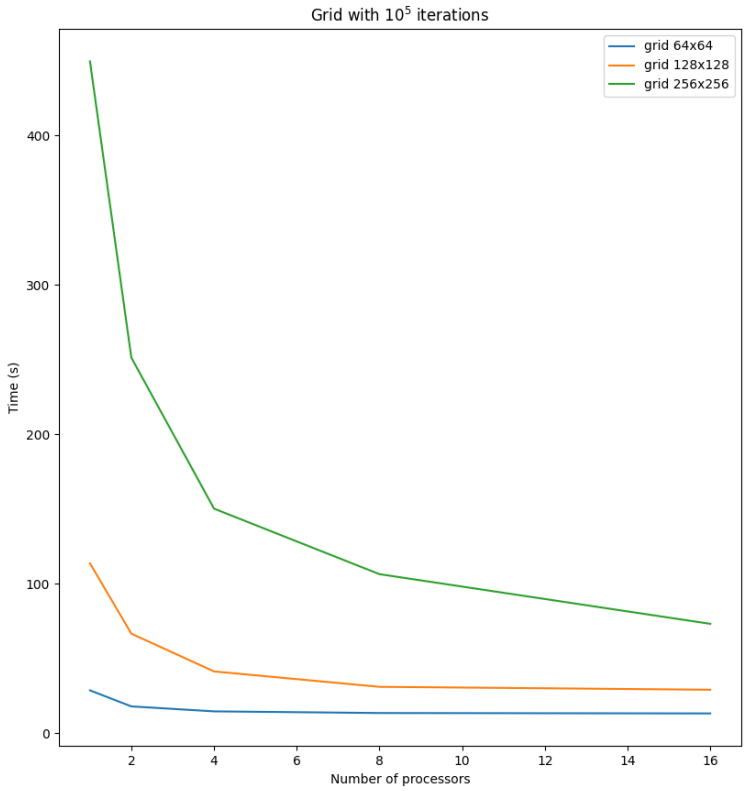

Grid of size 128x128
Number of processors: 1
Number of processors: 2
Number of processors: 4
Number of processors: 8
Number of processors: 16
Time (s)
Iterations


Grid of size 256x256
Number of processors: 1
Number of processors: 2
Number of processors: 4
Number of processors: 8
Number of processors: 16
Time (s)
Iterations

On peut constater que nous avons de grandes différences sur les temps d'exécutions pour un nombre élevé d'itérations, mais que sinon, les temps d'exécutions sont très proches. Cela est dû au fait que pour un nombre faible d'itérations, le temps de communication entre processeurs n'est pas négligeable, ce qui égalise les exécutions où on n'a pas beaucoup de processeurs, qui doivent alors calculer plus, et où on a beaucoup de processeurs, qui passent un temps non négligeable à se transmettre les informations.

C'est pour cela qu'on a décidé de prendre un nombre d'itérations élevé de 10^5 pour avoir un temps de communication entre processeurs négligeable comparé au temps de calcul.

Voici le graphe que j'obtiens pour un nombre fixé d'itérations:



J'ai l'impression que lorsqu'on fait fois 2 la taille de la grille, le temps d'exécution est multiplié par un facteur d'ordre de $O(n)\ldots$ Par exemple, on a pour 1

processeur un temps de 28.6201 pour une grille de 64×64, un temps de 113.64 pour une grille de taille 128×128 et un temps de 449.604 pour une grille de taille 256×256. De plus, pour 16 processeurs, on a un temps de 13.18 pour une grille de 64×64, un temps de 29.0374 pour une grille de taille 128×128 et un temps de 73.1513 pour une grille de taille 256×256 et on a environ $15 \times 2 = 30$ et $30 \times 2 = 60$. Le facteur est d'un peu plus de 2 tandis que pour un processeur, le facteur était de 4... On a donc une croissance relativement linéaire de notre temps d'exécution en fonction de la taille de la grille.

On peut remarquer que le temps de calcul ne varie plus à partir de 4 processeurs pour une grille de 64 et à partir de 8 processeurs pour une grille 2 fois plus grande, c'est à dire une grille de 128... Je pense que c'est parce que après, les processeurs mettent un temps certain pour discuter entre eux, et le temps de communication entre les processeurs devient non négligeable comparé au temps de calcul des processeurs.

Au fait, je pense qu'on pourrait faire un ratio pour savoir à partir de combien de processeurs notre problème ne va plus être exécuté avec des performances notables pour une taille de grille donnée. Pour l'instant, on peut remarquer que $\frac{64}{4} = 16$ et que $\frac{128}{8} = 16$. Donc on pourrait supposer que tant que les processeurs ont plus que 16 colonnes à calculer, on peut gagner des performances en ajoutant des processeurs au problème, et si les processeurs ont moins de 16 colonnes à calculer, on n'aura pas de gain notable de performance car le temps de communication est plus grand entre les processeurs et le temps de calcul est moins grand.

J'aurais bien aimé exécuter mon code avec 32 processeurs pour voir si ma supposition se vérifie également pour une grille de 256 car on aurait $\frac{256}{16} = 16$, et on pourrait observer que, comme je le suppose, on n'aurait plus de gain notable de performances...

Donc on peut conclure que notre problème doit tout d'abord être divisable entre plusieurs processeurs pour être parallélisable, ce qui peut déjà représenter beaucoup de difficultés. Et un autre défi est de trouver le juste compromis entre le gain de performances et le nombre de processeurs utilisés pour résoudre le problème car, comme on a pu le constater, multiplier le nombre de processeurs pour un problème donné va augmenter les performances, mais à un moment donné, les performances ne vont plus trop augmenter car les processeurs vont passer trop de temps à communiquer entre eux plutôt qu'à calculer. Donc il faut pour chaque problème savoir faire la balance entre le gain de performances et le nombre de processeurs utilisés si le problème est parallélisable.