Parallélisation avec MPI et OpenMP du Jeu de la Vie

Louis-César Pagès et Dorian Schwambach 1

I. Introduction

Présentation du jeu

Le jeu de la vie est un automate cellulaire conceptualisé par le mathématicien John Horton Conway, (Conway's Game of Life). C'est l'un des automates les plus connus. Sur un espace déterminé, une cellule, si elle existe, peut soit, mourir, survivre ou engendrer de nouvelles cellules voisines. Les règles d'évolution sont les suivantes : Si une cellule possède moins de 2 voisins elle meurt à la prochaine évolution, Si elle en possède 2 ou plus elle survie. Enfin si un espace sans cellule est entouré par au moins 3 cellules alors une cellule apparaîtra à cet endroit lors de l'évolution suivante.

Programme jeudelavie.c

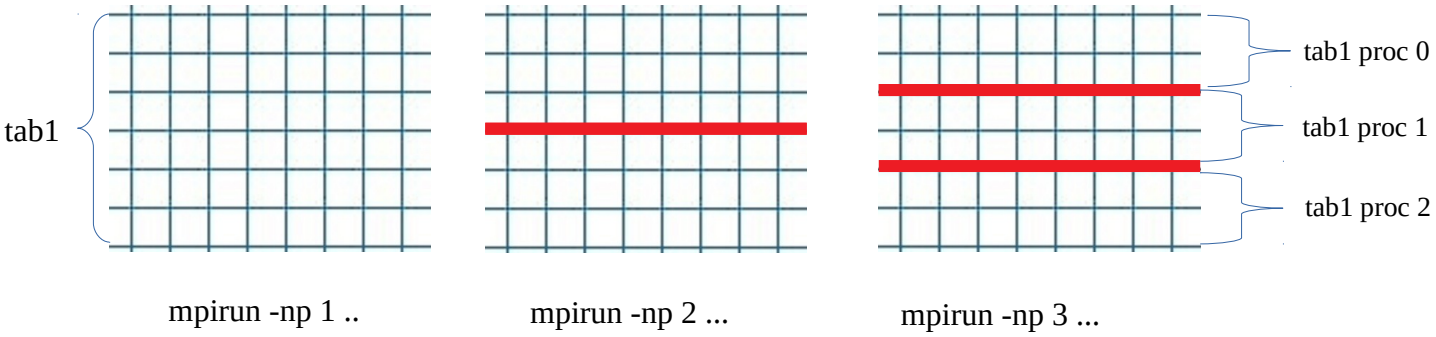
Dans le programme C, l'espace déterminé est représenté par une matrice où les cellules sont symbolisés par des 1 et des espaces vides par des 0. Par défaut le programme charge une matrice de 1200 lignes et 800 colonnes, puis calcule l'évolution pour un nombre d'itération donné ici 10001 et inscrit chaque sauvegarde (toutes les 1000 itérations) dans un fichier jdlv.out.

Le but de la parallélisation est de rendre le programme plus efficace lors du calcule de cette évolution. La partie OpenMP permet d'utiliser de décomposer le parcours de la matrice, par la fonction calculnouv, en plusieurs threads . Le framework MPI, rend possible la distribution entre processus de la matrice principale qui représente le jeu de la vie.

II. Parallélisation

Décomposition de la matrice

L'initialisation et le parcours de la matrice de 1200 lignes et 800 colonnes peuvent s'avérer coûteux en temps, surtout lorsqu'un unique thread est en charge de cela. La décomposition de la matrice principale permet de diviser le temps de parcours en nombre de processus crées avec MPI.



Pour parvenir à cela, on initialise une instance de partage MPI avec la fonction *MPI_Init()* ; On doit calculer, en fonction du rang de chaque processus la variable *max_row.* Cela permet d'établir la dernier ligne dans chaque matrice local tab1. Ainsi les sommes des lignes de la matrice local est égale au nombre de ligne de la matrice initiale.

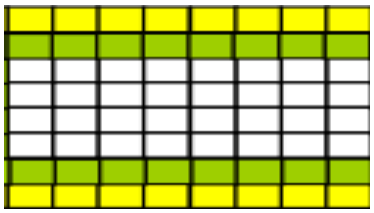
Chaque processus va ensuite initialiser leur matrice local avec la fonction *init(Tab tab).* Ils initialiseront *(HM/size)*LM* éléments si la *HM* peut être divisé par *size.*

Une solution alternative aurait été de laissé le processus 0 gérer l'initialisation de la matrice tab1 puis de la répartir entre les différents processus à l'aide d'un MPI_Scatter() ; Mais cela n'aurait pas

permis de réduire le temps de parcours de la matrice et le partage aurait coûtait en temps de communication..

Un fois la matrice divisée, il faut faire en sorte que l'évolution des cellules vivantes reste transparente pour chaque processus. Ainsi, si une cellule vivante arrive au bord d'une matrice et que ce dernier représente une frontière avec la matrice d'un processus voisin. La position de la cellule doit être récupérée par le voisin.

Pour ce faire, nous distinguerons 2 éléments importants dans les matrices.



Les lignes tab[1] et tab[maxrows-1] ici en vert doivent être envoyées à chaque itération au voisins précédant et suivant du processus.
Les lignes tab[0] et tab [maxrows] (les matrices ont une taille maxrows+2) ici en jaune, permettent de réceptionner les lignes envoyées par le processus précédant et suivant.

Grâce à ce modèle, un suivit dynamique de l'état de la matrice est possible entre les processus.

Chaque processus va ensuite rentrer dans le calcul de sa nouvelle matrice et cela pour un nombre *ITER* d'itérations :

```
88          calcnouv(t2, t1);
89          if(size > 1)envoyer_notif(t1);
90          if(size > 1)recevoir_notif(t1);
```

Décomposons cette partie :
Le processus calcul l'itération suivante de sa matrice, puis il envoie les lignes nécessaire à ses processus voisins à l'aide de la fonction *envoyer_notif(t1) ;*

```
216   int prev_rank = ((rank == 0) ?MPI_PROC_NULL:rank-1);
217   int next_rank = ((rank == size-1) ?MPI_PROC_NULL:rank+1);
218
219   MPI_Request request;
220   MPI_Isend(&(tab[1][0]), LM, MPI_CHAR, prev_rank, 0, MPI_COMM_WORLD,&request);
221   MPI_Isend(&(tab[max_rows-1][0]), LM, MPI_CHAR, next_rank, 0, MPI_COMM_WORLD, &request);
```

En fonction de son rang, il s'assigne un voisin précédent et suivant. Les cellules n'évoluant pas dans une matrice cyclique, le processus 0 et le dernier processus n'ont pas besoins d'avoir respectivement un voisin « précédant »et un « suivant ».

L'envoie est non bloquant pour ainsi permettre les processus d'attendre la réception des lignes nécessaire avec la fonction *recevoir_notif().*

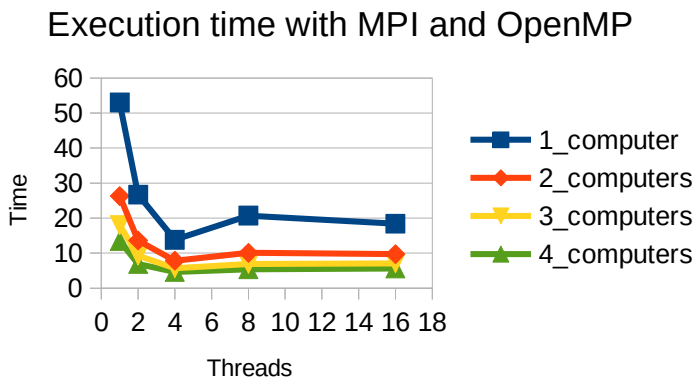
Enfin, chaque *sauv* itérations, on sauvegarde l'état local de la matrice.

```
97          for(int x=0 ; x< ((size == 1)?(max_rows+1):max_rows-1); x++)
98            for(int y=0 ; y<LM ; y++)
99              tsauvegarde[i/SAUV][x][y] = t1[x][y];
```

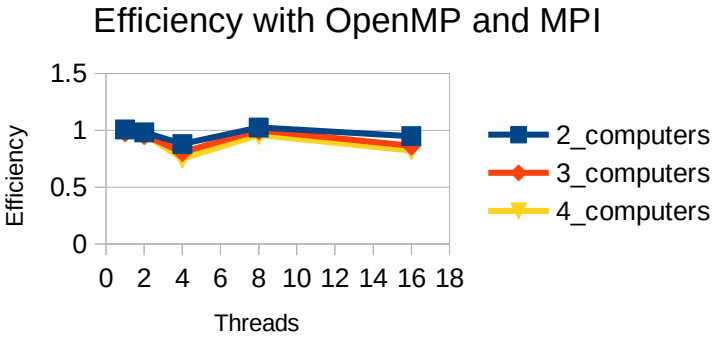
Si il y a un seul processus alors il faut enregistrer HM ligne (max_rows+1) si il y plus d'un processus alors HM/size (max_rows-1) lignes doivent être sauvegardées. Les deux dernières lignes d'une matrice étant ceux représentés par les deux premières lignes de la suivantes, elles ne doivent dont pas être comptabilisées.

III. Mesures

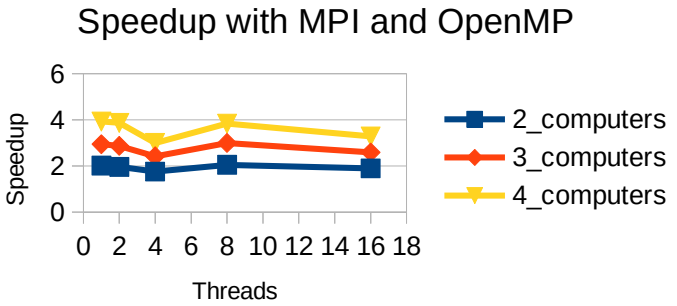
OpenMP et MPI



Les premières mesures concernent le temps de calcul du programme parallélisé utilisant OpenMp et Mpi. Les 2 framework ne pouvant fonctionner en même temps sur le même processeur (Machine) ; 1 unique processus est exécute par machine.

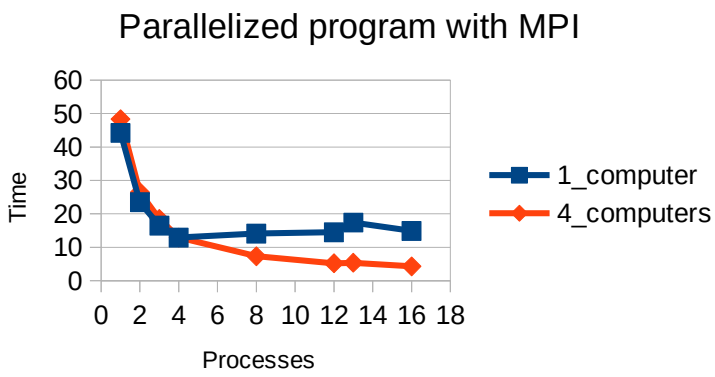


Ainsi, le programme est plus efficace exécuté avec 4 threads sur une seule machine qu'avec 16 threads. Cela s'explique par le nombre de coeurs que possède le processeur de la machine utilisée(ici un intel i5-6500, 4 coeurs, sans hyperthreading). Au delà de 4 threads, la création et l'utilisation de threads supplémentaire devient inefficace.



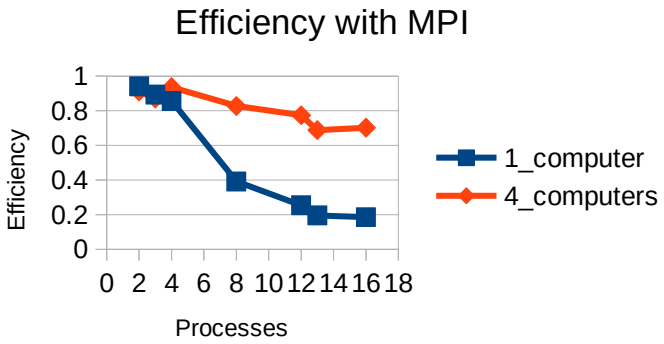
Globalement, l'efficacité de la parallélisation peut se résumer comme étant bon lorsque le programme est « distribué » sur plusieurs machine. 2 threads utilisées sur 4 machines (2*4 =8 threads en total)rend le programme plus rapide que simplement 2 threads sur 3 ou 2 machines (6 et 4 threads). Cette efficacité du de cette parallélisation diminue pour 4 threads à chaque machine et au-delà de 8 threads.

Uniquement MPI

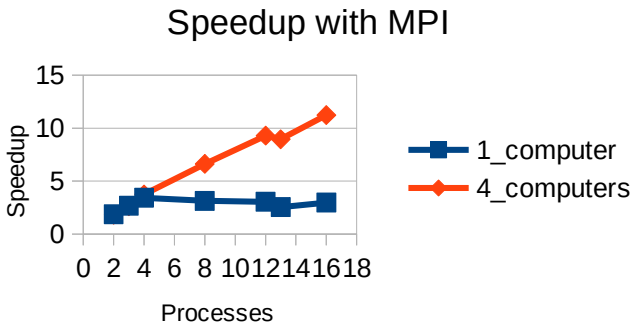


Les secondes mesures concernent le temps de calcul du programme parallélisé utilisant MPI.

Lorsque le programme est executé avec plus de 4 processus, on observe un gain de temps très considérable en utilisant 4 machines plutôt qu'une (+ de 2 fois plus rapide à partir de 8 processus). Cette différence saute aux yeux sur les graphiques de «speedup» et «d'efficiency». Cela s'explique par le fait que le processeur utilisé possède quatre coeurs et par la nature de MPI. En effet le temps sur une machine augmente à partir de 4 processus.



A noter que le temps d'exécution avec 4 machines et 1 processus est de 48s et est de 4,3s avec 4 machines et 16 processus. La parallélisation est fonctionnelle aussi bien avec des nombres pairs que des nombres impairs de processus. Au delà de 16 processus, la parallélisation n'est pas fonctionnelle en raison du découpage de la matrice.



IV. Conclusion

Le jeu de la vie est représente une évolution de cellules interdépendantes dans un espace délimité. A la différence de nombreux programmes parallélisable, ce n'est pas ici des tâches ou des calcules qui sont découpés en plusieurs processus mais bien le même programme avec les mêmes fonctions. Sa parallélisation consiste à découper cet espace délimité en nb processus, rendant ainsi les calcules moins coûteux.

Les processus doivent communiquer entre eux et se tenir informés à chaque itérations, cela afin de garantir l'intégrité de l'évolution en cours.

Les mesures établit, pour la partie uniquement MPI, démontrent une diminution presque linéaire du temps de calcul.