Introduction

Il y a 4 dossiers : Projet_original, Projet_omp, Projet_MPI et Projet_final.

Projet_original : les codes originals. Pour l'utiliser, entrez :
make all
./colonisation.exe

Projet_omp : les codes en mémoire partagée uniquement (utilisons OpenMP). Pour l'utiliser, entrez :
make all
OMP_NUM_THREADS=N ./colonisation.exe
(où N est le nombre de threads)

Projet_MPI : les codes en mémoire distribuée uniquement (utilisons MPI). Pour l'utiliser, entrez :
make all
mpirun -np N ./colonisation.exe
(où N est le nombre de processus)

Projet_final : les codes en mélangeant les deux parallélisations. Pour l'utiliser, entrez :
make all
mpirun -np N ./colonisation.exe
(où N est le nombre de processus)

Conflits mémoires

-**SDL**. Dans le programme en mémoire partagée, il y a des conflits mémoires quand nous affichons le graphe en utilisant SDL. Dans la fonction galaxie_renderer::render de fichier galaxie.cpp :

```
#pragma omp parallel for private(i,j) shared(data)
    for (i = 0; i < height; ++i )
        for (j = 0; j < width; ++j )
        {
            if (data[i*width+j] == habitee)
                #pragma omp critical
                rend_planete_habitee(j, i);
            if (data[i*width+j] == inhabitable)
                #pragma omp critical
                rend_planete_inhabitable(j, i);
        }
```

Ici il est obligatoire d'ajouter #pragma omp critical avant rend_planete_habitee(j, i) et rend_planete_inhabitable(j, i), et les deux fonctions utilisent SDL. Si nous supprimons #pragma omp critical, il y aura des erreur, car il y a des conflits mémoires quand plus que 1 thread utilise SDL. La phrase #pragma omp critical peut bien séparer les threads, les faisant utiliser indépendemment SDL, et évider des conflits mémoires.

-Cellule fantôme. Dans le programme en mémoire distribuée, nous découpons la grille de pixel par tranche dans la direction horizontale. De plus, nous ajoutons les cellules fantômes, donc il y a des cellules en common entre des processus différents. Quand le processus rank 0 reçoit des données des autres processus, les valeurs dans les cellules en common sont décidées par le processus qui envoie des donnée à la fin. Dans ce cas, nous n'avons pas besoin de ajouter des verrous car nous utilisons la dernière valeurs des cellules en common.

Accélérations

Nous ajoutons des codes pour calculer le temps de calcul et d'affichage pour chaque méthode, en calculant la moyenne des 100 boucles. Les résultats sont ci-dessous :

original :

calcul (ms)	31.153
affichage(ms)	1.112

Tableau 1 - le temps de calcul et d'affichage original

en mémoire partagée uniquement :

Nbres de threads	2	3	4	5	8	12	20
calcul (ms)	40.757	47.514	25.103	29.252	28.774	29.246	28.430
affichage(ms)	1.138	2.425	1.060	1.748	2.543	3.025	3.657

Tableau 2 - le temps de calcul et d'affichage en mémoire partagée uniquement

en mémoire distribuée uniquement :

Nbres de processus	2	3	4	5	8	12	20
calcul (ms)	72.860	52.581	35.285	39.621	19.161	12.265	19.225
affichage(ms)	1.141	2.364	2.981	2.796	2.403	6.110	7.801

Tableau 3 - le temps de calcul et d'affichage en mémoire distribuée uniquement

en mélangeant les deux parallélisations :

Nbres de processus	2	3	4	5	8	12	20
calcul (ms)	35.648	29.822	25.648	34.793	21.928	18.609	23.930
affichage(ms)	3.434	5.570	6.247	15.812	11.337	12.060	14.736

Tableau 4 - le temps de calcul et d'affichage en mélangeant les deux parallélisations

Nous trouvons que tous les trois méthodes peuvent accélérer le programme.

En mémoire partagée uniquement, il y a de l'accélération depuis un nombre de threads 4, et elle est maximale quand le nombre est 4. (Le test est fait sur un ordinateur de 4 CPU.)

En mémoire distribuée uniquement, il y a de l'accélération depuis un nombre de processus 8, et elle est maximale quand le nombre est 12 : le temps de calcul est seulement 12.265ms, mais celui d'affichage est 6.110ms, plus long que l'original.

En mélangeant les deux parallélisations, il y a de l'accélération depuis un nombre de processus 3, et elle est maximale quand le nombre est 12 : le temps de calcul est seulement 18.609 ms, mais celui d'affichage est 12.060ms, plus long que l'original.

En conclusion, nous obtenons la meilleure accélération en mémoire distribuée uniquement et en utilisant 12 processus. Par ailleurs, la prolongation de temps d'affichage en MPI en augmentant le nombre de processus est étrange, car l'affichage de graphe n'utilisent pas du tout MPI. Cela peut être une direction d'étude au futur...