Rapport de projet : Rania BEN KMICHA

IL s’agit de la parallélisation d'un programme qui simule stochastiquement la co–circulation d’un virus (de la grippe par exemple) et d’un second agent pathogène, en interaction dans une population humaine virtuelle.

Nos outils sont la parallélisation par MPI en mémoire distribuée et OpenMP en mémoire partagée. De plus l'affichage a été fait de manière asynchrone dans le but d'améliorer les performances de la simulation.

*Configuration de mon PC :

En utilisant la commande « lscpu », mon pc a 4 cœurs physiques et 2 threads par cœur donc au total , 8 threads (si le multithreading est activé).

*Compilation

La compilation est faite avec la commande « make all » dont les fichiers make file sont fournis avec le sujet.

J’ai utilisé deux fichiers sous les noms « Make_linux.inc » et « Make_linux.inc » pour distinguer la compilation en MPI et en OpenMP.

*Les fichiers sources :

simulation.cpp (le programme séquentiel)

simulation_sync_affiche_mpi.cpp

simulation_async_affiche_mpi.cpp

simulation_async_omp.cpp

simulation_async_mpi.cpp

Analyses des résultats :

*Simulation.cpp :

On fera le calcul du temps par pas de temps(un jour) :

Temps de simulation avec affichage (1)	Temps de simulation sans affichage	Temps d'affichage
0.048	0.025	0.021

Speed up pour le cas 1 : S =1

Pour le code en séquentiel, on remarque que l'affichage représente 43% du code ce qui nous incite alors à la parallélisation.

*Parallélisation affichage contre simulation(simulation_sync_affiche_mpi.cpp) :

Speed up : S=ts/tp

Temps de simulation (sans affichage)	Temps d'affichage	Speed up pour la simulation
0.0295	0.03	1.627

En parallélisant l'affichage et la simulation en 2 processus différents(proc 0 responsable de l'affichage et proc 1 responsable de la simulation) et en envoyant les données en mode synchrone, on remarque que le temps mis par la simulation est légèrement supérieur a celui en séquentiel qui est du bien évidement à l'instruction d'envoi des données au proc d'affichage.

On a aussi un gain de temps puisque le speed up est de 1.67.

L'envoi et la réception de données limite aussi la performance de proc plus rapide .

*Parallélisation affichage asynchrone contre simulation (simulation_async_affiche_mpi.cpp) :

Temps de simulation (sans affichage)	Temps d'affichage	Speed up pour la simulation
0.027	0.025	1.77

Lorsqu'on a utilise le mode asynchrone pour l'envoi et la réception des données entre les deux procs , on constate que le temps mis par le proc de simulation diminue et cela s'explique par le fait que le proc de simulation n'est pas obligé d'envoyer les données chaque jour a proc 0 ce qui lui rend plus rapide.

On remarque aussi que l'affichage de la simulation plus lente en termes de fréquence (à l'œil nu)

*Parallélisation OpenMP (simulation_async_omp.cpp) :

Pour un même nombre d'individu constant pour chaque thread:

Nombre de threads	Temps de simulation (sans affichage)	Speed up pour la simulation
2	0.013	3.69
4	0.0065	7.38
8	0.0047	10.21

Pour un nombre total constant d'individus :

Nombre de threads	Temps de simulation (sans affichage)	Speed up pour la simulation
2	0.016	3
4	0.0094	4.38

En modifiant maintenant la parallélisation en utilisant l' OpenMP, on remarque le temps de calcul et simulation en parallèle s'améliore beaucoup jusqu'obtenir un speed up de 10 en mode « static » et qui diminue en augmentant le nombre de threads.

On constate aussi que l'utilisation d'OpenMP ne garantit pas un résultat »identique » au code en séquentiel puisque la génération aléatoire utilisée dans individu.cpp et grippe.hpp va dépendre bien évidement de l'ordre des individus dans le tableau population et OpenMP ne respecte pas l'ordre utilisé dans le séquentiel.

*Parallélisation MPI de la simulation (simulation_async_mpi.cpp) :

Nombre de proc	Temps de simulation(sans affichage)	Temps d'affichage	Speed up
2	0.029	0.024	1.65
3	0.025	0.027	1.92
4	0.023	0.034	2.08

En parallélisant la partie de la simulation à l'aide de MPI en mode asynchrone sur plusieurs processus(pas un seul proc) en utilisant MPI_Comm_split ,on a toujours le problème de l'ordre des individus qui ne garantit pas les mêmes résultats du code en séquentiel.

*Bilan :

Ce projet est donc une continuité et application du notre cours avec l'utilisation des nouvelles fonctions comme MPI_Iprobe et MPI_Comm_Split pour MPI.

On remarque que le mode asynchrone nous garantit la rapidité de calcul .

De plus, l'utilisation de MPI_Iprobe nous facilite la communication entre le proc de simulation et le proc d'affichage en mode asynchrone pour détecter le temps d'envoi des données.

En outre, la parallélisation surtout à l'aide d'OpenMP n'est performante que lorsqu'on est dans le cas de CPU Bound et que ça reste une meilleure option dans notre cas de paralléliser en mémoire partagé qu'en mémoire distribuée puisqu'on a obtenu un speed up de 10 .

Le mode synchrone reste une solution si on a besoin d’afficher les données à jour mais ça risque de limiter le temps de simulation.