PROJET : compte rendu_IN203

Mon ordinateur :

4 coeurs et 8 threads : 8 threads au maximum peuvent être vraiment exploités

J'ai eu quelques soucis de compilation voici la commande pour compiler avec mpi que j'ai utilisé

```
mpic++ -std=c++17 `sdl2-config --cflags` -O3 -march=native -Wall graphisme/src/SDL2/sdl2.o
graphisme/src/SDL2/geometry.o graphisme/src/SDL2/window.o graphisme/src/SDL2/font.o
graphisme/src/SDL2/event.o graphisme/src/SDL2/texte.o graphisme/src/SDL2/image.o
graphisme/src/SDL2/formated_text.o agent_pathogene.o grille.o individu.o simulation.cpp -o
simulation.exe  `sdl2-config --cflags` -lSDL2_ttf -lSDL2_image  `sdl2-config --libs`
```

1. Mesure du temps

le temps passé dans la simulation par pas de temps sans affichage	0.0174226
le temps passé dans la simulation par pas de temps avec affichage	0.0618357
temps affichage par pas de tempss	0.03

Conclusion : On peut dire que l'affichage ralentie énormément le programme.

2. Parallélisation affichage contre simulation

```
(3*m_dim_x*m_dim_y+3) : le nombre d'entier dans un élément de struct grille
3*dimx*dimy = taille de m_statistique
```

```
ligne 127,128
////////////envoie au processus 0 de jours_écoulés//////////////////////////////

            MPI_Send(&jours_écoulés, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            MPI_Send(grille.getStatistiques().data(),3*largeur_grille*hauteur_grille,MPI_INT,0,0, MPI_COMM_WORLD);

ligne 245

//recevoir grille et jours_écoulés
            MPI_Recv(&jours, 1, MPI_INT, 1,0, MPI_COMM_WORLD, &status);
            MPI_Recv(grille_bis.getStatistiques().data(),3*largeur_grille*hauteur_grille,MPI_INT,1,0, MPI_COMM_WORLD, &status);
```

Temps passé pour calculer la simulation en moyenne (donc ce qui se passe dans le processus 1)**.**

Temps passé pour calculer la simulation en moyenne	**0.0645959 secondes**

Le calcul est plus long que ce qui a été mesuré à la question 1.

Comment expliquer cela : Le temps en plus est surement dû au temps nécessaire pour réaliser les envoie (qui sont ici bloquant : on est obligé d'attendre la réception par processus 0 pour avancer)

3. Parallélisation affichage asynchrone contre simulation

Le processus 0 envoie un message quand il est prêt ou non. On trouve un résultat satisfaisant puisque le calcul pour la simulation est équivalent à ce qui a été mesuré en séquentiel.

Temps moyen en simulation	0.0168688

4. Parallélisation OpenMP

Le problème avec juste ajouter “#pragma omp parallel for schedule(static)” : quand on fait appel au fonction personne.estContaminé(grippe ou agent) on fait appel aux fonctions nombreJoursSymptomatique() et nombreJoursDincubation() qui elles font appel à m_générateur_symptomatique(m_moteur_stochastique). Selon l’ordre dans lequel les individus sont appelés les valeurs retournées par la fonction ne sont pas les mêmes.

Donc forcément si les jours d’incubation et symptomatiques diffèrent par rapport à l’exécution séquentielle la propagation ne sera pas la même et ça va modifier le décompte des cas.

note:

std::default_random_engine (grain) : générateur aléatoire initialiser avec seed. Si on exécute avec le même seed on obtient la même valeur → pour m_moteur_stochastique

Mais ce n’est pas le cas pour

```
r std::gamma_distribution<double> m_générateur_incubation;
t= m_générateur_incubation(m_moteur_stochastique);
```

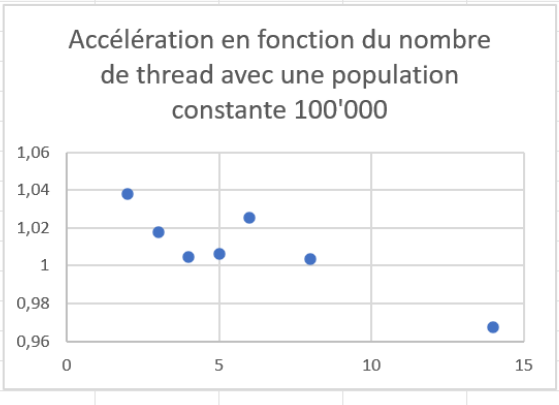
Pour un m_moteur_stochastique initialiser avec le même grain, le générateur_incubation renvoie les mêmes valeurs dans le même ordre mais différentes les unes des autres !

- Première tentative : j’ai essayé d’enregistrer dans un tableau les indices des personnes dans “population” à qui ont doit appliquer la fonction estContaminé. Pour ensuite appeler cette fonction en dehors de la parallélisation openMP. En s’assurant que le tableau est parcouru comme en séquentiel (pour une personne donné on regarde pour la grippe d’abord puis l’agent pathogène). Mais ça revient juste à séquentialiser une partie de la boucle for ce qui revient à utiliser la clause ordered de openmp bien plus simple à utiliser.

Tentative avec OPENMP ORDERED.

Pour une population constante de 100’000

Nombre de threads	2	3	4	5	6	8	14
Vitesse moyenne de simulation	0,016198	0,0165226	0,0167365	0,0167075	0,016394	0,0167559	0,0173765
Accélération	1,03792444	1,01753356	1,00452902	1,00627263	1,02551543	1,00336598	0,96753086



On remarque une diminution de l’accélération avec le nombre de thread de façon général jusqu’à 4 threads. Puis on a une légère remontée par passage à l’hyperthreading. Mais l’accélération est infime quand il y en a une du coup difficile de vraiment analyser ces résultats.

C’est surement du au fait que nous réalisons dans la boucle for une grande partie des opérations en séquentiel (avec la clause ordered) afin d’obtenir des résultats cohérents avec le fonctionnement purement séquentiel.

Accélération avec un nombre constant de population par thread de 50'000							
Nombre de thread	2	3	4	5	8	10	12
Population totale	100'000	150'000	200'000	250'000	400'000	500'000	600'000
Vitesse moyenne	0,016198	0,0245022	0,0357963	0,041902	0,072406	0,0946019	0,114815
Accélération	1,03792444	0,68615471	0,46966586	0,40122906	0,23219485	0,1777163	0,14642947

Le speed up diminue avec le nombre de thread, en effet même si la quantité de tâches effectuée par chaque thread apparait similaire, la communication et la gestion des différents threads par le processeur ralentissent énormément les calculs. Ce n'est intéressant qu'avec 2 threads.

5.Parallélisation MPI de la simulation

⚠️ Je n'ai pas réussi à paralléliser de façon à obtenir à nouveau les mêmes résultats. J'ai le même problème que pour la partie OPENMP que je n'ai pas réussi à régler cette fois ci.

QUE AVEC MPI

QUESTION 5 MPI				
	Nombre d'individu constant 100'000			
Nombre de processus	2	3	5	6
Vitesse moyenne de simulation	0,0161576	0,0124479	0,0148827	0,0164526
Accélération	1,04401644	1,35515227	1,13345025	1,02529691

Observation : Le maximum est autour de 4 threads au vue des spécificités de l'ordinateur ça paraît cohérent. Mais comme pour OpenMP l'accélération est infime.

Accélération avec un nombre constant de population par processus de 50'000					
Nombre de processus	3	5	7	8	9
Population totale	100'000	200'000	300'000	350'000	400'000
Vitesse moyenne	0,0124479	0,0153978	0,0419141	0,0592038	0,079769
Accélération	1,35061336	1,09186377	0,40111323	0,28397333	0,21076233

Parallélisation finale

	Population constante de 100'000 et deux threads par processeur				
Nombre de processus	2	3	5	6	
Vitesse moyenne de simulation	0,0161768	0,0147921	0,0213683	0,0164526	
Accélération	1,04277731	1,14039251	0,78943107	1,02529691	
	Population constante de 100'000 et trois threads par processeur				
Nombre de processus	2	3	5	6	
Vitesse moyenne de simulation	0,0163938	0,0158051	0,0371741	0,0519412	
Accélération	1,02897437	1,06730106	0,4537783	0,32476724	

On se doute bien qu'au vu des tests précédents que 2 threads par processeur sera le meilleur paramétrage. On vérifie tout de même en regardant avec 3 threads pas processeur.

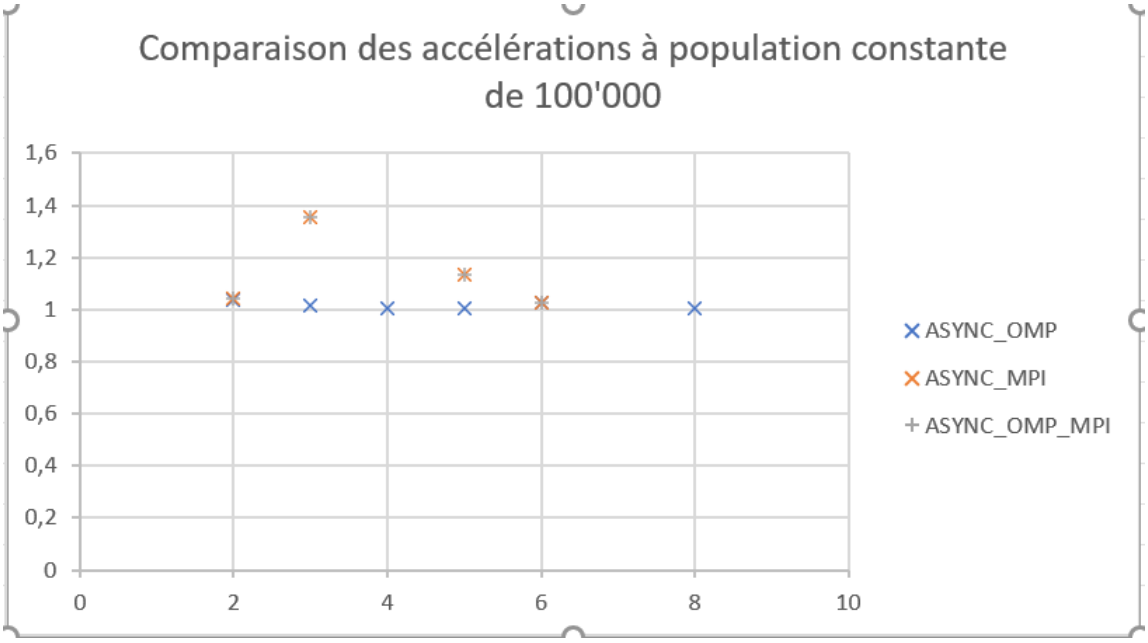
	Population constante de 50'000 par processus avec 2 threads par processus				
Nombre de processus	3	5	6	7	8
Population totale	100'000	200'000	250'000	300'000	350'000
Vitesse moyenne	0,0135058	0,022663	0,0432632	0,0579229	0,0787569
Accélération	1,24482074	0,74183912	0,3886051	0,29025308	0,21347082

	Population constante de 50'000 par thread avec 2 threads par processus		
Nombre de p	2	3	5
Population t	100'000	200'000	400'000
Vitesse moye	0,017863	0,0354322	0,0723751
Accélération	0,94118009	0,47449213	0,23229398

Bilan

- Sur les résultats obtenus

A population constante :



La parallélisation avec OpenMP n'a pas été très utile puisqu'on a les mêmes résultats que en séquentiel.

On voit que le paramétrage le plus performant est avec 3 processus et 2 threads par processus.

- Sur les connaissances acquises

Même si au final les lignes codées ne sont pas si importantes en terme de lignes, l'impact sur le processus est tout de même intéressant ne serait ce que pour l'affichage asynchrone sans parallélisation de la simulation.

Surtout, on se rend compte que lorsque le résultat dépend de l'ordre dans lequel on exécute nos taches, la parallélisation devient tout de suite plus ardue et surtout beaucoup moins intéressante.

J’ai énormément appris grâce à ce projet qui m’a vraiment forcé à m’interroger davantage sur le fonctionnement de la parallélisation (comment lier MPI et OpenMP). Et même si ce n’était pas le but premier du projet j’ai énormément appris sur la génération des nombres aléatoires (chose que j’avais beaucoup de mal à comprendre auparavant). Je suis tout de même déçue de ne pas avoir réussi à réellement accélérer les calculs.