Projet : Epidémiologie

0 lscpu

```
simon@DESKTOP-OSQNMVN: $ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           36 bits physical, 48 bits virtual
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   142
Model name:              Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Stepping:                12
CPU MHz:                 2112.000
CPU max MHz:             2112.0000
BogoMIPS:                4224.00
Hypervisor vendor:       Windows Subsystem for Linux
Virtualization type:     container
Flags:                   fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr s
                         se sse2 ss ht tm pbe syscall nx pdpelgb rdtscp lm pni pclmulqdq dtes64 est tm2 ssse3 fma cx16 xtpr
                         pdcm pcid sse4_1 sse4_2 movbe popcnt aes xsave osxsave avx f16c rdrand hypervisor lahf_lm abm 3dnow
                         prefetch fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt ibrs i
                         bpb stibp ssbd
```

1 Mesure du temps

On mesure le temps passé dans la simulation par pas de temps (un jour) avec et hors affichage et le temps passé à affichage, les résultats sont présenté dans la figure ci-dessous:

```
simon@DESKTOP-OSQNMVN:/mnt/p/Cours ENSTA/IN203/IN203_TD/IN203/Projet/sources$ cat Temps_default.dat | head -n 20
# jours_écoulés          tempsAvecAffichage      tempsHorsAffichage      tempsAffichage
1       0.121542        0.0199671       0.101575
2       0.0981656       0.0255751       0.0725905
3       0.0845563       0.0302888       0.0542675
4       0.0838299       0.0215805       0.0622494
5       0.0860733       0.0202987       0.0657746
6       0.0840344       0.0240486       0.0599858
7       0.0764598       0.0207187       0.0557411
8       0.0714361       0.0228525       0.0485836
9       0.0801958       0.02069 0.0595058
10      0.0787923       0.0200219       0.0587704
11      0.0773923       0.0207451       0.0566472
12      0.077165        0.0205147       0.0566503
13      0.0703546       0.0269652       0.0433894
14      0.0735012       0.0213476       0.0521536
15      0.0827654       0.0205567       0.0622087
16      0.0813059       0.0201818       0.0611241
17      0.0826943       0.0214524       0.0612419
18      0.0718881       0.0201386       0.0517495
19      0.0875747       0.0245545       0.0630202
```

Selon la figure, on constate que le temps passé à l'affichage occupe la majeure partie du temps total, qui est double par rapport au temps hors affichage.

En utilisant le programme `cal_mean_time.cpp` qu'il peut être compilé par `g++ cal_mean_time.cpp -o cal_mean_time.exe`, on peut obtenir le temps moyen de la simulation par pas de temps (unité: s):

$$t_0 = 0.0730809$$

2 Parallélisation affichage contre simulation

On parallélise le code sur deux processus. L'un s'occupe de l'affichage en synchrone (processus 0), et l'autre (processus 1) de la simulation. Le processus 1 envoie le tableau `grille.getStatistiques()` au processus 0 lorsque des statistiques pour les cases de la grille sont mise à jour, et puis le processus 0 affiche ce tableau.

On exécute le programme `simulation_sync_affiche_mpi.exe` par `mpiexec -np 2 ./simulation_sync_affiche_mpi.exe`, et puis on calcule le temps total moyen:

$$t_{sam} = 0.053104$$

Cette valeur est approximativement égal au temps passé à affichage qu'on a obtenue dans la première partie. En effet, on a deux processus exécutés en même temps et le temps total pour la simulation est le temps passé pour le processus le plus lent.

$$speed\ up = 0.0730809/0.053104 = 1.376$$

3 Parallélisation affichage asynchrone contre simulation

3.1 Stratégie

Dans le processus 0 (qui s'occupe de l'affichage) , on envoie asynchrone un message après avoir finit l'affichage (à l'aide de `MPI_Isend` & `MPI_Wait`) .

Dans le processus 1, on test si le processus 0 a envoyé ce message (`MPI_Iprobe`) , si oui, on envoie les donnés (c'est-à-dire le tableau `grille.getStatistiques()`) au processus 0; sinon, on continue.

3.2 Résultat

On exécute le programme `simulation_async_affiche_mpi.exe` par `mpiexec -np 2 ./simulation_async_affiche_mpi.exe`, et puis on calcule le temps total moyen:

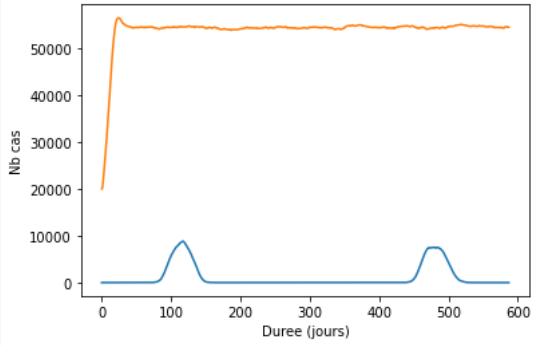
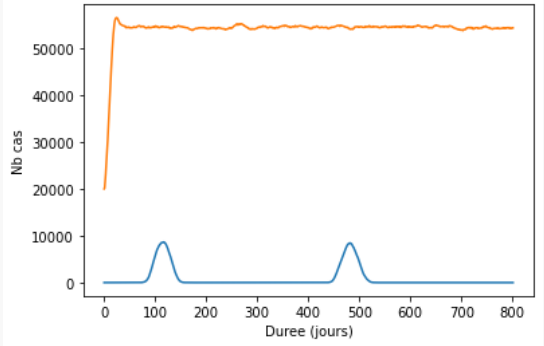
$$t_{aam} = 0.0206089$$

Cette valeur est approximativement égal au temps hors affichage qu'on a obtenue dans la première partie. En effet, dans le cas de l'affichage asynchrone, le temps passé à l'affichage n'influence pas le temps total de la simulation.

$$speed\ up = 0.0730809/0.0206089 = 3.546$$

4 Parallélisation OpenMP

- Les résultats pour la courbe de sortie

	
la simulation d'origine	parallélisation OpenMP

On exécute le programme `simulation_async_omp.exe` par `mpiexec -np 2 ./simulation_async_omp.exe`.

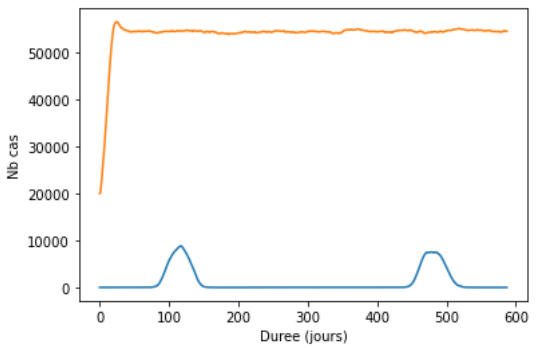
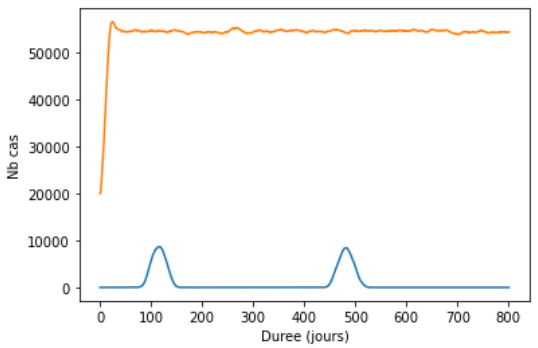
On parallélise le boucle qui est pour le mise à jour de la condition des individus en ajoutant `# pragma omp parallel for schedule(static)` avant le boucle. Dans ce cas, la variable `population`, qui est un tableau des individus, est partagée entre les threads.

On constate que le temps total est minimal quand le nombre de processus est 2, et pour un nombre d'individus global constant, on a:

$$speed\ up = 0.0730809/0.0133985 = 5.454$$

5 Parallélisation MPI de la simulation

- Les résultats pour la courbe de sortie

	
la simulation d'origine	parallélisation MPI

On exécute le programme `simulation_async_mpi.exe` par `mpiexec -np 4 ./simulation_async_mpi.exe`.

On utilise `MPI_Allreduce` pour récupérer les donnés dans le groupe pour la simulation. L'idée est qu'on distribue les individus dans les processus, et dans chaque processus, on met à jour la situation des individus, et puis des statistiques pour les cases de la grille. Enfin, nous intégrons ces données et transmettons les données mises à jour à chaque processus.

On constate que le temps total est minimal quand le nombre de processus est 4 dans mon ordinateur, et pour un nombre d'individus global constant, on a:

$$speed\ up = 0.0730809/0.0135698 = 5.386$$

5.1 Parallélisation finale

On exécute le programme `simulation_async_mpi_omp.exe` par `mpiexec -np 4 ./simulation_async_mpi_omp.exe`.

On constate que le programme n'est pas autant efficace qu'avant quand on le teste sur un ordinateur.

5.2 Bilan

- Nous pouvons diviser une grande tâche en plusieurs petites tâches pour un traitement parallélisé, et nous pouvons diviser ces petites tâches en tâches plus petites pour accélérer davantage le traitement.
- Nous pouvons accélérer notre programme à l'aide de l'affichage asynchrone.
- Plus précisément, avec ce projet, nous savons comment utiliser `MPI_Iprobe` pour tester si on a reçu un message, et comment utiliser `MPI_Comm_split` pour diviser des processus en groupes différents.

- En fin, nous savons aussi le théorie de la co–circulation d’un virus et d’un second agent pathogène, en interaction dans une population humaine.