Projet

Ziqi MA

0. lscpu

Le paramètre de notre cpu est ci-dessous:

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Stepping:                        9
CPU MHz:                         2700.000
CPU max MHz:                     3100,0000
CPU min MHz:                     400,0000
BogoMIPS:                        5399.81
Virtualization:                  VT-x
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                        512 KiB
L3 cache:                        3 MiB
NUMA node0 CPU(s):               0-3
```

1. Mesure du temps

Dans l'algorithme, on ajoute les codes pour mesurer du temps.

Dans le console, on tape:

```
$ make simulation.exe

$ ./simulation.exe
```

Les résultats sont dans le ficher `temps_init.dat`, on le copie ci-dessous:

```
Jours_ecoules    Temps_sans_affichage    Temps_avec_affichage    Temps_affichage
0       0.0375367       0.0676632       0.0301265
1       0.027845        0.048119        0.0202741
2       0.0275449       0.0490533       0.0215084
3       0.0271713       0.04758 0.0204087
4       0.028013        0.0453287       0.0173157
5       0.0276345       0.0475924       0.0199579
6       0.0278825       0.0507573       0.0228748
7       0.0287542       0.0477289       0.0189747
8       0.0279209       0.0479931       0.0200722
9       0.025959        0.0461408       0.0201817
10      0.0277346       0.0476554       0.0199208
11      0.0343384       0.0618636       0.0275253
12      0.0283261       0.0464956       0.0181695
13      0.0279005       0.0482516       0.0203511
14      0.0269366       0.0491439       0.0222073
15      0.031425        0.0511599       0.0197349
16      0.0253197       0.0440511       0.0187315
17      0.0241602       0.0410513       0.0168911
18      0.0235486       0.0407503       0.0172017
19      0.0233564       0.0401368       0.0167803
20      0.0246677       0.0416727       0.017005
21      0.0238191       0.0406445       0.0168253
22      0.0238949       0.0417371       0.0178422
23      0.0253863       0.0431101       0.0177237
24      0.0244172       0.0427433       0.0183262
25      0.0245632       0.0423081       0.0177449
26      0.0252153       0.0430035       0.0177882
27      0.0246415       0.0438739       0.0192325
28      0.0251333       0.0434271       0.0182937
29      0.026235        0.0451742       0.0189392
```

on choisit les premier 30 processus pour calculer le temps moyen.

	Temps_sans_affichage	Temps_avec_affichage	Temps_affichage
Moyen	0.026909387	0.046540357	0.01963097

Alors, on a constaté que le temps passé pour simuler le statistique est le temps passé à l'affichage sont similaires, donc dans les parties suivantes, on ne pourra pas négliger le temps de l'affichage.

2. Parallélisation affichage contre simulation

On ajoute des parties de communication mpi dans le code.

```
......
if (rank ==1){
            ......
            auto const &data_passe = grille.getStatistiques();
            MPI_Send(&jours_ecoules,1,MPI_INT,0,655,MPI_COMM_WORLD);

 MPI_Send(data_passe.data(),3*data_passe.size(),MPI_INT,0,666,MPI_COMM_WORLD);

            // Mesure du temps
            end = std::chrono::system_clock::now();
            temps_total = end-start;
            output_temp << jours_ecoules << "\t"<< temps_total.count() <<
std::endl;
            jours_ecoules += 1;
}
```

```
else if (rank == 0 && affiche){
    std::size_t jours_ecoules;
    auto &data_recv = grille.getStatistiques();
    MPI_Recv(&jours_ecoules,1,MPI_INT,1,655,MPI_COMM_WORLD,&status);

 MPI_Recv(data_recv.data(),3*data_recv.size(),MPI_INT,1,666,MPI_COMM_WORLD,&statu
s);
    afficheSimulation(ecran, grille, jours_ecoules);
}
```

et dans le console, on tape:

```
$ make simulation_sync_affiche_mpi.exe

$ mpiexec -np 2 ./simulation_sync_affiche_mpi.exe
```

Les resultats sont dans le fichier `temps_sync_affiche_mpi.dat`, on le copie ci-dessous:

Jours_ecoules	Temps_total
0	0.037265
1	0.0406129
2	0.0371439
3	0.0356314
4	0.0350252
5	0.0345035
6	0.0348965
7	0.0346673
8	0.0350337
9	0.0354235
10	0.0366349
11	0.0370939
12	0.0349945
13	0.0357898
14	0.0353676
15	0.0354352
16	0.0358415
17	0.0357899
18	0.0366021
19	0.0360837
20	0.0351194
21	0.0366014
22	0.0352796
23	0.0333964
24	0.0344302
25	0.0334837
26	0.0335733
27	0.0339273
28	0.0320467
29	0.0272876

on choisit les premier 30 processus pour calculer le temps moyen. le temps moyen pour la simulation est `0.03516`. L'accélération est $S_{syn} = \frac{0.04654}{0.03516} = 1.32$.

3. Parallélisation affichage asynchrone contre simulation

On ajoute des parties de `MPI_Iprobe` dans le code:

```
if (rank ==1){
    while(!flag){
                MPI_Iprobe(0,10,MPI_COMM_WORLD,&flag,&status);
    }
            MPI_Recv(&recv_flag,1,MPI_INT,0,10,MPI_COMM_WORLD,&status);
            MPI_Send(&jours_ecoules,1,MPI_INT,0,655,MPI_COMM_WORLD);

 MPI_Send(data_passe.data(),3*data_passe.size(),MPI_INT,0,666,MPI_COMM_WORLD);

            // Mesure du temps
            end = std::chrono::system_clock::now();
            temps_total = end-start;
            output_temp << jours_ecoules << "\t"<< temps_total.count() <<
std::endl;
            jours_ecoules += 1;
}
else if (rank == 0 && affiche){
    std::size_t jours_ecoules;
    auto &data_recv = grille.getStatistiques();
    int un_flag = 1;
    MPI_Isend(&un_flag,1,MPI_INT,1,10,MPI_COMM_WORLD,&reqs);
    MPI_Recv(&jours_ecoules,1,MPI_INT,1,655,MPI_COMM_WORLD,&status);

 MPI_Recv(data_recv.data(),3*data_recv.size(),MPI_INT,1,666,MPI_COMM_WORLD,&statu
s);
    afficheSimulation(ecran, grille, jours_ecoules);
    MPI_Wait(&reqs, &status);
}
```

et dans le console, on tape:

```
$ make simulation_async_affiche_mpi.exe

$ mpiexec -np 2 ./simulation_async_affiche_mpi.exe
```

Les resultats sont dans le fichier `temps_MPI_asynchrone.dat`, on le copie ci-dessous:

Jours_ecoules	Temps_total
0	0.0319653
1	0.038656
2	0.0259204
3	0.0269099
4	0.0260465
5	0.0270664
6	0.0250489
7	0.0243786
8	0.0278109
9	0.0280297
10	0.0252437
11	0.0250032
12	0.0256393
13	0.0245803
14	0.0258006
15	0.0252362
16	0.0254097
17	0.0249796
18	0.0246138
19	0.0250064
20	0.024901
21	0.0252359
22	0.0245829
23	0.0242258
24	0.0336466
25	0.0284502
26	0.0251943
27	0.0244685
28	0.0255861
29	0.0261608

on choisit les premier 30 processus pour calculer le temps moyen. le temps moyen pour la simulation est `0.02652`. L'accélération est $S_{asyn} = \frac{0.04654}{0.02652} = 1.75$, en comparant avec l'accélération précédente, on a constaté que, la méthode avec `MPI_Iprobe` est plus vite. En fait, ce temps est près égal au temps_sans_affichage dans la question une, donc avec `MPI_Iprobe`, le processu de l'affichage n'influence pas le processu de la simulation.

4. Parallélisation OpenMP

En prenant un nombre d'individus global constant

On ajoute `# pragma omp parallel for schedule(dynamic)` dans le code, et on ajoute des `bash` documents: `async_omp_glob_cst.sh` dans le dossier.

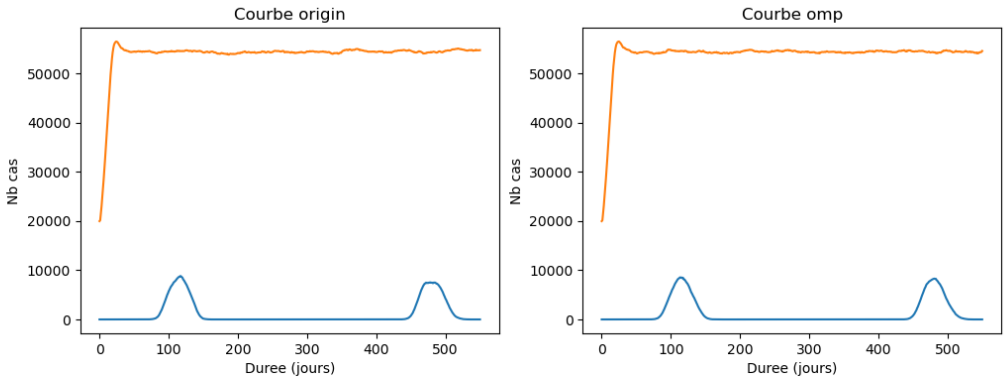
```bash
#!/bin/bash
for i in $(seq 1 6)
do
    mpiexec -np 2 ./simulation_async_omp.exe $i glob_cst
done
```

pour tester, dans le console, on tape:

```
$ make simulation_async_omp.exe

$ bash async_omp_glob_cst.sh
```

La courbe de statistuqe est identique que la courbe origin.



Les résultats sont dans les document `temps_async_omp_$(num_thread)_glob_cst.dat`.

On présente le temps des premier 30 processus pour le nombre de thread de 1 à 6 :

	Temps_total(s)					
Jours_ecoules	num_thr = 1	num_thr = 2	num_thr = 3	num_thr = 4	num_thr = 5	num_thr = 6
0	0.0373164	0.0326328	0.039941	0.039298	0.0375793	0.0457992
1	0.0474715	0.0653279	0.0574603	0.0516584	0.0626135	0.0628362
2	0.034481	0.0327677	0.0635319	0.0399508	0.0418232	0.0437946
3	0.0329678	0.0289883	0.0413494	0.0467298	0.0412827	0.0385434
4	0.0334066	0.0343605	0.031975	0.0457863	0.0352845	0.0461483
5	0.0340583	0.0364955	0.0278086	0.0338453	0.0332554	0.0328886
6	0.0334414	0.0338061	0.0305477	0.0331173	0.0355478	0.0283166
7	0.0334437	0.0319896	0.0320567	0.0490082	0.0350569	0.0406048
8	0.0330844	0.0334049	0.0277138	0.0329325	0.0337314	0.042629
9	0.0344029	0.031153	0.0295931	0.0316524	0.0360314	0.0412786
10	0.0343014	0.0309321	0.0323287	0.0320707	0.0375329	0.0414341
11	0.0339978	0.0305691	0.0284489	0.0324017	0.0365477	0.0358862
12	0.03588	0.0350953	0.0304464	0.0312075	0.0421712	0.0368692
13	0.0336782	0.030404	0.0289787	0.0317414	0.0354624	0.0363109
14	0.0338629	0.0321452	0.0286321	0.033518	0.0374871	0.0386272
15	0.035906	0.0346822	0.0315951	0.0452132	0.0452883	0.037762
16	0.0360401	0.0333714	0.0345904	0.0304155	0.039656	0.0492885
17	0.0343885	0.0318511	0.0353709	0.0433828	0.0395267	0.0339056
18	0.0329215	0.0302141	0.031378	0.0329801	0.0355561	0.0330871
19	0.0341897	0.033861	0.0323092	0.036222	0.0321794	0.0409578
20	0.034254	0.0282512	0.0326787	0.0343564	0.0341167	0.0375539
21	0.0332545	0.034272	0.0279057	0.0292592	0.0322886	0.0389227
22	0.0316273	0.03522	0.0288567	0.0341865	0.037194	0.0390097
23	0.0308372	0.0328743	0.0281382	0.0342138	0.032784	0.0375323
24	0.0304883	0.0316867	0.0288594	0.0329996	0.0313497	0.03798
25	0.0320827	0.0243353	0.0280078	0.0342767	0.0373988	0.0371269
26	0.0323081	0.021387	0.0290677	0.0324391	0.0382506	0.0384131
27	0.0284872	0.0215083	0.0268795	0.0291026	0.0382921	0.0391839
28	0.0312522	0.0250937	0.0282588	0.0328073	0.0385115	0.0374571
29	0.0297978	0.0217514	0.0276021	0.0354817	0.0406172	0.0378465
Temps moyen	0.033787647	0.03201439	0.032743683	0.03607516	0.037813903	0.0395998
Acceleration	1.377426503	1.45372128	1.421342844	1.290084368	1.23076424	1.175258461

Après, on calcule le temps moyen et accélération.

num_thread	1	2	3	4	5	6
Temps moyen(s)	0.0338	0.0320	0.0327	0.0361	0.0378	0.0396
Accélération	1.3774	1.4537	1.4213	1.2901	1.2308	1.1753

On a constaté alors, quand **le nombre de thread est 2**, la performance est le mieux, car le thread dans chaque core est 2 d'après `lscpu`.

En prenant un nombre d'individus constant par thread

On change le code précédent par `# pragma omp parallel for schedule(static)` , et on ajoute des `bash` documents: `async_omp_indv_cst.sh` dans le folder

```
#!/bin/bash
for i in $(seq 1 6)
do
    mpiexec -np 2 ./simulation_async_omp.exe $i indv_cst
done
```

pour tester, dans le console, on tape:

```
$ make simulation_async_omp.exe

$ bash async_omp_indv_cst.sh
```

Les résultats sont dans les document `temps_async_omp_$(num_thread)_indv_cst.dat`.

On présente le temps des premier 30 processus pour num_thread de 1 à 6 :

	Temps_total(s)					
Jours_ecoules	num_thr = 1	num_thr = 2	num_thr = 3	num_thr = 4	num_thr = 5	num_thr = 6
0	0.0402943	0.0264765	0.0301075	0.0474295	0.0266457	0.0458318
1	0.0528803	0.054343	0.064521	0.0393434	0.0588987	0.0524174
2	0.036173	0.0393393	0.0376394	0.0323816	0.0286032	0.0499132
3	0.0325223	0.0270613	0.0337671	0.0292228	0.0260262	0.0384573
4	0.0326961	0.0281659	0.0297999	0.0344026	0.0357129	0.0372309
5	0.0333851	0.0330657	0.0334661	0.0295524	0.0283765	0.0451064
6	0.0359831	0.0281199	0.0340056	0.0411693	0.0312098	0.0375942
7	0.0316099	0.0271309	0.0279453	0.050869	0.0275813	0.0309155
8	0.0334572	0.0274933	0.0269331	0.027293	0.0272786	0.0388782
9	0.033317	0.028624	0.0270717	0.0450556	0.0276935	0.0393775
10	0.0347619	0.0505408	0.031591	0.0282628	0.0262305	0.0343598
11	0.0397188	0.0830439	0.0290925	0.0412316	0.0284156	0.0362145
12	0.0326334	0.063988	0.0350725	0.0378124	0.028552	0.0365382
13	0.03286	0.0470486	0.0252578	0.0398784	0.0290383	0.0355439
14	0.0323611	0.0323132	0.0265873	0.034334	0.0315592	0.034983
15	0.0318471	0.0285093	0.0323289	0.0302559	0.0296975	0.0342329
16	0.0378196	0.0293331	0.0296486	0.0416026	0.0299555	0.035929
17	0.0383052	0.0285551	0.0289025	0.0377445	0.0305598	0.0371576
18	0.0377469	0.0317259	0.0354815	0.0349855	0.0314738	0.034191
19	0.0319095	0.0266648	0.0334582	0.0441959	0.0263814	0.0365425
20	0.0335743	0.0291963	0.0308762	0.0286053	0.0280306	0.0323827
21	0.0347888	0.022999	0.030771	0.0308857	0.02938	0.0308252
22	0.0277914	0.0239861	0.0287871	0.0295696	0.0287489	0.0376962
23	0.0295956	0.0253422	0.0341316	0.0379728	0.0253438	0.029625
24	0.0260916	0.0221172	0.0416357	0.0281787	0.0473935	0.0292444
25	0.0272414	0.0271391	0.0287382	0.0334813	0.0314956	0.0311949
26	0.0276	0.0267749	0.026863	0.0247648	0.0277609	0.0301648
27	0.0306194	0.0266781	0.0257812	0.0213562	0.0277351	0.0294305
28	0.0317018	0.0282483	0.035161	0.0293363	0.0263552	0.032819
29	0.0281236	0.0267781	0.0280163	0.029953	0.0245833	0.0293004
Temps moyen	0.03364699	0.03336006	0.032114627	0.034704217	0.030223897	0.036136597
Acceleration	1.383184647	1.395081424	1.449183902	1.341047414	1.539841157	1.287891066

On calcule le temps moyen et accélération.

num_thr	1	2	3	4	5	6
Temps moyen (s)	0.0336	0.0334	0.0321	0.0347	0.0302	0.0361
Accélération	1.3832	1.3951	1.4492	1.3410	1.5398	1.2879

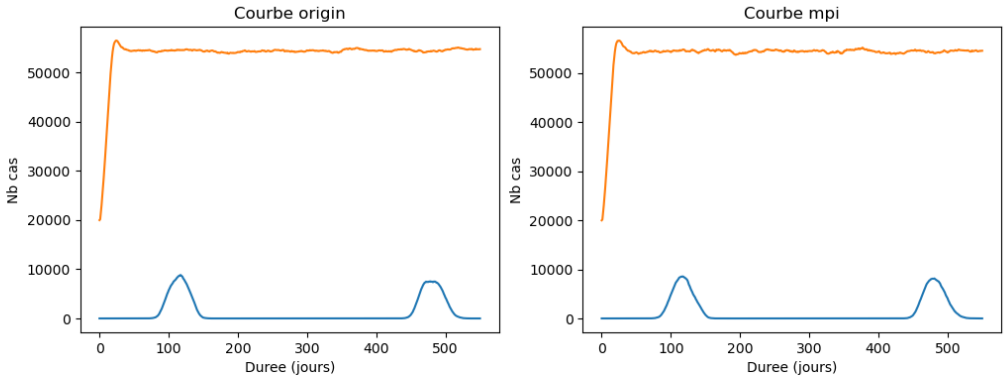
On a constaté que, quand **le nombre de thread est 5**, la performance est la mieux, mais elle est moins bon que le programme séquentiel (dans la partie 3, l'accélération est 1.75).

5. Parallélisation MPI de la simulation

La courbe de statistuqe est identique que la courbe origin.

On utilise `MPI_Comm_split` pour créer un groupe de l'affichage et un groupe de simulation.

La courbe de statistique tracé est identique que la courbe origin.



On ajoute ub `bash` documents: `async_mp.sh` dans le folder pour faciliter le teste.

```
#!/bin/bash
for i in $(seq 1 6)
do
    mpiexec -np $i ./simulation_async_mpi.exe $i
done
```

pour tester, dans le console, on tape:

```
$ make simulation_async_omp.exe

$ bash async_omp_indv_cst.sh
```

Les résultats sont dans les document `temps_async_mpi_$(num_thread)_thread.dat`.

	Temps_total(s)			
Jours_ecoules	num_proc = 2	num_proc = 3	num_proc = 4	num_proc = 5
0	0.0407658	0.0330107	0.0435372	0.0864108
1	0.0595731	0.0451886	0.11088	0.100833
2	0.040097	0.0379718	0.0410374	0.083676
3	0.0349487	0.0345194	0.0345771	0.0830816
4	0.0332916	0.036716	0.0307467	0.394512
5	0.0327826	0.0371775	0.0416582	0.05509
6	0.0321962	0.0331136	0.0384575	0.0550487
7	0.0329147	0.0278012	0.0452785	0.352852
8	0.0319096	0.0290691	0.0367202	0.0621736
9	0.0335978	0.0303395	0.0342628	0.055316
10	0.0322957	0.0310591	0.032493	0.0659113
11	0.0318455	0.0313559	0.0313221	0.0576817
12	0.0331893	0.0312649	0.0298476	0.198865
13	0.0326547	0.0311441	0.0304004	0.0991132
14	0.0328985	0.0315863	0.0341458	0.0645641
15	0.0340582	0.0332351	0.0303302	0.0530377
16	0.0319608	0.04562	0.0310277	0.0646876
17	0.0336553	0.0274307	0.0305076	0.0452079
18	0.0328003	0.0366574	0.0298479	0.305251
19	0.033345	0.0323788	0.0315936	0.0657543
20	0.0334845	0.0324412	0.0320763	0.0436017
21	0.0317338	0.0326106	0.0319594	0.251974
22	0.028886	0.0321858	0.0311176	0.0545771
23	0.0279198	0.0329559	0.0306826	0.0612839
24	0.0293489	0.0313071	0.0313016	0.0866151
25	0.0273205	0.0313529	0.0313739	0.0611672
26	0.027583	0.0311755	0.0225429	0.0624007
27	0.0278583	0.0335037	0.0256705	0.0444024
28	0.0275697	0.0317248	0.0285219	0.327495
29	0.0275368	0.0314553	0.0269167	0.0463407
Temps moyen	0.0330	0.0332	0.0354	0.1130
Accélération	1.4103	1.3999	1.3161	0.4120

On choisit les 30 premier jours pour calculer le temps moyen et accélération en variant le nombre de processus de 2 à 8. On a constaté que quand le nombre de processus est grand que 5, le temps de calcul devient trop grand. Donc on juste comparer les nombres de processus de 2 à 5

num_proc	2	3	4	5
Temps moyen(s)	0.0330	0.0332	0.0354	0.1130
Accélération	1.4103	1.3999	1.3161	0.4120

On a constaté que quand on choisit **le `num_proc` égal 2**, la performance est le mieux, mais l'accélération est inférieur que l'accélération de la partie 3, ce qui est **1.75**.

5.1 Parallélisation finale

on mélange les omp et les mpi, après, on écrit un `async_mpi_omp.sh` pour tester:

```
#!/bin/bash
for proc in $(seq 2 3)
do
    for thread in $(seq 1 4)
    do
        mpiexec -np $proc ./simulation_async_mpi_omp.exe $proc $thread
    done
done
```

Dans le console, on tape alors:

```
$ make simulation_async_mpi_omp.exe

$ bash async_mpi_omp.sh
```

Les résultats sont dans les documents `temps_async_mpiomp_proc$(num_proc)_thread$(num_thread).dat`. Ensuite, on calcule le temps moyen et l'accélération.

Pour le temps moyen (unité: s).

	num_thread = 1	num_thread = 2	num_thread = 3	num_thread = 4
num_proc = 2	0.042591137	0.03734437	0.038049697	0.040822247
num_proc = 3	0.042444457	0.045700997	0.043894157	0.057300453

Pour l'acceleration:

	num_thread = 1	num_thread = 2	num_thread = 3	num_thread = 4
num_proc = 2	1.092715613	1.246238724	1.223137215	1.140064641
num_proc = 3	1.096491831	1.018358535	1.060277803	0.812209979

Donc, on a vu alors que, le meilleur choix dans notre ordinateur est `num_proc = 2 et num_thread = 2`, ce qui est correspondant aux questions présidentes, mais la meilleure performance est moins bon que celle de parallèle avec un seul méthode.

5.2 Bilan

- `MPI_Iprobe` et `MPI_Comm_split` sont très utiles.
- La parallélisation asynchrone est mieux que la parallélisation synchrone.
- On peut partager les tâches parmi les commutateur, depuis, dans un commutateur, on peut le séparer dans les sous-commutateur et partager les tâches plus petites, mais la performance n'est plus améliorée en séparant les tâche couche par couche.