1. Introduction

In this project, we are going to convert the sequential NBody program provided to parallel version with the help of MPI, Pthread, OpenMP, CUDA and MPI with OpenMP. This report will give the details of the design of each version and compare their performance.

2. Methods

In the given sequential file, the program would basically calculate and update the positions, speeds and accelerations of several objects to simulate bodies in space.

a) Conduct resource-consuming computing step in parallel

Throughout the program, the part requiring the most computing resources is the calculation part where the program would calculate the interaction forces between every two of the objects and compute the speed, acceleration and updated position for each object. Actually, we can divide the computing task to several subtasks to be computed in parallel. As long as we make sure that all interaction forces between each object in each subtask and all other objects has been considered, the calculation result would be generally correct. In more details, assumed that there are totally N objects (from 1 to N), and a subtask is required to update the information of the object from n_1 to n_2, then we only need to do the following computing steps:

1) Compute interaction forces between objects in the range of [1, n_1-1] and objects in [n_1, n_2].
2) Compute interaction forces between objects in the range of [n_2+1, N] and objects in [n_1, n_2].
3) Compute interaction forces between any two objects in the range [n_1,n_2], notice that the force between any pair of them will be only computed once.

Then all interaction forces between objects in the range and all other objects will be calculated and only be calculated once.

b) Dividing the task into several packages

Down to the details, we need a way to split the task into packages with approximately equal size. Assumed the total number of objects is N. In the program is *totalTaskSize*, we apply the following formula to divide the task. For process (thread) with task number ***i***, it will calculate the point with from the start index to the end index (start and end index included):

$$StartIndex_i = (\frac{N}{totalTaskSize}) * i \ + \ min(N\%totalTaskSize, i)$$

$$EndIndex_i = (\frac{N}{totalTaskSize}) * (i+1) \ + \ min(N\%totalTaskSize, i+1) - 1$$

For instance, with 20 bodies, three processes would proceed the points at [0,6], [7,13], [14,19] respectively. In this case, the sizes of the sub-tasks are 7, 7 and 6 separately, which are approximately same.

c) Mechanism of plotting with updated parameters in GUI

Notice that the calculation in the program is repeating endlessly since it always need to update

the value of points after adjusting parameters and replot the whole diagram. Thus, we need to make sure that other processes or threads would interact with the master process and thread correctly to compute with updated parameters.

d) MPI

Since we use rank 0 process as the master program which generate GUI and fetch the updated parameters from users, we need to make sure rank 0 pass necessary information to other ranks to do the corresponding calculation work. Following procedures are designed to make sure each process does the calculation correctly.

Step 1: Rank 0 pack the plotting settings (current_bodies, current_space, max_mass, elapse, gravity and radius) as a struct together and then broadcast them to each process.

Step 2: All process except rank 0 unpack the data received from rank 0. Next, rank 0 would check whether parameters current_bodies, current_space, current_max_mass have changed. If any one of them have been changed, rank 0 would them regenerate a new pool of bodies. For other ranks, they would only need to check whether the value of current_bodies has been changed, if so, they would then reallocate a new body pool with the new size.

Step 3: Rank 0 would then broadcast the information (x, y, vx, vy, m) of all bodies to other ranks. Notice the accelerations is not necessary since they will be set to zero at the beginning of each iteration.

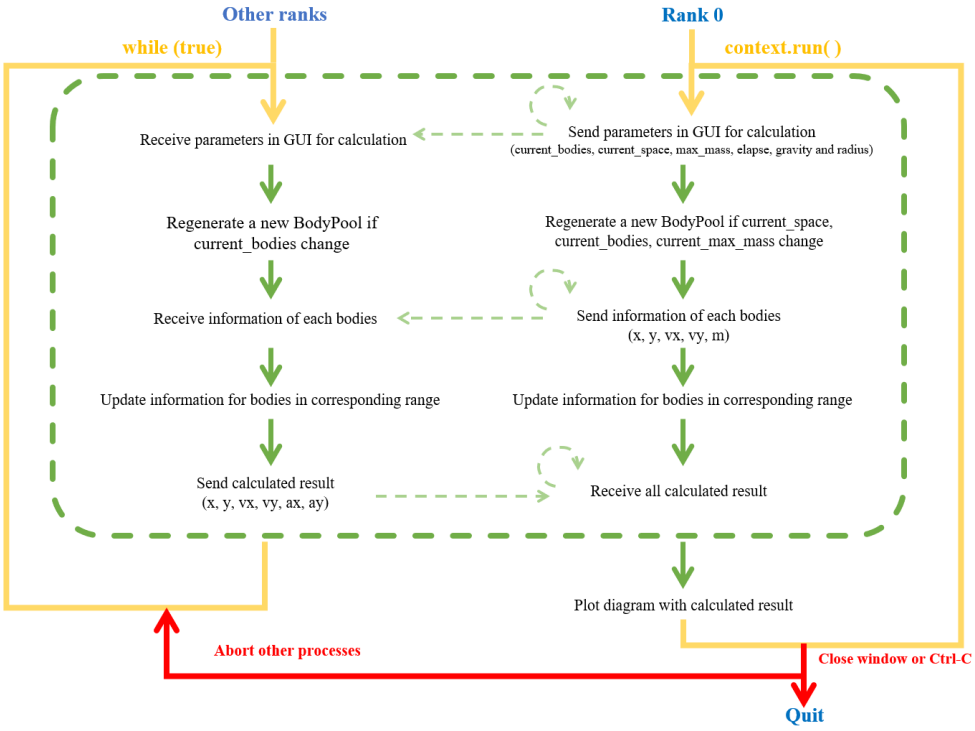
Step 4: All ranks would consider the interaction forces between any of the bodies in their corresponding range and all other bodies and make sure the interaction forces between each pair only be consider once. After computing the velocity and acceleration of each body in their correspond range, they would then calculate the updated positions for each body in range.

Step 5: Rank 0 would then gather the updated information (x, y, vx, vy, ax, ay) of bodies from each process and put them into its (rank 0's) body pool. Notice that *m* is not necessary to be send back to rank 0 since they will be modified. Although acceleration may not be necessarily sent back since they will not be used for plotting, the program still sends it back to make sure that rank 0 always has all the updated information for each iteration.

Step 6: Rank 0 process would then plot the diagram in GUI and all other processes would jump to step 1 and wait for rank 0. After plotting, rank 0 process would then jump back at step 1 and start a new round.

Step 7: If GUI window is closed or Ctrl-C is typed in the terminal, the rank 0 process would directly jump to step 7 and abort all other processes. In this case, you may see the abort signal prompted in the terminal which is a normal operation.

The details are also illustrated in the following figure:



e) Pthread/OpenMP/CUDA

Besides using multi-processes, we can also use multi-threads to finish the calculation task. In this project, the Pthread, OpenMP and CUDA are implemented mainly follow the same logic.

With multiple threads, updating the diagram with new parameters would be much easier since we can directly pass parameters to threads when creating them. Waiting for all the threads finish their calculation task, the master thread would then plot the diagram directly.

However, in this case, some potential data race would happen if all threads used the same BodyPool for calculation since functions like BodyPool.check_and_update( ) would update two bodies simultaneously. In this scenario, if one thread (handles bodies in range [0, 6]) is checking body 1 and body 7 while another thread (handles bodies in range [7, 13]) is checking body 7 and 8, potential data race would happen. To solve this problem, we would ask all threads first copy the original BodyPool. After all threads finish copying, they will do the corresponding calculation on their own copy and finally only update the information of their corresponding bodies (which would be mutually exclusive) to the original BodyPool. In this case, the data race is solved since no simultaneous write would be conduct on the same position.

Another potential solution to data race is adding mutex or lock to lock the resource when operating on the data of shared storage. However, this may cause some serious drawback on the performance of the whole program since in the extreme case, threads would wait for resources and do the calculation one by one which may somehow degrade the parallel performance to sequential operation. Thus, this project adopted the previous way (copy BodyPool and operate on its own copy) to solve data race in multi-thread. Although copy may cause some overhead, it can be ignored since the overhead of BodyPool is relatively small compared to wait for mutex especially in the case with large number of bodies (for example, if we need to calculate 100 bodies, using copy the overhead would be only 9*100 (9 vectors/arrays of size 100) and all copying can be done in

parallel, and the overhead for computing of each thread would be (10*90+10*(10-1)/2) assume we have 10 threads. However, in extreme case, using the mutex all threads would need to wait for each other and overall work in sequential which takes roughly 100*(100-1)/2 which is the complexity of computing in sequential).

As for the barrier, situation is a little different in CUDA where the barrier could only make sure all threads in one block are synchronized which means you are required to run the CUDA program with threads in one block to totally avoid the data race and please run on this way to check the data race.

Followings are the detailed steps:

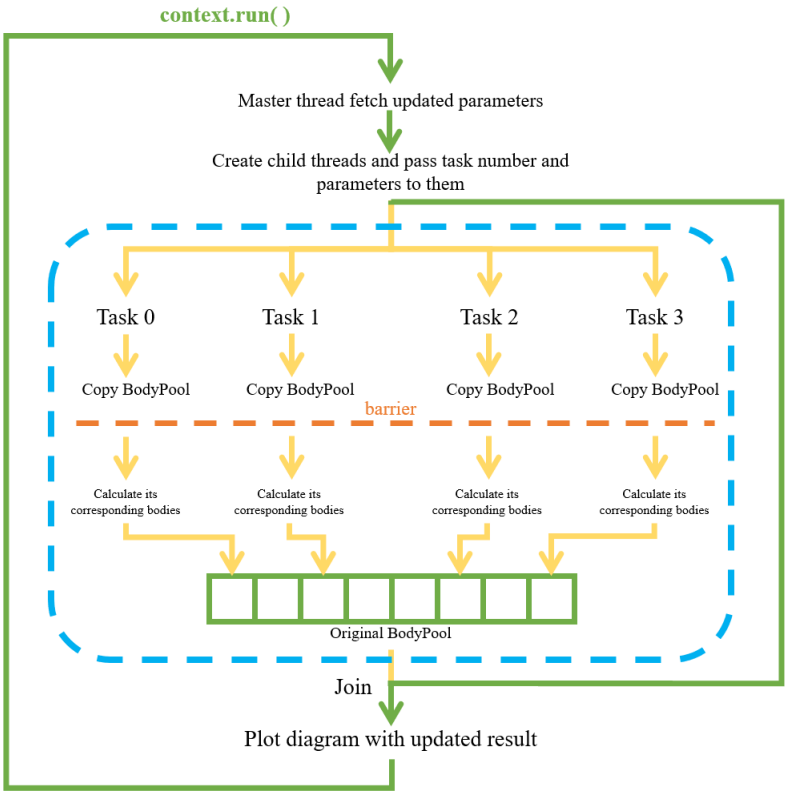
Step 1: Master program would create several threads and pass the corresponding parameters to them.

Step 2: All child threads would copy the original BodyPool. A barrier is set to make sure the following calculation would not start until all copy has been done.

Step 3: All child threads unpack the data passed from master program and then start calculation and update the information of the bodies in their corresponding range. After all calculation is finished, all threads would then copy the calculated data in their corresponding range to the original BodyPool. And the master program would wait all child threads finish their jobs by joining them.

Step 4: The program would then plot the results in the GUI and then jump to step 1.

The details are also illustrated in the following figure:



f) MPI with OpenMP

We can actually combine the multi-process and multi-thread mechanism together, that is, launch several processes and create multiple threads to finish the tasks assigned to each process.

Followings are the detailed steps:

Step 1: Rank 0 pack the plotting settings (current_bodies, current_space, max_mass, elapse, gravity and radius) as a struct together and then broadcast them to each process.

Step 2: All process except rank 0 unpack the data received from rank 0. Next, rank 0 would check whether parameters current_bodies, current_space, current_max_mass have changed. If any one of them have been changed, rank 0 would them regenerate a new pool of bodies. For other ranks, they would only need to check whether the value of current_bodies has been changed, if so, they would then reallocate a new body pool with the new size.

Step 3: Rank 0 would then broadcast the information (x, y, vx, vy, m) of all bodies to other ranks. Notice the accelerations is not necessary since they will be set to zero at the beginning of each iteration.

Step 4: Master program in each process would create several threads and pass the corresponding parameters to them.

Step 5: All child threads in one process would copy the original BodyPool. A barrier is set to make sure the following calculation would not start until all copy has been done.

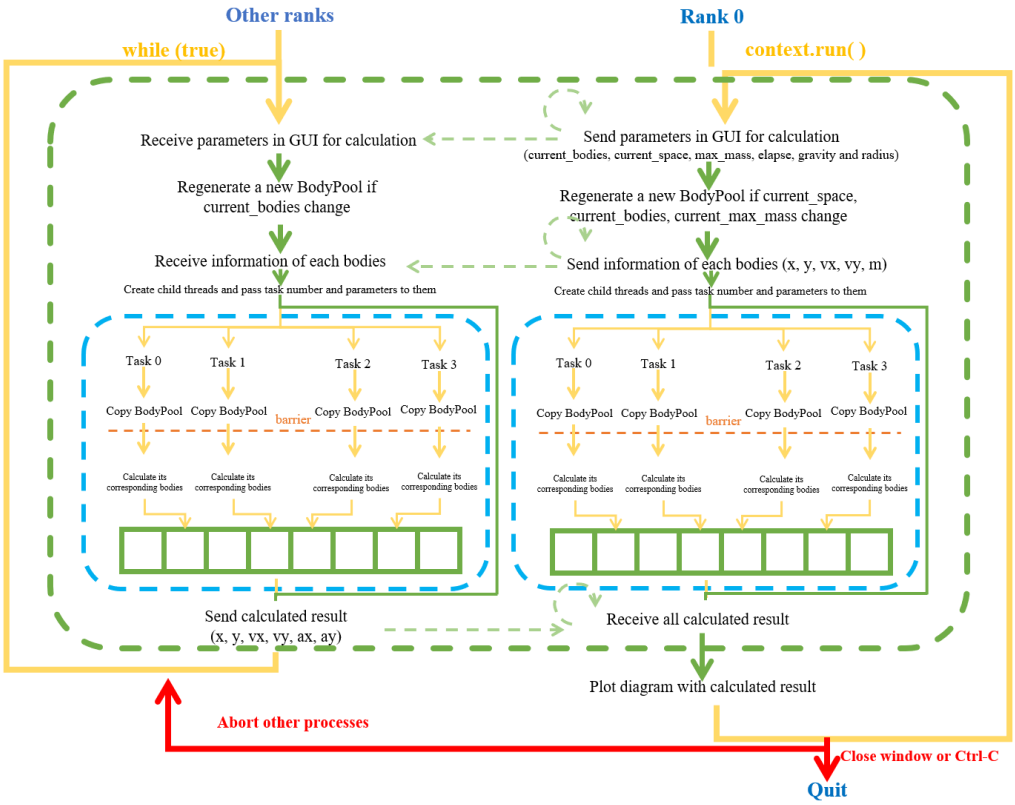
Step 6: All child threads in one process unpack the data passed from master program and then start calculation and update the information of the bodies in their corresponding range. After all calculation is finished, all threads would then copy the calculated data in their corresponding range to the original BodyPool. And the master program of each process would wait all child threads finish their jobs by joining them.

Step 7: Rank 0 would then gather the updated information (x, y, vx, vy, ax, ay) of bodies from each process and put them into its (rank 0's) body pool.

Step 8: Rank 0 process would then plot the diagram in GUI and all other processes would jump to step 1 and wait for rank 0. After plotting, rank 0 process would then jump back at step 1 and start a new round.

Step 9: If GUI window is closed or Ctrl-C is typed in the terminal, the rank 0 process would directly jump to step 7 and abort all other processes. In this case, you may see the abort signal prompted in the terminal which is a normal operation.

The details are also illustrated in the following figure:



3. Program execution

a) Compile and build

First, you need to Put program folder into "/pvfsmnt/(student_id)". And you need to type "rm -rf build" to remove the build folder.

Then follow the procedures to compile and build different version of programs:

i. MPI, Pthread, OpenMPI, MPI with OpenMP

1) Type "mkdir build"
2) Type "cd build"
3) Type "cmake .. -DCMAKE_BUILD_TYPE=Debug"
4) Type "cmake --build . -j4"

ii. CUDA

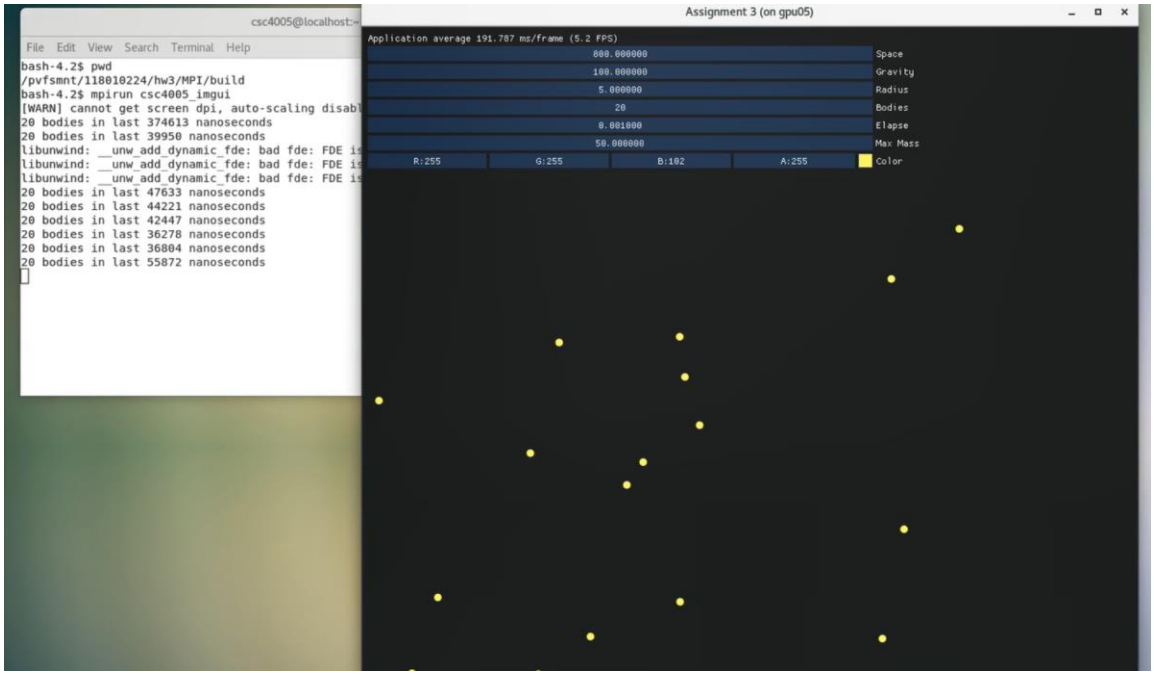
1) Type "mkdir build"
2) Type "cd build"
3) Type "source scl_source enable devtoolset-10"
4) Type "CC=gcc CXX=g++ cmake .."
5) Type "make -j12"

Then, you need to do some authorization settings before executing the programs. In a suitable environment (all the executions of programs in this project are done in the virtual machine set up in tutorial 1), type "xhost +" first. Then connect to the server using "ssh -Y {student id}@10.26.1.30". Then type "cd /pvfsmnt/$(whoami)", "cp ~/.Xauthority /pvfsmnt/$(whoami)", "export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority".

b) MPI

Since program run with multiple processes, you first need to allocate several cores. Type “salloc -n9 -t5” to request 9 cores for example. Then type “mpirun csc4005_imgui” in the build folder then it will prompt out the program.

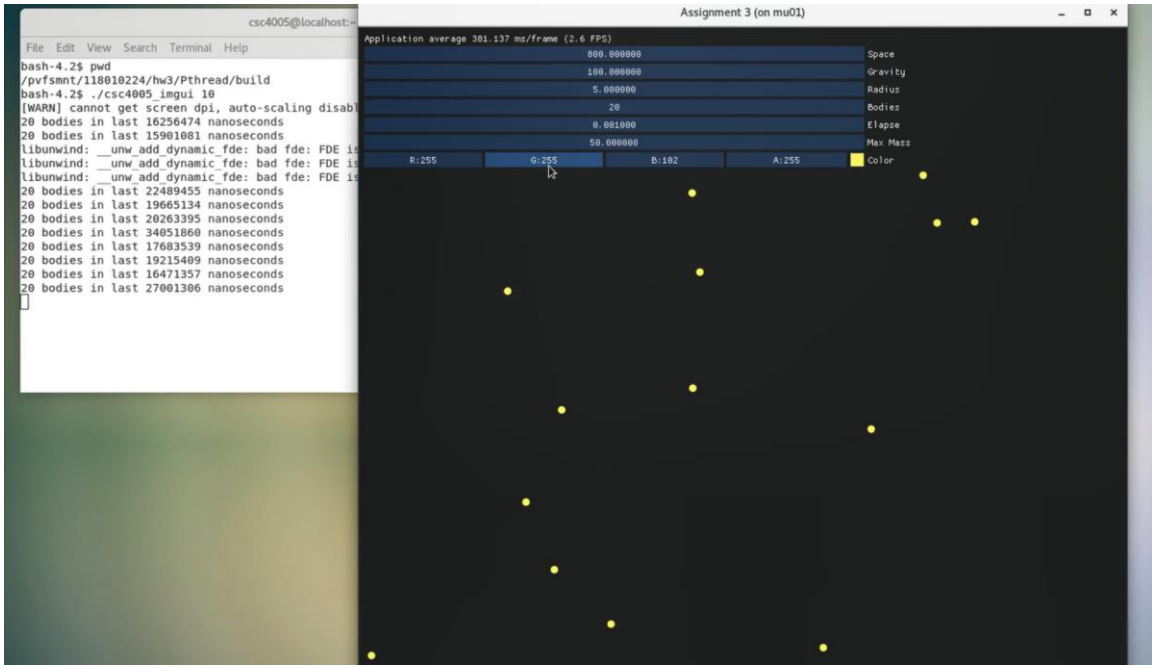
Running result (in submitted codes, the initial size has been changed to 200):



c) Pthread

The steps to execute the pthread version program is relative easier, you just need to type “./csc4005_imgui [number of threads]” directly, for example, type “./csc4005_imgui 10” for execute program in 10 threads. The default setting is 4, which would be set up automatically if the input parameters is invalid or not given.

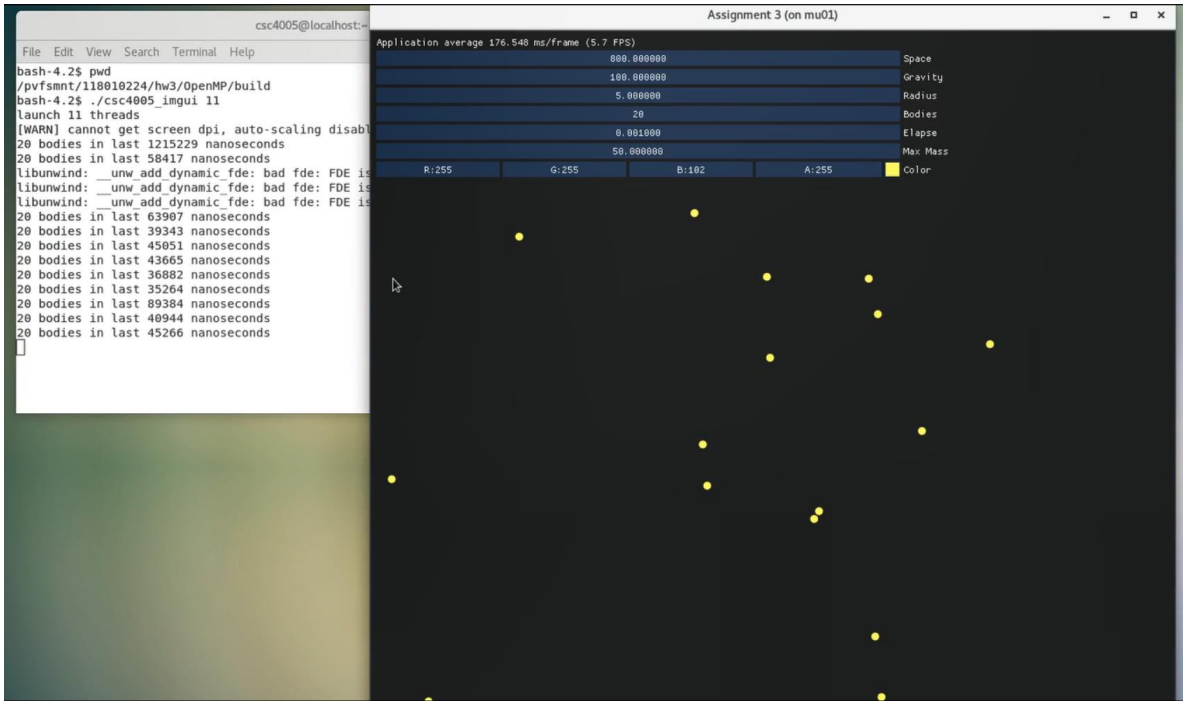
Running result (in submitted codes, the initial size has been changed to 200):



d) OpenMP

The steps to execute the OpenMP version program is same as Pthread, you just need to type "./csc4005_imgui [number of threads]" directly, for example, type "./csc4005_imgui 11" for execute program in 10 threads. The default setting is 4, which would be set up automatically if the input parameters is invalid or not given.

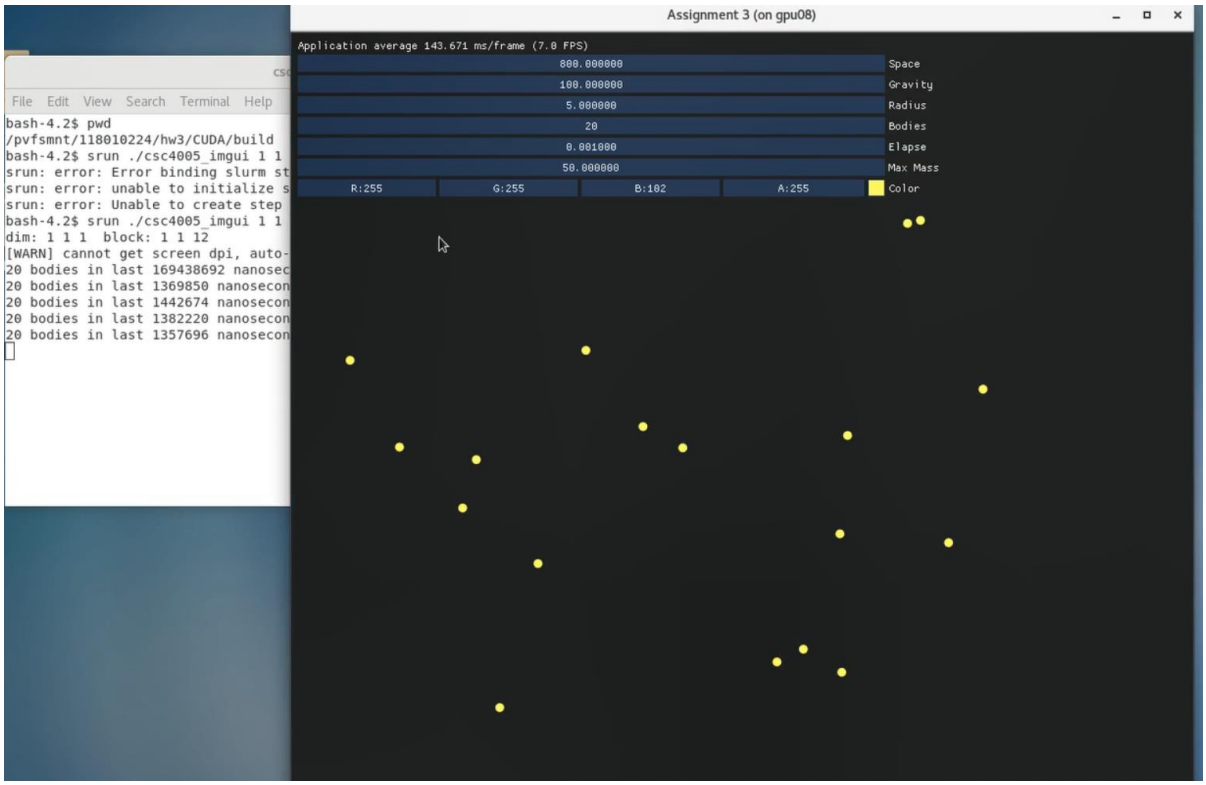
Running result (in submitted codes, the initial size has been changed to 200):



e) CUDA

The steps to execute the OpenMP version program is a little different. You first need to type "salloc -n1 -t10" to obtain one GPU node. Then you need to type "srun ./csc4005_imgui [dim.x] [dim.y] [dim.z] [block.x] [block.y] [block.z]" directly, for example, type "./csc4005_imgui 1 1 1 1 1 12" for execute program with the CUDA settings as dim(1,1,1) and block(1,1,20). The default setting for all value is 1, which would be set up automatically if the input parameters is invalid or not given.

Running result (in submitted codes, the initial size has been changed to 200):

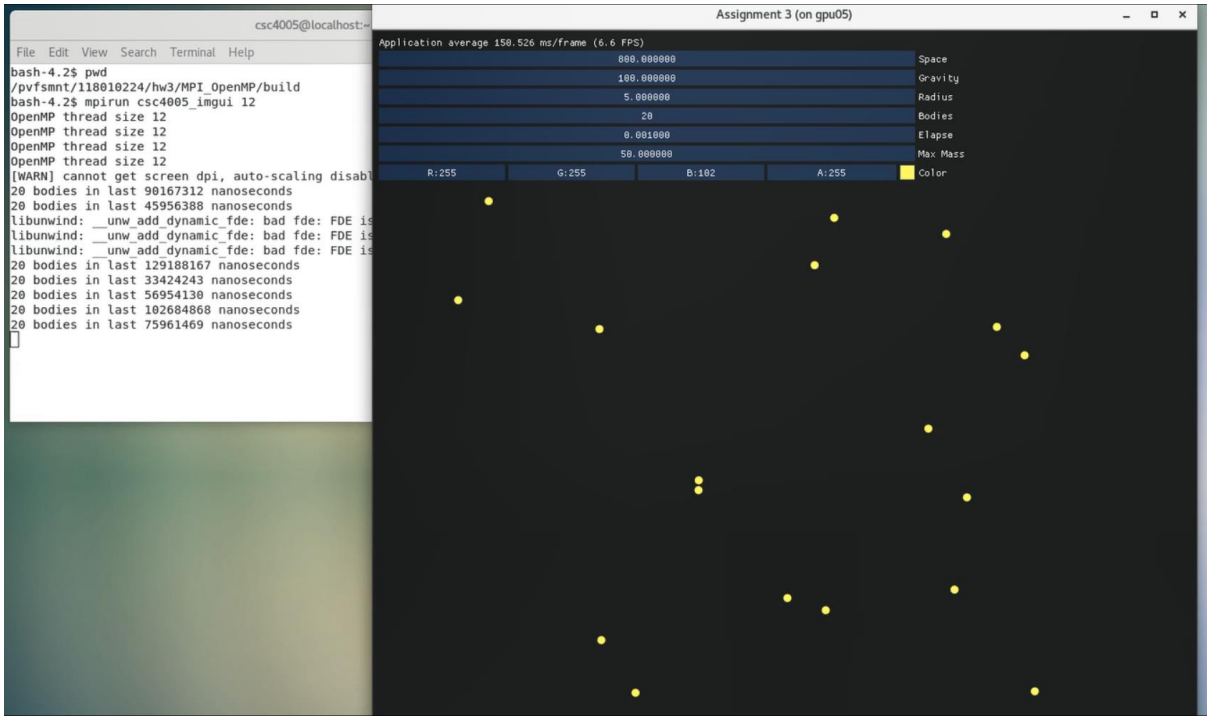


(Notice that the “srun error” in the snapshot is the error of allocating nodes in cluster, which has no relation with the codes)

f) MPI with OpenMP

Since program run with multiple processes, you first need to allocate several cores. Type “salloc -n4 -t10” to request 4 cores for example. Then you need to type “mpirun ./csc4005_imgui [number of threads in each process]”. The default setting for number of threads in each process is 4, which would be set up automatically if the input parameters is invalid or not given.

Running result (in submitted codes, the initial size has been changed to 200):



4. Performance analysis:
   a) Time complexity:

      In this analysis, we only consider the calculation task for one iteration. Also, we ignore the plotting steps.

      i. Sequential

      Assume there are totally *N* bodies, basically for one iteration the program would first check the interaction forces between each two of the bodies roughly takes $O\left(\frac{N(1+N)}{2}\right) = O(N^2)$. Then it will update the information of each point which takes $O(N)$. Totally it will take $O(N^2) + O(N) = O(N^2)$.

      ii. MPI

      Assume there are *N* bodies and *p* processes. At the beginning of each iteration, the rank 0 process would broadcast x, y, vx, vy, m of the BodyPool and gather the information of x, y, vx, vy, ax, ay from all other ranks after the calculation. Thus, the overhead of communication is roughly about $O(4N + 6N) = O(N)$. For each process, they need to handle $\frac{N}{p}$ bodies which will takes about $O\left(\left(N - \frac{N}{p}\right) * \frac{N}{p} + \frac{\frac{N}{p} * \left(\frac{N}{p} - 1\right)}{2}\right) = O(\frac{N^2}{p})$. Therefore, the total time complexity would be $O(N) + O\left(\frac{N^2}{p}\right) = O\left(\frac{N^2}{p}\right)$.

      iii. Pthread/OpenMP/CUDA

      Assume there are *N* bodies and *t* threads. Basically, all these three programs follow the same mechanism of MPI. Instead of broadcast and gather data, the threads would directly copy the BodyPool which will roughly take $O(7N)$ and finally all results would be directly copied to original BodyPool which means no extra time would be needed to gather calculated results. For each thread, they need to handle $\frac{N}{t}$ bodies which will takes about $O\left(\left(N - \frac{N}{t}\right) * \frac{N}{t} + \frac{\frac{N}{t} * \left(\frac{N}{t} - 1\right)}{2}\right) = O(\frac{N^2}{t})$. Therefore, the total time complexity would be $O(N) + O\left(\frac{N^2}{t}\right) = O\left(\frac{N^2}{t}\right)$.

      iv. MPI with OpenMP

      Assume there are *N* bodies and *t* threads totally. The time complexity of this would simply be the combination of the compartments of MPI and OpenMP which is then $O(N) + O(N) + O\left(\frac{N^2}{t}\right) = O\left(\frac{N^2}{t}\right)$.

   b) Data race test in Pthread and OpenMP

      Notice that all data race tests are conducted on virtual machine provided in the tutorials.

      i. Pthread

      To activate the data race mode, you need to do some modifications in the CMakeList.txt, which

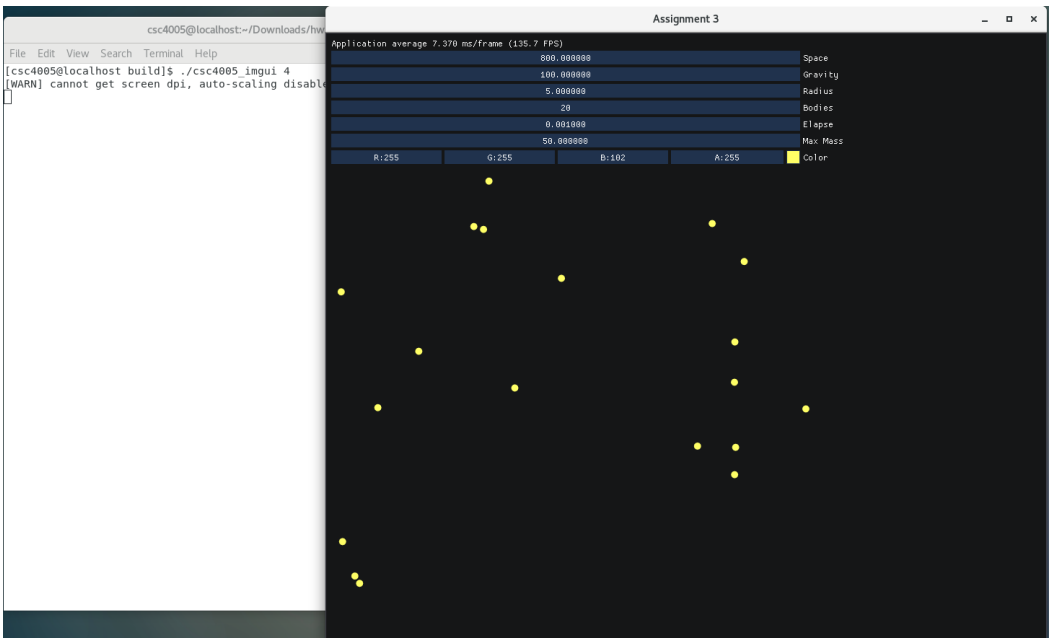
is shown in the followings:

CMakeLists.txt

Pthread > CMakeLists.txt

```
1   cmake_minimum_required(VERSION 3.2)
2   project(csc4005_imgui)
3
4   set(CMAKE_CXX_STANDARD 17)
5
6   find_package(SDL2 REQUIRED)
7   find_package(Freetype REQUIRED)
8   find_package(MPI REQUIRED)
9   find_package(Threads REQUIRED)
10  set(CMAKE_CXX_STANDARD 20)
11  set(OpenGL_GL_PREFERENCE "GLVND")
12  find_package(OpenGL REQUIRED)
13
14  set(CMAKE_CXX_FLAGS "-fsanitize=thread")
15
16  include_directories(
17          include
18          imgui
19          imgui/backends
20          ${SDL2_INCLUDE_DIRS}
21          ${FREETYPE_INCLUDE_DIRS}
22          ${MPI_CXX_INCLUDE_DIRS})
23
24  file(GLOB IMGUI_SRC
25          imgui/*.cpp
26          imgui/backends/imgui_impl_sdl.cpp
27          imgui/backends/imgui_impl_opengl2.cpp
28          imgui/misc/freetype/imgui_freetype.cpp
29          imgui/misc/cpp/imgui_stdlib.cpp
30          )
31  add_library(core STATIC ${IMGUI_SRC})
32  file(GLOB CSC4005_PROJECT_SRC src/*.cpp src/*.c)
33  add_executable(csc4005_imgui ${CSC4005_PROJECT_SRC})
34  get_filename_component(FONT_PATH imgui/misc/fonts/DroidSans.ttf ABSOLUTE)
```

Testing result (duration is commanded to show more clear result of data race checking):



ii. OpenMP

First, you need to do some modifications in CMakeList.txt before compile and build, which is shown in the followings:

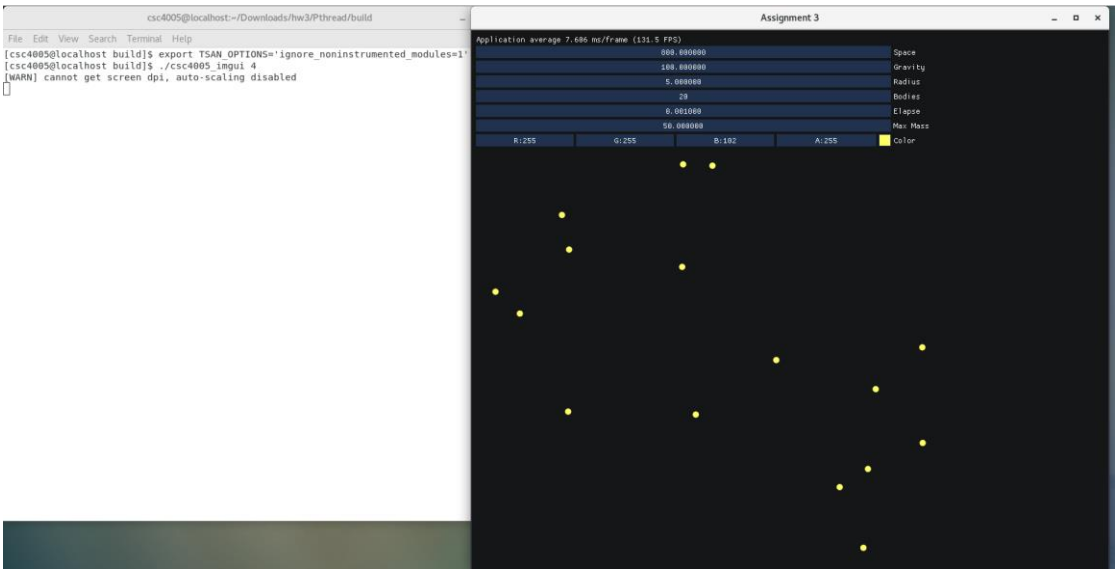
CMakeLists.txt

OpenMP > CMakeLists.txt

```
cmake_minimum_required(VERSION 3.2)
project(csc4005_imgui)

set(CMAKE_CXX_STANDARD 17)

find_package(SDL2 REQUIRED)
find_package(Freetype REQUIRED)
find_package(MPI REQUIRED)
find_package(Threads REQUIRED)
set(CMAKE_CXX_STANDARD 20)
set(OpenGL_GL_PREFERENCE "GLVND")
find_package(OpenGL REQUIRED)

find_package(OpenMP REQUIRED)
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
# set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS} -fsanitize=thread")
set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")

include_directories(
        include
        imgui
        imgui/backends
        ${SDL2_INCLUDE_DIRS}
        ${FREETYPE_INCLUDE_DIRS}
        ${MPI_CXX_INCLUDE_DIRS})

file(GLOB IMGUI_SRC
        imgui/*.cpp
        imgui/backends/imgui_impl_sdl.cpp
        imgui/backends/imgui_impl_opengl2.cpp
        imgui/misc/freetype/imgui_freetype.cpp
        imgui/misc/cpp/imgui_stdlib.cpp
        )
```

Before running the program, you need to type “ export TSAN_OPTIONS='ignore_noninstrumented_modules=1' ” to avoid some extra false positive errors.

Testing result (duration is commanded to show more clear result of data race checking):



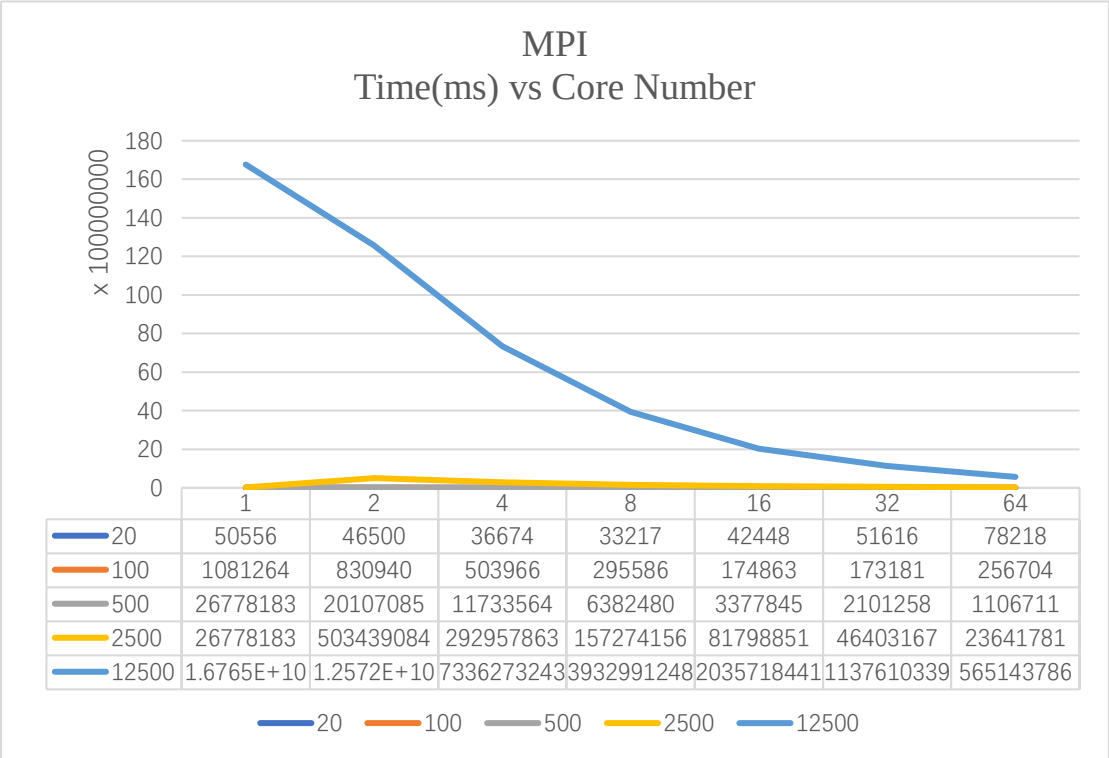
c) Performance Test

In this project, MPI, Pthread, OpenMP, CUDA and MPI with OpenMP would work in the same way as sequential version if total number of process or thread is 1. Thus, for testing, all versions with one process would work as the sequential version in their test group. All the test results generated from all versions would only focus on the calculation steps.

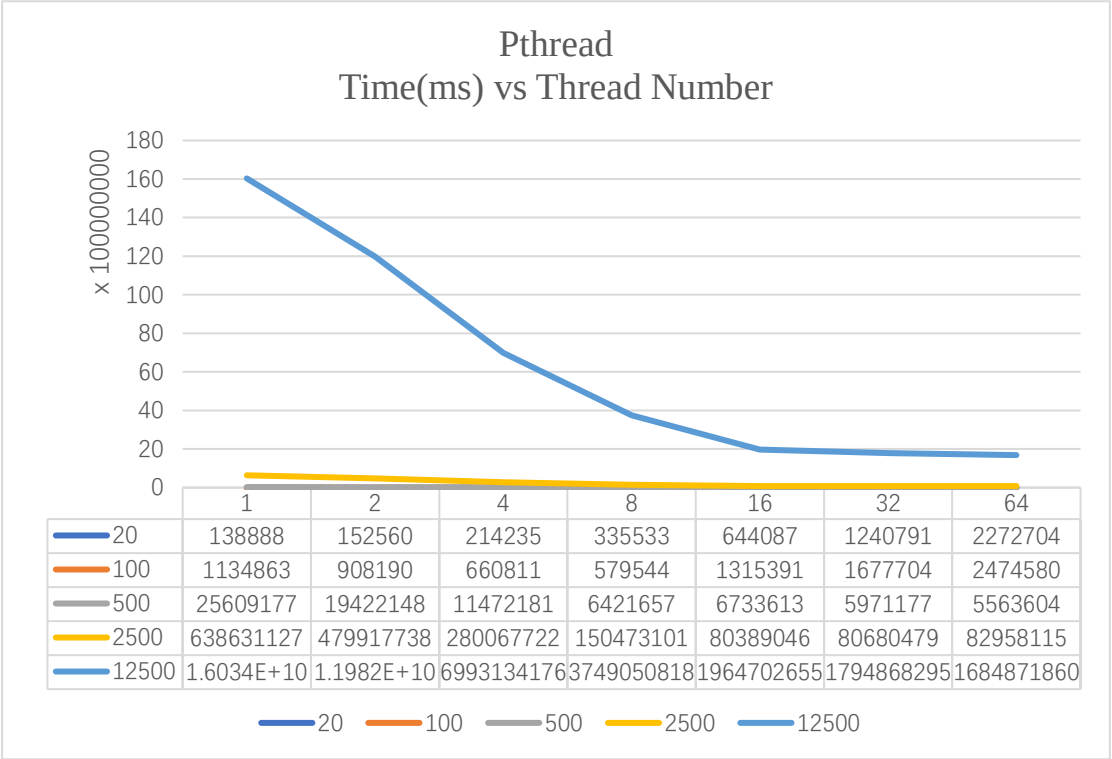
i. Time vs number of Core/Thread

Notice for testing, MPI will use one node for 1 process test and 2 nodes for all the remaining tests. For CUDA, we will only use one node. For MPI with OpenMP, we will only use one node for totally one thread test and two nodes for the remaining test.

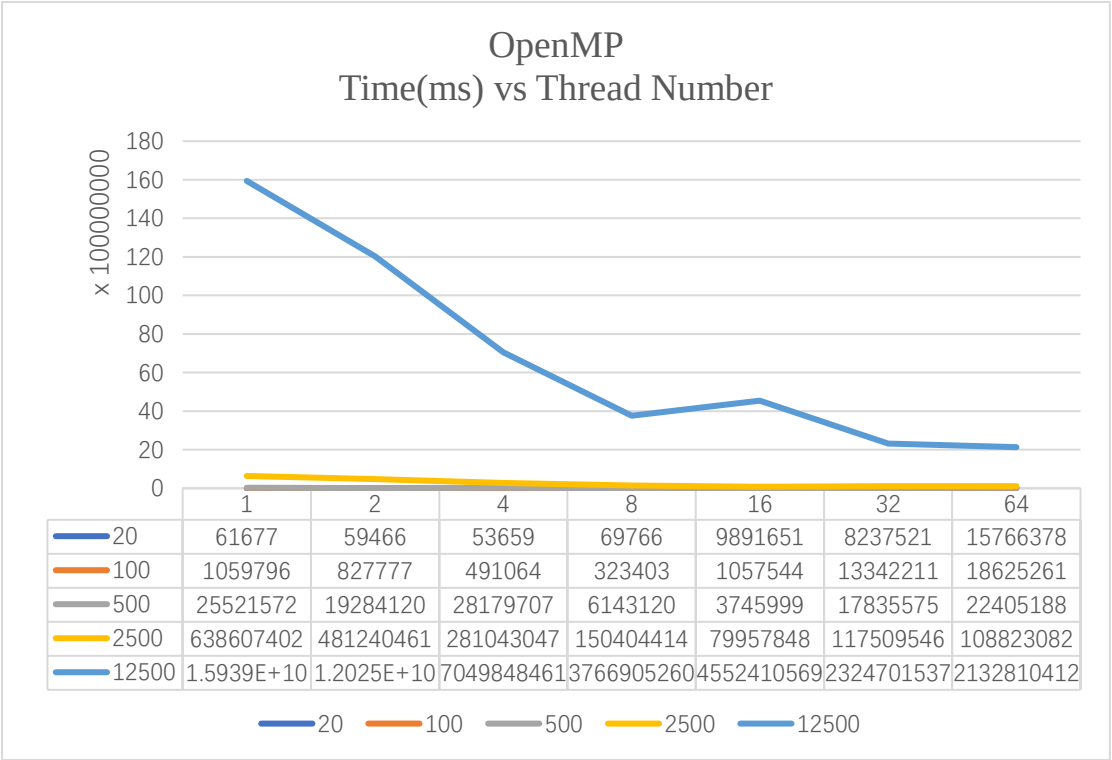
Part 1. all test sets:



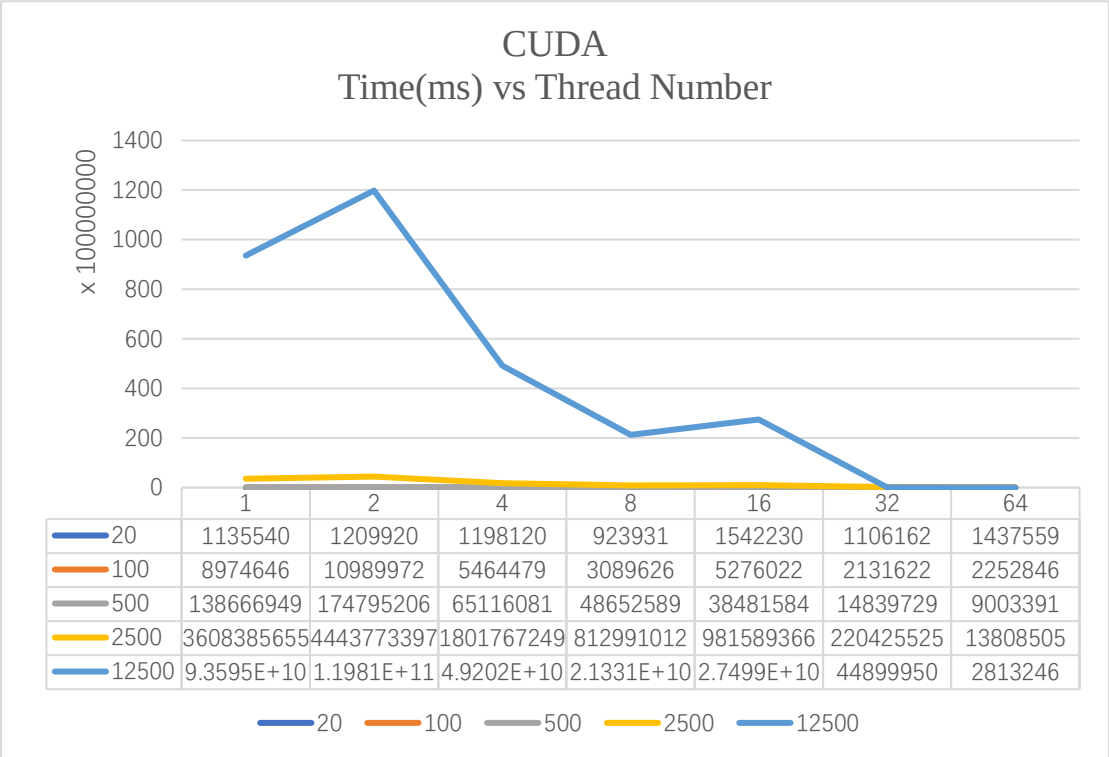
	1	2	4	8	16	32	64
20	50556	46500	36674	33217	42448	51616	78218
100	1081264	830940	503966	295586	174863	173181	256704
500	26778183	20107085	11733564	6382480	3377845	2101258	1106711
2500	26778183	503439084	292957863	157274156	81798851	46403167	23641781
12500	1.6765E+10	1.2572E+10	7336273243	3932991248	2035718441	1137610339	565143786

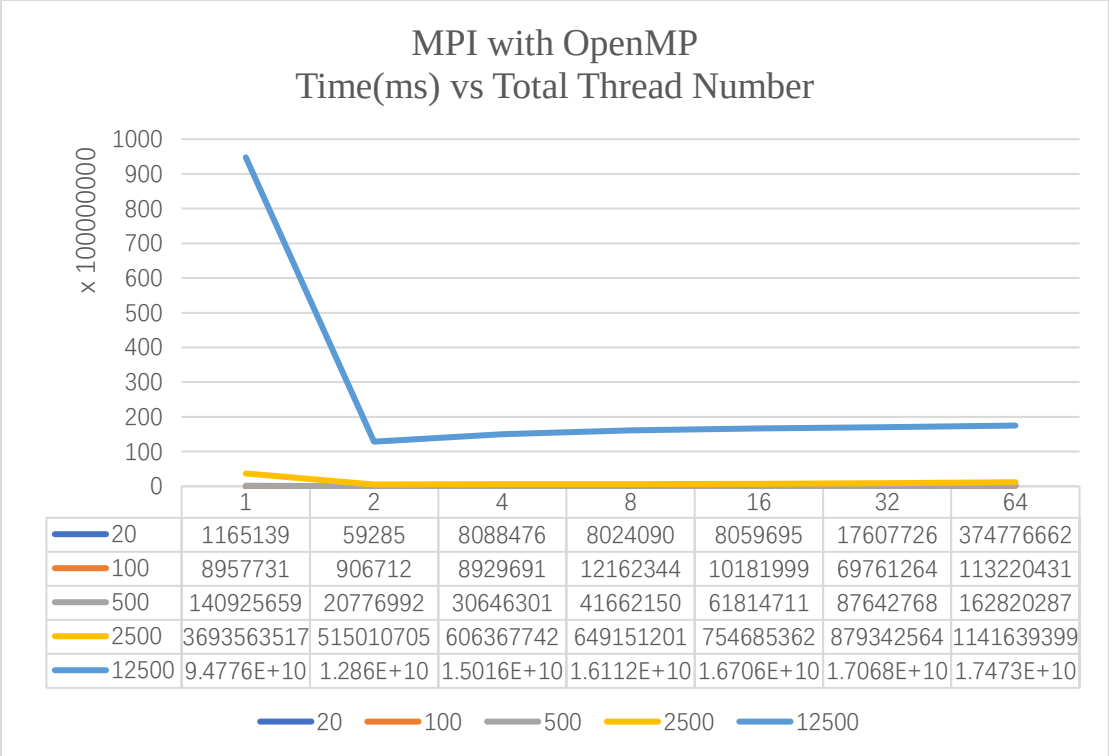


	1	2	4	8	16	32	64
20	138888	152560	214235	335533	644087	1240791	2272704
100	1134863	908190	660811	579544	1315391	1677704	2474580
500	25609177	19422148	11472181	6421657	6733613	5971177	5563604
2500	638631127	479917738	280067722	150473101	80389046	80680479	82958115
12500	1.6034E+10	1.1982E+10	6993134176	3749050818	1964702655	1794868295	1684871860



	1	2	4	8	16	32	64
20	61677	59466	53659	69766	9891651	8237521	15766378
100	1059796	827777	491064	323403	1057544	13342211	18625261
500	25521572	19284120	28179707	6143120	3745999	17835575	22405188
2500	638607402	481240461	281043047	150404414	79957848	117509546	108823082
12500	1.5939E+10	1.2025E+10	7049848461	3766905260	4552410569	2324701537	2132810412

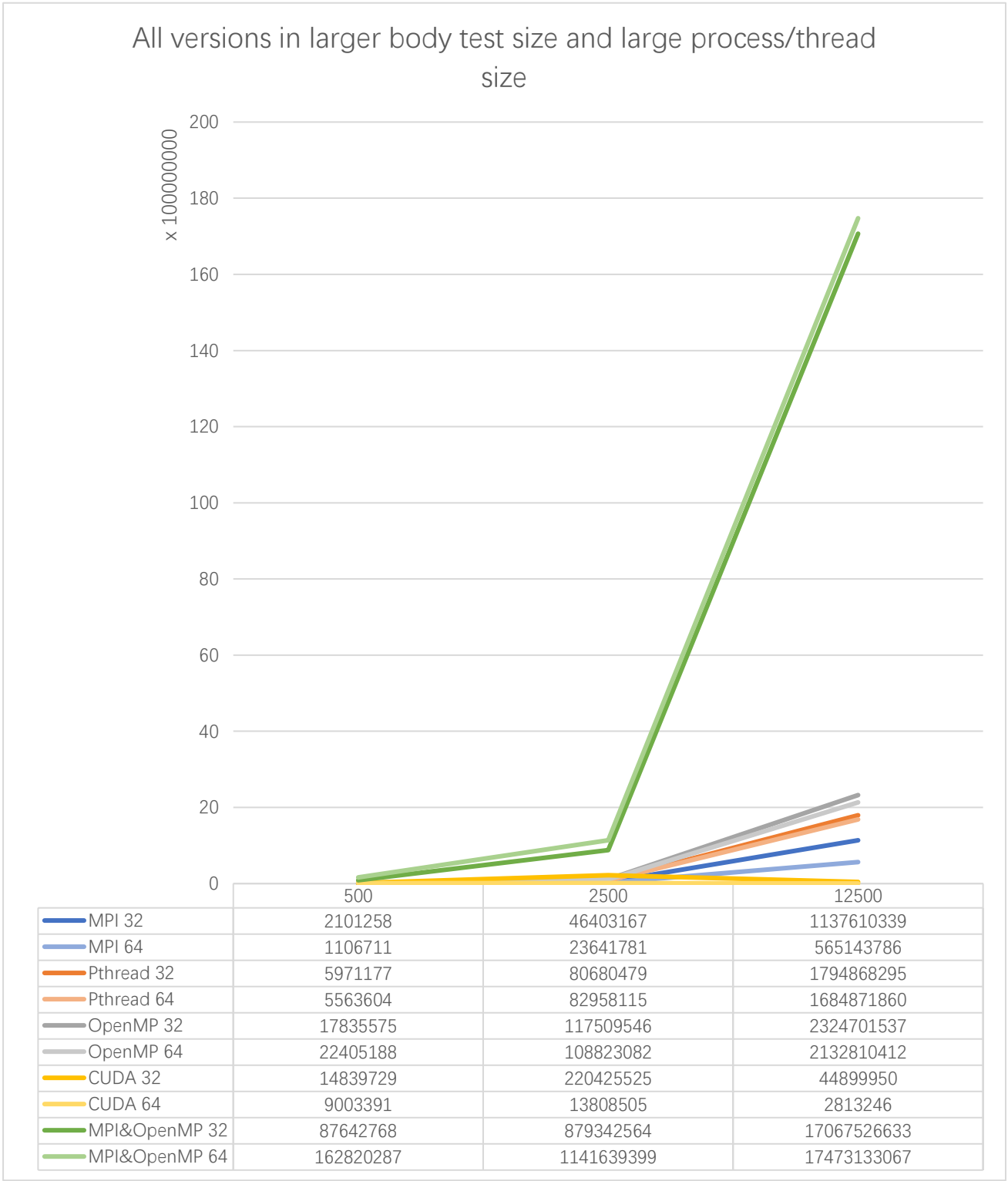




Generally, with more threads or cores involved, the running time decreased steadily for all versions in test of all data size, which is quite intuitive since more cores and threads would separate the computing task. However, there are some fluctuations in the results of OpenMP, CUDA and MPI with OpenMP. For OpenMP, the potential reason may be the time for creating threads and copying from the original BodyPool offset the improvement of performance with more cores divided the tasks. For CUDA, one highly possible reason for this fluctuation is the time for copying original BodyPool maintain the copy region with cudaMallocManaged. In terms of MPI with OpenMP, reasons may be the extra time spent on broadcasting and gathering the original BodyPool between process and copying that BodyPool within each thread and

communication between process in different nodes, which may significantly slow down the performance speed. Notice that, when applying with large number of threads (64 threads) which may exceed the physical number of threads, the performance not changed much. One possible reason for this may be the effect of switching time of threads from memory may be offset by time spent on huge computation.

Part 2. All versions in larger body test size and large process/thread size



	500	2500	12500
MPI 32	2101258	46403167	1137610339
MPI 64	1106711	23641781	565143786
Pthread 32	5971177	80680479	1794868295
Pthread 64	5563604	82958115	1684871860
OpenMP 32	17835575	117509546	2324701537
OpenMP 64	22405188	108823082	2132810412
CUDA 32	14839729	220425525	44899950
CUDA 64	9003391	13808505	2813246
MPI&OpenMP 32	87642768	879342564	17067526633
MPI&OpenMP 64	162820287	1141639399	17473133067

In the test with large data size and large thread or process size, the MPI with OpenMP generally works the worst among all mechanisms which is attribute the time spent multiple copying the original BodyPool (we can also take broadcast as a way of copying) and communication between

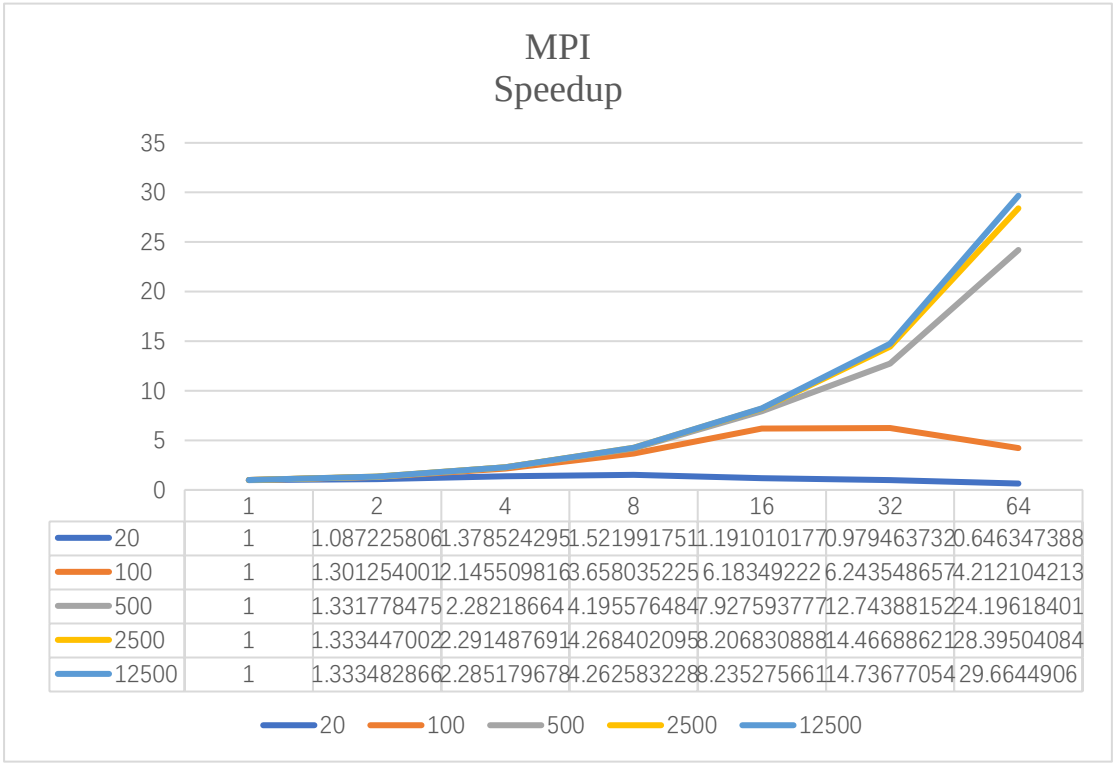
process in different nodes. Among all, CUDA performs the best, one of the possible reasons may be the relative powerful computing ability of each thread.

Part 3. Speedup

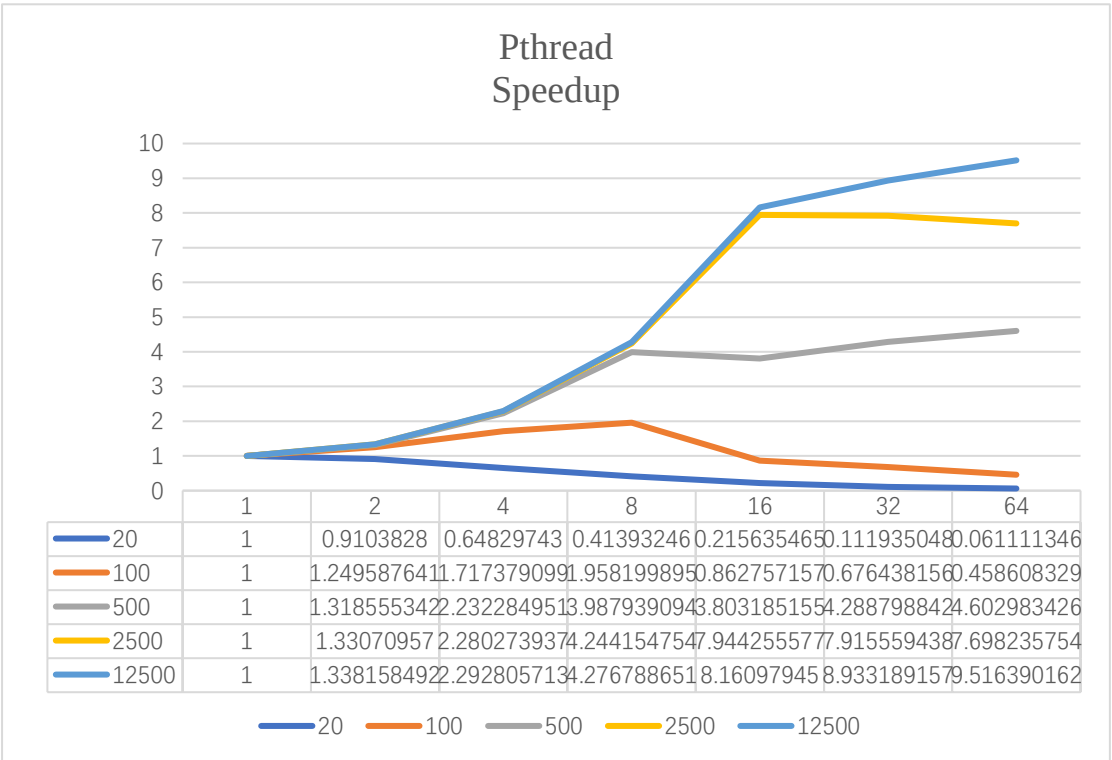
We can also calculate the Speedup of the program.

The formula of the Speedup is given as:

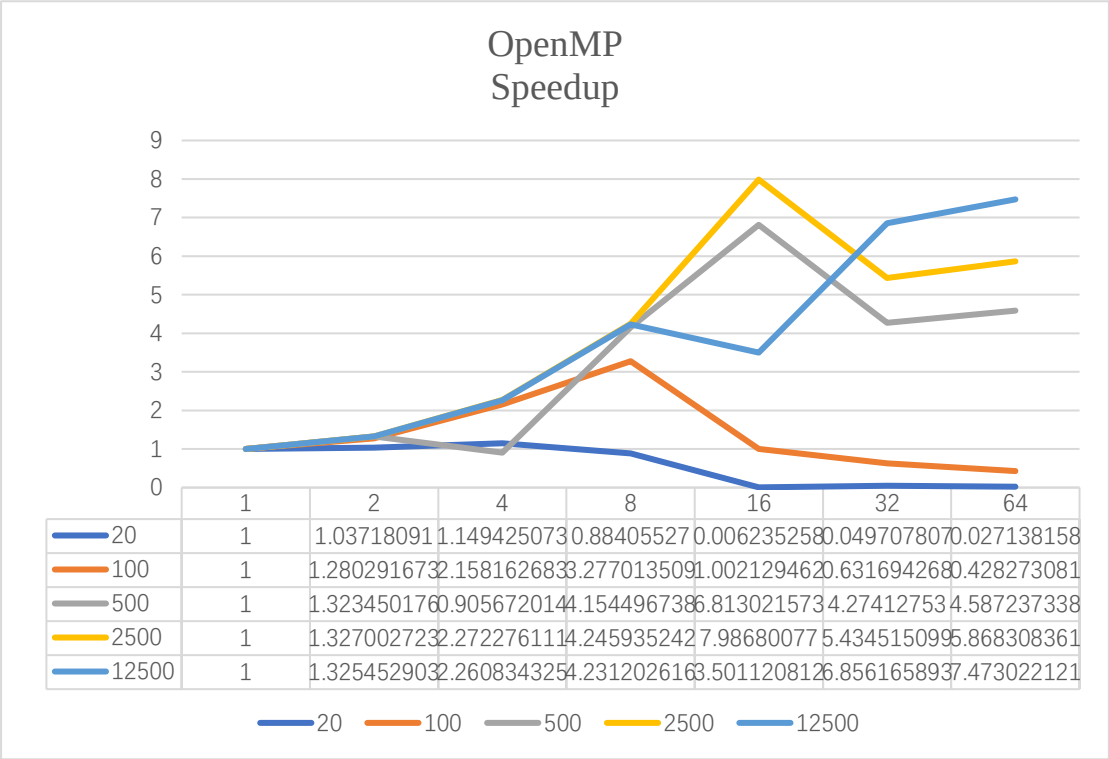
$$Speedup = \frac{running\ time\ of\ sequentail\ version}{running\ time\ of\ parallel\ version}$$



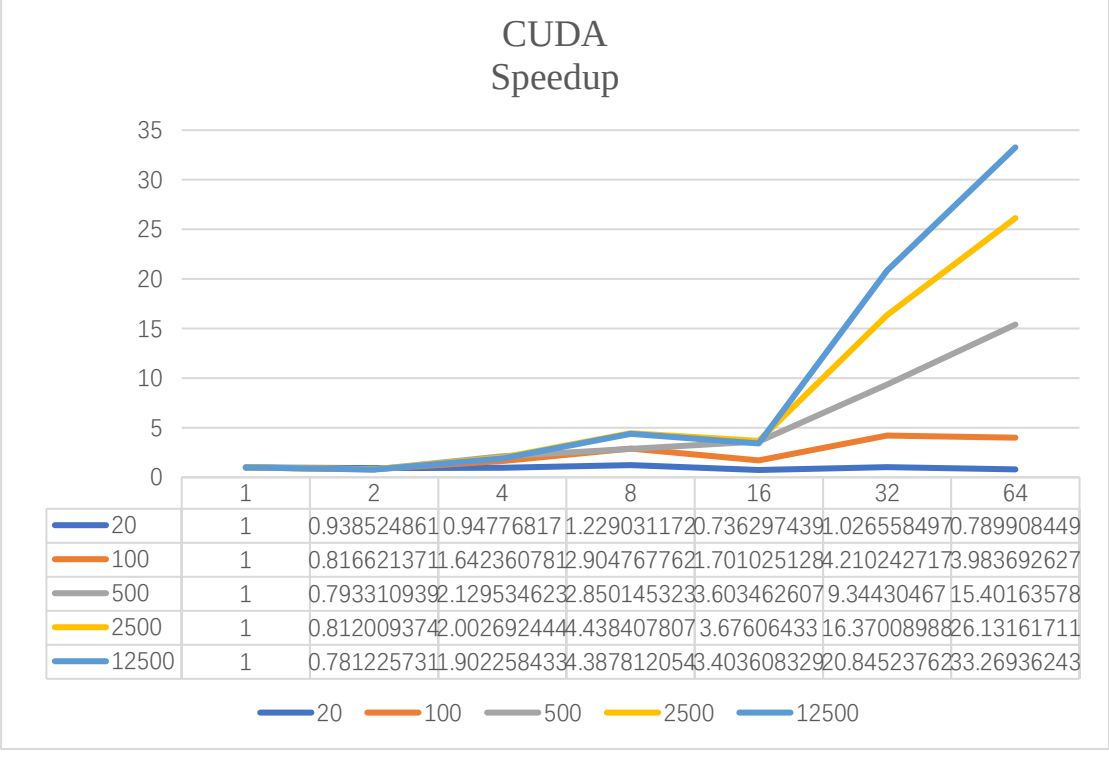
	1	2	4	8	16	32	64
20	1	1.087225806	1.378524295	1.521991751	1.191010177	0.979463732	0.646347388
100	1	1.301254001	2.145509816	3.658035225	6.18349222	6.243548657	4.212104213
500	1	1.331778475	2.28218664	4.195576484	7.927593777	12.74388152	24.19618401
2500	1	1.333447002	2.291487691	4.268402095	8.206830888	14.46688621	28.39504084
12500	1	1.333482866	2.285179678	4.262583228	8.235275661	14.73677054	29.6644906



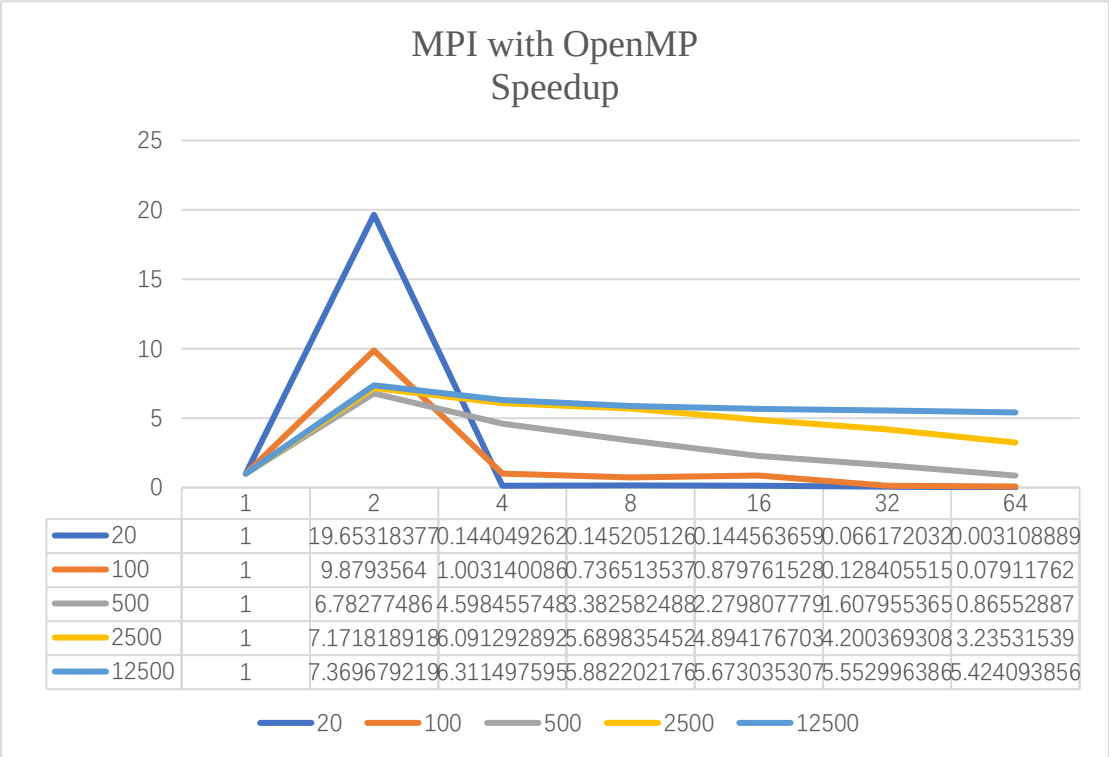
	1	2	4	8	16	32	64
20	1	0.9103828	0.64829743	0.41393246	0.215635465	0.111935048	0.061111346
100	1	1.249587641	1.717379099	1.958199895	0.862757157	0.676438156	0.458608329
500	1	1.318555342	2.232284951	3.987939094	3.803185155	4.288798842	4.602983426
2500	1	1.33070957	2.280273937	4.244154754	7.944255577	7.915559438	7.698235754
12500	1	1.338158492	2.292805713	4.276788651	8.16097945	8.933189157	9.516390162



	1	2	4	8	16	32	64
20	1	1.03718091	1.149425073	0.88405527	0.006235258	0.049707807	0.027138158
100	1	1.280291673	2.158162683	3.277013509	1.002129462	0.631694268	0.428273081
500	1	1.323450176	0.905672014	4.154496738	6.813021573	4.27412753	4.587237338
2500	1	1.327002723	2.272276111	4.245935242	7.98680077	5.434515099	5.868308361
12500	1	1.325452903	2.260834325	4.231202616	3.501120812	6.856165893	7.473022121



	1	2	4	8	16	32	64
20	1	0.938524861	0.947768171	1.229031172	0.736297439	1.026558497	0.789908449
100	1	0.816621371	1.642360781	2.904767762	1.701025128	4.210242717	3.983692627
500	1	0.793310939	2.129534623	2.850145323	3.603462607	9.34430467	15.40163578
2500	1	0.812009374	2.002692444	4.438407807	3.67606433	16.37008988	26.13161711
12500	1	0.781225731	1.902258433	4.387812054	3.403608329	20.84523762	33.26936243

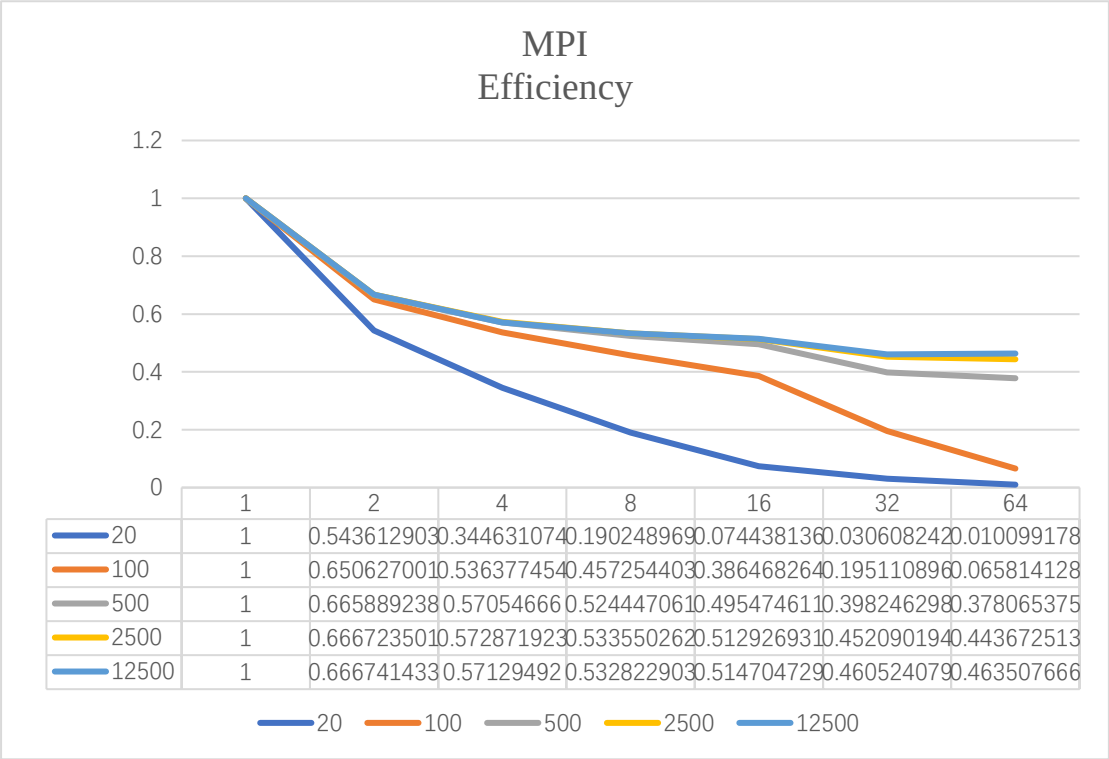


For MPI, Pthread, OpenMP, CUDA, the speedup generally increases which attribute to the fact that more computing resource divide the total computing task. For MPI with OpenMP, however, the speedup increases first and then remain the same. The reasons for this may be the extra time spent on broadcasting and gathering the original BodyPool between process and copying that BodyPool within each thread, which may significantly slow down the performance speed and communication between process in different nodes. In this case, the effect was balanced the best in MPI with OpenMP with small size of total threads.

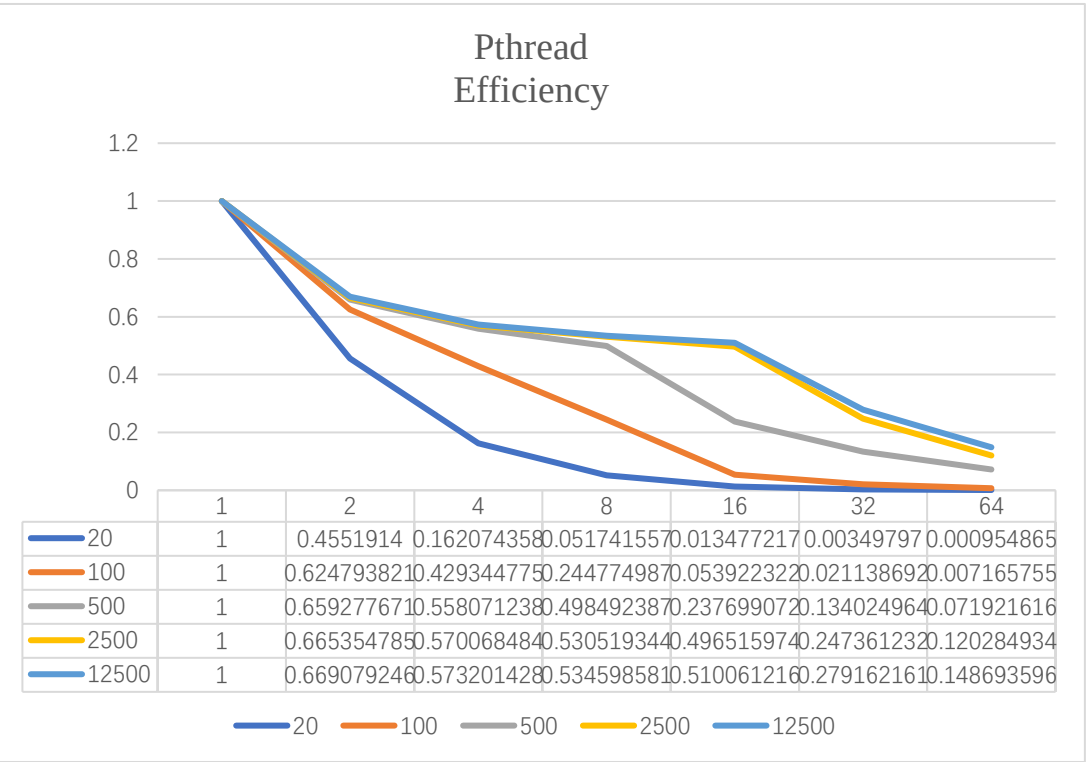
Part 4. Efficiency

The efficiency formula is given as:

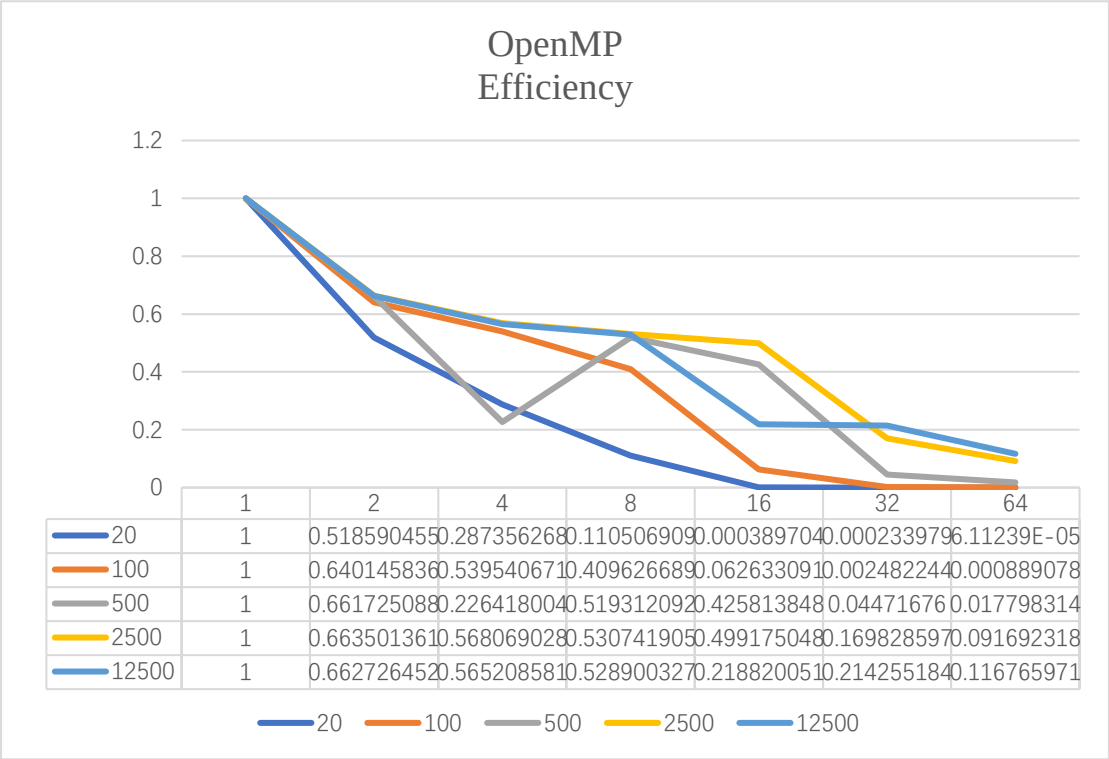
$$E = \frac{Speedup}{number\ of\ processes}$$



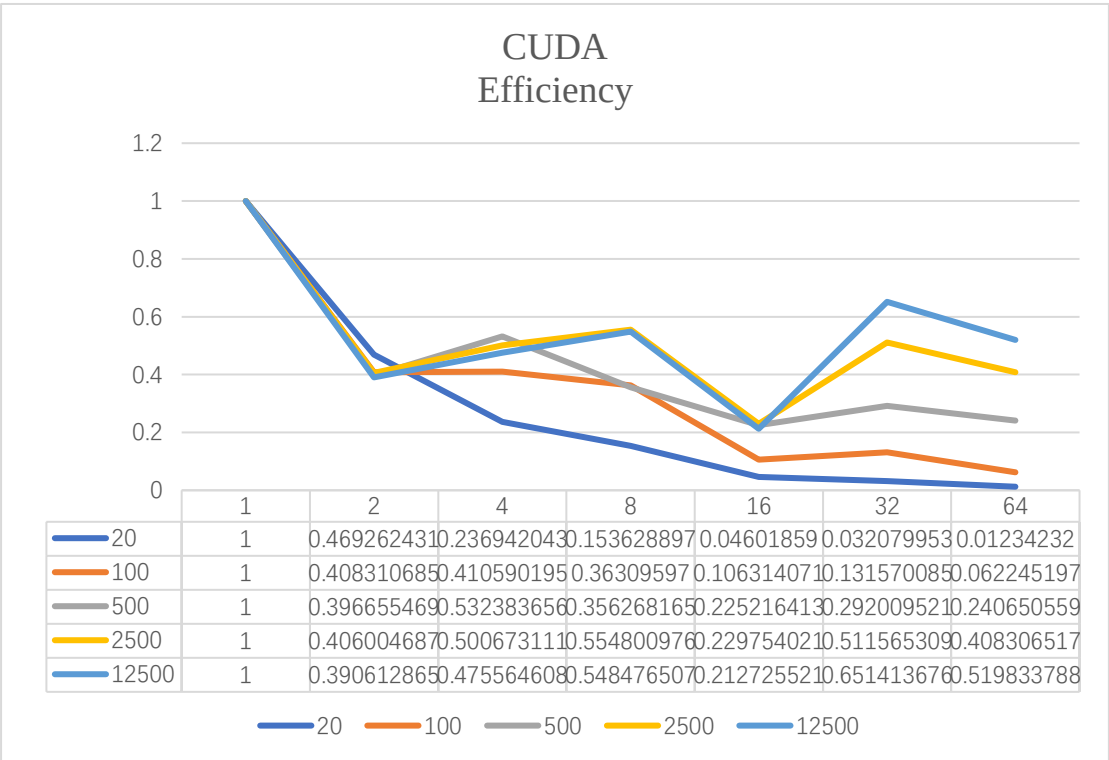
	1	2	4	8	16	32	64
20	1	0.543612903	0.344631074	0.190248969	0.074438136	0.030608242	0.010099178
100	1	0.650627001	0.536377454	0.457254403	0.386468264	0.195110896	0.065814128
500	1	0.665889238	0.57054666	0.524447061	0.495474611	0.398246298	0.378065375
2500	1	0.666723501	0.572871923	0.533550262	0.512926931	0.452090194	0.443672513
12500	1	0.666741433	0.57129492	0.532822903	0.514704729	0.460524079	0.463507666



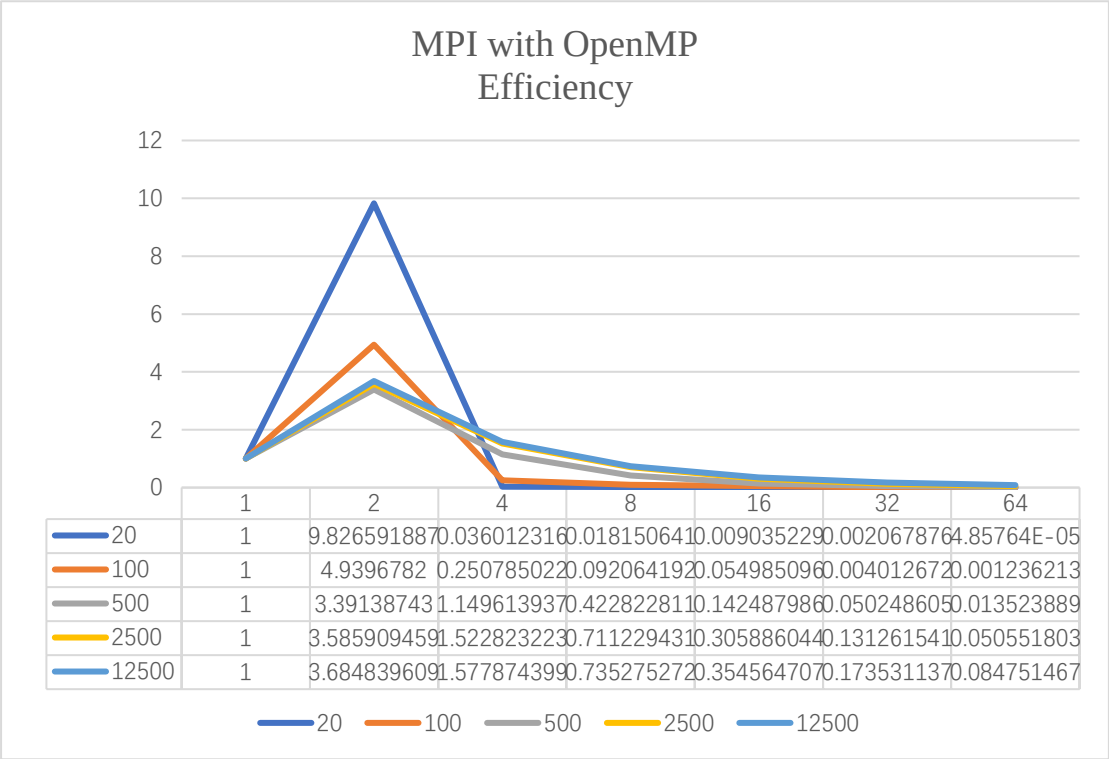
	1	2	4	8	16	32	64
20	1	0.4551914	0.162074358	0.051741557	0.013477217	0.00349797	0.000954865
100	1	0.624793821	0.429344775	0.244774987	0.053922322	0.021138692	0.007165755
500	1	0.659277671	0.558071238	0.498492387	0.237699072	0.134024964	0.071921616
2500	1	0.665354785	0.570068484	0.530519344	0.496515974	0.247361232	0.120284934
12500	1	0.669079246	0.573201428	0.534598581	0.510061216	0.279162161	0.148693596



	1	2	4	8	16	32	64
20	1	0.518590455	0.287356268	0.110506909	0.000389704	0.000233979	6.11239E-05
100	1	0.640145836	0.539540671	0.409626689	0.062633091	0.002482244	0.000889078
500	1	0.661725088	0.226418004	0.519312092	0.425813848	0.04471676	0.017798314
2500	1	0.663501361	0.568069028	0.530741905	0.499175048	0.169828597	0.091692318
12500	1	0.662726452	0.565208581	0.528900327	0.218820051	0.214255184	0.116765971



	1	2	4	8	16	32	64
20	1	0.469262431	0.236942043	0.153628897	0.04601859	0.032079953	0.01234232
100	1	0.408310685	0.410590195	0.36309597	0.106314071	0.131570085	0.062245197
500	1	0.396655469	0.532383656	0.356268165	0.225216413	0.292009521	0.240650559
2500	1	0.406004687	0.500673111	0.554800976	0.229754021	0.511565309	0.408306517
12500	1	0.390612865	0.475564608	0.548476507	0.212725521	0.651413676	0.519833788



Generally, the efficiency would decrease with the increase of size of processes and threads since with more threads or process, each process would be assigned with less subtask size which may spent relatively small time for computing and more time for waiting other processes or threads to finish their tasks (this would be highly possible when some threads need to handle relatively heavy task of computing).

5. Conclusion and Limitations

This report mainly discusses two parallel versions (MPI, Pthread, OpenMP, CUDA, MPI with OpenMP) of the given sequential N-Body sequential program. For sequential version, the design of program considers the potential data race and avoid it. Tests with different number of cores and threads and data size, the report shows the improvement of parallel calculation and also discuss the communication overhead and thread switching may offset the performance improvement brough by parallel calculation especially in OpenMP, CUDA and MPI with OpenMP.

To further improve the computation efficiency, we can separate the tasks into equal time-consuming pieces instead of equal size pieces. Notice that the calculations for each bodies in the required squares may be different (some points may be going to collide with a dozens of bodies and collision may takes a few more steps to compute than normal case). If these, the process or thread assigned with the bodies involved in collisions with many other bodies may takes more time for calculation while other threads or processes could do nothing but waiting. In this case, we can actually divide task into smaller pieces and assign these pieces to each process or thread first. After finishing their jobs, some processes or threads would then require new pieces from the remaining. In this case, the

execution time will be much approximately equal for all processes and threads.

Codes:

1) MPI:

(main.cpp)

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <algorithm>
#include <iostream>
#include <chrono>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

// define struct to pass information
typedef struct Arguments_S {
        int current_bodies;
        float current_space;
        float max_mass;
        float elapse;
        float gravity;
        float radius;
} Arguments;

static MPI_Datatype Arguments_TYPE;

int main(int argc, char **argv) {
    int rank;
    int processSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    // initialize MPI struct
    const int nitems=6;
    int blocklengths[6] = {1,1,1,1,1,1};
    MPI_Datatype types[6] = {MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT,
MPI_FLOAT};
    MPI_Aint     offsets[6];

    offsets[0] = offsetof(Arguments, current_bodies);
    offsets[1] = offsetof(Arguments, current_space);
    offsets[2] = offsetof(Arguments, max_mass);
    offsets[3] = offsetof(Arguments, elapse);
    offsets[4] = offsetof(Arguments, gravity);
    offsets[5] = offsetof(Arguments, radius);
```

```
    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &Arguments_TYPE);
    MPI_Type_commit(&Arguments_TYPE);

    UNUSED(argc, argv);
    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 200;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    static float current_space = space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;
    BodyPool pool(static_cast<size_t>(bodies), space, max_mass);

    if(rank==0){
        graphic::GraphicContext context{"Assignment 3"};
        context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 3", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
            ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
            ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
            ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
            ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
            ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100, "%f");
            ImGui::ColorEdit4("Color", &color.x);

            // boardcast updated plotting information to other ranks
            Arguments arguments;
            arguments.current_bodies = current_bodies;
            arguments.current_space = current_space;
            arguments.max_mass = current_max_mass;
            arguments.elapse = elapse;
            arguments.gravity = gravity;
            arguments.radius = radius;
```

```
            MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);

            if (current_space != space || current_bodies != bodies || current_max_mass !=
max_mass) {
                space = current_space;
                bodies = current_bodies;
                max_mass = current_max_mass;
                pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
            }

            // record calculation time
            auto begin = std::chrono::high_resolution_clock::now();

            // boardcast current information of all bodies
            MPI_Bcast(pool.x.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.y.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vx.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vy.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            // MPI_Bcast(pool.ax.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            // MPI_Bcast(pool.ay.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.m.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);

            {
                const ImVec2 p = ImGui::GetCursorScreenPos();


                // do calculations on the target bodies
                pool.update_for_tick(elapse, gravity, space, radius);

                // gather calculation result from all ranks
                // calculate the job size of each rank
                std::vector<int> taskSizeList;
                for(int i=0;i<processSize;i++){
                    int startPosition = (bodies/processSize)*i +
std::min(bodies%processSize, i);
                    int endPosition = (bodies/processSize)*(i+1) +
std::min(bodies%processSize, i+1)-1;
                    taskSizeList.push_back(endPosition-startPosition+1);
                }

                // calculate the strips of received data packages
                std::vector<int> strips;
                strips.push_back(0);
                for(int i=1;i<processSize;i++){
                    strips.push_back(strips[i-1]+taskSizeList[i-1]);
                }

                // calcualte job size of itself
```

```
                int startPosition = (bodies/processSize)*rank +
std::min(bodies%processSize, rank);
                int endPosition = (bodies/processSize)*(rank+1) +
std::min(bodies%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // rank 0 gather calculated result
                MPI_Gatherv(pool.x.data()+startPosition, jobSize, MPI_DOUBLE,
pool.x.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.y.data()+startPosition, jobSize, MPI_DOUBLE,
pool.y.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.vx.data()+startPosition, jobSize, MPI_DOUBLE,
pool.vx.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.vy.data()+startPosition, jobSize, MPI_DOUBLE,
pool.vy.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.ax.data()+startPosition, jobSize, MPI_DOUBLE,
pool.ax.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.ay.data()+startPosition, jobSize, MPI_DOUBLE,
pool.ay.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                // MPI_Gatherv(pool.m.data()+startPosition, jobSize, MPI_DOUBLE,
pool.m.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);

                // record calculation time
                auto end = std::chrono::high_resolution_clock::now();

                auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
                std::cout << pool.size() << " bodies in last " << duration << "
nanoseconds" << std::endl;

                // plot calculation results
                for (size_t i = 0; i < pool.size(); ++i) {
                    auto body = pool.get_body(i);
                    auto x = p.x + static_cast<float>(body.get_x());
                    auto y = p.y + static_cast<float>(body.get_y());
                    draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
                }


            }
            ImGui::End();
        });
    }
    else{
        while(true){

            // receive updated plotting parameters from rank 0
            Arguments arguments;
            MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);
```

```
            current_bodies = arguments.current_bodies;
            current_space = arguments.current_space;
            current_max_mass = arguments.max_mass;
            elapse = arguments.elapse;
            gravity = arguments.gravity;
            radius = arguments.radius;
            space = current_space;
            max_mass = current_max_mass;

            if (current_bodies != bodies) {
                bodies = current_bodies;
                pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
            }

            // receive current information of all bodies
            MPI_Bcast(pool.x.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.y.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vx.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vy.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            // MPI_Bcast(pool.ax.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            // MPI_Bcast(pool.ay.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.m.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);

            // do calculations on the target bodies
            pool.update_for_tick(elapse, gravity, space, radius);

            // calcualte job size of itself
            int startPosition = (bodies/processSize)*rank + std::min(bodies%processSize,
rank);
            int endPosition = (bodies/processSize)*(rank+1) + std::min(bodies%processSize,
rank+1)-1;
            int jobSize = endPosition-startPosition+1;

            // send calculation result
            MPI_Gatherv(pool.x.data()+startPosition, jobSize, MPI_DOUBLE, nullptr, nullptr,
nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.y.data()+startPosition, jobSize, MPI_DOUBLE, nullptr, nullptr,
nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.vx.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.vy.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.ax.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.ay.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            // MPI_Gatherv(pool.m.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
```

```
    }

    // if ploting windows closed, rank 0 would then stop all other processes
    if(rank==0){
        std::cout << "aborting all processes" << std::endl;
        MPI_Abort(MPI_COMM_WORLD, 0);
    }

    MPI_Type_free(&Arguments_TYPE);
    MPI_Finalize();

}
```

(body.hpp)

```
//
// Created by schrodinger on 11/2/21.
//
#pragma once

#include <random>
#include <utility>
#include <mpi.h>
#include <algorithm>

#include <iostream>

class BodyPool {
    // if after the collision, we do not separate the bodies a little bit, it may
    // results in strange outcomes like infinite acceleration.
    // hence, we will need to set up a ratio for separation.
    static constexpr double COLLISION_RATIO = 0.01;

public:
    // provides in this way so that
    // it is easier for you to send a the vector with MPI
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> vx;
    std::vector<double> vy;
    std::vector<double> ax;
    std::vector<double> ay;
    std::vector<double> m;
    // so the movements of bodies are calculated discretely.

    class Body {
        size_t index;
        BodyPool &pool;
```

```
    friend class BodyPool;

    Body(size_t index, BodyPool &pool) : index(index), pool(pool) {}

public:
    double &get_x() {
        return pool.x[index];
    }

    double &get_y() {
        return pool.y[index];
    }

    double &get_vx() {
        return pool.vx[index];
    }

    double &get_vy() {
        return pool.vy[index];
    }

    double &get_ax() {
        return pool.ax[index];
    }

    double &get_ay() {
        return pool.ay[index];
    }

    double &get_m() {
        return pool.m[index];
    }

    double distance_square(Body &that) {
        auto delta_x = get_x() - that.get_x();
        auto delta_y = get_y() - that.get_y();
        return delta_x * delta_x + delta_y * delta_y;
    }

    double distance(Body &that) {
        return std::sqrt(distance_square(that));
    }

    double delta_x(Body &that) {
        return get_x() - that.get_x();
    }

    double delta_y(Body &that) {
```

```
        return get_y() - that.get_y();
    }

    bool collide(Body &that, double radius) {
        return distance_square(that) <= radius * radius;
    }

    // collision with wall
    void handle_wall_collision(double position_range, double radius) {
        bool flag = false;
        if (get_x() <= radius) {
            flag = true;
            get_x() = radius + radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        } else if (get_x() >= position_range - radius) {
            flag = true;
            get_x() = position_range - radius - radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        }

        if (get_y() <= radius) {
            flag = true;
            get_y() = radius + radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        } else if (get_y() >= position_range - radius) {
            flag = true;
            get_y() = position_range - radius - radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        }
        if (flag) {
            get_ax() = 0;
            get_ay() = 0;
        }
    }

    void update_for_tick(
            double elapse,
            double position_range,
            double radius) {
        get_vx() += get_ax() * elapse;
        get_vy() += get_ay() * elapse;
        handle_wall_collision(position_range, radius);
        get_x() += get_vx() * elapse;
        get_y() += get_vy() * elapse;
        handle_wall_collision(position_range, radius);
    }

};
```

```
    BodyPool(size_t size, double position_range, double mass_range) :
            x(size), y(size), vx(size), vy(size), ax(size), ay(size), m(size) {
        std::random_device device;
        std::default_random_engine engine{device()};
        std::uniform_real_distribution<double> position_dist{0, position_range};
        std::uniform_real_distribution<double> mass_dist{0, mass_range};
        for (auto &i : x) {
            i = position_dist(engine);
        }
        for (auto &i : y) {
            i = position_dist(engine);
        }
        for (auto &i : vx) {
            i = 0;
        }
        for (auto &i : vy) {
            i = 0;
        }
        for (auto &i : m) {
            i = mass_dist(engine);
        }
    }

    Body get_body(size_t index) {
        return {index, *this};
    }

    void clear_acceleration() {
        ax.assign(m.size(), 0.0);
        ay.assign(m.size(), 0.0);
    }

    size_t size() {
        return m.size();
    }

    static void check_and_update(Body i, Body j, double radius, double gravity) {
        auto delta_x = i.delta_x(j);
        auto delta_y = i.delta_y(j);
        auto distance_square = i.distance_square(j);
        auto ratio = 1 + COLLISION_RATIO;
        if (distance_square < radius * radius) {
            distance_square = radius * radius;
        }
        auto distance = i.distance(j);
        if (distance < radius) {
            distance = radius;
        }
        if (i.collide(j, radius)) {
```

```
            auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                            + delta_y * (i.get_vy() - j.get_vy());
            auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
            i.get_vx() -= scalar * delta_x * j.get_m();
            i.get_vy() -= scalar * delta_y * j.get_m();
            j.get_vx() += scalar * delta_x * i.get_m();
            j.get_vy() += scalar * delta_y * i.get_m();
            // now relax the distance a bit: after the collision, there must be
            // at least (ratio * radius) between them
            i.get_x() += delta_x / distance * ratio * radius / 2.0;
            i.get_y() += delta_y / distance * ratio * radius / 2.0;
            j.get_x() -= delta_x / distance * ratio * radius / 2.0;
            j.get_y() -= delta_y / distance * ratio * radius / 2.0;
        } else {
            // update acceleration only when no collision
            auto scalar = gravity / distance_square / distance;
            i.get_ax() -= scalar * delta_x * j.get_m();
            i.get_ay() -= scalar * delta_y * j.get_m();
            j.get_ax() += scalar * delta_x * i.get_m();
            j.get_ay() += scalar * delta_y * i.get_m();
        }
    }

    void update_for_tick(double elapse,
                         double gravity,
                         double position_range,
                         double radius) {
        ax.assign(size(), 0);
        ay.assign(size(), 0);

        int rank;
        int processSize;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &processSize);

        int startPosition = (size()/processSize)*rank + std::min(static_cast<int>
(size())%processSize, rank);
        int endPosition = (size()/processSize)*(rank+1) + std::min(static_cast<int>
(size())%processSize, rank+1)-1;

        // std::cout << rank << " " << startPosition << " " << endPosition << std::endl;

        // consider the situations of bodies between that outsides the range and inside the
range
        for (size_t i = 0; i < static_cast<size_t>(startPosition); ++i) {
            for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j) {
                check_and_update(get_body(i), get_body(j), radius, gravity);
            }
        }
```

```
        for (size_t i = endPosition+1; i < size(); ++i) {
            for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j) {
                check_and_update(get_body(i), get_body(j), radius, gravity);
            }
        }

        // consider the situations of bodies inside the range
        for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
            for (size_t j = i + 1; j <= static_cast<size_t>(endPosition); ++j) {
                check_and_update(get_body(i), get_body(j), radius, gravity);
            }
        }

        // only update the position of bodies inside the range
        for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
            get_body(i).update_for_tick(elapse, position_range, radius);
        }
    }
};
```

2) Pthread

(main.cpp)

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <algorithm>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

int main(int argc, char **argv) {
    // use 4 threadsd as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }
```

```
    UNUSED(argc, argv);
    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 200;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    static float current_space = space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;
    BodyPool pool(static_cast<size_t>(bodies), space, max_mass);

    graphic::GraphicContext context{"Assignment 3"};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 3", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
        ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
        ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
        ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
        ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
        ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100, "%f");
        ImGui::ColorEdit4("Color", &color.x);

        if (current_space != space || current_bodies != bodies || current_max_mass !=
max_mass) {
            space = current_space;
            bodies = current_bodies;
            max_mass = current_max_mass;
            pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
        }

        {
            const ImVec2 p = ImGui::GetCursorScreenPos();

            // record calculation time
            auto begin = std::chrono::high_resolution_clock::now();
```

```
            pool.update_for_tick(elapse, gravity, space, radius, totalThreadSize, &pool);

            // record calculation time
            auto end = std::chrono::high_resolution_clock::now();

            auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
            std::cout << pool.size() << " bodies in last " << duration << " nanoseconds" <<
std::endl;

            // plot calculation results
            for (size_t i = 0; i < pool.size(); ++i) {
                auto body = pool.get_body(i);
                auto x = p.x + static_cast<float>(body.get_x());
                auto y = p.y + static_cast<float>(body.get_y());
                draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
            }

        }
        ImGui::End();
    });

}
```

(body.cpp)

```
//
// Created by schrodinger on 11/2/21.
//
#pragma once

#include <random>
#include <utility>
#include <mpi.h>
#include <algorithm>

#include <iostream>

class BodyPool;

// struct to hold parameters
struct Arguments{
    BodyPool * pool;
    double elapse;
    double gravity;
    double position_range;
    double radius;
    int taskNum;
```

```
    int totalThreadSize;
};

// use to hold the barrier
static pthread_barrier_t barrier;

class BodyPool {
    // if after the collision, we do not separate the bodies a little bit, it may
    // results in strange outcomes like infinite acceleration.
    // hence, we will need to set up a ratio for separation.
    static constexpr double COLLISION_RATIO = 0.01;

public:
    // provides in this way so that
    // it is easier for you to send a the vector with MPI
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> vx;
    std::vector<double> vy;
    std::vector<double> ax;
    std::vector<double> ay;
    std::vector<double> m;
    // so the movements of bodies are calculated discretely.

    class Body {
        size_t index;
        BodyPool &pool;

        friend class BodyPool;

        Body(size_t index, BodyPool &pool) : index(index), pool(pool) {}

    public:
        double &get_x() {
            return pool.x[index];
        }

        double &get_y() {
            return pool.y[index];
        }

        double &get_vx() {
            return pool.vx[index];
        }

        double &get_vy() {
            return pool.vy[index];
        }
```

```
        double &get_ax() {
            return pool.ax[index];
        }

        double &get_ay() {
            return pool.ay[index];
        }

        double &get_m() {
            return pool.m[index];
        }

        double distance_square(Body &that) {
            auto delta_x = get_x() - that.get_x();
            auto delta_y = get_y() - that.get_y();
            return delta_x * delta_x + delta_y * delta_y;
        }

        double distance(Body &that) {
            return std::sqrt(distance_square(that));
        }

        double delta_x(Body &that) {
            return get_x() - that.get_x();
        }

        double delta_y(Body &that) {
            return get_y() - that.get_y();
        }

        bool collide(Body &that, double radius) {
            return distance_square(that) <= radius * radius;
        }

        // collision with wall
        void handle_wall_collision(double position_range, double radius) {
            bool flag = false;
            if (get_x() <= radius) {
                flag = true;
                get_x() = radius + radius * COLLISION_RATIO;
                get_vx() = -get_vx();
            } else if (get_x() >= position_range - radius) {
                flag = true;
                get_x() = position_range - radius - radius * COLLISION_RATIO;
                get_vx() = -get_vx();
            }

            if (get_y() <= radius) {
                flag = true;
```

```
            get_y() = radius + radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        } else if (get_y() >= position_range - radius) {
            flag = true;
            get_y() = position_range - radius - radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        }
        if (flag) {
            get_ax() = 0;
            get_ay() = 0;
        }
    }

    void update_for_tick(
            double elapse,
            double position_range,
            double radius) {
        get_vx() += get_ax() * elapse;
        get_vy() += get_ay() * elapse;
        handle_wall_collision(position_range, radius);
        get_x() += get_vx() * elapse;
        get_y() += get_vy() * elapse;
        handle_wall_collision(position_range, radius);
    }

};

BodyPool(size_t size, double position_range, double mass_range) :
        x(size), y(size), vx(size), vy(size), ax(size), ay(size), m(size) {
    std::random_device device;
    std::default_random_engine engine{device()};
    std::uniform_real_distribution<double> position_dist{0, position_range};
    std::uniform_real_distribution<double> mass_dist{0, mass_range};
    for (auto &i : x) {
        i = position_dist(engine);
    }
    for (auto &i : y) {
        i = position_dist(engine);
    }
    for (auto &i : vx) {
        i = 0;
    }
    for (auto &i : vy) {
        i = 0;
    }
    for (auto &i : m) {
        i = mass_dist(engine);
    }
}
```

```
Body get_body(size_t index) {
    return {index, *this};
}

void clear_acceleration() {
    ax.assign(m.size(), 0.0);
    ay.assign(m.size(), 0.0);
}

size_t size() {
    return m.size();
}

static void check_and_update(Body i, Body j, double radius, double gravity) {
    auto delta_x = i.delta_x(j);
    auto delta_y = i.delta_y(j);
    auto distance_square = i.distance_square(j);
    auto ratio = 1 + COLLISION_RATIO;
    if (distance_square < radius * radius) {
        distance_square = radius * radius;
    }
    auto distance = i.distance(j);
    if (distance < radius) {
        distance = radius;
    }
    if (i.collide(j, radius)) {
        auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                        + delta_y * (i.get_vy() - j.get_vy());
        auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
        i.get_vx() -= scalar * delta_x * j.get_m();
        i.get_vy() -= scalar * delta_y * j.get_m();
        j.get_vx() += scalar * delta_x * i.get_m();
        j.get_vy() += scalar * delta_y * i.get_m();
        // now relax the distance a bit: after the collision, there must be
        // at least (ratio * radius) between them
        i.get_x() += delta_x / distance * ratio * radius / 2.0;
        i.get_y() += delta_y / distance * ratio * radius / 2.0;
        j.get_x() -= delta_x / distance * ratio * radius / 2.0;
        j.get_y() -= delta_y / distance * ratio * radius / 2.0;
    } else {
        // update acceleration only when no collision
        auto scalar = gravity / distance_square / distance;
        i.get_ax() -= scalar * delta_x * j.get_m();
        i.get_ay() -= scalar * delta_y * j.get_m();
        j.get_ax() += scalar * delta_x * i.get_m();
        j.get_ay() += scalar * delta_y * i.get_m();
    }
}
```

```
    void update_for_tick(double elapse, double gravity, double position_range, double
radius, int totalThreadSize, BodyPool * pool) {
        ax.assign(size(), 0);
        ay.assign(size(), 0);

        std::vector<pthread_t> threads(totalThreadSize);

        pthread_barrier_init(&barrier, NULL, totalThreadSize);

        // initialize each thread with task information
        for(int i=0;i<totalThreadSize;i++){
            pthread_create(&threads[i], nullptr, update_for_tick_inside, new Arguments{
                    .pool = pool,
                    .elapse = elapse,
                    .gravity = gravity,
                    .position_range = position_range,
                    .radius = radius,
                    .taskNum = i,
                    .totalThreadSize = totalThreadSize
            });
        }

        // wait until all the threads finish
        for (auto & i : threads) {
            pthread_join(i, nullptr);
        }

        pthread_barrier_destroy(&barrier);
        // printf("\n\n");

    }

    static void * update_for_tick_inside(void *arg_ptr) {
        // retrieve information from passed parameters
        auto arguments = static_cast<Arguments *>(arg_ptr);
        BodyPool * pool = arguments->pool;
        BodyPool copyPool = *pool;

        // wait until all threads has finish copying data in bodypool
        pthread_barrier_wait(&barrier);

        double elapse = arguments->elapse;
        double gravity = arguments->gravity;
        double position_range = arguments->position_range;
        double radius = arguments->radius;
        int taskNum = arguments->taskNum;
        int totalThreadSize = arguments->totalThreadSize;
```

```
        int startPosition = (copyPool.size()/totalThreadSize)*taskNum +
std::min(static_cast<int> (copyPool.size())%totalThreadSize, taskNum);
        int endPosition = (copyPool.size()/totalThreadSize)*(taskNum+1) +
std::min(static_cast<int> (copyPool.size())%totalThreadSize, taskNum+1)-1;

        // printf("%d %d %d\n", taskNum, startPosition, endPosition);

        // consider the situations of bodies between that outsides the range and inside the
range
        for (size_t i = 0; i < static_cast<size_t>(startPosition); ++i) {
            for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j) {
                copyPool.check_and_update(copyPool.get_body(i), copyPool.get_body(j),
radius, gravity);
            }
        }
        for (size_t i = endPosition+1; i < copyPool.size(); ++i) {
            for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j) {
                copyPool.check_and_update(copyPool.get_body(i), copyPool.get_body(j),
radius, gravity);
            }
        }

        // consider the situations of bodies inside the range
        for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
            for (size_t j = i + 1; j <= static_cast<size_t>(endPosition); ++j) {
                copyPool.check_and_update(copyPool.get_body(i), copyPool.get_body(j),
radius, gravity);
            }
        }

        // only update the position of bodies inside the range
        for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
            copyPool.get_body(i).update_for_tick(elapse, position_range, radius);
        }

        for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
            // printf("%d x[%zu] %f %f\n", taskNum, i, copyPool.x[i], pool->x[i]);
            pool->x[i] = copyPool.x[i];
            pool->y[i] = copyPool.y[i];
            pool->vx[i] = copyPool.vx[i];
            pool->vy[i] = copyPool.vy[i];
            pool->ax[i] = copyPool.ax[i];
            pool->ay[i] = copyPool.ay[i];
        }

        return nullptr;
    }

};
```

3) OpenMP

(main.cpp)

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <algorithm>
#include <iostream>
#include <chrono>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

int main(int argc, char **argv) {
    // use 4 threads as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    printf("launch %d threads\n", totalThreadSize);

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }

    UNUSED(argc, argv);
    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 200;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    static float current_space = space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;
    BodyPool pool(static_cast<size_t>(bodies), space, max_mass);

    omp_set_num_threads(totalThreadSize);
```

```
    graphic::GraphicContext context{"Assignment 3"};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 3", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
        ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
        ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
        ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
        ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
        ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100, "%f");
        ImGui::ColorEdit4("Color", &color.x);

        if (current_space != space || current_bodies != bodies || current_max_mass !=
max_mass) {
            space = current_space;
            bodies = current_bodies;
            max_mass = current_max_mass;
            pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
        }

        {
            const ImVec2 p = ImGui::GetCursorScreenPos();

            // record calculation time
            auto begin = std::chrono::high_resolution_clock::now();

            pool.update_for_tick(elapse, gravity, space, radius, totalThreadSize, &pool);

            // record calculation time
            auto end = std::chrono::high_resolution_clock::now();

            auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
            std::cout << pool.size() << " bodies in last " << duration << " nanoseconds" <<
std::endl;

            // plot calculation results
            for (size_t i = 0; i < pool.size(); ++i) {
                auto body = pool.get_body(i);
```

```
                auto x = p.x + static_cast<float>(body.get_x());
                auto y = p.y + static_cast<float>(body.get_y());
                draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
            }

        }
        ImGui::End();
    });

}
```

(body.hpp)

```
//
// Created by schrodinger on 11/2/21.
//
#pragma once

#include <random>
#include <utility>
#include <omp.h>

class BodyPool {
    // provides in this way so that
    // it is easier for you to send a the vector with MPI
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> vx;
    std::vector<double> vy;
    std::vector<double> ax;
    std::vector<double> ay;
    std::vector<double> m;
    // so the movements of bodies are calculated discretely.
    // if after the collision, we do not separate the bodies a little bit, it may
    // results in strange outcomes like infinite acceleration.
    // hence, we will need to set up a ratio for separation.
    static constexpr double COLLISION_RATIO = 0.01;
public:

    class Body {
        size_t index;
        BodyPool &pool;

        friend class BodyPool;

        Body(size_t index, BodyPool &pool) : index(index), pool(pool) {}

    public:
```

```
        double &get_x() {
            return pool.x[index];
        }

        double &get_y() {
            return pool.y[index];
        }

        double &get_vx() {
            return pool.vx[index];
        }

        double &get_vy() {
            return pool.vy[index];
        }

        double &get_ax() {
            return pool.ax[index];
        }

        double &get_ay() {
            return pool.ay[index];
        }

        double &get_m() {
            return pool.m[index];
        }

        double distance_square(Body &that) {
            auto delta_x = get_x() - that.get_x();
            auto delta_y = get_y() - that.get_y();
            return delta_x * delta_x + delta_y * delta_y;
        }

        double distance(Body &that) {
            return std::sqrt(distance_square(that));
        }

        double delta_x(Body &that) {
            return get_x() - that.get_x();
        }

        double delta_y(Body &that) {
            return get_y() - that.get_y();
        }

        bool collide(Body &that, double radius) {
            return distance_square(that) <= radius * radius;
        }
```

```
    // collision with wall
    void handle_wall_collision(double position_range, double radius) {
        bool flag = false;
        if (get_x() <= radius) {
            flag = true;
            get_x() = radius + radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        } else if (get_x() >= position_range - radius) {
            flag = true;
            get_x() = position_range - radius - radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        }

        if (get_y() <= radius) {
            flag = true;
            get_y() = radius + radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        } else if (get_y() >= position_range - radius) {
            flag = true;
            get_y() = position_range - radius - radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        }
        if (flag) {
            get_ax() = 0;
            get_ay() = 0;
        }
    }

    void update_for_tick(
            double elapse,
            double position_range,
            double radius) {
        get_vx() += get_ax() * elapse;
        get_vy() += get_ay() * elapse;
        handle_wall_collision(position_range, radius);
        get_x() += get_vx() * elapse;
        get_y() += get_vy() * elapse;
        handle_wall_collision(position_range, radius);
    }

};

BodyPool(size_t size, double position_range, double mass_range) :
        x(size), y(size), vx(size), vy(size), ax(size), ay(size), m(size) {
    std::random_device device;
    std::default_random_engine engine{device()};
    std::uniform_real_distribution<double> position_dist{0, position_range};
    std::uniform_real_distribution<double> mass_dist{0, mass_range};
```

```
        for (auto &i : x) {
            i = position_dist(engine);
        }
        for (auto &i : y) {
            i = position_dist(engine);
        }
        for (auto &i : vx) {
            i = 0;
        }
        for (auto &i : vy) {
            i = 0;
        }
        for (auto &i : m) {
            i = mass_dist(engine);
        }
    }

    Body get_body(size_t index) {
        return {index, *this};
    }

    void clear_acceleration() {
        ax.assign(m.size(), 0.0);
        ay.assign(m.size(), 0.0);
    }

    size_t size() {
        return m.size();
    }

    static void check_and_update(Body i, Body j, double radius, double gravity) {
        auto delta_x = i.delta_x(j);
        auto delta_y = i.delta_y(j);
        auto distance_square = i.distance_square(j);
        auto ratio = 1 + COLLISION_RATIO;
        if (distance_square < radius * radius) {
            distance_square = radius * radius;
        }
        auto distance = i.distance(j);
        if (distance < radius) {
            distance = radius;
        }
        if (i.collide(j, radius)) {
            auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                            + delta_y * (i.get_vy() - j.get_vy());
            auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
            i.get_vx() -= scalar * delta_x * j.get_m();
            i.get_vy() -= scalar * delta_y * j.get_m();
            j.get_vx() += scalar * delta_x * i.get_m();
```

```
            j.get_vy() += scalar * delta_y * i.get_m();
            // now relax the distance a bit: after the collision, there must be
            // at least (ratio * radius) between them
            i.get_x() += delta_x / distance * ratio * radius / 2.0;
            i.get_y() += delta_y / distance * ratio * radius / 2.0;
            j.get_x() -= delta_x / distance * ratio * radius / 2.0;
            j.get_y() -= delta_y / distance * ratio * radius / 2.0;
        } else {
            // update acceleration only when no collision
            auto scalar = gravity / distance_square / distance;
            i.get_ax() -= scalar * delta_x * j.get_m();
            i.get_ay() -= scalar * delta_y * j.get_m();
            j.get_ax() += scalar * delta_x * i.get_m();
            j.get_ay() += scalar * delta_y * i.get_m();
        }
    }

    void update_for_tick(double elapse,
                         double gravity,
                         double position_range,
                         double radius,
                         int totalThreadSize,
                         BodyPool * pool) {
        ax.assign(size(), 0);
        ay.assign(size(), 0);

        #pragma omp parallel
        {
            BodyPool copyPool = *pool;

            // make sure all threads finish copy
        #pragma omp barrier

            int taskNum = omp_get_thread_num();

            int startPosition = (size()/totalThreadSize)*taskNum +
std::min(static_cast<int> (size())%totalThreadSize, taskNum);
            int endPosition = (size()/totalThreadSize)*(taskNum+1) +
std::min(static_cast<int> (size())%totalThreadSize, taskNum+1)-1;

            // printf("%d %d %d\n", taskNum, startPosition, endPosition);

            // consider the situations of bodies between that outsides the range and inside
the range
            for (size_t i = 0; i < static_cast<size_t>(startPosition); ++i) {
                for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j)
{
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
```

```
                }
            }
            for (size_t i = endPosition+1; i < size(); ++i) {
                for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j)
{
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
                }
            }

            // consider the situations of bodies inside the range
            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                for (size_t j = i + 1; j <= static_cast<size_t>(endPosition); ++j) {
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
                }
            }

            // only update the position of bodies inside the range
            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                copyPool.get_body(i).update_for_tick(elapse, position_range, radius);
            }

            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                // printf("%d x[%zu] %f %f\n", taskNum, i, copyPool.x[i], pool->x[i]);
                pool->x[i] = copyPool.x[i];
                pool->y[i] = copyPool.y[i];
                pool->vx[i] = copyPool.vx[i];
                pool->vy[i] = copyPool.vy[i];
                pool->ax[i] = copyPool.ax[i];
                pool->ay[i] = copyPool.ay[i];
            }

        }

        // printf("\n\n");

    }
};
```

4) CUDA

(main.cpp)

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
```

```cpp
#include <iostream>
#include <chrono>
#include <algorithm>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

int atoiCustomized(char ** argv, int argc, int argvIndex){
    int result = 1;

    if(argc>argvIndex){
        result = atoi(argv[argvIndex]);
    }

    // avoid invalid input
    if(result==0){
        result = 1;
    }

    return result;
}

int main(int argc, char **argv) {
    dim3 grid_size;
    dim3 block_size;

    // use 1 as default value
    grid_size.x = atoiCustomized(argv, argc, 1);
    grid_size.y = atoiCustomized(argv, argc, 2);
    grid_size.z = atoiCustomized(argv, argc, 3);
    block_size.x = atoiCustomized(argv, argc, 4);
    block_size.y = atoiCustomized(argv, argc, 5);
    block_size.z = atoiCustomized(argv, argc, 6);

    // cudaThreadSetLimit(cudaLimitMallocHeapSize, 10*100*sizeof(double));

    printf("dim: %d %d %d  block: %d %d %d\n", grid_size.x, grid_size.y, grid_size.z,
           block_size.x, block_size.y, block_size.z);

    UNUSED(argc, argv);
    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 200;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    static float current_space = space;
    static float current_max_mass = max_mass;
```

```
    static int current_bodies = bodies;
    static bool nanOccur = false;

    BodyPool * pool = new BodyPool(static_cast<size_t>(bodies), space, max_mass);

    graphic::GraphicContext context{"Assignment 3"};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 3", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
        ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
        ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
        ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
        ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
        ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100, "%f");
        ImGui::ColorEdit4("Color", &color.x);

        // reconstruct the map when necessary
        if (nanOccur == true || current_space != space || current_bodies != bodies ||
current_max_mass != max_mass) {
            space = current_space;
            bodies = current_bodies;
            max_mass = current_max_mass;
            free(pool);
            pool = new BodyPool(static_cast<size_t>(bodies), space, max_mass);
            nanOccur = false;
        }

        {
            const ImVec2 p = ImGui::GetCursorScreenPos();

            // record calculation time
            auto begin = std::chrono::high_resolution_clock::now();

            // calculate information for each body
            pool->update_for_tick(elapse, gravity, space, radius, grid_size, block_size,
pool);

            // record calculation time
```

```
            auto end = std::chrono::high_resolution_clock::now();

            auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
            std::cout << pool->size() << " bodies in last " << duration << " nanoseconds"
<< std::endl;

            // plot calculation results
            for (int i = 0; i < pool->size(); ++i) {
                auto body = pool->get_body(i);
                if(isnan(body.get_x()) || isnan(body.get_y())){
                    nanOccur = true;
                    printf("nan occured bodypool reset\n");
                    break;
                }
                auto x = p.x + static_cast<float>(body.get_x());
                auto y = p.y + static_cast<float>(body.get_y());
                draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
            }

        }
        ImGui::End();
    });

}
```

(body.hpp)

```
//
// Created by schrodinger on 11/2/21.
//
#pragma once

#include <random>
#include <utility>
#include <omp.h>

class BodyPool;

__device__
int getBlockId() {
  return blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
}

__device__
int getLocalThreadId() {
  return (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) +
threadIdx.x;
```

```
}

__device__
int getThreadId() {
  int blockId = getBlockId();
  int localThreadId = getLocalThreadId();
  return blockId * (blockDim.x * blockDim.y * blockDim.z) + localThreadId;
}

__global__
void cuda_update_for_tick(double elapse,
                          double gravity,
                          double position_range,
                          double radius,
                          int totalThreadSize,
                          BodyPool * pool);

__global__
void test(int testNum);


class BodyPool {
    // so the movements of bodies are calculated discretely.
    // if after the collision, we do not separate the bodies a little bit, it may
    // results in strange outcomes like infinite acceleration.
    // hence, we will need to set up a ratio for separation.
    static constexpr double COLLISION_RATIO = 0.01;

public:
    // provides in this way so that
    // it is easier for you to send a the vector with MPI
    double * x;
    double * y;
    double * vx;
    double * vy;
    double * ax;
    double * ay;
    double * m;
    int _size;

    class Body {
        int index;
        BodyPool &pool;

        friend class BodyPool;

        __device__ __host__
        Body(int index, BodyPool &pool) : index(index), pool(pool) {}
```

```
public:
    __device__ __host__
    double &get_x() {
        return pool.x[index];
    }
    __device__ __host__
    double &get_y() {
        return pool.y[index];
    }
    __device__ __host__
    double &get_vx() {
        return pool.vx[index];
    }
    __device__ __host__
    double &get_vy() {
        return pool.vy[index];
    }
    __device__ __host__
    double &get_ax() {
        return pool.ax[index];
    }
    __device__ __host__
    double &get_ay() {
        return pool.ay[index];
    }
    __device__ __host__
    double &get_m() {
        return pool.m[index];
    }
    __device__ __host__
    double distance_square(Body &that) {
        auto delta_x = get_x() - that.get_x();
        auto delta_y = get_y() - that.get_y();
        return delta_x * delta_x + delta_y * delta_y;
    }
    __device__ __host__
    double distance(Body &that) {
        return std::sqrt(distance_square(that));
    }
    __device__ __host__
    double delta_x(Body &that) {
        return get_x() - that.get_x();
    }
    __device__ __host__
    double delta_y(Body &that) {
        return get_y() - that.get_y();
    }
    __device__ __host__
    bool collide(Body &that, double radius) {
```

```
        return distance_square(that) <= radius * radius;
    }

    // collision with wall
    __device__ __host__
    void handle_wall_collision(double position_range, double radius) {
        bool flag = false;
        if (get_x() <= radius) {
            flag = true;
            get_x() = radius + radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        } else if (get_x() >= position_range - radius) {
            flag = true;
            get_x() = position_range - radius - radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        }

        if (get_y() <= radius) {
            flag = true;
            get_y() = radius + radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        } else if (get_y() >= position_range - radius) {
            flag = true;
            get_y() = position_range - radius - radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        }
        if (flag) {
            get_ax() = 0;
            get_ay() = 0;
        }
    }

    __device__ __host__
    void update_for_tick(
            double elapse,
            double position_range,
            double radius) {
        get_vx() += get_ax() * elapse;
        get_vy() += get_ay() * elapse;
        handle_wall_collision(position_range, radius);
        get_x() += get_vx() * elapse;
        get_y() += get_vy() * elapse;
        handle_wall_collision(position_range, radius);
    }

};

__host__
BodyPool(size_t size, double position_range, double mass_range){
```

```
    _size = size;

    x = (double*)malloc(size*sizeof(double));
    y = (double*)malloc(size*sizeof(double));
    vx = (double*)malloc(size*sizeof(double));
    vy = (double*)malloc(size*sizeof(double));
    ax = (double*)malloc(size*sizeof(double));
    ay = (double*)malloc(size*sizeof(double));
    m = (double*)malloc(size*sizeof(double));

    std::random_device device;
    std::default_random_engine engine{device()};
    std::uniform_real_distribution<double> position_dist{0, position_range};
    std::uniform_real_distribution<double> mass_dist{0, mass_range};
    for (int i=0;i<size;i++) {
        x[i] = position_dist(engine);
        y[i] = position_dist(engine);
        m[i] = mass_dist(engine);
        vx[i] = 0;
        vy[i] = 0;
        ax[i] = 0;
        ay[i] = 0;
    }
}

__host__
~BodyPool(){
    free(x);
    free(y);
    free(vx);
    free(vy);
    free(ax);
    free(ay);
    free(m);
}

__device__ __host__
Body get_body(int index) {
    return {index, *this};
}

__device__ __host__
void clear_acceleration() {
    memset(ax, 0, sizeof(ax));
    memset(ay, 0, sizeof(ay));
}

__device__ __host__
int size() {
```

```
    return _size;
}

__device__ __host__
static void check_and_update(Body i, Body j, double radius, double gravity) {
    auto delta_x = i.delta_x(j);
    auto delta_y = i.delta_y(j);
    auto distance_square = i.distance_square(j);
    auto ratio = 1 + COLLISION_RATIO;
    if (distance_square < radius * radius) {
        distance_square = radius * radius;
    }
    auto distance = i.distance(j);
    if (distance < radius) {
        distance = radius;
    }
    if (i.collide(j, radius)) {
        auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                      + delta_y * (i.get_vy() - j.get_vy());
        auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
        i.get_vx() -= scalar * delta_x * j.get_m();
        i.get_vy() -= scalar * delta_y * j.get_m();
        j.get_vx() += scalar * delta_x * i.get_m();
        j.get_vy() += scalar * delta_y * i.get_m();
        // now relax the distance a bit: after the collision, there must be
        // at least (ratio * radius) between them
        i.get_x() += delta_x / distance * ratio * radius / 2.0;
        i.get_y() += delta_y / distance * ratio * radius / 2.0;
        j.get_x() -= delta_x / distance * ratio * radius / 2.0;
        j.get_y() -= delta_y / distance * ratio * radius / 2.0;
    } else {
        // update acceleration only when no collision
        auto scalar = gravity / distance_square / distance;
        i.get_ax() -= scalar * delta_x * j.get_m();
        i.get_ay() -= scalar * delta_y * j.get_m();
        j.get_ax() += scalar * delta_x * i.get_m();
        j.get_ay() += scalar * delta_y * i.get_m();
    }
}

__host__
void update_for_tick(double elapse,
                     double gravity,
                     double position_range,
                     double radius,
                     dim3 grid_size,
                     dim3 block_size,
                     BodyPool * pool) {
```

```
        memset(ax, 0, sizeof(ax));
        memset(ay, 0, sizeof(ay));

        int totalThreadSize = grid_size.x * grid_size.y * grid_size.z * block_size.x *
block_size.y * block_size.z;

        // allocate device memory
        BodyPool * cudaPoolCopy;
        cudaMallocManaged((void**)&cudaPoolCopy, sizeof(*pool));
        cudaMemcpy(cudaPoolCopy, pool, sizeof(*pool), cudaMemcpyHostToDevice);

        // allocate memory for arrays
        double *xcopy, *ycopy, *vxcopy, *vycopy, *axcopy, *aycopy, *mcopy;
        cudaMallocManaged((void**) &(xcopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(ycopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(vxcopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(vycopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(axcopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(aycopy), sizeof(double)*pool->size());
        cudaMallocManaged((void**) &(mcopy), sizeof(double)*pool->size());

        // copy values to arrays
        cudaMemcpy(xcopy, pool->x, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(ycopy, pool->y, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(vxcopy, pool->vx, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(vycopy, pool->vy, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(axcopy, pool->ax, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(aycopy, pool->ay, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);
        cudaMemcpy(mcopy, pool->m, sizeof(double)*pool->size(), cudaMemcpyHostToDevice);

        cudaPoolCopy->x = xcopy;
        cudaPoolCopy->y = ycopy;
        cudaPoolCopy->vx = vxcopy;
        cudaPoolCopy->vy = vycopy;
        cudaPoolCopy->ax = axcopy;
        cudaPoolCopy->ay = aycopy;
        cudaPoolCopy->m = mcopy;

        cuda_update_for_tick<<<grid_size, block_size>>>(elapse, gravity, position_range,
radius, totalThreadSize, cudaPoolCopy);
        cudaDeviceSynchronize();

        for (int i = 0; i < pool->size(); ++i) {
            pool->x[i] = cudaPoolCopy->x[i];
            pool->y[i] = cudaPoolCopy->y[i];
            pool->vx[i] = cudaPoolCopy->vx[i];
            pool->vy[i] = cudaPoolCopy->vy[i];
            pool->ax[i] = cudaPoolCopy->ax[i];
            pool->ay[i] = cudaPoolCopy->ay[i];
```

```
        }

        cudaFree(cudaPoolCopy);
        cudaFree(xcopy);
        cudaFree(ycopy);
        cudaFree(vxcopy);
        cudaFree(vycopy);
        cudaFree(axcopy);
        cudaFree(aycopy);
        cudaFree(mcopy);

        // printf("\n\n");

    }
};

__global__
void cuda_update_for_tick(double elapse,
                          double gravity,
                          double position_range,
                          double radius,
                          int totalThreadSize,
                          BodyPool * pool) {

    int taskNum = getThreadId();

    // allocate memory for itself
    BodyPool * copyPool;
    cudaMalloc((void**)&copyPool, sizeof(*pool));
    copyPool->_size = pool->size();

    // allocate memory for arrays
    double *xcopy, *ycopy, *vxcopy, *vycopy, *axcopy, *aycopy, *mcopy;
    cudaMalloc((void**) &(xcopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(ycopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(vxcopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(vycopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(axcopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(aycopy), sizeof(double)*pool->size());
    cudaMalloc((void**) &(mcopy), sizeof(double)*pool->size());

    // copy values to arrays
    memcpy(xcopy, pool->x, sizeof(double)*pool->size());
    memcpy(ycopy, pool->y, sizeof(double)*pool->size());
    memcpy(vxcopy, pool->vx, sizeof(double)*pool->size());
    memcpy(vycopy, pool->vy, sizeof(double)*pool->size());
    memcpy(axcopy, pool->ax, sizeof(double)*pool->size());
    memcpy(aycopy, pool->ay, sizeof(double)*pool->size());
    memcpy(mcopy, pool->m, sizeof(double)*pool->size());
```

```
    copyPool->x = xcopy;
    copyPool->y = ycopy;
    copyPool->vx = vxcopy;
    copyPool->vy = vycopy;
    copyPool->ax = axcopy;
    copyPool->ay = aycopy;
    copyPool->m = mcopy;

    // make sure all threads finish copy
    __syncthreads();

    int startPosition = (copyPool->size()/totalThreadSize)*taskNum + min(static_cast<int>
(copyPool->size())%totalThreadSize, taskNum);
    int endPosition = (copyPool->size()/totalThreadSize)*(taskNum+1) +
min(static_cast<int> (copyPool->size())%totalThreadSize, taskNum+1)-1;

    // printf("%d %d %d\n", taskNum, startPosition, endPosition);

    // consider the situations of bodies between that outsides the range and inside the
range
    for (int i = 0; i < startPosition; ++i) {
        for (int j = startPosition; j <= endPosition; ++j) {
            copyPool->check_and_update(copyPool->get_body(i), copyPool->get_body(j),
radius, gravity);
        }
    }

    for (int i = endPosition+1; i < copyPool->size(); ++i) {
        for (int j = startPosition; j <= endPosition; ++j) {
            copyPool->check_and_update(copyPool->get_body(i), copyPool->get_body(j),
radius, gravity);
        }
    }

    // consider the situations of bodies inside the range
    for (int i = startPosition; i <= endPosition; ++i) {
        for (int j = i + 1; j <= endPosition; ++j) {
            copyPool->check_and_update(copyPool->get_body(i), copyPool->get_body(j),
radius, gravity);
        }
    }

    // only update the position of bodies inside the range
    for (int i = startPosition; i <= endPosition; ++i) {
        copyPool->get_body(i).update_for_tick(elapse, position_range, radius);
    }

    // update data back to host
```

```
    for (int i = startPosition; i <= endPosition; ++i) {
        pool->x[i] = copyPool->x[i];
        pool->y[i] = copyPool->y[i];
        pool->vx[i] = copyPool->vx[i];
        pool->vy[i] = copyPool->vy[i];
        pool->ax[i] = copyPool->ax[i];
        pool->ay[i] = copyPool->ay[i];
    }

    cudaFree(copyPool);
    cudaFree(xcopy);
    cudaFree(ycopy);
    cudaFree(vxcopy);
    cudaFree(vycopy);
    cudaFree(axcopy);
    cudaFree(aycopy);
    cudaFree(mcopy);

}
```

5) MPI with OpenMP

(main.cpp)

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <algorithm>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

// define struct to pass information
typedef struct Arguments_S {
        int current_bodies;
        float current_space;
        float max_mass;
        float elapse;
        float gravity;
        float radius;
} Arguments;

static MPI_Datatype Arguments_TYPE;

int main(int argc, char **argv) {
    int rank;
    int processSize;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    // use 4 threads for OpenMP as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }

    printf("OpenMP thread size %d\n", totalThreadSize);
    omp_set_num_threads(totalThreadSize);

    // initialize MPI struct
    const int nitems=6;
    int blocklengths[6] = {1,1,1,1,1,1};
    MPI_Datatype types[6] = {MPI_INT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT,
MPI_FLOAT};
    MPI_Aint     offsets[6];

    offsets[0] = offsetof(Arguments, current_bodies);
    offsets[1] = offsetof(Arguments, current_space);
    offsets[2] = offsetof(Arguments, max_mass);
    offsets[3] = offsetof(Arguments, elapse);
    offsets[4] = offsetof(Arguments, gravity);
    offsets[5] = offsetof(Arguments, radius);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &Arguments_TYPE);
    MPI_Type_commit(&Arguments_TYPE);

    UNUSED(argc, argv);
    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 200;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    static float current_space = space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;
    BodyPool pool(static_cast<size_t>(bodies), space, max_mass);
```

```
    if(rank==0){
        graphic::GraphicContext context{"Assignment 3"};
        context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 3", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
            ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
            ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
            ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
            ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
            ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100, "%f");
            ImGui::ColorEdit4("Color", &color.x);

            // boardcast updated plotting information to other ranks
            Arguments arguments;
            arguments.current_bodies = current_bodies;
            arguments.current_space = current_space;
            arguments.max_mass = max_mass;
            arguments.elapse = elapse;
            arguments.gravity = gravity;
            arguments.radius = radius;

            MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);

            if (current_space != space || current_bodies != bodies || current_max_mass !=
max_mass) {
                space = current_space;
                bodies = current_bodies;
                max_mass = current_max_mass;
                pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
            }

            // record calculation time
            auto begin = std::chrono::high_resolution_clock::now();

            // boardcast current information of all bodies
            MPI_Bcast(pool.x.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.y.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
            MPI_Bcast(pool.vx.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vy.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.ax.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.ay.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.m.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);

            {
                const ImVec2 p = ImGui::GetCursorScreenPos();

                // do calculations on the target bodies
                pool.update_for_tick(elapse, gravity, space, radius, totalThreadSize,
&pool);

                // gather calculation result from all ranks
                // calculate the job size of each rank
                std::vector<int> taskSizeList;
                for(int i=0;i<processSize;i++){
                    int startPosition = (bodies/processSize)*i +
std::min(bodies%processSize, i);
                    int endPosition = (bodies/processSize)*(i+1) +
std::min(bodies%processSize, i+1)-1;
                    taskSizeList.push_back(endPosition-startPosition+1);
                }

                // calculate the strips of received data packages
                std::vector<int> strips;
                strips.push_back(0);
                for(int i=1;i<processSize;i++){
                    strips.push_back(strips[i-1]+taskSizeList[i-1]);
                }

                // calcualte job size of itself
                int startPosition = (bodies/processSize)*rank +
std::min(bodies%processSize, rank);
                int endPosition = (bodies/processSize)*(rank+1) +
std::min(bodies%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // rank 0 gather calculated result
                MPI_Gatherv(pool.x.data()+startPosition, jobSize, MPI_DOUBLE,
pool.x.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.y.data()+startPosition, jobSize, MPI_DOUBLE,
pool.y.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.vx.data()+startPosition, jobSize, MPI_DOUBLE,
pool.vx.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.vy.data()+startPosition, jobSize, MPI_DOUBLE,
pool.vy.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.ax.data()+startPosition, jobSize, MPI_DOUBLE,
pool.ax.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
                MPI_Gatherv(pool.ay.data()+startPosition, jobSize, MPI_DOUBLE,
pool.ay.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gatherv(pool.m.data()+startPosition, jobSize, MPI_DOUBLE,
pool.m.data(), taskSizeList.data(), strips.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD);

                // record calculation time
                auto end = std::chrono::high_resolution_clock::now();

                auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
                std::cout << pool.size() << " bodies in last " << duration << "
nanoseconds" << std::endl;


                // plot calculation results
                for (size_t i = 0; i < pool.size(); ++i) {
                    auto body = pool.get_body(i);
                    auto x = p.x + static_cast<float>(body.get_x());
                    auto y = p.y + static_cast<float>(body.get_y());
                    draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
                }

                // printf("\n\n");

            }
            ImGui::End();
        });
    }
    else{
        while(true){

            // receive updated plotting parameters from rank 0
            Arguments arguments;
            MPI_Bcast(&arguments, 1, Arguments_TYPE, 0, MPI_COMM_WORLD);
            current_bodies = arguments.current_bodies;
            current_space = arguments.current_space;
            max_mass = arguments.max_mass;
            elapse = arguments.elapse;
            gravity = arguments.gravity;
            radius = arguments.radius;

            if (current_space != space || current_bodies != bodies || current_max_mass !=
max_mass) {
                space = current_space;
                bodies = current_bodies;
                max_mass = current_max_mass;
                pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
            }
```

```
            // receive current information of all bodies
            MPI_Bcast(pool.x.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.y.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vx.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.vy.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.ax.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.ay.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(pool.m.data(), bodies, MPI_DOUBLE, 0, MPI_COMM_WORLD);

            // do calculations on the target bodies
            pool.update_for_tick(elapse, gravity, space, radius, totalThreadSize, &pool);

            // calcualte job size of itself
            int startPosition = (bodies/processSize)*rank + std::min(bodies%processSize,
rank);
            int endPosition = (bodies/processSize)*(rank+1) + std::min(bodies%processSize,
rank+1)-1;
            int jobSize = endPosition-startPosition+1;

            // send calculation result
            MPI_Gatherv(pool.x.data()+startPosition, jobSize, MPI_DOUBLE, nullptr, nullptr,
nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.y.data()+startPosition, jobSize, MPI_DOUBLE, nullptr, nullptr,
nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.vx.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.vy.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.ax.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.ay.data()+startPosition, jobSize, MPI_DOUBLE, nullptr,
nullptr, nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gatherv(pool.m.data()+startPosition, jobSize, MPI_DOUBLE, nullptr, nullptr,
nullptr, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
    }

    // if ploting windows closed, rank 0 would then stop all other processes
    if(rank==0){
        std::cout << "aborting all processes" << std::endl;
        MPI_Abort(MPI_COMM_WORLD, 0);
    }

    MPI_Type_free(&Arguments_TYPE);
    MPI_Finalize();

}
```

(body.hpp)

```cpp
//
// Created by schrodinger on 11/2/21.
//
#pragma once

#include <random>
#include <utility>
#include <mpi.h>
#include <algorithm>
#include <omp.h>

class BodyPool {
    // if after the collision, we do not separate the bodies a little bit, it may
    // results in strange outcomes like infinite acceleration.
    // hence, we will need to set up a ratio for separation.
    static constexpr double COLLISION_RATIO = 0.01;

public:
    // provides in this way so that
    // it is easier for you to send a the vector with MPI
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> vx;
    std::vector<double> vy;
    std::vector<double> ax;
    std::vector<double> ay;
    std::vector<double> m;
    // so the movements of bodies are calculated discretely.

    class Body {
        size_t index;
        BodyPool &pool;

        friend class BodyPool;

        Body(size_t index, BodyPool &pool) : index(index), pool(pool) {}

    public:
        double &get_x() {
            return pool.x[index];
        }

        double &get_y() {
            return pool.y[index];
        }

        double &get_vx() {
```

```
        return pool.vx[index];
    }

    double &get_vy() {
        return pool.vy[index];
    }

    double &get_ax() {
        return pool.ax[index];
    }

    double &get_ay() {
        return pool.ay[index];
    }

    double &get_m() {
        return pool.m[index];
    }

    double distance_square(Body &that) {
        auto delta_x = get_x() - that.get_x();
        auto delta_y = get_y() - that.get_y();
        return delta_x * delta_x + delta_y * delta_y;
    }

    double distance(Body &that) {
        return std::sqrt(distance_square(that));
    }

    double delta_x(Body &that) {
        return get_x() - that.get_x();
    }

    double delta_y(Body &that) {
        return get_y() - that.get_y();
    }

    bool collide(Body &that, double radius) {
        return distance_square(that) <= radius * radius;
    }

    // collision with wall
    void handle_wall_collision(double position_range, double radius) {
        bool flag = false;
        if (get_x() <= radius) {
            flag = true;
            get_x() = radius + radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        } else if (get_x() >= position_range - radius) {
```

```
            flag = true;
            get_x() = position_range - radius - radius * COLLISION_RATIO;
            get_vx() = -get_vx();
        }

        if (get_y() <= radius) {
            flag = true;
            get_y() = radius + radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        } else if (get_y() >= position_range - radius) {
            flag = true;
            get_y() = position_range - radius - radius * COLLISION_RATIO;
            get_vy() = -get_vy();
        }
        if (flag) {
            get_ax() = 0;
            get_ay() = 0;
        }
    }

    void update_for_tick(
            double elapse,
            double position_range,
            double radius) {
        get_vx() += get_ax() * elapse;
        get_vy() += get_ay() * elapse;
        handle_wall_collision(position_range, radius);
        get_x() += get_vx() * elapse;
        get_y() += get_vy() * elapse;
        handle_wall_collision(position_range, radius);
    }

};

BodyPool(size_t size, double position_range, double mass_range) :
        x(size), y(size), vx(size), vy(size), ax(size), ay(size), m(size) {
    std::random_device device;
    std::default_random_engine engine{device()};
    std::uniform_real_distribution<double> position_dist{0, position_range};
    std::uniform_real_distribution<double> mass_dist{0, mass_range};
    for (auto &i : x) {
        i = position_dist(engine);
    }
    for (auto &i : y) {
        i = position_dist(engine);
    }
    for (auto &i : vx) {
        i = 0;
    }
```

```
        for (auto &i : vy) {
            i = 0;
        }
        for (auto &i : m) {
            i = mass_dist(engine);
        }
    }

    Body get_body(size_t index) {
        return {index, *this};
    }

    void clear_acceleration() {
        ax.assign(m.size(), 0.0);
        ay.assign(m.size(), 0.0);
    }

    size_t size() {
        return m.size();
    }

    static void check_and_update(Body i, Body j, double radius, double gravity) {
        auto delta_x = i.delta_x(j);
        auto delta_y = i.delta_y(j);
        auto distance_square = i.distance_square(j);
        auto ratio = 1 + COLLISION_RATIO;
        if (distance_square < radius * radius) {
            distance_square = radius * radius;
        }
        auto distance = i.distance(j);
        if (distance < radius) {
            distance = radius;
        }
        if (i.collide(j, radius)) {
            auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                            + delta_y * (i.get_vy() - j.get_vy());
            auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
            i.get_vx() -= scalar * delta_x * j.get_m();
            i.get_vy() -= scalar * delta_y * j.get_m();
            j.get_vx() += scalar * delta_x * i.get_m();
            j.get_vy() += scalar * delta_y * i.get_m();
            // now relax the distance a bit: after the collision, there must be
            // at least (ratio * radius) between them
            i.get_x() += delta_x / distance * ratio * radius / 2.0;
            i.get_y() += delta_y / distance * ratio * radius / 2.0;
            j.get_x() -= delta_x / distance * ratio * radius / 2.0;
            j.get_y() -= delta_y / distance * ratio * radius / 2.0;
        } else {
            // update acceleration only when no collision
```

```
            auto scalar = gravity / distance_square / distance;
            i.get_ax() -= scalar * delta_x * j.get_m();
            i.get_ay() -= scalar * delta_y * j.get_m();
            j.get_ax() += scalar * delta_x * i.get_m();
            j.get_ay() += scalar * delta_y * i.get_m();
        }
    }

    void update_for_tick(double elapse,
                         double gravity,
                         double position_range,
                         double radius,
                         int totalThreadSize,
                         BodyPool * pool) {
        ax.assign(size(), 0);
        ay.assign(size(), 0);

        int rank;
        int processSize;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &processSize);

        int processStartPosition = (size()/processSize)*rank + std::min(static_cast<int>
(size())%processSize, rank);
        int processEndPosition = (size()/processSize)*(rank+1) + std::min(static_cast<int>
(size())%processSize, rank+1)-1;
        int processTaskSize = processEndPosition - processStartPosition + 1;

        // printf("rank %d %d %d %d\n", rank, processStartPosition, processEndPosition,
processTaskSize);

        #pragma omp parallel
        {
            BodyPool copyPool = *pool;

            // make sure all threads finish copy
        #pragma omp barrier

            int taskNum = omp_get_thread_num();

            int startPosition = (processTaskSize/totalThreadSize)*taskNum +
std::min(processTaskSize%totalThreadSize, taskNum) + processStartPosition;
            int endPosition = (processTaskSize/totalThreadSize)*(taskNum+1) +
std::min(processTaskSize%totalThreadSize, taskNum+1)-1 + processStartPosition;

            // printf("thread %d %d %d (rank %d)\n", taskNum, startPosition, endPosition,
rank);
```

```
            // consider the situations of bodies between that outsides the range and inside
the range
            for (size_t i = 0; i < static_cast<size_t>(startPosition); ++i) {
                for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j)
{
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
                }
            }
            for (size_t i = endPosition+1; i < size(); ++i) {
                for (size_t j = startPosition; j <= static_cast<size_t>(endPosition); ++j)
{
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
                }
            }

            // consider the situations of bodies inside the range
            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                for (size_t j = i + 1; j <= static_cast<size_t>(endPosition); ++j) {
                    check_and_update(copyPool.get_body(i), copyPool.get_body(j), radius,
gravity);
                }
            }

            // only update the position of bodies inside the range
            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                copyPool.get_body(i).update_for_tick(elapse, position_range, radius);
            }

            for (size_t i = startPosition; i <= static_cast<size_t>(endPosition); ++i) {
                // printf("%d x[%zu] %f %f\n", taskNum, i, copyPool.x[i], pool->x[i]);
                pool->x[i] = copyPool.x[i];
                pool->y[i] = copyPool.y[i];
                pool->vx[i] = copyPool.vx[i];
                pool->vy[i] = copyPool.vy[i];
                pool->ax[i] = copyPool.ax[i];
                pool->ay[i] = copyPool.ay[i];
            }

        }
    }
};
```