

# CSC4005 FA22 HW03

Haoran Sun (haoransun@link.cuhk.edu.cn)

## 1 Introduction

A typical  $n$ -body simulation problem would usually involve a calculation of  $n \times n$  interactions. Therefore, the complexities would be  $O(n^2)$ . For example, a gravitational  $n$ -body simulation is the computer simulation of particles under the influence of gravity. Also, (bio) molecular dynamics simulation, which simulates the dynamics of chemical molecules under different conditions is also a typical  $n$ -body problem. Usually, the computation of the interactions could be split into mutually independent part—which indicates that  $n$ -body problem can be massively parallelized.

In this project, a 2-D gravitational  $n$ -body simulation is implemented. Despite a sequential version, the program is also accelerated by common parallelization libraries: MPICH, OpenMP, Pthread, and CUDA. The performance of each method is evaluated.

## 2 Method

### 2.1 System setup

The systems contains  $n$  particles with random generated position  $\mathbf{x}_i$  on a 2-D plane and their mass  $m_i$  ( $\mathbf{x}_i \in \mathbb{R}^2, i = 1, \dots, n$ ). The force that the particle  $j$  exerted on the particle  $i$  is

$$\mathbf{F}_{ij} = (\mathbf{x}_j - \mathbf{x}_i) \frac{Gm_i m_j}{r_{ij}^3}$$

Hence the acceleration of the  $i$ th particle is

$$\mathbf{a}_i = \sum_j \frac{\mathbf{F}_{ij}}{m_i} = \sum_j (\mathbf{x}_j - \mathbf{x}_i) \frac{Gm_j}{r_{ij}^3}$$

To update the system, the Verlet algorithm is implemented to calculate the position of particles during the time evolution.

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \mathbf{a}(t)\Delta t^2$$

The reason to choose the Verlet algorithm rather than the Euler's method which mentioned in the homework instructions is that the Verlet algorithm follows the conservation law of energy but the Euler's method doesn't.

### 2.2 Program design and implementation

The programs are written in the C++ programming language. MPICH, Pthread, OpenMP, and CUDA libraries were used for parallelization. Besides, OpenGL is used for visualization purposes. Also, to improve the performance, the MPI version is further accelerated using OpenMP.

Despite MPI version written separately in `src/main.mpi.cpp`, the main program of other version are all wrapped in `src/main.cpp`. Particularly, CUDA functions are compiled in a separated library.

## 2.3 Usage

*Remark.* For convenience, one can directly build the program by `scripts/build.sh` to compile all targets.

To simplify the compiling process, the CMake build system is used to compile programs and link libraries. One can execute the following lines to build executables.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release -DGUI=ON
cmake --build build
```

To disable the GUI feature, one can set `-DGUI=OFF` in the first line. The compiled programs and libraries are shown in the `build/bin` and `build/lib`. One can directly execute `build/bin/main*.gui` for a visualized demonstration.

```
./build/bin/main.seq.gui
./build/bin/main.omp.gui
./build/bin/main.pth.gui
./build/bin/main.mpi.gui
./build/bin/main.cu.gui
```

Figure 1: Sample GUI window

One can customize the running parameters such as the number of particles  $n$  and simulation steps according to the following lines. `-nt` is for number of threads, `--Tx` and `--Ty` is to set CUDA 1-D grid size and block size.

```
./build/bin/main.seq -n 100 --nsteps 10000 --record 1
./build/bin/main.omp -nt 10 -n 100 --nsteps 10000 --record 1
./build/bin/main.omp -nt 10 -n 100 --nsteps 10000 --record 1
./build/bin/main.cu --Tx 16 --Ty -n 100 --nsteps 10000 --record 1
mpirun -np 10 ./build/bin/main.mpi -n 100 --nsteps 10000 --record 1
```

*Remark.* To execute MPI + OpenMP hybrid program, one just append `-nt [n]` parameters when executing the MPI program. For example, the following line initializes a program with 10 MPI process, and each process has 2 OpenMP threads.

```
mpirun -np 10 ./build/bin/main.mpi -nt 2
```

## 2.4 Performance evaluation

The program was executed under different configurations to evaluate performance. With 20 different CPU core numbers (from 1 to 20 with increment 1,  $p = 1, 2, \dots, 20$ ) and 20 different  $n$  (from 50 to 1000 with increment 50), 400 cases in total were sampled for sequential, MPI, OpenMP, and Pthread programs. Test for CUDA program is implemented separately since GPU is much faster than all CPU programs and only large-scale performance will be discussed on CUDA program. Recorded runtime is analyzed through the Numpy package in Python. Figures were plotted through the Matplotlib and the Seaborn packages in Python. Analysis codes were written in `analysis/main.ipynb`.

### 3 Result and discussion

*Remark.* Again, since GPU is much faster than CPU, I would discuss their performances separately. Also, the discussion will focus on large-scale cases.

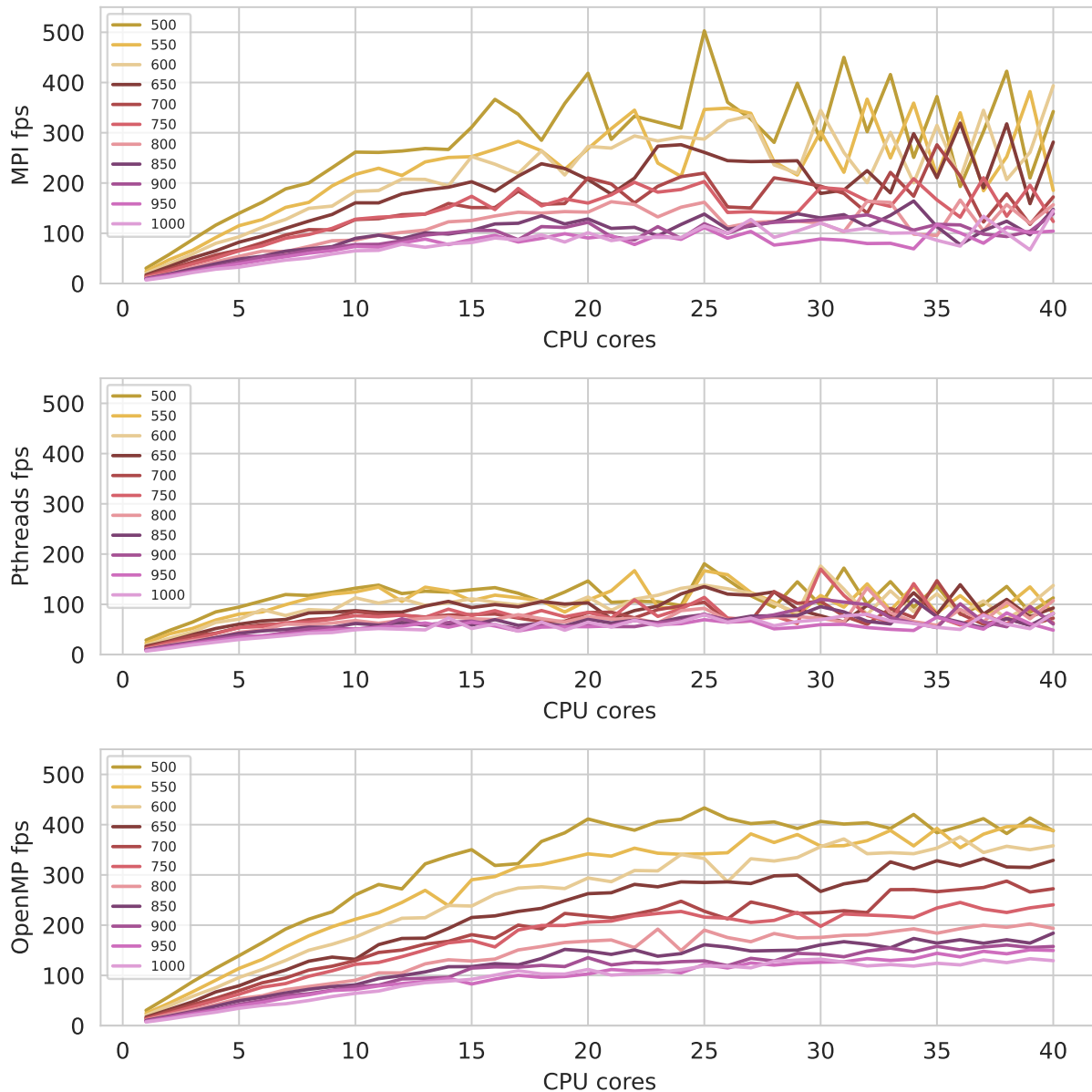
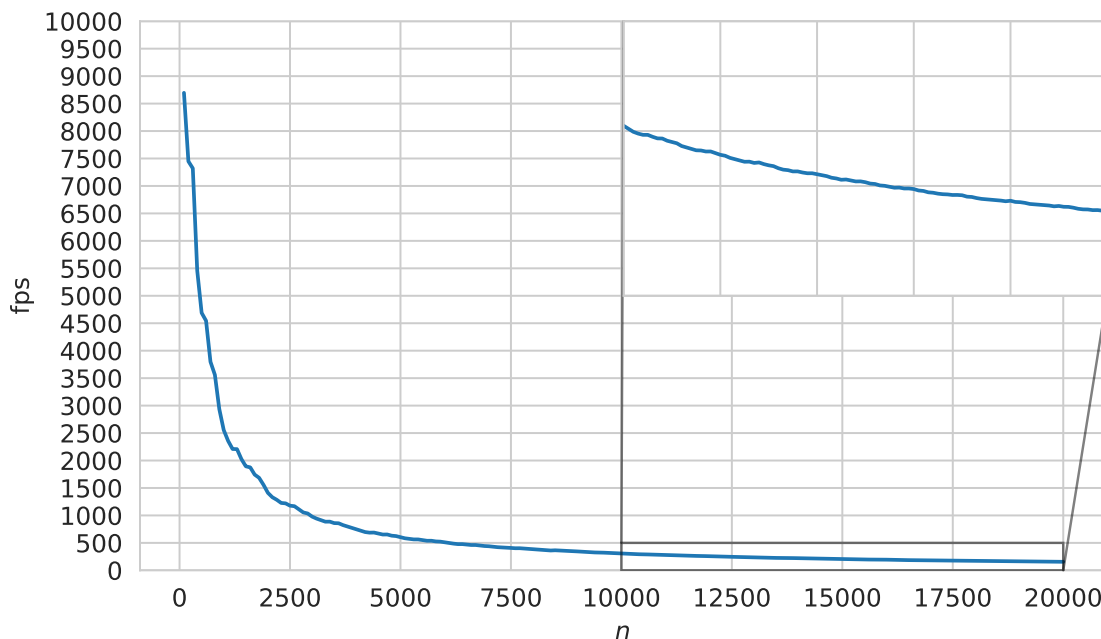


Figure 2: fps vs the number of threads/processes plot.

#### 3.1 CPU parallelization

From Figure 2, we can know that when  $n$  ranging from 500 to 1000, MPI and OpenMP programs have similar performance when the number of processes/threads is under 20: fps steadily increases

Figure 3: CUDA fps vs  $n$  plot.

with threads/processes number while decreases with  $n$ . The MPI program becomes quite unstable when the number of cores exceeds 20: the reason might be the unstable communication traffic and CPU resources. Meanwhile, the Pthread program has low and relatively constant performance. That may result from the Pthread function `compute_pth` in `src/utlis.h`. In each iteration (each frame), the program will initialize  $nt$  threads, perform the computation parallelly and then merge these threads. Different from OpenMP which is fully optimized, the initialization and joining of threads in each iteration could be much more time-costly. To fix this issue, one may initialize threads at the start of the program, and join all threads after finishing all calculations.

The heatmap which indicate the rate of acceleration plotted in the Figure A.1 provides some direct visualization of the performances of parallel variants.

### 3.2 GPU parallelization

GPU parallelization is much more massive than CPU parallelization. This allows one to implemented  $n > 10^4$  with high fps, as Figure 3 shows. Notably, the gpu shared memory is used to accelerate the read operations. (please refer to `__shared` type and `__syncthreads` function in `cuda.lib.cu`). According to NVIDIA, the memory access on shared memory is approximately 100x faster than the global (`__device__`) memory access.

## 4 Conclusion

In conclusion, four parallel computing schemes for  $n$ -body simulation are implemented and their performances are evaluated. For large, ignoring the precision, one may use GPU to accelerate the

calculation.

## A Supplementary figures

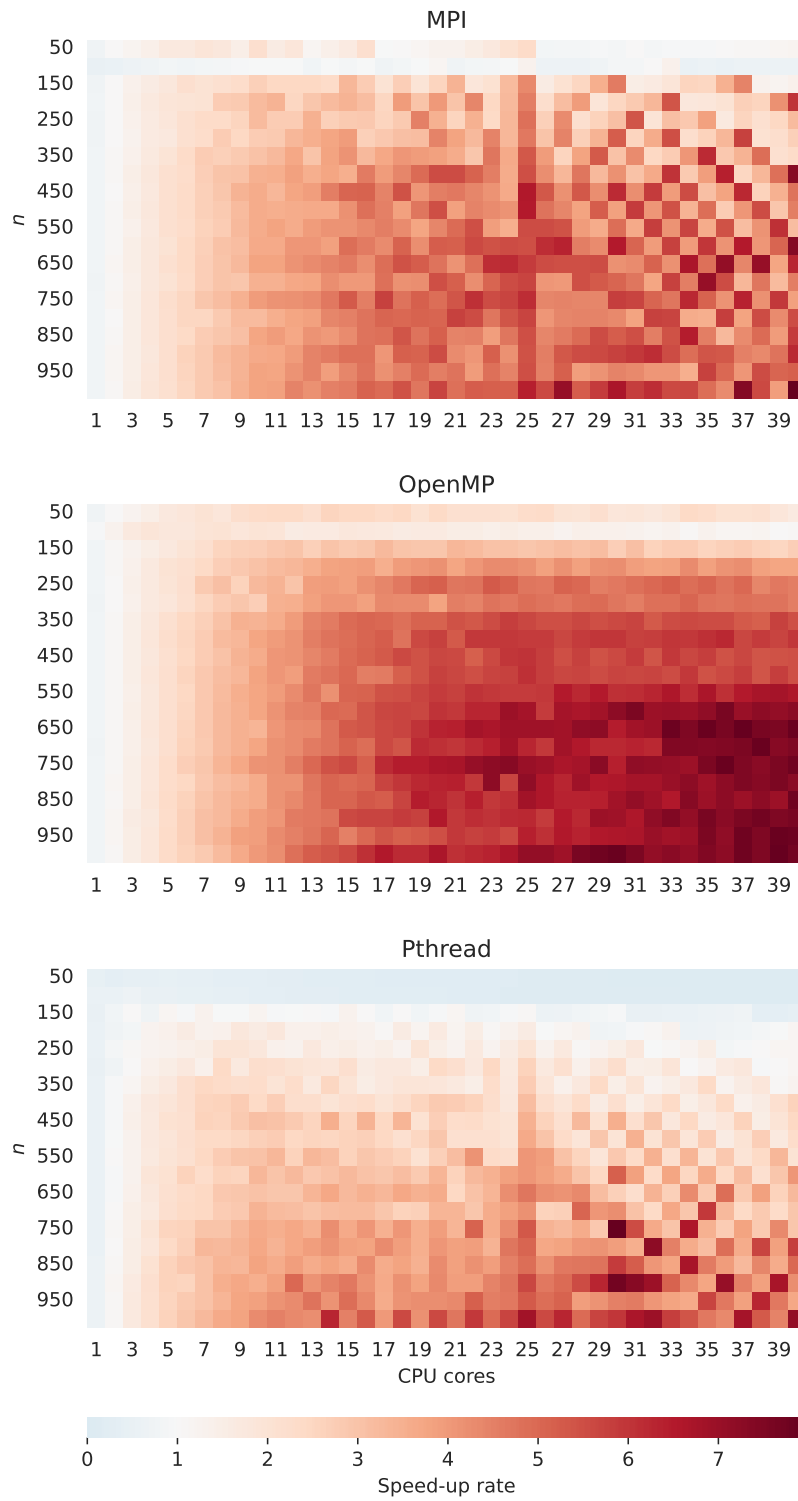


Figure A.1: CUDA fps vs  $n$  plot.

## B Source code

CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.20)
2 project(hw03 LANGUAGES CXX CUDA)
3
4 # set output path
5 set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
6 set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
7 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
8
9 # set include libraires
10 include_directories(src)
11
12 set(CMAKE_CXX_STANDARD 11)
13
14 # add src folder
15 add_subdirectory(src)

```

src/CMakeLists.txt

```

1 find_package(MPI REQUIRED)
2 find_package(CUDA REQUIRED)
3 find_package(Threads REQUIRED)
4 find_package(OpenMP REQUIRED)
5
6 # options
7 # gui option
8 option(GUI "OPENGL Rendering" OFF)
9
10 # omp flags
11 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
12
13 # libraries
14 add_library(cudalib cudalib.cu)
15 set(THREADS_PREFER_PTHREAD_FLAG ON)
16 include_directories(
17     ${MPI_INCLUDE_PATH}
18     ${CUDA_INCLUDE_DIRS}
19 )
20 link_libraries(
21     ${MPI_LIBRARIES}
22     ${CUDA_LIBRARIES}
23     cudalib
24 )
25
26
27 # targets & libs
28 add_executable(main.seq main.cpp)
29 add_executable(main.omp main.cpp)
30 add_executable(main.pth main.cpp)
31 add_executable(main.cu main.cpp)
32 add_executable(main.mpi main.mpi.cpp)
33 target_compile_definitions(main.omp PUBLIC OMP)
34 target_compile_definitions(main.pth PUBLIC PTH)
35 target_compile_definitions(main.cu PUBLIC CUDA)
36
37 # opengl & glut
38 if(GUI)
39     find_package(OpenGL REQUIRED)
40     find_package(GLUT REQUIRED)
41     include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})
42     link_libraries(${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
43     add_executable(main.omp.gui main.cpp)
44     add_executable(main.cu.gui main.cpp)

```

```

45     add_executable(main.seq.gui main.cpp)
46     add_executable(main.pth.gui main.cpp)
47     add_executable(main.mpi.gui main.mpi.cpp)
48     target_compile_definitions(main.seq.gui PUBLIC GUI)
49     target_compile_definitions(main.mpi.gui PUBLIC GUI)
50     target_compile_definitions(main.omp.gui PUBLIC GUI OMP)
51     target_compile_definitions(main.cu.gui PUBLIC GUI CUDA)
52     target_compile_definitions(main.pth.gui PUBLIC GUI PTH)
53 endif()

```

src/main.mpi.cpp

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  #include <memory.h>
5  #include <chrono>
6  #include "const.h"
7  #include "utils.h"
8  #ifdef GUI
9  #include "gui.h"
10 #endif
11
12 void compute(){
13     // main program
14     char type[] = "mpi";
15     int start_idx, end_idx;
16     int jobsize = N / size;
17     auto t0 = std::chrono::high_resolution_clock::now();
18     auto t1 = std::chrono::high_resolution_clock::now();
19     auto t2 = std::chrono::high_resolution_clock::now();
20     double t;
21     partition(N, size, rank, &start_idx, &end_idx);
22     if (rank == 0) printf("Start MPI version.\n");
23     for (int s = 0; s < nsteps; s++){
24         // transfer data
25         MPI_Bcast(xarr, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
26         MPI_Barrier(MPI_COMM_WORLD);
27
28         // calculate dx
29         vec_assign_const(dxarr, 0, N*dim);
30         verlet_at2_part_omp(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx,
31                             end_idx);
32         // verlet_at2_part(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx,
33                             end_idx);
34         vec_add_part(dxarr, dxarr, xarr, 1.0, 1.0, N*dim, start_idx*dim, end_idx*dim);
35         vec_add_part(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim, start_idx*dim, end_idx*dim);
36         float *tmp = xarr;
37         xarr = xarr0;
38         xarr0 = tmp;
39         MPI_Barrier(MPI_COMM_WORLD);
40         verlet_add_part_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax,
41                             start_idx, end_idx);
42         // verlet_add_part(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax, start_idx,
43                             end_idx);
44
45         // transfer data
46         if (rank==0) MPI_Gather(MPI_IN_PLACE, jobsize*dim, MPI_FLOAT, xarr+start_idx*dim,
47                                 jobsize*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
48         else MPI_Gather(xarr+start_idx*dim, jobsize*dim, MPI_FLOAT, xarr, jobsize*dim,
49                         MPI_FLOAT, 0, MPI_COMM_WORLD);
50         MPI_Barrier(MPI_COMM_WORLD);
51         // solve tail case
52         if (N%jobsize!=0) {
53             if (rank==0){
54                 MPI_Recv(xarr+(N/size*size)*dim, (N%jobsize)*dim, MPI_FLOAT, size-1, 0,
55                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```



```

49     }
50     else if (rank+1==size){
51         MPI_Send(xarr+(N/size*size)*dim, (N%jobsize)*dim, MPI_FLOAT, 0, 0,
52                 MPI_COMM_WORLD);
53     }
54     MPI_Barrier(MPI_COMM_WORLD);
55
56     // opengl
57     if (rank==0){
58         #ifdef GUI
59             // calculating fps
60             int step = 200;
61             if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now
62                 ();
63             else if (s%(step*2)==0 && s!=0) {
64                 t2 = std::chrono::high_resolution_clock::now();
65                 t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).
66                     count();
67                 printf("fps: %f frame/s\n", step/t);
68             }
69             glClear(GL_COLOR_BUFFER_BIT);
70             glColor3f(1.0f, 0.0f, 0.0f);
71             glPointSize(2.0f);
72
73             // gl points
74             glBegin(GL_POINTS);
75             float xi;
76             float yi;
77             float xmin, xmax, ymin, ymax;
78             for (int i = 0; i < N; i++){
79                 xi = xarr[i*dim+0];
80                 yi = xarr[i*dim+1];
81                 glVertex2f(xi, yi);
82             }
83             glEnd();
84
85             glFlush();
86             glutSwapBuffers();
87             #endif
88         }
89     }
90
91     // record data
92     if (rank==0 && record==1){
93         t2 = std::chrono::high_resolution_clock::now();
94         t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
95         double fps = nsteps / t;
96         runtime_record(type, N, size, fps);
97     }
98 }
99
100 int main(int argc, char* argv[]){
101     // mpi initializatio
102     MPI_Init(NULL, NULL);
103     // fetch size and rank
104     MPI_Comm_size(MPI_COMM_WORLD, &size);
105     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
106
107     // parse arguments
108     char buff[200];
109     for (int i = 0; i < argc; i++){
110         strcpy(buff, argv[i]);
111         if (strcmp(buff, "-n")==0){
112             std::string num(argv[i+1]);
113             N = std::stoi(num);
114         }
115     }

```

```

113     if (strcmp(buff, "--nt")==0){
114         std::string num(argv[i+1]);
115         nt = std::stoi(num);
116     }
117     if (strcmp(buff, "--xmin")==0){
118         std::string num(argv[i+1]);
119         xmin = std::stof(num);
120     }
121     if (strcmp(buff, "--xmax")==0){
122         std::string num(argv[i+1]);
123         xmax = std::stof(num);
124     }
125     if (strcmp(buff, "--ymin")==0){
126         std::string num(argv[i+1]);
127         ymin = std::stof(num);
128     }
129     if (strcmp(buff, "--ymax")==0){
130         std::string num(argv[i+1]);
131         ymax = std::stof(num);
132     }
133     if (strcmp(buff, "--nsteps")==0){
134         std::string num(argv[i+1]);
135         nsteps = std::stof(num);
136     }
137     if (strcmp(buff, "--record")==0){
138         std::string num(argv[i+1]);
139         record = std::stoi(num);
140     }
141 }
142
143 // print info
144 if (rank == 0) print_info(N, nsteps);
145
146 // initialization
147 // array allocation
148 marr = (float *)malloc(sizeof(float) * N);
149 xarr = (float *)malloc(sizeof(float) * N * dim);
150 xarr0 = (float *)malloc(sizeof(float) * N * dim);
151 dxarr = (float *)malloc(sizeof(float) * N * dim);
152 // random generate initial condition
153 if (rank == 0){
154     random_generate(xarr, marr, N, dim);
155     // initialize xarr0
156     vec_add(xarr0, xarr0, xarr, 0, 1, N*dim);
157 }
158 // transfer data
159 MPI_Bcast(marr, N, MPI_FLOAT, 0, MPI_COMM_WORLD);
160 MPI_Bcast(xarr, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
161 MPI_Bcast(xarr0, N*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
162
163 // omp options
164 omp_set_dynamic(0);
165 omp_set_num_threads(nt);
166
167 // main computing program
168 if (rank==0){
169     #ifdef GUI
170     glutInit(&argc, argv);
171     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
172     glutInitWindowPosition(0, 0);
173     glutInitWindowSize(500, 500);
174     glutCreateWindow("N Body Simulation Sequential Implementation");
175     glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
176     glutDisplayFunc(&compute);
177     glutKeyboardFunc(&guiExit);
178     gluOrtho2D(xmin, xmax, ymin, ymax);
179     glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);

```

```

180     glutMainLoop();
181     #else
182     compute();
183     #endif
184 }
185 else {
186     compute();
187 }
188
189 // mpi finalization
190 MPI_Finalize();
191 }

```

## src/utils.h

```

1  #pragma once
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <iostream>
5  #include <math.h>
6  #include <mpi.h>
7  #include <omp.h>
8  #include <pthread.h>
9  #include <sys/stat.h>
10 #include <sys/types.h>
11
12 void print_info(int N, int nsteps){
13     printf("Name: Haoran Sun\n");
14     printf("ID: 119010271\n");
15     printf("HW: N-Body Simulation\n");
16     printf("Set N to %d, nsteps to %d\n", N, nsteps);
17 }
18
19 void partition(int nsteps, int size, int idx, int *start_ptr, int *end_ptr){
20     *start_ptr = nsteps / size * idx;
21     *end_ptr = nsteps / size * (idx+1);
22     if (idx+1==size) *end_ptr = nsteps;
23 }
24
25 void map_idx_to_pair(int N, int idx, int *i_ptr, int *j_ptr){
26     int work = N*(N-1) / 2;
27     int tmp = (-1 + sqrt(8*idx+9)) / 2;
28     int idx_ = tmp * (tmp+1) / 2 - 1;
29     if (idx_ < idx) tmp += 1;
30     idx_ = tmp * (tmp+1) / 2 - 1;
31     *i_ptr = tmp;
32     *j_ptr = tmp - 1 + idx - idx_;
33     // printf("mmm %d %d\n", *i_ptr, *j_ptr);
34 }
35
36 float norm(float *x, int dim){
37     float r = 0;
38     for (int i = 0; i < dim; i++){
39         r += pow(x[i], 2);
40     }
41     r = sqrt(r);
42     return r;
43 }
44
45 void get_xij(int i, int j, int dim, float *xarr, float *xij, int N){
46     for (int k = 0; k < dim; k++){
47         xij[k] = xarr[j*dim+k] - xarr[i*dim+k];
48     }
49 }
50
51 void print_arr(float *arr, int n){
52     for (int i = 0; i < n; i++){

```



```

53     printf("%10.2f  ", arr[i]);
54 }
55 printf("\n");
56 }
57
58 void vec_add(float *a, float *b, float *c,
59             float fac1, float fac2, int dim){
60     for (int i = 0; i < dim; i++){
61         a[i] = fac1*b[i] + fac2*c[i];
62     }
63 }
64
65 void vec_add_omp(float *a, float *b, float *c,
66                 float fac1, float fac2, int dim){
67     #pragma omp parallel for
68     for (int i = 0; i < dim; i++){
69         a[i] = fac1*b[i] + fac2*c[i];
70     }
71 }
72
73 void vec_add_part(float *a, float *b, float *c,
74                  float fac1, float fac2, int dim,
75                  int start_idx, int end_idx){
76     for (int i = start_idx; i < end_idx; i++){
77         a[i] = fac1*b[i] + fac2*c[i];
78     }
79 }
80
81 void verlet_at2(int dim, float *marr, float *xarr, float *xarr0,
82                float *dxarr, float dt, float G, int N, float cut){
83     for (int idx = 0; idx < N*(N-1)/2; idx++) {
84         int i, j;
85         map_idx_to_pair(N, idx, &i, &j);
86         // printf("%d %d\n", i, j);
87         float xij[dim];
88         float tmp[dim];
89         float mi = marr[i];
90         float mj = marr[j];
91         // get xij
92         get_xij(i, j, dim, xarr, xij, N);
93         // compute rij
94         float rij = norm(xij, dim);
95         float fac = 1.0;
96         if (rij < cut) {
97             rij = cut;
98         }
99         // compute intermediate variable
100        for (int k = 0; k < dim; k++){
101            tmp[k] = xij[k]*G/pow(rij, 3);
102        }
103        // add to dx
104        vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, mj*dt*dt, dim);
105        vec_add(dxarr+j*dim, dxarr+j*dim, tmp, 1.0, -mi*dt*dt, dim);
106    }
107 }
108
109
110 void verlet_at2_omp(int dim, float *marr, float *xarr, float *xarr0,
111                    float *dxarr, float dt, float G, int N, float cut){
112     #pragma omp parallel for
113     for (int i = 0; i < N; i++){
114         float tmp[dim];
115         for (int j = 0; j < dim; j++) tmp[j] = 0;
116         for (int j = 0; j < N; j++){
117             if (j!=i){
118                 float xij[dim];
119                 float mi = marr[i];

```

```

120         float mj = marr[j];
121         // get xij
122         get_xij(i, j, dim, xarr, xij, N);
123         // compute rij
124         float rij = norm(xij, dim);
125         float fac = 1.0;
126         if (rij < cut) {
127             rij = cut;
128         }
129         // compute intermediate variable
130         for (int k = 0; k < dim; k++){
131             tmp[k] += xij[k]*G/pow(rij, 3) *mj*dt*dt;
132         }
133     }
134 }
135 vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
136 }
137 }
138
139 void verlet_at2_part(int dim, float *marr, float *xarr, float *xarr0,
140 float *dxarr, float dt, float G, int N, float cut,
141 int start_idx, int end_idx){
142     for (int i = start_idx; i < end_idx; i++){
143         float tmp[dim];
144         for (int j = 0; j < dim; j++) tmp[j] = 0;
145         for (int j = 0; j < N; j++){
146             if (j!=i){
147                 float xij[dim];
148                 float mi = marr[i];
149                 float mj = marr[j];
150                 // get xij
151                 get_xij(i, j, dim, xarr, xij, N);
152                 // compute rij
153                 float rij = norm(xij, dim);
154                 float fac = 1.0;
155                 if (rij < cut) {
156                     rij = cut;
157                 }
158                 // compute intermediate variable
159                 for (int k = 0; k < dim; k++){
160                     tmp[k] += xij[k]*G/pow(rij, 3)*mj*dt*dt;
161                 }
162             }
163         }
164         vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
165     }
166 }
167
168 void verlet_at2_part_omp(int dim, float *marr, float *xarr, float *xarr0,
169 float *dxarr, float dt, float G, int N, float cut,
170 int start_idx, int end_idx){
171     #pragma omp parallel
172     {
173         int omp_start_idx, omp_end_idx;
174         partition(end_idx-start_idx, omp_get_num_threads(), omp_get_thread_num(),
175         &omp_start_idx, &omp_end_idx);
176         for (int i = start_idx+omp_start_idx; i < start_idx+omp_end_idx; i++){
177             float tmp[dim];
178             for (int j = 0; j < dim; j++) tmp[j] = 0;
179             for (int j = 0; j < N; j++){
180                 if (j!=i){
181                     float xij[dim];
182                     float mi = marr[i];
183                     float mj = marr[j];
184                     // get xij
185                     get_xij(i, j, dim, xarr, xij, N);
186                     // compute rij

```

```

187         float rij = norm(xij, dim);
188         float fac = 1.0;
189         if (rij < cut) {
190             rij = cut;
191         }
192         // compute intermediate variable
193         for (int k = 0; k < dim; k++){
194             tmp[k] += xij[k]*G/pow(rij, 3)*mj*dt*dt;
195         }
196     }
197 }
198 vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
199 }
200 }
201 }
202
203 void verlet_add(float *a, float *b, float *c, int N, int dim,
204 int xmin, int xmax, int ymin, int ymax){
205     for (int i = 0; i < N; i++){
206         float x = b[i*dim+0] + c[i*dim+0];
207         float y = b[i*dim+1] + c[i*dim+1];
208         if (x < xmin) x += 2 * (xmin - x);
209         else if (x > xmax) x += 2 * (xmax - x);
210         if (y < ymin) y += 2 * (ymin - y);
211         else if (y > ymax) y += 2 * (ymax - y);
212         a[i*dim+0] = x;
213         a[i*dim+1] = y;
214     }
215 }
216
217 void verlet_add_omp(float *a, float *b, float *c, int N, int dim,
218 int xmin, int xmax, int ymin, int ymax){
219     #pragma omp parallel for
220     for (int i = 0; i < N; i++){
221         float x = b[i*dim+0] + c[i*dim+0];
222         float y = b[i*dim+1] + c[i*dim+1];
223         if (x < xmin) x += 2 * (xmin - x);
224         else if (x > xmax) x += 2 * (xmax - x);
225         if (y < ymin) y += 2 * (ymin - y);
226         else if (y > ymax) y += 2 * (ymax - y);
227         a[i*dim+0] = x;
228         a[i*dim+1] = y;
229     }
230 }
231
232 void verlet_add_part(float *a, float *b, float *c, int N, int dim,
233 int xmin, int xmax, int ymin, int ymax,
234 int start_idx, int end_idx){
235     for (int i = start_idx; i < end_idx; i++){
236         float x = b[i*dim+0] + c[i*dim+0];
237         float y = b[i*dim+1] + c[i*dim+1];
238         if (x < xmin) x += 2 * (xmin - x);
239         else if (x > xmax) x += 2 * (xmax - x);
240         if (y < ymin) y += 2 * (ymin - y);
241         else if (y > ymax) y += 2 * (ymax - y);
242         a[i*dim+0] = x;
243         a[i*dim+1] = y;
244     }
245 }
246
247 void verlet_add_part_omp(float *a, float *b, float *c, int N, int dim,
248 int xmin, int xmax, int ymin, int ymax, int start_idx, int end_idx){
249     #pragma omp parallel
250     {
251         int omp_start_idx, omp_end_idx;
252         partition(end_idx-start_idx, omp_get_num_threads(), omp_get_thread_num(),
253                 &omp_start_idx, &omp_end_idx);

```

```

254     for (int i = start_idx+omp_start_idx; i < start_idx+omp_end_idx; i++){
255         float x = b[i*dim+0] + c[i*dim+0];
256         float y = b[i*dim+1] + c[i*dim+1];
257         if (x < xmin) x += 2 * (xmin - x);
258         else if (x > xmax) x += 2 * (xmax - x);
259         if (y < ymin) y += 2 * (ymin - y);
260         else if (y > ymax) y += 2 * (ymax - y);
261         a[i*dim+0] = x;
262         a[i*dim+1] = y;
263     }
264 }
265 }
266
267 void vec_assign_const(float *a, float c, int dim){
268     for (int i = 0; i < dim; i++){
269         a[i] = c;
270     }
271 }
272
273 void random_generate(float *xarr, float *marr, int N, int dim){
274     for (int i = 0; i < N; i++){
275         for (int j = 0; j < dim; j++){
276             float x = (float) rand() / RAND_MAX * 4 - 2;
277             xarr[i*dim+j] = x;
278         }
279         float m = (float) rand() / RAND_MAX + 1;
280         marr[i] = m;
281     }
282 }
283
284
285 void compute_seq(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr, int N,
286     int dim,
287     float G, float dt, float radius){
288     float *tmp;
289     float *xarr = *xarr_ptr;
290     float *xarr0 = *xarr0_ptr;
291     vec_assign_const(dxarr, 0, N*dim);
292     verlet_at2(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
293     vec_add(dxarr, dxarr, xarr, 1.0, 1.0, N*dim); // dx: x(t)
294     vec_add(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim); // dx: x(t-dt)
295     *xarr0_ptr = xarr;
296     *xarr_ptr = xarr0; // switch pointers
297     xarr = *xarr_ptr;
298     xarr0 = *xarr0_ptr;
299     verlet_add(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax); // xarr = xarr(0)
300     + dxarr
301 }
302
303 void compute_omp(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr,
304     int N, int dim, float G, float dt, float radius){
305     float *xarr = *xarr_ptr;
306     float *xarr0 = *xarr0_ptr;
307     float *tmp;
308     vec_assign_const(dxarr, 0, N*dim);
309     verlet_at2_omp(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
310     vec_add_omp(dxarr, dxarr, xarr, 1.0, 1.0, N*dim); // dx: x(t)
311     vec_add_omp(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim); // dx: x(t-dt)
312     *xarr0_ptr = xarr;
313     *xarr_ptr = xarr0; // switch pointers
314     xarr = *xarr_ptr;
315     xarr0 = *xarr0_ptr;
316     verlet_add_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax); // xarr =
317     xarr(0) + dxarr
318 }
319
320 typedef struct pthArgs{

```

```

318     int dim;
319     float *marr;
320     float *xarr;
321     float *xarr0;
322     float *dxarr;
323     float dt;
324     float G;
325     int N;
326     float cut;
327     int nt;
328     int idx;
329     pthread_barrier_t *barr_ptr;
330 } PthArgs;
331
332 void *compute_pth_callee(void *vargs){
333     // initialization
334     PthArgs args = *(PthArgs *) vargs;
335     int dim = args.dim;
336     float *marr = args.marr;
337     float *xarr = args.xarr;
338     float *xarr0 = args.xarr0;
339     float *dxarr = args.dxarr;
340     float dt = args.dt;
341     float G = args.G;
342     int N = args.N;
343     float radius = args.cut;
344     int nt = args.nt;
345     int idx = args.idx;
346     pthread_barrier_t *barr_ptr = args.barr_ptr;
347     int start_idx, end_idx;
348
349     // verlet algorithm
350     partition(N, nt, idx, &start_idx, &end_idx);
351     verlet_at2_part(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx, end_idx)
352     ;
353     // vector add
354     vec_add_part(dxarr, dxarr, xarr, 1.0, 1.0, N*dim, start_idx*dim, end_idx*dim);
355     pthread_barrier_wait(barr_ptr);
356     vec_add_part(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim, start_idx*dim, end_idx*dim);
357     pthread_barrier_wait(barr_ptr);
358     float *tmp = xarr;
359     xarr = xarr0;
360     xarr0 = tmp;
361     pthread_barrier_wait(barr_ptr);
362     verlet_add_part(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax, start_idx,
363                     end_idx);
364
365     return NULL;
366 }
367
368 void compute_pth(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr,
369 int N, int dim, float G, float dt, float radius, int nt){
370     float *tmp;
371     float *xarr = *xarr_ptr;
372     float *xarr0 = *xarr0_ptr;
373     pthread_t threads[nt];
374     pthread_barrier_t barr;
375     PthArgs args_arr[nt];
376     pthread_barrier_init(&barr, NULL, nt);
377     // call verlet
378     vec_assign_const(dxarr, 0, N*dim);
379     for (int i = 0; i < nt; i++){
380         args_arr[i] = (PthArgs){.dim=dim, .marr=marr, .xarr=xarr, .xarr0=xarr0,
381                                 .dxarr=dxarr, .dt=dt, .G=G, .N=N, .cut=radius,
382                                 .nt=nt, .idx=i, .barr_ptr=&barr};
383         pthread_create(&threads[i], NULL, compute_pth_callee, (void *)&args_arr[i]);
384     }

```





```

383 // join threads
384 for (int i = 0; i < nt; i++)
385     pthread_join(threads[i], NULL);
386 // switch pointers
387 *xarr_ptr = xarr0;
388 *xarr0_ptr = xarr;
389 }
390
391 void arr_check_if_identical(float *a, float *b, int dim){
392     for (int i = 0; i < dim; i++){
393         if (a[i]!=b[i]){
394             printf("fuck\n");
395             exit(1);
396         }
397     }
398 }
399
400 void runtime_record(char *jobtype, int N, int nt, double fps){
401     const char *folder = "data";
402     mkdir(folder, 0777);
403     FILE* outfile;
404     char filebuff[200];
405     snprintf(filebuff, sizeof(filebuff), "../s/runtime_%s.txt", folder, jobtype);
406     outfile = fopen(filebuff, "a");
407     fprintf(outfile, "%10d %5d %10.4f\n", N, nt, fps);
408     fclose(outfile);
409     printf("Runtime added in %s.\n", filebuff);
410 }

```

## src/utils.cuh

```

1 #pragma once
2 #include <cuda.h>
3 #include <cuda_runtime.h>
4 #include <cuda_runtime_api.h>
5 #include <cuda_device_runtime_api.h>
6 #include <driver_types.h>
7
8 void initialize_cu(float *marr, float *xarr, int N, int dim, int Tx, int Ty,
9     float xmin, float xmax, float ymin, float ymax);
10 void compute_cu(float *xarr, int nsteps, int N, int dim, float G, float dt, float cut);
11 void finalize_cu();

```

## src/const.h

```

1 #pragma once
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <iostream>
5
6 // global variables
7 // computing-related constants
8 int N = 30; // number of particles
9 int nsteps = 1e5; // number of steps
10 int dim = 2; // dimension
11 float radius = 0.01; // gravity cut-off
12 float G = 0.1; // gravity constant
13 float dt = 0.001; // time step
14 float *marr; // mass array
15 float *xarr; // position array at time t
16 float *xarr0; // position array at time t - dt
17 float *varr; // velocity array
18 float *dxarr; // position shift array
19 float *dvarr; // velocity shift array
20 float xmin = -10;
21 float xmax = 10;

```

```
22 float ymin = -10;
23 float ymax = 10;
24
25 // IO & runtime options
26 int record = 0;
27 int nt = 1;
28
29 // mpi parameters
30 int size, rank;
31 float *xarr_copy;
32
33 // cuda parameters
34 int Tx = 16;
35 int Ty = 16;
```