

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 `build/lib/libcudalib.a`.

One can refer to A.1 to understand the program design.

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
```

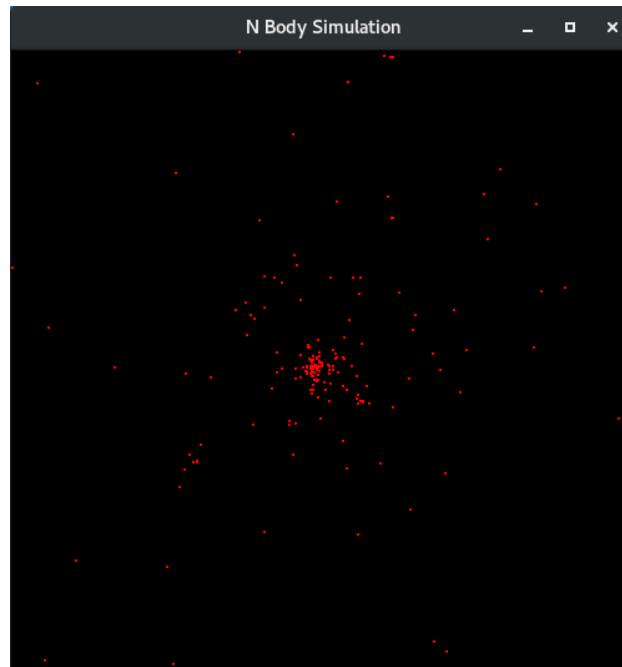


Figure 1: Sample GUI window

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 can 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, which have $10 \times 2 = 20$ threads in total.

```
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.

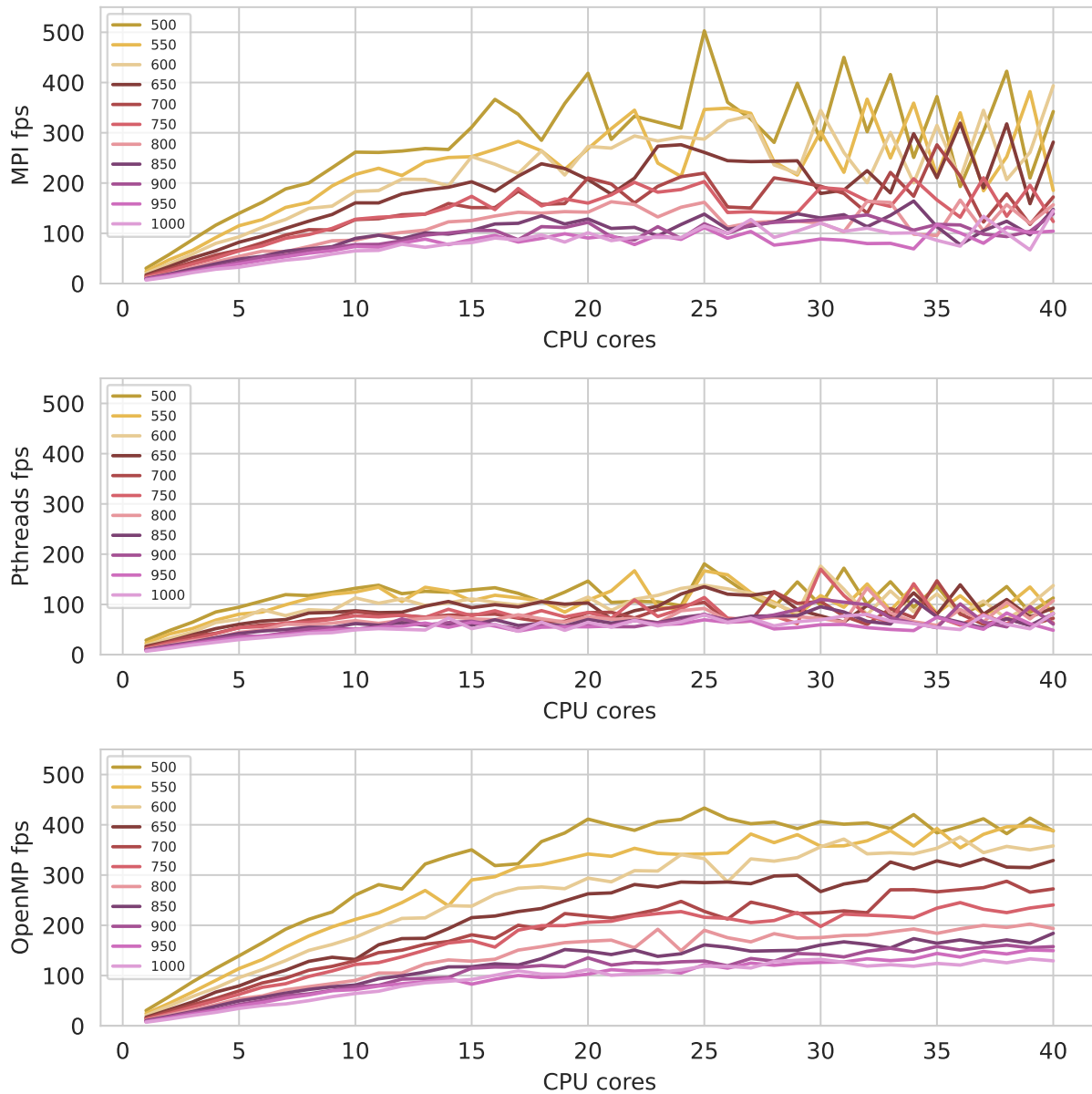
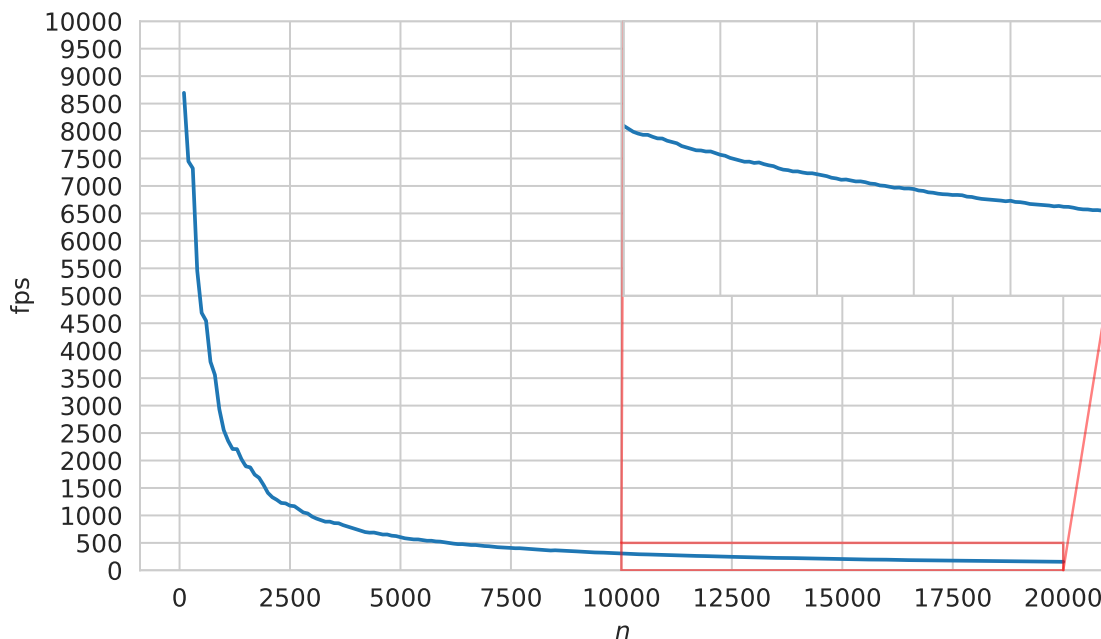


Figure 2: fps vs the number of threads/processes plot.

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/utils.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.2 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 `cudalib.cu`). According to NVIDIA, the memory access on shared memory is approximately 100× faster than the global (`__device__`) memory access.

For example, a naive vector addition in CUDA could be written as

```

1  __global__ void VecAdd(int *a, int *b, int *c, long int dim){
2      // thread partition
3      int start_idx = dim / (blockDim.x * gridDim.x) * threadIdx.x;
4      int end_idx   = dim / (blockDim.x * gridDim.x) * (threadIdx.x+1);
5      if (threadIdx.x+1==blockDim.x) end_idx = dim;
6      // vector add

```

```

7   for (int i = 0; i < dim; i++){
8       c[i] = a[i] + b[i];
9   }
10 }

```

During the calculation, each thread in GPU will require to access the memory independently. When the overall thread number is large, the memory miss could cost a huge amount of time. However, in CUDA, we can split those threads into different blocks: for example, if one call a kernel function kernel by kernel<<<16,64>>>(), then he is asking CUDA to generate 16 blocks where each block has 64 threads, overall $16 \times 64 = 1024$ threads. Similarly, kernel<<<1,1024>>>() also calls the function with 1024 threads. In principle, VecAdd<<<16,64>>>(a, b, c, dim) and VecAdd<<<1,1024>>>(a, b, c, dim) has no difference. Now consider, if we can let threads in each block, share a part of memory, then can it reduce the time cost by memory miss? Have a look at the following function

```

1  #define BLOCKSIZE 64
2  __global__ void sharedMemVecAdd(int *a, int *b, int *c, long int dim){
3      // block partition
4      int block_start_idx = dim / gridDim.x * blockIdx.x;
5      int block_end_idx = dim / gridDim.x * (blockIdx.x + 1);
6      if (blockIdx.x+1==gridDim.x) block_end_idx = dim;
7      int total_task = block_end_idx - block_start_idx;
8      // shared memory partition
9      int num_iter = (total_task + BLOCK_SIZE - 1) / BLOCK_SIZE;
10     // block-wise shared memory
11     __shared__ int a_t[BLOCK_SIZE*2];
12     __shared__ int b_t[BLOCK_SIZE];
13     __shared__ int c_t[BLOCK_SIZE];
14     __syncthreads();
15
16     // main program
17     for (int i = 0; i < num_iter; i++){
18         if (threadIdx.x+i*BLOCK_SIZE < block_end_idx){
19             // thread
20             // copy data
21             a_t[threadIdx.x] = a[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
22             b_t[threadIdx.x] = b[block_start_idx + threadIdx.x + BLOCK_SIZE*i];
23             __syncthreads();
24
25             // vector add
26             c_t[threadIdx.x] = a_t[threadIdx.x] + b_t[threadIdx.x];
27
28             // copy data back
29             c[block_start_idx + threadIdx.x + BLOCK_SIZE*i] = c_t[threadIdx.x];
30             __syncthreads();
31         }
32     }
33 }

```

One should convince himself that sharedMemVecAdd<<<16,BLOCKSIZE>>>(a, b, c, dim) do the exact same work as VecAdd. So what is the difference here? In each block, CUDA will create a shared memory, that is a fast memory accessible by ALL threads within this block. During the computation, the block will first read a memory block, then perform computation; after all threads finish the computation, the threads will write data back to the global memory.

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

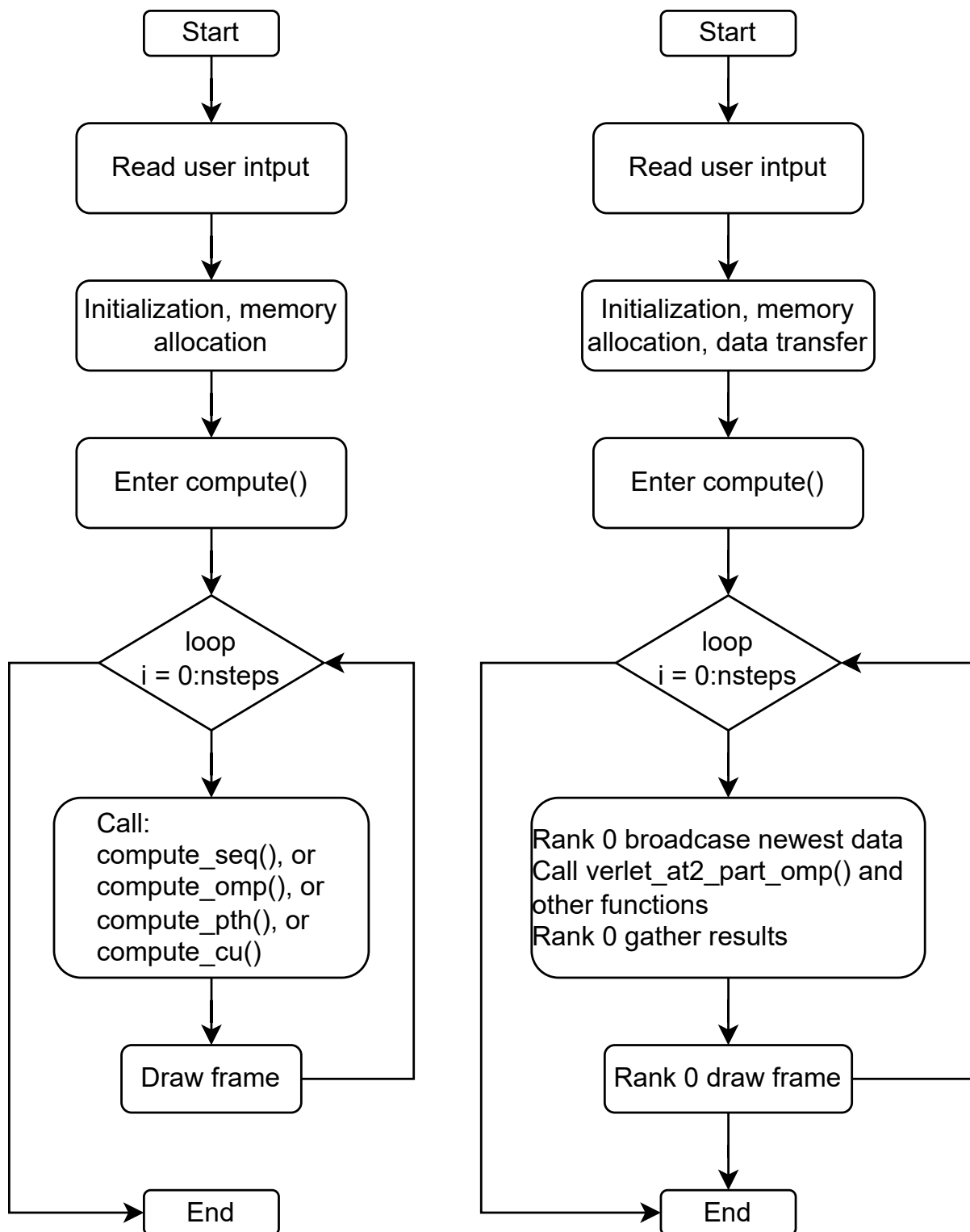
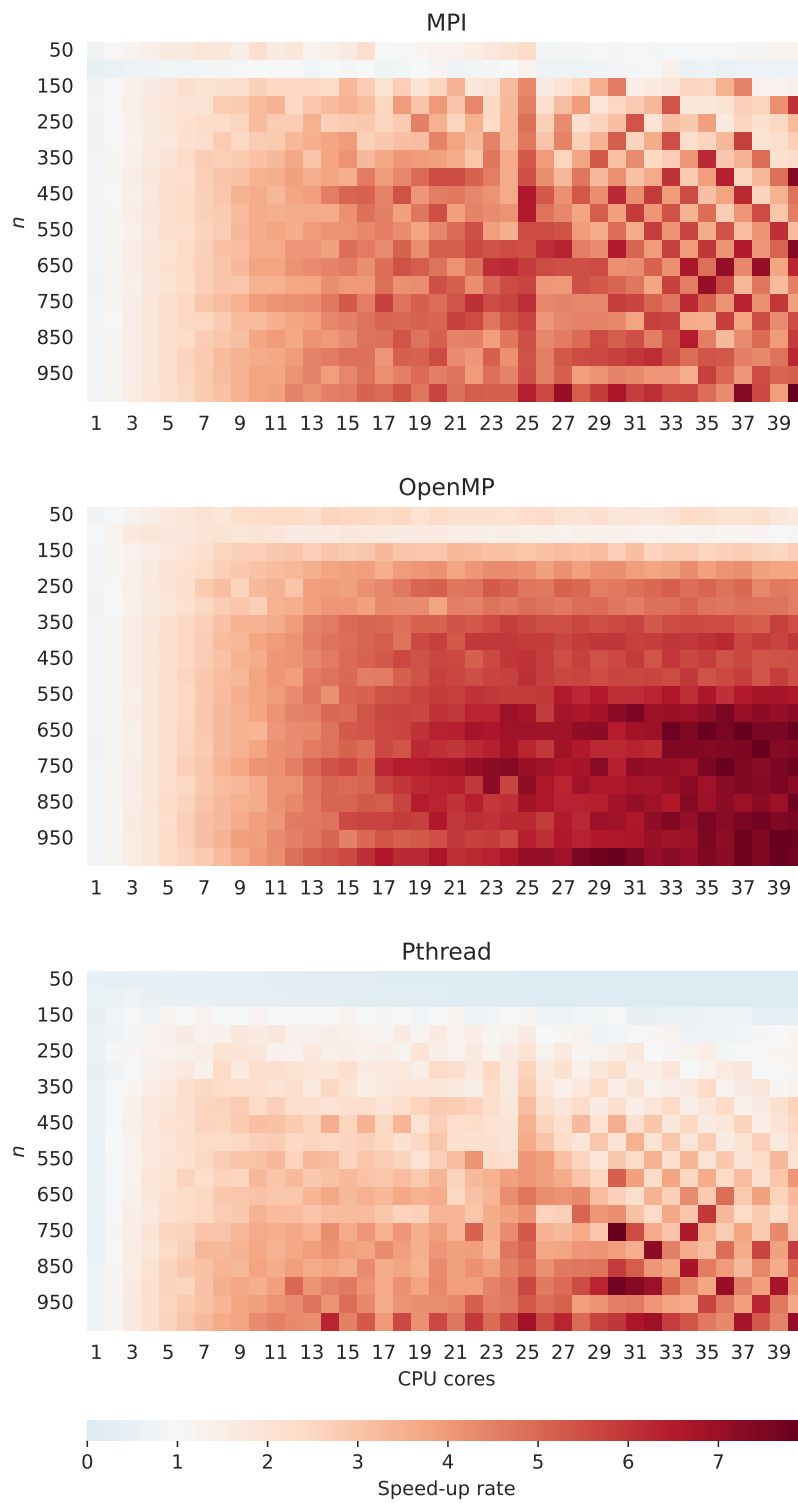


Figure A.1: Program flowchart

Figure A.2: 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.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  #include "utils.cuh"
9  #ifdef GUI
10 #include "gui.h"
11 #endif
12
13 void compute(){
14     // running type buffer
15     char type[1000];
16     // start timing
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     // main program
22     for (int s = 0; s < nsteps; s++){
23         // verlet omp
24         #ifdef OMP
25             if (s==0) {
26                 printf("Start OpenMP version.\n");
27                 strcpy(type, "omp");
28             }
29             compute_omp(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius);
30         #elif CUDA
31             if (s==0) {
32                 printf("Start CUDA version.\n");
33                 strcpy(type, "cuda");
34             }
35             compute_cu(xarr, nsteps, N, dim, G, dt, radius);
36         #elif PTH
37             if (s==0) {
38                 printf("Start Pthread version.\n");
39                 strcpy(type, "pth");
40             }
41             compute_pth(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius, nt);
42         #else
43             if (s==0) {
44                 printf("Start sequential version.\n");
45                 strcpy(type, "seq");
46             }
47             compute_seq(&xarr, &xarr0, dxarr, marr, N, dim, G, dt, radius);
48         #endif
49
50
51         // calculating fps
52         int step = 200;
53         #ifdef GUI
54             step = 30;
55         #endif

```

```

56     if (s%step==0 && s%(step*2)!=0) t1 = std::chrono::high_resolution_clock::now();
57     else if (s%(step*2)==0 && s!=0) {
58         t2 = std::chrono::high_resolution_clock::now();
59         t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count()
        ;
60         printf("fps: %f frame/s\n", step/t);
61     }
62
63     // opengl
64     #ifdef GUI
65     glClear(GL_COLOR_BUFFER_BIT);
66     glColor3f(1.0f, 0.0f, 0.0f);
67     glPointSize(2.0f);
68
69     // gl points
70     glBegin(GL_POINTS);
71     float xi;
72     float yi;
73     float xmin, xmax, ymin, ymax;
74     for (int i = 0; i < N; i++){
75         xi = xarr[i*dim+0];
76         yi = xarr[i*dim+1];
77         glVertex2f(xi, yi);
78     }
79     glEnd();
80
81     glFlush();
82     glutSwapBuffers();
83     #endif
84 }
85
86 // record data
87 if (record==1){
88     t2 = std::chrono::high_resolution_clock::now();
89     t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
90     double fps = nsteps / t;
91     runtime_record(type, N, nt, fps);
92 }
93 }
94
95 int main(int argc, char *argv[]){
96     // parse argument
97     char buff[200];
98     for (int i = 0; i < argc; i++){
99         strcpy(buff, argv[i]);
100         if (strcmp(buff, "-n")==0){
101             std::string num(argv[i+1]);
102             N = std::stoi(num);
103         }
104         if (strcmp(buff, "-nt")==0){
105             std::string num(argv[i+1]);
106             nt = std::stoi(num);
107         }
108         if (strcmp(buff, "--xmin")==0){
109             std::string num(argv[i+1]);
110             xmin = std::stof(num);
111         }
112         if (strcmp(buff, "--xmax")==0){
113             std::string num(argv[i+1]);
114             xmax = std::stof(num);
115         }
116         if (strcmp(buff, "--ymin")==0){
117             std::string num(argv[i+1]);
118             ymin = std::stof(num);
119         }
120         if (strcmp(buff, "--ymax")==0){
121             std::string num(argv[i+1]);

```

```

122         ymax = std::stof(num);
123     }
124     if (strcmp(buff, "--nsteps")==0){
125         std::string num(argv[i+1]);
126         nsteps = std::stof(num);
127     }
128     if (strcmp(buff, "--record")==0){
129         std::string num(argv[i+1]);
130         record = std::stoi(num);
131     }
132     if (strcmp(buff, "--Tx")==0){
133         std::string num(argv[i+1]);
134         Tx = std::stoi(num);
135     }
136     if (strcmp(buff, "--Ty")==0){
137         std::string num(argv[i+1]);
138         Ty = std::stoi(num);
139     }
140 }
141 // omp options
142 #ifdef OMP
143 omp_set_dynamic(0);
144 omp_set_num_threads(nt);
145 #endif
146
147 // print info
148 print_info(N, nsteps);
149
150 // array allocation
151 marr = (float *)malloc(sizeof(float) * N);
152 xarr = (float *)malloc(sizeof(float) * N * dim);
153 xarr0 = (float *)malloc(sizeof(float) * N * dim);
154 dxarr = (float *)malloc(sizeof(float) * N * dim);
155
156 // random generate initial condition
157 random_generate(xarr, marr, N, dim);
158 print_arr(xarr, 8);
159
160 // initialization
161 vec_add(xarr0, xarr0, xarr, 0, 1, N*dim);
162
163 // cuda initialize
164 #ifdef CUDA
165 Tx = 16;
166 Ty = 16;
167 initialize_cu(marr, xarr, N, dim, Tx, Ty, xmin, xmax, ymin, ymax);
168 #endif
169
170 // start timing
171 auto t1 = std::chrono::high_resolution_clock::now();
172 // main program
173 #ifdef GUI
174 glutInit(&argc, argv);
175 glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
176 glutInitWindowPosition(0, 0);
177 glutInitWindowSize(500, 500);
178 glutCreateWindow("N Body Simulation");
179 glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
180 glutDisplayFunc(&compute);
181 glutKeyboardFunc(&guiExit);
182 gluOrtho2D(xmin, xmax, ymin, ymax);
183 glutSetOption( GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);
184 glutMainLoop();
185 #else
186 compute();
187 // cudaDeviceSynchronize();
188 #endif

```

```

189
190 // end timing
191 auto t2 = std::chrono::high_resolution_clock::now();
192 double t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t1).count();
193
194 printf("Duration: %fs\n", t);
195
196 // free
197 free(marr);
198 free(xarr);
199 free(xarr0);
200 free(dxarr);
201
202 #ifdef CUDA
203 // cudafree
204 finalize_cu();
205 cudaDeviceSynchronize();
206 #endif
207
208 return 0;
209 }

```

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,
    end_idx);
31 // verlet_at2_part(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius, start_idx,
    end_idx);
32 vec_add_part(dxarr, dxarr, xarr, 1.0, 1.0, N*dim, start_idx*dim, end_idx*dim);
33 vec_add_part(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim, start_idx*dim, end_idx*dim);
34 float *tmp = xarr;
35 xarr = xarr0;
36 xarr0 = tmp;
37 MPI_Barrier(MPI_COMM_WORLD);
38 verlet_add_part_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax,
    start_idx, end_idx);
39 // verlet_add_part(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax, start_idx
    , end_idx);

```

```

40
41 // transfer data
42 if (rank==0) MPI_Gather(MPI_IN_PLACE, jobsize*dim, MPI_FLOAT, xarr+start_idx*dim
    , jobsize*dim, MPI_FLOAT, 0, MPI_COMM_WORLD);
43 else MPI_Gather(xarr+start_idx*dim, jobsize*dim, MPI_FLOAT, xarr, jobsize*dim,
    MPI_FLOAT, 0, MPI_COMM_WORLD);
44 MPI_Barrier(MPI_COMM_WORLD);
45 // solve tail case
46 if (N%jobsize!=0) {
47     if (rank==0){
48         MPI_Recv(xarr+(N/size*size)*dim, (N%jobsize)*dim, MPI_FLOAT, size-1, 0,
49             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
50     }
51     else if (rank+1==size){
52         MPI_Send(xarr+(N/size*size)*dim, (N%jobsize)*dim, MPI_FLOAT, 0, 0,
53             MPI_COMM_WORLD);
54     }
55 MPI_Barrier(MPI_COMM_WORLD);
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     // gl points
73     glBegin(GL_POINTS);
74     float xi;
75     float yi;
76     float xmin, xmax, ymin, ymax;
77     for (int i = 0; i < N; i++){
78         xi = xarr[i*dim+0];
79         yi = xarr[i*dim+1];
80         glVertex2f(xi, yi);
81     }
82     glEnd();
83     glFlush();
84     glutSwapBuffers();
85     #endif
86 }
87 }
88 // record data
89 if (rank==0 && record==1){
90     t2 = std::chrono::high_resolution_clock::now();
91     t = std::chrono::duration_cast<std::chrono::duration<double>>(t2-t0).count();
92     double fps = nsteps / t;
93     runtime_record(type, N, size, fps);
94 }
95 }
96 }
97
98 int main(int argc, char* argv[]){
99     // mpi initializatio
100     MPI_Init(NULL, NULL);

```

```

101 // fetch size and rank
102 MPI_Comm_size(MPI_COMM_WORLD, &size);
103 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
104
105 // parse arguments
106 char buff[200];
107 for (int i = 0; i < argc; i++){
108     strcpy(buff, argv[i]);
109     if (strcmp(buff, "-n")==0){
110         std::string num(argv[i+1]);
111         N = std::stoi(num);
112     }
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");
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/cudalib.cu

```

1  #include "utils.cuh"
2  #include "const.cuh"
3  #define BLOCK_SIZE 1024
4
5  #define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
6  inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
7  {
8      if (code != cudaSuccess)
9      {
10         fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
11         if (abort) exit(code);
12     }
13 }
14
15 __device__ void get_xij_d(int i, int j, int dim, float *xarr, float *xij, int N){
16     for (int k = 0; k < dim; k++){
17         xij[k] = xarr[j*dim+k] - xarr[i*dim+k];
18     }
19 }
20
21 __device__ void partition_d(int nsteps, int size, int idx, int *start_ptr, int *end_ptr)
22 {
23     *start_ptr = nsteps / size * idx;
24     *end_ptr = nsteps / size * (idx+1);
25     if (idx+1==size) *end_ptr = nsteps;
26 }
27
28 __device__ float norm_d(float *x, int dim){
29     float r = 0;
30     for (int i = 0; i < dim; i++){
31         r += x[i]*x[i];
32     }
33     r = sqrt(r);
34     return r;
35 }
36
37 __device__ void vec_add_d(float *a, float *b, float *c,
38     float fac1, float fac2, int dim){
39     for (int i = 0; i < dim; i++){
40         a[i] = fac1*b[i] + fac2*c[i];

```

```

40     }
41 }
42
43
44 __global__ void vec_add_cu(float *a, float *b, float *c, int dim){
45     int size = blockDim.x * gridDim.x;
46     int idx = blockDim.x*blockIdx.x + threadIdx.x;
47     int start_idx, end_idx;
48     partition_d(dim, size, idx, &start_idx, &end_idx);
49     for (int i = start_idx; i < end_idx; i++){
50         a[i] = b[i] + c[i];
51     }
52 }
53
54 __global__ void verlet_add_cu(float *a, float *b, float *c, int N, int dim,
55     int xmin, int xmax, int ymin, int ymax){
56     int size = blockDim.x * gridDim.x;
57     int idx = blockDim.x*blockIdx.x + threadIdx.x;
58     int start_idx, end_idx;
59     partition_d(N, size, idx, &start_idx, &end_idx);
60     for (int i = start_idx; i < end_idx; i++){
61         float x = b[i*dim+0] + c[i*dim+0];
62         float y = b[i*dim+1] + c[i*dim+1];
63         if (x < xmin) x += 2 * (xmin - x);
64         else if (x > xmax) x += 2 * (xmax - x);
65         if (y < ymin) y += 2 * (ymin - y);
66         else if (y > ymax) y += 2 * (ymax - y);
67         a[i*dim+0] = x;
68         a[i*dim+1] = y;
69     }
70 }
71
72 __global__ void gather_dx_cu(float *a, float *b, float *c, int dim){
73     int size = blockDim.x * gridDim.x;
74     int idx = blockDim.x*blockIdx.x + threadIdx.x;
75     int start_idx, end_idx;
76     partition_d(dim, size, idx, &start_idx, &end_idx);
77     for (int i = start_idx; i < end_idx; i++){
78         a[i] += b[i] - c[i];
79     }
80 }
81
82 __global__ void print_arr_cu(float *arr, int dim){
83     for (int i = 0; i < dim; i++){
84         printf("%f ", arr[i]);
85     }
86     printf("\n");
87 }
88
89 __device__ void print_arr_d(float *arr, int dim){
90     for (int i = 0; i < dim; i++){
91         printf("%f ", arr[i]);
92     }
93     printf("\n");
94 }
95
96 __global__ void verlet_at2_cu(const int dim, float *marr, float *xarr, float *xarr0,
97     float *dxarr, float dt, float G, int N, float cut){
98     // partition
99     int size = gridDim.x;
100    int idx = blockIdx.x;
101    int block_start_idx, block_end_idx;
102    partition_d(N, size, idx, &block_start_idx, &block_end_idx);
103    // if (threadIdx.x==0) printf("%d %d\n", block_start_idx, block_end_idx);
104    // shared memory
105    __shared__ float marr_t[BLOCK_SIZE];
106    __shared__ float xarr_l_t[BLOCK_SIZE*2];

```

```

107  __shared__ float xarr_g_t[BLOCK_SIZE*2];
108  __shared__ float dxarr_t[BLOCK_SIZE*2];
109  // G*dt*dt factor
110  float fac = G*dt*dt;
111  for (int i = block_start_idx; i < block_end_idx; i+=BLOCK_SIZE){
112      // tmp variables
113      float tmpx = 0.0;
114      float tmpy = 0.0;
115      if (i + threadIdx.x < block_end_idx){
116          // get local coords
117          xarr_l_t[threadIdx.x*dim+0] = xarr[i*dim+threadIdx.x*dim+0];
118          xarr_l_t[threadIdx.x*dim+1] = xarr[i*dim+threadIdx.x*dim+1];
119      }
120      __syncthreads();
121      // N loop
122      for (int j = 0; j < N; j+=BLOCK_SIZE){
123          if (threadIdx.x + j < N){
124              marr_t[threadIdx.x] = marr[threadIdx.x+j];
125              xarr_g_t[threadIdx.x*dim+0] = xarr[threadIdx.x*dim+j*dim+0];
126              xarr_g_t[threadIdx.x*dim+1] = xarr[threadIdx.x*dim+j*dim+1];
127          }
128          __syncthreads();
129          // if (blockIdx.x==0 && threadIdx.x==0 && j==0) print_arr_d(xarr_g_t, 8);
130          for (int k = 0; k < BLOCK_SIZE; k++){
131              if (k + j < N && threadIdx.x + j < N){
132                  // compute xij
133                  float xij0 = xarr_g_t[k*dim+0] - xarr_l_t[threadIdx.x*dim+0];
134                  float xij1 = xarr_g_t[k*dim+1] - xarr_l_t[threadIdx.x*dim+1];
135                  float rij = sqrt(xij0*xij0 + xij1*xij1);
136                  if (rij < cut) rij = cut;
137                  tmpx += xij0/(rij*rij*rij) * marr_t[k]*fac;
138                  tmpy += xij1/(rij*rij*rij) * marr_t[k]*fac;
139              }
140              // assign value to shared memory
141          }
142          if (i + threadIdx.x < block_end_idx){
143              // assign value back to global memory
144              dxarr_t[threadIdx.x*dim+0] = tmpx;
145              dxarr_t[threadIdx.x*dim+1] = tmpy;
146          }
147          __syncthreads();
148          if (i + threadIdx.x < block_end_idx){
149              dxarr[threadIdx.x*dim+i*dim+0] = dxarr_t[threadIdx.x*dim+0];
150              dxarr[threadIdx.x*dim+i*dim+1] = dxarr_t[threadIdx.x*dim+1];
151          }
152          __syncthreads();
153      }
154  }
155
156  // cuda initialize program
157  void initialize_cu(float *marr, float *xarr, int N, int dim, int Tx, int Ty,
158      float xmin, float xmax, float ymin, float ymax){
159      printf("cuda initialize\n");
160      // cuda parameters
161      Tx_cu = Tx;
162      Ty_cu = Ty;
163      xmin_d = xmin;
164      xmax_d = xmax;
165      ymin_d = ymin;
166      ymax_d = ymax;
167      // cuda memory allocation
168      gpuErrchk( cudaMalloc((void **) &marr_d, sizeof(float)*N));
169      gpuErrchk( cudaMalloc((void **) &xarr_d, sizeof(float)*N*dim));
170      gpuErrchk( cudaMalloc((void **) &xarr0_d, sizeof(float)*N*dim));
171      gpuErrchk( cudaMalloc((void **) &dxarr_d, sizeof(float)*N*dim));
172      // copy
173      gpuErrchk( cudaMemcpy(marr_d, marr, sizeof(float)*N, cudaMemcpyHostToDevice) );

```

```

174     gpuErrchk( cudaMemcpy(xarr_d, xarr, sizeof(float)*N*dim, cudaMemcpyHostToDevice) );
175     gpuErrchk( cudaMemcpy(xarr0_d, xarr, sizeof(float)*N*dim, cudaMemcpyHostToDevice) );
176
177     // print check: passed
178     // print_arr_cu<<<1,1>>>(marr_d, N);
179     cudaDeviceSynchronize();
180 }
181
182 // switch pointers
183 __global__ void swap(float * &a, float * &b){
184     float *tmp = a;
185     a = b;
186     b = tmp;
187 }
188
189 // verlet cuda callee
190 void compute_cu(float *xarr, int nsteps, int N, int dim, float G, float dt, float cut){
191     // verlet cuda main program
192     float *tmp;
193     cudaMemset(dxarr_d, 0x00, sizeof(float)*N*dim);
194     cudaDeviceSynchronize();
195     verlet_at2_cu<<<32,BLOCK_SIZE>>>(dim, marr_d, xarr_d, xarr0_d, dxarr_d, dt, G, N,
        cut); // dx: acc
196     cudaDeviceSynchronize();
197     gather_dx_cu<<<Tx_cu,Ty_cu>>>(dxarr_d, xarr_d, xarr0_d, N*dim);
198     cudaDeviceSynchronize();
199     tmp = xarr_d;
200     xarr_d = xarr0_d;
201     xarr0_d = tmp;
202     verlet_add_cu<<<Tx_cu,Ty_cu>>>(xarr_d, xarr0_d, dxarr_d, N, dim, xmin_d, xmax_d,
        ymin_d, ymax_d);
203
204     cudaDeviceSynchronize();
205     cudaMemcpy(xarr, xarr_d, sizeof(float)*N*dim, cudaMemcpyDeviceToHost);
206
207     #ifdef GUI
208     // copy x to host
209     cudaMemcpy(xarr, xarr_d, sizeof(float)*N*dim, cudaMemcpyDeviceToHost);
210     cudaDeviceSynchronize();
211     #endif
212 }
213
214 // cuda finalize program
215 void finalize_cu(){
216     // free
217     printf("cuda finalize\n");
218     gpuErrchk( cudaFree(marr_d) );
219     gpuErrchk( cudaFree(xarr_d) );
220     gpuErrchk( cudaFree(xarr0_d) );
221     gpuErrchk( cudaFree(dxarr_d) );
222 }

```

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 void verlet_at2_omp(int dim, float *marr, float *xarr, float *xarr0,
110                    float *dxarr, float dt, float G, int N, float cut){
111     #pragma omp parallel for
112     for (int i = 0; i < N; i++){
113         float tmp[dim];
114         for (int j = 0; j < dim; j++) tmp[j] = 0;
115         for (int j = 0; j < N; j++){
116             if (j!=i){
117                 float xij[dim];
118                 float mi = marr[i];
119                 float mj = marr[j];
120                 // get xij
121                 get_xij(i, j, dim, xarr, xij, N);
122                 // compute rij
123                 float rij = norm(xij, dim);
124                 float fac = 1.0;
125                 if (rij < cut) {
126                     rij = cut;
127                 }
128                 // compute intermediate variable
129                 for (int k = 0; k < dim; k++){
130                     tmp[k] += xij[k]*G/pow(rij, 3) *mj*dt*dt;
131                 }
132             }
133         }
134         vec_add(dxarr+i*dim, dxarr+i*dim, tmp, 1.0, 1.0, dim);
135     }
136 }
137
138 void verlet_at2_part(int dim, float *marr, float *xarr, float *xarr0,
139                     float *dxarr, float dt, float G, int N, float cut,
140                     int start_idx, int end_idx){
141     for (int i = start_idx; i < end_idx; i++){
142         float tmp[dim];
143         for (int j = 0; j < dim; j++) tmp[j] = 0;
144         for (int j = 0; j < N; j++){
145             if (j!=i){
146                 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,
    int dim,
    float G, float dt, float radius){
286     float *tmp;
287     float *xarr = *xarr_ptr;
288     float *xarr0 = *xarr0_ptr;
289     vec_assign_const(dxarr, 0, N*dim);
290     verlet_at2(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
291     vec_add(dxarr, dxarr, xarr, 1.0, 1.0, N*dim); // dx: x(t)
292     vec_add(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim); // dx: x(t-dt)
293     *xarr0_ptr = xarr;
294     *xarr_ptr = xarr0; // switch pointers
295     xarr = *xarr_ptr;
296     xarr0 = *xarr0_ptr;
297     verlet_add(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax); // xarr = xarr(0)
298     + dxarr
299 }
300
301 void compute_omp(float **xarr_ptr, float **xarr0_ptr, float *dxarr, float *marr,
    int N, int dim, float G, float dt, float radius){
302     float *xarr = *xarr_ptr;
303     float *xarr0 = *xarr0_ptr;
304     float *tmp;
305     vec_assign_const(dxarr, 0, N*dim);
306     verlet_at2_omp(dim, marr, xarr, xarr0, dxarr, dt, G, N, radius); // dx: acc
307     vec_add_omp(dxarr, dxarr, xarr, 1.0, 1.0, N*dim); // dx: x(t)
308     vec_add_omp(dxarr, dxarr, xarr0, 1.0, -1.0, N*dim); // dx: x(t-dt)
309     *xarr0_ptr = xarr;
310     *xarr_ptr = xarr0; // switch pointers
311     xarr0 = *xarr0_ptr;
312     xarr = *xarr_ptr;
313     verlet_add_omp(xarr, xarr0, dxarr, N, dim, xmin, xmax, ymin, ymax); // xarr =
314     xarr(0) + dxarr
315 }
316
317 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 }
385 // join threads
386 for (int i = 0; i < nt; i++)
387 pthread_join(threads[i], NULL);
388 // switch pointers
389 *xarr_ptr = xarr0;
390 *xarr0_ptr = xarr;
391 }
392
393 void arr_check_if_identical(float *a, float *b, int dim){
394 for (int i = 0; i < dim; i++){
395 if (a[i]!=b[i]){
396 printf("fuck\n");
397 exit(1);
398 }
399 }
400 }
401
402 void runtime_record(char *jobtype, int N, int nt, double fps){
403 const char *folder = "data";
404 mkdir(folder, 0777);
405 FILE* outfile;
406 char filebuff[200];
407 snprintf(filebuff, sizeof(filebuff), "./%s/runtime%s.txt", folder, jobtype);
408 outfile = fopen(filebuff, "a");
409 fprintf(outfile, "%10d %5d %10.4f\n", N, nt, fps);
410 fclose(outfile);
411 printf("Runtime added in %s.\n", filebuff);
412 }

```



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 = 200; // 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;

```