CSC4005 Assignment3 Report

Introduction

The objective of this assignment is to simulate an astronomical **N-body** system in two-dimensions. In this experiment, five versions of programs have been designed: sequential implementation, Pthread implementation, MPI implementation, OpenMP implementation and MPI + OpenMP version. The souce code has been attached to this report.

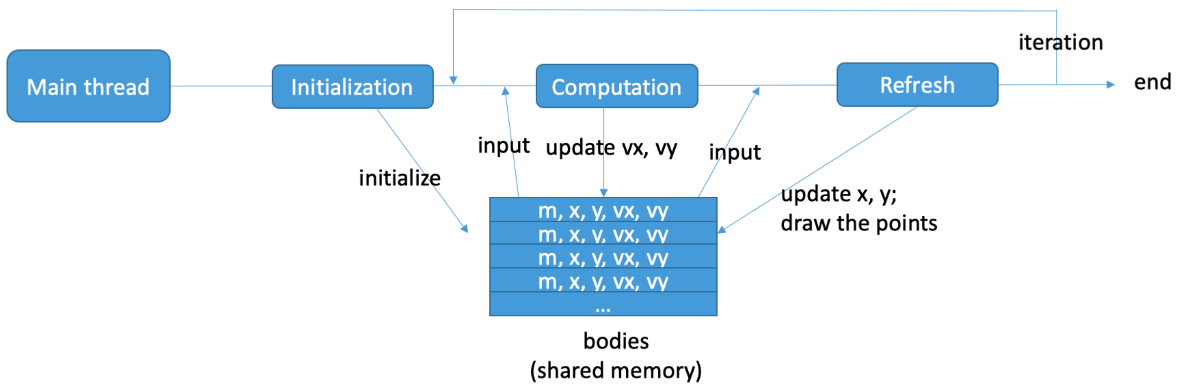
In the simulation, each body would be initialized with random mass, position and velocity, except one specific body with very large mass and zero velocity in the beginning. The chosen GUI system in the programs is Xlib.

This report is going to elaborate the detailed implementations and analyze the performances of different versions.

Program design

The logic of the sequential version is the basis of the logic of parallel versions. Generally, the parallel versions derive from the sequential version by parallelize the parallelizable work and keep the parts that are not parallelizable. Thus, in this part, the detailed implementation of N-body simulation in sequential version would be elaborated, but for parallel versions, we only focus on how to parallelize the work.

Sequential_version



The graph above shows the basic logic of the N-body simulation with sequential version. There are several main work to be done in order, the detail is as follows:

1. Initialization of the bodies.
    - Create a new type of structure for body object

```
typedef struct _body {
    double mass;
    double x, y;     // coordinate
    double vx, vy;   // velocity
}body;
```

- Initialize the data of bodies and store it in an array

```
body bodies[NUM_BODIES];  // contain all the info of bodies
for (i = 0; i < NUM_BODIES; ++i){
    // One body with large mass and 0 initial velocity.
    if (i == 0) {
        bodies[i].mass = 999999999;
        // let the large-mass body close to center
        bodies[i].x = SIZE /3 + (rand()/(double)RAND_MAX) * SIZE / 3;
        bodies[i].y = SIZE /3 + (rand()/(double)RAND_MAX) * SIZE / 3;
        bodies[i].vx = 0;
        bodies[i].vy = 0;
    }else{
        // randomize mass
        bodies[i].mass = 1000 + rand() % 10000; // from 1000 to 11000
        // randomize position
        bodies[i].x = (rand()/(double)RAND_MAX) * SIZE;
        bodies[i].y = (rand()/(double)RAND_MAX) * SIZE;
        // randomize velocity
        bodies[i].vx = rand() % 10 - 5;    // from -5 to +5
        bodies[i].vy = rand() % 10 - 5;
    }
}
```

Here, one body was set with large mass and still in the beginning, and the positions, mass and velocity of all other bodies were randomized.

2. Computation of the change of velocity for each body.

```
// compute total acceleration and change velocity for each body
for (i = 0; i < NUM_BODIES; ++i){
    for (j = 0; j < NUM_BODIES; ++j){
        if (i == j) continue;
        dx = bodies[j].x - bodies[i].x;
        dy = bodies[j].y - bodies[i].y;
        r = sqrt(dx * dx + dy * dy);     // distance
    // determine whether collision happens
    if (r < 2 * BODY_RADIUS){
      // elastic collision
        if (i < j){          // avoid repetitive computation
            // horizontal velocity change
            v1 = ((bodies[i].mass-bodies[j].mass)*bodies[i].vx + 
2*bodies[j].mass*bodies[j].vx)/(bodies[i].mass+bodies[j].mass);
            v2 = ((bodies[j].mass-bodies[i].mass)*bodies[j].vx + 
2*bodies[i].mass*bodies[i].vx)/(bodies[i].mass+bodies[j].mass);
            bodies[i].vx = v1;
            bodies[j].vx = v2;
            // vertical velocity change
            v1 = ((bodies[i].mass-bodies[j].mass)*bodies[i].vy + 
2*bodies[j].mass*bodies[j].vy)/(bodies[i].mass+bodies[j].mass);
            v2 = ((bodies[j].mass-bodies[i].mass)*bodies[j].vy + 
2*bodies[i].mass*bodies[i].vy)/(bodies[i].mass+bodies[j].mass);
            bodies[i].vy = v1;
            bodies[j].vy = v2;
        }
        // compute accerlaration between two bodies
        a = G * bodies[j].mass / (4 * BODY_RADIUS * BODY_RADIUS);
        ax = a * dx / (2 * BODY_RADIUS);
        ay = a * dy / (2 * BODY_RADIUS);
    }else {
        // compute accerlaration between two bodies
        a = G * bodies[j].mass / (r * r);   // universal gravitation
        ax = a * dx / r;
        ay = a * dy / r;
    }
    // acceleration changes velocity
    bodies[i].vx += ax;
    bodies[i].vy += ay;
  } // end for j
} // end for i
```

Here, the program computes the net accelaration of each body, by totaling up the acceleration between the chosen body and all other bodies. Then it determines whether collision happens - if happens, the velocity of the bodies would change by following the law of elastic collision. At last, it updates the velocity of each body with the computed net accelration.

3. Update the position of each body and draw on the screen.

```
// refresh every 10 iterations
if (count % 10 == 0)
    XClearWindow(display, win);

// bouncing at boundary and draw
for (i = 0; i < NUM_BODIES; ++i){
    // displacement made by velocity
    bodies[i].x += bodies[i].vx;
    bodies[i].y += bodies[i].vy;

    // left boundary
    if (bodies[i].x < 0){
        bodies[i].x = -bodies[i].x;
        bodies[i].vx = -bodies[i].vx/999;   // 99.9% energy loss
    }
    // right boundary
    if (bodies[i].x > SIZE){
        bodies[i].x = 2 * SIZE - bodies[i].x;
        bodies[i].vx = -bodies[i].vx/999;   // 99.9% energy loss
    }
    // down boundary
    if (bodies[i].y < 0){
        bodies[i].y = -bodies[i].y;
        bodies[i].vy = -bodies[i].vy/999;   // 99.9% energy loss
    }
    // up boundary
    if (bodies[i].y > SIZE){
        bodies[i].y = 2 * SIZE - bodies[i].y;
        bodies[i].vy = -bodies[i].vy/999;   // 99.9% energy loss
    }

    // refresh every 10 iterations
    if (count % 10 == 0) {
        XDrawPoint (display, win, gc, bodies[i].x, bodies[i].y);
        usleep(10);
    }
} // end for i
```

Here, the positions of all the bodies were updated on basis of the velocity. Also, to avoid that the updated positions were outside the window, it would bounce back with 99.9% velocity loss if it is outside the window. After that, the updated points would be drawn. Noti that the program draw the points every 10 iterations.

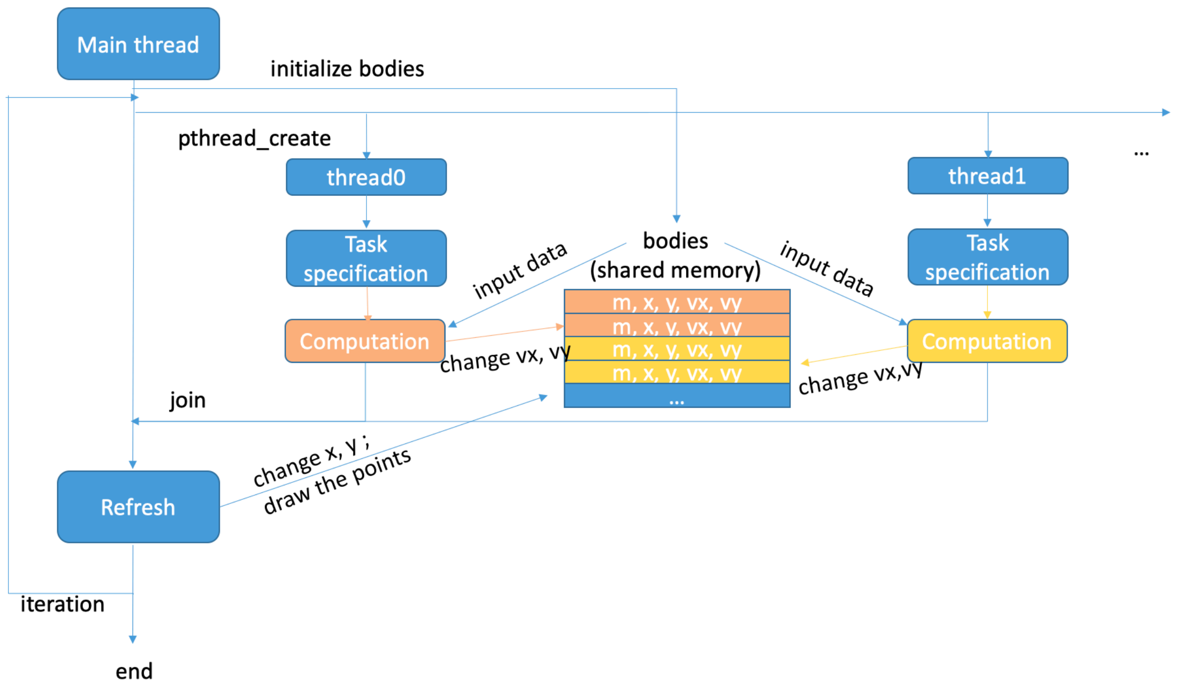
4. Repeat step 2-3 for a certain times (iteration).

```
int count = 0;
while (count < ITERATIONS){
    count += 1;
    // ...
    // code of step 2 and 3
    // ...
}
```

To keep the running of simulation, step 2 and 3 were iterated for a given times.

The above are the detailed implementation of the sequential version.

Pthread_version



The graph above shows the logic of pthread version. According to the sequential version, there are three main work are independent so that they could be parallelized. Observing that the work of initialization and refreshing only traverse all the bodies once, but the computation has a nested traversal, which is a $O(n^2)$ complexity. Therefore, in pthread version, the computation part was parallelized to each thread and the initialization and refreshing (refreshing was parallelized in the source code) were done in master thread.

In each thread, it would specify the target bodies according to its own thread id. The bodies were assigned to each thread as evenly as possible.

```
num = NUM_BODIES / NUM_THREADS;
remain = SIZE % NUM_THREADS;
if (tid < remain){
    num ++;
    start = num * tid;
}else{
    start = num * tid + remain;
}
end = start + num;
```

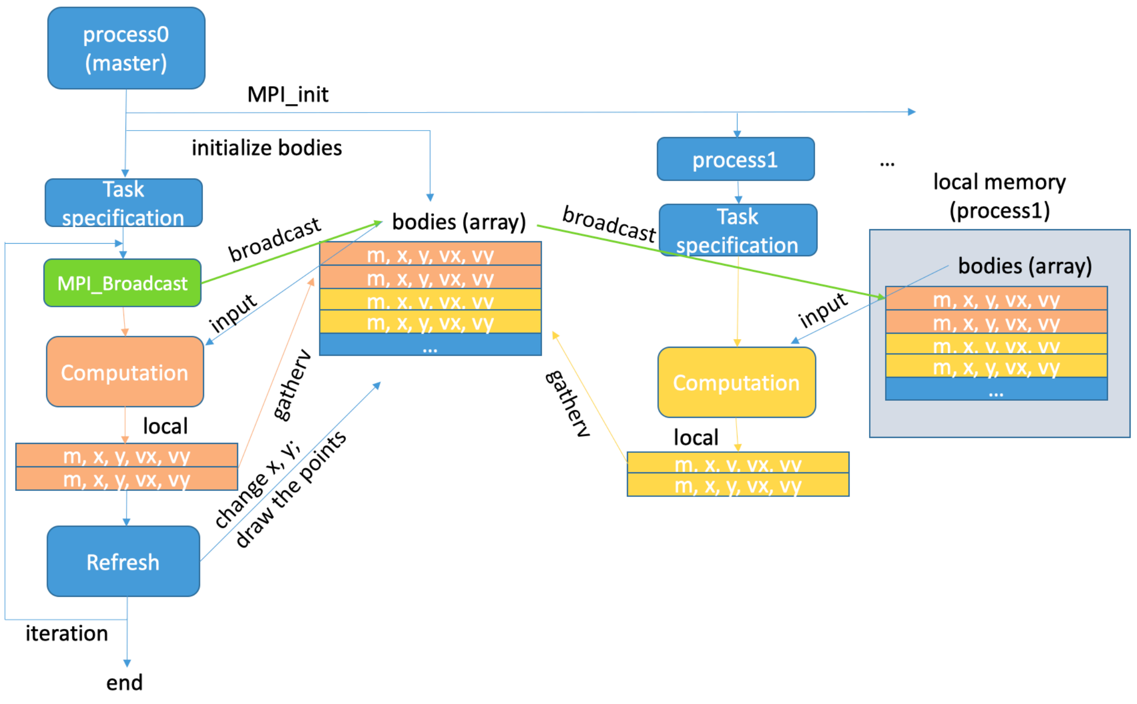
Then it directly access the data in shared memory to compute the acceleration and update the velocity of the target bodies in the shared memory. After that all the threads join to make sure all the data has been updated, and then refresh the screen same as what the sequential version did (In the source code, the refreshing work is also parallelized, which is a little bit different with the graph).

```
//  distribute the work of velocity computation
for (i = 0; i < NUM_THREADS; ++ i){
    input_data[i].count = count;
    rc = pthread_create(&thread[i], NULL, cal_func, &input_data[i]);
    if (rc) {
        fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
        return EXIT_FAILURE;
    }
}
// gather the threads
for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thread[i], &return_data);
}
```

(where cal_func() is the function of calculation).

The program would iterate for a given times, and pthreads would be created and joined once (In the source code, twice) in each iteration.

MPI version



The graph above shows the logic of MPI version.

In this version, the initialization was done in the master process. However, since it is hard to communicate with a structure in MPI, the data was represented by array here. Every 5 elements represent the data of one body - *mass, x, y, vx, vy* in order.

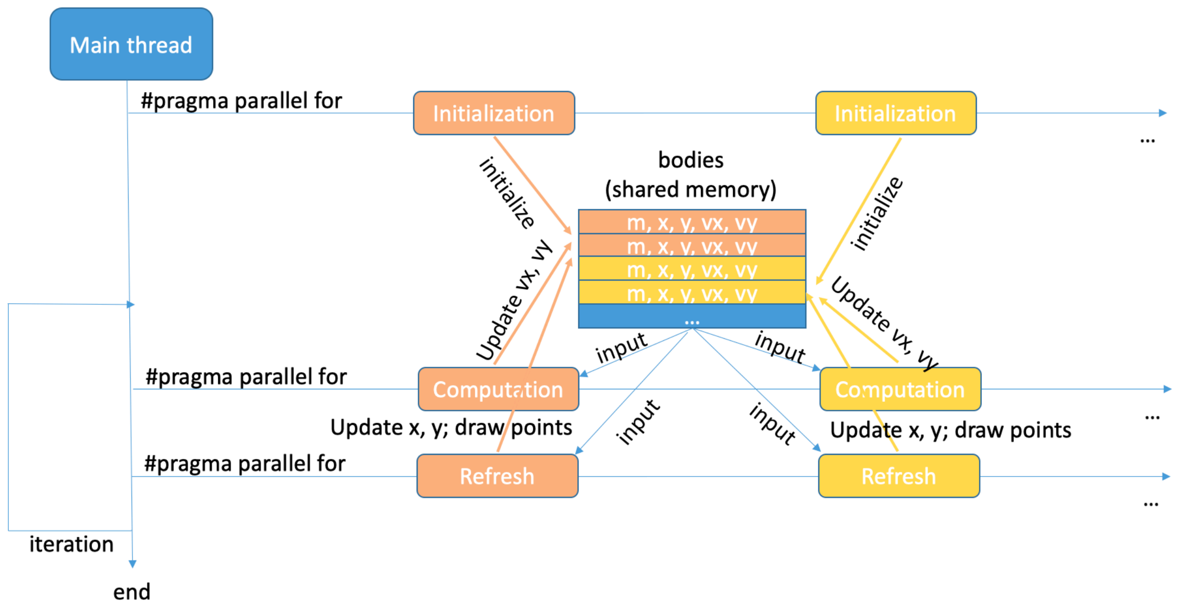
After the initialization, the master process will broadcast the whole array 'bodies' to each process. Each process would first specify the target bodies and create a local array for storing the data. The data will be copied from the delivered array 'bodies' first. And each process will do the computation based on the delivered array 'bodies' and then update the results to the local array. After that, the data of all process would be gathered to the master process and master process will do the refreshing sequentially.

```
// Broadcast the updated info of bodies to every node
MPI_Bcast(bodies, 5*NUM_BODIES, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
// Gather the results
MPI_Gatherv(local_data, count[taskid], MPI_DOUBLE, bodies, count, disp,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

The program would iterate for a given times, and there would be one broadcast and one gather in each iteration.

OpenMP version



According to the sequential version, for initialization, computation and refreshing, it was work by for loop. Thus, in OpenMP version, these for loop were simply parallelized, and all the operations would be done directly on the data stored in the shared memory.

```
#pragma omp parallel for private(dx, dy, r, v1, v2, a, ax, ay)
for (i = 0; i < NUM_BODIES; ++i){
    for (j = 0; j < NUM_BODIES; ++j){
        // ... (code for computation)
    }
}
```

MPI + OpenMP version

Based on the MPI version, the for loop of initialization in the master process and the for loop of computation in all processes were parallelized by OpenMP.

How to run

Sequential version

1. In the directory of source code, type the following command and enter to compile the program:

   ```
   g++ nbody_seq.cpp -lX11 -o ./nbody_seq.out
   ```

   then an executable file *nbody_seq.out* will be genereted in the directory.

2. In the same directory, type the following command and enter to run the program:

   ```
   ./nbody_seq.out (NUM_BODIES)
   ```

   where NUM_BODIES is a optional parameter that could specify the number of bodies in the simulation.

   Default setting: NUM_BODIES = 200.

Pthread version

1. In the directory of source code, type the following command and enter to compile the program:

   ```
   g++ nbody_pthread.cpp -lX11 -lpthread -o ./nbody_pthread.out
   ```

   then an executable file *nbody_pthread.out* will be genereted in the directory.

2. In the same directory, type the following command and enter to run the program:

   ```
   ./nbody_pthread.out (NUM_BODIES) (NUM_THREADS)
   ```

   where NUM_BODIES and NUM_THREADS are optional parameters that could specify the number of bodies in the simulation and the number of threads used for simulation.

   (NOTE: to specify the NUM_THREADS, the NUM_BODIES should be specify first)

   Default setting: NUM_BODIES = 200, NUM_THREADS = 2.

MPI version

1. In the directory of source code, type the following command and enter to compile the program:

   ```
   mpic++ nbody_mpi.cpp -lX11 -o ./nbody_mpi.out
   ```

   then an executable file *nbody_mpi.out* will be genereted in the directory.

2. In the same directory, type the following command and enter to run the program:

```
mpirun -np N ./nbody_mpi.out (NUM_BODIES)
```

where N specifies the number of processes and NUM_BODIES is optional to specify the number of bodies in the simulation.

Default setting: NUM_BODIES = 200.

OpenMP version

1. In the directory of source code, type the following command and enter to compile the program:

   ```
   g++ nbody_openmp.cpp -fopenmp -lX11 -o ./nbody_openmp.out
   ```

   then an executable file *nbody_pthread.out* will be genereted in the directory.
2. In the same directory, type the following command and enter to run the program:

   ```
   ./nbody_openmp.out (NUM_BODIES) (NUM_THREADS)
   ```

   where NUM_BOcDIES and NUM_THREADS are optional parameters that could specify the number of bodies in the simulation and the number of threads used for simulation.

   (NOTE: to specify the NUM_THREADS, the NUM_BODIES should be specify first)

   Default setting: NUM_BODIES = 200, NUM_THREADS = 2.

MPI + OpenMP version

1. In the directory of source code, type the following command and enter to compile the program:

   ```
   mpic++ nbody_mpi+openmp.cpp -fopenmp -lX11 -o ./nbody_mpi+openmp.out
   ```

   then an executable file *nbody_mpi.out* will be genereted in the directory.
2. In the same directory, type the following command and enter to run the program:

   ```
   mpirun -np N ./nbody_mpi+openmp.out (NUM_BODIES) (NUM_THREADS)
   ```

   where N specifies the number of processes, NUM_BODIES and NUM_THREADS are optional to specify the number of bodies in the simulation and the number of threads used for simulation.

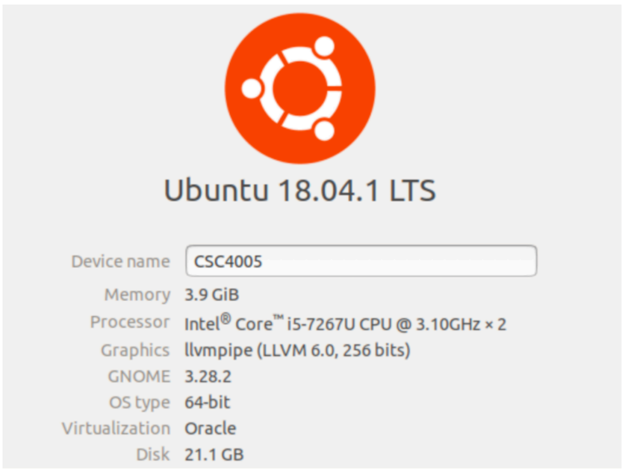
   (NOTE: to specify the NUM_THREADS, the NUM_BODIES should be specify first)

   Default setting: NUM_BODIES = 200, NUM_THREADS = 2.

Sample output

Running environment

The sample is running on the virtual machine (with *Xlib* and *mpich-3.2.1* installed):



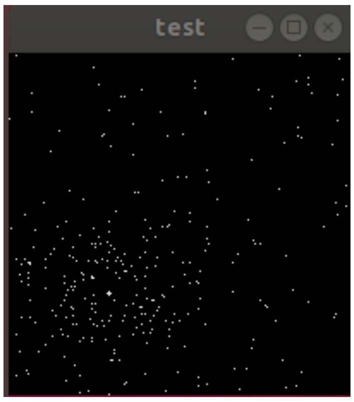
Ubuntu 18.04.1 LTS

Device name	CSC4005
Memory	3.9 GiB
Processor	Intel® Core™ i5-7267U CPU @ 3.10GHz × 2
Graphics	llvmpipe (LLVM 6.0, 256 bits)
GNOME	3.28.2
OS type	64-bit
Virtualization	Oracle
Disk	21.1 GB

Sample display

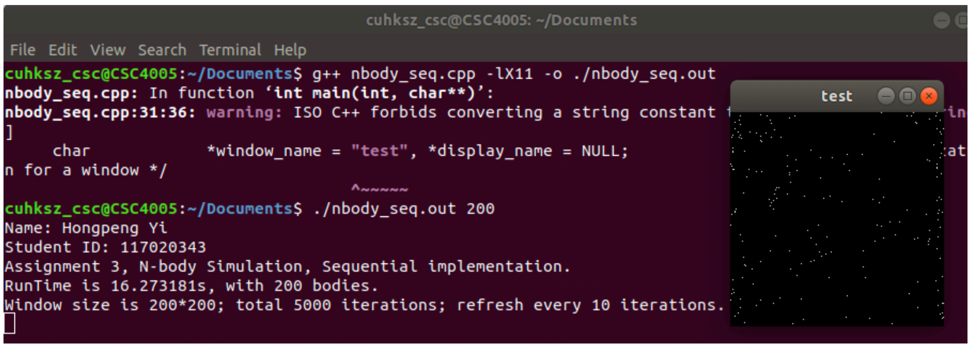
Since the simulation is a dynamic process, a simple screen shot could show the movement of bodies. There is a video attached to this report that could see the sample movement.

Most of time, the graph would be as follows. Most of bodies gather around the body with large mass and meanwhile process the orbital motion. This phenomenon becomes obvious when the large-mass body approaches the boarder. However, when the large-mass body hit the boarder, other bodies would suddenly spread out since the large-mass body lose its velocity and there is no space for other bodies to do orbital motions. It may take some time for bodies gather together again.



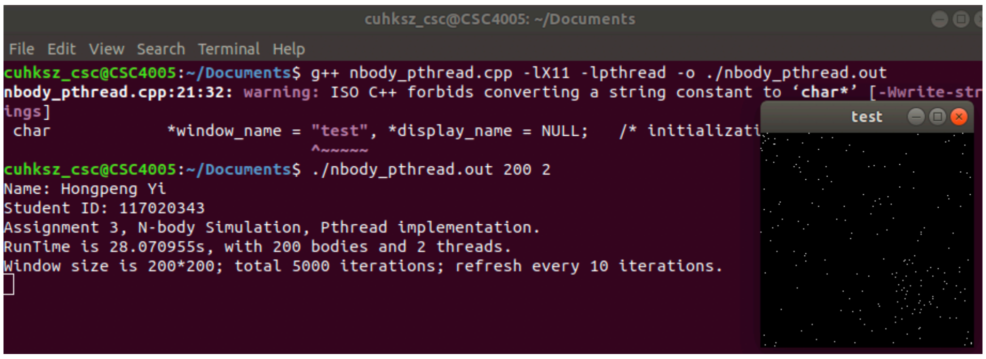
It should be noted that, the bodies move fast and the window is small, so that sometimes the body movement might be not obvious. In case of this, try a few more times or increase the window size.

Sequential



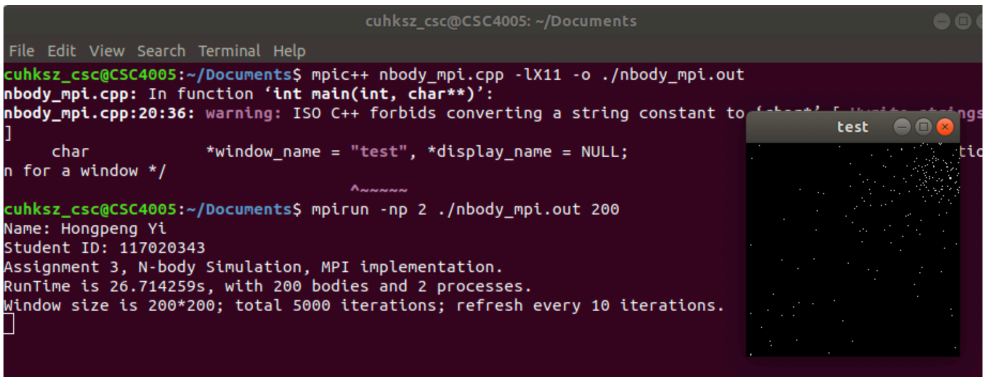
```
cuhksz_csc@CSC4005:~/Documents$ g++ nbody_seq.cpp -lX11 -o ./nbody_seq.out
nbody_seq.cpp: In function ‘int main(int, char**)’:
nbody_seq.cpp:31:36: warning: ISO C++ forbids converting a string constant t
]
     char             *window_name = "test", *display_name = NULL;
n for a window */
                                   ^~~~~~
cuhksz_csc@CSC4005:~/Documents$ ./nbody_seq.out 200
Name: Hongpeng Yi
Student ID: 117020343
Assignment 3, N-body Simulation, Sequential implementation.
RunTime is 16.273181s, with 200 bodies.
Window size is 200*200; total 5000 iterations; refresh every 10 iterations.
```

Pthread version


cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
test

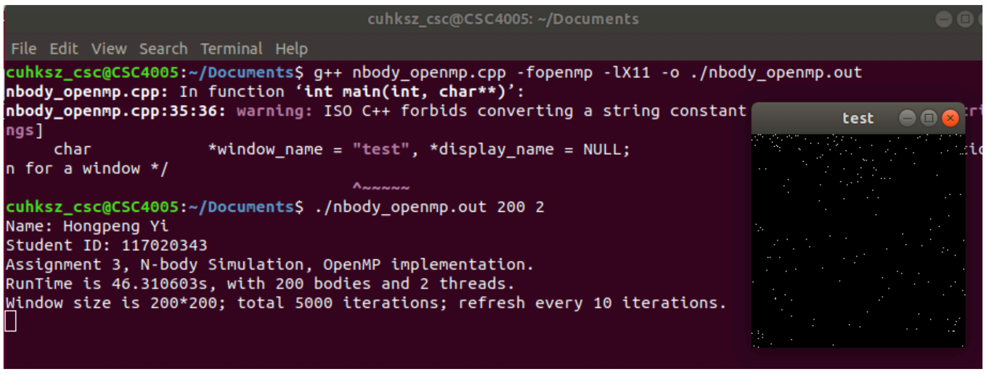
```
cuhksz_csc@CSC4005:~/Documents$ g++ nbody_pthread.cpp -lX11 -lpthread -o ./nbody_pthread.out
nbody_pthread.cpp:21:32: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-str
ings]
 char            *window_name = "test", *display_name = NULL;    /* initializati
                                ^~~~~~
cuhksz_csc@CSC4005:~/Documents$ ./nbody_pthread.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 3, N-body Simulation, Pthread implementation.
RunTime is 28.070955s, with 200 bodies and 2 threads.
Window size is 200*200; total 5000 iterations; refresh every 10 iterations.
```

MPI version


cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
test

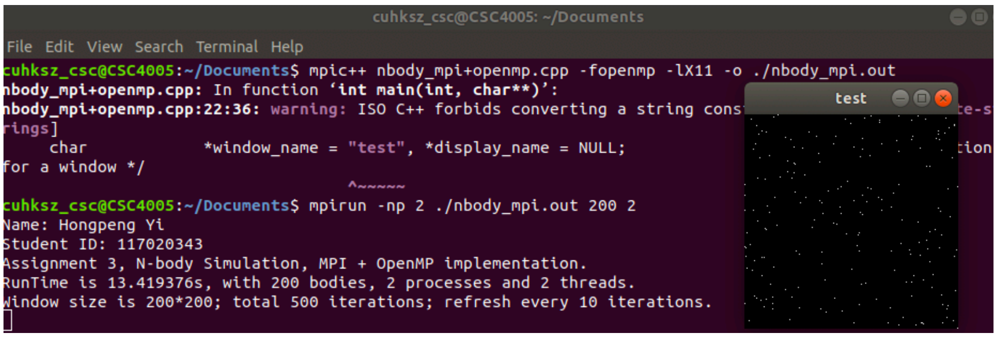
```
cuhksz_csc@CSC4005:~/Documents$ mpic++ nbody_mpi.cpp -lX11 -o ./nbody_mpi.out
nbody_mpi.cpp: In function ‘int main(int, char**)’:
nbody_mpi.cpp:20:36: warning: ISO C++ forbids converting a string constant to ‘char*’ [-Wwrite-strings
]
     char            *window_name = "test", *display_name = NULL;               tio
n for a window */
                                    ^~~~~~
cuhksz_csc@CSC4005:~/Documents$ mpirun -np 2 ./nbody_mpi.out 200
Name: Hongpeng Yi
Student ID: 117020343
Assignment 3, N-body Simulation, MPI implementation.
RunTime is 26.714259s, with 200 bodies and 2 processes.
Window size is 200*200; total 5000 iterations; refresh every 10 iterations.
```

OpenMP version


cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
test

```
cuhksz_csc@CSC4005:~/Documents$ g++ nbody_openmp.cpp -fopenmp -lX11 -o ./nbody_openmp.out
nbody_openmp.cpp: In function ‘int main(int, char**)’:
nbody_openmp.cpp:35:36: warning: ISO C++ forbids converting a string constant                 ri
ngs]
     char            *window_name = "test", *display_name = NULL;                 io
n for a window */
                                    ^~~~~~
cuhksz_csc@CSC4005:~/Documents$ ./nbody_openmp.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 3, N-body Simulation, OpenMP implementation.
RunTime is 46.310603s, with 200 bodies and 2 threads.
Window size is 200*200; total 5000 iterations; refresh every 10 iterations.
```

MPI + OpenMP version

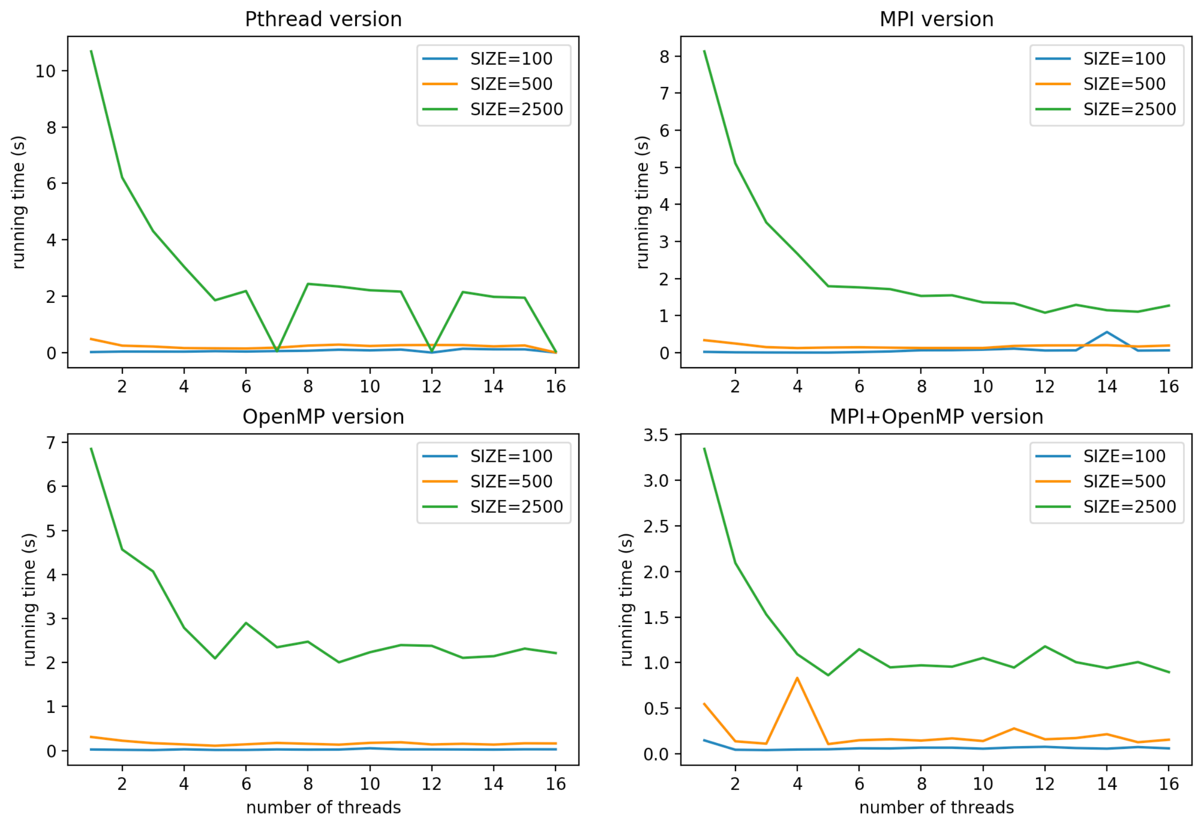

cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
test

```
cuhksz_csc@CSC4005:~/Documents$ mpic++ nbody_mpi+openmp.cpp -fopenmp -lX11 -o ./nbody_mpi.out
nbody_mpi+openmp.cpp: In function ‘int main(int, char**)’:
nbody_mpi+openmp.cpp:22:36: warning: ISO C++ forbids converting a string cons                ce-s
rings]
     char            *window_name = "test", *display_name = NULL;                 tion
for a window */
                                    ^~~~~~
cuhksz_csc@CSC4005:~/Documents$ mpirun -np 2 ./nbody_mpi.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 3, N-body Simulation, MPI + OpenMP implementation.
RunTime is 13.419376s, with 200 bodies, 2 processes and 2 threads.
Window size is 200*200; total 500 iterations; refresh every 10 iterations.
```

Performance analysis

Performance on parallelization

To analyze the performance of the parallel versions of programs, the programs are run on a cluster with three different number of bodies - 100, 500, 2500 and with 1 - 16 processes/threads respectively. The window size has been fixed as 200×200 and the the number of iterations has been fixed as 50. For MPI+OpenMP version, the number of MPI processes was fixed as 4. Totally $4 \times 3 \times 16 = 192$ data points were generated and plotted as the following graph：

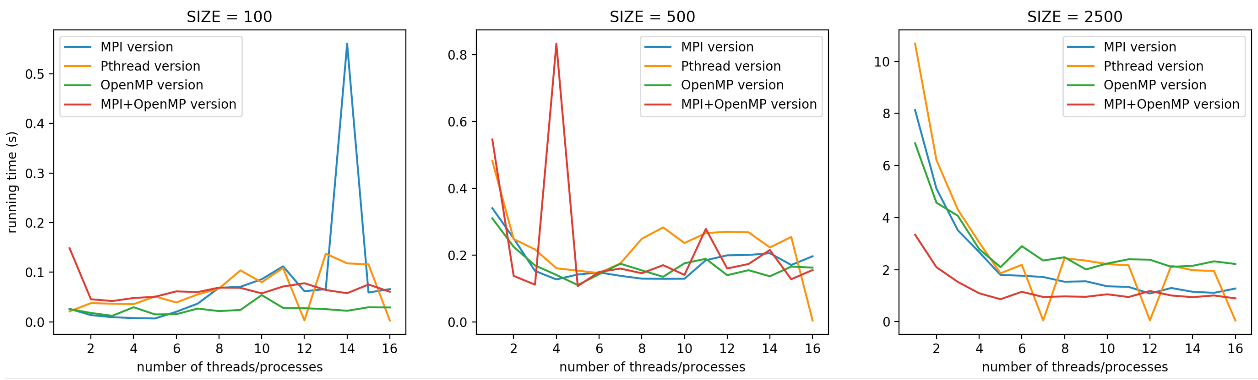


The graphs above show the running time against the number of threads/processes with 3 different data size for each parallel version, which could give an insight of how well the parallelization performed with different processes/threads. Observing the graphs, we could find that:

- For all parallel versions, with large data size (2500), the running time is roughly decreasing as the number of threads/processes increases. This does make sense that multithreads or multiprocesses divided the task so that increase the running speed.
- For all parallel versions, as the number of threads/processes increases, the running time decreases fast when the number of threads/processes is smaller than or equal to 5 but remains at a certain level when the number of threads/processes is larger than 5. The performance is improved obviously when the program is parallelized with no more than 5 threads/processes. The reason for this might be that the first virtual machine was assigned with 5 physical cores so that the communication cost affects the running time when the number of threads/processes is larger than 5.
- Comparing the performance of different data size, the larger data size causeS the longer running time. The running time increases faster than data size. This shows that the time complexity of this algorithm is larger than $O(n)$. In fact, according to the program design,

the complexity is probably $O(n^2)$ since the computation part in the simulation has a nested loop.

Pthread vs MPI vs OpenMP vs MPI+OpenMP



The graphs above show the running time against the number of threads/processes with 3 different data size for each parallel version, which could intuitively compare the performance of different parallel versions. Observing the graphs, we could find that:

- When the data size is small , the curves fluctuate a lot and the running time of different versions are about the same level. This might be explained as the communication cost, which means the running time is dominated by communication time when the data size is small. The communication time could be flatten with large running time, as the other two graphs show.
- When the data size is large, the improvement on the performance is obvious. Comparing the four curves, we could find that the performances of Pthread version, MPI version and OpenMP version are similar but the performance of MPI+OpenMP version is significantly better than other versions.

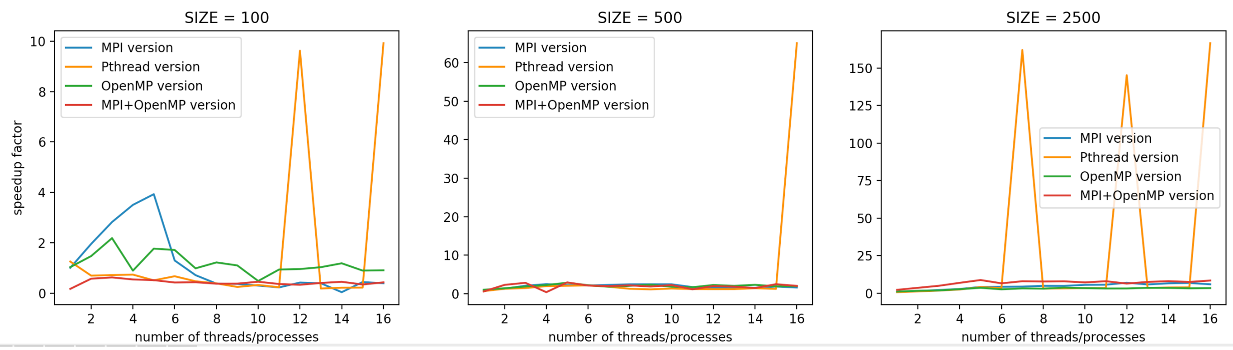
Sequential vs Parallel

In the comparison between parallel version and sequential version, *speedup factor* could give a clear insight about how the parallel computation performs, which measures the increase in speed by using multiprocessors.

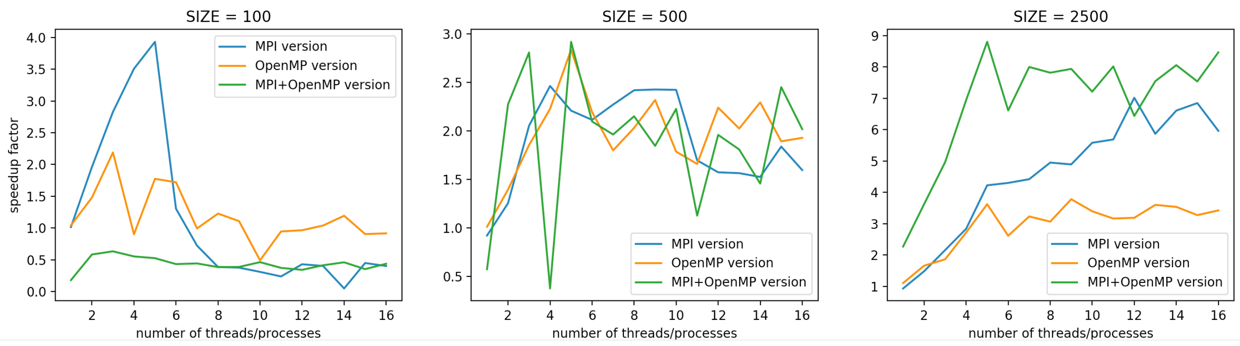
The ***speedup factor*** is given as

$$S(n) = \frac{t_s}{t_p}$$

, where t_s is the execution time on a single processor (sequential) and t_p is the execution time on a multiprocessor. Therefore, the speedup factors for different numbers of threads/processes with different data size could be computed, and they were plotted as follows:



From the graph above, we found that there are some abnormal points in the Pthread version, which have an extremely high speed up and mean while scale up the whole graph. To observe the speedup performance of other versions, the curve of Pthread version was removed:



According to the speedup graph above, we could find that:

- When the data size was small, the speedup for MPI and OpenMP was larger than 1 with small number of threads/processes but became small and even less than 1 when the number of threads/processes became larger. The MPI+OpenMP version had a bad performance on speed up when the data size was small since its speed was even less than the sequential version.
- When the data size was medium, all the three versions performed well on the speedup factor around a certain level.
- When the data size was large, there was a significant speedup for each parallel version and the speedup increased with the increasing number of threads/processes, especially when it is smaller than 5. Moreover, for the performance on speedup, the MPI+OpenMP version is better than the MPI version and the MPI version is better than the OpenMP version.

Conclusion

In this experiment, five implementations of N-body Simulation - Sequential version, MPI version, Pthread version, OpenMP version and MPI+OpenMP version have been successfully designed and test. The performance analysis on these five programs have been presented. The parallel computation does a good job on speedup when the data size is large, especially for MPI+OpenMP version. The combination of MPI and OpenMP is supposed to be a good choice for parallel programming on a cluster.