CS3211 Project 2 Report

Xu Yinan, A0179919E

April 21, 2018

1 Introduction

In this project, we developed a simulation of 2D version of the pool table in one process, and then an Open MPI parallelized version. The program is deployed and evaluated on Tembusu machines.

For the pool table, all the small and large particles are intially stationary with their locations specified randomly and by the specification file respectively. To emphasize parallelism, we use the pass-through Frobisher effect first. The simulation runs for about 100 steps and the board is recorded using a `ppm` file.

The report includes discussions on the implementation of the simulation along with the performance analysis of the program and comparison between the sequential and the parallelized version.

2 Implementation

2.1 Definition of the Problem

The pool table is a square region containing small and large particles that continuously accelerate towards each other according to the gravitational effect. The mass and radius of particles are determined by the specification file. The positions of the large particles are specfied by the file, whereas the small particles are randomly located.

The force between particle i, j is given by

$$\vec{F} = G\frac{m_i m_j}{r^2}\frac{\vec{r}}{r}$$

where G is the gravitational constant, m_i, m_j are the mass of particle i, j respectively and $\vec{r}$ is the relative distance of the two particles.

Considering the emphasis on parallelism and my free time, we assume the system follows the pass-through effect.

Thus, the acceleration particle j puts on particle i is given by

$$\vec{a_{ji}} = G\frac{m_j}{r^2}\frac{\vec{r}}{r}, \tag{1}$$

the corresponding velocity change is

$$\vec{v_t} = \vec{v_0} + \vec{a_{ji}}\Delta t, \tag{2}$$

and the displacement is

$$\vec{x} = \vec{v_0}\Delta t + \frac{1}{2}\vec{a_{ji}}\Delta t^2 = \vec{v_t}\Delta t - \frac{1}{2}\vec{a_{ji}}\Delta t^2. \tag{3}$$

2.2 Helper Functions

Helper functions are defined in four header files, including `logging.h`, `pfile.h`, `spec.h` and `mympi.h`.

`logging.h` is a logger tool written by Gerard Lledó Vives and originally included in a FUSE-based ext4 file system implementation[1]. Since it's only for debug use and does not work at all in the release version of the program, we just ignore its details.

`pfile.h` defines the structure of the specification file and `ppm` file. It also contains some helper functions for transformations between the in-memory data structures and on-disk files. Moreover, it contains some extra definitions for the in-memory data strutures that are needed by Open MPI to register a self-defined `MPI_Datatype`.

`spec.h` defines the basic in-memory data structures for particles and the computation specification like `time_slop`.

`mympi.h` is simply a wrapper for the Open MPI library that extends the original library

with some more self-defined `MPI_Datatype`. By calling `init_mympi()`, the wrapper will register the user-defined structures and then we can use these datatypes in MPI directly.

2.3 Program Design

Main program is defined in `pool.c` with the corresponding header file `pool.h`.

The simulation program starts with parsing the command line arguments and checking whether they're legal. After initializing the parameters for the simulation, we initialize the pool with randomly located small particles and specified large particles with zero velocity. In every step, we directly compute the acceleration of a particle that is imposed by particles in the adjacent regions. To make movements of the particles more visible, we use $G = 6.67$. We treat the acceleration as a vector and add all the vectors into one vector following the rules of vector addition. We calculate the updated velocity and then the displacement, which finishes one step of the simulation. Finally, we collect all the regions and embed them into one `PPMFile` structure for writing onto disk.

In the design, we have considered some rearrangements to the code that may affect the performance of the program, as describe in Section 3.

2.4 Usage

We have provided a `Makefile` for compilation and several scripts for running the program.

To compile the sequential and parallelized version of the program, use `make` and `make seq` respectively. To compile it in debug mode, use `make debug` and `make seqdebug` respectively instead. Also, use `make bmp` to convert `ppm` files in the `ppmresults` folder to the visualizable `bmp` files in `bmpresults` folder. Use `make profile` to compile the program with profiling support. Use `make clean` to remove all the output.

To run the program, use `./run.sh <np>` and `./seqrun.sh <np>` for parallelized and sequential version of the program respectively. Use `./profilerun.sh <np>` to run the parallelized profiled program.

If one would like to run the program manually, the usage of the program given by

```
Usage: pool <specfile> <ppmfile> [np]

  <specfile>  path to specification file
  <ppmfile>   output path to ppm file
              (without file extension)
  [np]        number of processes or regions.
              Only for the sequential verison
              and should be a perfect square.
```

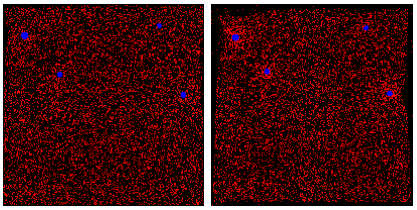
in both versions.

The program would print the execution time of the simulation as a real number to `stdout`.

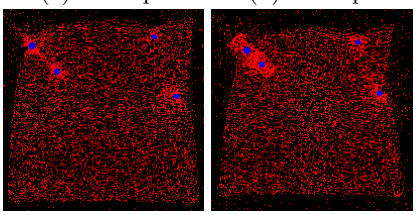
We also provide a Python script for automatically running the program and writing the execution time into a `txt` file.

2.5 Results

Use the default specification file in Figure 14, we have the results as shown in Figure 1.



(a) 25 steps (b) 50 steps



(c) 75 steps (d) 100 steps

Figure 1: Output pool tables with the modified gravatational effect with $G = 6.67$

3 Discussion

We've adopted some techniques to optimize the program's performance. All the techniques mentioned below are for the Open MPI parallelized version unless otherwise noted.

In addition to the approaches discussed in this section, there's one more optimization technique discussed in Section 4.4 regarding the in-memory data structures, which is not used in the submitted version but for test usage.

3.1 In-memory Data Structures

All the data structures used in the simulation are well designed. Though C is not an object-oriented programming language, we can still pack the data into `struct` and should benefit a lot from abstraction when building such a complex system with many various datatypes.

Some information in the specification file is transferred across processes only once during the simulation like `time_slot`, `time_step` and `horizon`, so we pack them into one structure named `CompSpec` and broadcast them to all processes in the beginning.

For the `Pool` structure, it should contain the information on different number of particles maybe with different mass. To dynamically allocate space for these particles, the only choice is to declare pointers in the structure and do allocation at runtime using `malloc` or `calloc`.

However, it brings a problem in communication. Even though pointers can be transferred using MPI by explicitly using the `uint64_t` (or `uint32_t` in a 32-bit machine) datatype, it's of no meaning since different MPI processes cannot read each other's memory at all.

Therefore, to transfer the information of the pool quickly and avoid futile work, we define the struture as

```
typedef struct {
    /* size of the pool table       */
    uint32_t  size;
    /* number of small particles    */
    uint32_t  small_num;
    /* mass of small particles      */
    Mass      small_mass;
    /* radius of small particles    */
    Radius    small_rad;
    /* number of large particles    */
    uint32_t  large_num;
    /* locations of small particles */
    Location *small_ptc;
    /* info about large particles,
       including mass, radius,
       positions., position.y       */
    Particle *large_ptc;
} Pool;
```

where: (1) non-pointer data are located continuously (and 64-bit aligned without empty padding) from the beginning of the structure. For transmissions, we only send and receive the first five fields of the structure to avoid the useless pointers. (2) data abstraction is preserved. One pool table still lies in one structure, where the particles can be easily located and its number be properly recorded. (3) transmissions of particle information are quite easy. By the first five fields, we can determine the length of the particle array. We then simply access to `small_ptc` and `large_ptc`, and use MPI communication primitives for transmissions.

Most large data structures that may differ as the problem scales are dynamically allocated at runtime and duly freed. This would reduce the used memory and fully optimize the program's runtime performance.

3.2 Use Open MPI Asynchronous Communication Primitives

MPI has defined two types of communication primitives, that is, the synchronous and the asynchronous primitives. Asynchronous functions are non-blocking, which allows threads to continue doing computations while communication with another thread is still pending[2].

For better performance, we use the asynchronous primitives for transmissions in most cases. Every time after calling asynchronous functions, we do something else until we have to use the data that has been sent or waited to be received.

For example, when computing the acceleration, we need to know the positions of all the particles in the adjacent regions, which is about $10000 \times 2 \times 8$ Bytes $= 156.25$ KB for each adjacent region. This should take a lot of time for transmission. Therefore, after calling `mympi_isend`, we first compute the impacts of the particles in the same region. After that we compute impacts of each region one by one, and only before we compute region i's impact do we call `mympi_wait` to ensure the complete of transmission and availability of the data.

3.3 Simplify Calculation

The adjacent regions can be determined when `rank` and `size` are known. Therefore, we use an array `adj_rank` to store the rank of adjacent regions and don't compute them in every step to reduce computation complexity.

Noticing that small particles in the pool are equally weighted, we know that the force between two small particles are the same except for their reverse orientation. Therefore, when computing the force between two small particles in one region, we update their acceleration simultaneoulsly, as shown below.

```
for (int i = 0; i < small_num; i++) {
  for (int j = i + 1; j < small_num; j++) {
    // compute ptc[j]'s impact on ptc[i]
    Location r_ji = {
      small_ptc[j].x - small_ptc[i].x,
      small_ptc[j].y - small_ptc[i].y
    };
    COMPUTE_ACCEL_2PTC_RAW(r_ji, small_mass,
      small_mass, small_acc[i], small_acc[j]);
  }
}
```

Besides, since all the acceleration computation includes multiplying the gravatational constant G, we can avoid duplicate multiplications by applying G in the end to the overall acceleration value. Note that previously we only use the `COMPUTE_ACCEL_2PTC_RAW` macro which does not apply the G constant, now we use another macro `ACCEL_RAW_TO_FINAL` that converts the raw acceleration (the value without applying G constant) to the final value.

Another simplification regarding calculation is discussed in Section 3.5.

3.4 Minimize Transmissions

As mentioned in Section 3.2, the data transferred between processes is more than 150KB for every adjacent region in every round of simulation, which makes the communication time really significant.

To reduce the size of data transferred between processes, we minimally reduce it to an array of `Location` and an array of `Particle` for small and large particles respectively. `Location` and `Particle` are defined as shown below.

```
typedef double Radius;
typedef double Mass;

typedef struct {
  double x;       // x-coordinate
  double y;       // y-coordinate
} Location;

typedef struct {
  Radius   rad;   // radius
  Mass     mass;  // mass
  Location loc;   // location
} Particle;
```

Since the mass and radius of small particles are the same in all the regions, we transfer `Location` only for small particles. For large particles, we transfer `Particle`.

Moreover, after the computation finishes, it's time for gathering the results and writing the `ppm` file into disk. We convert the pool table in every process into a `PPMFile` and then transfer it to the root process, instead of transferring the whole pool board.

3.5 Problem Simplification

Originally in the pass-through version of simulation, the computation itself is not much complicated, except for computing the square root. However, this computation will be done for millions of times. Therefore, reducing the number of multiplications and divisions in every computation unit would contribute a lot to optimizations.

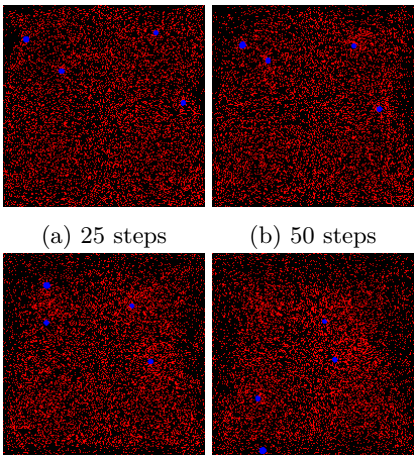
We can optimize the computation by using

$$\vec{F} = G\frac{m_i m_j}{r}\frac{\vec{r}}{r} = G\frac{m_i m_j}{|r|^2}\vec{r},$$

which does not change the fundamental property of gravatational effect - the force increases rapidly when the two particles move towards each other and become much close. With this optimization, for two particles, the calculation contains only four multiplications and two divisions (further optimized to three multiplication and one division if both of them are small particles).

To see how much it would influence the results, I've tried both versions of gravatational force. The original gravatational effect produces the results as shown in Figure 2. Pre-

viously we have shown the results of current optimized program in Figure 1.



(a) 25 steps (b) 50 steps

(c) 75 steps (d) 100 steps

Figure 2: Pool tables adopting the original gravatational effect except for $G = 667$) during the simulation.

As we can see, small particles in the modified version would gather more quickly and be affected by the large particles more remarkably than those in the original version. Large particles move more in the original version since G has been set to 667.

Simulation under the default specification in Figure 14 shows that if we adopt the original gravatational force (proportional to squared distance), the execution time is 35.0371 seconds whereas the optimized version uses only 18.6491 seconds. Thus, the call for `sqrt` does cost a lot.

4 Analysis

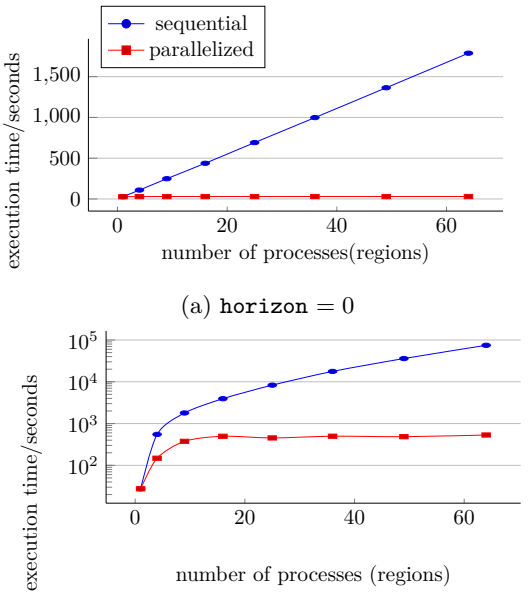
In this section we discuss about the performance of the program. Raw data is listed in Section 5.2.

4.1 Execution Time

We run the program on Tembusu machines, using different number of processes. Note that there are not much differences between large and small particles, so we only change the number of small particles in the comparison, though large particles may cause longer communication time.

4.2 Sequentiality v.s. Parallelism

To compare the sequential version and parallelized version, we use data from Table 1, 2, 5 and plot them as shown in Figure 3.



(a) `horizon` $= 0$

(b) `horizon` $= 1$

Figure 3: Execution time of the simulation

Note that in the second graph we have set the mode of y axis to be `log`. We can conclude that the execution time of the sequential program will strictly increase as the number of processes (also the number of region being simulated) grows up, whereas in the parallelized version the execution time will become nearly a constant. For example, when `horizon` $= 0$, execution time of the sequential program is somewhat proportional to the number of processes, since the time complxity is simply a linear function of the number of regions.

Obviously the MPI parallelized program is much much faster than the sequential one, and the speedup (by Amdahl's law) is as shown in Figure 4.

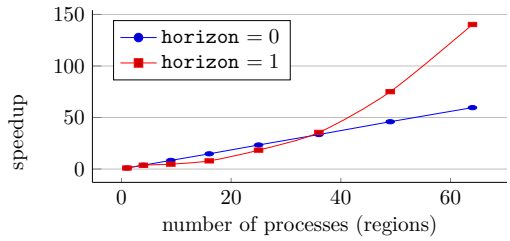
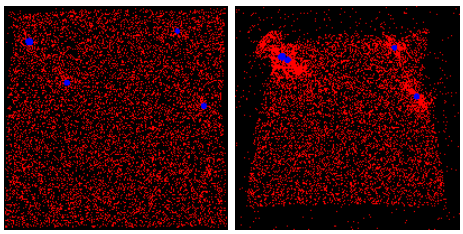


Figure 4: Speedup of the parallelized program compared to the sequential one



(a) 100 steps (b) 300 steps

Figure 5: Sequential simulation results

It can be drawn from the graph that the more processes (regions) we use, the more speedup we could get from changing the sequential program into parallel. Also when there're many enough regions, the more adjacent regions we consider, the more speedup we will achieve by parallelism.

Since there's no closed form of execution time with respect to number of processes, it becomes a little difficult to compute the speedup by Gustafson's law. But if we draw horizontal lines on Figure 3b, then the ratio of the x-coordinates of the two intersection points is the speedup. From the graph, we can see it's obvious that the speedup would increase as the number of processes (regions) increases (and is greater than the speedup by Amdahl's law).

Moreover, there're also accuracy problems with the sequential and parallelized simulation.

Figure 2 shows the result after 100 steps when paralleling the program. However, in Figure 5, the pool table after 100 steps of simulation does not look like the corresponding parallelized one. Only after 300 steps, the sequential result has approximately become the same as parallelized result in 100 steps.

Both the sequential and parallelized results were generated using the same paramaters as shown in Figure 14. I've not figured out the reasons why there's such a large deviation since in both of two versions we use the same data structures and macros for computing the acceleration, velocity and displacement. The weird thing is that the computation has never been parallelized. The computation for one particle has always been done in one process in the same order, since for computating the adjacent regions' impact, we use `for` loops and compute in the same order in both versions. Therefore, there shouldn't be any difference in the results.

The only reason I came up with is the randomness in generating the small particles. But I don't think the random number generator will always produce different distributions for 20000 numbers exactly in the two programs.

One of my friend also came to the same mysterious accuracy problem, so I think it should be something that we've missed. I will think about it in the days ahead.

4.3 When the Problem Scales

In this section we focus on the performance of MPI parallelized program when size of the program scales.

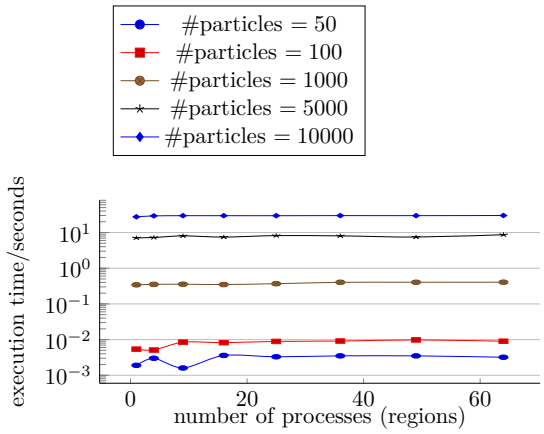
If we fix `horizon`, number of particles and then change the number of processes, theoretically the execution time will increase as the problem size grows up. Results show that the increase is quite smooth (Figure 6).

When `horizon` $= 0$, it even looks like a constant time. For `horizon` > 0, although the execution time increases, its growth rate is slower than that of processes.

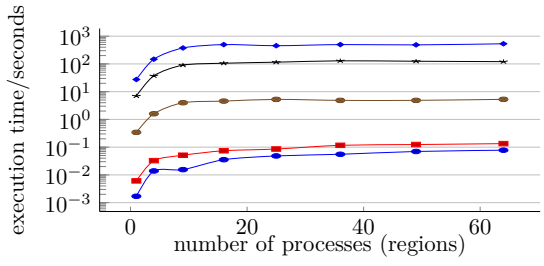
Thus, for a fixed problem complexity, the more processors we have, the more jobs we can do in approximately the same time. That is also what Gustafson's law emphasizes on.

If we fix the number of processes and keep adding particles, let's see what happens.

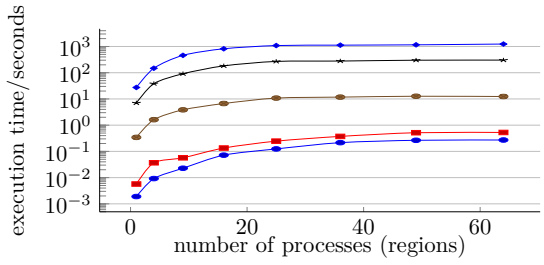
As shown in Figure 7, execution time increases at a speed that is mostly determined by `horizon` and barely influenced by the number of processes. The exection time tends to



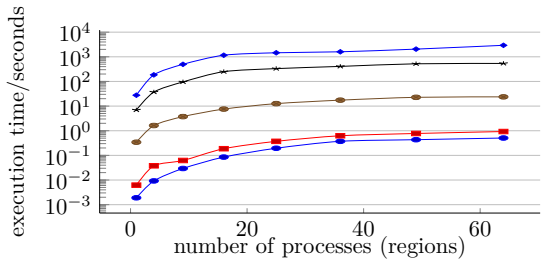
(a) `horizon` = 0



(b) `horizon` = 1

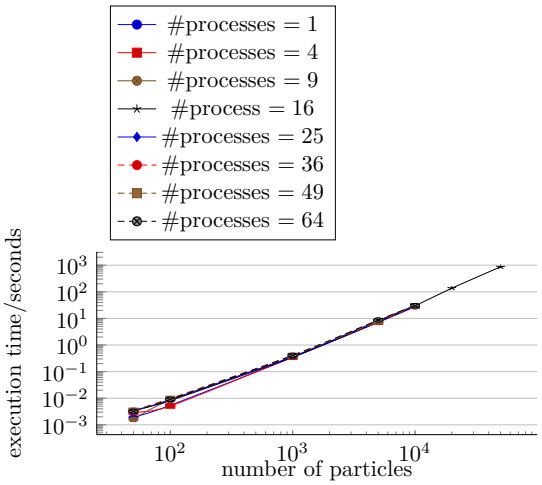


(c) `horizon` = 2

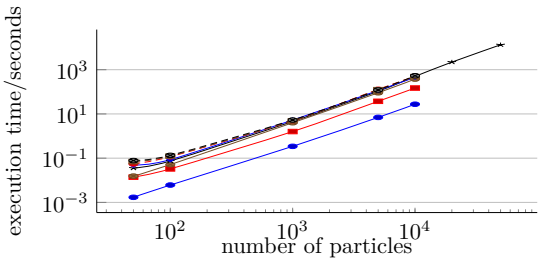


(d) `horizon` = 3

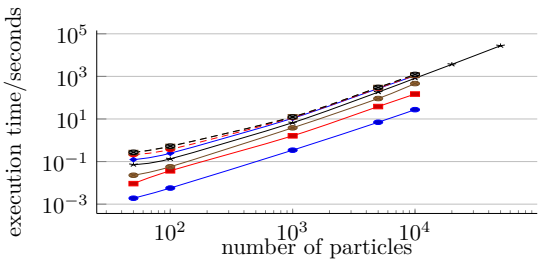
Figure 6: Execution time of the parallelized program when the number of processes changes



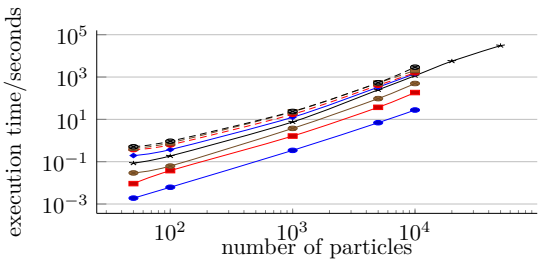
(a) `horizon` = 0



(b) `horizon` = 1



(c) `horizon` = 2



(d) `horizon` = 3

Figure 7: Execution time of the parallelized program when the number of particles changes

have a polynomial relationship $\Theta(n^k), k \approx 1.5$ with the number of particles, which coincides with the theoratical results $\Theta(n)$.

We can also see from the results that when the number of particles is greater than 50000, execution time may be more than 30000 seconds, which is approximately 8.3 hours.

Now let's fix the number of particles and see how `horizon` influences the execution (Figure 8).

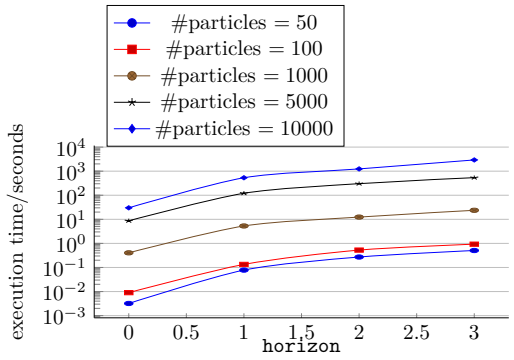


Figure 8: Execution time of the parallelized program when using 64 processes

Theoretically, execution time should be linear with squared `horizon` since the number of adjacent regions, impacts of which are considered, is $(2 * \texttt{horizon} + 1)^2 \in O(\texttt{horizon}^2)$.

Let ratio $= \frac{\text{execution time}}{(2*\texttt{horizon}+1)^2}$ and we plot the results in Figure 9. As we can see, the ratio tends to be a constant, which satisfies our theoretical analysis.

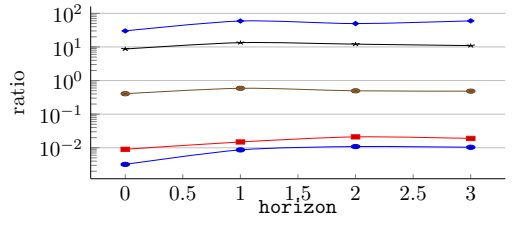


Figure 9: The relationship between execution time and `horizon` (legends same as Figure 8)

4.4 Communication Time

As we know, when the number of processes grows up, though computation may be accelerated (or more jobs can be done in the same time), the communication time also increases. This part of time may have great impacts on the performance. We use Vampir[3] to profile the MPI program and analyze its inter-process communication issues. Due to copyright reasons, though we've recorded the information with 1 to 64 processes, we would only discuss on no more than 16 processes (where communication costs should have less impacts than computation).

Let's first see how the accumulated exclusive communication time and the percentage of it in the overall time change with number of processes (the blue line in Figure 12 and Figure 13). When there're 16 processes, communication time takes even more than 45%. Thus, the communication time really matters.

What can we do to reduce the communication cost?

One approach I can think of is to use 32-bit `float` instead of 64-bit `double`, in which way we can at most reduce the communication data by 50%.

However, (maybe) the weird things happen. When I change the datatype for `Location` and `Mass`, `Radius` to `float`, leaving the computation part remaining using `double`, execution time even increases. In one process, the execution time becomes 32.7213 seconds and with four processes, the time is 233.9371 seconds. Compared to previous results of 27.4733 seconds and 148.2235 seconds, the new program is quite slow (at least in the computation jobs).

It sounds strange but actually it's not a surprising thing. As a hardware guy who is quite interested in computer architecture, from the Intel 64 Manual [4], I know that both the scalar single-precision multiplication and the double-precision instructions (`mulss` and `mulsd`) take the same four clock cycles to finish. Therefore, in these two cases where there's only a really small amount of transferred data and computation costs the most time, casting from `float` to `double` does cost a lot. According to

the manual, the casting instruction (`cvtss2si`) also takes four cycles.

Well, if one has enough memory or never transfer the data, then there're few reasons for using `float` instead `double`. So, don't hesitate to use `double`.

Now to improve the execution time, we replace all the `double` variables with `float` and use `float` through the computation. Still with the default parameters from Figure 14, the results are quite positive (Figure 10), where the elapsed time is reduced almost 50%.

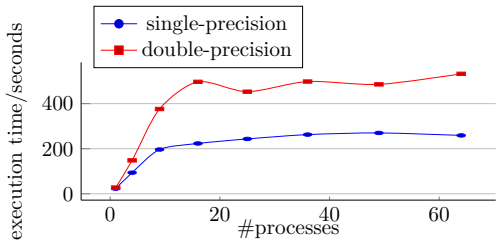
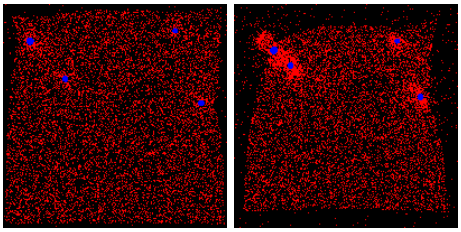


Figure 10: Execution time with `horizon` = 1 and 1000 particles



(a) 50 steps (b) 100 steps

Figure 11: Simulation results using single-precision data structures and arithmetic

The problem that someone may worry about is the accuracy of `float` compared to `double`. With single-precision floating-point numbers, we can obtain the output files as shown in Figure 11. At least I cannot see much deviation between the original results in Figure 1 and these two.

For more evidence, we profile the single-precision version as well and compare it with the double-precision program (Figure 12 and 13).

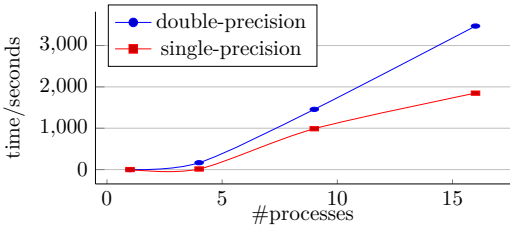


Figure 12: Accumulated communication time

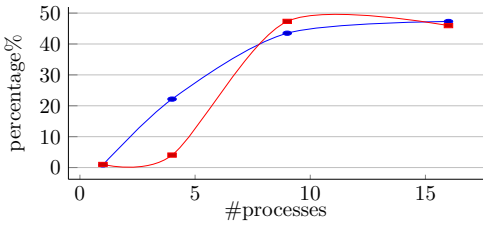


Figure 13: Percentage of the communication time in overall execution time

When there're four processes, communication takes less than 5%, which implies that the program doesn't need to wait for communication any more, contrary to the double-precision version. Although the percentage still remains high when there're more processes, the accumulated communication time has been cut down to a lower level, thus making the program more effcient.

The other approach to optimize the communication is to use `MPI_Waitany`, which will return one finished index, instead of `MPI_Wait`. Previously we loop over all the adjacent regions in a fixed order. With the first-come-first-serve strategy we can fully optimize waiting time.

I haven't got the results since when I run the program on Tembusu, usually `ps -aux` shows that my program's cpu usage is 0.0%. Even it finished after a long time, the results are abnormal (Table 6). For example, 16 processes cost 832 seconds and 770 seconds in two same runs. Using the original program with 4 processes, it takes even 210 seconds to finish. Maybe the servers are currently busy with some tasks (21 April night), since it shouldn't have such great fluctuation. Last several weeks the servers were normal.

To summary up, in this section, we discuss about the communication issues and argue that it's better to use single-precision float-point number and (maybe) `MPI_Waitany` in the simulation.

5 Appendix

5.1 Code Documentation

All the header files have been discussed in Section 2.2 and usage of all other files has also been covered in Section 2.4. Therefore, in this section, we simply list all the data structures and functions implemented by myself in C programing language.

C source code consists of five header files, including `pool.h`, `logging.h`, `pfile.h`, `spec.h` and `mympi.h`, and six `.c` files, five with the same names and one named `poolseq.c` for sequential simulation.

- `Radius`, `Mass`

  Datatype for radius and mass of a particle, defined as `double`

- `Location { double x; double y; }`

  Structure describing the location of a particle, including x-coordinate and y-coordinate.

- ```
  Particle {
    Radius   rad;
    Mass     mass;
    Location loc;
  }
  ```

  Structure describing a particle, including radius, mass and location.

- ```
  GridSpec {
    uint32_t  size;
    uint32_t  small_num;
    Mass      small_mass;
    Radius    small_rad;
    uint32_t  large_num;
    Particle *large_ptc;
  }
  ```

  Structure describing the grid (board, pool table) specification, including size of one pool region, number, mass and radius of small particles, number of large particles and their information.

- ```
  CompSpec {
    int   time_slot;
    float time_step;
    int   horizon;
  }
  ```

  Structure describing the computatin specification, including number of the time slots to be simulated, step size (time step) of every run and horizon (half size of the square).

- `Spec { CompSpec cs; GridSpec gs; }`

  Structure describing the specification for the simulation, including specifications for computation and the pool table.

- ```
  RGBPixel {
    uint8_t r;
    uint8_t g;
    uint8_t b;
  }
  ```

  Structure describing one pixel in `ppm` files in RGB format, including red, green and blue channel.

- ```
  PPMFile {
    uint64_t size;
    RGBPixel *pixels;
  }
  ```

  Structure describing a `ppm` file, including size (width and height) of the photo and pixels (stored row by row).

- `CompSpecType`, `PoolType`, `LocationType`, `ParticleType`, `RGBPixelType`

  `MPI_Datatype` for `CompSpec`, `Pool`, `Location`, `Particle` and `RGBPixel`. Unregistered before calling `init_mympi`.
  ```

- `Pool {`

  ```
  uint32_t  size;
  uint32_t  small_num;
  Mass      small_mass;
  Radius    small_rad;
  uint32_t  large_num;
  Location *small_ptc;
  Particle *large_ptc;
  }
  ```

  Structure describing the pool table, including size (also width and height), number of small particles, mass, radius of small particles, number of large particles, location of every small particle and information on every large particle.

- `Accel_t`, `Vel_t`

  Type for acceleration and velocity, defined as `double`.

- `Accel { Accel_t ax; Accel_t ay; }`

  Structure describing the acceleration of a particle, including components on x and y axis.

- `Velocity { Vel_t vx; Vel_t vy; }`

  Structure describing the velocity of a particle, including components on x and y axis.

- `int read_spec_from_file(`

  ```
  const char *filename,
  Spec *spec
  )
  ```

  Read specification from a file in the format as shown in Figure 14. `spec` needs to be allocated first but `spec->gs.large_ptc` will be allocated dynamically according to the input specification file. Returns non-zero numbers if failed.

- `int print2ppm(`

  ```
  const Pool *pool,
  const char *path
  )
  ```

  Output `pool` into a `ppm` file whose path is specified by `path`. `path` needs to be a directory. Returns non-zero numbers if failed.

- `int ppm2file(`

  ```
  const PPMFile *ppm,
  const char *path
  )
  ```

  Output `ppm` to file whose path is specified by `path`. `path` needs to be a directory. Returns non-zero numbers if failed.

- `int pool2ppm(`

  ```
  const Pool *pool,
  PPMFile *ppm
  )
  ```

  Convert `pool` into `ppm` format stored in memory. `ppm` needs to be allocated first but `ppm->pixels` will be allocated dynamically according to size of the pool. Returns non-zero numbers if failed.

- `int init_mympi()`

  Initizalize `mympi` environment, which includes self-defined MPI datatypes for simulation. Returns non-zero numbers if failed.

- `mympi_bcast`, `mympi_isend`, `mympi_irecv`, `mympi_waitall`, `mympi_wait`, `mympi_send`, `mympi_recv`, `mympi_barrier`

  Wrapper functions for `MPI_Bcast`, `MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`, `MPI_Wait`, `MPI_Send`, `MPI_Recv`, `MPI_Barrier`, where `comm` is defined as `MPI_COMM_WORLD`.

- In sequential version:

  ```
  int run(int argc, char *argv[])
  ```

  In parallelized version:

  ```
  int run(int rank, int size,
          int argc, char *argv[])
  ```

  Running the simulation, which begins with reading specification files.

5.2 Collected Data

Table 1, 2, 3, 4 reflect the execution time of the parallelized program, while Table shows the execution time of the sequential program. All the results are obtained using the default parameters shown in Figure 14 unless otherwise noted.

```
TimeSlots: 100
TimeStep: 0.1
Horizon: 1
GridSize: 200
NumberOfSmallParticles: 10000
SmallParticleMass: 0.0001
SmallParticleRadius: 0.0001
NumberOfLargeParticles: 4
8 1.5 56.5 70.21
6 1 156 20.1
12 3.1 20 30
8 1.7 180 90
```

Figure 14: Default parameters of the simulation

Table 1: Execution time of the parallelized program when `horizon` = 0

time/s #particles / #processes	50	100	1000	5000	10000	20000	50000
1	0.0019	0.0054	0.3429	7.0821	27.4829	-	-
4	0.0030	0.0051	0.3531	7.2692	29.2902	-	-
9	0.0016	0.0085	0.3562	8.0558	29.6309	-	-
16	0.0036	0.0082	0.3490	7.4596	29.6444	137.2558	861.8314
25	0.0033	0.0089	0.3681	8.1983	29.6175	-	-
36	0.0035	0.0091	0.4031	8.0626	29.7985	-	-
49	0.0035	0.0098	0.4040	7.4688	29.7074	-	-
64	0.0032	0.0090	0.4059	8.6851	30.0033	-	-

Table 2: Execution time of the parallelized program when `horizon` = 1

time/s #particles / #processes	50	100	1000	5000	10000	20000	50000
1	0.0017	0.0061	0.3426	6.9746	27.4733	-	-
4	0.0139	0.0327	1.5979	37.2951	148.2235	-	-
9	0.0155	0.0512	3.9938	90.3270	376.1248	-	-
16	0.0354	0.0743	4.5745	105.8938	496.7950	2200.9912	13329.4369
25	0.0483	0.0857	5.2703	115.4488	453.0209	-	-
36	0.0554	0.1166	4.8540	129.4312	498.0560	-	-
49	0.0699	0.1243	4.8635	124.8612	485.5048	-	-
64	0.0778	0.1340	5.2938	120.7748	532.3078	-	-

Table 3: Execution time of the parallelized program when `horizon` = 2

time/s #particles / #processes	50	100	1000	5000	10000	20000	50000
1	0.0019	0.0057	0.3426	7.0176	27.4181	-	-
4	0.0092	0.0368	1.6182	38.3220	148.1225	-	-
9	0.0228	0.0570	3.8448	91.2998	459.8063	-	-
16	0.0719	0.1342	6.6878	182.1946	818.3910	3658.3177	27571.7473
25	0.1243	0.2460	10.7850	269.4800	1086.1783	-	-
36	0.2156	0.3733	11.7528	278.8278	1124.2637	-	-
49	0.2651	0.5155	12.6770	299.9271	1158.9334	-	-
64	0.2717	0.5272	12.3862	302.7807	1240.5233	-	-

Table 4: Execution time when `horizon`= 3

time/s #particles / #processes	50	100	1000	5000	10000	20000	50000
1	0.0019	0.0062	0.3425	6.9490	27.6517	-	-
4	0.0092	0.0378	1.6276	37.1357	184.9982	-	-
9	0.0294	0.0621	3.7421	95.7116	495.2453	-	-
16	0.0853	0.1865	7.5575	249.0195	1157.6074	5638.0601	30517.5672
25	0.1948	0.3716	12.5872	331.8684	1456.9483	-	-
36	0.3726	0.6255	17.4338	407.4336	1597.0198	-	-
49	0.4333	0.7738	22.7071	520.7917	2052.0152	-	-
64	0.5066	0.9281	23.6921	537.6383	2926.2521	-	-

Table 5: Execution time of the sequential program

time/seconds `horizon` / #regions	0	1	2	3
1	27.4767	27.4444	27.3760	27.4579
4	109.1160	550.8104	551.0297	549.8375
9	248.4509	1791.8073	3222.1316	3216.9257
16	437.7138	3936.2836	-	-
25	691.2138	8308.0838	-	-
36	998.0485	17705.7742	-	-
49	1363.9840	35954.1161	-	-
64	1787.7458	74666.7526	-	-

Table 6 shows the execution time of the single-precision version and the `MPI_Waitany` version of the program.

Table 6: Execution time of modified version of the program

time/s \ #processes / version	1	4	9	16	25	36	49	64
single-precision	22.7911	93.6934	196.3274	223.1662	243.4202	262.5972	269.9987	258.9388
`MPI_Waitany`	27.5230	178.5560	376.9604	832.4868 770.5914	899.5052	834.6954	920.3665	938.7403

Table 7 shows the accumulated exclusive time of different function groups recorded by VampirTrace. Table 8 shows the time-consuming MPI functions. `MPI_Barrier` is used only when we output the results into `ppm` file, which we don't care about regarding the performance.

Table 7: Accumulated exclusive time of different function groups

time/seconds \ #processes / group		1	4	9	16
double-precision	Application	27.5517	584.0187	1892.5418	3868.9825
	MPI	0.2390	166.2682	1456.5773	3470.9612
single-precision	Application	22.8320	345.4949	1099.0663	2174.4321
	MPI	0.2309	14.5464	987.3581	1848.5711

Table 8: Time-consuming MPI functions

time/seconds \ #processes / function		1	4	9	16
double-precision	`MPI_Waitall`	-	82.8961	1267.2125	2729.7575
	`MPI_Wait`	-	81.4825	168.7205	714.9178
	`MPI_Barrier`	-	0.8470	18.1870	21.8132
	`MPI_init`	0.2386	1.0194	2.3847	4.2147
single-precision	`MPI_Waitall`	-	9.7549	889.6732	1569.9230
	`MPI_Wait`	-	3.5033	85.8099	256.1313
	`MPI_Barrier`	-	0.0395	9.4943	16.9993
	`MPI_init`	0.2306	1.2295	2.3174	5.3165

References

[1] Gerard, "gerard/ext4fuse," 2018.

[2] T. S. Blog, "Mpi – tutorial 5 – asynchronous communication," 16 Jul. 2009.

[3] G.-T. GmbH, "Vampir basics," 2018.

[4] Intel, "Intel 64 and ia-32 architectures optimization reference manual," 11 Apr. 2018.