香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005
Parallel Computing

Project 3

Author	**Student ID**
Luo Haoyan	119010221

November 14, 2022

Contents

1 Introduction

N-body problem is an physical and astronomical problem which study the movement mechanism of several bodies in space. This project aims at simulating arbitrary number of bodies' movement in the 2-dimensional space. The basic assumptions are:

1. . All the bodies follows no other than the Newton's rules.
2. The bodies are absolutely rigid bodies, which means that if collision happens, the bodies won't reshape or break.
3. There is a boundary for the space During the benchmark, I fix it to 800*800and the boundary is also absolutely rigid. And for the number of iteration I fix it to 100. The number of bodies is changed during benchmarking.

I adopted 6 different approaches for the simulation, which are Sequential, MPI, Pthread, OpenMP, CUDA, and MPI + OpenMP versions. The implementation details will be illustrated in the following sections. Comprehensive experiments are conducted and performance under different configurations in four dimensions are analyzed: different number of threads/cores used; performance of Sequential program vs. MPI program vs. Pthread program vs. OpenMP program vs. CUDA program; performance under different problem size (number of bodies); performance of MPI program vs. OpenMP program vs. MPI+OpenMP program. You can check the appendix for the steps to run my program.

2 Method

2.1 Sequential Version

The sequential program for the N-body computation is basically the implementation of updating position and updating velocity. In my implementation, I distribute two different types of boundary collision to those two function: checking boundary collision while updating position and checking body collision while updating velocity. For example, to check the boundary collision, you can refer to the following code snippet:

```
if (x[i] <= sqrt(radius2) || x[i] >= bound_x - sqrt(radius2)) vx[i] = -vx[i];
if (y[i] <= sqrt(radius2) || y[i] >= bound_y - sqrt(radius2)) vy[i] = -vy[i];
```

Note that in my implementation, I consider the collision among bodies as the direct reverse of its original speed (as suggested in the tutorial). In the function for updating velocity, there are two for loop to conduct the desired function. The outer for loop means the corresponding body in this round (later in the parallel implementation, the outer for loop refers to the body that assigned to different cores/threads), and the inner for loop is responsible for calculating the relative information with all other bodies. You can refer to the following code snippet for the calculation of the new speed:

```
deltaX = x[j] - x[i];
deltaY = y[j] - y[i];
distance = sqrt((deltaX * deltaX) + (deltaY * deltaY));
acceleration = gravity_const * m[j] / (distance * distance + err);
vx[i] = vx[i] + dt * acceleration * deltaX/distance;
vy[i] = vy[i] + dt * acceleration * deltaY/distance;
```

2.2 MPI Version

The program of MPI version can be divided into three parts:

- Broadcast the basic information and the states of the bodies from root process to other process;
- Divide the problem of updating accelerations and checking collisions to do parallel computing, and merge the results;
- Divide the problem of updating velocity, position and checking collisions to do parallel computing, and merge the results.

Following Figure shows the overall structure of my MPI program.

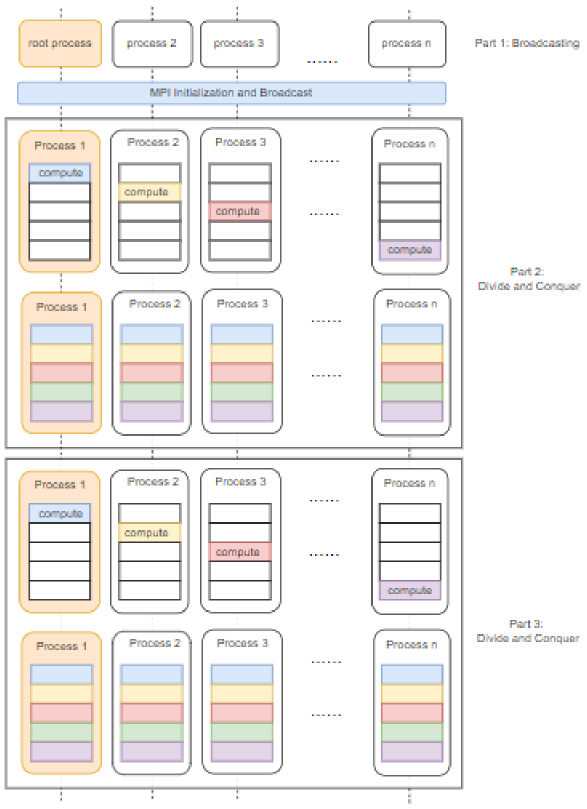


Figure 1: *The overall structure of MPI program*

2.2.1 Initialization and Broadcasting

In the first part, the root process first deals with the basic information of the input from user, including the number of bodies and the number of iterations, and then broadcast these meta-information to other processes. Then the root process will initialize the state of all bodies and store them in different array buffers. Note that the information we broadcast here includes the overall information of bodies such as the total mass and total coordinates. The following code snippet shows the corresponding broadcasted information:

```
// send common data to all processes
MPI_Ibcast(body_count, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
MPI_Ibcast(displacements, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
MPI_Ibcast(total_m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Ibcast(total_x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Ibcast(total_y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
```

2.2.2 Data Allocation and Gathering

Similar with the previous implementation of MPI program, we have to allocate the data to different processes to perform the divide-and-conquer strategy to increase the efficiency. Here, to further boost the performance, I choose the non-blocking operation to divide and gather the workload to different processes. The following code snippet shows the gathering of the data with non-blocking operation:

```
// wait and get local data
MPI_Igatherv(local_x, num_my_body, MPI_DOUBLE, total_x, body_count, displacements,
↪ MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Igatherv(local_y, num_my_body, MPI_DOUBLE, total_y, body_count, displacements,
↪ MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Igatherv(local_vx, num_my_body, MPI_DOUBLE, total_vx, body_count, displacements,
↪ MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Igatherv(local_vy, num_my_body, MPI_DOUBLE, total_vy, body_count, displacements,
↪ MPI_DOUBLE, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

2.2.3 Divide and Conquer: Updating Velocities and Positions

This part is simply call the implemented functions that defined in the sequential program. However, there is one thing worth noticing that in my implementation, I consider the overall information (total mass, total coordinates) as the input to the function as following:

```
    // perform calculation
    update_velocity(total_m, local_x, local_y, local_vx, local_vy, total_x, total_y,
↪   num_my_body);
    update_position(local_x, local_y, local_vx, local_vy, num_my_body);
```

This is because the workload distributed to different process is the computation task of a single body. That means: a process needs to compute all the needed operations related to that body and update its states, which includes the operations with respect to all other bodies (up to the total number of bodies). Meanwhile, each process would only update the states of the allocated bodies in order to prevent from data race.

2.3 Pthread Version

For the pthread parallel program, there is no need to broadcast information from root process to other processes since Pthread can share the memory among threads. The overall structure of Pthread program is shown below:

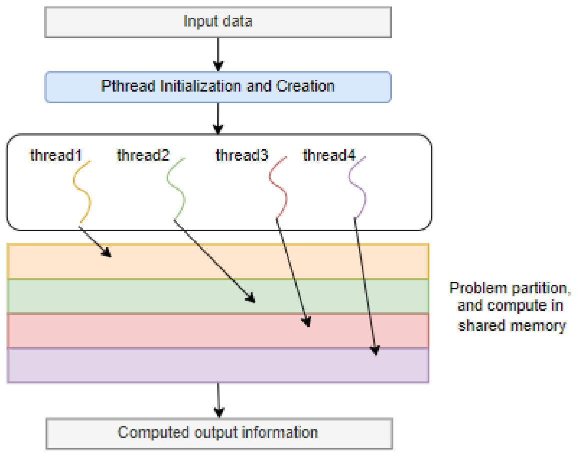

Input data
Pthread Initialization and Creation
thread1
thread2
thread3
thread4
Problem partition, and compute in shared memory
Computed output information

Figure 2: *The overall structure of Pthread program*

2.3.1 Threads and Mutex Initialization

The basic computation data of problem is initialized and stored in share memory, so different threads have the ability to access the shared information. Besides, the allocated number of threads are created and they will execute the same function with different input data. To avoid the data race, some mutex locks are also created in this part.

2.3.2 Problem Division

To better illustrate the idea of pthread parallel computing, you can refer to the following figure:

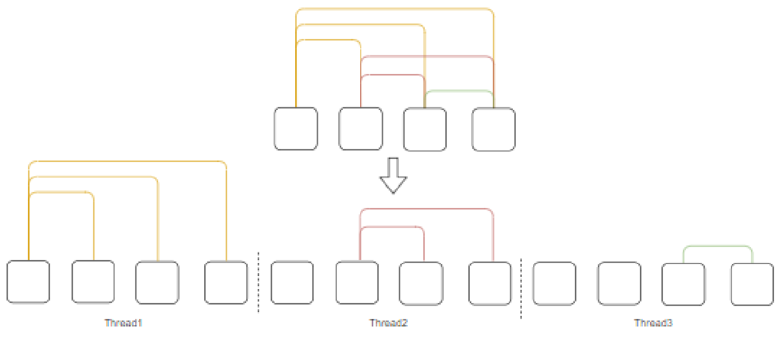


Figure 3: *The calculation routine of Pthread program*

The problem division of this approach is partition the outer-loop of the updating. As shown in the above figure, if there are 4 bodies to be calculated, the sequential version will calcuate the each pair of the bodies one by one. While in the pthread version, suppose there are 3 threads, then, each thread would be incharge of one body and its pairs with other bodies. That's thread 1 would calculate body 1 with 2, 3, 4. Thread 2 would calcualte body 2 with 3, 4. Thread 3 will compute body 3 with 4. You can refer to the following code snippets:

```
int num_my_body = n_body / num_thread;
int remain = n_body % num_thread;
if (id_thread < remain) num_my_body++;
int displacement = num_my_body * id_thread;
double* local_m = my_arg->m + displacement;
double* local_x = my_arg->x + displacement;
double* local_y = my_arg->y + displacement;
double* local_vx = my_arg->vx + displacement;
double* local_vy = my_arg->vy + displacement;
```

2.3.3 Parallel Computing

Given the partition with the above mentioned mechanism, each thread would perform the function with their own workload. Notice that all the data are stored in the shared memory which can be accessed by all threads, so we have to take the possible situation of data race into consideration. To prevent data race from happening, n mutex locks are initialized (n is the number of bodies). Once the operation regarding to the state of body is evolved, the mutex locks will be evoked. The detailed implementation is generally the same with the sequential computation, and you can refer the details to Figure 2.

2.4 CUDA Version

As for the CUDA version, there are mainly four components to realize the parallel computing function:

1. Malloc memory in CPU for storing the states of bodies in host, and mallo memory in GPU for storing the states of bodies in device.
2. Randomly generate states of bodies in host and copy the information from host to device.
3. Launch the threads to perform computation in parallel.
4. Perform divide and conquer strategy at device, and copy the results back to host.

Below is a general description figure that summarize the dataflow of CUDA program. Details are elaborated in the following sections.

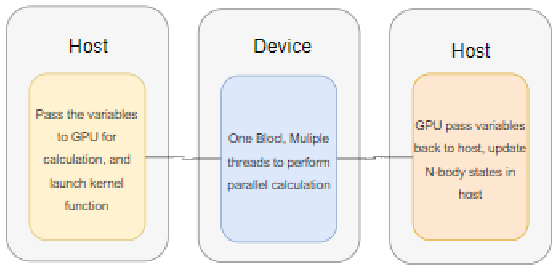

Host
Pass the variables to GPU for calculation, and launch kernel function
Device
One Blocl, Muliple threads to perform parallel calculation
Host
GPU pass variables back to host, update N-body states in host

Figure 4: *The general description of CUDA program*

2.4.1 Malloc Memory in Host and Device

Compared to other implementation of parallel computing, the memory structure of CUDA program is relatively more complicated. To ensure the operation efficiency and the correctness of data manipulation both in the host and the device, we first malloc two distinguished memory space in both host and device for storing the states of N-bodies.

2.4.2 Random Generation and Data Transfer

First, the host will be in charge of randomly generating position and velocity of the body. After random genertion, to keep the data consistency between host and device as well as prepare for the parallel computation in GPU, the states of bodies would be copy from host to device.

2.4.3 Problem Partition and Parallel Computing

In this part, the main calculation routine would be performed. The host would launch the kernel threads to perform the parallel computation. Similar to the previous parallel paradigm, the workload is divided with the same division strategy (as shown in Figure 3). In addition, one of the most important thing is to keep the data free from data race (since the memory declare with device could be accessed by all the kernel threads in one block). Following code snippet shows the call of functions:

```
update_velocity<<<n_block, block_size>>>(device_m, device_x, device_y, device_vx,
↪  device_vy, n_body);
update_position<<<n_block, block_size>>>(device_x, device_y, device_vx, device_vy,
↪  n_body);
```

Detials for the structure in the part as well as the overall detailed structure of CUDA program is shown in the following figure.

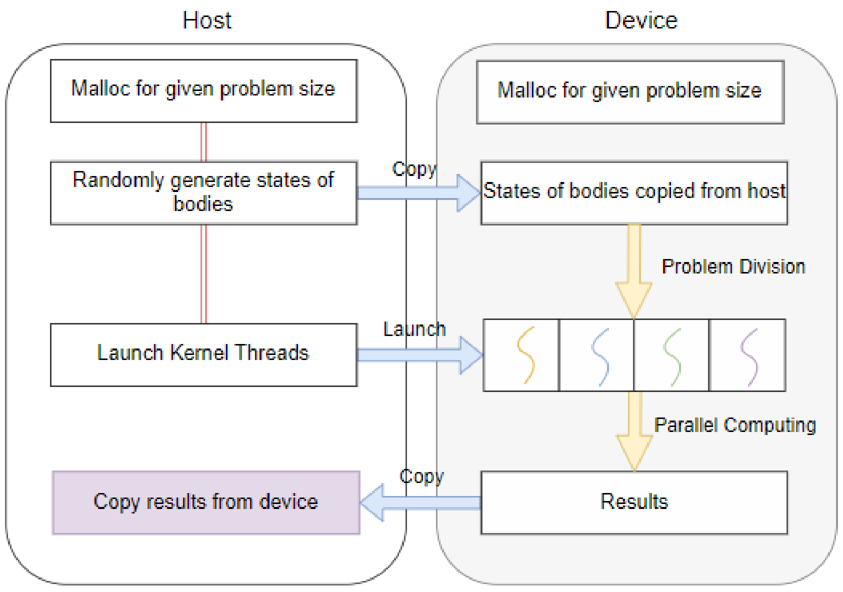

Host
Device
Malloc for given problem size
Malloc for given problem size
Randomly generate states of bodies
Copy
States of bodies copied from host
Problem Division
Launch Kernel Threads
Launch
Parallel Computing
Copy results from device
Copy
Results

Figure 5: *The overall detailed structure of CUDA program*

2.4.4 Results Integration

After computing (updating of acceleration first then updating the velocity and position) of each kernel threads. To keep the data consistency again, the results would be copied back to the host memory which can be used by host to Paint the graph of simulation. The following code snippet is responsible for the mentioned operation:

```
cudaMemcpy(x, device_x, n_body * sizeof(double), cudaMemcpyDeviceToHost);
cudaMemcpy(y, device_y, n_body * sizeof(double), cudaMemcpyDeviceToHost);
```

2.5 OpenMP Version

The OpenMP implementation routine is almost the same as the Pthread version, since they all share the core memory. And OpenMP version is very easy to implement since it would automatically allocate thread for the user. The overall structure of the OpenMP program is shown below.

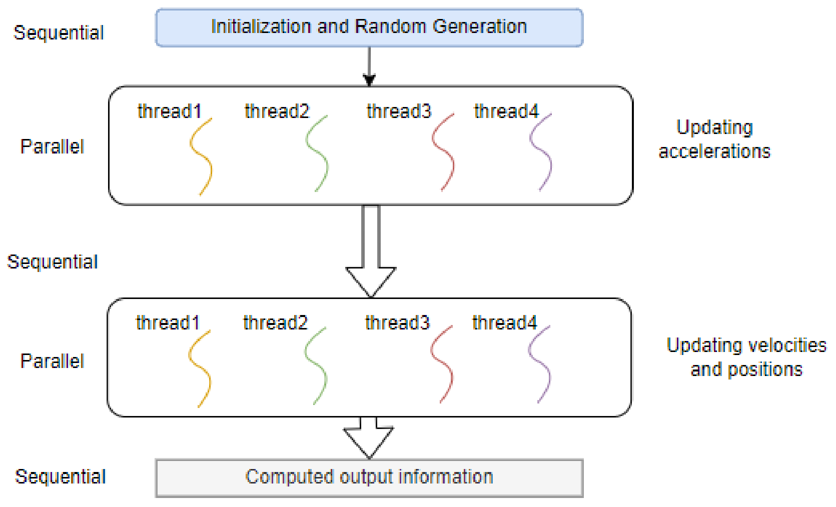


Figure 6: *The overall structure of OpenMP program*

You can consider the above paradigm as a fork-join strategy. The sequential stage (indicated on the left of the figure) performs the initialization of the problem, and random generation of the states. While the parallel stage conducts two computations as before: one is updating the acceleration and the other is updating the velocity and the position (see the code below using updating velocity as an example):

```
omp_set_num_threads(n_omp_threads);
#pragma omp parallel for
for (int i = 0; i < n_body; i++) {
    update_velocity(m, x, y, vx, vy, i);
}
```

To avoid the data race, lock and barrier are used. Each thread will parallelly update the velocity and wait other threads to finish. After all threads reach the barrier, they will update the distance information as well. A mutex lock is also set when the speed information is accessed, which ensures that there will be only one body being modified at one time.

2.6 Bonus: MPI + OpenMP Version

Since MPI allows communication among different cores, and each cores allow different number of threads to execute the same function with different workload, it is naturally a good idea to combine these two approaches together. The design of MPI + OpenMP program is just basically adding the OpenMP approach to the MPI design. Figure 7 shows the overall structure of MPI+OpenMP.

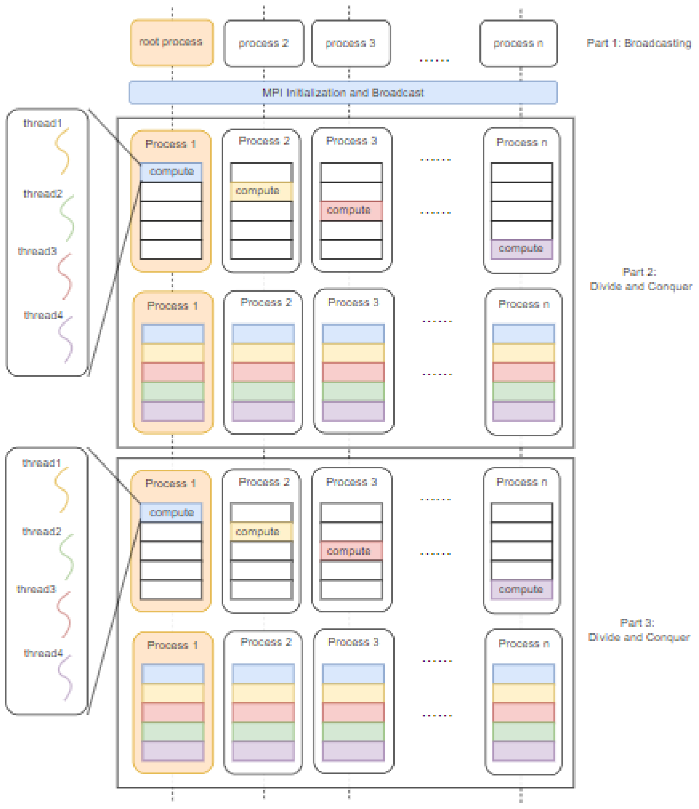


Figure 7: *The overall structure of MPI+OpenMP program*

The sub-problems would be assigned to each process similar to MPI program. However, the workload in each process is further sub-divided into different threads by OpenMP and then computed parallelly. After all the threads finish their computation (which means all the processes finish their computations as well), the data would be gathered and to the root process. In my implementation, I named the combination as *openmpi*.

3 Experiments

Comprehensive experiments are conducted to test and analyze the performance of parallel computing by MPI programming and multi-threaded programming. Various input data size are tested (range from size 100 to size 10000, here the size means the input resolution, thus the actually size of the data is $size^2$). Here, small size (100 - 1000), medium size (2000 - 4000), and large size (8000 - 10000) are respectively defined. All the test result ran by sbatch can be found in corresponding *.out* files.

3.1 Evaluation Metrics

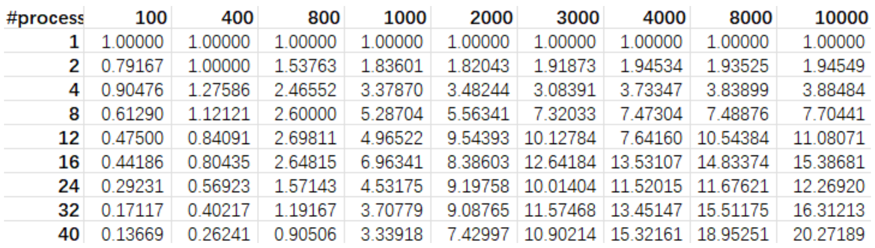
In addition to the running time of the program, another evaluation metric: Speedup Factor, is introduced in the evaluatioin part. Parallel speedup is defined as the ratio of the time required to compute some function using a single processor (t_s) divided by the time required to compute it using P processors (t_p), the formula can be explicitly written as:

$$S(n) = \frac{Execution\ time\ using\ one\ processor(single\ processor\ system)}{Execution\ time\ using\ amultiprocessor\ with\ n\ processors} = \frac{t_s}{t_p}$$

3.2 Results

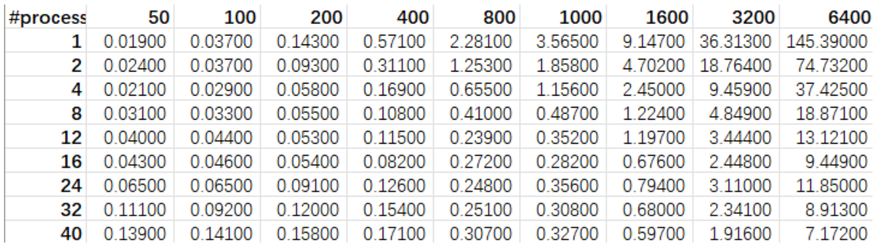
3.2.1 Main results

The Speedup results (based on the equation above) of Sequential version, MPI version, Pthread version, OpenMP version, CUDA version, and the bonus (OpenMP + MPI) are presented the following tables. The running time results for different configurations are also provided i(in the output file with the submitted code). Note that all the experiments in my implementation fix the number of iteration to 500.



#process	100	400	800	1000	2000	3000	4000	8000	10000
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.79167	1.00000	1.53763	1.83601	1.82043	1.91873	1.94534	1.93525	1.94549
4	0.90476	1.27586	2.46552	3.37870	3.48244	3.08391	3.73347	3.83899	3.88484
8	0.61290	1.12121	2.60000	5.28704	5.56341	7.32033	7.47304	7.48876	7.70441
12	0.47500	0.84091	2.69811	4.96522	9.54393	10.12784	7.64160	10.54384	11.08071
16	0.44186	0.80435	2.64815	6.96341	8.38603	12.64184	13.53107	14.83374	15.38681
24	0.29231	0.56923	1.57143	4.53175	9.19758	10.01404	11.52015	11.67621	12.26920
32	0.17117	0.40217	1.19167	3.70779	9.08765	11.57468	13.45147	15.51175	16.31213
40	0.13669	0.26241	0.90506	3.33918	7.42997	10.90214	15.32161	18.95251	20.27189

Figure 8: *The speedup results of MPI program*

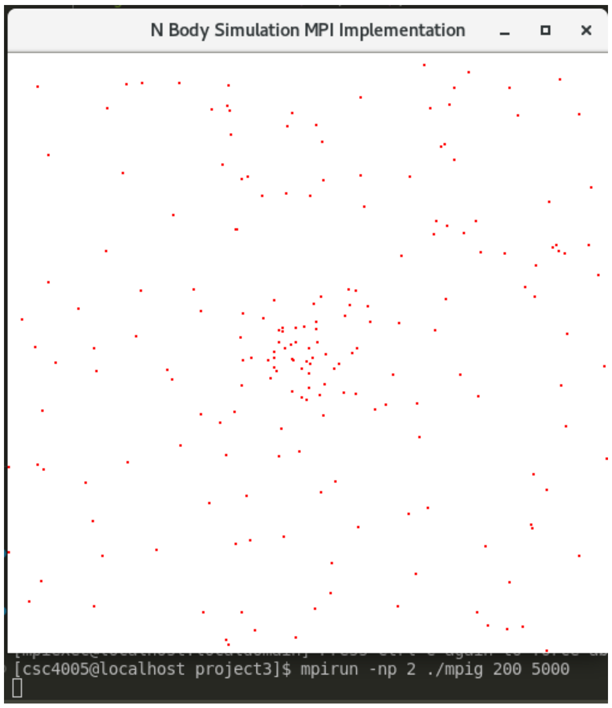


#process	50	100	200	400	800	1000	1600	3200	6400
1	0.01900	0.03700	0.14300	0.57100	2.28100	3.56500	9.14700	36.31300	145.39000
2	0.02400	0.03700	0.09300	0.31100	1.25300	1.85800	4.70200	18.76400	74.73200
4	0.02100	0.02900	0.05800	0.16900	0.65500	1.15600	2.45000	9.45900	37.42500
8	0.03100	0.03300	0.05500	0.10800	0.41000	0.48700	1.22400	4.84900	18.87100
12	0.04000	0.04400	0.05300	0.11500	0.23900	0.35200	1.19700	3.44400	13.12100
16	0.04300	0.04600	0.05400	0.08200	0.27200	0.28200	0.67600	2.44800	9.44900
24	0.06500	0.06500	0.09100	0.12600	0.24800	0.35600	0.79400	3.11000	11.85000
32	0.11100	0.09200	0.12000	0.15400	0.25100	0.30800	0.68000	2.34100	8.91300
40	0.13900	0.14100	0.15800	0.17100	0.30700	0.32700	0.59700	1.91600	7.17200

Figure 9: *The running time results of MPI program*

3.2.2 Image Display

The GUI outputs for MPI version as an example when the input size is 800x800 are shown below. The parallel program uses 2 processes/threads and the number of iteration as as a demo example.



```
[csc4005@localhost project3]$ mpirun -np 2 ./mpig 200 5000
```

Figure 10: *The demo output of MPI program*

3.3 Performance Evaluation: Different Number of Threads or Cores

According to the results shown in the above Table, increases in speedup are both observed. To better understand the results , the visualization of the results for different number of threads is presented in the following figure:

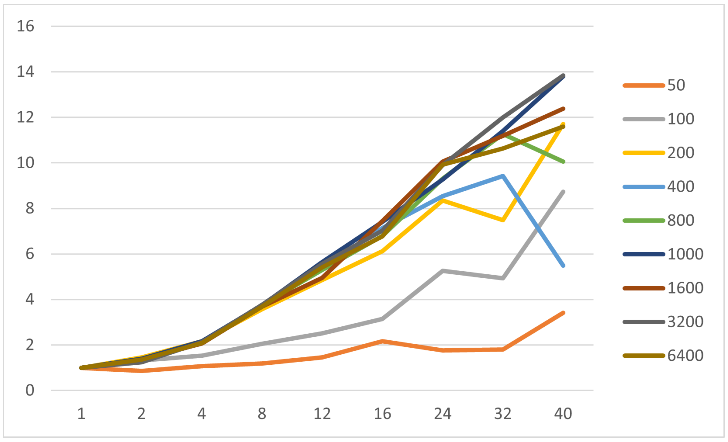


Figure 11: *The relationship between number of processes and speedup performance given different number of size (MPI)*

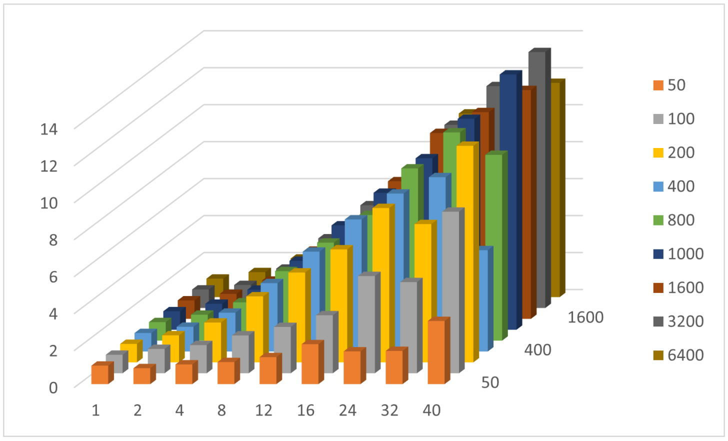


Figure 12: *The relationship between number of processes and speedup performance given different number of size (Pthread)*

We can have the following three observations:

1. As shown in the figure, for relatively small problem size, as the number of cores increases, the overall performance of the parallel program increases (as long as it is not under a relatively small size).

2. From the above figure we can observe that the parallel approach will first enhance the performance as the number of cores increases in the small-medium input settings, and then the performance will fluctuate (see size=1000 as an example) but basically remain the same.

3. With a relatively large problem size, the performance keeps increasing as the number of cores increases. There are some surge in some configuration settings but the trend is generally upward

Analysis The observations above enlighten that the optimal number of processes for a given configuration depends on many factors such as problem size. As the number of cores increase, workload are more evenly to be distributed to each core and perform the parallel computing, thus, each core's workload is decreased and total speed enhances. Particularly, parallel computing can improve performance notably when the number of processes fits the problem size (i.e. the number of input elements can be fully divided by the number of processes). Moreover, it can be found that when the problem size is large (greater than 8000), the performance can reach its maximum and can even increase as the problem size gets larger.

For cases of pthread and openmp, the treads are basically the same. As shown in the figure, for problem size ¿ 200, the performance would increase nearly linearly as the number of threads increases. Then, the tendency of the increasing would cease when number of threads are greater than 32. We can have the following three observations:

1. As the number of threads increases, the performance (speedup) is almost linearly increasing when the input size is medium and large. For example, when the input size is 10000 and the number of threads allocated is 40, the speedup factor can be up to 12.
2. However, when the input size is small, the performance improvement is not that obvious and might even harm the performance (see size 100 as an example).
3. Another interesting thing is the increase of each line is not smooth (which means the performance improvement will sometimes stop for a while as the number of threads increases).

Analysis The generally increasing tendency can be explained by Amdal's law:

$$S(n) = \frac{n}{1 + (n-1)f}$$

Since there is no data dependency in Mandelbrot set computation (all pixels would not affect each other), serial section (f in amdal's law) is quite low and can perfectly explain the performance gain as the number of threads increases. In addition, similar with MPI program, the performance gain is most obvious when the problem size is large or medium, while when the problem size is small, the overhead of allocating data may offset or exceed the performance gain given by parallel computing.

Moreover, the reason why the performance gain is small when the problem size is small is that the data allocation overhead can offset and even exceed the performance gain from parallel computing. The conclusion basically meets the statement of Amdahl's law, limited by the serial fraction (In program, like some preparation work done by root process), the improvement of performance that increasing threads brings is bounded

3.4 Performance Evaluation: Different number of N-bodies

As it is shown in Figure. the perfromance would keep increasing as the problem size becomes greater. However, as the problem size keep increasing all the performance of all the approach would slow down and even cease.

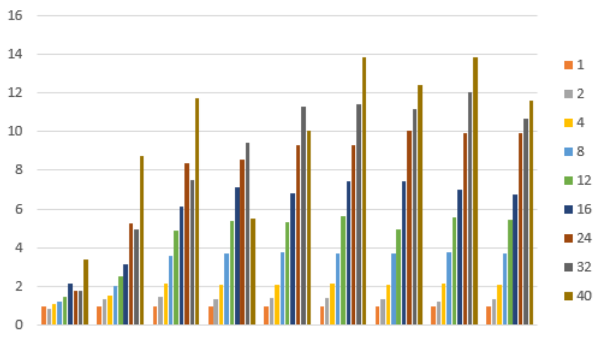


Figure 13: *The visualization of different input number of bodies*

From the figures above, we can observe the following:

1. When the problem size is small, no matter the number of threads/cores allocated is large or small, the performance will increase as the size increases.

2. When the problem size is medium, for smaller number of threads/cores, the performance will "stop" to increase as the size increases and keep flatten. This is more obvious in MPI program (since pthread program here use dynamic scheduling, which will stressed in following session).

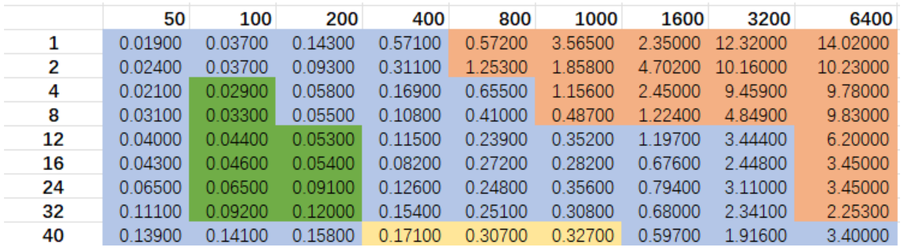
3. Some fluctuations at MPI program are observed when the number of cores allocated is large (see 40 as an example). This observation is in line with the previous analysis about the allocation overhead.

Analysis The increased performance when the problem size is relatively small can be explained by Gustafson's law, which claims that as the problem size goes up, parallel computing will have a higher performance because:

$$speedup\ factor = S(n) = n + (1 - n)s$$

where s is the number of cores available and n is the proportion of sequential computing. As the problem size increases, the sequential proportion n will decrease, which makes the speedup more close to the number of cores. However, there also exists a reasonable boundary for this improvement as the problem size getting larger and larger. Thus, the performance improvement will slow down and cease when the problem size is too large (relative to the number of cores/threads). The trends are generally observed in all implementations.

3.5 Performance Evaluation: Different Parallel Approaches



	50	100	200	400	800	1000	1600	3200	6400
1	0.01900	0.03700	0.14300	0.57100	0.57200	3.56500	2.35000	12.32000	14.02000
2	0.02400	0.03700	0.09300	0.31100	1.25300	1.85800	4.70200	10.16000	10.23000
4	0.02100	0.02900	0.05800	0.16900	0.65500	1.15600	2.45000	9.45900	9.78000
8	0.03100	0.03300	0.05500	0.10800	0.41000	0.48700	1.22400	4.84900	9.83000
12	0.04000	0.04400	0.05300	0.11500	0.23900	0.35200	1.19700	3.44400	6.20000
16	0.04300	0.04600	0.05400	0.08200	0.27200	0.28200	0.67600	2.44800	3.45000
24	0.06500	0.06500	0.09100	0.12600	0.24800	0.35600	0.79400	3.11000	3.45000
32	0.11100	0.09200	0.12000	0.15400	0.25100	0.30800	0.68000	2.34100	2.25300
40	0.13900	0.14100	0.15800	0.17100	0.30700	0.32700	0.59700	1.91600	3.40000

Figure 14: *The best running time summation of all results*

We have the best configuration of Pthread colored in blue, openmp colored in green, cuda colored in yellow and mpi colored in orange. Based on comparison table, we can make the following conclusion:

1. MPI outperforms others when the problem size is large and number of cores/threads is small. The greater the problem size, the more advantage that MPI take (especially when the number of cores is under 16).

2. . Pthread generally outperforms others when the problem size is small and number of threads is large.

3. OpenMP takes advantage when the problem size is small and the number of threads is between 8 and 64.

Analysis It can be observed that MPI takes its advantage of seperate memory space. Because other 3 approaches all rely on the shared memory. When the problem size becomes larger, the time to access shared memory becomes costy. That's why the greater the problem size, the more advantage that MPI take. However, after 32 cores/threads, Pthread would outperform MPI. That's because the communication overhead of MPI brought by the crossing Node communication. Since each node equip 32 cores, when the number of cores exceed 32, the communication should be cross the nodes.

At the same time, Pthread avoid the heavy communication overhead of MPI when the problem size is small. In addtion, it avoid the communication and memory copy between CPU and GPU like CUDA performs. What's more, the scheduling overhead of OpenMP is also avoided by Pthread. In general Pthread avoid the dominant affects that other approaches would met when problem size is small, so it outperforms others under these configuration. So pthread performs greate in most of the configurations.

The relatively worse performance can be explained by the overhead of movement of data between host and device (GPU overhead). In additon, there is also a kernel launch overhead added. Since in my program, to keep the data consistency between host and device, all the vector of the states of bodies shall be copied from CPU to GPU and back from GPU to CPU again. This process of transfering data cause a great penalty when the number of threads and problem size are both not in large scale.

3.6 Performance Evaluation: MPI vs. OpenMP vs. MPI+OpenMP

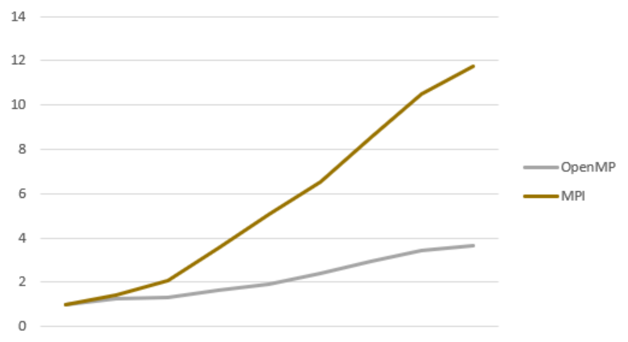


Figure 15: *Comparison between MPI and OpenMPI implementation, with 4 threads and 4 threads + 4 processes used in comparison*

Surprisingly, we found that the performance of openmpi is worse than the mpi (we choose mpi to compare since it is generally better than openmp). This phenomenon can be explained by the problem size is not large enough. For example, the biggest problem this experiment tested is 6400. If the cores is 64, it means that each threads of MPI deals with a sub-problem with 100. As it is shown in Figure 16, when computing such small problem size, the sequential version would outperform the parallel one. Therefore under this configuration, adding OpenMP to MPI will decrease the performance since the problem size is too small.

However, theoretically, when the problem size is large enough, the MPI+OpenMP version would get a better performance, to verify it, I test bonus program with problem size of 10000. And the result shows that the openmpi can take nearly 90 seconds to solve the problem while mpi need to take up to 240 seconds. This result verified the assumption.

Appendix

A Steps to Run My Program

Following are the steps to run my code (First please ensure that you are under the project folder). Note that the bonus implementation is in the file named of *openmpi.cpp*.

1. Compile: To compile all the program using Makefile, type the following commands in terminal:

```
make all
```

If you want to clear all the executable files and recompile, type

```
make clean
```

2. Run: To run different programs you can type the following commands respectively. The commands include both command line version and GUI version. For sequential version:

```
./seq $n_body $n_iterations
./seqg $n_body $n_iterations
```

MPI:

```
mpirun -np $n_processes ./mpi $n_body $n_iterations
mpirun -np $n_processes ./mpig $n_body $n_iterations
```

Pthread:

```
./pthread $n_body $n_iterations $n_threads
./pthreadg $n_body $n_iterations $n_threads
```

CUDA:

```
./cuda $n_body $n_iterations
./cuda $n_body $n_iterations
```

OpenMP:

```
openmp $n_body $n_iterations $n_omp_threads
openmpg $n_body $n_iterations $n_omp_threads
```

OpenMP+MPI:

```
mpirun -np $n_processes ./openmpi $n_body $n_iterations $n_threads
mpirun -np $n_processes ./openmpig $n_body $n_iterations $n_threads
```

You can also run them in a batch style if you are using a cluster:

```
sbatch seq.sh

sbatch mpi.sh

sbatch pthread.sh

sbatch openmp.sh

sbatch cuda.sh

sbatch openmpi.sh
```

You can check the results in the corresponding output file (under the slurm script folder).

B Running Time Results

Table 1: *The running time results for Pthread version* ***Note:*** *The unit is second in scientific notation.*

#process	100	400	800	1000	2000	3000	4000	8000	10000
1	0.00376	0.04347	0.18125	0.27513	1.05216	2.39160	4.20471	16.84305	26.39679
2	0.00436	0.03304	0.12396	0.20581	0.75111	1.70310	3.10676	13.54801	19.52260
4	0.00353	0.02853	0.08408	0.13109	0.50775	1.10417	2.03446	7.91678	12.67000
8	0.00315	0.02124	0.05077	0.07442	0.27994	0.64104	1.14114	4.47487	7.15209
12	0.00258	0.01736	0.03729	0.05093	0.19840	0.42364	0.85034	3.03129	4.87441
16	0.00174	0.01386	0.02965	0.03851	0.15479	0.32279	0.56476	2.40119	3.89836
24	0.00212	0.00826	0.02170	0.03221	0.11294	0.25785	0.41815	1.70008	2.65793
32	0.00209	0.00882	0.02423	0.02917	0.09340	0.20964	0.37590	1.40351	2.48265
40	0.00110	0.00497	0.01547	0.05020	0.10470	0.17318	0.33965	1.21653	2.27738

Table 2: *The running time results for OpenMP version* ***Note:*** *The unit is second in scientific notation.*

#thread	100	400	800	1000	2000	3000	4000	8000	10000
1	0.00376	0.04347	0.18125	0.27513	1.05216	2.39160	4.20471	16.84305	26.39679
2	0.00576	0.03475	0.13992	0.19814	0.77331	1.66242	2.94254	11.98018	18.73173
4	0.00410	0.03333	0.11388	0.14763	0.52356	1.14105	2.00886	8.12982	12.81560
8	0.00276	0.02641	0.06767	0.09465	0.32186	0.67401	1.18089	4.72223	7.47310
12	0.00235	0.02268	0.05599	0.07571	0.23183	0.48717	0.85737	3.34717	5.23219
16	0.00222	0.01796	0.04667	0.06109	0.19722	0.38290	0.68112	2.59864	4.04404
24	0.00272	0.01478	0.04012	0.05689	0.15264	0.29820	0.52935	1.98085	3.09464
32	0.00305	0.01261	0.03859	0.04874	0.13558	0.24780	0.43881	1.63063	2.51142
40	0.00431	0.01186	0.03329	0.04230	0.11336	0.22583	0.38264	1.41606	2.24328

Table 3: *The running time results for CUDA version* ***Note:*** *The unit is second in scientific notation.*

#thread	100	400	800	1000	2000	3000	4000	8000	10000
1	0.00376	0.04347	0.18125	0.27513	1.05216	2.39160	4.20471	16.84305	26.39679
2	0.00425	0.03247	0.10253	0.15540	0.56450	1.25592	2.22376	8.90606	14.09925
4	0.00268	0.02103	0.06758	0.09268	0.30501	0.66781	1.18926	4.57073	7.33884
8	0.00201	0.01365	0.04480	0.05642	0.17902	0.38132	0.64342	2.52530	3.92990
12	0.00186	0.01206	0.03152	0.05091	0.13874	0.28748	0.47667	1.90291	2.89396
16	0.00186	0.01097	0.03162	0.04017	0.11774	0.23308	0.40013	1.57791	2.40306
24	0.00249	0.00922	0.02643	0.02997	0.10393	0.19172	0.34179	1.28052	1.96857
32	0.00279	0.00872	0.02534	0.03017	0.09298	0.16957	0.28278	1.10986	1.69952
40	0.00322	0.00822	0.01894	0.02569	0.08395	0.16594	0.26642	1.02701	1.56432