CSC4005-Assignment-3 Report

Name: HuangPengxiang

Student ID: 119010108

Objective: This Assignment requires to four parralle version of nbody computation program using MPI, Pthread, CUDA, and OpenMP. The bonus part is use MPI with OpenMP to implement parallel computation. Also, The program should be tested in the cluster in order to get the experimental data. The experiment analysis session and conclusion session which concludes the comparison result is necessary to be included in this report as well.

Introduction

- In physics, the n-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other, they are all moving by obeying the Newtons law. An N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, and every body will be inlfuenced by other bodies and only be influenced by the gravity force. The gravity between N-body should be described by the follwing equation: $F = G\frac{m_1 \times m_2}{r^2}$. Also, in this simulation, the size is not infinite, so the collision beween each body and bouncing from the boundary should also be considered.
- However, If we apply sequential method to calculate it, the compuation time will be quite large with the number of body increase. Theoretically, Sequential computation will calculate gravity between each body and movement one by one if we pick a fixed body number for a specific image size. Therefore, This program aims to

investiaget different version of Parrallel computing to optimize the compuatation speed of Nbody calculation. In this experiment, I designed **Five parrallel computing version including MPI , Pthread, CUDA, OpenMP, MPI+OpenMP** and compare those version with each other to find out the best one in computing performance for different scenario.

- The basic problems and tasks are: given a number of body's location and initial speed, and the program should perform the parrallel computing strictly following the rule of Newtons law calculation for whatever the size of bodies and number of cores/threads. During the process, the collision and bouncing should also be included to change the speed and acceleration. Running the parrallel computation program in the cluster to get the experimental data and analyze the performance for specific situation such as different cores/thread and different input size.
- The rest of report is mainly containing Four part: **Running Guidance , The Design Approach, Performance Analysis** , and **Conclusion**. The Running Guidance shows you The files i include to finish the project and how to run it on the server. It Also includes a script file to show you how I get my experimental data on the server. The Design Approach mainly tells How I design my five version of parralle computation for specific motivation and the methods to implement those code. The Performance Analysis is the comparision of performance between those approach implementing the Mandelbrot computation. I set different variable for those program to observe in what scenario which one will perform better. The Conclusion part is focused on the result of analysis, and give the table of analysis results between those Mandelbrot set computing program.

Running Guidance

My program inlcude the file **MPI_Version, Pthread_Version, CUDA_Version, OpenMP_Version, MPI+OpenMP_Version**, and **Appendix** which contains the experimental data and some analysis pictures.

To build those five versions of parallel computation, basically you should follow the instruction below:

For MPI, Pthread, OpenMP, MPI+OpenMP:

```
$ cd /* The location of one of these version */
$ mkdir build && cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Debug
$ cmake --build . -j4
```

For CUDA:

```
$ mkdir build && cd build
$ source scl_source enable devtoolset-10
$ CC=gcc CXX=g++ cmake ..
$ make -j12
```

To test my program and see the graphic ouput, you need to follow the instructions below:

For MPI Version

if you want to test my program in your own computer, you may need to

```
mpirun -np num_of_cores ./csc_4005_imgui
#such like mpirun -np 3 ./csc_4005_imgui
```

if you want to test my program in cluster, I also designed a running script which located in `/Project3/Appendix/`, you can found a script named mpi_3.sh and use it to test my program. You need to use `sbatch` command to submit my program to cluster, and use `xvfb-run` to see the terminal output. However, in this way, you can not see my graphic output. Since salloc command somehow didn't work on cluster for my computer, I only use sbatch to check the terminal output.

```
# submit the work to cluster before you test
$ cd /* where mpi_3 located */
$ sbatch mpi_3.sh # I set the time limit is 1 minitue, you can see the output in a short time
$ cat slurm.out
# You may need to type control + c to end the program since it is the dead loop
```

For Pthread Version && OpenMP Version

You may need to add one more argument to test my program, I ask user to input the number of thread before running it. You could aslo use `xvfb-run` to close the graphic output.

```
./csc_4005_imgui 16 # means the number of thread in my progam will be 16
```

For CUDA Version

You also need to add one more argument to test my program, I ask user to input the number of thread before running it. You could aslo use `xvfb-run` to close the graphic output.

```
# you have to salloc for CUDA, or you may unable to run my program.
$ salloc -n1 -t10
$ srun /csc4005_imgui 16 # means the number of thread in program will be 16
```

For MPI+OpenMP Version

You should add one more argument after normal `mpirun`. I ask user to input the number of thread and number of cores in the program, you may also use `xvfb-run` to close the graphic output.

```
$ mpirun -np num_of_cores ./csc_4005_imgui num_of_thread_each_core
# such like mpirun -np 3 ./csc_4005_imgui 8 means there are 3 cores totally and each core will
use 8 thread to parallel computation
```

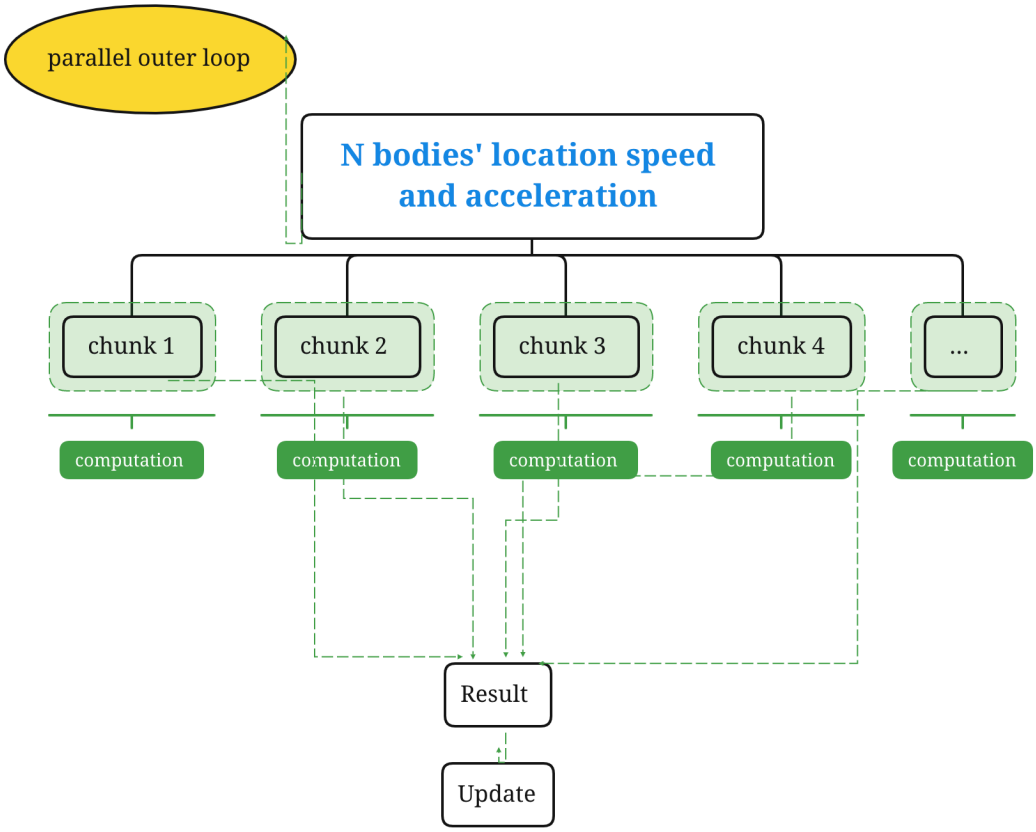
IMPORTANT Declaration

when running my CUDA_Version, you may have the chance to encounter the problem **malloc falied**. When I finish my program and test it 5 days before deadline, It all works good, but when I run it one day before deadline with the same porgram, it suddenly out of work and tell me it could not malloc the memory for cuda and failed at `cudaMallocManaged` command, which I use it to store and send the information to device for calculation. And at that time the cluster is so crowd and I guess there are no sufficient space for me to malloc such a huge memory in GPU since when I try to calculate the 3200 bodies, the malloc memory will increase dramatically. And I try it on another day, it still works good. Hence, when you encounter the malloc problem, retry it or don't use too large body size.

Design Approach

Parallel Computing Basic Design

The main task is to calcualte every bodies location, speed, and acceleration in the fixed size and update it after calculation. If we assume there are N bodies in the fixed space, then we have $\frac{n(n+1)}{2}$ of calculation needed to be done. after $\frac{n(n+1)}{2}$ then need to update the location. It means that we have **two for loop** for the whole sequential computation. In this project, I choose the **outer loop** to parallel computing it.



Code Design

From the previous fogure, I found that if we follow the computation, and distribute every task based on the outer for loop. **The task won't be evenly distributed** sincce they have different inner loop. some thread/core which have small rank number may have many tasks while those thread/core which have big rank number will take much less tasks to compute. So it may influence the parallel computing. But we **can not change every inner loop to the same iteration n**, it may slow down all the compuation and won't improve the parallel efficiency compared with sequential one.

Also, A serious problem will occur when we apply parallel compuation, which is **Data Race**. A data race occur when two threads try to access a same variable and at least one of the threads want to write to that variable. In this problem, The data race may **occur in two place**:

- one is when two thread modify the same body's location, speed, and acceleration. It may be belated to deal with the collision and bouncing.
- The other one is when one thread already finish the speed calculation and goes into the loop for distance

update, while another thread is still computate and updat e the same body's speed or other Information. It occur basically **because the task is not evenly distribute**, some rank may take very little time to goes into distacne updating `for loop`.

Data race will seriously influence the N-body simulation. Hence, I **sloved the data race for every version I write.**

MPI Design

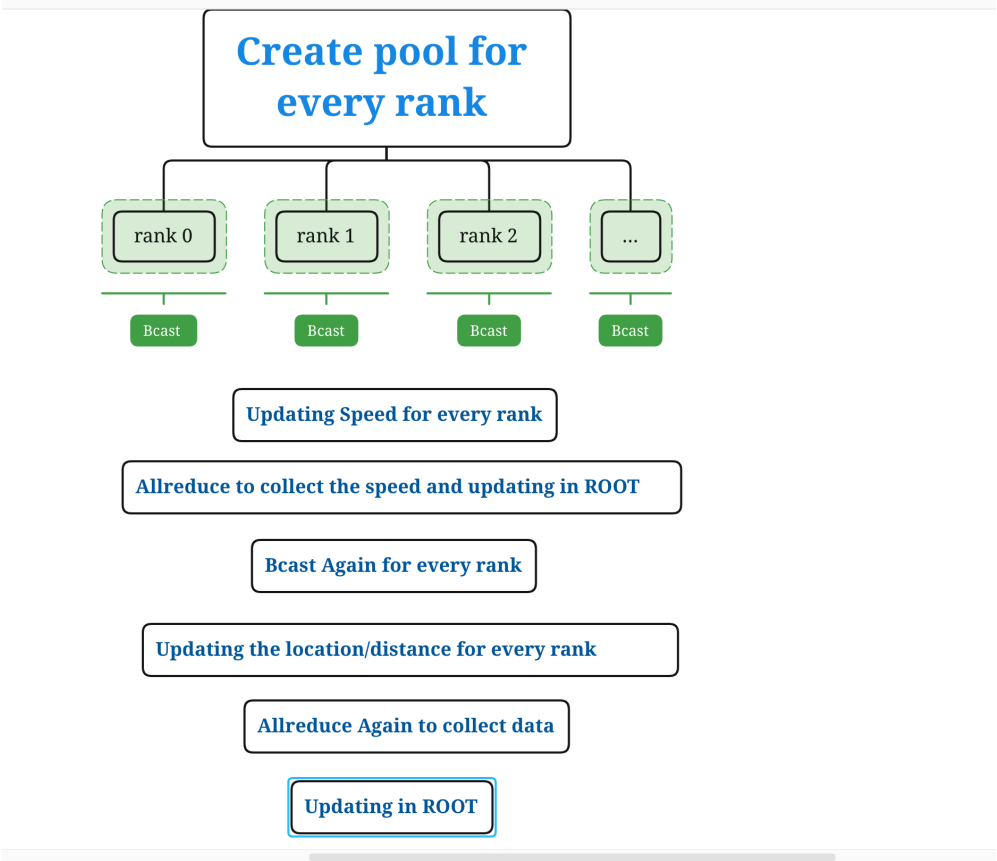
To design parallel computation via MPI, it mainly compose of 8 parts:

- **Creation:** create a pool for every rank
- **Bcast:** board cast all the information for every rank's pool, make them have all the information the same in theri body pool.
- **Calculation 1**: calculate the speed in every rank and update thier speed and acceleration.
- **Allreduce**: use allreduce function to collect all the speed, acceleration, and distance, then update it.
- **Bacast again**: boardcast again to update the speed in each rank.
- **Calculation 2**: calculate the distance this time, and update each body's distance
- **Allreduce again**: allreduce function is used again to collect and gather each body's information.
- **Update**: Update all the information in ROOT.

Data Race handle:

Since MPI use its own memory in its core, **bascically there won't be data race in MPI version since they calculate their speed and distance separately.**

The reason why I divide the speed calculation and distance updating into two parts is to make sure there is no data race between each rank. when one rank try to update the distance and boncing or collision happen because the spific speed while other rank change its speed and the collision or boncing won' t happen. The data race occur. Hence, I parallel change the speed and collect it, then parallel change the distance for evry body.



-- Bcast:

I create a struct to pass the basic pool information to other ranks, such like space, bodies number, elapse and so on, which is more convenient to pass them one by one.

The struct I create:

```
struct params_float{
    float gravity;
    float space;
    float radius;
    float elapse;
    float max_mass;
}__attribute__((packed));
```

I changed the some vertor lik pool.x from private to public, where I could directly change it in main function. And I could use it to Bcast all the information in ROOT.

The information I Bcast:

```
MPI_Bcast(&pool.x[0], pool.x.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.y[0], pool.y.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.vx[0], pool.vx.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.vy[0], pool.vy.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.ax[0], pool.ax.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.ay[0], pool.ay.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&pool.m[0], pool.m.size() , MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

-- Allreduce:

function `Allreduce` is used to sum all of the value from each rank to a specific location. I use it to sum all of the change of speed in every rank. and I reduce (n-1)* `previous value` is the result of the updating one.

```
 /* allreduce function to calculate the sum of every value */
  MPI_Allreduce(&pool.x[0], &recv_x[0],pool.x.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pool.y[0], &recv_y[0],pool.y.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pool.vx[0], &recv_vx[0],pool.vx.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pool.vy[0], &recv_vy[0],pool.vy.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pool.ax[0], &recv_ax[0],pool.ax.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  MPI_Allreduce(&pool.ay[0], &recv_ay[0],pool.ay.size(), MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

/*  calculate and update the final value, since allreduce calculate the sum, hence we minus (n-
1)*original value  */
  for (size_t i = 0; i < pool.size(); i++){
      pool.x[i] = recv_x[i] - (num_processes - 1) * copy_x[i];
      pool.y[i] = recv_y[i] - (num_processes - 1) * copy_y[i];
      pool.vx[i] = recv_vx[i] - (num_processes - 1) * copy_vx[i];
      pool.vy[i] = recv_vy[i] - (num_processes - 1) * copy_vy[i];
      pool.ax[i] = recv_ax[i];    // we do not reduce a since every a goes into function will
be 0;
      pool.ay[i] = recv_ay[i];
  }
```

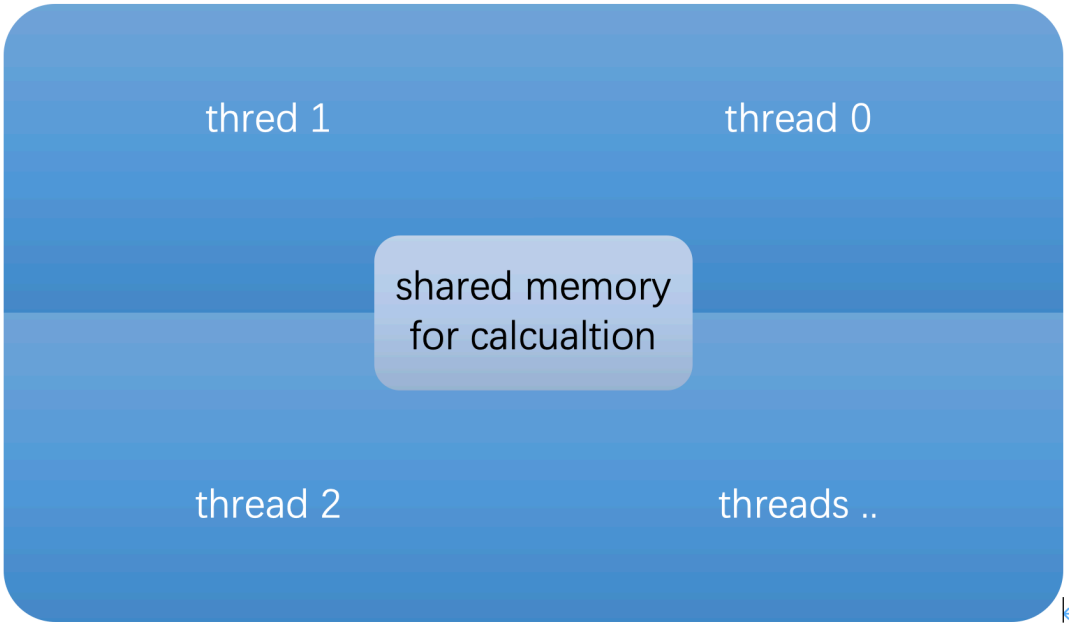
-- Final update:

after calculate the speed and distance, we final update the distance, speed and acceleration for ROOT. Note here: we don't update the a, since every time when goes into `update_speed` function, it wll assign a to zero.

```
/* assign the final value for root */
for (size_t i = 0; i < pool.size(); i++){
    pool.x[i] = recv_x[i] - (num_processes - 1) * copy_x[i];
    pool.y[i] = recv_y[i] - (num_processes - 1) * copy_y[i];
    pool.vx[i] = recv_vx[i] - (num_processes - 1) * copy_vx[i];
    pool.vy[i] = recv_vy[i] - (num_processes - 1) * copy_vy[i];
}
```

Pthread Design

For implementation via Pthread, I aslo divide the bodies into different chunks. But unlike MPI Version, since Ptrhead can share the memory bettween every thread. in this version, there is no need to boardcast the information from root to each part and send and receive the information between each part.



To design parallel computation via Pthread, it mainly compose of 3 parts:

- Pthread Creation
- Pthread Calculation in Shared Memory
- Pthread Join

-- Pthread Creation:

create the mutex to avoid data race. In creation part, initialize the mutex and create the thread.

```
for (int j = 0; j < 2000; j++) {
    pthread_mutex_init(&mutex[j], NULL);
}

for (int j = 0; j < threadnum; j++) {
    pthread_create(&threads[j],&attr,Pthread_update,(void*)&thread_paras[j]);
}
```

-- Pthread Calculation:

How I avoid data race:

I set a mutex for every body, when some thread is modifying this body, it will locked until it finish its job, next thread wiill continue to modify this body.

```
 for (size_t i = min; i < max; ++i) {
    for (size_t j = i + 1; j < pool->size(); ++j) {
        pthread_mutex_lock(&mutex[i]);
        pthread_mutex_lock(&mutex[j]);
        check_and_update(pool->get_body(i), pool->get_body(j), radius, gravity);
        pthread_mutex_unlock(&mutex[i]);
        pthread_mutex_unlock(&mutex[j]);
    }
}
```

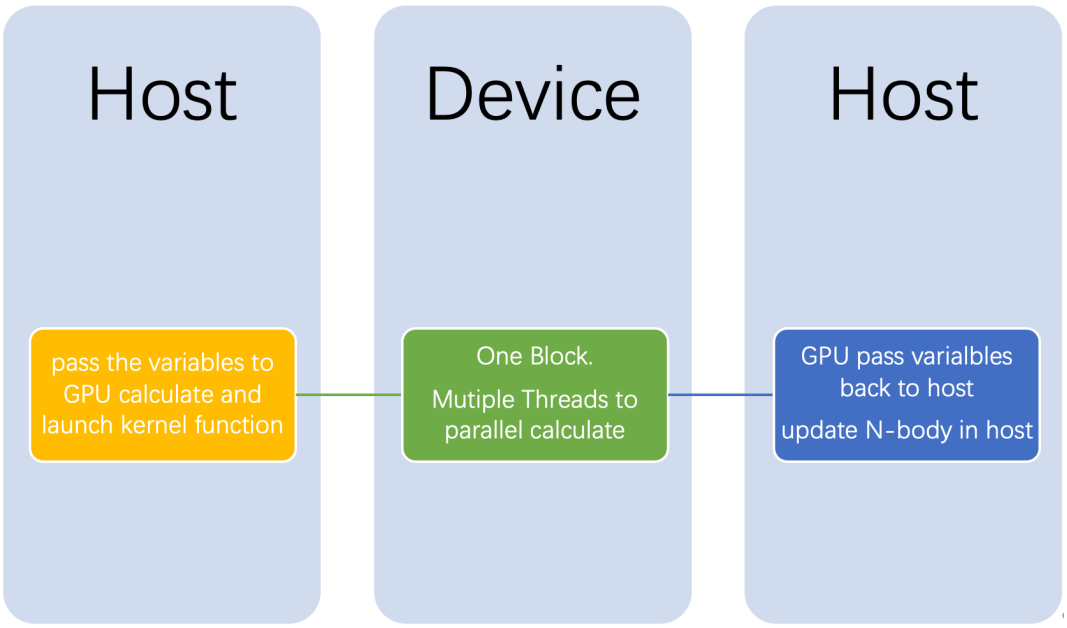
-- Pthread Join:

Join all the thread and after calculation done, destroy them.

```
for (int i = 0; i < threadnum; i++){
    pthread_join(threads[i],NULL);
}
pthread_attr_destroy(&attr);
free(threads);
```

CUDA Design

Unlike Pthread and MPI, CUDA implement the parallel computation in GPU instead of CPU. And basically, CUDA have very high speed calcualtion but CUDA has very limited fucntion to use since GPU do not support high level calculation from a hardware point of view. Hence, the CUDA design is more difficult than others.



To design parallel computation via MPI, it mainly compose of 3 parts:

- **Initialisation**: Launch the kernel and send the variables need to calculated from host to device.
- **Caculation:** GPU parallel calculate the result
- **Send back:** send the information back and update the N-body pool.

-- Initialisation:

I used `cudaMallocManaged` function to malloc the array to store the information like speed, acceleration, distance, that is mainly becuase, first the CUDA can not use `std::vector` directly, so I can not directly use the original function in `body.hpp`. second, `cudaMallocManaged` command will malloc the space where is accessible for both host and device. after that, I lauch the kernel function and pass the arguemnt in it to directly use.

```
/* malloc the memory */
cudaMallocManaged(&device_m,sizeof(double) * pool.m.size());
cudaMallocManaged(&device_x,sizeof(double)* pool.m.size());
cudaMallocManaged(&device_y,sizeof(double)* pool.m.size());
cudaMallocManaged(&device_vx,sizeof(double)* pool.m.size());
cudaMallocManaged(&device_vy,sizeof(double)* pool.m.size());
cudaMallocManaged(&device_ax,sizeof(double)* pool.m.size());
cudaMallocManaged(&device_ay,sizeof(double)* pool.m.size());

/* lauch the kernel */
mykernel<<<1, threadnum>>>(space,gravity,radius,elapse,max_mass, bodies, pool.size(),
threadnum,
                                                device_m, device_x, device_y,device_vx, device_vy,
device_ax, device_ay );
```

-- Calculation:

since CUDA can not use `std::vector` directly, then I **rewrite the funtion** for CUDA to update the speed, acceleration and distance **based on** `array`.

```
/* rewrite the function for cuda to compute */

__device__ void CU_check_and_update(size_t i, size_t j, double radius, double gravity,
     double * device_m, double * device_x,double * device_y ,double * device_vx, double *
device_vy, double * device_ax, double * device_ay )

__device__ void CU_handle_wall_collision(size_t i, double position_range, double radius,
     double * device_m, double * device_x,double * device_y ,double * device_vx, double *
device_vy, double * device_ax, double * device_ay )

__device__ void CU_update_for_tick(size_t i , double elapse, double space, double radius,
     double * device_m, double * device_x,double * device_y ,double * device_vx, double *
device_vy, double * device_ax, double * device_ay )
```

Moreover, since CUDA do not have mutex operation function, then **I design a mutex to avoid the data race**. actually, I designa **atomic muetx:**

```
/* lock */
__device__ void CU_lock(size_t i ){
    bool flag = true;
    do{
        flag = atomicCAS(& ( culock[i] ),0,1);
        __threadfence_system();
    }while(flag);
```

```
}

/* unlock */
__device__ void CU_unlock(size_t i){
    atomicCAS(&(culock[i]),1,0);
    __threadfence_system();
}
```

OpenMP Design

The OpenMP design is almost the same as Pthread version, since they are all shared by one core memory. The design of them are very simialr.

And OpenMP is also very esay to implement since it will automatically allocate thread for you, the only thing you need is just to write

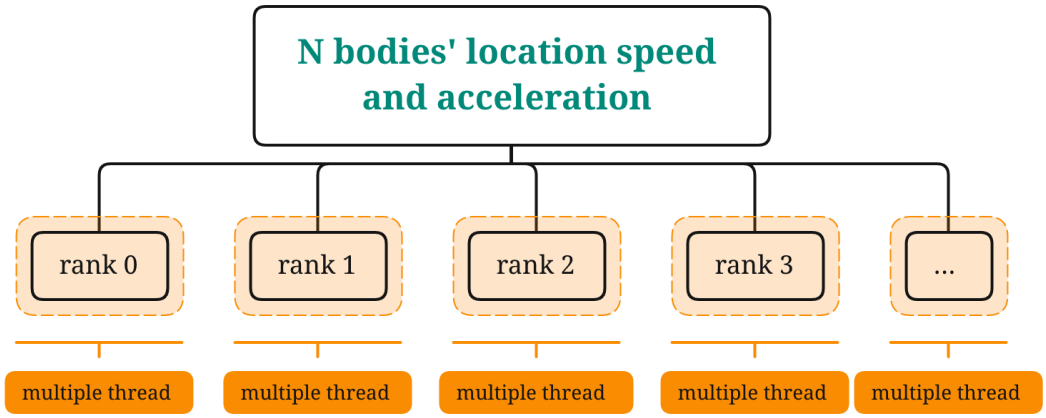
```
 #pragma omp parallel num_threads(threadnum)

for (size_t i = min; i < max; ++i) {
  for (size_t j = i + 1; j < size(); ++j) {
      /* set a mutex to avoid the data race */
      omp_set_lock(&mutex[i]);
      omp_set_lock(&mutex[j]);
      check_and_update(get_body(i), get_body(j), radius, gravity);
      omp_unset_lock(&mutex[i]);
      omp_unset_lock(&mutex[j]);
  }
}

// set a barrier to avoid data race
#pragma omp barrier

for (size_t i = min; i < max; ++i) {
      get_body(i).update_for_tick(elapse, position_range, radius);
}
```

How I avoid data race

I set a `lock` and `barrier` to void the data race. Each thread will parallel update the speed and waiting other thread to finish. after they all reach the barrier, then they will parallel update the distance as well. when they are updating the speed, I also set a muetx to each body to avoid the data race. Every body at each time will only be modified by once.

MPI+OpenMP Design

The bonus part is the combiantion of MPI and OpenMP, which means firstly distribute the tasks to different cores, and in each core, also apllied OpenMP to create mutiple thread to parallel compute it.



How to avoid data race here?

it is the same as OpenMP one since **The data race won't occur in different cores**. Then we can also add mutex after we lauch the multiple thread.

Experiment Design

Totally 5 Topic I investigated in this project:

- **Sequential vs Parallel**
- **Number of cores/thread**
- **Pthread vs OpenMP vs CUDA vs MPI**
- **OpenMP+MPI vs MPI vs OpenMP**

-- Design Sequential vs Parallel

To compare the sequential and parallel computing in a broader way, I set a different choice in size and number of cores/thread to see which one performs better under some situation. The choices for body size is (20, 100, 200, 500, 1000, 2000, 5000). The choice for number of thread/cores is (1,2,4,8,16,32,64). And I let #core/#thread = 1 be the sequential one, and let others to compare.

To compare their Performance more clearly, I use the speed up (speed of pthread computing/ sequential computing) as the criterion .Then, the observations focus on:

- Whether parallel version has a better performance than sequential one under different situations.
- How much can the parallel one imporve the performance.

-- Design Number of cores/thread

To design this experiment, I first fixed the problem size equals to middle size 100, and use different number of cores to test the performance under different thread. In MPI version, I write a script to test Performance in static scheduling.

```
#! /bin/bash
for (( j = 0; j <= 20; j++ ));
do
xvfb-run mpirun -n $j /pvfsmnt/119010108/Project_2/Pthread_Version/build/csc4005_imgui
done
```

And we aim to found out whether when the cores/thread increase, the speed of parallel computing will also increase, or it may occur other relationship.

The goal is to found out that:

- The speed tendancy during the thread/core change
- The major reason for the speed decrease when size decrease

-- Design Pthread vs OpenMP vs CUDA

The reason to design this experiment is used for compare those parallel version used **multiple threads** to compute. To set experiment in a more clear way. I choose (20,200,2000) bodies size to represent small middle, and large body size. And I also set number of threads ranged from (2,4,8,16,32,64) to compare those performance. This design is used to found out: If we apply the same condition in **thread parallel computing**, which one could be better under some situation. And What's the difference between them and why it happen. I design this experiment is used for compare those parallel version used by **cores** and **threads**, and the diffenrence between those two verison. And the parallel difference betweem **distributing tasks in CPU and distributing tasks in GPU**. So I set body size is (20 200, 1000, 2000) to represent the size range and I set numer of cores/threads is (2,4,8,16,32,64) for this experiment. The goal is to found out: The goal is to

- Under what situation which version performs better
- The major reason why this model performs worse in som situation
- Under what situation CPU distribution is better than GPU distribution
- The major reason why this model performs worse in some situation

-- Design OpenMP+MPI vs MPI vs OpenMP

This is used for compare the performance when we both apply openmp and MPI in the same time. which means we both allcoate many cores for tasks and also parallel finish their job in differene threads. The main goal for this design is to find out:

- The tendancy when we add OpenMP in MPI
- The reason why have this chanege

Performance Analysis

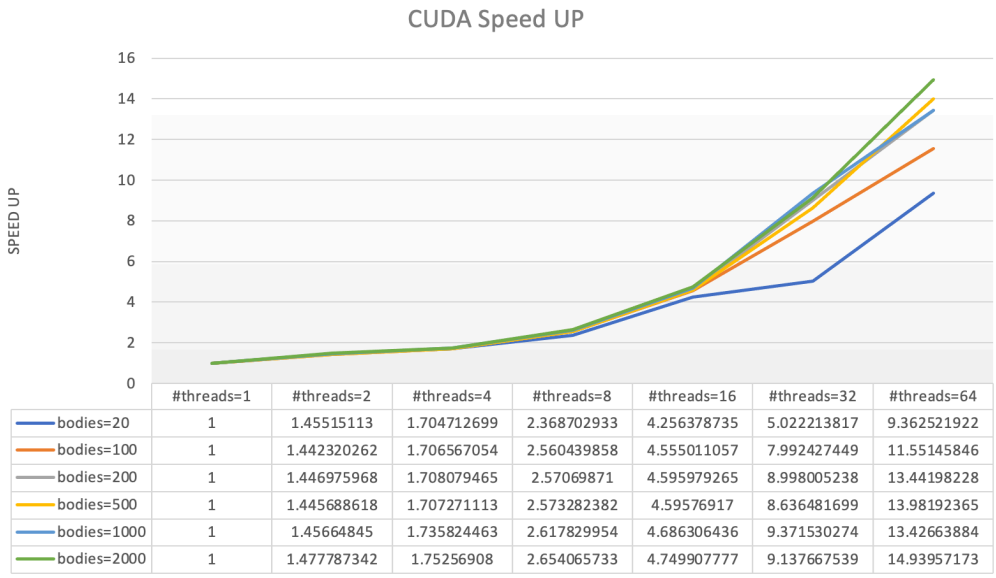
Sequential vs Parallel

For CUDA and OpenMP: (The Reason I choose these two is because the comparision between paralllel and sequential in MPI and Pthread are already done in Project 1 and Project 2)

-- **CUDA Result:**

Speed up	increase						
4	#threads=1	#threads=2	#threads=4	#threads=8	#threads=16	#threads=32	#threads=64
9 bodies=20	1	1.45515113	1.704712699	2.368702933	4.256378735	5.022213817	4.362521922
0 bodies=100	1	1.442320262	1.706567054	2.560439858	4.555011057	7.992427449	11.55145846
7 bodies=200	1	1.446975968	1.708079465	2.57069871	4.595979265	8.998005238	13.44198228
2 bodies=500	1	1.445688618	1.707271113	2.573282382	4.59576917	8.636481699	13.98192365
bodies=1000	1	1.45664845	1.735824463	2.617829954	4.686306436	9.371530274	13.42663884
bodies=2000	1	1.477787342	1.75256908	2.654065733	4.749907777	9.137667539	14.93957173

Then, I draw a figure to see it more clear:



	#threads=1	#threads=2	#threads=4	#threads=8	#threads=16	#threads=32	#threads=64
bodies=20	1	1.45515113	1.704712699	2.368702933	4.256378735	5.022213817	9.362521922
bodies=100	1	1.442320262	1.706567054	2.560439858	4.555011057	7.992427449	11.55145846
bodies=200	1	1.446975968	1.708079465	2.57069871	4.595979265	8.998005238	13.44198228
bodies=500	1	1.445688618	1.707271113	2.573282382	4.59576917	8.636481699	13.98192365
bodies=1000	1	1.45664845	1.735824463	2.617829954	4.686306436	9.371530274	13.42663884
bodies=2000	1	1.477787342	1.75256908	2.654065733	4.749907777	9.137667539	14.93957173

-- Discussion:

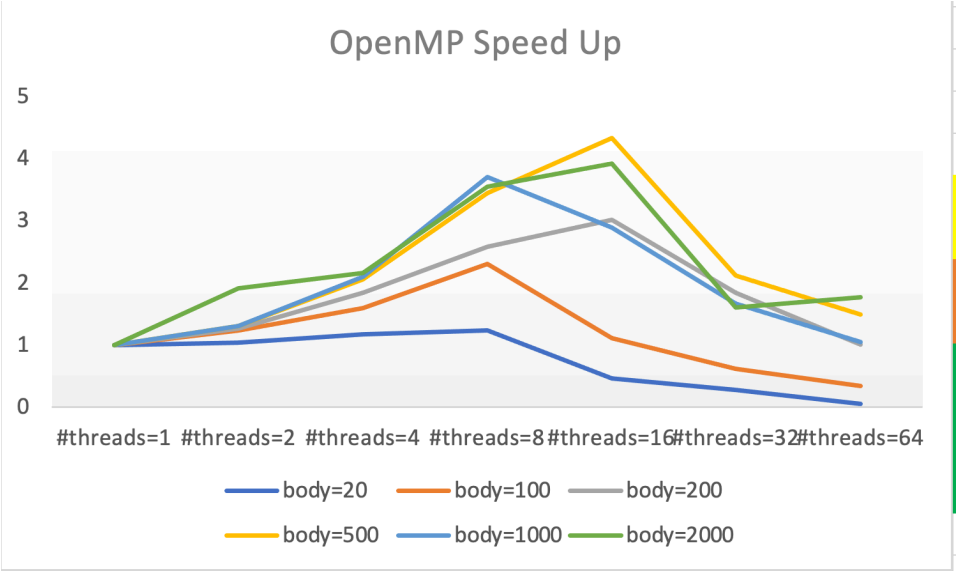
- In my program, I found that the parralle version is better than sequential one for every case I had choose **in CUDA**. From the chart I just show, The speed up is always greater than 1 and it gradually increase as #thread and size increase. That is because when the number of thread increase, the task will be less for each thread, and also the thread share one memory so that they do not need to pass the information to each other. Hence, The parallel pthread computation has the better performance than sequential one
- In CUDA, . it enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation. The parallel section of the whole problem takes more significant role in the whole computing procedure. Therefore, distributed the large scale work to each treads would distribute the workload and reduce the total computing time, especially when the size is bigger.
- The multi-thread computing is better than sequential computing especilally when problem size is larg. When the numberof threads is larger and problem size is greater, the advantage of multi-thread computing is more obvious.

-- OpenMP Result

The Green Part reprent the parallel version is better while the red one represent the sequential one is better.

Speed Up	#threads=1	#threads=2	#threads=4	#threads=8	#threads=16	#threads=32	#threads=64
body=20	1	1.041003809	1.168217249	1.236954207	0.463071863	0.274878713	0.052331017
body=100	1	1.231418919	1.591842082	2.304628225	1.112297833	0.621589359	0.340190021
body=200	1	1.261106015	1.839881464	2.578989682	3.01677116	1.846589274	1.004252433
body=500	1	1.302772128	2.052986664	3.448049992	4.331709624	2.115286021	1.493339855
body=1000	1	1.30580248	2.100414151	3.700252711	2.88921422	1.668372738	1.051660923
body=2000	1	1.914228562	2.16255144	3.550329018	3.91610937	1.60798585	1.770704297

I also draw a picture to see it more clearly:



-- Discussion:

- We can see The parallel OpenMP performs better than sequential when the thread number are not too large and the body size is also not too small. This is beacuse OpenMP create multiple thread to compute the body location at the same time. which will increase the speed of calculation.
- However, When the thread number dramatically increase while the body number is very small, The OpenMP will perform worse under this situation. **Especially when the number of thread is greater than 32**. That is because for each core in the cluster, it only have 32 cpu to execute the program. Which means if there are number of the threads which is greater than 32, there are only 32 threads execute program simultaneously. Hence, When the size is too small, the proragm may don't need those thread to calculate, but those threads were created and wait other threads to execute, which will consume a lot of time, and it may cause the parallel version performs worse than sequential one.

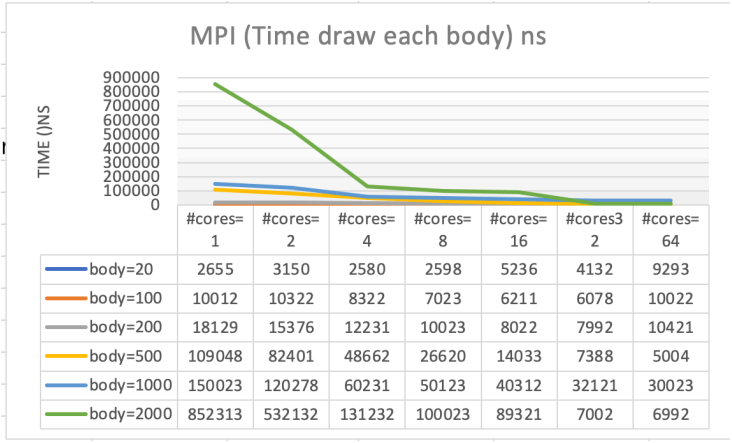
Number of cores/thread

-- Result:

For MPI Version:

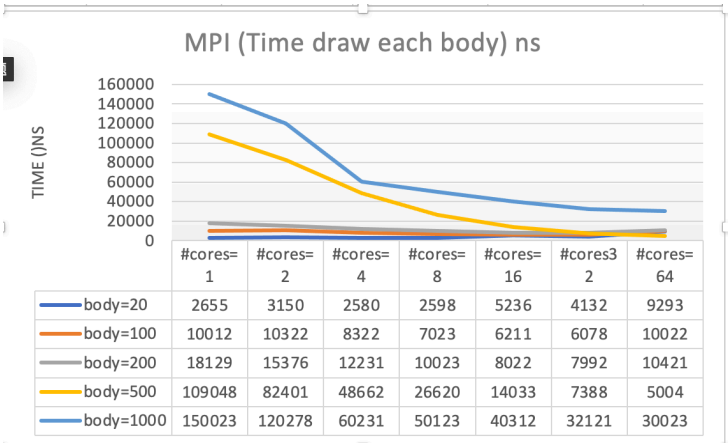
The y-ordinate represent the time that average time to draw each body.

The figure shows that as the number of threads increases (from 1 to 64) overall, the performance would increase dramatically from number of core 4, and the time for each body will derease dramatically, nearly like exponentially increase function if choose the small size. But in this figure, it is not clear for the largsize speed during core number change.



	#cores=1	#cores=2	#cores=4	#cores=8	#cores=16	#cores32	#cores=64
body=20	2655	3150	2580	2598	5236	4132	9293
body=100	10012	10322	8322	7023	6211	6078	10022
body=200	18129	15376	12231	10023	8022	7992	10421
body=500	109048	82401	48662	26620	14033	7388	5004
body=1000	150023	120278	60231	50123	40312	32121	30023
body=2000	852313	532132	131232	100023	89321	7002	6992

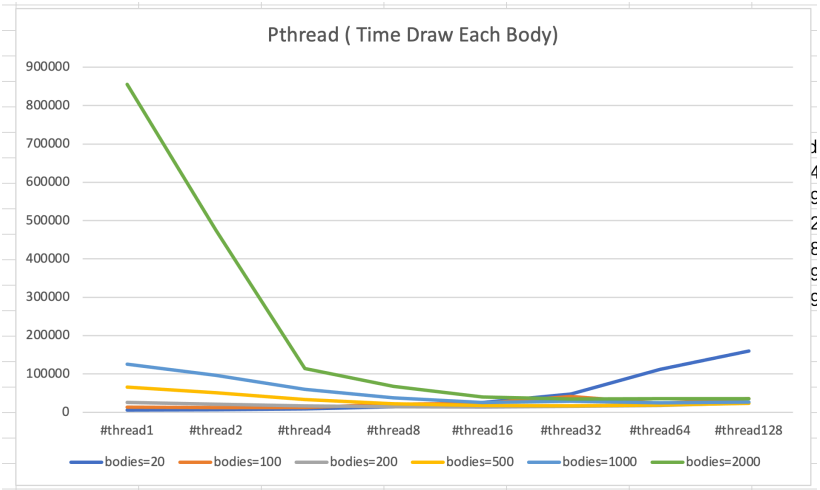
See the core influence in small size more clearly, The figure bleow shows the tendancy of MPI-Cores in large problem size. We can find that if the size is larger, the influence of number core will reduce. Even when the size is extremely large (such as 2000), The speed will have only little improvement in speed.



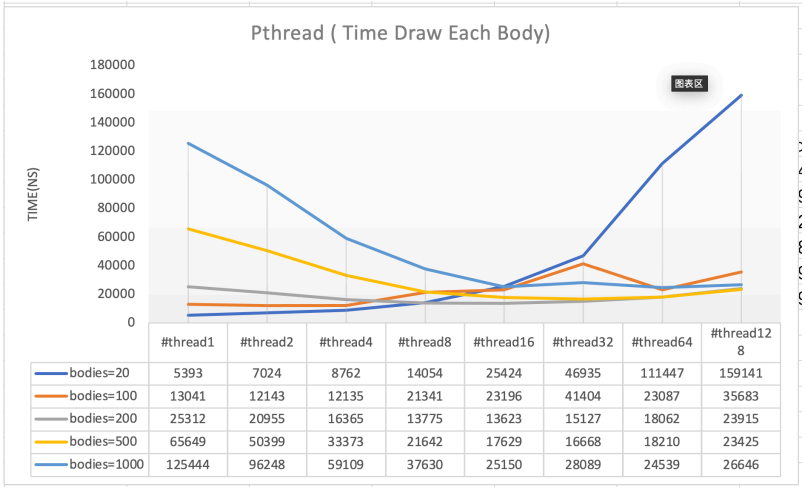
	#cores=1	#cores=2	#cores=4	#cores=8	#cores=16	#cores32	#cores=64
body=20	2655	3150	2580	2598	5236	4132	9293
body=100	10012	10322	8322	7023	6211	6078	10022
body=200	18129	15376	12231	10023	8022	7992	10421
body=500	109048	82401	48662	26620	14033	7388	5004
body=1000	150023	120278	60231	50123	40312	32121	30023

For Pthread Version:

The Result is showing below:



To see it more clearly, I Move the large body size to make it more clear to see.



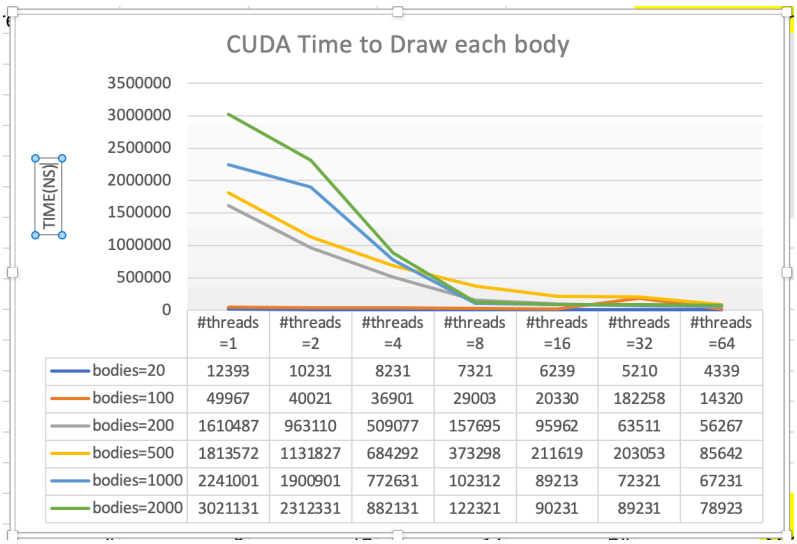
	#thread1	#thread2	#thread4	#thread8	#thread16	#thread32	#thread64	#thread128
bodies=20	5393	7024	8762	14054	25424	46935	111447	159141
bodies=100	13041	12143	12135	21341	23196	41404	23087	35683
bodies=200	25312	20955	16365	13775	13623	15127	18062	23915
bodies=500	65649	50399	33373	21642	17629	16668	18210	23425
bodies=1000	125444	96248	59109	37630	25150	28089	24539	26646

This figure shows that:

- For the small problem size, when the thread number increase, it will reduce first and then increase dramatically, that is because if the problem size is too small and the the number of core in one node is limited, and when the thread reach the max number of core, other thread wil**l wait until OS allocate a available core to it for calculation.**
- For the large problem size, there still have some limited cores, but the problem size is too big, hence the major influence here is the problem size other than thread number. so when increasing the number of cores, the time to draw each body will decrease at the same time.

For CUDA Version:

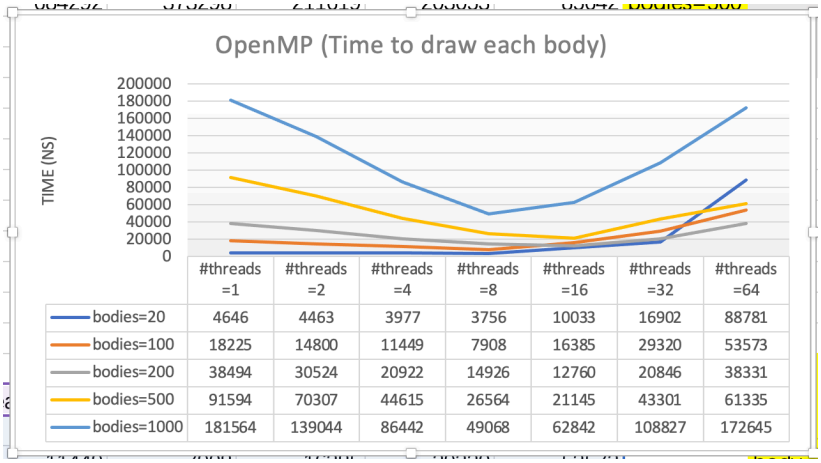
The Result is showing below:



	#threads =1	#threads =2	#threads =4	#threads =8	#threads =16	#threads =32	#threads =64
bodies=20	12393	10231	8231	7321	6239	5210	4339
bodies=100	49967	40021	36901	29003	20330	182258	14320
bodies=200	1610487	963110	509077	157695	95962	63511	56267
bodies=500	1813572	1131827	684292	373298	211619	203053	85642
bodies=1000	2241001	1900901	772631	102312	89213	72321	67231
bodies=2000	3021131	2312331	882131	122321	90231	89231	78923

This figure shows that The CUDA will consistently increase the speed when increasing the number of threads, It won't decrease like Pthread when the numb er of threads increase while the problem size is too small.

For OpenMP Version:



	#threads =1	#threads =2	#threads =4	#threads =8	#threads =16	#threads =32	#threads =64
bodies=20	4646	4463	3977	3756	10033	16902	88781
bodies=100	18225	14800	11449	7908	16385	29320	53573
bodies=200	38494	30524	20922	14926	12760	20846	38331
bodies=500	91594	70307	44615	26564	21145	43301	61335
bodies=1000	181564	139044	86442	49068	62842	108827	172645

From This figure, we could observe that the figure is moe like the pthread one, when the small problem size, when the thread number increase, it will reduce first and then increase dramatically, For the large problem size, there still decrease the draw body time.

-- Discussion:

- For Pthread and OpenMP, when the number of the cores/thread increase, the speed will decrease first and when they meet number of threads reach 32. The speed will increase afrer that. hat is because for each core in the cluster, it only have 32 cpu to execute the program. Which means if there are number of the threads which is greater than 32, there are only 32 threads execute program simultaneously. Hence, When the size is too small, the proragm may don't need those thread to calculate, but those threads were created and wait other threads to execute, which will consume a lot of time, and it may cause the parallel version performs worse than sequential one. But For the large size, The Pthread and OpenMP will also increase the speed as the number of threads increase.
- For CUDA, The speed will consistently increase and it won't reach the threashhold. That is because CUDA use GPU to calculate, and GPU has many cores while CPU have only 32 in cluster. So when we increase the number of threads, the speed will consistently increase at this time.
- For MPI, if the size is larger, the influence of number core will reduce. Even when the size is extremely large (such as 2000), The speed will have only little improvement in speed. That is because when the size is too small, the communication between each core is too little and the speed is fast. However, when the size increase, the Communication increase and it is very time consuming hence it will decrease instead.
- 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. It means there must be a critical point for the improvement to reach ceasation. Also compared with another problem: Odd- even sort transposition sort, whose performance may decrease as the number of processes is relative large, parallel Mandelbrot set computation's performance will cease but not decrease. This is because there are little communication between threads in this problem (all threads perform its work independently). Thus the communication overhead would not be significant when the number of threads increase.

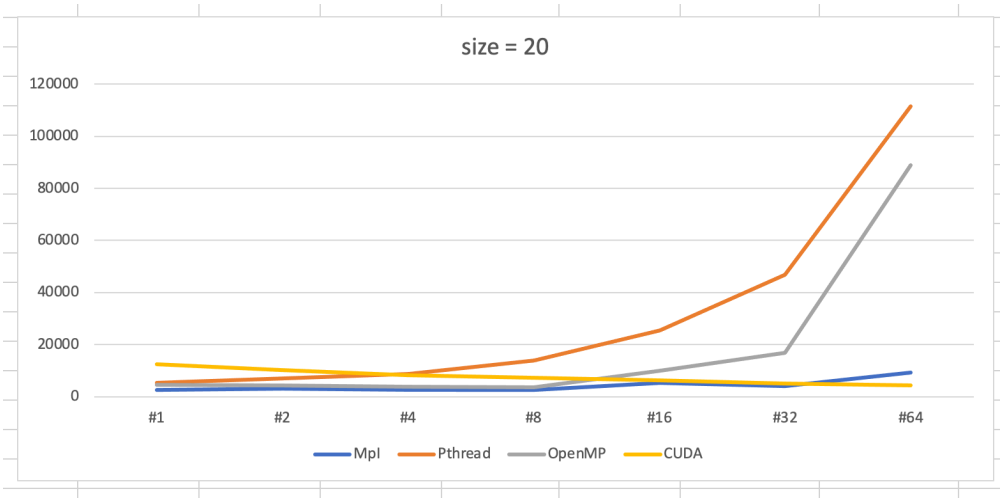
Pthread vs MPI vs OpenMP vs CUDA

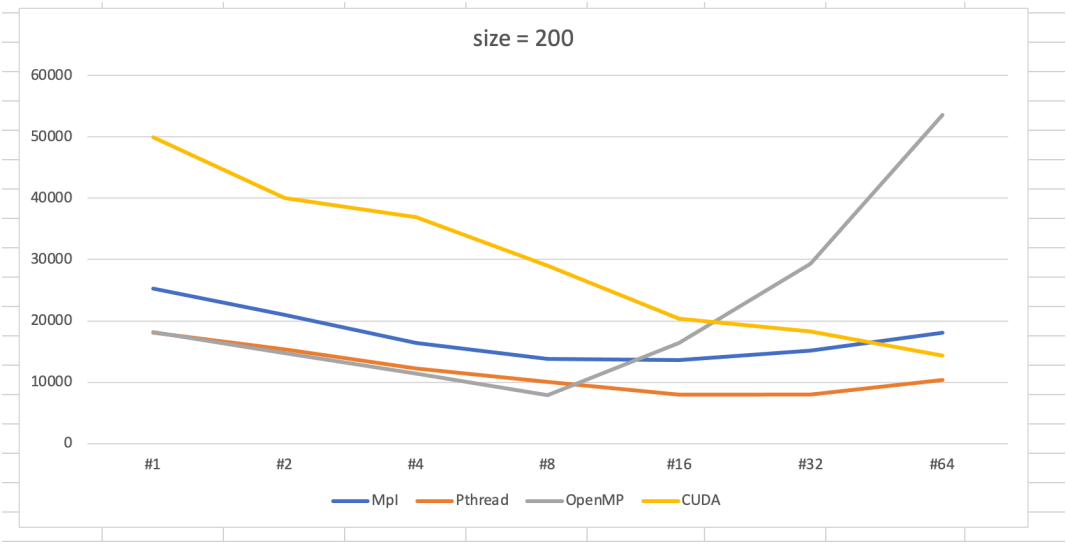
I compare all these four version, and put them into a whole table. The result is showing below:

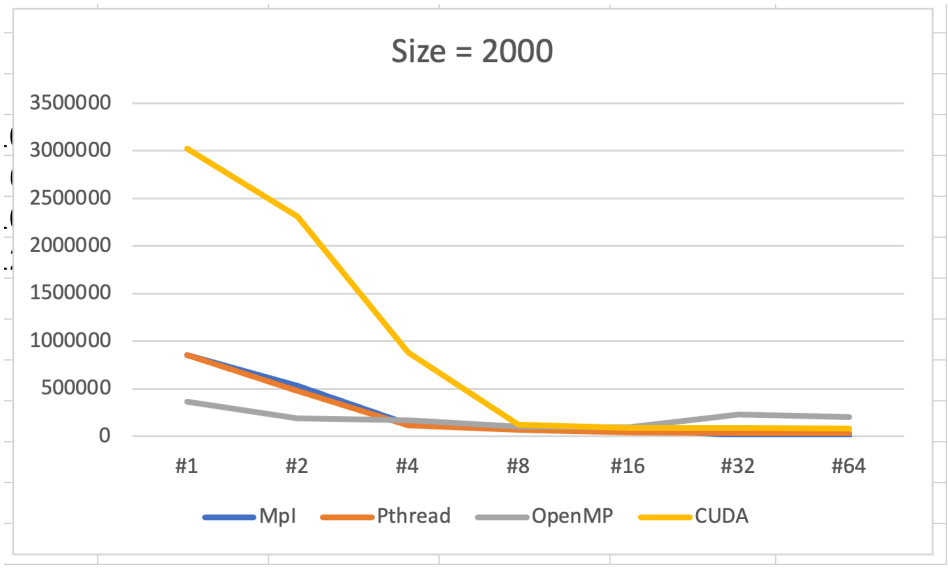
	cores/threads							
	1	2	4	8	16	32	64	128
bodies=20	2655	3150	2580	14054	25424	46935	111447	159141
bodies=100	10012	12143	12135	21341	23196	41404	23087	35683
bodies=200	25312	30524	20922	13775	13623	15127	18062	23915
bodies=500	65649	70307	44615	21642	17629	16668	18210	23425
bodies=100	125444	139044	86442	37630	25150	28089	24539	26646
bodies=200	854997	475280	113462	67592	40215	33943	34888	35073
	Pthread							
	MPI							
	OpenMP							
	CUDA							

We can aslo draw the figure:

The y-ordinate represent the time that average time to draw each body.







From the previous figure, we can also conclude that: The Ptrhead performs most better than others, especially when the problem size is very large. And CUDA performs least in this proeject, MPI could perform very well when the number of problem size is very small, howeve, the size increase, MPI performs won't be so good. OpenMP performs as almost the same as OpenMP.

-- Discussion:

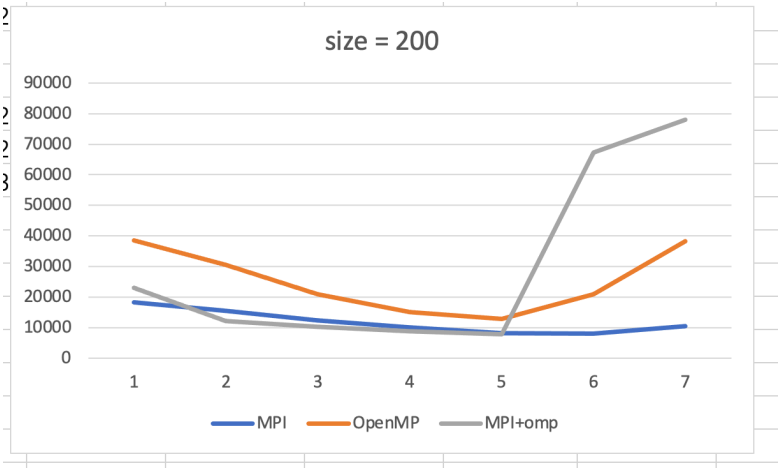
- For MPI, When the problem size is small and number of cores/thread are very few at the same time, MPI can perform the better job than others . that is mainly becasue MPI can have muti core to process and compute the pixels while pthread only have one core to do the computation, which will cost much more time than MPI. At the same time, due to few elements input, MPI will also have less communication between each core. Thus, MPI could perform better than Pthread when the input size is not very large.
- For CUDA, since CUDA use GPU not CPU, then when we use CUDA, we may need time to launch the kernel function and also need time to pass all the variable to the GPU, and after the calulation CUDA need to send the value back to the CPU, which will be time consuming. since GPU has many cores to do the calculation, so when the number of problem size increase, CUDA may perform better than other. But for more complicated computation, CUDA may perform worse than others since CUDA are unable not do the high-level calculation.
- For OpenMP, OpenMP performs as close as the Pthread, But OpenMP most of the time perform little worse than pthread in this project. The difference is that: Pthreads is a very low-level API for working with threads. On the other hand, OpenMP is much higher level, is more portable and doesn't limit you to using C. But lauchg more high-level API may have more chance to use more hidden function that we dont even know it laugh. So the low-level creation thread will not be explicit to us. It may consume more time to lauch openMP thread, so it may be little worse than Pthread.
- For Pthread, When the problem size and number of core/thread is large, Pthread will performs better than MPI. Especially, When the size is very large(2000), MPI performs very poor compared with pthread. that is mainly becasue when the problem size increase, the communication beteween each core will aslo increase as well. So MPI will cost much time on communication, while pthread have the advantage of sharing memory. Pthread does not need to communicate between each thread since they can modify their own canvas in their scope. Moreover, MPI need much time on synchronisation since it need to synchronize the root rank and other rank. Ptrhead only need the join function to join every thread together, which cost less time in communication.

MPI vs OpenMP vs MPI+OpenMP

For The bonus part, I pick two size to see the comparison between these performance.

-- Result:

The y-ordinate represent the time that average time to draw each body.



We can observe that when we add more threads in the MPI version, It will not always increase the performance. instead, When the size is small, omp+mpi version performs not good as MPI. But When the size increase, the performance of MPI+OpenMP will increase and perform better than MPI and OpenMP.

-- Discussion:

- since a node has only 32 core. It can be understood that a node can open countless threads, but only 32 threads run at the same time, which is scheduled by the OS (in simple terms, imagine round robin, and each thread runs in time slice). The 32+ core can only be used in combination with mpI, which synchronizes data to process on multiple nodes, each of which is responsible for pulling multiple threads. Hence when the size is large, The OpenMP + MPI version could performs better than these two.
- However, when the size is too small, too many thread for one node will decrease the speed instead since it need time to matin the Synchronous, and also need time to create thread an mutex, which is unnecessary here. so the speed will decrease instead.

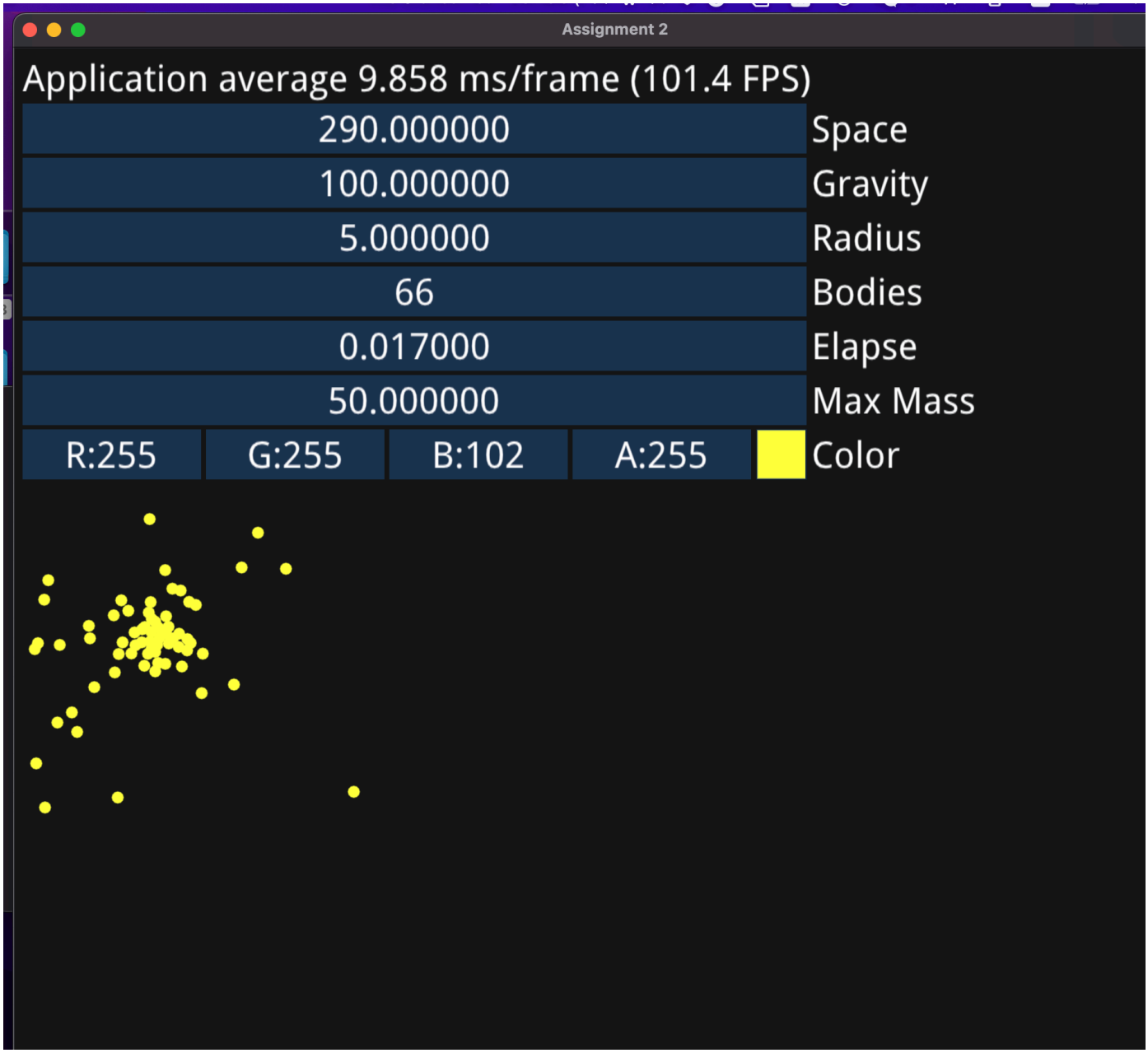
Conclusion

As the conclusion:

- Parralle computation will performs well nearly in every case than sequential one

- in general, applying more threads or cores for the Mandelbrot set computation would increase the parallel computing performance and the increaing tendency is stable.
- For small size and number of cores/threads, MPI will performs better than CUDA, Pthread and OpenMP computation. For large size and more cores/threads, Pthread has the better performace in computation than others. OpenMP has the cloest value to Pthread. CUDA performs not good and kind of time consuming when size is not large.
- To use the thread and core fully, we could add. more threads in one core when we use MPI, as the size increase, It will help to increase the parallel performance speed.

Demo Output



```
The speed is (time per body): 15472 ms
The speed is (time per body): 13153 ms
The speed is (time per body): 14921 ms
The speed is (time per body): 14306 ms
The speed is (time per body): 13523 ms
The speed is (time per body): 13506 ms
The speed is (time per body): 15514 ms
The speed is (time per body): 15406 ms
The speed is (time per body): 15980 ms
The speed is (time per body): 15919 ms
The speed is (time per body): 16121 ms
The speed is (time per body): 16313 ms
The speed is (time per body): 14624 ms
The speed is (time per body): 13380 ms
The speed is (time per body): 15402 ms
The speed is (time per body): 15192 ms
The speed is (time per body): 16765 ms
The speed is (time per body): 14135 ms
```