CSC4005-Assignment-2 Report

Name: HuangPengxiang

Student ID: 119010108

Objective: This Assignment requires to two parralle version of Mandelbrot sort computation program using MPI and Pthread. 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 also necessary to be included in this report.

CSC4005-Assignment-2 Report
- Introduction
- Running Guidance
  - *For MPI Version*:
  - *For Pthread Version && Dynamic Pthread Version:*
- Design Approach
  - Parallel Computing Design
  - Code Design
    - MPI Design
    - Pthread Design
    - Dynamic Pthread Design
  - Experiment Design
- Performance Analysis
  - Sequential vs parallel
  - Number of Cores/Threads
  - Problem Size
  - MPI vs Ptrhead
  - Static vs Dynamic
- Conclusion
  - Demo Result

Introduction

- The Mandelbrot set is used to refer both to a general class of fractal sets and to a particular instance of such a set. It is a fractal that when plotted on a computer screen roughly resembles a series of heart-shaped picture. The computaiton of Madnelbrot set is achieved by recursive iterations. Settle a initial value $Z_0 = 0$ and compute each Z_k by using the recursive equation $Z_{k+1} = Z_{k^2} + c$ for a fixed complex value c.
- However, if we apply the sequential method to calculate it, The compuation time will be quite large with the image size increase. Theoretically, Sequential computation will calculate $size^2$ Pixels one by one if we pick a fixed side length for a picture. Therefore, This program aims to investiaget different version of Parrallel computing to optimize the compuatation speed of Mandelbrot Set. In this experiment, I designed **Three parrallel computing version including `MPI`, `static Pthread` and `dynamic Pthread`** and compare those version with Squential one to find out the best one in computing performance for different scenario.
- The basic problems and tasks are: given a number set, and the program should perform the parrallel computing strictly following the rule of Mandbrot set calculation for whatever the size of picture and number of cores/threads. 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 three 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**, and **Dynamic_Pthread_Version**, and **Appendix** which contains the experimental data and some analysis pictures.

To build three versions of parrallel computation, you have to follow the instruction below:

```
$ cd /* The location of one of these version */
$ mkdir build && cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Debug // you may need to set debug here since Release may
automatically optimize my code
$ cmake --build . -j4
```

To test my program and see the graphic output, you need to follow the instruction 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 `/Project2/Appendix/`, you can found a script named `mph_2.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_2 located */
$ sbatch mpi_2.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 && Dynamic Pthread Version:

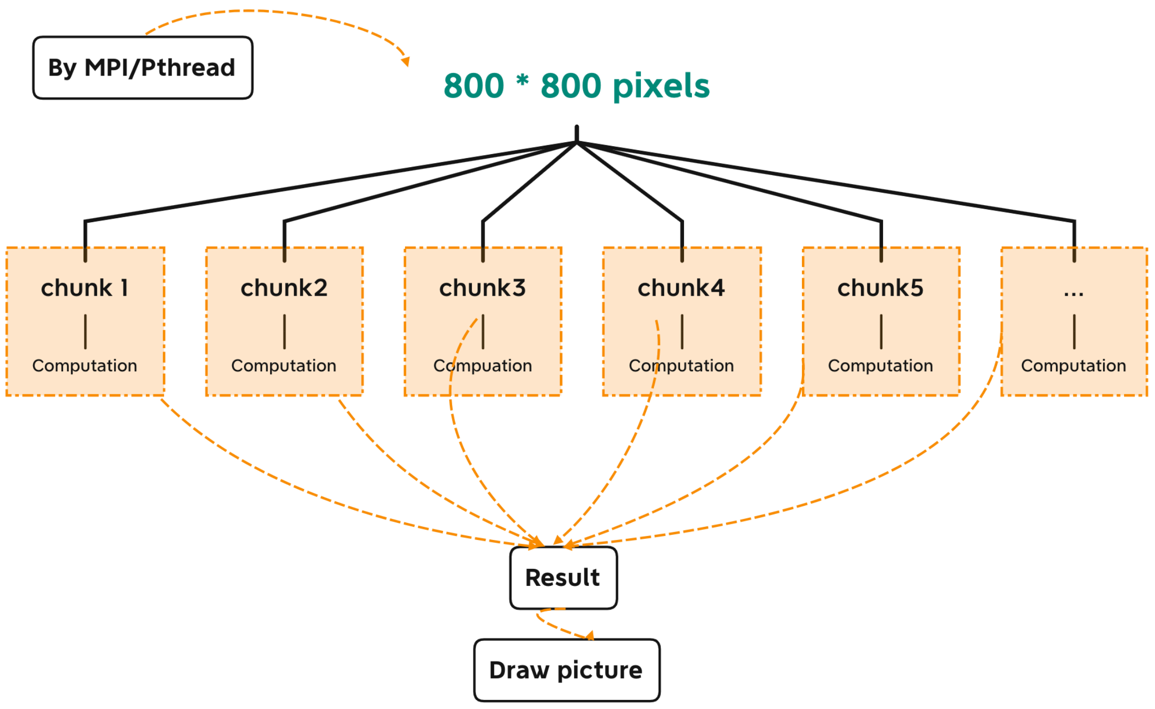
You may need to add one more argument to test my program, I ask user to input the number of thread before running it.

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

Design Approach

Parallel Computing Design

The main computation task is calculate every pixel's value in the figure. The basic idea is divide the whole picture into many chunk and use parrallel computing method to calculate each chunk, then gather all of chunk result into one to draw the Mandelbrot set figure.



Code Design

From the previous fogure, I found that if we evenly distribute the task to each chunk, the number fo calcualtion for each chunk is not the same, since for a Mandelbrot Set figure, the points needed to be calculate in center is much more than those in the margin, which means there is **bottleneck** in Mandelbrot Set. So for MPI and Ptrhead version, I evenly distribute the task, while for dynamic pthread version, I also design a way to dynamicly calculate the pixels which avoid the bottleneck to some extent.

MPI Design

To design parallel computation via MPI, it mainly compose of 5 parts:

- **Bcast**: BoardCast the information to each rank
- **Division**: Root send the initial array to other rank to calculate
- **Calculation**: Non-Root rank receive the array and calculate, Root calculate its own array at the same time.
- **Gather**: Non-Root rank send the result to the Root, Root gather all the information
- Output

-- **BoardCast**:

I used 3 times of `MPI_Bcast` function in my program, each time I boardcast the initial information such as `center_x`, `center_y`, and so on. The reason I use boadcast are that it will spread those information I used in every rank and also **BoarCast will synchronize all the rank**, since it will wait all rank receive the information.

The information I have boardcasted:

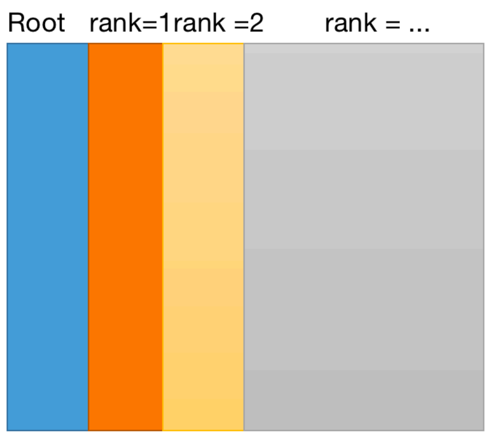
```
MPI_Bcast(&center_x, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&center_y, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&k_value, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&scale, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

-- **Division**:

- First I `malloc` a global array in order to store all the results from other rank and root.
- Then I malloc the array for each rank, the **tasks will be evenly distributed** but may have different lenghth depend on the remaining.
- Root send the array to other rank, and another wait for the send and recieve its tasks

How I divide the tasks?

Divide The Column



How I implemet synchronisation?

Basically, I use `ISend` , `Irecieve` and `wait` three function to implement the synchronisation. The Root will send the information to other rank, and wait for other rank to recieve. at the same time, other rank will wait for the information, once received, will send the signal to Root. After waiting other recieve the information, all the rank will execute their next step.

```
// Send
MPI_Request request;
MPI_Isend(localarr, local_len * size, MPI_INT, i, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUSES_IGNORE);

// Recieve
MPI_Irecv(localarr, size*local_len ,MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
```

-- **Caculation**:

This part basically follow the rule of Mandelbrot set calculation. each rank calculate the theri pixels and store the number k into the local array.

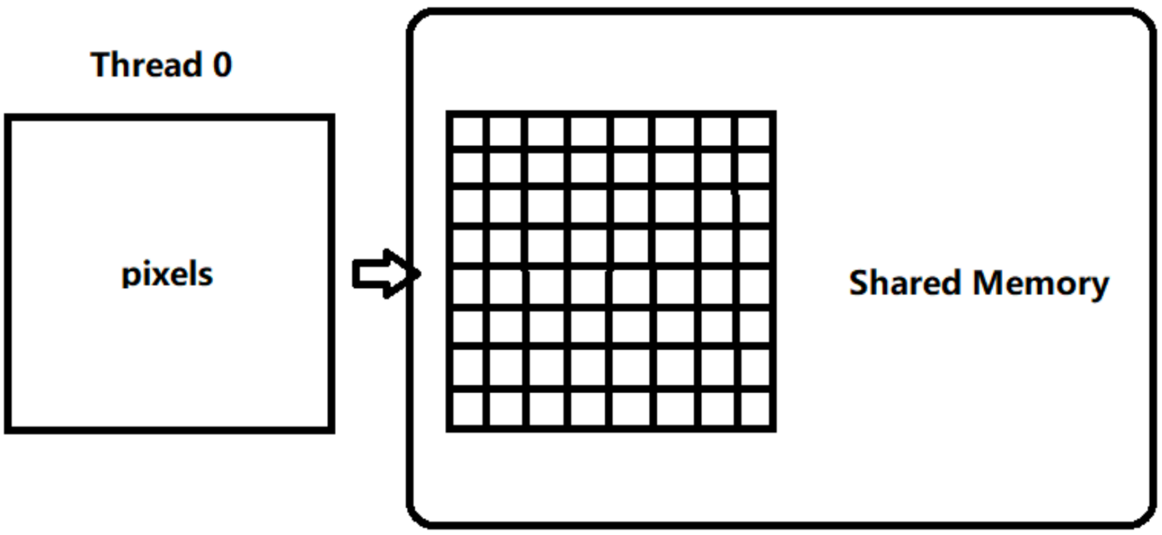
-- **Gather**:

The gather part is very simlar to the division part. The local array calulate the result and send the array to Root. I also used `Isend` `Irecieve`, and `wait` function to implement synchronisation.

```
//send
MPI_Request sendreq;
MPI_Isend(localarr, local_len * size, MPI_INT, 0, 0, MPI_COMM_WORLD, &sendreq);
MPI_Wait(&sendreq, MPI_STATUSES_IGNORE);

// recieve
int * localarr = (int*)std::malloc(sizeof(int) * size * local_len);
MPI_Irecv(localarr, size*local_len ,MPI_INT, i, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
```

Pthread Design

For implementation via Pthread, I aslo evenly divide the pixels 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.

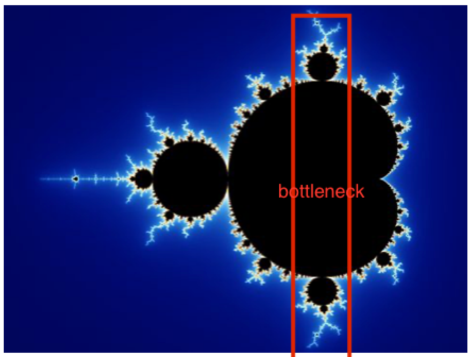


Hence, each thread have access to calculate and write the value into `canvas`. The implementation is much more easier than MPI Version, Here is the sample method I have used:

```
// Pthread Initialization
pthread_t *threads=(pthread_t*)malloc(sizeof(pthread_t)*threadnum);

// distribute the task to each thread
for (int j = 0; j < threadnum; j++) {
    pthread_create(&threads[j],&attr,Pthreadcal,(void*)&thread_paras[j]);
}

Thread calculation function  Here...
Write the value into canvas

// join every thread for synchronisation
for (int i = 0; i < threadnum; i++){
    pthread_join(threads[i],NULL);
}

I/O Here...
```

Dynamic Pthread Design

A brief introduction of bottleneck: the chunk which has to do the most calcualtion than others and will cost a lot of time since other rank may wait until it finished.



To reduce the bottleneck, The most important difference between Pthread and dynamic Ptrhead is **Task Distribution**, in this version, The task is not evenly distributed. I divide task into *size* chunk, which means that if the picture is 800 $\times$ 800 pixels, then it will be divided into 800 chunks. And each thread will calculate one chunk at a time. After they finish one chunk, it will find another chunk to calculate. Hence I set a variable named `count` which will monitor the chunk have been calculated. Also I set a `mutex` to avoid deadlock since every thread will modify the `count`. Each thread wiil have the access to modify the `count` after they finish the calculate, and it will find a new task to calculate until meet the count max.

```
count = threadnum - 1;
int current  = myid;
while (count < size && current < size){
      for (int j = 0; j < size; ++j){
          double x = (static_cast<double>(j) - cx) / zoom_factor;
          double y = (static_cast<double>(current) - cy) / zoom_factor;
          std::complex<double> z{0, 0};
          std::complex<double> c{x, y};
          int k = 0;
          do {
              z = z * z + c;
              k++;
          } while (norm(z) < 2.0 && k < k_value);
          canvas[{current,j }] = k;
      }

      /* lock the count and update the task */
      pthread_mutex_lock(&cal_mutex);
      current = count + 1;
      count++;
      pthread_mutex_unlock(&cal_mutex);
      if (count >= size) break;
}
```

Experiment Design

Totally 5 topic I investigated in this project:

- **Sequential** vs **Parrallel**
- **Number of cores/thread**
- **Problem size**
- **MPI vs Ptrhead**
- **Static scheduling** vs **Dynamic scheduling**

-- **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 size is (100, 400, 800, 2000, 5000, 10000, 20000). 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: 1. Whether parallel version has a better performance than sequential one under different situations.
   2. 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 800, 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
```

In Pthread version, you need input the thread number by yourself in the terminal after the command `./csc4005_imgui` .

-- **Design Problem size:**

In this design of experiment, I divide The problem into three modes: Small (100, 400, 800), medium(2000,5000), and large (10000, 20000). In addition, to make it more general I test different problem sizes on 1, 2, 4, 8, 16, 32, 64 threads/cores. Therefor, the tendency of performance change as the problem size increase will be observed. The goal is to find out that:

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

-- **Design MPI vs Pthread**

in this design, I mainly focus on the comparation between MPI and Pthread on the parallel Mandelbrot set computation. To make the comparation fair, the scheduling strategy of MPI and Pthread are both static scheduling. Then I fixed the same core/thread and to obeserve the performance difference under each situation. The goal is to find out that:

- Under what situation MPI performs better
- The major reason why this model performs worse in som situation

-- **Design staic vs Dynamic**

In this design, I compare the performace between static scheduling and dynamic scheduling. The problem with size equal to 800 is tested on 8, 16, 32 threads via dynamic scheduling and static scheduling. The goal is find out that

- The overall performance on them
- The reason of why dynamic faster under some situation.

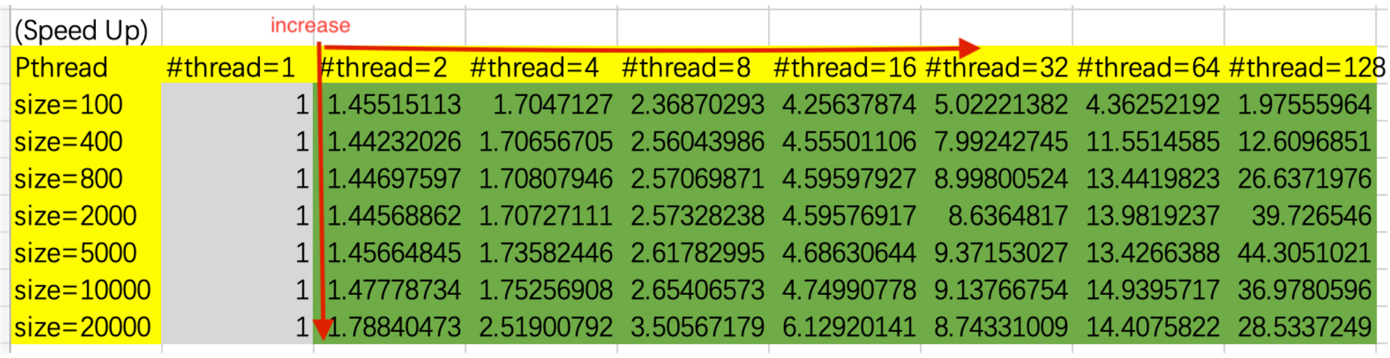
Performance Analysis

Sequential vs parallel

-- **Result**

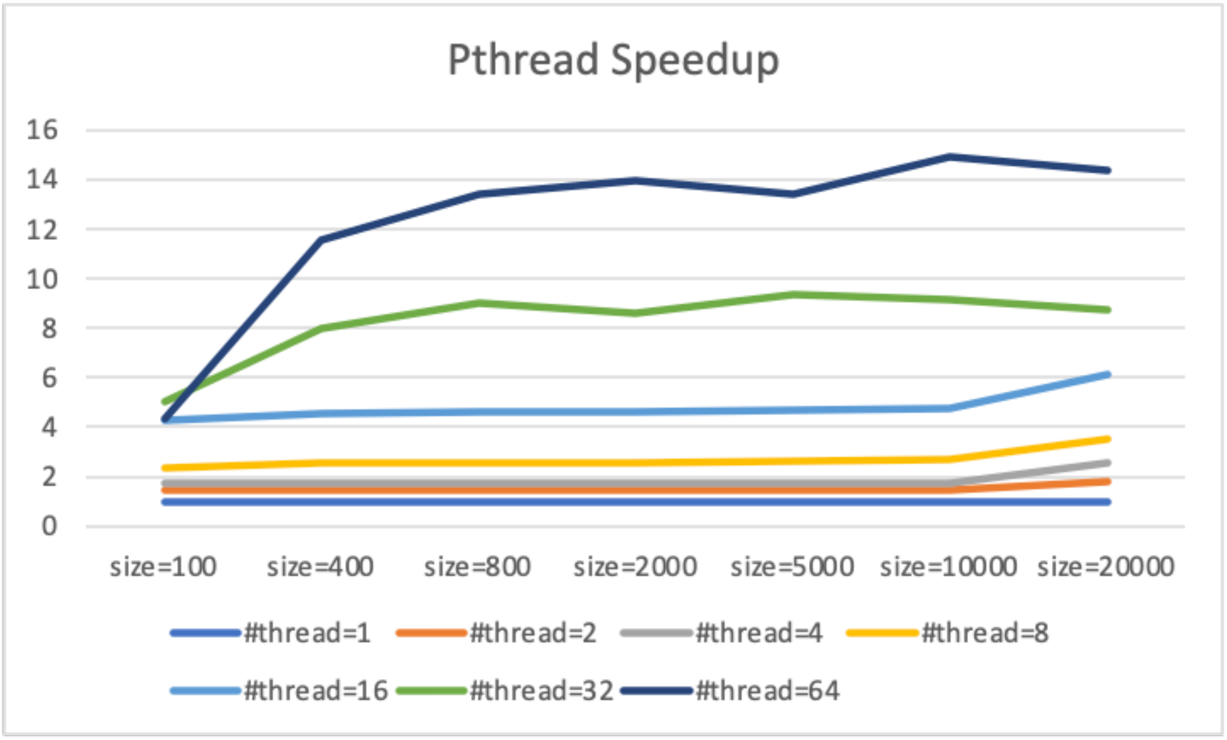
For Pthread: (I just choose pthread to compare the sequential and paralle since the comparison in MPI already be done in Project 1)

Green represent the the speed up is greater than 1, which means parralle performs better than sequential one.



(Speed Up)								
Pthread	#thread=1	#thread=2	#thread=4	#thread=8	#thread=16	#thread=32	#thread=64	#thread=128
size=100	1	1.45515113	1.7047127	2.36870293	4.25637874	5.02221382	4.36252192	1.97555964
size=400	1	1.44232026	1.70656705	2.56043986	4.55501106	7.99242745	11.5514585	12.6096851
size=800	1	1.44697597	1.70807946	2.57069871	4.59597927	8.99800524	13.4419823	26.6371976
size=2000	1	1.44568862	1.70727111	2.57328238	4.59576917	8.6364817	13.9819237	39.726546
size=5000	1	1.45664845	1.73582446	2.61782995	4.68630644	9.37153027	13.4266388	44.3051021
size=10000	1	1.47778734	1.75256908	2.65406573	4.74990778	9.13766754	14.9395717	36.9780596
size=20000	1	1.78840473	2.51900792	3.50567179	6.12920141	8.74331009	14.4075822	28.5337249

Then, I draw the figure to see the result more clearly.



-- **Discussion**:

- In my program, I found that the parralle version is better than sequential one for every case I had choose. 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
- 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.
- However, when problem size is small, the sequential section and the time for each threads to access to the

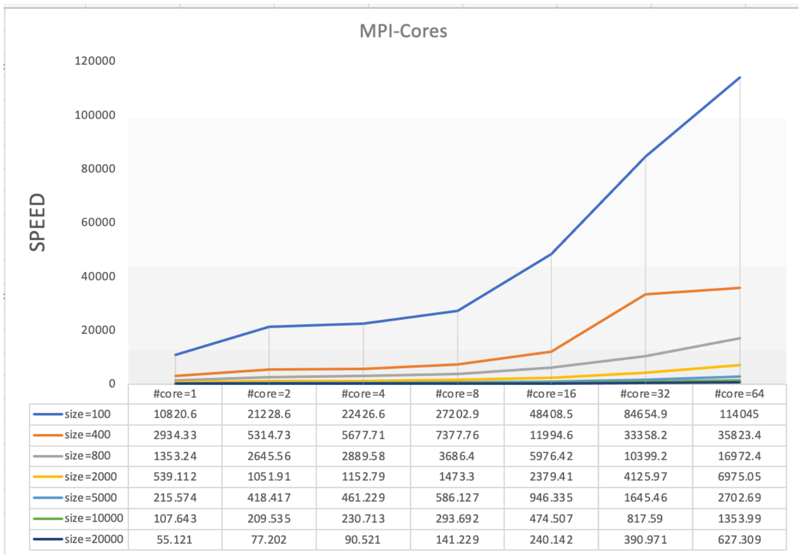
shared memory play a more significant role. The improvement that parallel computing brings is less than the delay cause by the sequential section and shared memory access.
- The multi-thread computing is better than sequential computing especilally when problem size is larg. When the numberof threads is larger and problem size is greater, the advantage of multi-thread computing is more obvious.

Number of Cores/Threads

-- **Result**

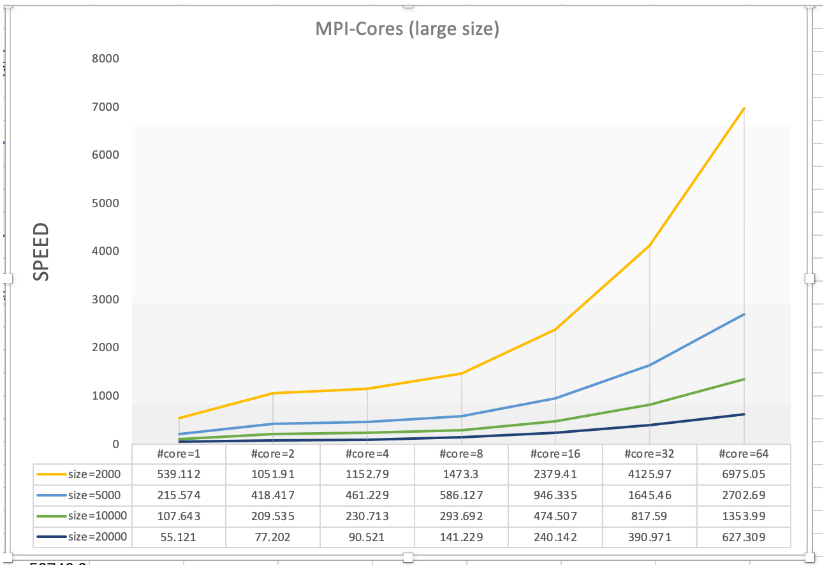
For MPI Version:

The figure shows that as the number of threads increases (from 1 to 64) overall, the performance would increase dramatically from **number of core 8**, 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.



	#core=1	#core=2	#core=4	#core=8	#core=16	#core=32	#core=64
size=100	10820.6	21228.6	22426.6	27202.9	48408.5	84654.9	114045
size=400	2934.33	5314.73	5677.71	7377.76	11994.6	33358.2	35823.4
size=800	1353.24	2645.56	2889.58	3686.4	5976.42	10399.2	16972.4
size=2000	539.112	1051.91	1152.79	1473.3	2379.41	4125.97	6975.05
size=5000	215.574	418.417	461.229	586.127	946.335	1645.46	2702.69
size=10000	107.643	209.535	230.713	293.692	474.507	817.59	1353.99
size=20000	55.121	77.202	90.521	141.229	240.142	390.971	627.309

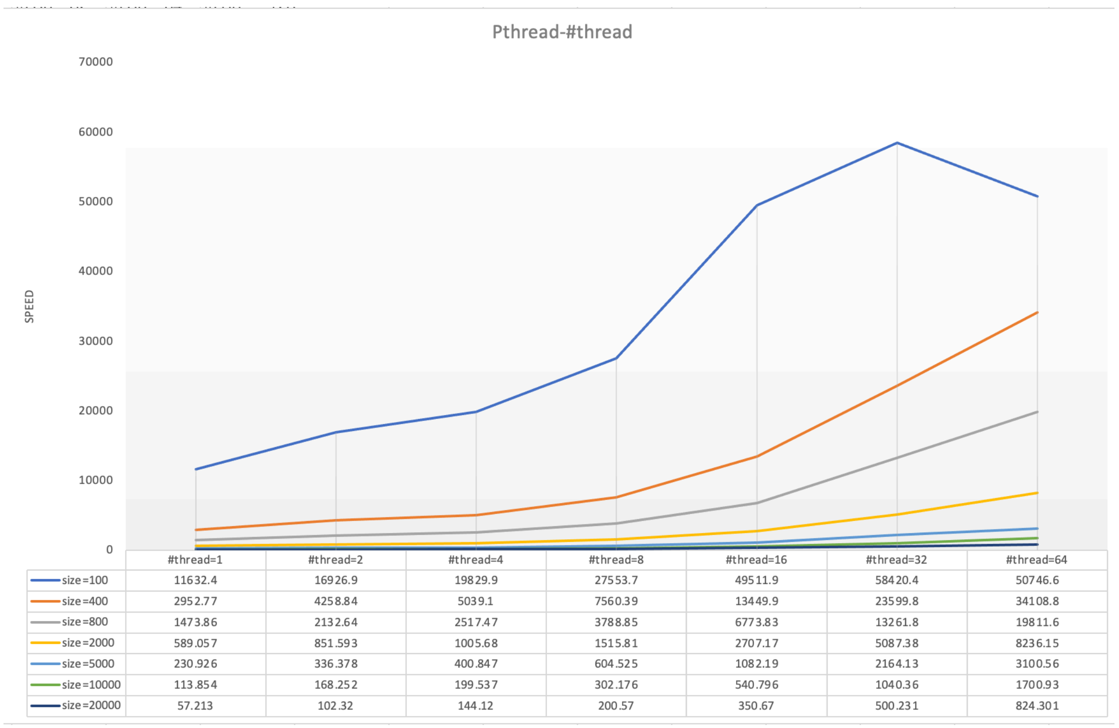
See the core influence in large size more clearly, The figure bleow shows the tendancy of MPI-Cores in large problem size. We can find that if the size is larger, the influence of number core will reduce. Even when the size is extremely large (such as 20000), The speed will have only little improvement in speed.



	#core=1	#core=2	#core=4	#core=8	#core=16	#core=32	#core=64
size=2000	539.112	1051.91	1152.79	1473.3	2379.41	4125.97	6975.05
size=5000	215.574	418.417	461.229	586.127	946.335	1645.46	2702.69
size=10000	107.643	209.535	230.713	293.692	474.507	817.59	1353.99
size=20000	55.121	77.202	90.521	141.229	240.142	390.971	627.309

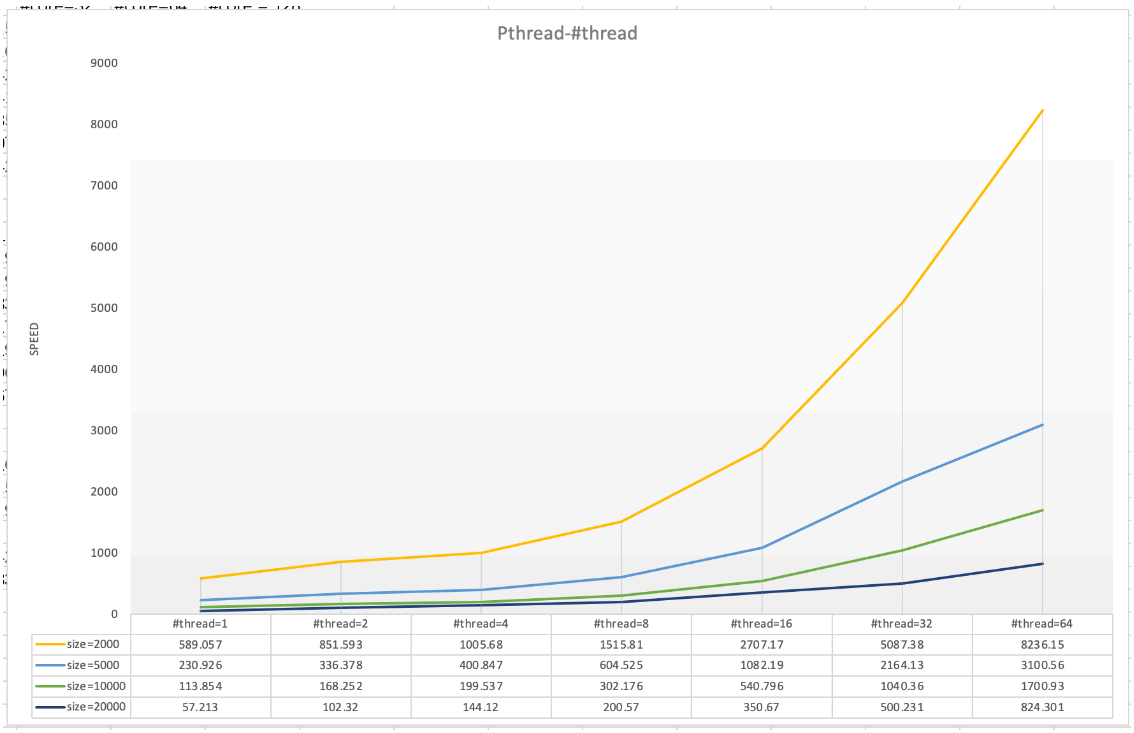
For Pthread:

The figure showing below is the Pthread speed tendancy during different thread number. the performance would increase dramatically from **number of thread 8**, But in this figure, it is not clear for the largsize speed during core number change.



	#thread=1	#thread=2	#thread=4	#thread=8	#thread=16	#thread=32	#thread=64
size=100	11632.4	16926.9	19829.9	27553.7	49511.9	58420.4	50746.6
size=400	2952.77	4258.84	5039.1	7560.39	13449.9	23599.8	34108.8
size=800	1473.86	2132.64	2517.47	3788.85	6773.83	13261.8	19811.6
size=2000	589.057	851.593	1005.68	1515.81	2707.17	5087.38	8236.15
size=5000	230.926	336.378	400.847	604.525	1082.19	2164.13	3100.56
size=10000	113.854	168.252	199.537	302.176	540.796	1040.36	1700.93
size=20000	57.213	102.32	144.12	200.57	350.67	500.231	824.301

See the Performance tendancy in large size, The figure bleow shows the tendancy of Ptrhead-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 20000), The speed will have only little improvement in speed.



	#thread=1	#thread=2	#thread=4	#thread=8	#thread=16	#thread=32	#thread=64
size=2000	589.057	851.593	1005.68	1515.81	2707.17	5087.38	8236.15
size=5000	230.926	336.378	400.847	604.525	1082.19	2164.13	3100.56
size=10000	113.854	168.252	199.537	302.176	540.796	1040.36	1700.93
size=20000	57.213	102.32	144.12	200.57	350.67	500.231	824.301

-- **Discussion**

Based on the previous figure, We can easily obeserved that:

- For both model, First is that the performance(using speed(pixels/ns)) of parallel computing is generally increasing in nearly quadratical way(the increasing rate is around 0.418) as the number of threads increases.

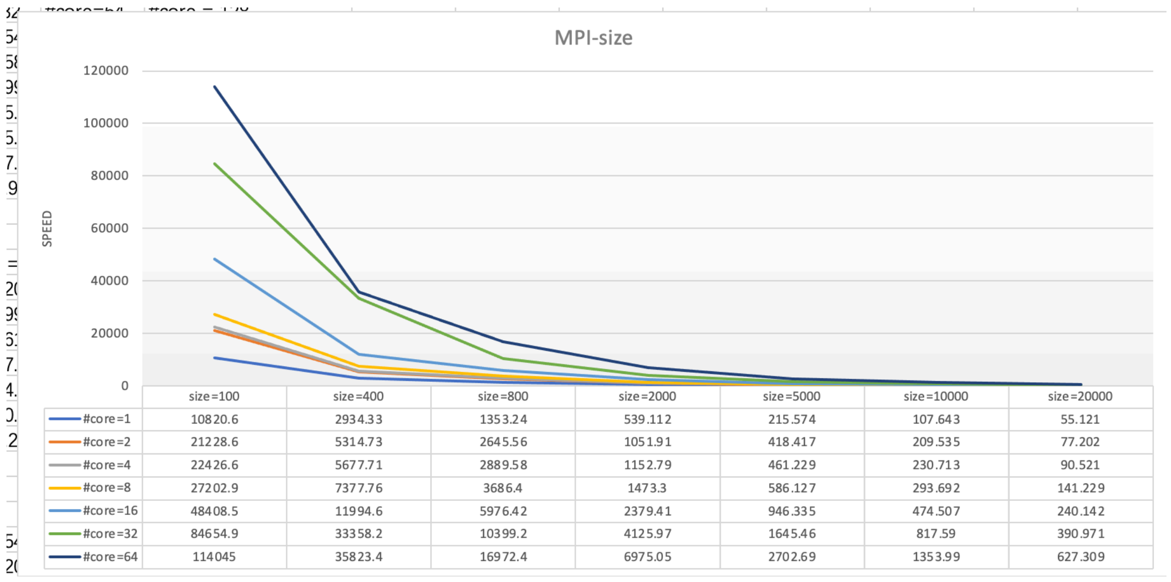
- For Ptrhead, there is an decrease when size is too small(such like 100), the speed will decrease instead when thread number increase from 32 to 64. This happened simply becasue the bottle neck. may some of threads in 64 threads meet the bottleneck and need more time to calculate it, however, other thread already finish their calculation so they have to wait the some bottleneck thread finish their computation. In the rest disscussion, I will proof it in detail.
- For small and medium size (100 - 2000), the speed will dramatically increase as the number of cores/threads increase, since more cores/thread represent more calcualte unit for computation job. And aslo size is not too large, the speed will increase dramatically with cores/thread increase.
- For large size(5000,10000,20000), The speed will increase very slow. Even for the very large size(2000) the speed gradually increase when thread is approch to 16. it means that when the size is large, the size may become the major reason that mainly affect the speed of calcualtion.
- 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.

Problem Size

-- Result

For MPI:

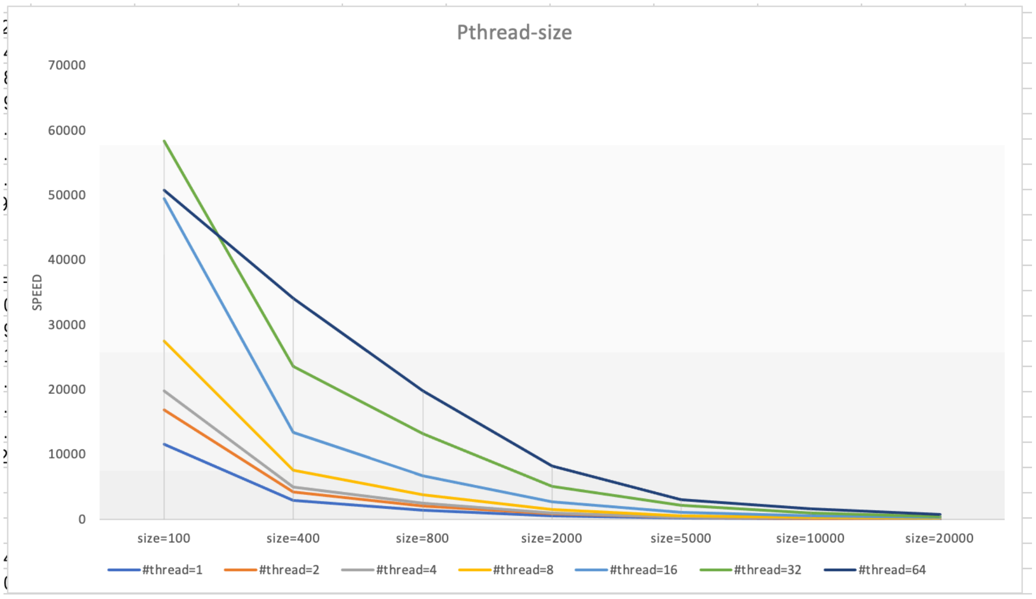
The figure shows that the tendancy of speed as the size increases (from 100 to 20000).



	size=100	size=400	size=800	size=2000	size=5000	size=10000	size=20000
#core=1	10820.6	2934.33	1353.24	539.112	215.574	107.643	55.121
#core=2	21228.6	5314.73	2645.56	1051.91	418.417	209.535	77.202
#core=4	22426.6	5677.71	2889.58	1152.79	461.229	230.713	90.521
#core=8	27202.9	7377.76	3686.4	1473.3	586.127	293.692	141.229
#core=16	48408.5	11994.6	5976.42	2379.41	946.335	474.507	240.142
#core=32	84654.9	33358.2	10399.2	4125.97	1645.46	817.59	390.971
#core=64	114045	35823.4	16972.4	6975.05	2702.69	1353.99	627.309

For Ptrhead:

The figure shows that the tendancy of speed as the size increases (from 100 to 20000).



-- **Discussion**

- For both model, the speed of calculation will decease dramatically first and gradually tend to flat. When the size is extremely large (like 20000), thread number perform s little contibution on the speed.
- For small size, The speed differs a lot. And during my experiment, The **Variance** are very high. The value showing in the chart is the average value after calculation. That is simly becasue the size is too small and each thread have very few task to calculate, and they can finish their job in a very short time, which may cost the high variance.
- Also, The general increasing tendency can be explained by the Aldam's law, which is:

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

  Notice that Mandelbrot Set Computation is ideally an Embarrassingly parallel computation, which means **there is no data dependency in this problem(all the pixels would not effect each other).** Therefore, the serial section (f in aldam's law) is quite low(ideally 0) in the computation. This is the reason why the increasing tendency could be maintained as the number of threads increases and size decrease.

MPI vs Ptrhead

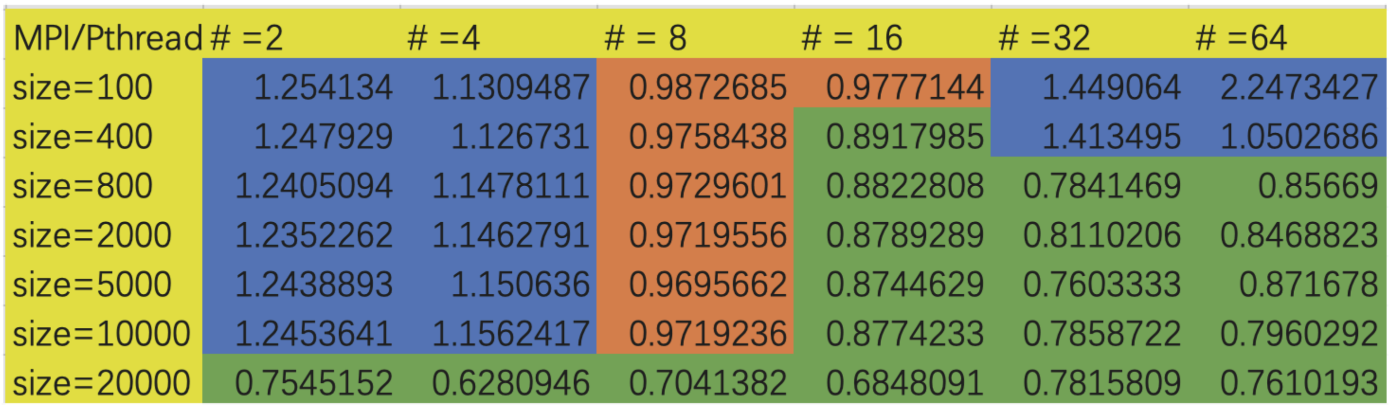
-- **Result**

The chart represent MPI/Pthread is showing below:

This is the chart which represent $Speed_{MPI}/Speed_{Pthread}$

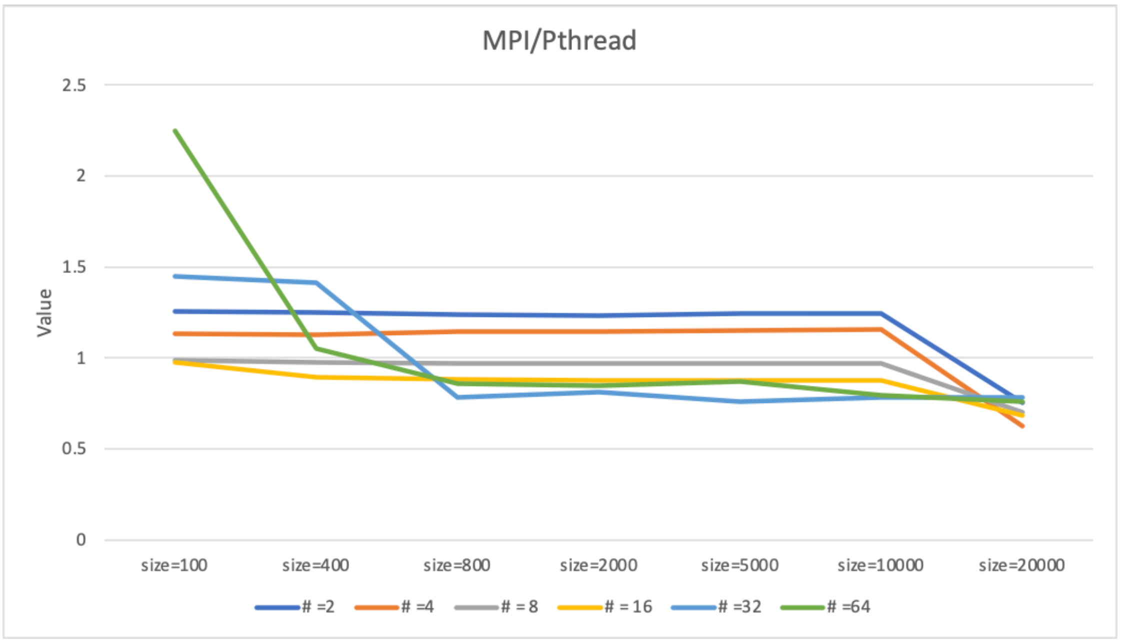
`#` represent the number of core/thread. **Green** color represent the Pthread is much better than MPI, **Bule** color represent MPI performs better than Pthread, and

Orange color represent they performs as equal.



MPI/Pthread	# =2	# =4	# = 8	# = 16	# =32	# =64
size=100	1.254134	1.1309487	0.9872685	0.9777144	1.449064	2.2473427
size=400	1.247929	1.126731	0.9758438	0.8917985	1.413495	1.0502686
size=800	1.2405094	1.1478111	0.9729601	0.8822808	0.7841469	0.85669
size=2000	1.2352262	1.1462791	0.9719556	0.8789289	0.8110206	0.8468823
size=5000	1.2438893	1.150636	0.9695662	0.8744629	0.7603333	0.871678
size=10000	1.2453641	1.1562417	0.9719236	0.8774233	0.7858722	0.7960292
size=20000	0.7545152	0.6280946	0.7041382	0.6848091	0.7815809	0.7610193

And I also draw the pciture for this chart:



--- **Discussion**

From the previous chart and figure, we can observe the tendancy of the rate MPI/Pthread.

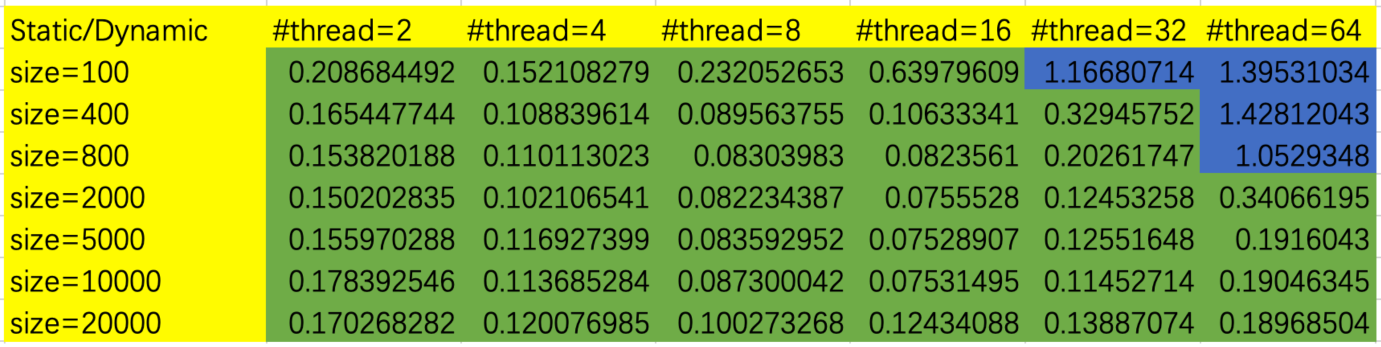
- When the problem size is small and number of cores/thread are very few at the same time, MPI can perform the better job than Pthread. that is mainly becasue MPI can have muti core to process and compute the pixels while pthread only have one core to do the computation, which will cost much more time 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.
- When the problem size and number of core/thread is large, Pthread will performs better than MPI. Especially, When the size is very large(20000), 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.
- As for the middle problem size and Middle number of core/thread, they performs as similar in computatiaon speed. there may be a tradeoff area for those two version. since the size is not large and small, and the number of thread is not too large and small as well. They performs almost equal when problem size range from 400 to 5000 at core/thread 8.
- There is an also interesting fact that when size is very small(100), and number of cores/thread is large, Pthread will perform very poor than MPI. That is becasue If there are many thread to calculate very small task, which

means one process have to divide many thread for calculation, and each thread only calculate very small number of jobs, which is also a waste, since there are many thread needed to be joined and one core is also burdened since there are many thread need to calculate.

Static vs Dynamic

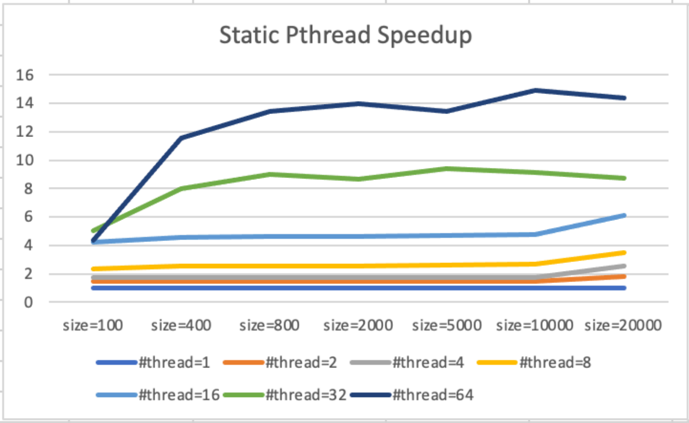
-- **Result**

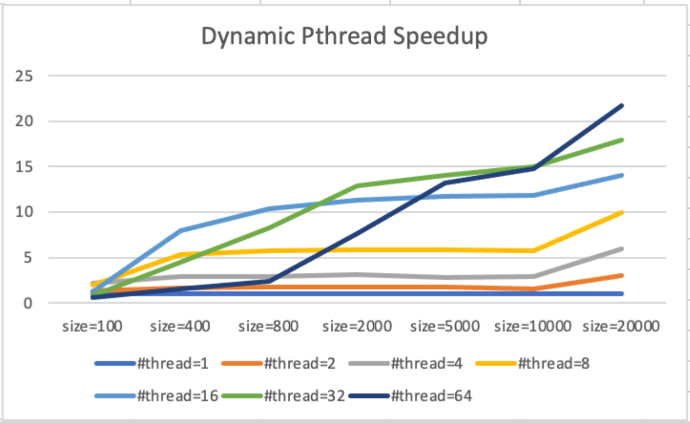
The chart showing below represent $Speed_{StaticPthread}/Speed_{DynamicPthread}$. **Green** represent dynamic performs better, **Blue** represent static performs better.



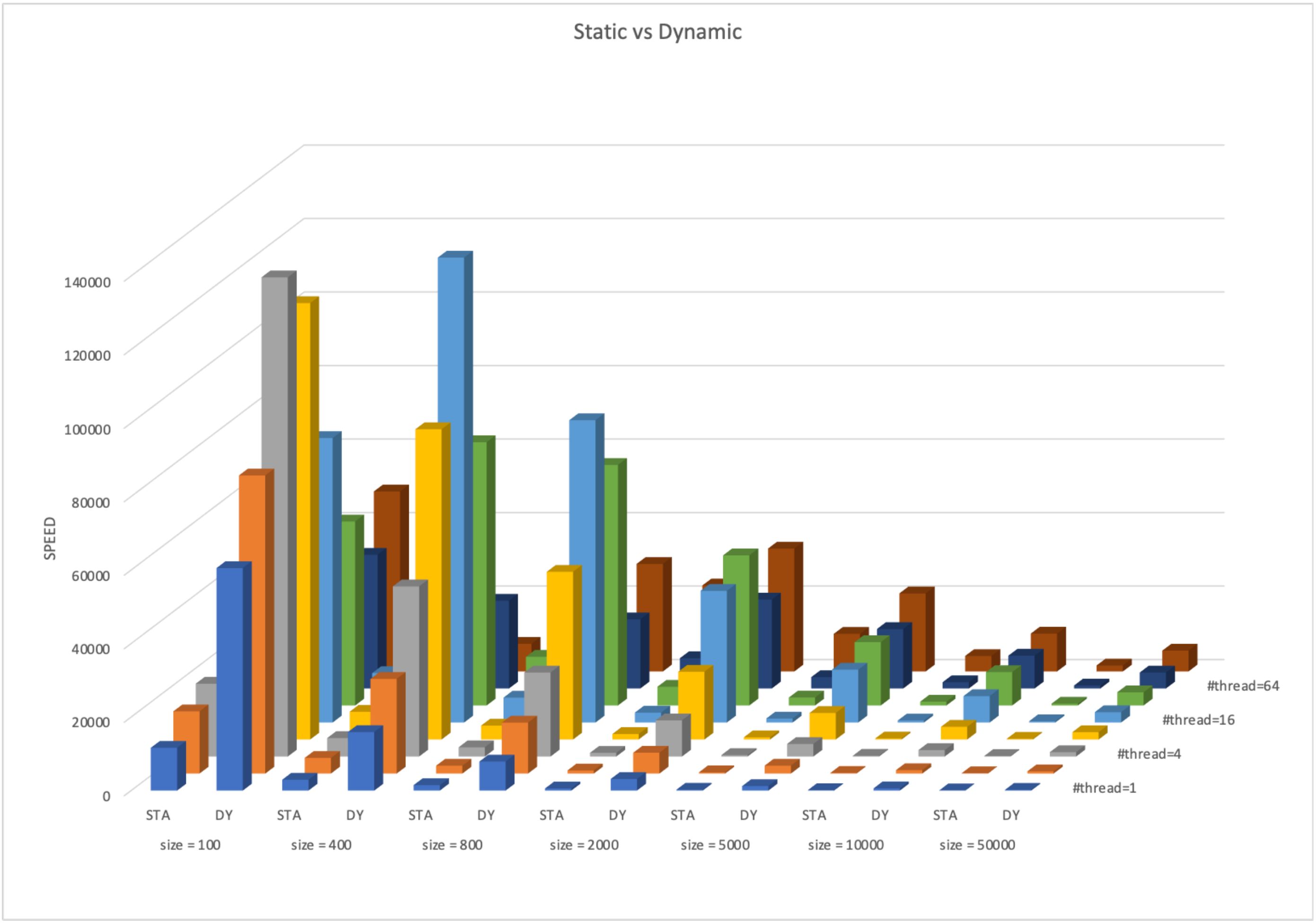
Static/Dynamic	#thread=2	#thread=4	#thread=8	#thread=16	#thread=32	#thread=64
size=100	0.208684492	0.152108279	0.232052653	0.63979609	1.16680714	1.39531034
size=400	0.165447744	0.108839614	0.089563755	0.10633341	0.32945752	1.42812043
size=800	0.153820188	0.110113023	0.08303983	0.0823561	0.20261747	1.0529348
size=2000	0.150202835	0.102106541	0.082234387	0.0755528	0.12453258	0.34066195
size=5000	0.155970288	0.116927399	0.083592952	0.07528907	0.12551648	0.1916043
size=10000	0.178392546	0.113685284	0.087300042	0.07531495	0.11452714	0.19046345
size=20000	0.170268282	0.120076985	0.100273268	0.12434088	0.13887074	0.18968504

The figure showing below is the comparsion of static and dynamci ptrhead performance during the same condition





To compare it in three dimension way, which will see it more clearly:



-- **Discussion**

From the previous figure, We can obeserve that:

- Dynamic scheduling really performs a better job than Static one, since Dynamic wiil effeciently avoid the bottleneck problem. However, when the size is too small and number of cores/thread is very large, satic will abnormally performs better than static, that is because dynamic pthread has a mutex in it to update the count, when the size is too smalll, it may cose much time.
- However, when using dynamic scheduling, the thread who complete its local work can continue to compute next block of pixels. The overall workload distribution is more close to evenly distributed. Therefore, the bottleneck thread has much less workload than the one in static scheduling. So the overall performance is better.
- Dynamic's Speed up grows more like a linear function. That is dynamic wiill reduce the bottleneck problem, which will increase in a more linear way, since the thread will also increase linearly. However, static will not grows so linear like that, that is becasue it have more bottleneck to deal with, which will cost much time in some thread.

Conclusion

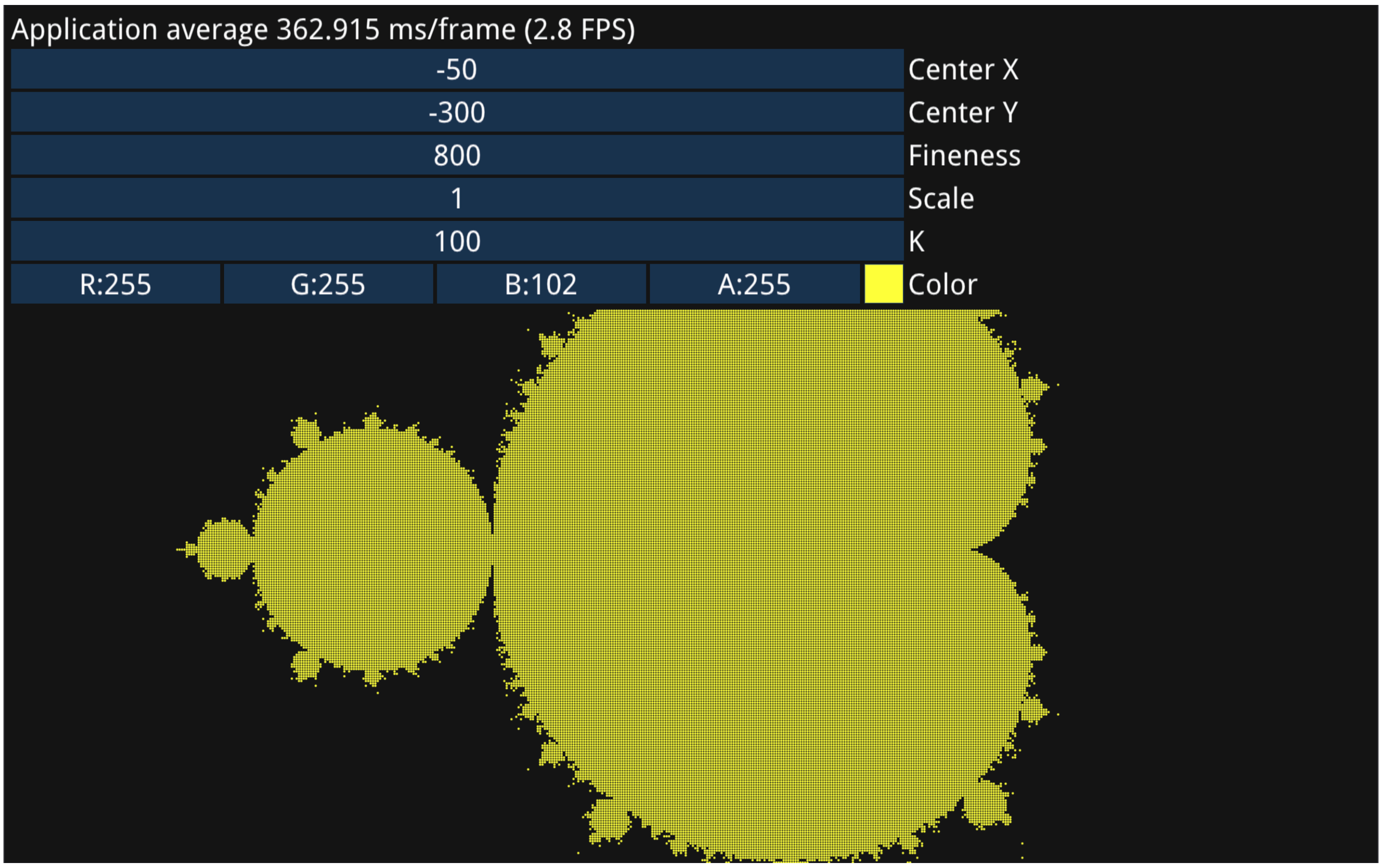
As the conclusion:

- 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 Pthread computation. For large size and more cores/threads, Pthread has the better performace in computation than MPI

- To avoid the bottleneck, The dynamic scheduling of tasks will efficiently reduce many calculation time, which is helpful in parallel computation.

Demo Result

The demo figure:



Application average 362.915 ms/frame (2.8 FPS)

-50 Center X

-300 Center Y

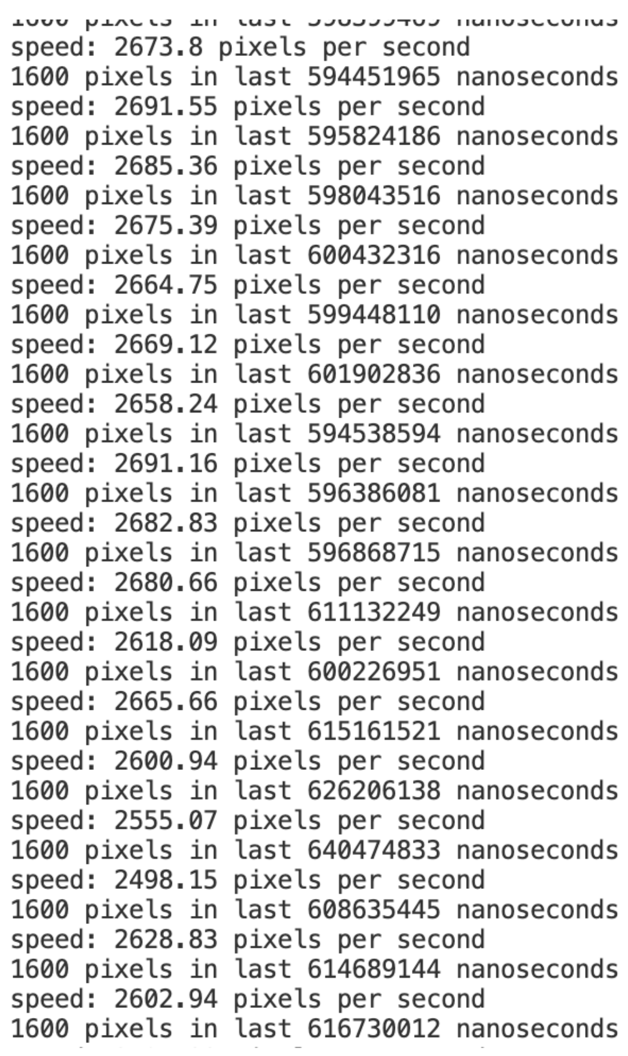
800 Fineness

1 Scale

100 K

R:255 G:255 B:102 A:255 🟨 Color

The demo terminal output:



```
1600 pixels in last 598399469 nanoseconds
speed: 2673.8 pixels per second
1600 pixels in last 594451965 nanoseconds
speed: 2691.55 pixels per second
1600 pixels in last 595824186 nanoseconds
speed: 2685.36 pixels per second
1600 pixels in last 598043516 nanoseconds
speed: 2675.39 pixels per second
1600 pixels in last 600432316 nanoseconds
speed: 2664.75 pixels per second
1600 pixels in last 599448110 nanoseconds
speed: 2669.12 pixels per second
1600 pixels in last 601902836 nanoseconds
speed: 2658.24 pixels per second
1600 pixels in last 594538594 nanoseconds
speed: 2691.16 pixels per second
1600 pixels in last 596386081 nanoseconds
speed: 2682.83 pixels per second
1600 pixels in last 596868715 nanoseconds
speed: 2680.66 pixels per second
1600 pixels in last 611132249 nanoseconds
speed: 2618.09 pixels per second
1600 pixels in last 600226951 nanoseconds
speed: 2665.66 pixels per second
1600 pixels in last 615161521 nanoseconds
speed: 2600.94 pixels per second
1600 pixels in last 626206138 nanoseconds
speed: 2555.07 pixels per second
1600 pixels in last 640474833 nanoseconds
speed: 2498.15 pixels per second
1600 pixels in last 608635445 nanoseconds
speed: 2628.83 pixels per second
1600 pixels in last 614689144 nanoseconds
speed: 2602.94 pixels per second
1600 pixels in last 616730012 nanoseconds
```