

Assignment 2: Mandelbrot Set Computation

Name: Qin Peiran Student ID: 119010249

Part 1: Introduction

Mandelbrot set is the set of points in a complex plane that are quasi-stable when computed by iterating the function

$$z_{k+1} = z_k^2 + c$$

This project is seeking a parallel computing way for the mandelbrot set computation. That means this report would implement a parallel computing way for the mandelbrot set computation by `MPI` and `Pthread`. Apart from the implementation, my project would also investigate the following specific questions:

1. Investigation of the relationship between **number of threads/cores** and the performance of parallel computing via MPI or Pthread.
2. Investigation of the relationship between **problem size** and the performance of parallel computing via MPI/Pthread.
3. **Comparison of different features between MPI and Pthread** under different situations.
4. Investigation between **sequential computing and multi-threads computing**.
5. Comparison between **dynamic scheduling and static scheduling**.

Before the data analysis for the above specific questions, Design approach of will first be introduced.

Part 2: Design Approaches

2-1: Code Design

Pthread Design

For implementation via Pthread, this program basically uses `dynamic scheduling` to assign the workload to each threads and let the root thread do the I/O work. Except for parts of I/O and GUI, my design consists of mainly 3 parts:

1. Division of whole problem
2. Pthreads initialization
3. Parallel computing and dynamic scheduling.

- Devision of whole problem

- This part is mainly for the preparation of the dynamic scheduling. Pixels will be evenly divided into several small blocks. Each block is defined by a basis and length(offset), and being stored in shared memory, so that when dymanic scheduling, each threads can approach to each blocks. The divison here using the **block division**. Therefore, the problem would like:

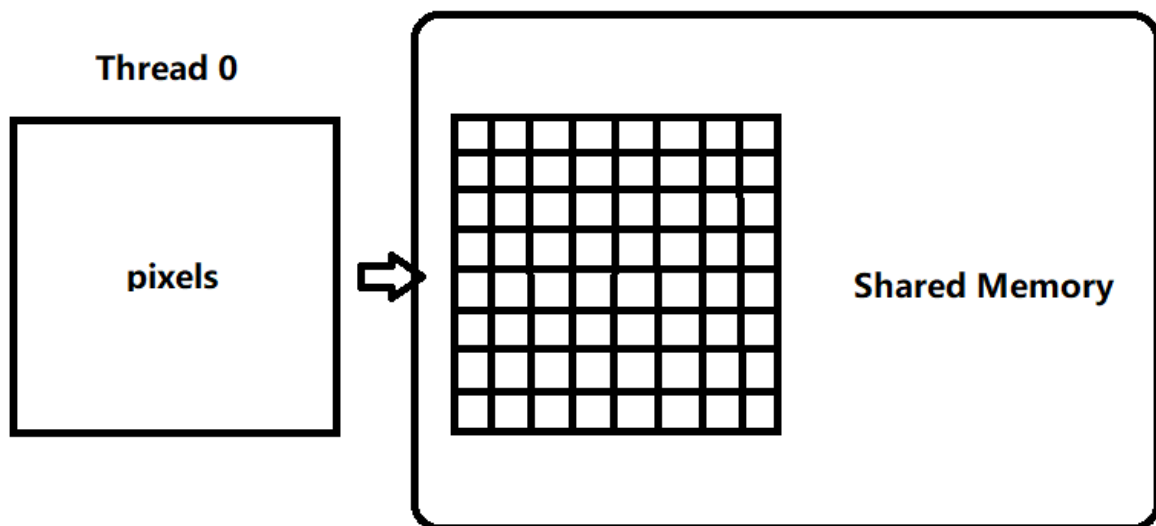


Figure 1: Division of problem into blocks of pixels

- Pthreads Initialization

- Numbers of Pthreads would be initialized and executing the function `calculate()`, which is the function to be executed parallelly. In addition, the `rank(number)` of each thread would be passed as argument.

```
pthread_create(&threads[rank], NULL, calculate, (void *)&thread_ids[rank]);
```

- In addition to the pthread initialization, a mutex lock would also be initialized to protect the shared data. In this way, the same block of pixels can only be calculated once by one thread.

- Parallel computing and Dynamic scheduling

- The basic idea of implementing dynamic scheduling here is:
 - At beginning, each thread competes for the mutex lock to access block of pixels. After one has access the block of pixel, it will release the lock, and other threads would compete for the block next to it, until all the threads are calculating the pixels.

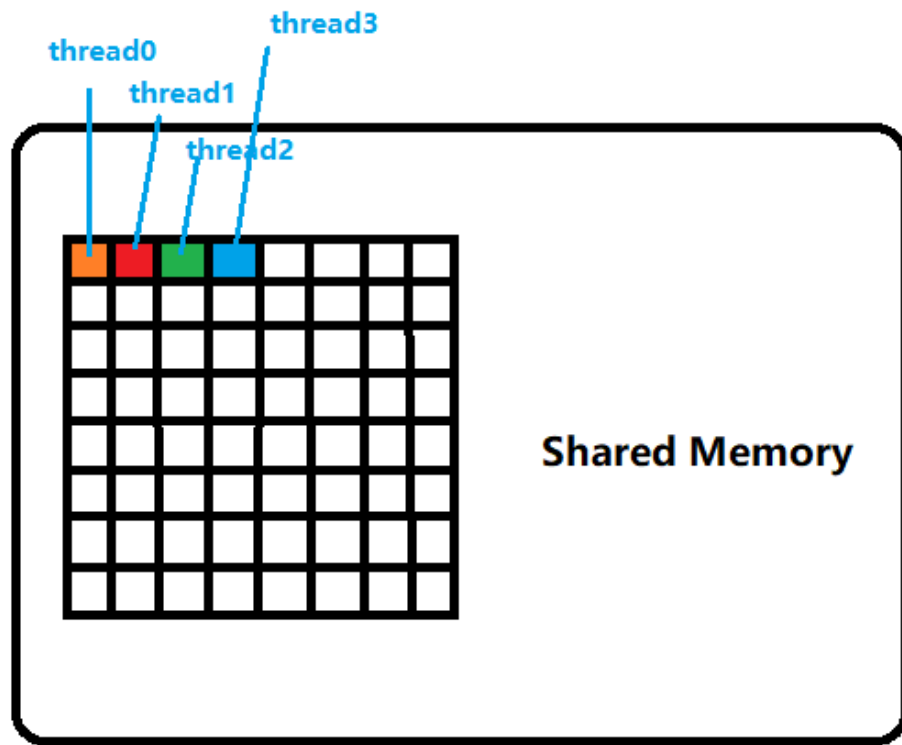


Figure 2a: blocks of pixels are initially assigned to each thread (suppose 4 threads here)

- Then certain thread may finish the computing earlier than others, so this thread would get the mutex lock and being assigned with next block of pixels. Therefore, once one thread finish its local job, it will compete for the mutex lock and get the basis and length of next block of pixels. This strategy will ensure the balanced of the total pixels assigned to each thread, which will reduce the bottleneck and enhance the general performance (the analysis of static scheduling and dynamic scheduling will be investigated later.)

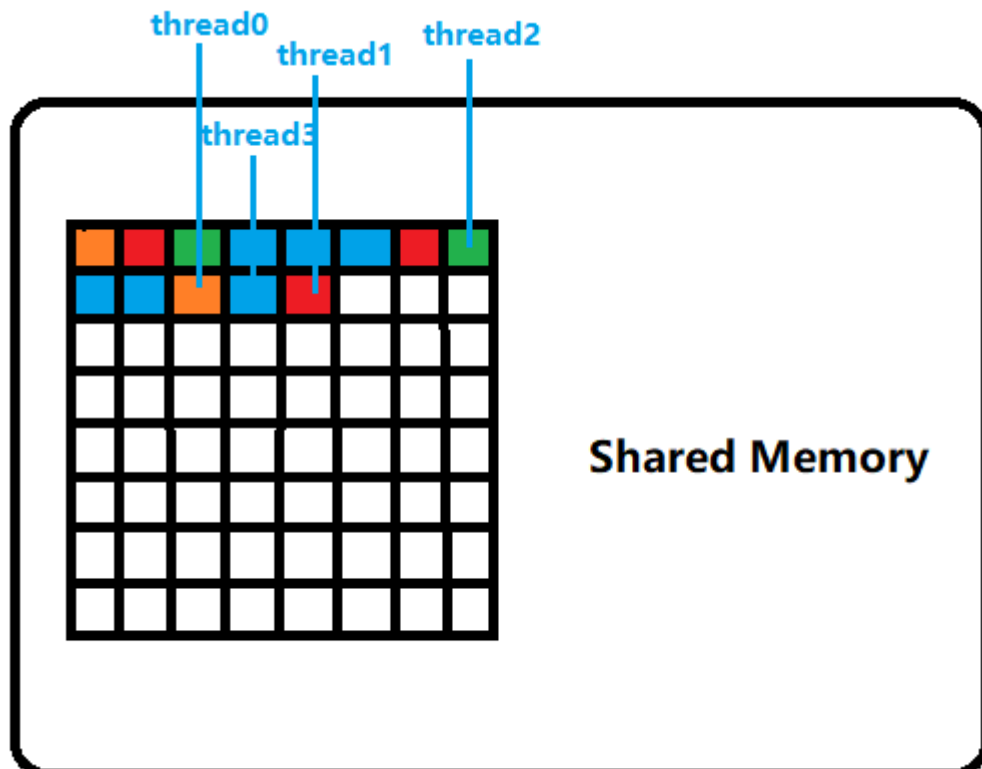


Figure 2b: of pixels distributed to each threads during parallel computing

Remark: as it is shown in the picture, some thread (like thread 2) deal with less blocks of pixels than others, which due to the workload of its blocks of pixels is higher than other blocks of pixels.

- To make sure the workload can be distributed as evenly as possible, number of blocks are set to be 1/1000 of the problem size. (means if number of pixels is 100000000, then it will be divided into 100000 blocks).
- After all the blocks of pixels are calculated, threads will exit and the root thread would be in charge of I/O.

- Static scheduling

- To investigate the difference in performance between dynamic scheduling and static scheduling, I also design a program which is completely same as above idea except for scheduling strategy. It means all the pixels will be assigned evenly to each thread at the beginning.

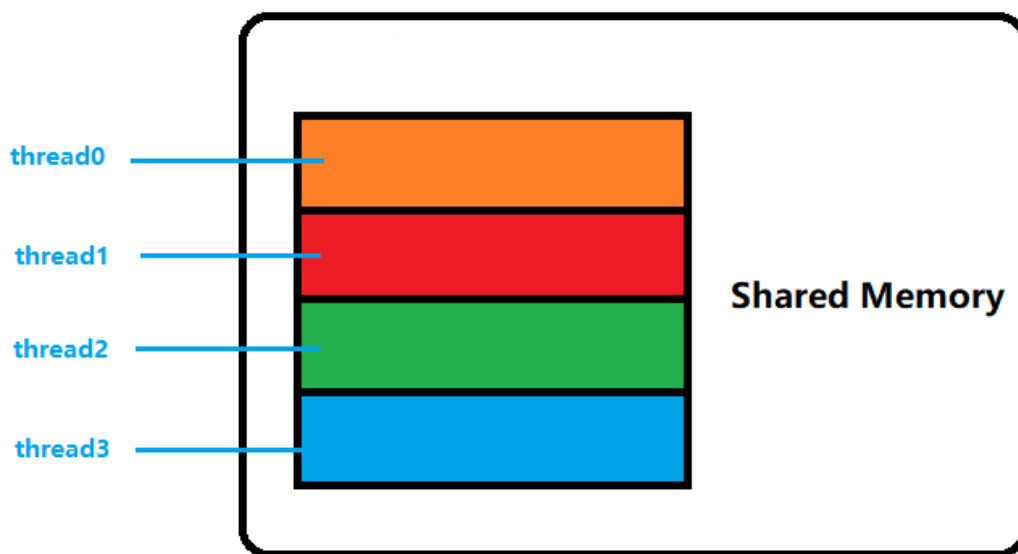
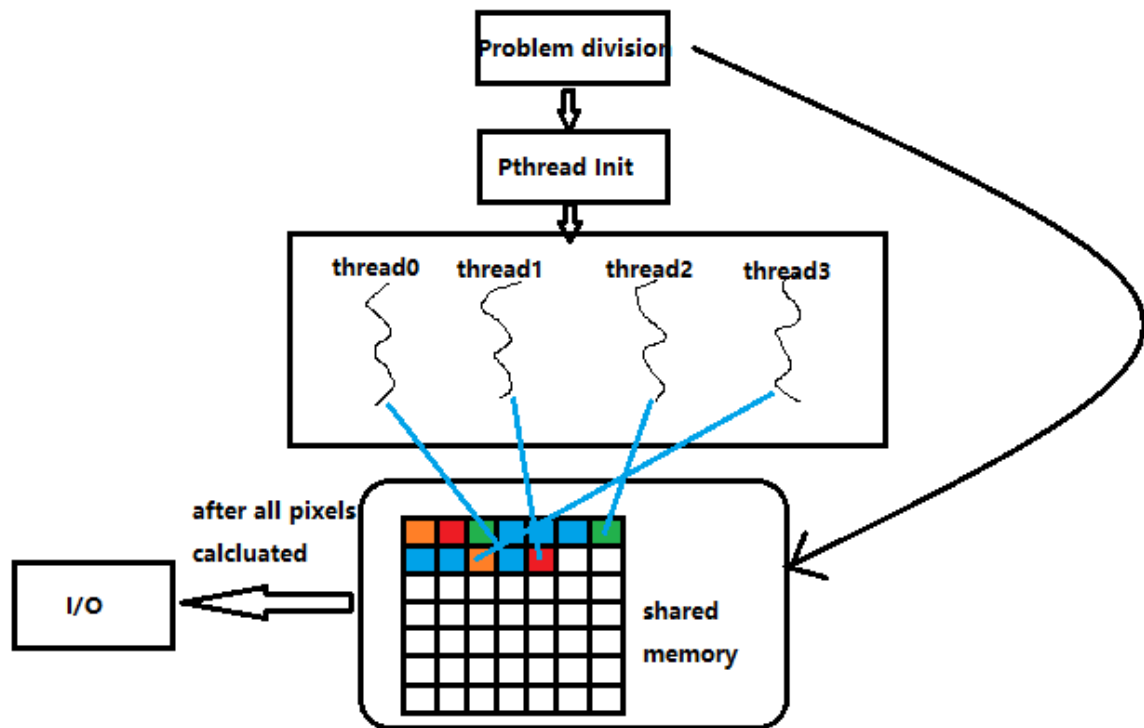


Figure 3: static scheduling with evenly distributed (suppose 4 threads here)

- Overall diagram of Pthread design



MPI Design

Design of MPI program are mainly composed of 4 parts:

- Broadcast of necessary informations
- problem division and parallel computing
- Results gathering
- Output

- Broadcast of necessary informations

- The necessary information here includes: problem size, scale, center of the picture, and arbitrary limit of mandelbrot set computation.

```

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

- Problem division

- First, each process will malloc a memory of Square based on the problem size, and they will directly calculate pixels and store in their local Square
- Based the necessary information broadcast above, all the process will follow the rule of evenly distribution of pixels. They would recognize their own region of pixels to calculate based on their rank and basic informations.
 - After the evenly distribution of pixels, all the processes call the function `calculate()` and perform computation of their local data in parallel and store the results in their local Square .

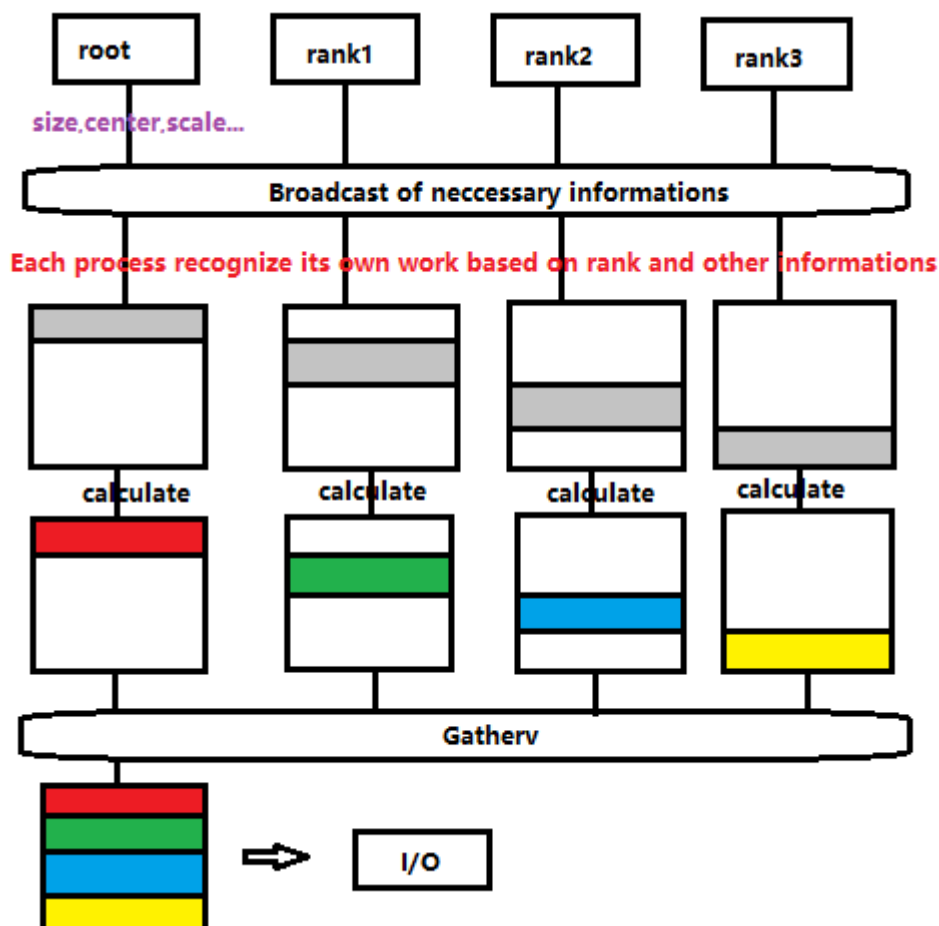
- Gethering

- Because the results each process calculated are stored in their local `Square`, they are required to be gathered to the root process.
- `Gatherv()` is adopted in this program.

```
int * ptr;  
ptr = &sub_canvas.buffer[0];  
ptr += offset_array[rank]; /* ptr points to the local memory */  
int * r_ptr;  
r_ptr = &canvas.buffer[0]; /* r_ptr points to the destination memory */  
MPI_Gatherv(ptr, sub_size_array[rank], MPI_INT, r_ptr, sub_size_array, offset_array, MPI_INT, 0,  
MPI_COMM_WORLD);
```

After the gathering, all the result would be stored in the memory of root process. Finally, the root process would be in charge of I/O.

- Overall diagram of MPI design



2-2 Experiments Design

Totally five topic will be investigated in this projects, which are:

1. **Number of threads/cores** and the performance of parallel computing via MPI or Pthread.
2. **Problem size** and the performance of parallel computing via Pthread.
3. **MPI and Pthread**
4. **Sequential computing and multi-threads computing.**

5. Dynamic scheduling and static scheduling.

This report will explain the idea of experiments design in detail:

- Design 1a: Number of threads and parallel computing performance

- In this design, I set the problem size to be constant and in middle-size (800), then change the number of threads from 1 to 20. What's more, the scheduling strategy I adopted here is static scheduling. (Discussion about static and dynamic scheduling will be conducted in design 5)
- The detail method is shown below:

```
#!/bin/bash
for ((j = 0; j <= 20; j++ ));
do
xvfb-run /pvfsmnt/119010249/csc4005-project2-all/csc4005-project2-pthread/build/csc4005_imgui $j
800 $j
done
```

- Notice, in `xvfb-run ./csc4005_imgui arg1 arg2 arg3`, `xvfb-run` means running without gui, `arg1` means number of threads, `arg2` means the square root of problem size (since the picture is 2-dimension, the number of pixels to be calculated would be arg2^2), `arg3` is the number of blocks(problem division for dynamic scheduling. Since we use static scheduling here, the number of blocks are set to be equal to the number of threads).
- Then, I observe the tendency of performance change (using speedup compared with sequential one here) as the number of threads change.

- Design 1b: Investigate the relationship between number of threads and parallel computing performance in a more general way

- What's more, to make the results more general. This project further conducts an experiment with wider choices of problem size and number of threads. The result will be also shown in next part. To observe this result, we can investigate a more genral relationship between number of threads and parallel improvement.

- Design 1c: Number of cores and parallel computing performance of MPI

- Except for the implementation way of parallel computing (using MPI), this design of experiment is totally similar with design 1a. 1-20 cores are tested on the problem with size equal to 800 using static scheduling. The code of running this experiment is:

```
#!/bin/bash
for ((j = 0; j <= 20; j++ ));
do
xvfb-run mpirun -n $j /pvfsmnt/119010249/csc4005-project2-all/csc4005-project2-
pthread/build/csc4005_imgui 800
done
```

- After getting the result, the tendcency of performance change(using speed up compared with sequential computing) will be observed.

- Design 2: Problem size and the performance of parallel computing via Pthread

- In this design of experiment, The problem size are selected to be three modes: Small (800, 1600, 3200), medium(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, 128 threads.

- Also, static scheduling and dynamic scheduling implementation would be tested and observed separately.
- Then, the tendency of performance change as the problem size increase will be observed.

- Design 3: MPI and Pthread

- In this design, I mainly focus on the comparison between MPI and Pthread on the parallel Mandelbrot set computation. To make the comparison fair, the scheduling strategy of MPI and Pthread are both static scheduling. What's more, the comparison mainly focus on two dimensions.
- The first dimension is the number of cores/threads. 1, 2, 4, 8, 16, 32, 64, 128Cores/threads will be tested on the same problem and compared.
- The second dimension is the problem size. Small(50, 200, 400), middle(800, 1600, 3200) and large size(5000, 10000, 20000) will be tested.

- Design 4: Comparison between Sequential computing and multi-threads computing

- To compare the sequential performance and parallel performance of Pthread in a broader way. Sequential computing will be first performed. Then 7 choices of threads(2, 4, 8, 16, 32, 64, 128) are tested on 9 different problem sizes (50, 200, 400, 800, 1600, 3200, 5000, 10000, 20000). Then we expected to get a 7*9 table of computing speed.
- To compare each choices of cores with the sequential one, I make the speedup (speed of pthread computing/ sequential computing) as the criterion. Then, the observations focus on:
 1. Whether Pthread has a better performance than sequential one under different situations.
 2. How much can the Pthread improve the performance

- Design 5: Comparison between Dynamic scheduling and Static scheduling

- In this design, I implement two ways of scheduling, which are static scheduling and dynamic scheduling. To compare their performance, problem with size equal to 800 is tested on 8, 16, 32 threads via dynamic scheduling and static scheduling.
- The analysis mainly focus on two sub-dimensions.
- The first dimension is the overall performance. The program's speed of computing the whole problem will be compared.
- To observe whether each thread is assigned with fair workload, the second dimension is the variance of the duration time of each threads. Duration time of each thread will be recorded and finally compare the variance of them.

- Remark

- Fives disgned above will be performed and the results will be analysis in the following part. There are five analysis corresponding to these five designs.

Part 3: Performance Analysis

Analysis 1a: number of threads and performance change

- Results

- Based on the design 1a, we get the result about the relationship between performance change and number of threads. The tatble below shows the results:

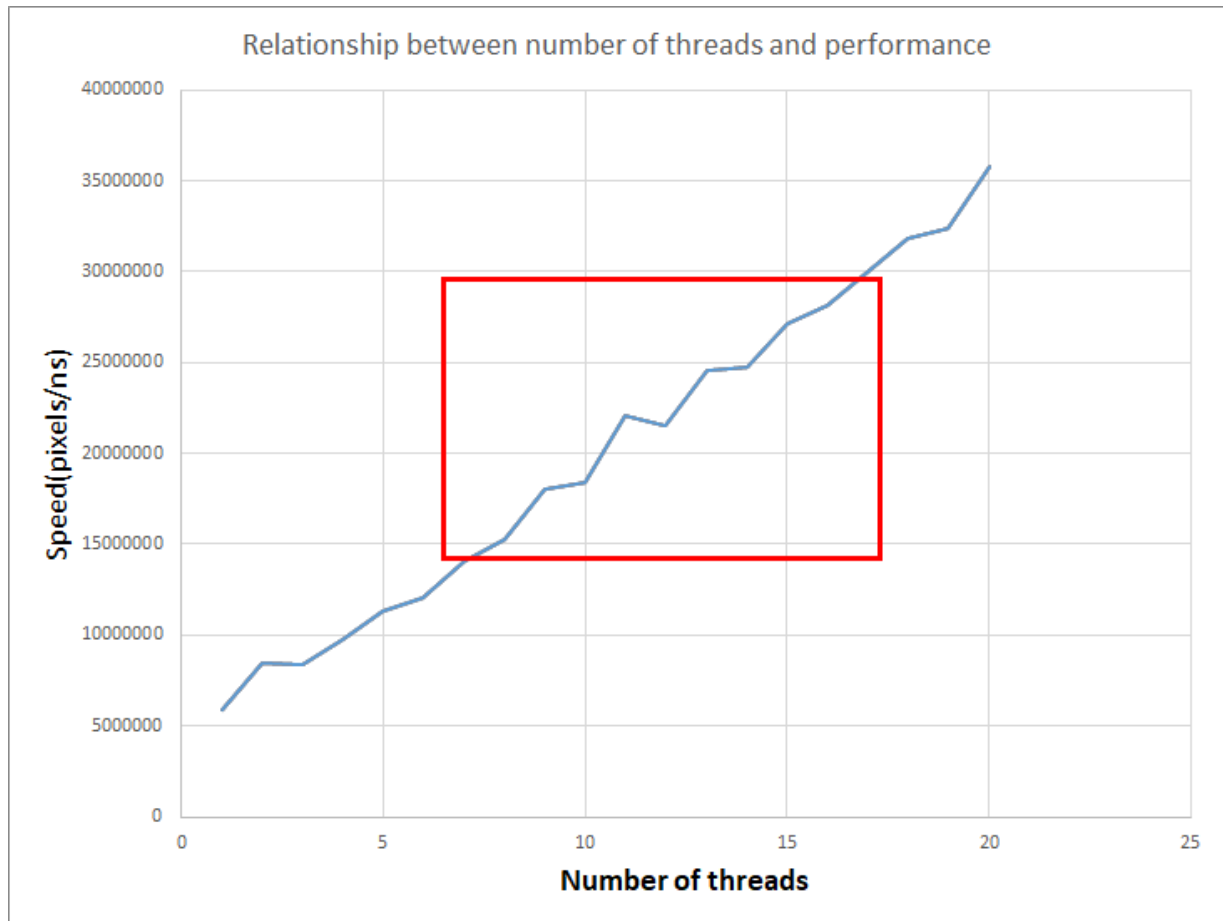


Figure 4. The relationship between number of threads and computing performance

- There are two features we can attain from the figure:
 - First is that the performance(using speed(pixels/ns)) of parallel computing is generally increasing in nearly linear way(the slope is around 0.318) as the number of threads increases.
 - Another interesting thing is: the increase is not smooth as the number of threads increases by one (as it can see in the red block in Figure 4). The tendency is nearly: increase-stop-increase-stop-... And this phenomenon is also noticed in other problem size and number of cores.

- Conclusion

- In general, applying more threads for the Mandelbrot set computation would increase the parallel computing performance and the increasing tendency is stable (slope is nearly 0.318).
- However the increase tendency is not smoothly and there is a periodic change (increase-stop-increase-stop).

- Discussion

- 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.

- Regarding to the not smoothly increasing, it can be explained by the shortback of static scheluing on computing the mendelbrot set. As it is shown in Figure 5, the pixel's workload is not evenly distributed. However, the static sheduling used in this experiment is **Stripe partition**, whose bottleneck of parallel computing may not be reduced significantly as the number of threads increase by 1 because of the feature of Mandelbrot Set. The figure showing below inllustrate the number of threads increase from 3 to 4. And it is found that the workload of the bottle-neck thread may not decrease significantly as the number of threads plus one.

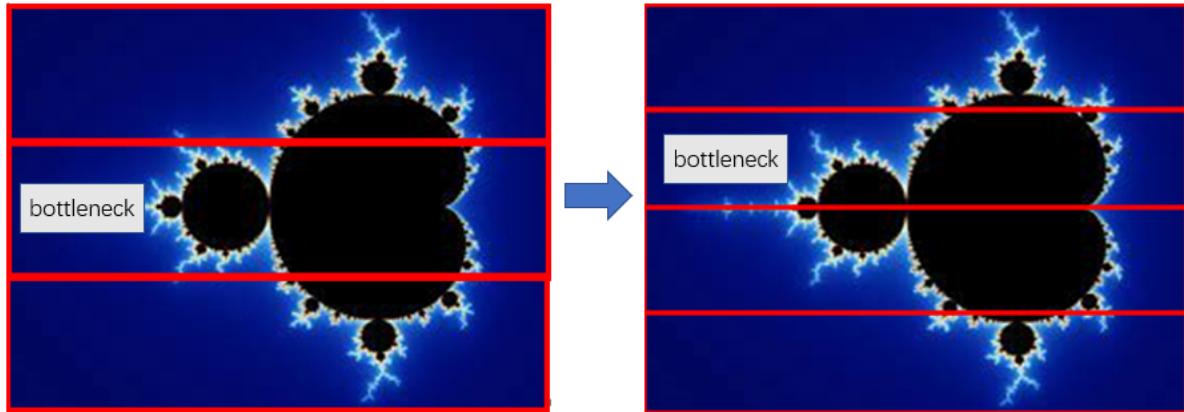


Figure 5. Problem with Stripe partition: e.g. when number of threads increase from 3 to 4

To verify this idea with data. I conduct a profiling of each threads' computing time when the number of threads equal to 8, 9, 10.

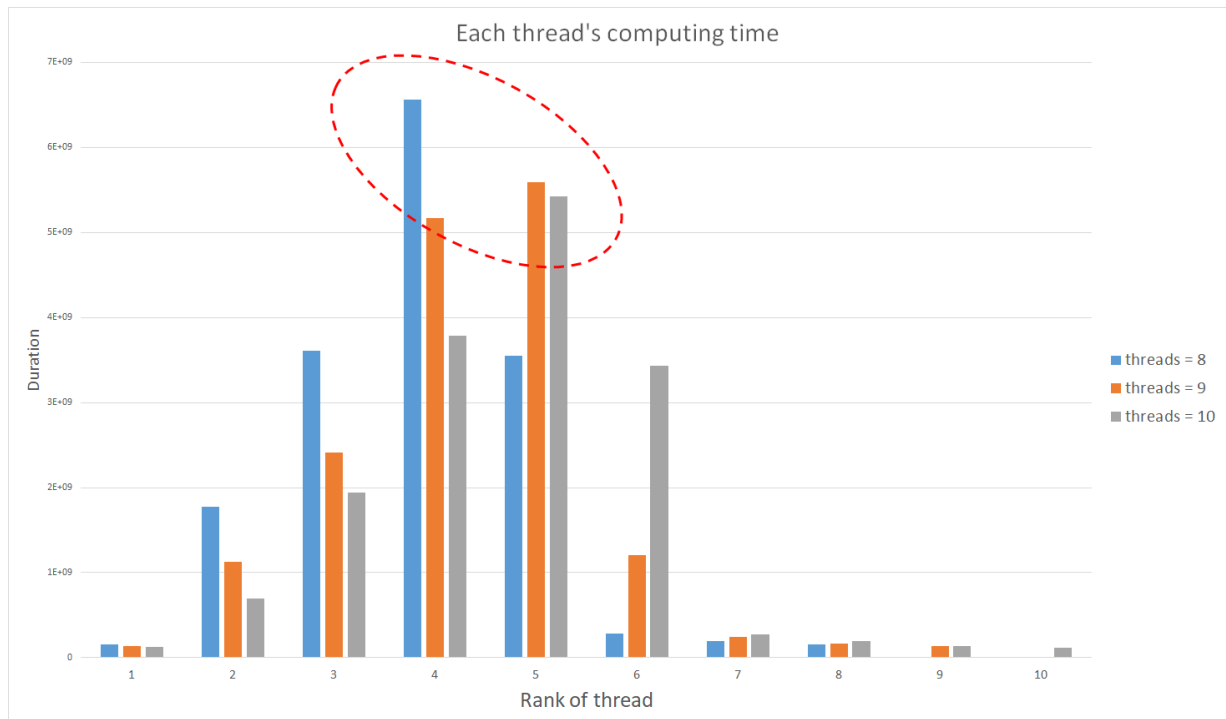


Figure 6. Each thread's local computing time when number of threads are 8, 9, 10

Figure 6 shows each threads computing time when the total threads number are 8, 9, 10 respectively. The region highlighted by dotted line includes the the threads which are bottleneck. The blue rectangle is the bottleneck when number of threads is 8. The orange rectangle is the bottleneck when number of threads is 9. The grey one is the bottleneck when number of threads is 10.

It is obvious that when number of thread increase from 8 to 9, the workload of bottle neck thread decreases significantly. However, when number of threads increase from 9 to 10, the worload of bottleneck thread change slightly. This result verifies the idea of stripe partition's drawback in Mandelbrot set computation and explains the non-smooth increasing of performance when number of threads increase by 1.

Analysis 1b: number of threads and performance change(more general)

- Results:


 image-20211028212232043

Figure 7. relationship between number of threads and multi-thread computing performance

- The figure shows that as the number of threads increases (from 1 to 128), the performance would increase firstly dramatically. Then after certain critical point(around 64 in this experiment), the improvement will slow down and cease.

- Conclusion

- When the number of threads are not large, increasing amount of threads will significantly improve the performance of Mandelbrot set computation. However, after certain limited point (64 in this experiment), the increasing tendency will slow down and finally cease. Therefore, it is low-profit to further increase the number of threads.

- Discussion

- 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.

Analysis 1c: number of cores and performance change

- Results:

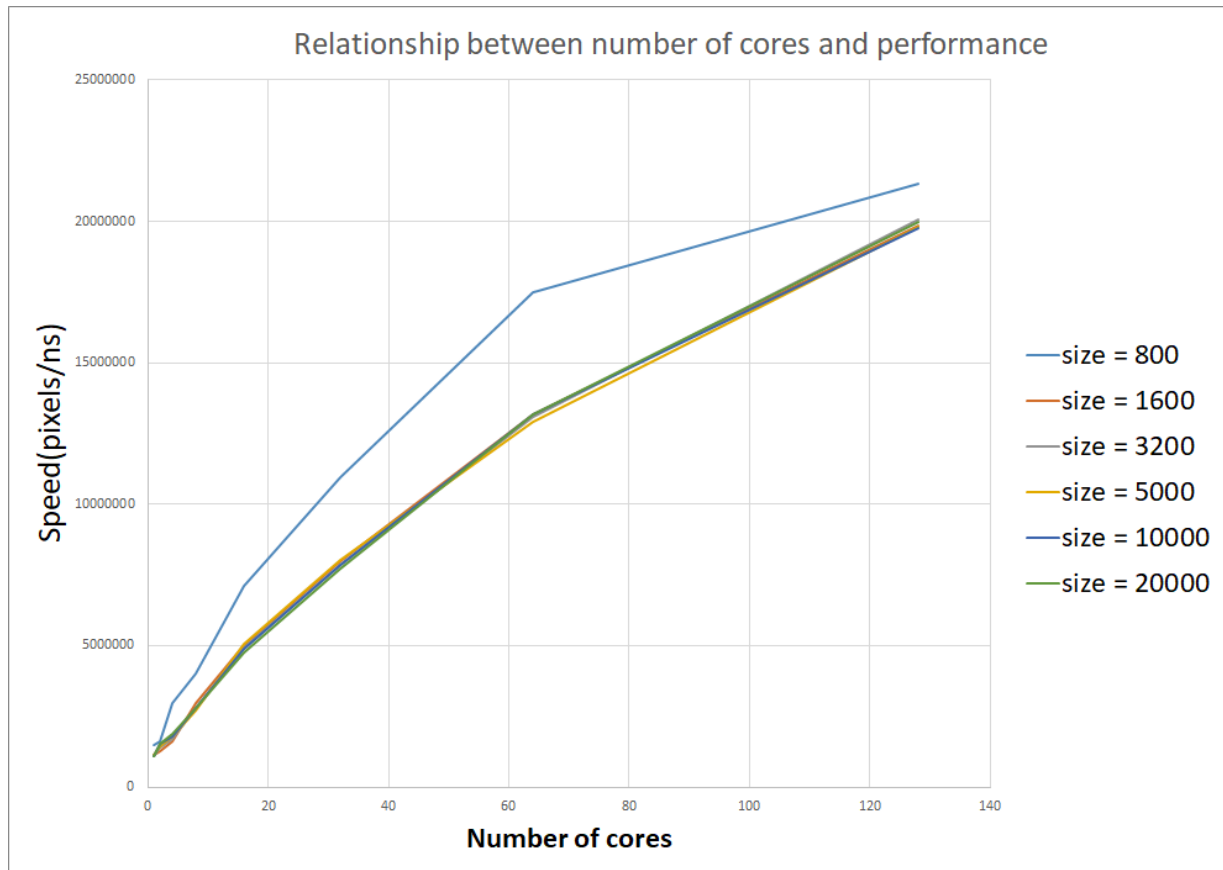


Figure 8. Relationship between number of cores and performance improvement via MPI

- This experiment is using MPI. The Figure shows that as the number of cores increases, the performance would keep increasing nearly linearly. After certain point (64 cores in this experiment), the increase will slow down by a small amount but would not cease.

- Conclusion

- When using MPI to implement the Mandelbrot set computation, when number of cores increase, the performance will increase with slightly slow down after certian point (64 cores in this problem).

- Discussion

- The conclusion above can be explained that: 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. The slightly slow down due to the affect of the serial section mentioned by Amdel's law: as the number of cores increase, the serial part will perform a more significant row.
- However, why the increasing tendency doesn't cease like the pthread's experiment when the number of threads is relatively large? This research question would be discussed in comparison between MPI and Pthread.

Analysis 2: problem size and performance change

- Result and Conclusion:

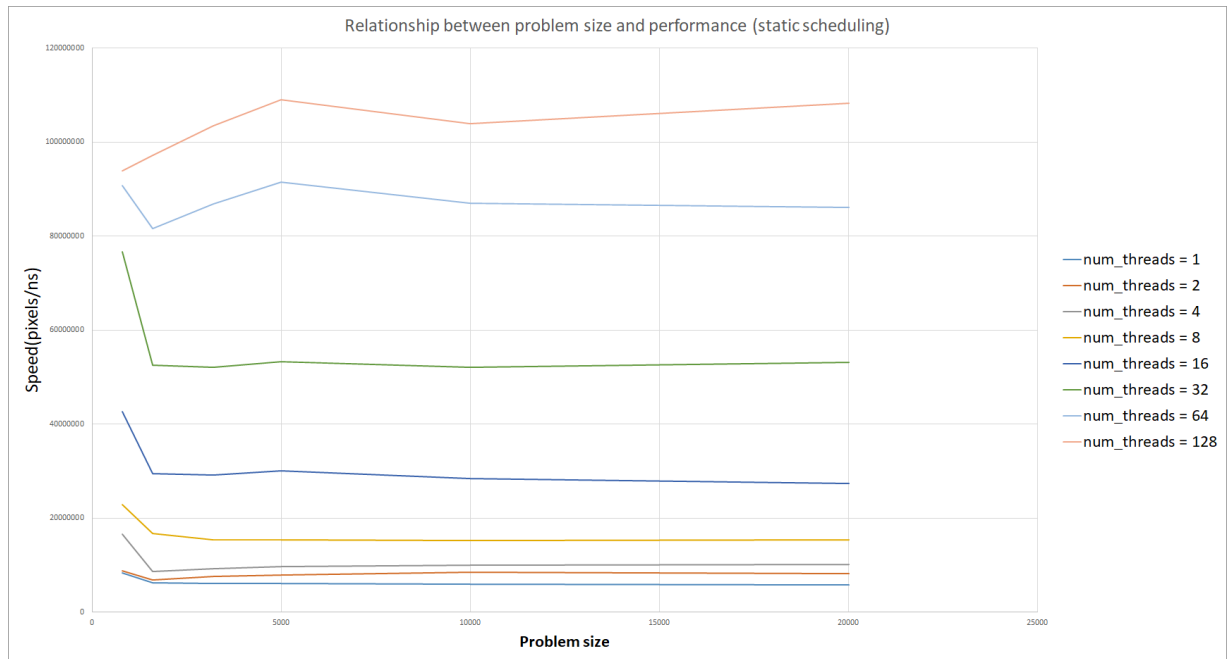


Figure 9a. Relationship between problem size and performance change(static scheduling)

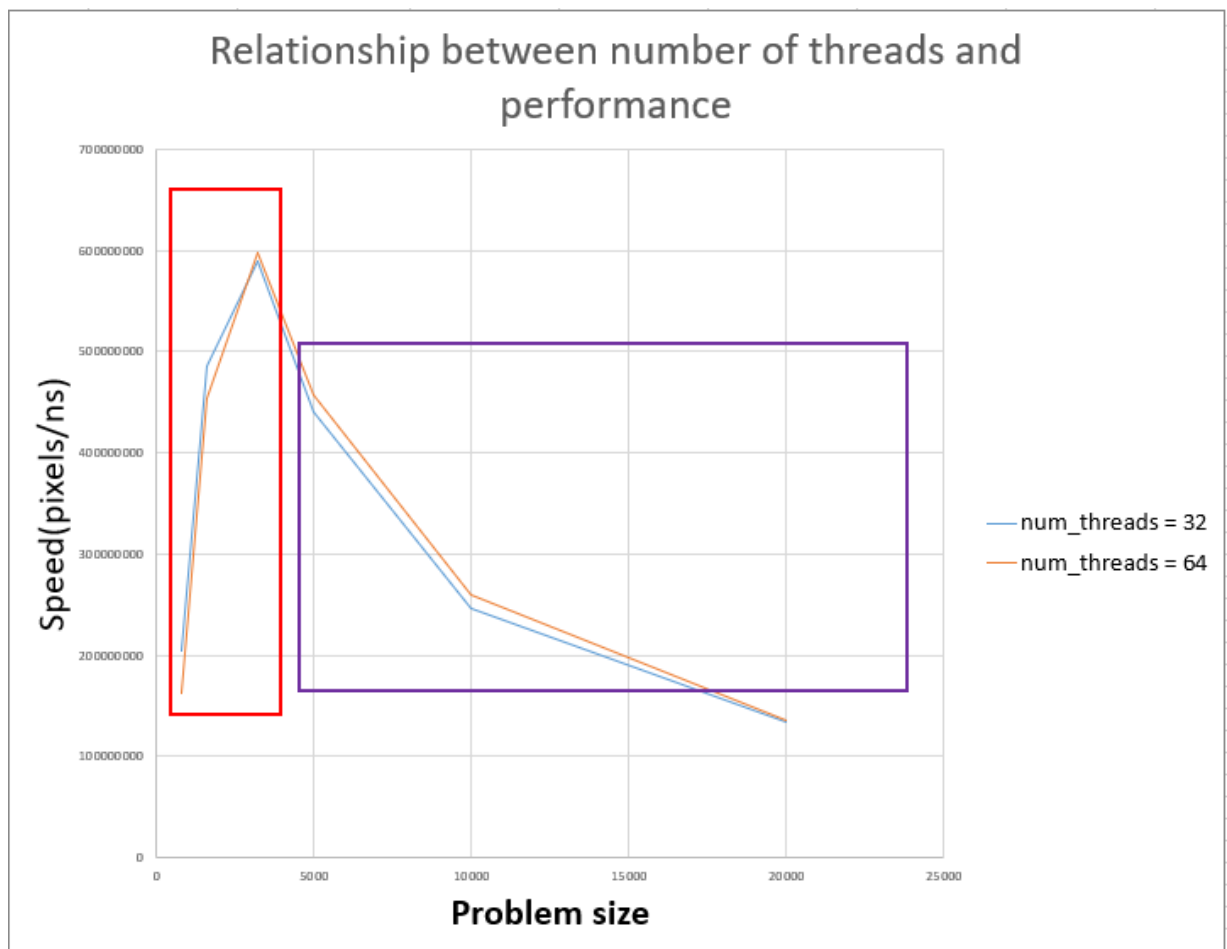


Figure 9b. Relationship between problem size and performance change(dynamic scheduling)

- Result and conclusion 1: As it is shown in the figure 9a, as the problem size change the performance would generally keep flat (for static scheduling). That means the problem size would have little influence on the parallel performance.

- Result and conclusion 2: Figure 9b shows the when the problem size is relatively small (between 800 to 3200), as the problem size increases, the performance increases dramatically. However, after problem size equals 3200, for the relatively middle and large problem size, the performance would decrease greatly.

- Discussion

- As for the Result and conclusion 1, this phenomenon can be explained by the bottleneck of static scheduling. The overall speed depends on the speed of bottleneck thread. Because of the static scheduling, the portion of workload assigned to each thread remains the same, regardless to the increasing of problem size. That means the bottleneck thread would be assigned with the same ratio of workload though the problem size increases. The bottleneck thread's speed would not change since it is assigned with same portion of workload (for example, if the overall workload is double, the workload it should deal with will also double). Therefore, the overall speed will remain with little changes though the problem size increases.
- As for the Result and conclusion 2, for dynamic scheduling, increasing performance when problem size is relatively small (800-3200) can be explained by the Gustafson's law, which claims that as the problem size goes up, parallel computing will have a higher performance because of the formulation:

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

As the problem size increase, s in the above function will become less and less. Therefore, the speedup are more close to the number of cores. However, there exists a boundary for the improvement that the increase of problem size brings meanwhile the communication overhead for dynamic scheduling is becoming gradually significant. These two factors in general decrease the performance of parallel computing when the problem size is becoming larger.

Analysis 3: MPI and Pthread

- Result and conclusion: Overall comparison

Pthread / MPI	50	200	400	800	1600	3200	5000	10000	20000
1	3.274929547	8.874748552	10.66276175	5.651939655	5.509142053	5.557803468	5.488789238	5.372727273	5.37037037
2	3.179462237	10.0205529	12.12928737	5.474282888	5.366286439	5.447247706	5.470790378	5.578947368	5.407894737
4	1.167275433	8.166195739	12.87205426	5.645782479	5.395604396	5.481132075	5.422535211	5.491712707	5.462365591
8	0.719004559	1.595623433	12.28354884	5.698761486	5.592373792	5.532796318	5.625688073	5.496402878	5.510791367
16	0.376736737	1.100259532	4.004132136	6.007314877	5.876314951	5.906493506	5.944554455	5.831622177	5.780590717
32	0.13882947	1.207906259	4.164709784	7.006724163	6.592102626	6.646650327	6.639875389	6.625158831	6.888601036
64	0.111092515	0.496535153	1.62153789	5.188666301	6.195799951	6.625732422	7.072284499	6.612462006	6.541033435
128	0.060186466	0.25646655	0.887241685	4.405525111	4.89353306	5.159917092	5.507952537	5.255308392	5.415

Figure 10. Overall comparison between Pthread and MPI

- Data in Figure 10 shows (Speed of Pthread / Speed of MPI). Blue block means Pthread outperforms MPI and Brown block means MPI means MPI outperforms Pthread. Meanwhile, darker the color, the greater advantage one got.
- The results shows that Pthread outperform MPI regarding to Mandelbrot set computation in general. Especially when number of threads/cores is in small scale and the problem size is relatively large. Moreover, MPI is better when number of threads/cores is in large scale and the problem size is relatively small.

- **Discussion:**

- Pthread outperform MPI in general can be explained by the communication overhead. Because Pthread uses shared memory while MPI uses `Bcast()` and `Gatherv()` to communicate, each thread's computation for Pthread is more independent with each other than parallel computation via MPI. Therefore, when problem size is large, each communication will take longer time (more data to be transferred).
- However, when number of cores/threads increase, the advantage of Pthread become weaker and worse than MPI when number problem size is relatively small and number of cores/threads are in large scale. The architecture of cores may result in this phenomenon. As Figure 11 shows below, when the number of threads are in large scale, their access to memory may cross the link, and thus the access time would be much greater.

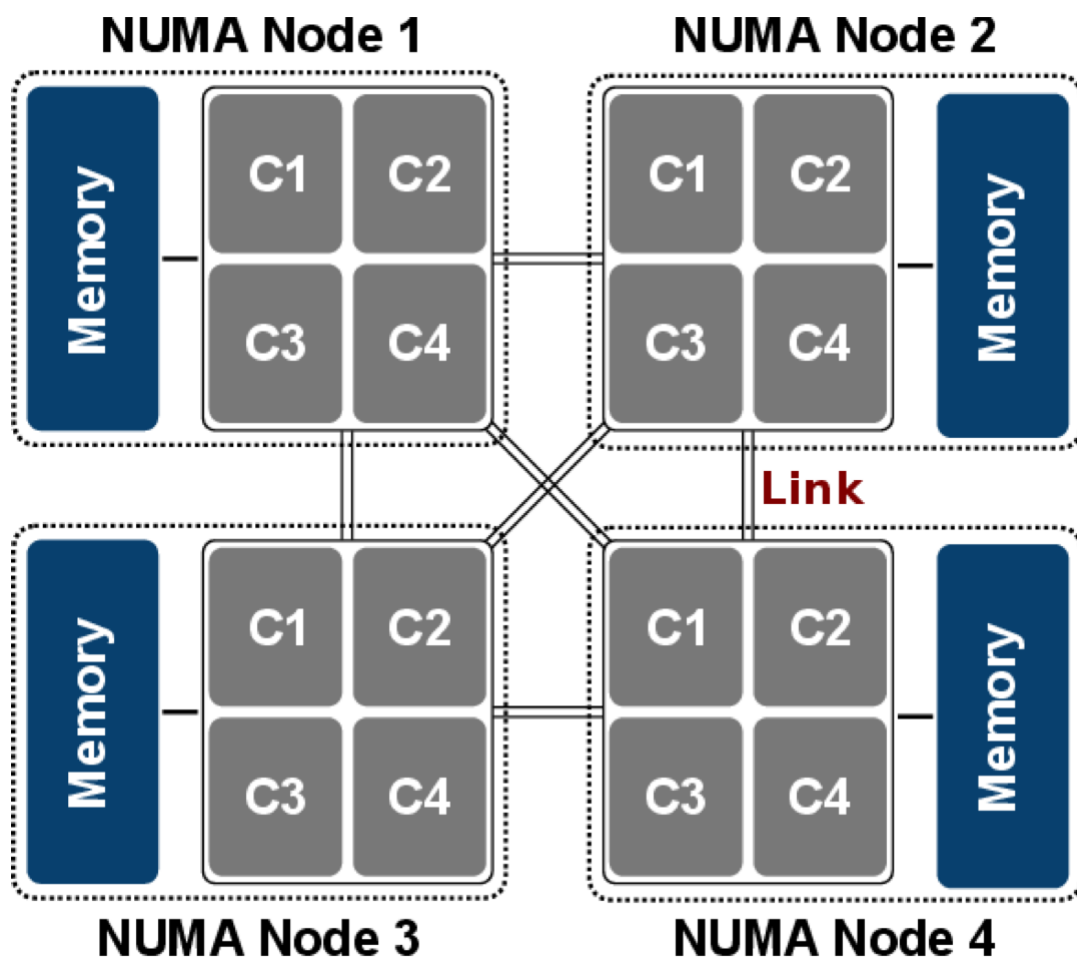


Figure 11. Simplex architecture of NUMA

Analysis 4: Pthread and sequential

- **Result and conclusion:**

Pthread / MPI	50	200	400	800	1600	3200	5000	10000	20000
1	1	1	1	1	1	1	1	1	1
2	0.965897095	1.817420893	1.916717022	1.055195424	1.080929283	1.235049402	1.300653595	1.434856176	1.417241379
4	0.446056857	2.217810318	3.093033582	1.984366063	1.378861374	1.510660426	1.572712418	1.681895093	1.751724138
8	0.315956941	0.573486882	4.0138248	2.719542421	2.65841205	2.50026001	2.504901961	2.585448393	2.64137931
16	0.176636657	0.464251286	1.592502564	5.088751192	4.706152668	4.730109204	4.905228758	4.805414552	4.724137931
32	0.06091464	0.532624576	1.724428697	9.138798856	8.396221598	8.46125845	8.706699346	8.822335025	9.168965517
64	0.046880576	0.218962753	0.718798066	10.81458532	13.03063569	14.11284451	14.94771242	14.72419628	14.84137931
128	0.023444005	0.115483873	0.39917509	11.18894185	15.51263722	16.82943318	17.82434641	17.58883249	18.67241379

Figure 12. Comparison between Pthread and Sequential computing

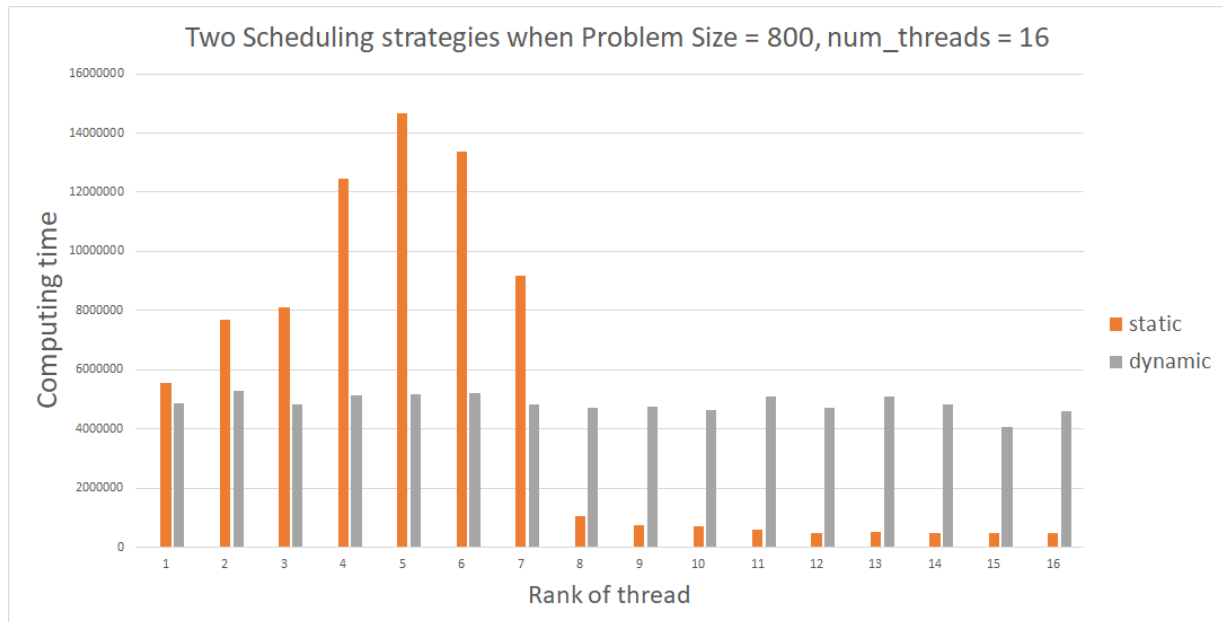
- In Figure 12, each block means the speedup compared with the sequential computing. The green color highlight the situation multi-thread computing is worse than sequential computing. The purple color highlights the situation that multi-thread computing outperforms the sequential computing.
- We can make the following conclusions:
 - The multi-thread computing is better than sequential computing when problem size is large (> 400 in this experiment). When the number of threads is larger and problem size is greater, the advantage of multi-thread computing is more obvious.
 - When problem size is small (50 - 400 in this experiment), it is not wise to use many threads for the mandelbrot set computation because the performance is worse than sequential computing.

- Discussion

- Two discussions can be drawn to explain the conclusion above:
 - When the problem size is bigger, it means the parallel section of the whole problem takes a more and more significant role in the whole computing procedure. Therefore, distributed the large scale work to each threads would distribute the workload and reduce the total computing time.
 - 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. Therefore, it will be worse than directly using sequential computing.

Analysis 5: Dynamic scheduling and static scheduling

- Remark: part of the comparison is included in the analysis 1a, which discuss that stripe partition of static scheduling would cause the performance to be increased in a non-smooth way. In analysis 5, there will be a more general analysis between scheduling and static based on the experiment of design 5.



	Performance(speed)	Variance among each threads
Static scheduling	22822400	5324702
Dynamic scheduling	117435200	303452

Figure 13. Comparison between static scheduling and dynamic scheduling

- From the histogram, it is obvious that the workload of each thread for static scheduling is dramatically unbalanced, while the workload of each thread for dynamic scheduling is relatively quite balanced. (We can also acknowledge it by variance, where variance of static scheduling is much bigger than dynamic scheduling).
- Also, the performance of dynamic scheduling outperforms static's greatly (5.145 times larger).

- Discussion:

- The higher performance can be explained by the bottleneck thread. When the workload is distributed very unbalanced, one thread may be assigned with a workload which is largely heavier than the average workload. Therefore, although other threads have finished their local work, they have to wait for the bottleneck thread to complete, thus the total computing time depends on the bottleneck thread.
- 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.

Supplementary

- Since there are two implementations (MPI, Pthread), there are two separately directory: `/csc4005-project2-MPI` and `/csc4005-project2-pthread`

Pthread: compile and run

- cd to the corresponding file:

```
$ cd /csc4005-project2-pthread
```

- compile the code:

```
$ mkdir build  
$ cd build  
$ cmake .. -DCMAKE_BUILD_TYPE=Release  
$ cmake --build . -j4
```

- run the program with 16 threads and problem size of 800

```
$ ./csc4005_imgui 16 800 640
```

Arguments: my program need to pass argument to it. The format is like `./csc4005_imgui arg1 arg2 arg3`, where `arg1` is the number of threads, `arg2` is the problem size, `arg3` is the number of blocks for dynamic scheduling (640 means divide the all pixels into 640 blocks).

MPI: compile and run

- cd to the corresponding file:

```
$ cd /csc4005-project2-MPI
```

- compile the code:

```
$ mkdir build  
$ cd build  
$ cmake .. -DCMAKE_BUILD_TYPE=Release  
$ cmake --build . -j4
```

- salloc the cores on Slurm

```
$ salloc -n32 -N1 -t10
```

- run the program with problem size = 800 by 32 cores

```
$ mpirun ./csc4005_imgui 800
```

Arguments: my program need to pass argument to it. The format is like `mpirun ./csc4005_imgui arg1`, here `arg1` is the problem size.