香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

CSC4005
Parallel Computing

Project 1

Author	**Student ID**
Luo Haoyan	119010221

October 31, 2022

Contents

1 Introduction

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

$$Z_{k+1} = Z_k^2 + c$$

where Z_{k+1} is the $(k+1)_{th}$ iteration of the complex number $Z = a + bi$ given initial value 0 and c corresponds to the position of the given point.

However, if we apply the sequential method to calculate it, The compuation time will be quite large with the image $size^2$ increase. Theoretically, Sequential computation will calculate Pixels one by one if we pick a fixed side length for a picture. The basic problems and tasks are: given a number set, and the program should perform the parrallel computing strictly following the rule of set calculation for whatever the size of picture and number of cores/threads. Running the parrallel computation program in the cluster and verify in vm to get the experimental data and analyze the performance for specific situation such as different cores/thread and different input size.

Thus, this project requires two parallel version of Mandelbrot set computation program using MPI and Pthread. Comprehensive experiments are conducted and performance under different configurations in four dimensions are analyzed: performance of Sequential program vs. MPI program vs. Pthread program; different size of output images (problem size); different number of threads/cores used; dynamic scheduling and static scheduling. You can check the appendix for the steps to run my program.

2 Method

2.1 MPI Version

Figure 1 shows the overview structure of my MPI program. The design of this program mainly include the following parts:

- MPI initialization and define new data structure for MPI
- Broadcast of neccessary information
- Problem division and parallel computation
- Data gathering

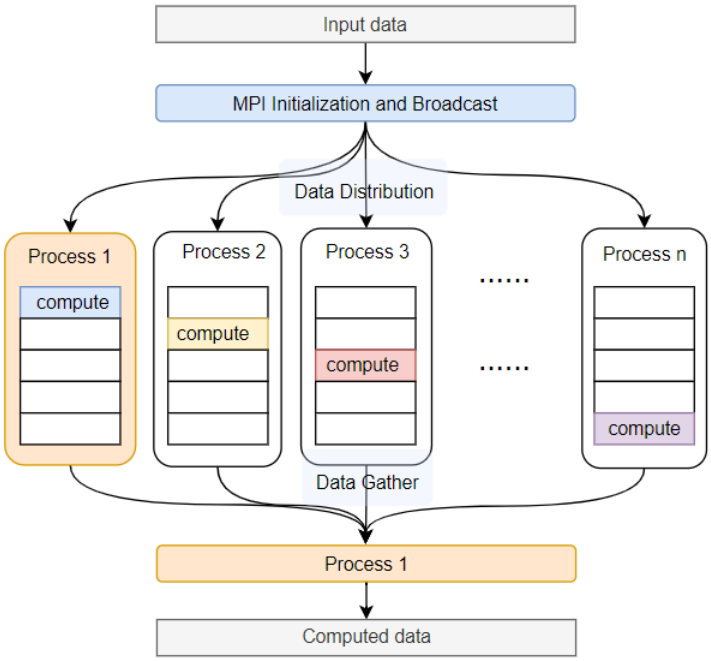

Figure 1: *Overall structure of MPI program*

2.1.1 Initialization and Broadcasting

The necessary information that needs to broadcast here includes the element counting array and displacements array (since I use scatterv in the following data allocation part instead of scatter). The corresponding codes are shown below:

```
MPI_Ibcast(element_count, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
MPI_Ibcast(displacements, world_size, MPI_INT, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Another important thing is that we need to define a new MPI datatype that stores the point type (which contains the data of the coordinate and color), part of the corresponding codes are shown below:

```
MPI_Datatype types[3] = { MPI_INT, MPI_INT, MPI_FLOAT };
MPI_Type_create_struct(3, lengths, displacements, types, &PointType);
MPI_Type_commit(&PointType);
```

In the codes above, PointType is created that can be used in the data allocation and gathering process which would envoke MPI functions.

2.1.2 Data Allocation and Problem Division

Following the structure of the template, the root process (process 1 in the overall figure) will call master function and all other function will call slave function. In my implementation, I exploit the functionality of non-blocking operation to improve the performance. The corresponding codes are shown below:

```
    MPI_Iscatterv(data, element_count, displacements, PointType, my_element,
↪   num_my_element, PointType, 0, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
```

This code snippet is the non-blocking scatter function called in the master process, where PointType is the predefined MPI structure. Note that we have to call the wait function in all processes because we have to ensure the data is allocated properly before they are computed.

2.1.3 Data Gathering

Since the results of each process are calculated and stored in their own local array, they need to be gathered to the master process. Following the non-blocking implementation above, the following code snippet serves as the gathering function:

```
    MPI_Igatherv(my_element, num_my_element, PointType, data, element_count,
↪   displacements, PointType, 0, MPI_COMM_WORLD, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Similarly, we need to call the wait function (especailly on root process) so that when GUI is plotting it would not miss the data that is calculated in other processes.

2.2 Pthread Version

2.2.1 Static Scheduling

The overall structure of pthread static scheduling implementation is shown below. The mechanism of static scheduling is relatively simple. In this implementation, all the pixels will be assigned evenly (or nearly evenly) to each thread at the beginning (which is defined in the worker function that would be call by each thread). The following code snippet shows how the program allocate different work to differnt worker function in each thread and join them together to have the final results.

```
    for (int thd = 0; thd < n_thd; thd++) pthread_create(&thds[thd], NULL, worker,
↪   &args[thd]);
    for (int thd = 0; thd < n_thd; thd++) pthread_join(thds[thd], NULL);
```

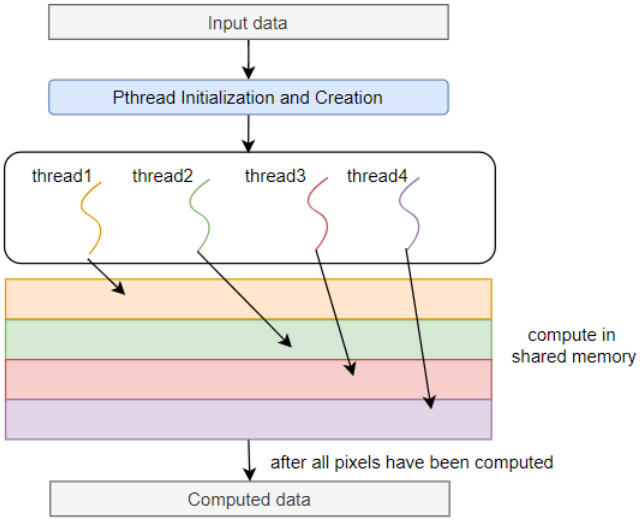

Figure 2: *Overall structure of Pthread static scheduling program, here 4 threads are used as an example*

2.2.2 Dynamic Scheduling

In the dynamic scheduling version, the key part is that the computing is divided into different small blocks (in my program, the block size is set to 100, which means each block contains 100 data points that need to be computed). The total number of blocks would be $size^2$ / 100. The overall structure can be shown below.

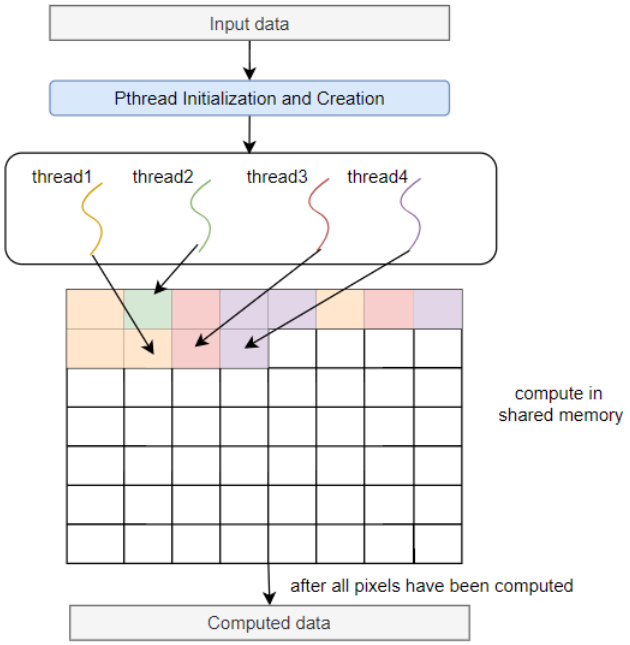

Figure 3: *Overall structure of Pthread dynamic scheduling program, here 4 threads are used as an example*

The basic idea of this dynamic scheduling is shown in the following bullet points:

1. To record whether the program finishes computing all the blocks by different threads, an indicator is created to record the process at the very beginning:

   ```
   int current_block_num = 0;
   ```

   Note that this variable would need to be access by different threads whenever a thread wants to compute (need this indicator to determine which data to compute), thus mutex mechanism need to be implemented to protect this variable. (The data itself actually do not need to protect since the number of data in a blocks and the displacements of the data blocks are predefined, so as long as the thread is accessing the right block, no data read/write conflict would occur)

2. At the beginning, each thread competes for the mutex lock to access the indicator variable and do the incremental operation on it (which means the current block is computed). After it has its block indicator variable, it can unlock the mutex so that other threads can compete and access the current block number. The following code snippet shows the corresponding opertaions:

   ```
   pthread_mutex_lock(&mutex);
   while (current_block_num != block_size) {
       int num_my_element = element_count[current_block_num];
       int displacement = displacements[current_block_num];
       current_block_num++;
       pthread_mutex_unlock(&mutex);
       Point* p = data + displacement;
       for (int index = 0; index < num_my_element; index++) {
           compute(p);
           p++;
       }
   pthread_mutex_lock(&mutex);
   }
   ```

3. After all the blocks have been computed (the current block indicator reaches the total block size), threads will exist and the root thread would be in charge of the later I/O operation.

3 Experiments

Comprehensive experiments are conducted to test and analyze the performance of parallel computing by MPI programming and multi-threaded programming. Various input data size are tested (range from size 100 to size 10000, here the size means the input resolution, thus the actually size of the data is $size^2$). Here, small

size (100 - 1000), medium size (2000 - 4000), and large size (8000 - 10000) are respectively defined. All the test result ran by sbatch can be found in corresponding *.out* files.

3.1 Evaluation Metrics

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

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

3.2 Results

3.2.1 Main results

The Speedup results (based on the equation above) of both MPI version and Pthread version (static scheduling and dynamic scheduling) are presented in Table 1, Table 4, and Table 3. The running time results for different configurations are also provided in Table 5, Table 6, and Table 7 (See appendix).

Table 1: *The Speedup results for MPI program from input size 100 to input size 10000.* ***Note:*** *Speedup for sequential sorting algorithm constantly equals to 1.*

#process	100	400	800	1000	2000	3000	4000	8000	10000
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.86173	1.31584	1.46217	1.33679	1.40081	1.40426	1.35341	1.24321	1.35211
4	1.06580	1.52361	2.15568	2.09878	2.07220	2.16598	2.06674	2.12751	2.08341
8	1.19226	2.04622	3.56999	3.69700	3.75856	3.73082	3.68465	3.76392	3.69078
12	1.45885	2.50432	4.86107	5.40206	5.30324	5.64533	4.94474	5.55640	5.41538
16	2.15977	3.13614	6.11364	7.14374	6.79727	7.40906	7.44517	7.01445	6.77126
24	1.77014	5.26080	8.35153	8.54299	9.31596	9.27534	10.05549	9.90719	9.93133
32	1.79637	4.92969	7.48081	9.43120	11.26465	11.40791	11.18561	12.00069	10.63252
40	3.41636	8.73945	11.71514	5.48084	10.04970	13.80999	12.37960	13.84516	11.59086

3.2.2 Image Display

The GUI outputs for MPI version and Pthread version (here, I choose dynamic scheduling as an example) when the input size is 800x800 are shown below. The parallel program uses 10 processes/threads as a demo example.

3.3 Performance Evaluation: Different Number of Processes for MPI

According to the results shown in Table 1, increases in speedup are both observed. To better understand the results , the visualization of the results for different number of threads is presented in the following figure:

Table 2: *The Speedup results for Pthread program (static scheduling) from input size 100 to input size 10000.* ***Note:*** *Speedup for sequential sorting algorithm constantly equals to 1.*

#thread	100	400	800	1000	2000	3000	4000	8000	10000
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.65266	1.25097	1.29534	1.38856	1.36060	1.43863	1.42894	1.40591	1.40920
4	0.91569	1.30419	1.59159	1.86359	2.00964	2.09597	2.09309	2.07176	2.05974
8	1.36308	1.64578	2.67821	2.90675	3.26900	3.54834	3.56062	3.56675	3.53224
12	1.59847	1.91650	3.23739	3.63401	4.53843	4.90915	4.90417	5.03203	5.04508
16	1.69356	2.41997	3.88329	4.50363	5.33508	6.24608	6.17319	6.48148	6.52733
24	1.38314	2.94094	4.51712	4.83646	6.89328	8.02020	7.94312	8.50292	8.52985
32	1.23334	3.44699	4.69620	5.64514	7.76052	9.65145	9.58202	10.32919	10.51069
40	0.87132	3.66526	5.44525	6.50434	9.28128	10.59036	10.98879	11.89432	11.76708

Table 3: *The Speedup results for Pthread program (dynamic scheduling) from input size 100 to input size 10000.* ***Note:*** *Speedup for sequential sorting algorithm constantly equals to 1.*

#thread	100	400	800	1000	2000	3000	4000	8000	10000
1	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
2	0.88340	1.33861	1.76778	1.77042	1.86390	1.90426	1.89081	1.89119	1.87221
4	1.40329	2.06734	2.68177	2.96847	3.44958	3.58128	3.53556	3.68498	3.59686
8	1.87058	3.18508	4.04610	4.87684	5.87725	6.27193	6.53489	6.66973	6.71690
12	2.02260	3.60567	5.75071	5.40471	7.58370	8.31907	8.82109	8.85122	9.12135
16	2.02043	3.96299	5.73234	6.84975	8.93663	10.26085	10.50832	10.67430	10.98466
24	1.51106	4.71271	6.85859	9.18161	10.12386	12.47464	12.30198	13.15328	13.40912
32	1.34744	4.98281	7.15253	9.12043	11.31614	14.10358	14.86902	15.17579	15.53189
40	1.16853	5.28961	9.56842	10.70991	12.53366	14.41287	15.78230	16.40013	16.87425

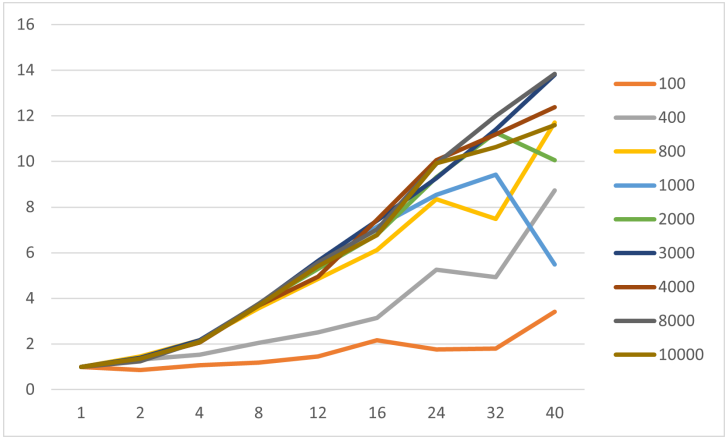

Figure 6: *The relationship between number of processes and speedup performance given different number of size*

We can have the following three observations:

1. As the number of cores increases, the overall performance of the parallel program increases (as long as

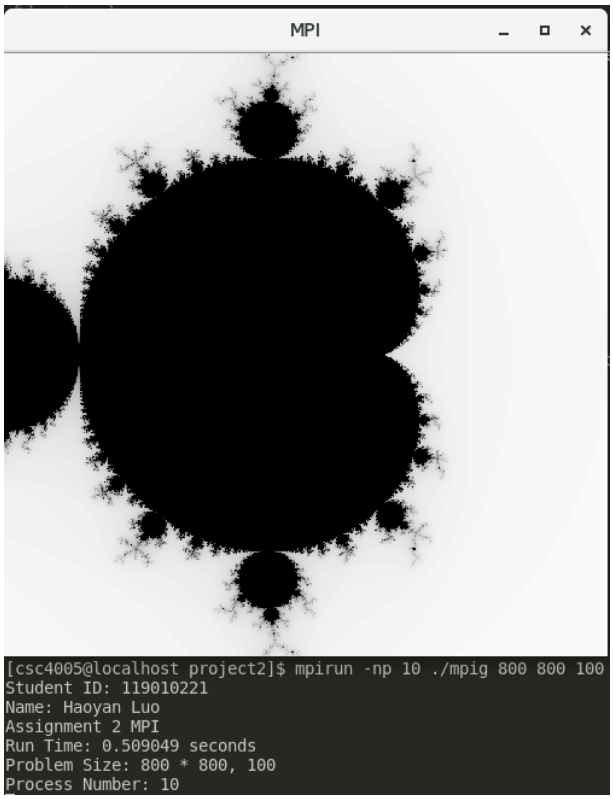

Figure 4: *GUI result for MPI program*

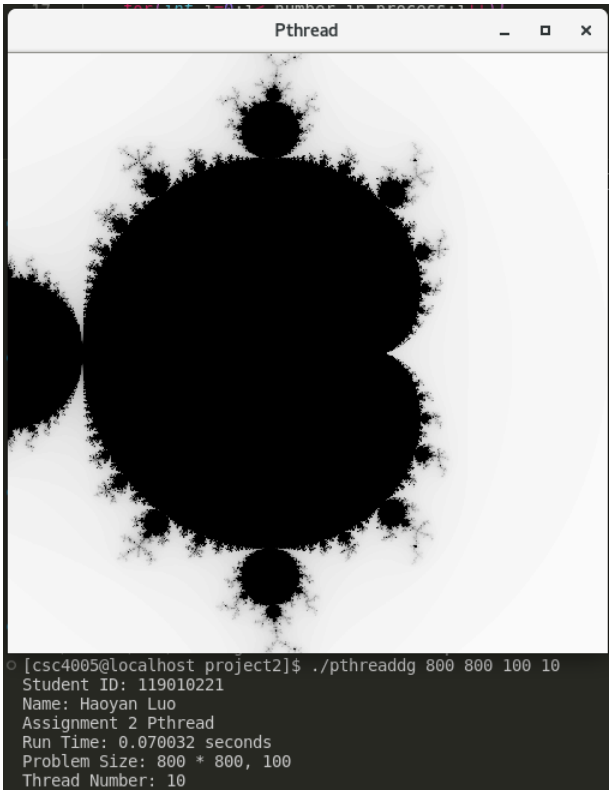

Figure 5: *GUI result for Pthread program (dynamic scheduling)*

it is not under a relatively small size).

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

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

Analysis The observations above enlighten that the optimal number of processes for a given configuration depends on many factors such as problem size. As the number of cores increase, workload are more evenly to be distributed to each core and perform the parallel computing, thus, each core's workload is decreased and total speed enhances.

Particularly, parallel computing can improve performance notably when the number of processes fits the problem size (i.e. the number of input elements can be fully divided by the number of processes). Moreover, it can be found that when the problem size is large (greater than 8000), the performance can reach its maximum and can even increase as the problem size gets larger.

3.4 Performance Evaluation: Different Number of Threads for Pthread

According to the results shown in Table 4 and Table 3, increases in speedup are both observed on two programs. Note that the dynamic scheduling version are generally better than the static version (analysis is conducted in the following session), pthread dynamic scheduling version is used here for analyzing the effect of different number of threads.

To better understand the results in a clearer way, the visualization of the results for different number of threads is presented in the following figure:

We can have the following three observations:

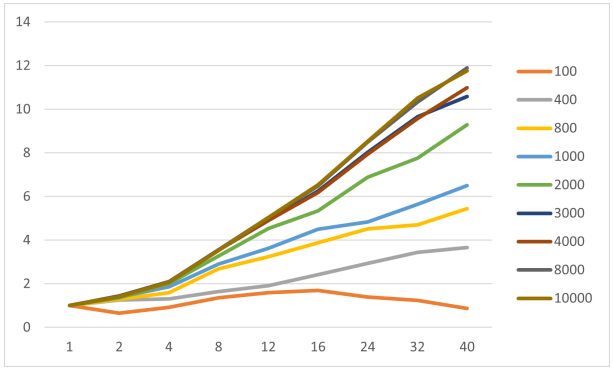

Figure 7: *GUI result for Pthread static scheduling program*

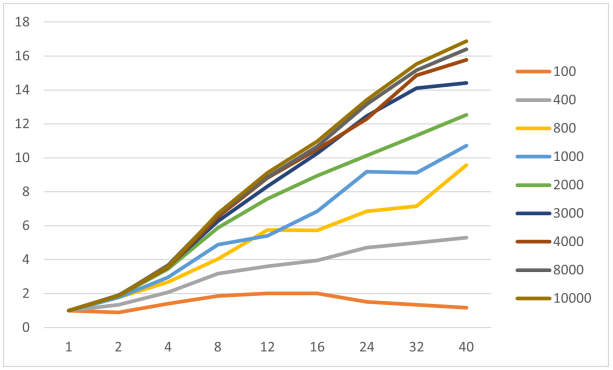

Figure 8: *GUI result for Pthread dynamic scheduling program*

1. As the number of threads increases, the performance (speedup) is almost linearly increasing when the input size is medium and large. For example, when the input size is 10000 and the number of threads allocated is 40, the speedup factor can be up to 12.

2. However, when the input size is small, the performance improvement is not that obvious and might even harm the performance (see size 100 as an example).

3. Another interesting thing is the increase of each line is not smooth (which means the performance improvement will sometimes stop for a while as the number of threads increases).

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

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

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

offset or exceed the performance gain given by parallel computing.

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

3.5 Performance Evaluation: Different Size of Output Images

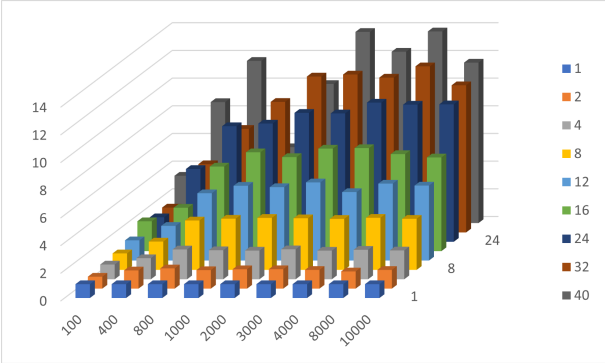

Figure 9: *Relationship between problem size and speedup for different number of processes*

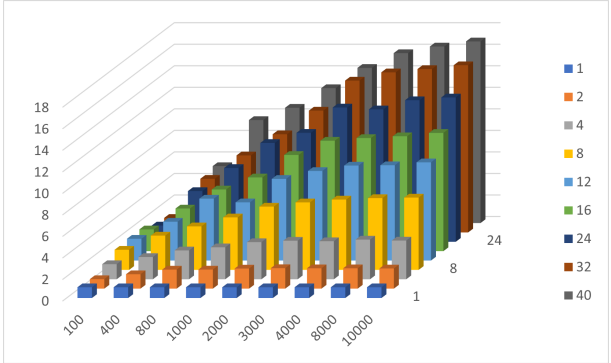

Figure 10: *Relationship between problem size and speedup for different number of threads (dynamic scheduling)*

From the figures above, we can observe the following:

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

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

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

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

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

where s is the number of cores available and n is the proportion of sequential computing. As the problem size increases, the sequential proportion n will decrease, which makes the speedup more close to the number

of cores. However, there also exists a reasonable boundary for this improvement as the problem size getting larger and larger. Thus, the performance improvement will slow down and cease when the problem size is too large (relative to the number of cores/threads).

3.6 Performance Evaluation: Sequential vs. Pthread vs. MPI

To compare sequential computing and parallel computing (MPI program and Pthread version), we can refer back to Figure 6 and Figure 8 or refer to the following figures: The observation is straightforward (and

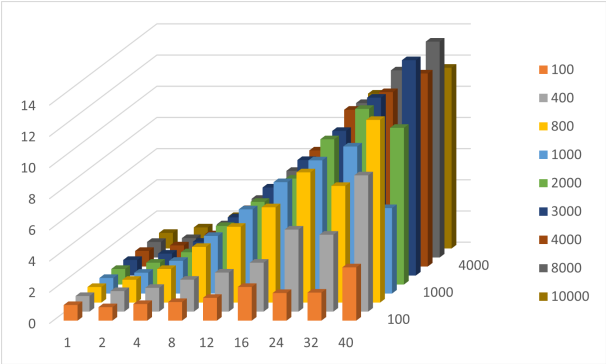

Figure 11: *Performance comparison for different number of processes, the left most column is sequential performance.*

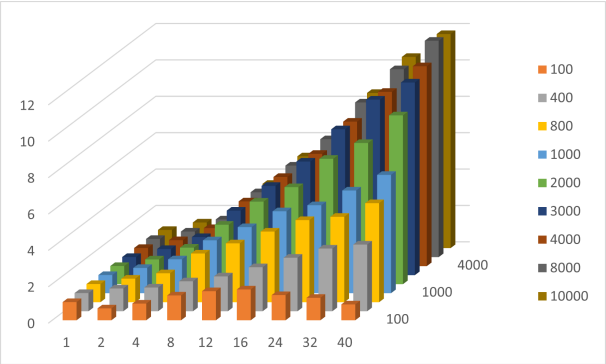

Figure 12: *Performance comparison for different number of threads (static version), the left most column is sequential performance.*

similar to the previous discussion) that when the problem size is small, parallel implementation's boost on performance is relatively small and may sometimes harm the performance (see size 100 as an example) due to the overhead of data allocation. When the problem size is medium or large, the performance gain becomes more obvious.

Table **??** is used to compare the performance between MPI program and Pthread program. Data in the table indicates running time of MPI / running time of Pthread. Thus, when the result is larger than 1, this indicates Pthread has more robust performance than MPI. Conversely, if the result is less than 1, this indicates MPI's performance is better. The results show that, in general, Pthread outperform MPI in regard

Table 4: *Comparison table between Pthread program (static scheduling) and MPI program from input size 100 to input size 10000.* ***Note:*** *Pthread is better if larger than 1, while MPI is better if smaller than 1.*

#thread/process	100	400	800	1000	2000	3000	4000	8000	10000
2	1.02515	1.01731	1.20901	1.32438	1.33059	1.35605	1.39707	1.52121	1.38465
4	1.31665	1.35687	1.24405	1.41438	1.66470	1.65343	1.71069	1.73206	1.72643
8	1.56894	1.55657	1.13336	1.31913	1.56370	1.68111	1.77355	1.77202	1.81991
12	1.38644	1.43978	1.18301	1.00049	1.43001	1.47362	1.78393	1.59298	1.68434
16	0.93548	1.26365	0.93763	0.95885	1.31474	1.38491	1.41143	1.52176	1.62225
24	0.85364	0.89582	0.82124	1.07475	1.08672	1.34493	1.22341	1.32765	1.35018
32	0.75009	1.01077	0.95612	0.96705	1.00457	1.23630	1.32930	1.26458	1.46079
40	0.34204	0.60526	0.81676	1.95407	1.24717	1.04366	1.27486	1.18454	1.45582

to Mandelbrot set computation. Particularly, when the problem size is medium or large, Pthread has a increasingly stronger performance. On the other hand, MPI is better when the problem size is small and the number of processes/threads allocated in relatively large.

Analysis Pthread that outperforms MPI in general can be explained by the communication overhead (as discussed above). Pthread uses shared memory while MPI uses $MPI_Bcast()$ and $MPI_Gatherv()$ (in my implementation I use non-blocking operation to improve the performance) to communicate, each thread's computation for Pthread is more independent with each other than parallel computation via MPI. Therefore, when problem size is larger, it have to allocate longer running time to transfer the data.
For the second phenomenon, one possible explanation is that when the number of threads are in large scale, they need to access the memory through the links that crossed different nodes, and thus results in a larger access time and computation time.

3.7 Performance Evaluation: Static Scheduling vs. Dynamic Scheduling

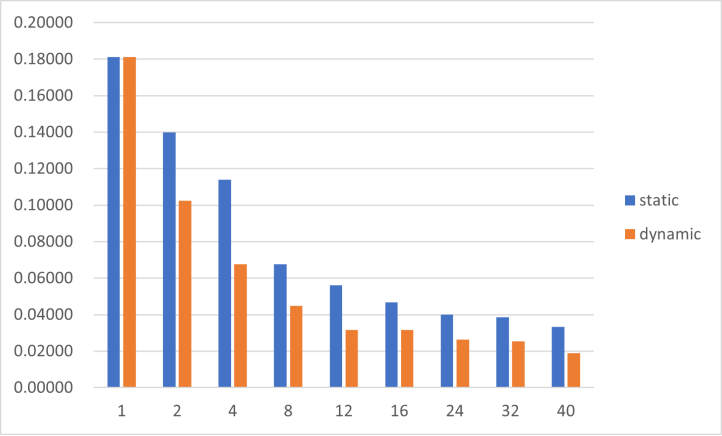

Figure 13: *Comparison between static scheduling and dynamic scheduling, given 800 as input size and different threads.*

From the figure above we observe that dynamic scheduling is indeed generally better than static scheduling. To explore and verify the reason behind, the following figure can also be referred to:

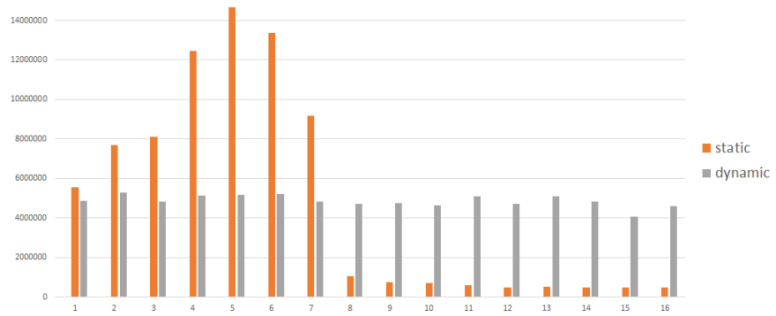

Figure 14: *Comparison of running time among different threads, given 800 as input size and 16 threads as an example.*

From the histogram, it is clear that the workload of each thread is highly imbalanced, while the workload of each thread for dynamic scheduling is balanced. This indicates that static scheduling would "waste" some of the resources (threads that have finished their work) while dynamic programming utilizes them in a more compact way.

When the workload is distributed very imbalanced, one thread may be assigned with a work that is heavier (harder to compute) than the average workload, which will result in a bottleneck thread. Thus, the total computing time will largely depend on the bottleneck thread. While in dynamic scheduling, the thread that finishes its work can continue to compete for other data for computation and no thread would be "waste", and thus the performance is better.

4 Conclusion

Parallel computation, generally, provides large performance gain corresponding to sequential computation. And in most cases, applying more threads or cores for the tasks (for example, in this Manelbrot set computation task) would increase the parallel computing performance and the increasing tendency is stable. In addition, different size of the input task has different suitable configurations (i.e. number of threads/cores). But in general, Pthread program has better performance (shown in my experiments). Meanwhile, to avoid the computing bottleneck in pthread, dynamic scheduling can be efficiently reduce calculation time and make use of the computation resources.

Appendix

A Steps to Run My Program

The codes of Pthread (both static scheduling version and dynamic scheduling version) and MPI version are in *pthread_static.cpp*, *pthread_dynamic.cpp* and *mpi.cpp*, respectively. Following are the steps to run my code (First please ensure that you are under the project folder). To avoid segmentation fault on cluster, please raise the upper limit of stack resource in advanced (since the test data is large). In my experiment, the following command works fine:

```
ulimit -s 10240000
```

1. MPI: To run the MPI program, first you need to compile it with GUI:

```
mpic++ mpi.cpp -o mpig -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL
↪  -lm -DGUI -std=c++11
```

or without GUI (if you test it on cluster):

```
mpic++ mpi.cpp -o mpi -std=c++11
```

Then run it by (here, X_RESN means the resolution of x axis, Y_RESN means the resolution of y axis, $max\_iteration$ is a parameter of Mandelbrot Set computation, and use 4 processes as an example) an iterative way:

```
salloc -n4 -p Debug
mpirun -np 4 ./mpi $X_RESN $Y_RESN $max_iteration
mpirun -np 4 ./mpig $X_RESN $Y_RESN $max_iteration
```

or in a batch style:

```
sbatch mpi_submission.sh
```

You can check the results in the corresponding output file.

2. Pthread(Static): For pthread static version, first you need to compile it with or without GUI (similar with above):

```
g++ pthread_static.cpp -o pthreads -lpthread -O2 -std=c++11
g++ pthread_static.cpp  -o pthreadsg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
↪  -lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11
```

Then you can either test the program in an iterative way: (here, size 1000x1000, 100 iterations and 20 threads are used as an example)

```
salloc -n1 -c20 -p Project
srun ./pthreads 1000 1000 100 20
```

or in a batch style:

```
sbatch pthreads_submission.sh
```

You can check the results in the corresponding output file.

3. Pthread(Dynamic): For pthread static version, first you need to compile it with or without GUI (similar with above):

```
g++ pthread_dynamic.cpp -o pthreadd -lpthread -O2 -std=c++11
g++ pthread_dynamic.cpp  -o pthreaddg -I/usr/include -L/usr/local/lib -L/usr/lib
↪  -lglut -lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11
```

Then you can either test the program in an iterative way: (here, size 1000x1000, 100 iterations and 20 threads are used as an example)

```
salloc -n1 -c20 -p Project
srun ./pthreadd 1000 1000 100 20
```

or in a batch style:

```
sbatch pthreadd_submission.sh
```

You can check the results in the corresponding output file.

B Running Time Results

Table 5: *The running time results for sequential version (process number equals to 1) and MPI version from input size 100 to input size 10000.* ***Note:*** *The unit is second in scientific notation.*

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

Table 6: *The running time results for sequential version (thread number equals to 1) and Pthread static scheduling version from input size 100 to input size 10000.* ***Note:*** *The unit is second in scientific notation.*

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

Table 7: *The running time results for sequential version (thread number equals to 1) and Pthread dynamic scheduling version from input size 100 to input size 10000.* ***Note:*** *The unit is second in scientific notation.*

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