



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

PARALLEL PROGRAMMING

Assignment2 Mandelbrot Set Computation

Author:
Shi Wenlan

Student Number:
119010265

October 17, 2022

Contents

1	Introduction	2
2	Method	2
2.1	Flow charts	2
2.2	Extra explanations	4
3	Result	5
3.1	Interpretation of Program Input Arguments	5
3.2	How to run my code	5
3.3	Output screenshots	7
3.4	Raw data	11
3.5	Data analysis	14
4	Conclusion	21

1 Introduction

In this assignment, I programmed three versions of Mandelbrot Set Computation: two MPI version (Send-Recv version and Scatter-Gather version) and a Pthread version. And I test their performances using different numbers of cores on different datasize.

If the program is compiled and ran in GUI version, it will show a figure as the output. In the figure, the points in black area are points within Mandelbrot Set, while the points in white area are points outside the Mandelbrot Set. The points are scaled to $[-1, 1] \times [-1, 1]$.

Introduction of Mandelbrot Set Computation: Mandelbrot Set is a set of points c in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function:

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

When Z_{k+1} is the $(k+1)_{th}$ iteration of the complex number $Z_0 = 0 + 0i$ and is a complex number giving the position of the point in the complex plane.

Assume $Z_k = a + bi$, then

$$Z_{k+1} = (a^2 - b^2 + c_{real}) + (2ab + c_{imaginary})i$$

2 Method

2.1 Flow charts

The general idea flow charts of my three versions of program are as following:

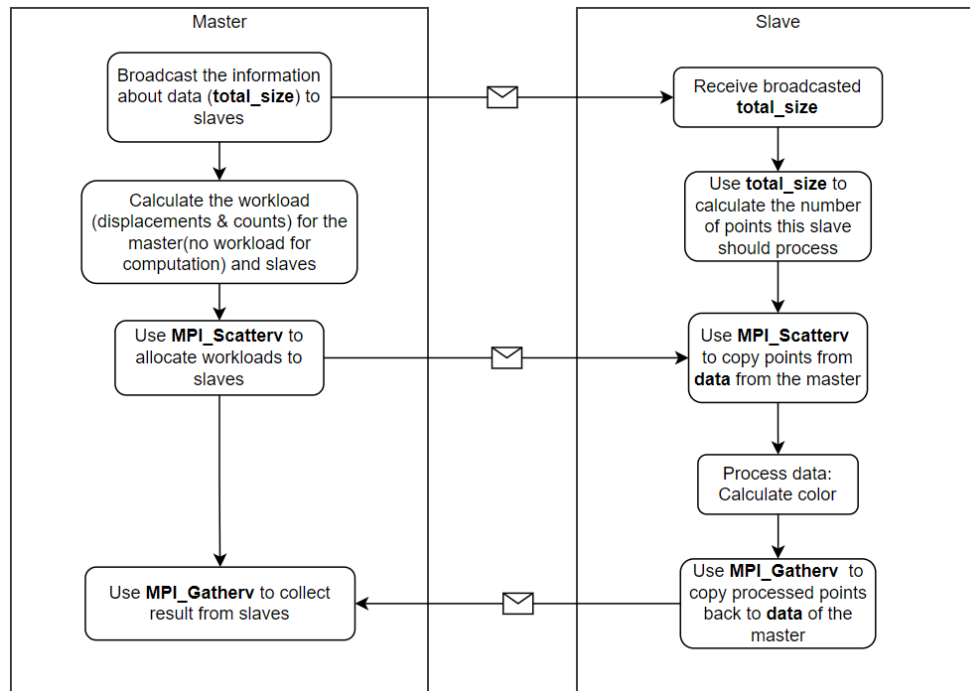


Figure 1: General Flow Chart of MPI Scatter-Gather Version

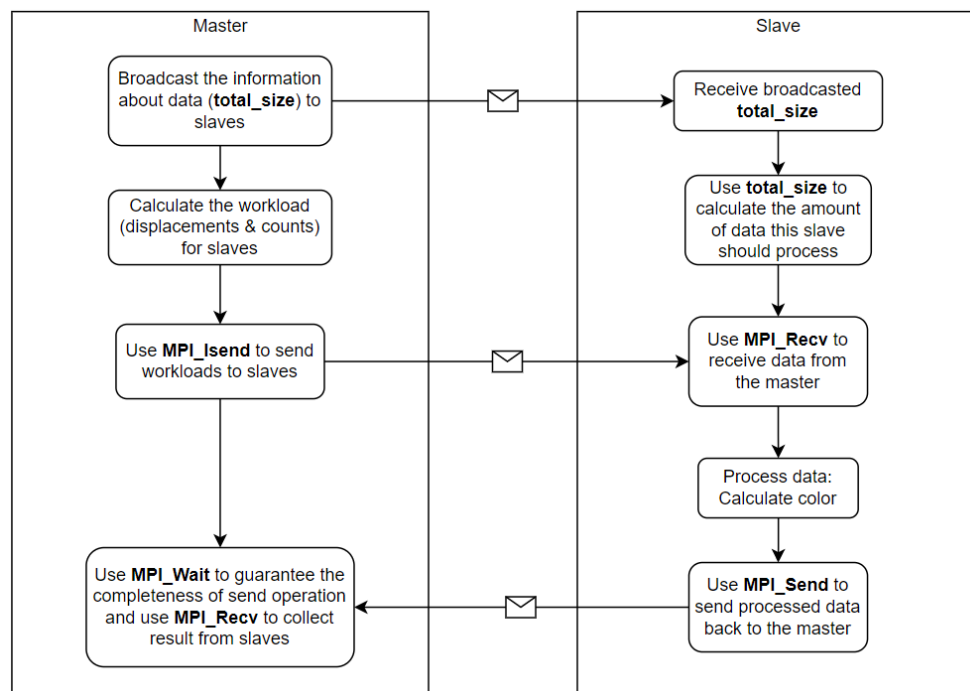


Figure 2: General Flow Chart of MPI Send-Recv Version

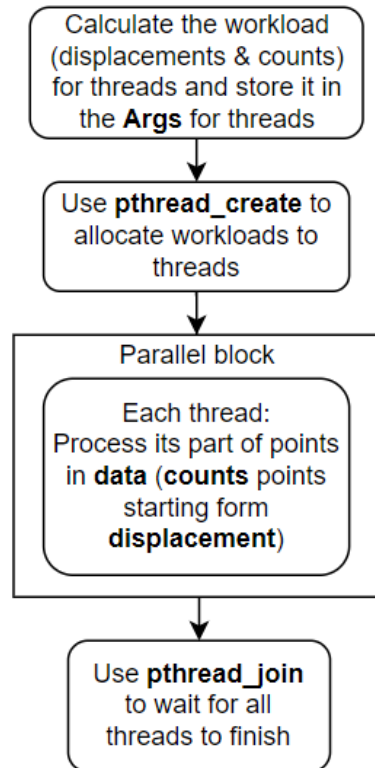


Figure 3: General Flow Chart of Pthread Version

2.2 Extra explanations

The following are some extra explanations:

Why there are two MPI versions: When I was coding, I met strange bugs. In the beginning, I tried to circumvent the bug by using another implementation. But this was just a failed attempt, actually the bugs were caused by other reasons. Anyway, the code has already been written and debugged, and it is too wasteful not to use it as material for experimental analysis.

Why to use *MPI_Bcast* in MPI versions: Processors do not have shared memory. Therefore, given the fact that only the master processor initiates the data, none of slave processors knows any information about data. So, if slave processors want to compute it own workload, it must get *total_size* of data from the master processor. In other words, the master processor should broadcast *total_size* to all slave

processors. Pthread version does not need to worry about this, since threads are created after data initiation. All threads share memory of data.

Why to use dynamic memory management in MPI versions: Because the amount of data that each process needs to receive and process can be so large that it exceeds the capacity limit of statically allocated memory.

3 Result

3.1 Interpretation of Program Input Arguments

$X_RESN \times Y_RESN$ is the image size, or problem size, and can be regarded as sample size. The larger they are, the higher the resolution of the output figure can reach.

$max_iteration$ is the max number of computation resolution, the larger it is, the more binary the output figure will be.

n_proc is the number of cores the MPI program will use.

n_thd is the number of threads the Pthread program will use.

3.2 How to run my code

If you want to reproduce the experiment in Subsection *Output screenshots*: Open source code folder in the terminal, guarantee that there are enough processors available, then type in the following instruction to run the script.

```
1 /bin/bash compile_run_gui.sh
```

If you want to reproduce the experiment in Subsection *Raw data*: Open source code folder in the terminal, guarantee that there are enough processors available, then type in the following instruction to run the script.

```
1 /bin/bash compile_run.sh
```

If you want to run the code more flexibly: Open source code folder in the terminal, guarantee that there are enough processors available, then select useful instructions from following manual.

```
Sequential without GUI

g++ sequential.cpp -o seq -O2 -std=c++11

Sequential with GUI

g++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm sequential.cpp -o seqg -DGUI -O2 -std=c++11

MPI without GUI

mpic++ mpi_ScatterGather.cpp -o mpi_ScatterGather -std=c++11
mpic++ mpi_SendRecv.cpp -o mpi_SendRecv -std=c++11

MPI with GUI

mpic++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm mpi_ScatterGather.cpp -o mpi_ScatterGatherg -DGUI -std=c++11

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

Figure 4: Instructions to Compile (Part 1)

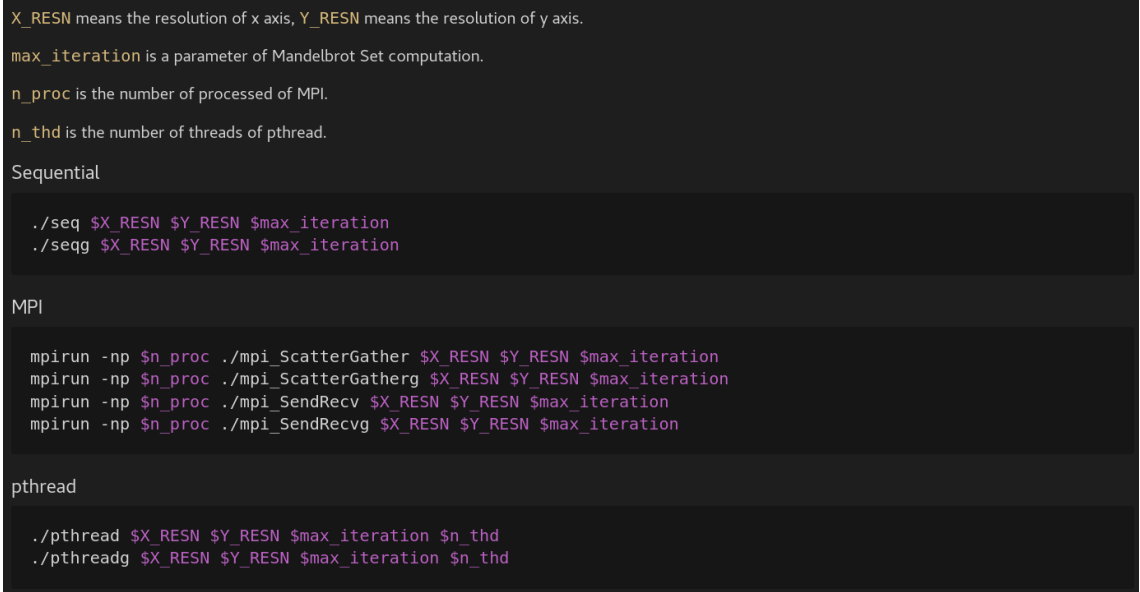
```
pthread without GUI

g++ pthread.cpp -lpthread -o pthread -O2 -std=c++11

pthread with GUI

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

Figure 5: Instructions to Compile (Part 2)



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.

n_proc is the number of processed of MPI.

n_thd is the number of threads of pthread.

Sequential

```
./seq $X_RESN $Y_RESN $max_iteration
./seqg $X_RESN $Y_RESN $max_iteration
```

MPI

```
mpirun -np $n_proc ./mpi_ScatterGather $X_RESN $Y_RESN $max_iteration
mpirun -np $n_proc ./mpi_ScatterGatherg $X_RESN $Y_RESN $max_iteration
mpirun -np $n_proc ./mpi_SendRecv $X_RESN $Y_RESN $max_iteration
mpirun -np $n_proc ./mpi_SendRecvg $X_RESN $Y_RESN $max_iteration
```

pthread

```
./pthread $X_RESN $Y_RESN $max_iteration $n_thd
./pthreadg $X_RESN $Y_RESN $max_iteration $n_thd
```

Figure 6: Instructions to run

3.3 Output screenshots

Here are screenshots of GUI version compile and run:

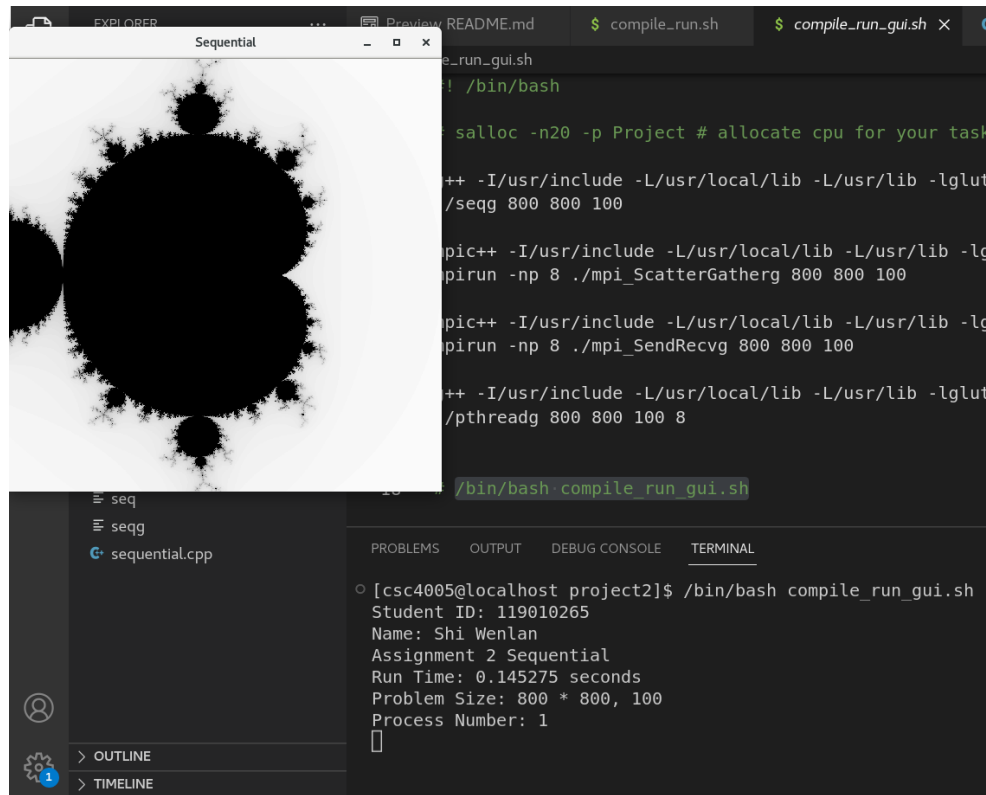


Figure 7: Screenshot of Sequential Version

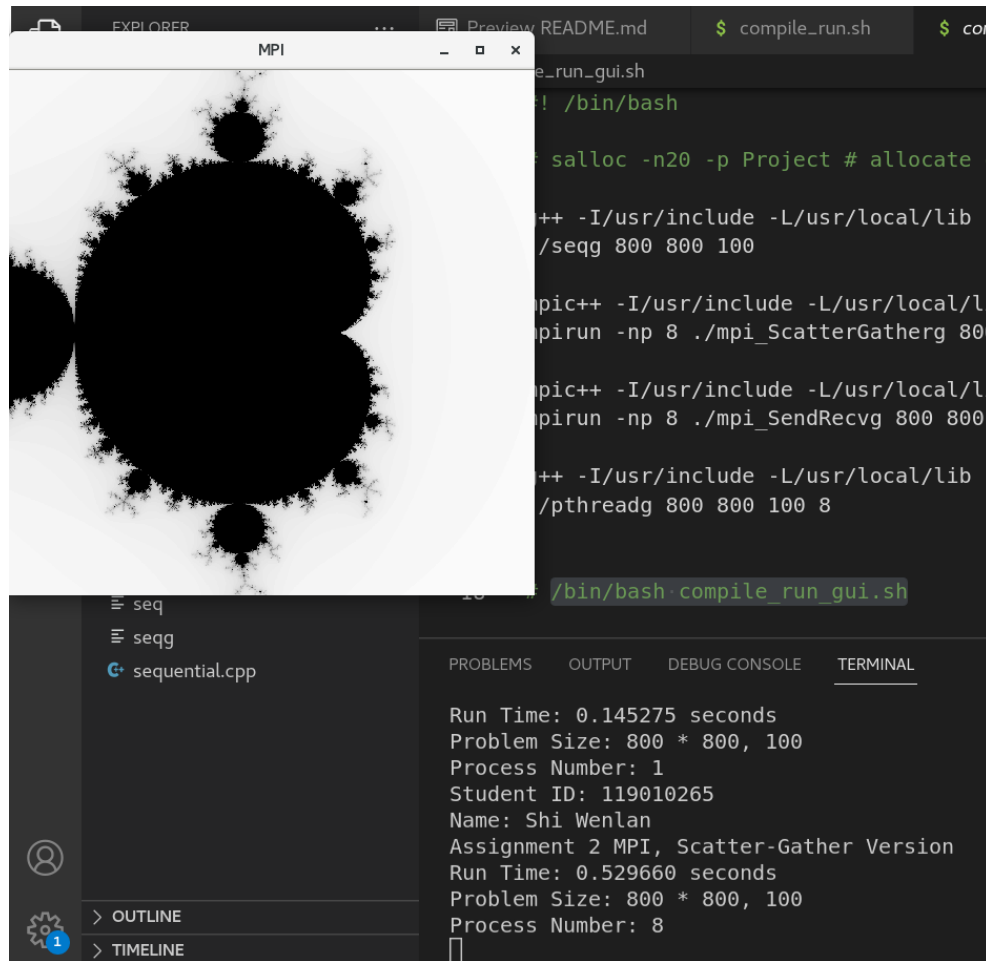


Figure 8: Screenshot of MPI Scatter-Gather Version

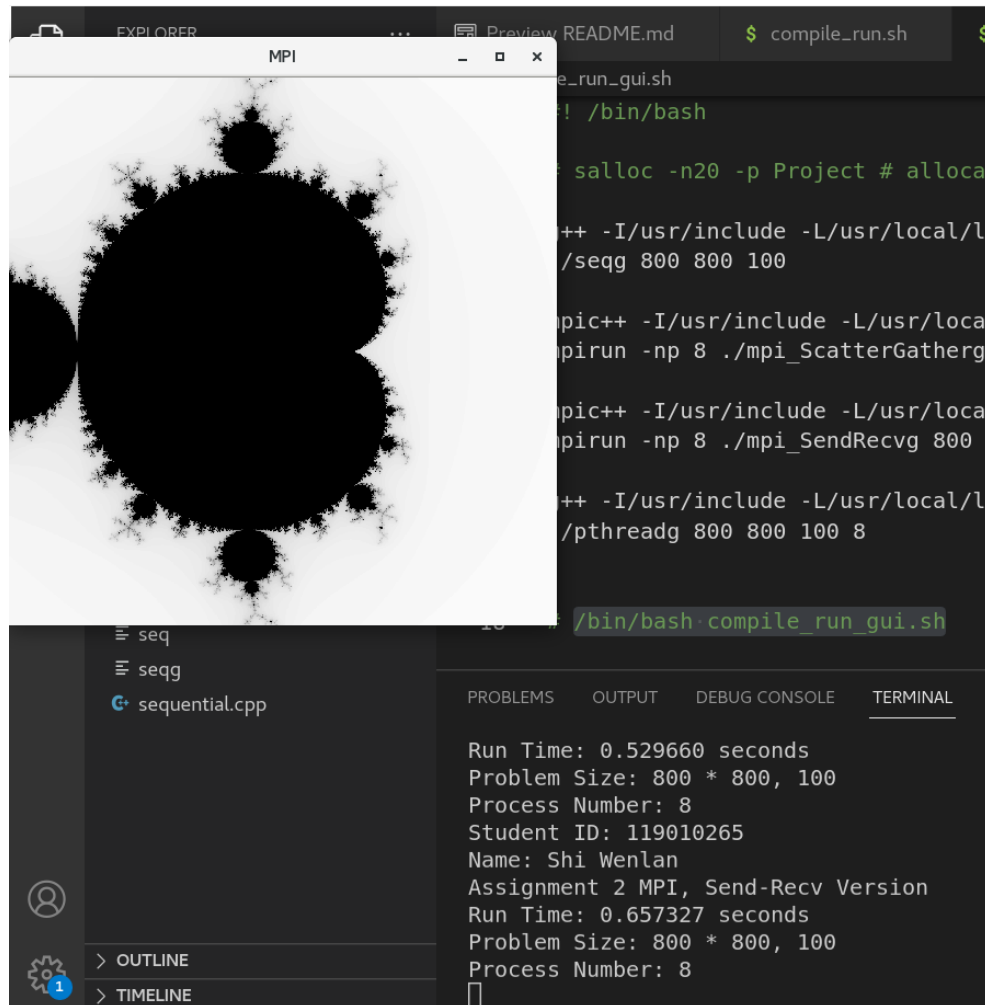


Figure 9: Screenshot of MPI Send-Recv Version

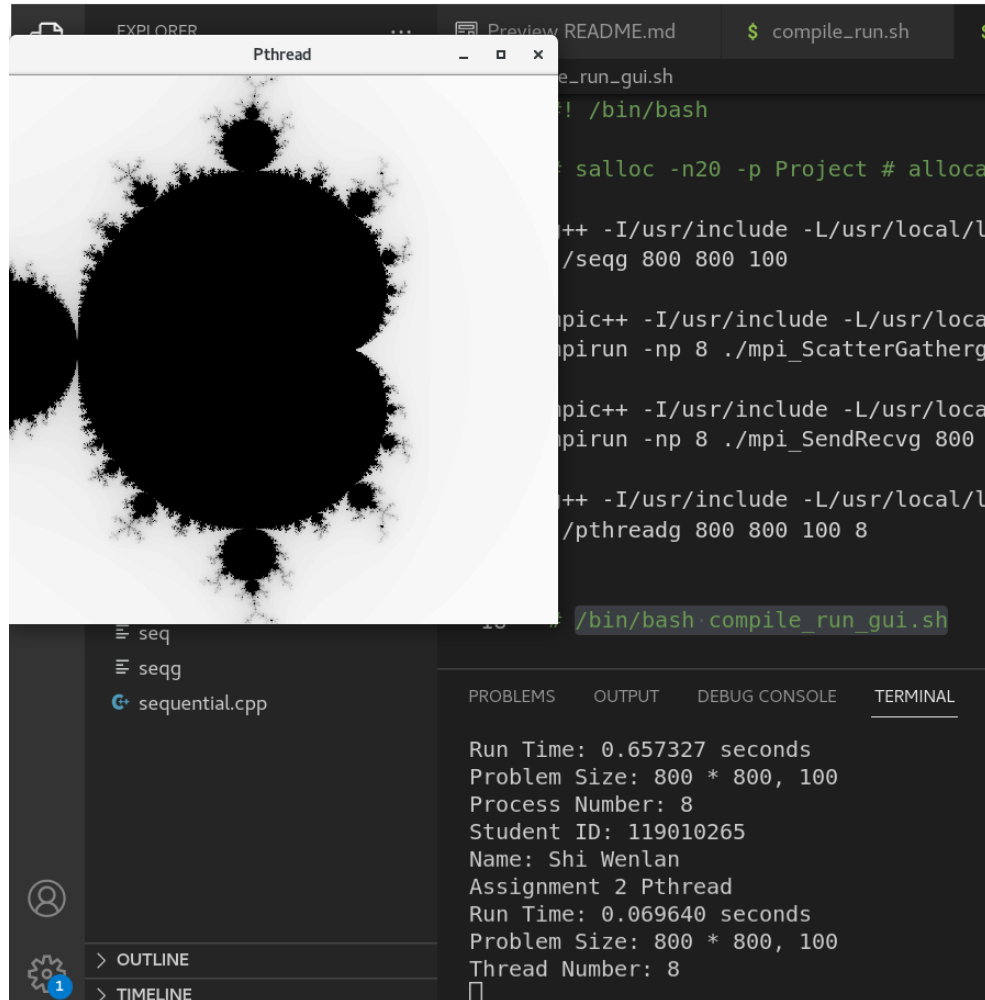


Figure 10: Screenshot of Pthread Version

3.4 Raw data

data size	1000 × 1000	5000 × 5000	10000 × 10000
Execution time (seconds)	0.309899	7.439914	29.789450

Table 1: Sequential Execution time (seconds)

Processors \ data size	1000×1000	5000×5000	10000×10000
2	0.313879	7.772260	30.931939
3	0.217966	5.244493	20.909448
4	0.185267	4.445501	17.572151
5	0.149337	3.493681	14.123003
6	0.127379	2.918179	11.551285
7	0.115223	2.510780	10.189563
8	0.092523	2.123018	8.318544
9	0.081413	1.982463	7.943061
10	0.076197	1.707549	6.747237
11	0.072701	1.716176	6.760453
12	0.071561	1.533359	6.001326
13	0.103188	1.483827	5.800043
14	0.057903	1.389510	5.384379
15	0.080582	1.230490	4.915316
16	0.064805	1.210909	4.856796
17	0.047823	1.125110	4.480552
18	0.084888	1.136396	4.420563
19	0.088450	1.067087	4.145985
20	0.103708	1.073494	4.159689

Table 2: MPI Scatter-Gather Execution time (seconds)

Processors \ data size	1000×1000	5000×5000	10000×10000
2	0.313430	7.783181	30.934894
3	0.219264	5.256856	21.416628
4	0.181712	4.529499	17.798973
5	0.143436	3.634487	14.277937
6	0.123819	2.851712	11.656381
7	0.114138	2.635857	10.523852
8	0.096214	2.178623	8.608862
9	0.085538	2.037043	8.180792
10	0.077309	1.768990	7.062807
11	0.080311	1.785742	6.968779
12	0.072571	1.606612	6.300949
13	0.074400	1.476302	6.084333
14	0.059939	1.420853	5.586150
15	0.076278	1.319661	5.225304
16	0.086542	1.311409	5.109994
17	0.090021	1.201514	4.646738
18	0.069726	1.188342	4.715726
19	0.088649	1.093866	4.429327
20	0.089598	1.073389	4.339421

Table 3: MPI Send-Recv Execution time (seconds)

Processors \ data size	100×100	1000×1000	5000×5000	10000×10000
1	0.008760	0.304023	7.464786	29.789857
2	0.005505	0.223779	5.124986	20.375044
3	0.005215	0.193408	4.311012	17.237118
4	0.004218	0.139509	3.416883	13.620355
5	0.003651	0.126334	2.777030	11.061246
6	0.003293	0.105780	2.477177	9.754155
7	0.004909	0.085245	2.039272	8.061867
8	0.003957	0.080715	1.932025	7.730023
9	0.003699	0.067912	1.636138	6.527865
10	0.002456	0.065346	1.586914	6.398034
11	0.002217	0.061564	1.423798	5.690970
12	0.002180	0.066408	1.405441	5.475287
13	0.002086	0.058198	1.314687	5.087019
14	0.002008	0.051768	1.209911	4.724400
15	0.001964	0.056483	1.177414	4.531056
16	0.003036	0.050842	1.058185	4.171055
17	0.001925	0.050555	1.052542	4.114684
18	0.001734	0.041439	0.951863	3.776670
19	0.004217	0.045793	0.931850	3.695633
20	0.001629	0.041615	0.909574	3.503542

Table 4: Pthread Execution time (seconds)

3.5 Data analysis

The following figure compares the execution time of sequential program and different implementations of parallel programs on the same dataset and with only one computational core/thread. The horizontal axis is the datasize and the vertical axis is the execution time.

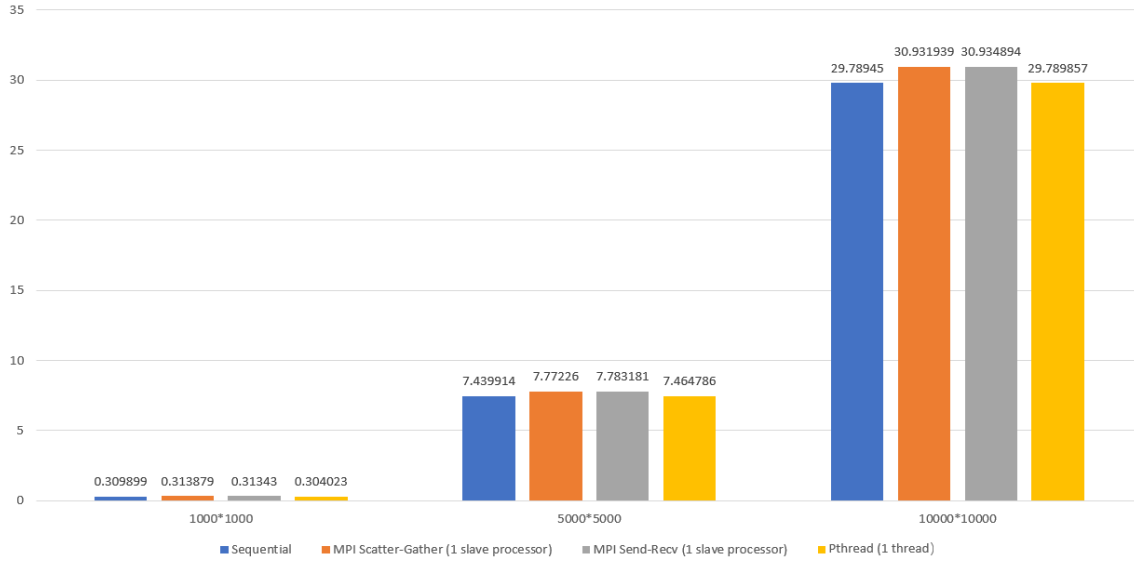


Figure 11: Sequential vs. MPI Scatter-Gather vs. MPI Send-Recv vs. Pthread

Sequential vs. MPI Scatter-Gather vs. MPI Send-Recv vs. Pthread: It can be seen that when all programs use only one computational processor/thread, their execution time from low to high is: sequential, Pthread, MPI Scatter-Gather, MPI Send-Recv. And the gap between sequential version and Pthread versions keeps small. But the gap between sequential version and MPI versions increases as the data size increases. This is because the sequential version manipulates directly on original data. While Pthread version has to create threads to operate on original data independently. The creation of threads leads to some extra fixed costs. MPI versions have to copy data from the master processor to the slave processor, and then copy result from the slave processor to the master processor after computation. The transmission and replication of data caused more overhead, and this overhead was proportional to data size. Besides, MPI Send-Recv version distributes tasks and gathers results sequentially, thus it would take a bit more time than MPI Scatter-Gather version, which does that in parallel.

The following figures show the execution time under different configurations (data size and number of computational cores/threads). The horizontal axis is the number of computational cores/threads and the vertical axis is the execution time. Notes that the number of computational cores equals to number of cores the MPI program used minus 1, because MPI versions use master-slave model, and the master processor is only responsible for distributing tasks and gathering results of tasks, not calculating.

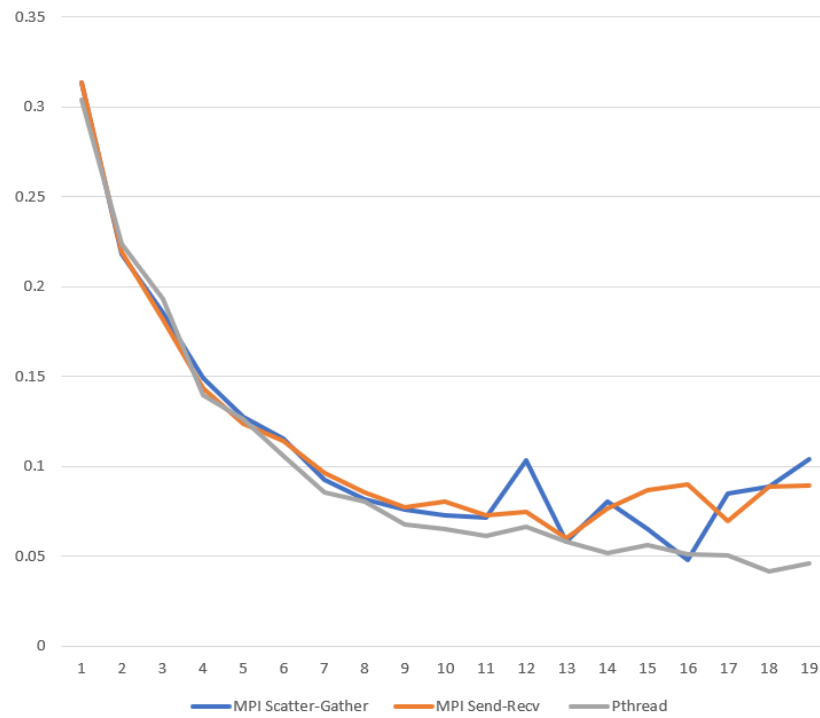


Figure 12: Execution time vs. Number of Processors/threads on small datasize 1000×1000

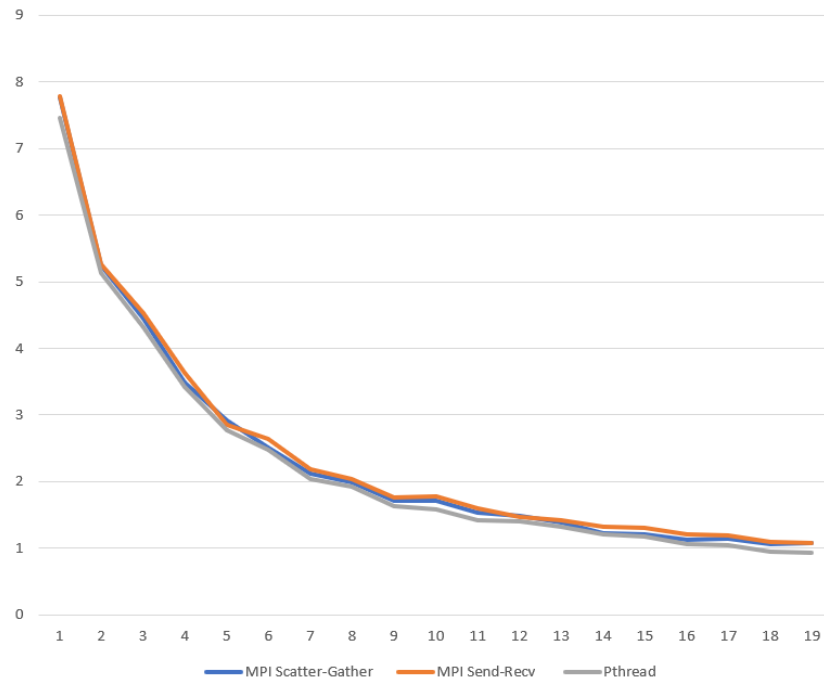


Figure 13: Execution time vs. Number of Processors/threads on middle datasize 5000×5000

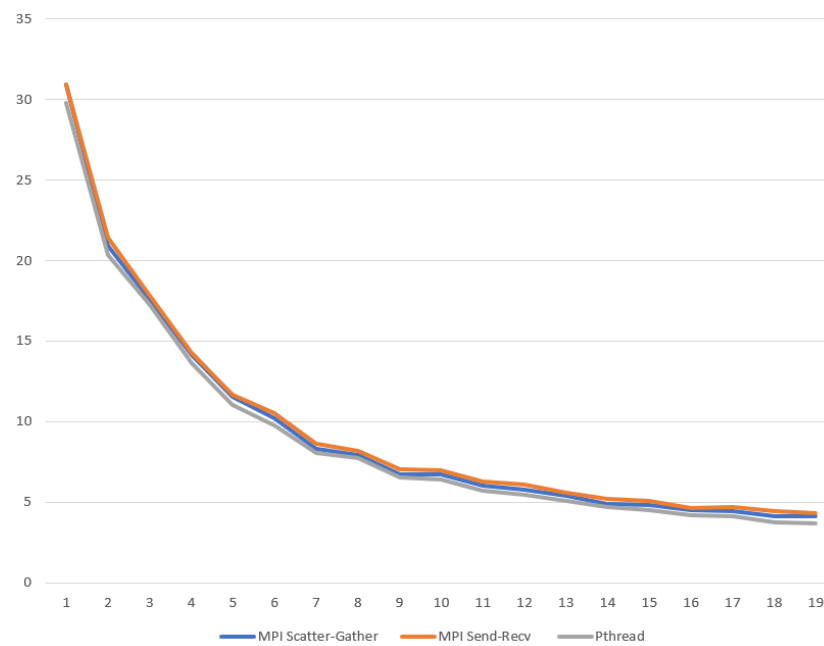


Figure 14: Execution time vs. Number of Processors/threads on large datasize 10000×10000

The following figure shows the speedup (sequential execution time (parallel version using 1 processor) / parallel execution time) under different configurations (datasize and number of computational cores/threads). The horizontal axis is the number of computational cores/threads and the vertical axis is the speedup.



Figure 15: Speedup vs. Number of Processors/threads on small datasize 1000×1000

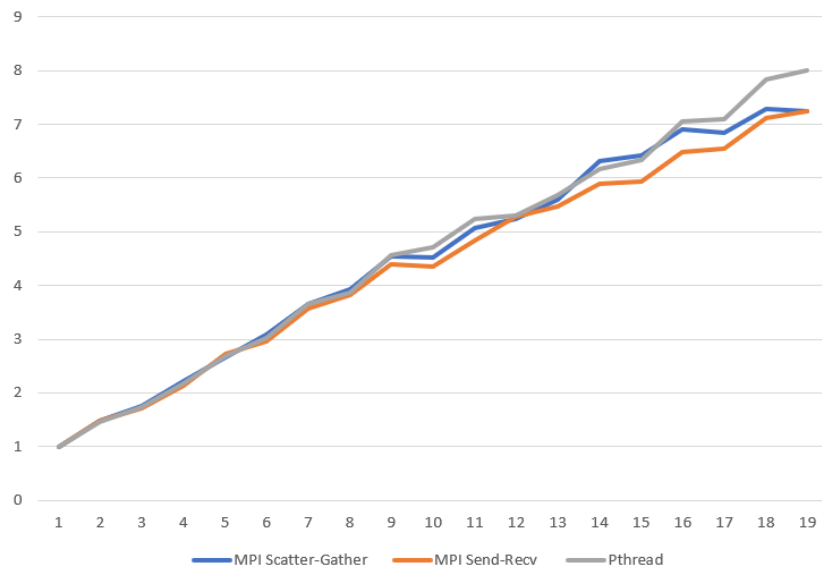


Figure 16: Speedup vs. Number of Processors/threads on middle datasize 5000×5000

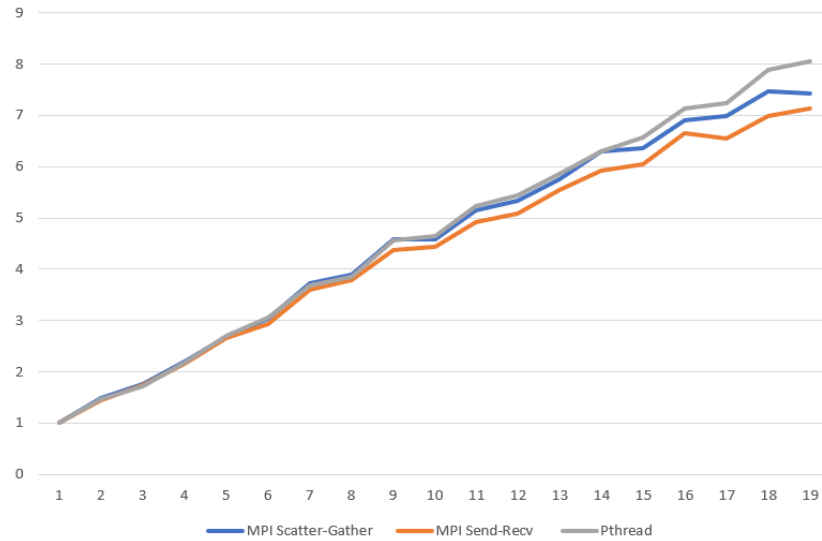


Figure 17: Speedup vs. Number of Processors/threads on large datasize 10000×10000

The number of cores/threads vs. Execution time: It can be seen that when datasize is large enough, execution time decreases as the number of computational cores/threads being used increases, and this effect is apparent when there are only a few cores/threads being used. It shows the diminishing marginal utility. This is because when the number of cores/threads being used increases, the data scale each processor/thread processes decreases, and the extent of reduction also decreases. In MPI cases, the total overhead of message communication between processors increases, which further reduces the marginal benefit and even makes it negative. In Pthread case, the total overhead of threads creation increases, which would also further reduce the marginal benefit.

Datasize vs. Execution time It can be seen that when the datasize is large, the reduction in execution time is significant, and the speedup increases steadily and linearly when the number of cores/threads increases. However when the datasize is small, the reduction in MPI versions' execution time is trivial, and the speedup is also unstable and finally decreasing when the number of cores increases.

This is because in MPI versions' situation, the overhead of message communication is related to datasize and the number of processors. The former part is flexible, and the latter part is fixed. Therefore, when the datasize is small, the benefit can hardly cover the overhead if the number of processors is large, while when the datasize is large, the net benefit is more apparent.

In contrast, in Pthread version's situation, the overhead of thread creation is only

related to the number of threads, which is fixed and quite lightweight. And in above experiment, the smallest data size is still not small enough to show the case when its overhead cover its benefit. In order to observe this case, I added an extra set of experiment on tiny datasize (100×100). And the result is as following:

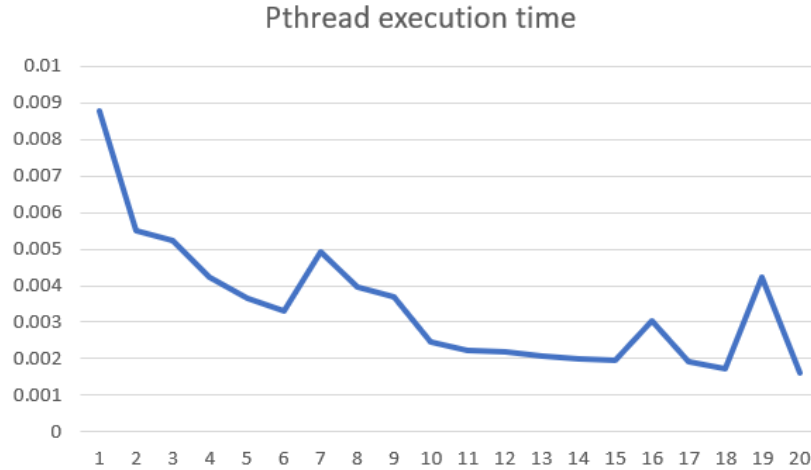


Figure 18: Execution time vs. Number of Threads on tiny datasize 100×100

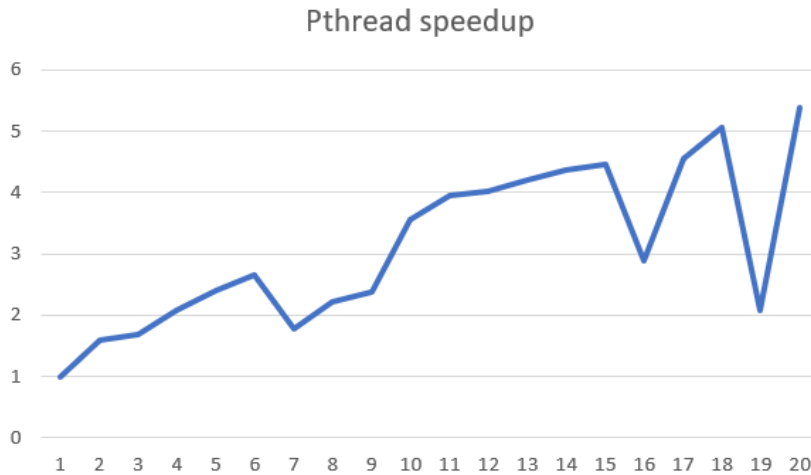


Figure 19: Speedup vs. Number of Threads on tiny datasize 100×100

Unusual surge in execution time An unusual surge in execution time is observable. It can be caused by two possible reasons:

1. When the time interval is small, it may be due to the measurement error caused by

the running environment. For example, the machine may be running other high-load tasks at the same time.

2. It is because the the cores are allocated in different nodes, the overhead of message passing increases a lot due to this and even cover the benefit.

The following figure shows the Cost-efficiency (speedup / number of cores/threads being used) under different number of cores/threads. The horizontal axis is the number of cores and the vertical axis is the Cost-efficiency.

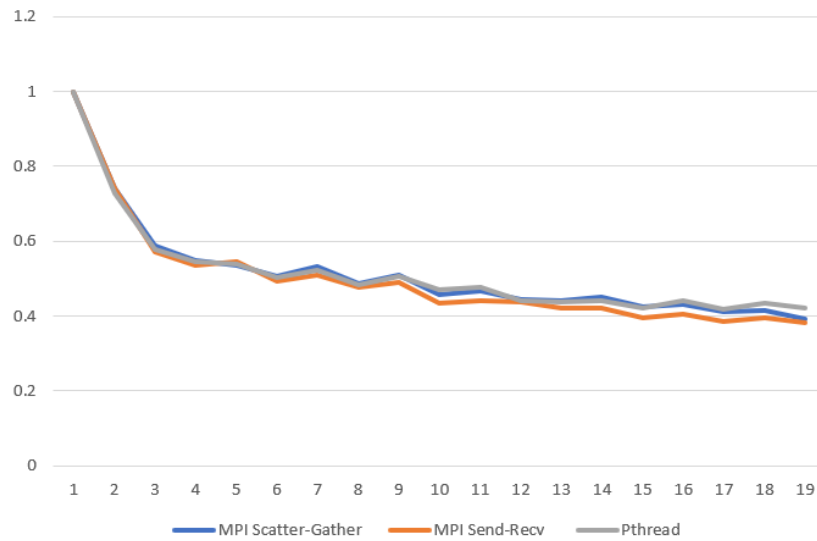


Figure 20: Cost-efficiency vs. Number of Processors/threads on large datasize 10000×10000

The number of cores vs. Cost-efficiency It can be seen that when datasize is large, the Cost-efficiency keeps decreasing as the number of cores/threads being used increases, and gradually converges to about 0.4.

4 Conclusion

My experiment results show that parallel can greatly speedup the computation task when datasize is large, and the number of cores/threads being used are not so large given the existance of diminishing marginal utility. In this environment, Pthread version shows the best performance. The datasize threshold to show the effect of this version is the lowest. And the overhead of this version is also the smallest. In other words, Pthread version can play a better role in more occasions.