INF3201 - Parallel Programming
Assignment 1 - MPI

Narae Park and Jonas Fladset Hoel

September 17, 2020

1 Introduction

The Mandelbrot set is a set of complex numbers using the quadratic polynomial $z(0) = z, z(n+1) = z(n) * z(n) + z, n = 0, 1, 2, ...$ where z is a constant number [1]. Because the Mandelbrot set calculated by iterating a function with varying input parameters, there is potential to speed up the calculation by calculating multiple points in the Mandelbrot set simultaneously across multiple processors. Using multiple hosts with access to more processing cores than a single host alone (parallelizing) would enable further time savings, given that the overhead of coordinating the communication between these hosts is not greater than the gain provided by more processors.

This paper describes the design of a program that uses an implementation of the MPI (Message Passing Interface) spesification to parallelize the generation of a Mandelbrot set [3]. It also presents experimental data produced by their implementation, and compares the time consumed to generate identical Mandelbrot sets by the baseline sequential implementation and various parallel implementations. The goal is to identify an approach that has the greatest time efficiency compared to the sequential implementation, if there is an improvement.

2 Sequential Solution Analysis

In order to provide a baseline for any improvements by parallelizing the Mandelbrot set, it is first calculated using a sequential implementation which only uses a single processor.

2.1 Sequential profiling and speedup

Profiling the sequential version (see figure 1) shows that the execution of the solve() function on each pixel takes the most time(97.8%). Thus, processing the execution of the solve() in parallel would shorten the performance time and increase efficiency.

2.2 Sequential - Speedup

To evaluate its performance in parallel programming, we can use speed-up and efficiency as a measure. The definitions of speed-up and efficiency are as follows (See figure 2).[4]

By parallelizing the serial solutions and distributing works, overall performance improvement can be achieved. Ideally, the speed should be increased in proportion to the number of processes. However, due to communication overhead, the proportional speed improvement cannot be obtained in reality, which can be

```
[npa013@uvcluster code]$ gprof ./RoadMapGProf
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 32.33     28.04    28.04 2810034755     0.00     0.00  complex_magn2
 27.99     52.31    24.27 44000000     0.00     0.00  solve
 23.59     72.77    20.46 2791580833     0.00     0.00  complex_add
 13.83     84.76    11.99 2791580833     0.00     0.00  complex_squared
  0.59     85.27     0.51 44000000     0.00     0.00  translate_y
  0.56     85.75     0.48       11     0.04     7.82  CreateMap
  0.46     86.15     0.40                             frame_dummy
  0.31     86.42     0.27 44000000     0.00     0.00  translate_x
  0.03     86.45     0.03       11     0.00     0.00  dump_data
  0.00     86.45     0.00        2     0.00     0.00  get_usecs
  0.00     86.45     0.00        1     0.00    86.05  RoadMap
```

Figure 1: Profiling sequential solution

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \qquad E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Figure 2: Definition of Speedup and Efficiency

seen by efficiency.

2.3 Assumptions & Expectations

Each pixel in the Mandelbrot set is completely independent of the results of other pixels, so it is a good example for parallel work. Since the amount of calculation is different for each pixel and the distribution is spatially partial, it is important to distribute the work in consideration of load balance. Ideally, the speed should be increased by the number of processes, but it is thought that there is a limit to the performance increase due to communication overhead. Through the above analysis, the assumptions and projections we set up are as follows.

2.3.1 Assumptions

1. There will be performance improvements if work is carried out in parallel.
2. Each pixel will have a different workload.
3. There will be communication overhead.

2.3.2 Expectations

1. Allocating multiple of processes will improve overall work speed / performance.
2. Equal distribution of the workload will improve speed/performance.
3. If too many hosts are used, the performance will be slightly degraded due to communication overhead-/cost.

3 Design

The primary focus of this section will be detailing the design of the two main design variants of the parallel implementation: the static variant and the dynamic variant. The static variant also has two different methods of allocating work, which will be explained as well.

3.1 Static scheme

The static variant of the implementation has each process working on a pre-determined section of the Mandelbrot set, scaling with how many processors are available to the program. Each process knows what section of the Mandelbrot set to calculate and does not need to communicate with the other processes until it is done calculating and sends the data to the master process. If it is itself the master process, it will calculate its section of the Mandelbrot set and then receive data from the other processes linearly (from process 1, 2, 3... n). Pseudocode can be seen in listing 1.

```
int main (){
    Initialize MPI and variables
    Retrieve own rank number and number of other ranks
    Based on rank number, determine boundries to calculate
    if (rank == 0){
        Calculate Mandelbrot set based on boundries from before
        for (i = 1; i < ranks; i++){
            Retrieve Mandelbrot data from all other ranks with MPI_Recv()
            Add recieved data to the local array of Mandelbrot data
        }
        Write mandelbrot data to disk
    }
    if (rank != 0){
        Calculate Mandelbrot set based on boundries from before
        Send calculated data to master rank with MPI_Send()
    }
    Finalize MPI
}
```

Listing 1: Static implementation

There are two methods to how the program determines which rank calculates what section of the set - linearly (rank #0 gets the first row, rank #1 gets the second one, etc.) and round-robin style. Both of these variants will now be explained in detail.

3.1.1 Row block partitioning (linear)[4]

Naive: Row Based Partition Scheme[2]

The first and simplest form of deciding how to partition up the set for the available processors is by simply dividing the height of the plot by the number of processors. This results in equally large rows for each process, although the amount of processing power required to finish each row varies. See figure 3 for a visual representation of this divison of labour.

When the work is not evenly divided by the number of processes, we let the last process do the rest.

Advantages: The number of communications is minimal. Therefore, the communication cost is low.
Disadvantage: The workload per process is not evenly distributed.

3.1.2 Row cyclic partitioning (round-robin)[4]

Alternating : Row Based Partition Scheme [2]
Another approach to partitioning up the Mandelbrot set between all the processors is with row cycle partitioning, or round-robin style. With this method, each rank calculates alternating rows of pixels, so as to

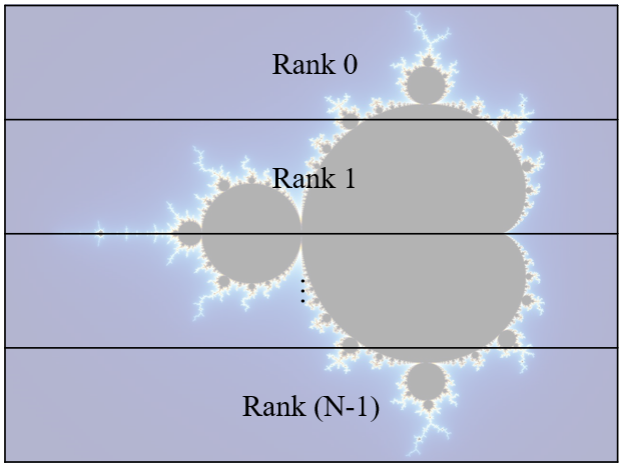

Figure 3: Static Row block partition

more evenly distribute the workload. Figure 4 illustrates this roughly, although in practice the row split is in singular pixel rows rather than row boxes that occupy large amounts of pixel rows as shown in the figure, but it helps to visualize the idea of round-robin partitioning.

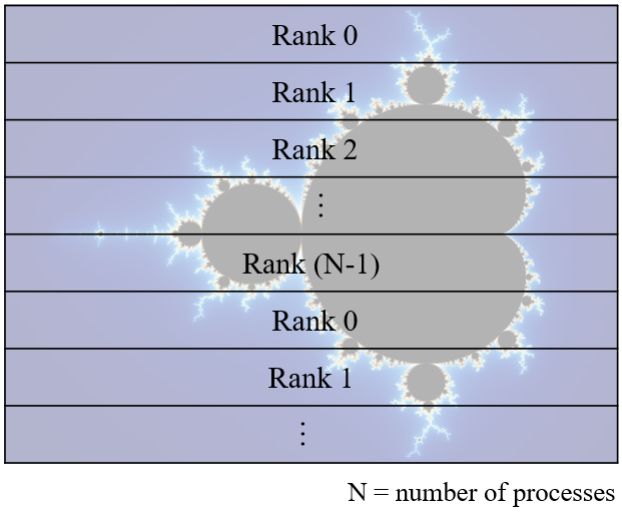

Figure 4: Static Row cyclic partition

By allowing to specify the number of rows to be cycled, it can operate as both cyclic and cyclic blocks.
Advantages: The workload is distributed more evenly compared to row block partitioning.
Disadvantages: The number of communication increases and the communication cost increases. More complex implementation.

3.2 Dynamic scheme

First Come First Served : Row Based Partition Scheme [2]
The dynamic variant aims to more evenly distribute the workload out on all the processors. In this approach,

the master process takes on a fully managerial job and does no actual Mandelbrot set calculations itself. Instead, it is responsible for distributing tasks to all the other processes and assembling the final data when they are done calculating and have sent the results back. While this does mean there is one fewer process working on solving the task, theoretically fewer processes will be left doing no productive work while others are overloaded and there should be less time spent at the end waiting for the final process to finish. Pseudo code for this variant is shown in listing 2, and a visual representation of how the master distributes the work can be seen in figure. 5

```
int main (){
    Initialize MPI and variables
    if (rank == 0){
        for each rank{
            Send index of current working row to all ranks
        }
        while(!done){
            Receive work done from any other rank
            Store the received work locally
            if (current_row != last_row){
                Send next work row to an available worker
            }
            else {
                All work is done, send termination to workers
            }
        }
    }
    else {
        while (1){
            Recieve work order from master
            if (work_order == -1 ){
                break;
            }
            Solve Mandelbrot
            Send results to master
        }
    }
    Finalize MPI
}
```

Listing 2: Dynamic implementation

Advantages: At runtime, tasks are allocated according to actual needs, so the work can be most evenly distributed.
Disadvantages: The master process of allocating work does not participate in computational work, which means that one process is lost. Communication overhead can occur due to increased number of communications.

4 Experiments

This section presents the experimental data produced by implementations of the designs described previously. Unless otherwise stated, all the results provided were obtained by executing the relevant implementation on UiT's locally hosted uvcluster. When comparing different variables across solutions (static vs. dynamic, static linear vs. static round-robin, etc.), only the variable shown in the data is changed while the others remain at a set default value. These default values are:
Hosts: 8
Threads: 8
Resolution: 2000x2000
Zooms: 10

All results are the average of five runs in order to account for, among other things: the innate unpre-

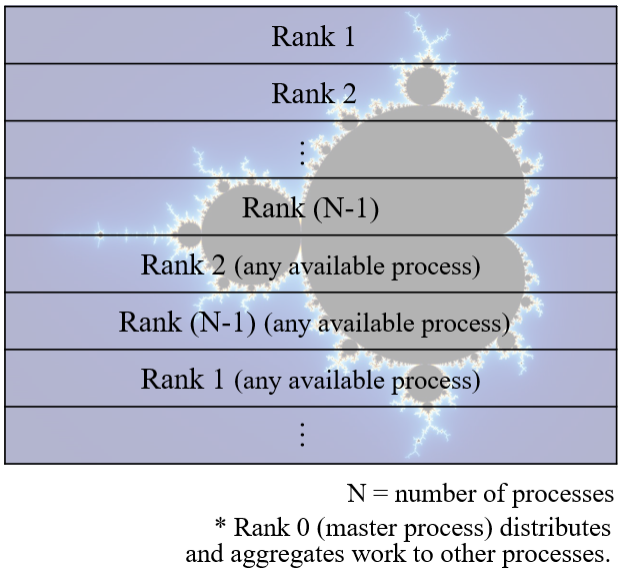

Figure 5: Dynamic Row (block) partition

dictability of inter-host communications and the delays that follow, differences in server load caused by other users, variances in physical distance between the executing hosts, etc.
And When running each solution for comparison, the same host file was maintained to minimize the impact of hardware variables.

Due to the large time difference between the sequential and parallel implementations, the experimental data produced by the sequential one will only be shown in the tables, not the graphs. The reasoning behind this is that if sequential time results were to be included in the graphs together with the three other solutions, the variance between the three parallel solutions would not be visible to the naked eye.

4.1 Resolution variance

Table 1: Rumtime according to resolution changes

	400	800	1200	1600	2000	2400	2800	3200
sequential (avg)	0.516234	2.035123	4.556123	8.104235	12.610243	18.214523	24.829645	32.426802
static row (avg)	0.167162	0.436782	0.903208	1.555443	2.385732	3.407463	4.595222	6.01065
static row (std)	0.003132	0.012521	0.012379	0.012088	0.009432	0.006641	0.012472	0.013255
static row RR (avg)	0.151951	0.411138	0.828038	1.434072	2.188831	3.121271	4.213025	5.485823
static row RR (std)	0.001566	0.001357	0.004101	0.011773	0.008319	0.005566	0.005092	0.009093
dynamic row (avg)	0.174504	0.435446	0.829463	1.348863	1.973175	2.782521	3.696732	4.696184
dynamic row (std)	0.003261	0.003375	0.009524	0.007413	0.010118	0.011896	0.018689	0.025277

As the resolution increases, the execution time increases in proportion for all three solutions. It is seen that when the resolution is increased, the speedup of the dynamic solution is relatively fastest.

4.2 Process number variance

Because the sequential implementation by design does not use multiple processors, it is not included in this section. However, the result produced by the sequential implementation using the default parameters for resolution and zoom was 12.610243, as shown in table 1.

* In the case of row cyclic solution(round robin), an error occurred when handling odd or many processes. We haven't yet been able to fix this due to time constraints, but we will fix it later.

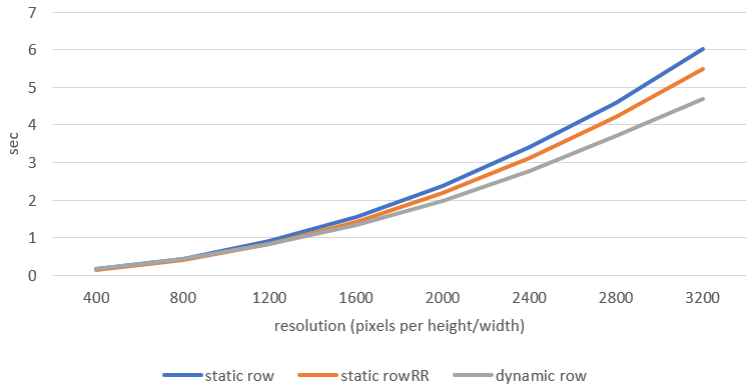

Figure 6: Runtime according to resolution changes

Table 2: Rumtime according to number of processes

	2	4	8	16	32	64	128	3
static row (avg)	7.662161	4.050854	2.390997	1.603962	1.723934	2.841213	3.982618	5.4059
static row (std)	0.022292	0.00523	0.013422	0.017601	0.004346	0.166428	0.039019	0.004438
static row RR (avg)	5.26273	2.70435	2.191078	2.004573	*	*	*	*
static row RR (std)	0.005266	0.005436	0.011282	0.003876	*	*	*	*
dynamic row (avg)	10.58014	3.570112	1.967204	1.231465	3.00198	3.698462	6.360369	5.320291
dynamic row (std)	0.004308	0.003254	0.01176	0.00977	0.089566	0.112462	0.649671	0.004769

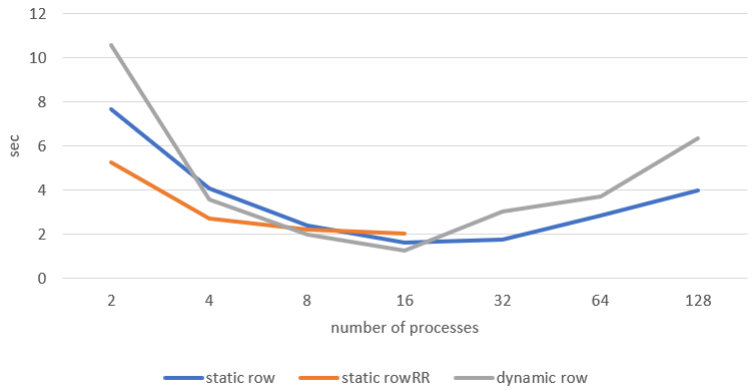

Figure 7: Runtime according to number of processes

As the number of processes increases, the performance time is reduced to confirm the effectiveness of parallel programming. However when more than a certain number of processes are allocated, the performance time is rather longer. This may be an in-solution issue using a block-based functions. In the future, we want to implement and test non-block based solutions.

4.3 Host number variance

The number of hosts generally did not affect the speed of performance if there were more than two hosts. What is interesting is that when a host is one, it would be faster because that communications occur between only internal processes, but rather higher. This could be due to the fact that the default number of threads used in these tests are 8, and one node might not have exclusive access to 8 hardware threads.

Table 3: Runtime according to number of hosts (nodes)

	1	2	3	4	8	16
sequential (avg)	12.610243	N/A	N/A	N/A	N/A	N/A
static row (avg)	4.146669	2.385868	2.389549	2.387066	2.384424	2.159616
static row (std)	0.042981	0.009087	0.0133	0.013277	0.011904	0.010799
static row RR (avg)	2.991592	2.191406	2.190756	2.19762	2.197023	2.147043
static row RR (std)	0.054749	0.00887	0.012167	0.014016	0.011547	0.011903
dynamic row (avg)	4.116663	1.969688	1.970514	1.975176	1.976126	1.999376
dynamic row (std)	0.541137	0.009101	0.008893	0.009819	0.006279	0.012159

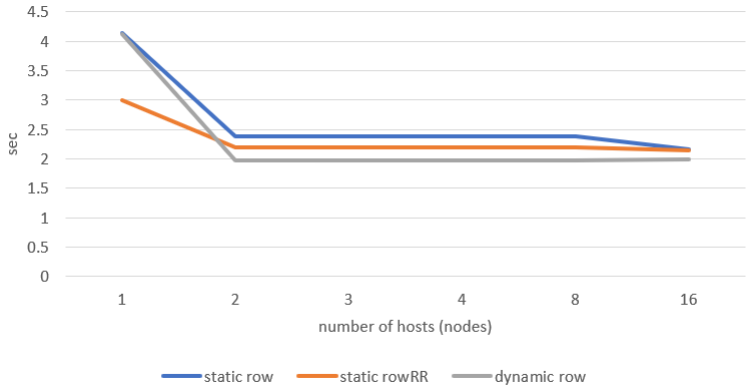

Figure 8: Runtime according to number of hosts (nodes)

Table 4: Rumtime according to number of zoom levels

	5	10	15	20	25
sequential (avg)	6.661231	12.699033	18.611235	24.571890	30.457129
static row (avg)	1.208877	2.160386	3.100834	4.037237	4.98853
static row (std)	0.010302	0.007422	0.018567	0.010992	0.013908
static row RR (avg)	1.170012	2.138196	3.121115	4.091904	5.050271
static row RR (std)	0.013733	0.005261	0.018783	0.013225	0.005826
dynamic row (avg)	1.088722	2.005584	2.9101	3.81532	4.712103
3dynamic row (std)	0.014692	0.007166	0.021258	0.017994	0.03027

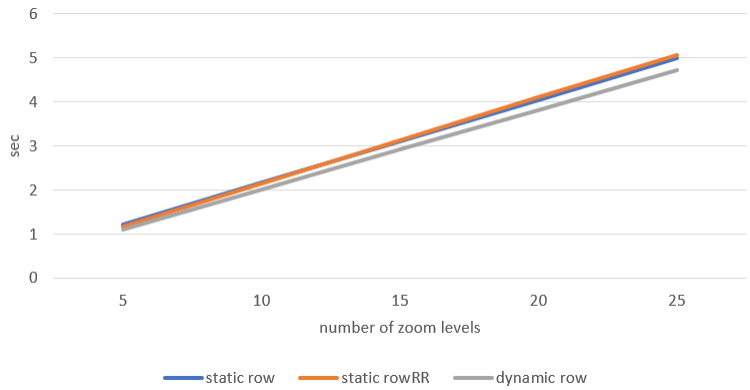

Figure 9: Runtime according to number of zoom levels

4.4 Zoom levels variance

As the number of zoom levels increases, it can be seen that all three solutions have linearly proportional performance times. The zoom level serves to increase the workload proportionally.

5 Discussion

The speedups and efficiencies of parallel solutions through the above experiments are as follows. As the number of processes increases, the speed-up increases and efficiency decreases gradually.

Table 5: Speedups and Efficiencies of parallel programs

Processes		1	2	4	8	16
parallel (static)	S	1.22264921	1.658667	3.125036	5.306343	7.909742
	E	1.22264921	0.829334	0.781259	0.663293	0.494359
parallel (staticRR)	S	1.221903893	2.41153	4.703282	5.79294	6.32552
	E	1.221903893	1.205765	1.17582	0.724117	0.395345
parallel (dynamic)	S	-*	1.199811	3.553071	6.410394	10.27621
	E	-*	0.599906	0.888268	0.801299	0.642263

* In a dynamic solution, the root process is implemented in charge of allocation and collection of work, not actual calculation.

Using MPI, each process' results are copied to the buffer and handed over to the root process. When the buffer got too big, it would occationally overflow the buffer and cause the program to have a segmentation fault. To solve this problem, the data that is being sent via the buffer could be cut down into smaller messages. However, this would in turn cause more communication overhead, so there is a balance that must be kept here. Therefore, it is necessary to think about how to find this balance point.

Column-based partitioning has not been tested due to time limitations, but speedup is expected to be slightly lower than row-based partitioning due to the cache line processing.

Investigating the profiling data for the parallel dynamic and static implementations reveals how the threads communicate with each other, approximately how long they have to delay while waiting for a response and other useful insights. Score-P was used to generate the profiling and trace data, while Vampir was used to provide a graphical interface to analyze it. See figures 10 and 11 for some examples of some of the data presented by Vampir.

In figure 11, it is clear that there is a lot more communication between the master and worker processes in the dynamic implementation than with the static implementation. Note especially the difference in time scale, where figure 11 is measured at this zoom level in milliseconds, figure 10 is scaled to actual seconds, yet the dynamic implementation has many more messages being passed. It should also be noted that these are examples with 16 threads, so they must carry a lot of communication overhead regardless of implementation. Vampir shows that the implementations have a rather glaring bottleneck, that being the use of MPI_Send and MPI_Recv. This is especially prominent in the dynamic variant, where messages are being passed much more frequently than in the static variant, even though they are much smaller. These message passing functions are by default blocking, meaning that the functions do not return a value (read: allow the processor to continue) until the receiver has actually received the data that is being sent. Improving the implementations by using non-blocking and/or buffering variants of MPI_Send and MPI_Recv where possible would probably improve the performance of all implementations significantly to varying degrees.

Despite these potentials for improvement in the implementation, the speedup provided by parallelizing the Mandelbrot set calculation is significant in all the parallel implementations. Compared to the sequential implementation, the speedups are between 4x and up to nearly 7x depending on what variables are being changed. These speedups are with 8 nodes and 8 threads compared to the sequential singular thread. The

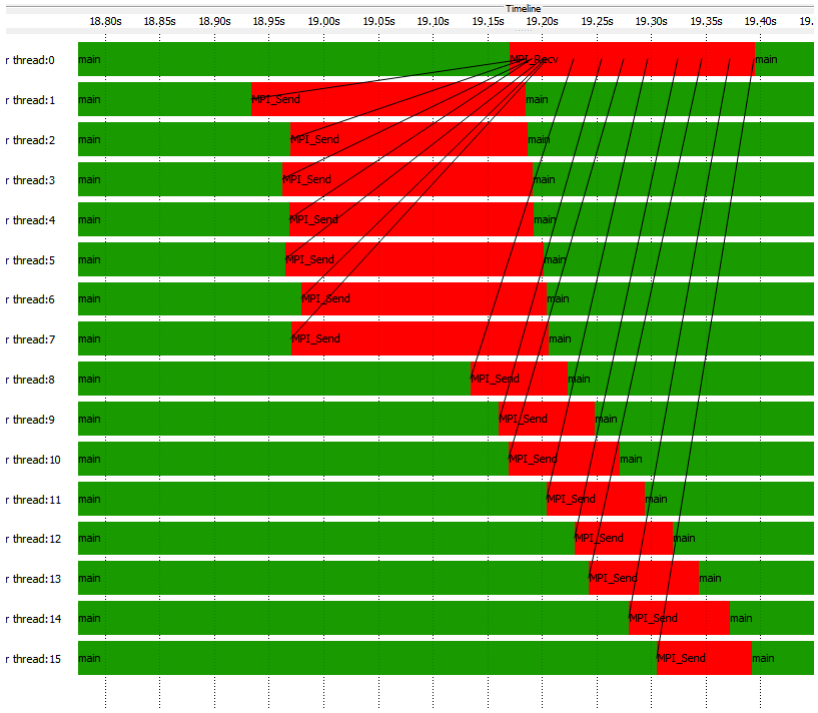

Figure 10: Static Vampir data w 16 threads & 2 nodes

Figure 11: Dynamic Vampir data w 16 threads & 2 nodes

highest speedup ratio seems to be at 3200x3200 resolution, with the dynamic implementation (see table 1), with a speedup ratio of 6,9. Given the aforementioned room for improvement on the communication overhead front, this appears to be very good.

Another thing that is also made apparent by the Vampir tracing data for the static implementation (figure 10) is the fact that the master thread will stop and wait for thread #1 to finish its work and send the data over the MPI interface, regardless if any other threads are ready or not. This could be improved by having the master thread wait for any thread to send data, not sequentially as is the case now. This should further cut down on time lost being blocked.

To summarize this section:
- Buffers should be kept small to avoid segmentation faults and traffic congestion
- Only row-based sectioning has been tested, but it is assumed that column-based sectioning would provide marginally worse results due to how the Mandelbrot set is positioned
- The implementations used to produce the experiment data has room for improvement, particularly in adding non-blocking message passing functions
- Parallelzing the Mandelbrot set calculations were highly successful regardless of these shortcomings

6 Summary

Let's look at our expectations again.

1. Allocating multiple of processes will improve overall work speed / performance.
2. Equal distribution of the workload will improve speed/performance.
3. If too many processes are used, the performance will be slightly degraded due to communication overhead/cost.

The results of investigating the expectations through the above experiments are as follows.

1. Allocating multi-processes has improved overall work speed and efficiency.
2. The even distribution of workload improved the speed and performance of work.
3. When too many work processes are used, it has been confirmed that there is a decrease in speed and performance due to communication cost.

In summary, parallelzing the Mandelbrot set generation using the design described above gives significant speedup compared to sequential. However, at some point the communication overhead produced by having to orchestrate traffic to and from many nodes and threads becomes greater than the gain provided by their resources.

References

[1] P. Alfeld. The mandelbrot set (http://www.math.utah.edu/ alfeld/math/mandelbrot/mandelbrot.html).

[2] B. Manesha Samarasekara Vitharana Gamage and V. Monn Baskaran. Efficient generation of mandelbrot set using message passing interface. *arXiv e-prints*, pages arXiv–2007, 2020.

[3] A. Mathematics and C. Science. Message passing interface (https://www.mcs.anl.gov/research/projects/mpi/).

[4] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.