HPC Report

Uros Zivanovic

Github repo: https://github.com/Chromeilion/hpc_project

Instructions on compiling and running the code can be found in the GitHub repository.

Exercise 1

Here we test MPI BCast and MPI Gather operations using the OSU Micro-Benchmarks software. All tests are done on ORFEO THIN nodes. Therefore, the nodes we are running on have 768Gb of RAM and 2 NUMA nodes each containing 12 cores for a total of 24 cores of compute.

To give a quick description of the BCast operation, it consists of some process sending some value to all other processes, therefore doing a broadcast.
The Gather operation on the other hand receives a value from all other processes.
This means that after an MPI Gather, the gathering process will have an array of values, with one element per other process.
For example, if we have 5 processes, upon completing a Gather operation we would have received values from all other processes into an array of size 4.

Compiling OSU Micro-Benchmarks

To ensure optimal performance, the benchmarking software is compiled on the machine type it is going to run on using the version of MPI installed on that machine.
This step is done independently from running the actual tests so as to save on compute.
After compiling, the binaries are installed into a custom prefix which can then be used when running tests.
Therefore, the compiler is able to fully utilize the features of the Intel Xeon Gold 6126 cpus and openMPI version 4.1.6 on the THIN nodes. Compiling like this also avoids any need for static linking which can cause issues with OpenMPI.

Comparing Algorithms

Each of the 2 tested operations (BCast, Gather) has multiple algorithms implemented in MPI. These are chosen through the coll_tuned_bcast_algorithm and coll_tuned_gather_algorithm command line arguments. We choose the first 3 algorithms for both operations.

Therefore, for BCast we have the following:

1. Basic linear
2. Chain
3. Pipeline

And for Gather we have:

1. Basic linear
2. Binomial
3. Linear sync

The differences between these algorithms comes down to the way that nodes are organized when sending messages between each other. The basic linear algorithm for example arranges all nodes in a tree of depth 2 with the calling node as the root. Then the calling node communicates with each leaf node in the array sequentially.
Usually, MPI selects the algorithm automatically based on certain heuristics.

Experimental Details

Each benchmark is run 10 times using 2 THIN nodes, with the final result per benchmark being the average of these runs. We test for 3 parameters; message size, latency, and number of processes. While OSU Micro-Benchmarks automatically tests the message size and latency, the number of processes is controlled through the *-np* flag when using *mpirun* and we always keep it even (2, 4, …, 48). Lastly, the processes are mapped by node, meaning that there will always be an equal number of processes per node (in this case half of processes on one and half on the other).

Results

Some of the plots provided (e.g. *Fig 4*) use the average normalized latency. This value is calculated by first dividing all latencies by their corresponding message sizes and then taking the average for a specific number of processes.
For example, a message of size 100 with latency 200 will be normalized to $\frac{200}{100} = 2$. Therefore, the final normalized average showcases how well an algorithm performs for a certain number of processes across a wide range of message sizes.

BCast

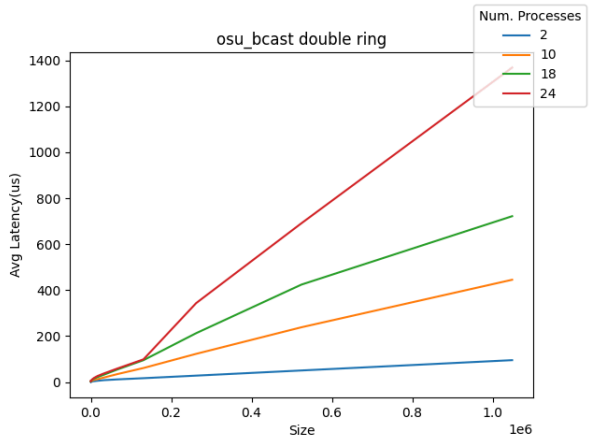

Fig. 1

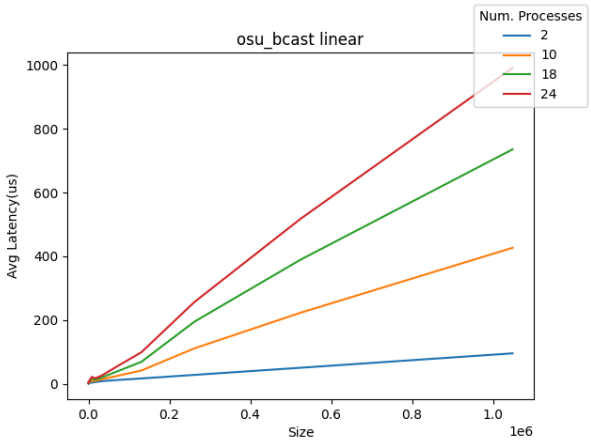

Fig. 2

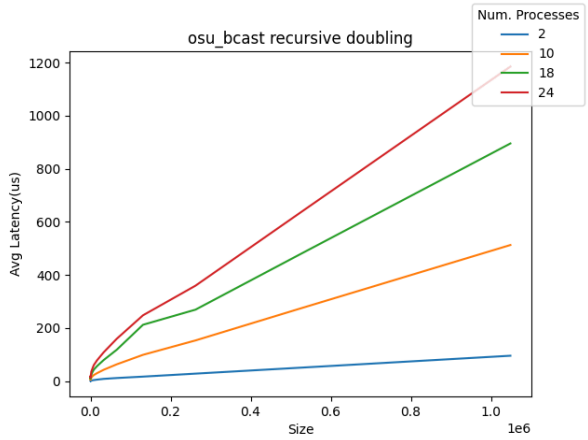

Fig. 3

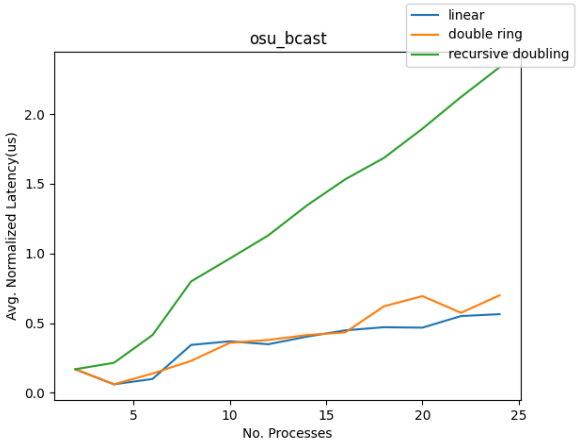

Fig. 4

Observing *Fig. 4*, we see significantly worse average normalized latency with the recursive doubling algorithm, while both linear and double ring algorithms give similar results.
Comparing *Fig.3* with *Fig. 1* and *Fig. 2*, we can also see that the scaling of the recursive doubling algorithm is significantly worse at lower message sizes.
Although these plots showcase how good each algorithm is across many message sizes, it's also interesting to look into the performance when keeping message size static.

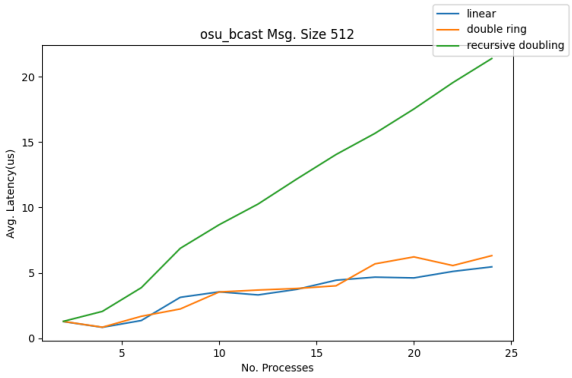

Fig. 5

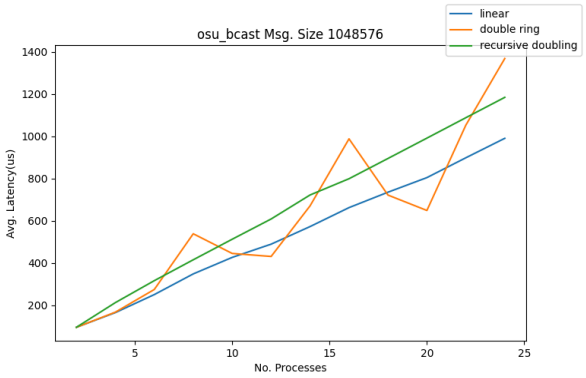

Fig. 6

Although the recursive doubling algorithm performs worse than the other two when small message sizes are used as shown in *Fig. 5*, it actually performs similarly to the other algorithms at large message sizes as shown in *Fig. 6*.

Gather

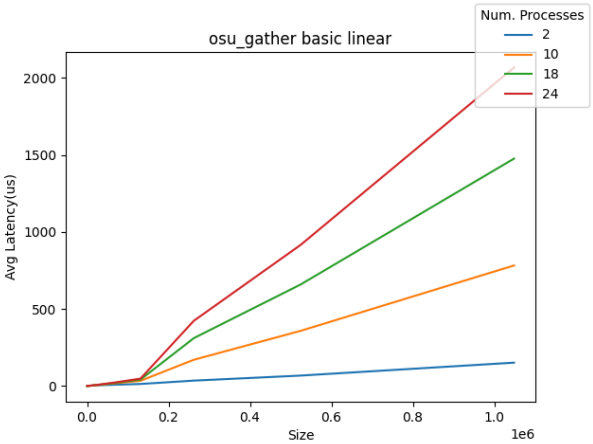

Fig. 7

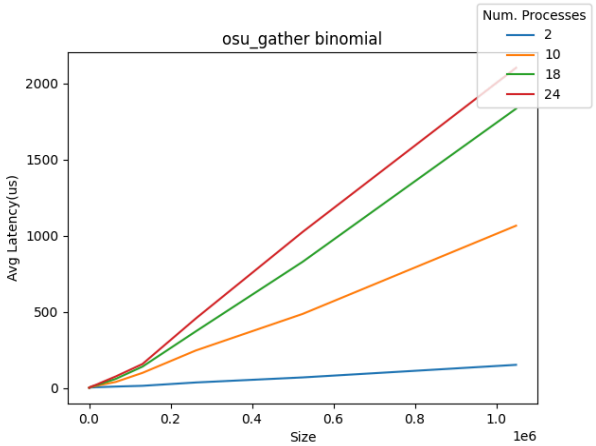

Fig. 8

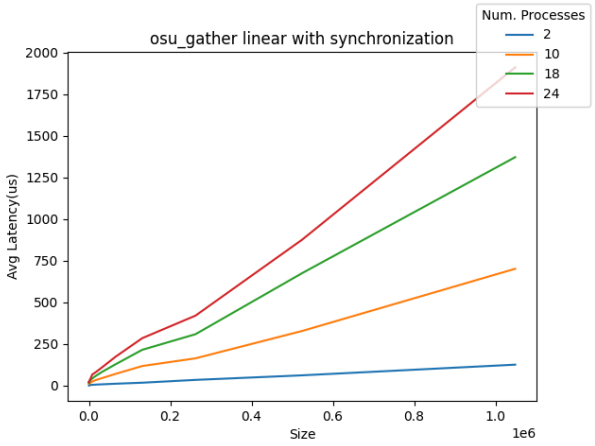

Fig. 9

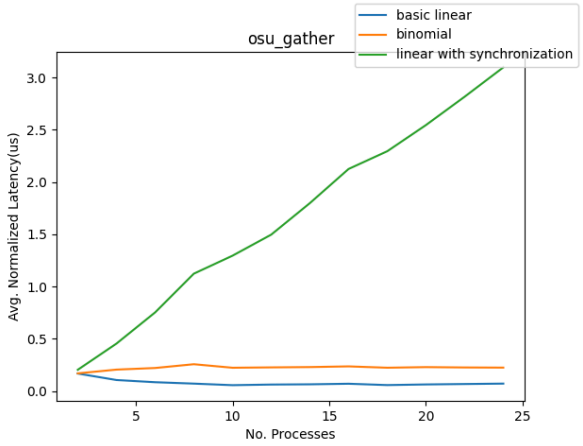

Fig. 10

With the gather operation we see the biggest difference in the linear with synchronization algorithm in *Fig. 10*. The reason for why it looks so much worse than the other algorithms comes down to its performance at low message sizes however.
Looking at *Fig. 9*, we see that the algorithm scales very poorly at low message sizes, with large differences in average latency between different numbers of processes at small message sizes.

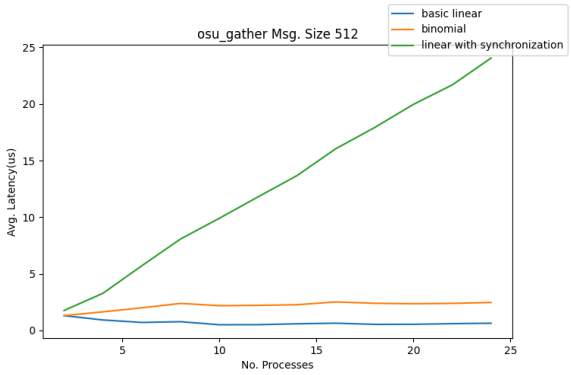

Fig. 11

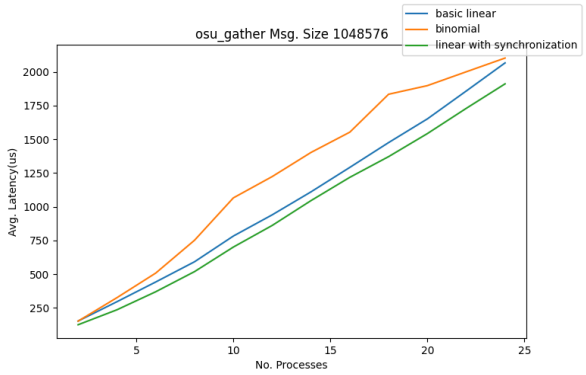

Fig. 12

When keeping the message size constant, we can see quite clearly how the linear with synchronization algorithm goes from performing poorly at a low message size (*Fig. 11*), to being even slightly better than the rest at large message sizes (*Fig. 12)*.
Overall, the basic linear algorithm seems to provide the best all round performance between these 3, although with large messages linear with synchronization may perform a bit better.

Exercise 2

Here we showcase the results of exercise 2a + 2c. 2a is done using MPI point to point calls while 2c is done using OMP.

2a

Here we implement and showcase the performance of an Allgather algorithm on a ring data structure. Allgather is an operation which is very similar to Gather, however instead of only 1 node receiving data from the rest, all nodes receive data from all other nodes.
For example, if each node shares its rank with the other nodes, at the end of the Allgather, all nodes will have a list of all ranks.

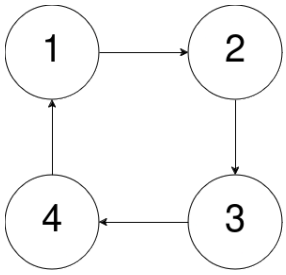

Fig. 13, an illustration of a ring structure.

The algorithm is implemented on a ring, meaning that each process only ever sends data to the process one rank higher than its own, and only ever receives data from the process one rank lower than its own. For a number of processes *P,* performing an Allgather takes *P-1* steps. During the algorithm, each node sends the value it currently has stored to the next node in the ring, and receives a value from the previous node in the ring. In the next step, the value previously received is sent on to the next node and another is received. Once this is complete, all nodes have shared their values with all other nodes.

Although this algorithm has the benefit of being simple and versatile, it is also limited by its diameter, which is equal to *P-1* (diameter being the maximal distance between two processes in the ring). This means that for any algorithm needing to share data between distant processes (such as Allgather), the lower bound on the number of required steps is *P-1*. Since the algorithm presented here takes exactly that many steps, it is optimal on this data structure.

When conducting tests, we rerun the algorithm 4 times and use the average value. The official OpenMPI implementations are tested using OSU Micro-benchmarks, also rerunning 4 times. Processes are distributed by equally over 2 nodes, and only even numbers of processes are tested.

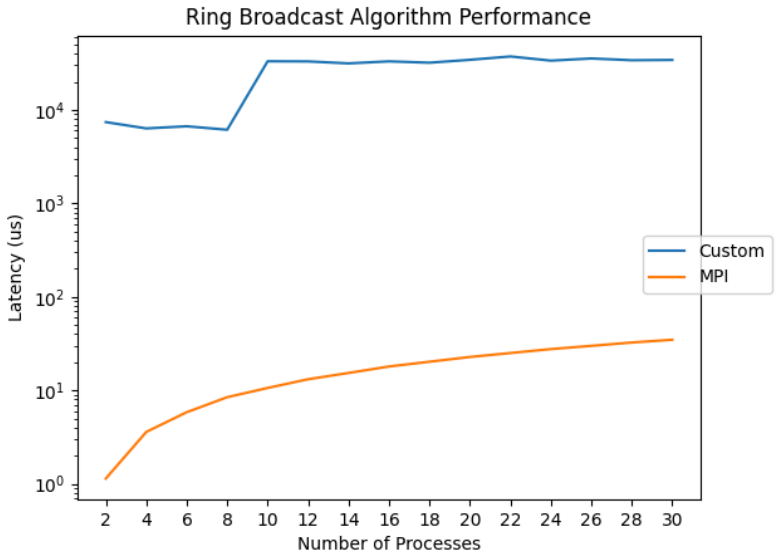

Fig. 14a, performance of the custom Allgather vs the official OpenMPI implementation

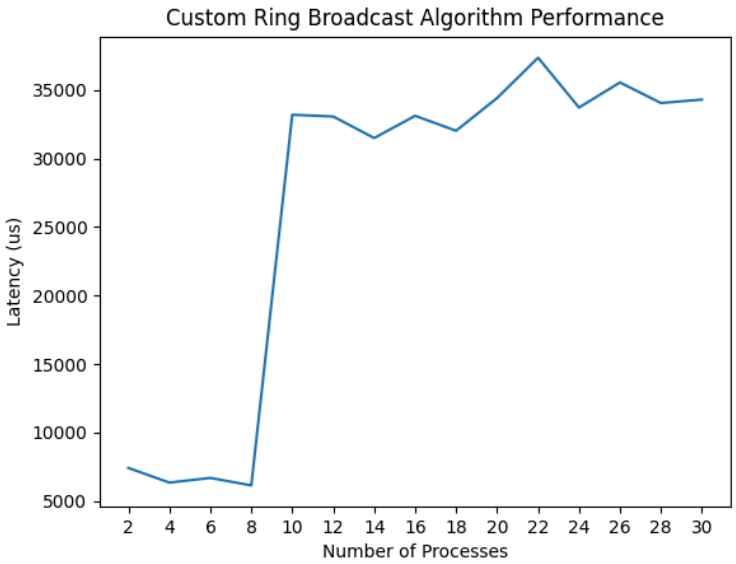

Fig. 14b, performance of only the custom Allgather implementation

When comparing the custom implementation to the official one (also utilizing a ring data structure) in *Fig. 14a*, we see that the custom one ends up being significantly slower. In *Fig. 14b* we can also see a sudden jump in latency when more than 8 processes are used. Additionally, aside from the big jump, the performance seems to slowly go up over time. We hypothesize that the sudden jump in latency comes from a slow connection being introduced into the ring.

Because all data flows through each processing node at some point, this means that the speed of the algorithm is limited by the slowest connection in the ring. Therefore, it's possible that by adding a process that's running on a different NUMA node or even a completely different

machine, the algorithm will suddenly slow down significantly. The custom implementation also heavily relies on MPI_Sendrecv, which may have its own scaling properties influencing the overall algorithm.

2c

We implement multithreaded Mandelbrot set calculation using OpenMP. Although this problem is quite easy to parallelize, it does offer some challenges. To be precise, the computation is not uniformly distributed on the complex plane, but instead focuses on the very narrow border region of the Mandelbrot set. Therefore, naively splitting up the complex plane into square regions and uniformly assigning these to processors can lead to unbalanced processing times, resulting in most of the program waiting on one process which received a significantly higher workload than the rest.

To deal with this issue, we implement a 3 step algorithm that first runs using a very low number of iterations, then a medium number, and finally with the maximum number of iterations. By splitting up the processing, we can cut out large regions of the complex plane that are obviously not in the Mandelbrot set during phase 1 and 2. This results in a much more uniform compute distribution across processors and better overall CPU utilization. The downside of this procedure is that we need to redistribute the different regions of the complex plane after the first and second steps, which can take some time and scales with the size of the requested final image.

Taking a step back, let's first describe the basic working of the algorithm. The inputs to the program are the output image resolution, complex plane area to be rendered, and the number of iterations for each pixel. First, we assign each pixel of the output image to its closest coordinate on the complex plane. Each pixel is represented by a struct with the properties *x, y,* and *data.* The *data* attribute is an unsigned char and is used to store the number of iterations required before being able to decide if the pixel is in the Mandelbrot set.

```
// A semifinished pixel. Useful for caching.
typedef struct {
    pixel *pxl; // Actual pixel object
    unsigned int iter_no; // The current iteration of the pixel
    bool completed; // Whether the pixel has been fully computed
    complex long double c_data; // The computed complex value of the pixel
    complex long double z_data; // Computed value for the current iteration
} unfinished_pixel;
```

Fig. 15, the unfinished pixel struct.

In addition to this, each pixel is wrapped using an additional struct, which stores the pixel itself, as well as the current iteration the pixel is on, whether the pixel is completed, and the pixel's complex coordinates and current computed value from the current iteration of Mandelbrot set calculation. All pixel structs are stored in one array (*Pix*), while unfinished pixel structs are

stored in another (*U*). Additionally, we also maintain a third array consisting of pointers (*U**), which point to unfinished pixels in the *U* array that have the completed flag set to false.

To parallelize the code, we simply use *OMP PARALLEL FOR* on the *U** array. During the first step we keep a low number of iterations. Once the parallel computation is complete, we iterate through *U** in the main thread and move all pixels still marked incomplete to the front of the array, saving how many pixels are still left. Then the leftover pixels are distributed over all processors again, this time running with a higher max number of iterations. Because the current complex value and iteration number are saved in the struct, the threads do not have to start from zero.

The next time we need to remove completed pixels, we don't have to iterate through the entirety of *U** any more, but only until the number of incomplete pixels we saved earlier. Therefore, the biggest overhead of this algorithm occurs during the single threaded section when we're recalculating *U** for the first time, after the first iteration most "easy" pixels have been removed, and *U** becomes much smaller.

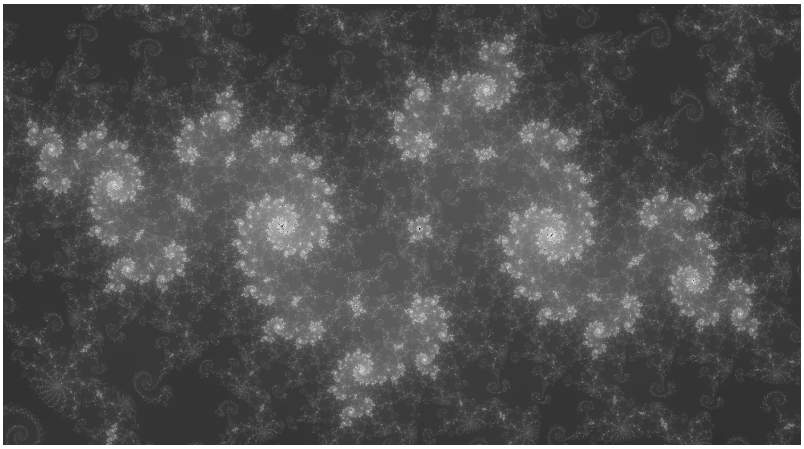

Fig. 16, an example render of a section of the Mandelbrot set

To give additional motivation on the utility of the unfinished pixel struct (*Fig. 15*), it stores the iteration of the pixel as an unsigned int. This is the exact same information as what is stored in the *data* attribute of the pixel struct. However, because we use an unsigned int, we can use much higher values for the number of iterations. This makes it possible to zoom in much further into the Mandelbrot set while still being able to output to a PGM image. When rendering the final image, we simply normalize the number of iterations to be between 0 and 255.

Another important detail in the unfinished pixel struct is the use of a complex long double to represent the actual complex number data for the pixel. The precision of the complex numbers used when doing calculations directly impacts how far one is able to zoom into the complex plane. By using a relatively high precision we enable images such *Fig. 16* to be made, which otherwise result in corruption when using the regular double data type. This precision does have a large impact on the speed of all numerical calculations however. Therefore, if one only wants

to render low depth sections of the Mandelbrot set, changing to lower precision would be worth it.

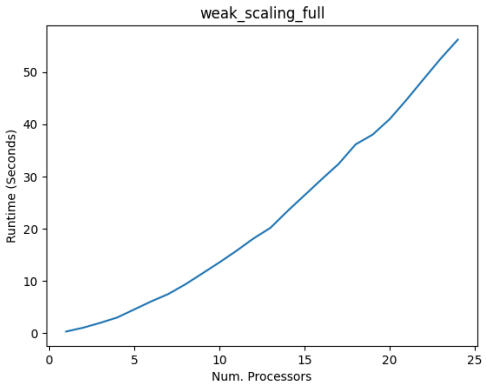

Fig. 17, full weak scaling of the algorithm

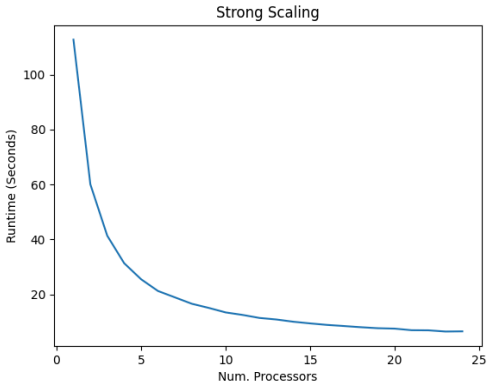

Fig. 18, strong scaling of the algorithm

When benchmarking the code, we test for both weak and strong scaling. Weak scaling is tested by linearly scaling the number of iterations and image resolution with the number of processes. We scale up to a max of 500 iterations and a square image resolution with width 500. Strong scaling is tested by keeping the iterations constant at 500 and a constant size square window with width 2000. We then increase the number of processors. Each test is run 4 times and the average is used as the final value for each point.

Overall, the scaling of the algorithm is not terrible. The weak scaling is close to linear, and strong scaling also performs nicely. In *Fig. 18*, we can see that the strong scaling seems to be leveling off as we increase the number of processors, which is to be expected. The single threaded portion of the code starts to dominate the runtime after a certain number of processors has been added. That being said, the image being rendered is quite computationally expensive so a runtime under 6 seconds is impressive.

Misc Mandelbrot Set Implementations

Although slightly unrelated to the original task, here we present a little bit of extra fun that can be had with the Mandelbrot set and the code presented above.

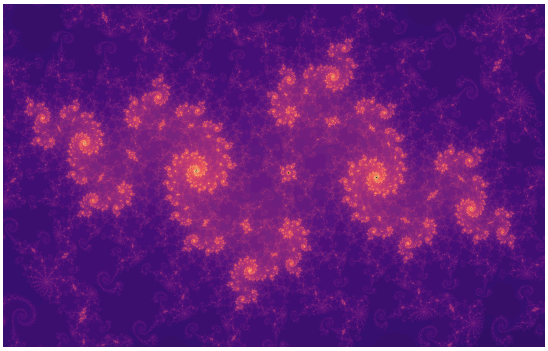

Fig. 19

By wrapping the original C code using Python, one can easily use the calculated values from the C code and the Matplotlib library to create beautiful images such as the one presented in *Fig. 19*. Matplotlib is useful because it contains many color pallets, making it easy to experiment.
In addition, one can also automatically create a video where one zooms into the Mandelbrot set. An example generated by the code presented here and in the GitHub repo can be found here.