1.0 Introduction

The Mandelbrot Set is a mathematical set of complex numbers discovered in the 1970s and defined by an iterating simple equation:

$$z_{i+1} = z_i^2 + \kappa$$

To visualize the Mandelbrot set, we can create an image where each pixel corresponds to a point on the complex plane. We can then color each pixel based on whether the corresponding complex number is in the set or not. This report will focus on the task of parallelizing the calculation of the Mandelbrot Set and generating its corresponding images over static and dynamic task assignments. Moreover, the report will discuss performance, speedup, efficiency, and relevant topics.

1.1 Setup

First, we will begin by setting up our device by installing the OpenMPI from https://www.open-mpi.org/software/ompi/v4.1/. Then, we set up the environment and paths on the terminal and properly check if it works by checking if the following input gives a correct version output:

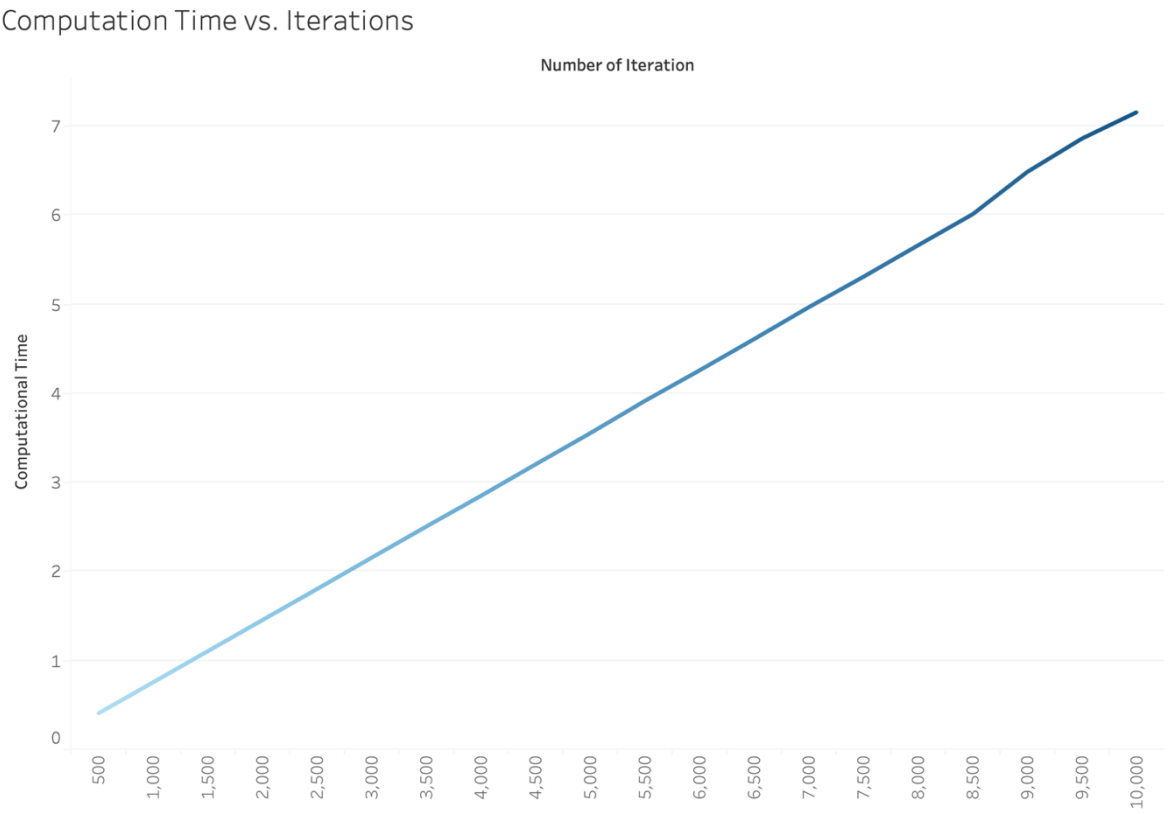
```
$HOME/opt/usr/local/bin/mpirun –version
mpirun (Open MPI) 4.1.5
```

1.2.0 Sequential Code

The sequential code contains the regular way of coding where we simply create the method and assume an iteration number. A small bash script was created to automate the iterations and the output which will later be discussed and be shown useful as we discuss the dynamic and static codes. Only the computation time was computed as no communication time is taken into consideration as one processor is assumed to be performing the task. (I have no knowledge if MacOS parallelizes such simple code programs, but I am assuming not for this assignment).

Code available at https://github.com/OsamaShamout/Parallel

Result



We can clearly notice how the computation time increased almost in a linear fashion with the iterations. This is to say that as more pixels were coming into play, of course, more computations were needed, and more processing time was needed. Yet, this is just preliminary as we need it to compare with the dynamic and static outputs.

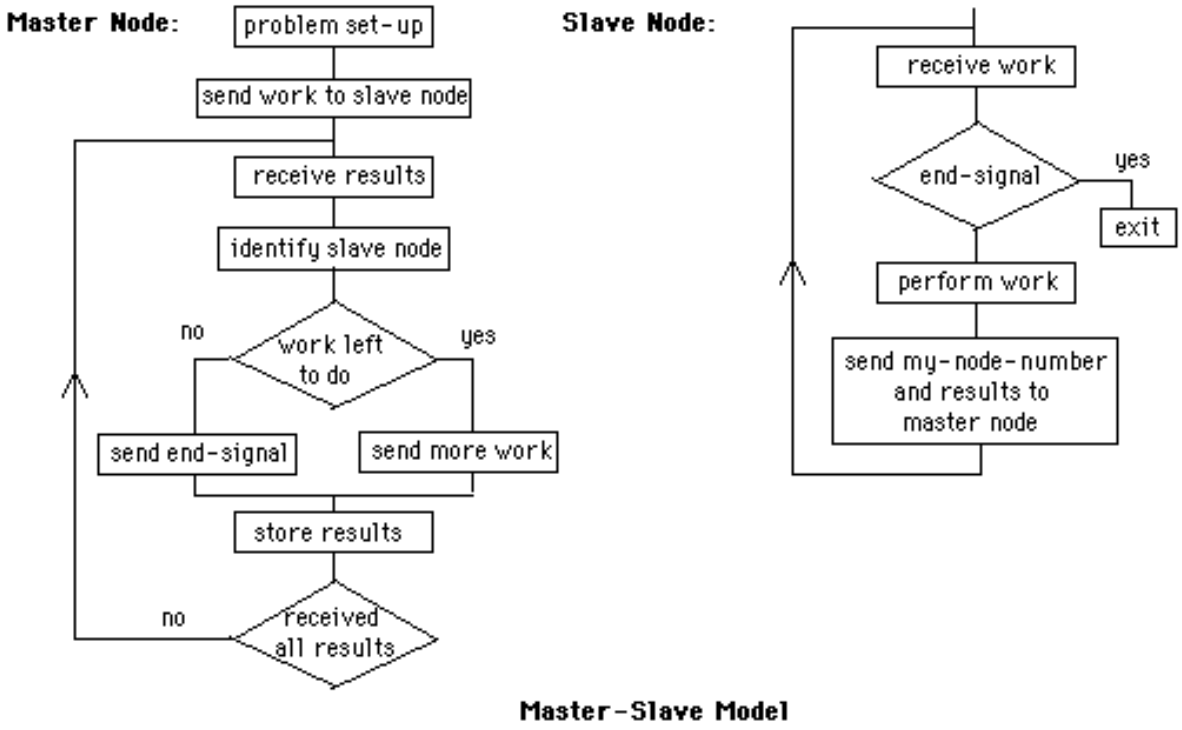
2.0.0 Dynamic Code

The used code has been provided from the following repository:

https://github.com/OsamaShamout/Parallel Contribution of Matrinohmann

Dynamic parallelization involves distributing parts of the pixels initially and distributing the remaining parts as processers become available. This happens whenever and whichever processor become available. Both the master and the slave processors in the code use a **dynamic** method of allocating tasks. The master process assigns rows to the slave processes as they become available, and the slave processes receive and compute rows as they are assigned to them by the master process. This allows the tasks to be distributed dynamically and evenly among the available resources. *Notice the allocation and deallocation of processes.*

In essence, the main() function initializes MPI and sets up the parallel processing by creating two different processes: a master process and multiple slave processes. The master process is responsible for coordinating the work of the slave processes and generating the final image while the slave processes generate portions of the image respectively by the master_proc() and slave_proc() functions and called from main() (depending on the process ID).



2.0.1 Dynamic Screenshots

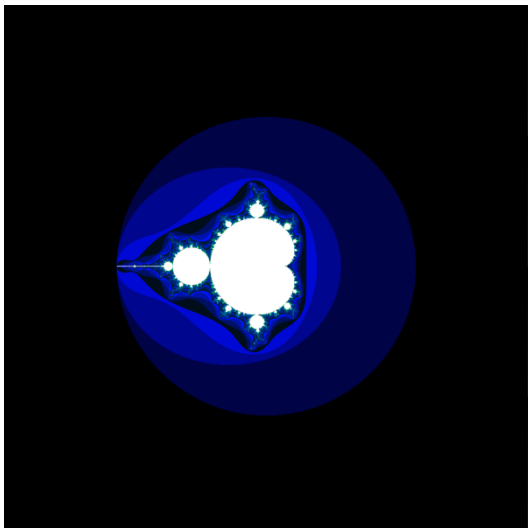


Figure 1: np=4, n=20000, a=3.5

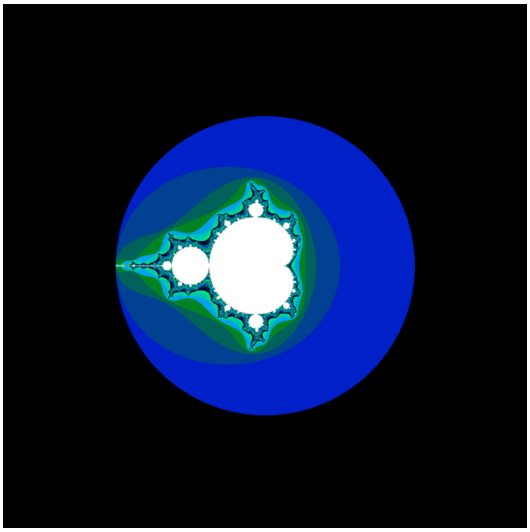


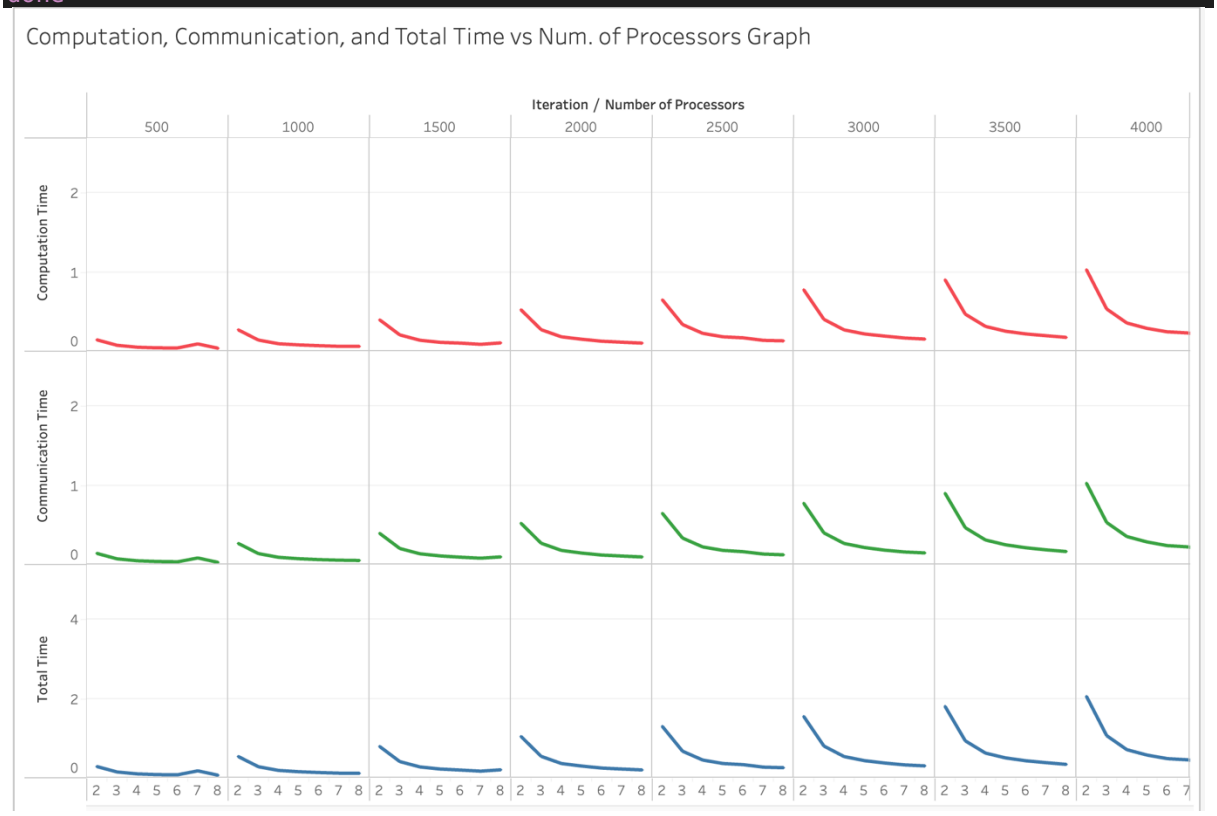
Figure 2: np 4, n=2000, a=3.5

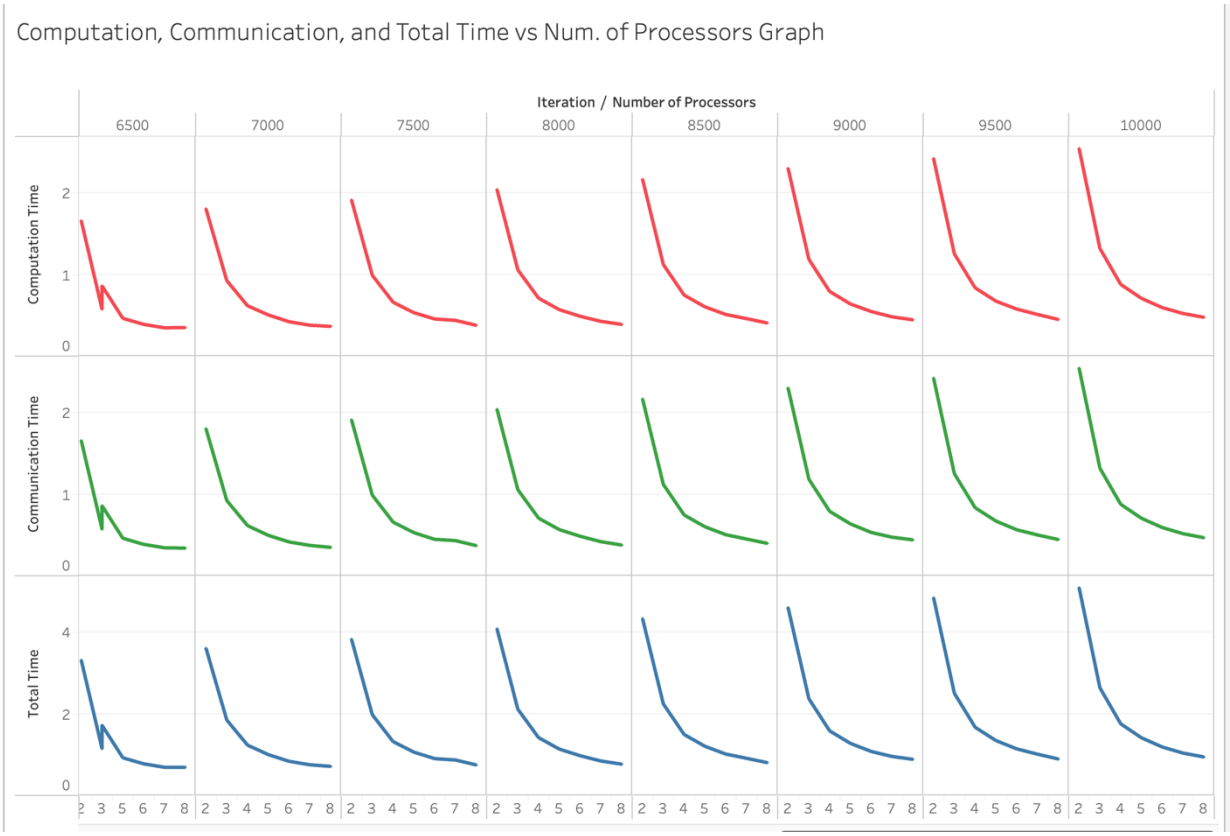
2.0.2 Methodology

We are running a MacBook Pro M1, with 8 cores, compiling using Apple Clang. Therefore, we will be trying for 8 processors. Also, scaling for up to 10,000 iterations running 500 a step. To save time, I wrote run a small shell script to automate the process and record the maximum iteration, computational and communication times. The number of processes is deduced from the script itself. The script is also available in the repo.

```
#!/bin/bash

# This script will run the mandelbrot program with different values of n
for ((j=2; j<=10; j++)); do
    for (( i=0; i<=10000; i+=500 )); do
        output="output.txt"
        if [ ! -f "$output" ]; then
            touch "$output"
        fi
        echo "Running iteration $i"
        mpirun -np "$j" ./mandelbrot -n "$i" -a 3.5 >> "$output" 2>&1
        sleep 1
    done
done
```



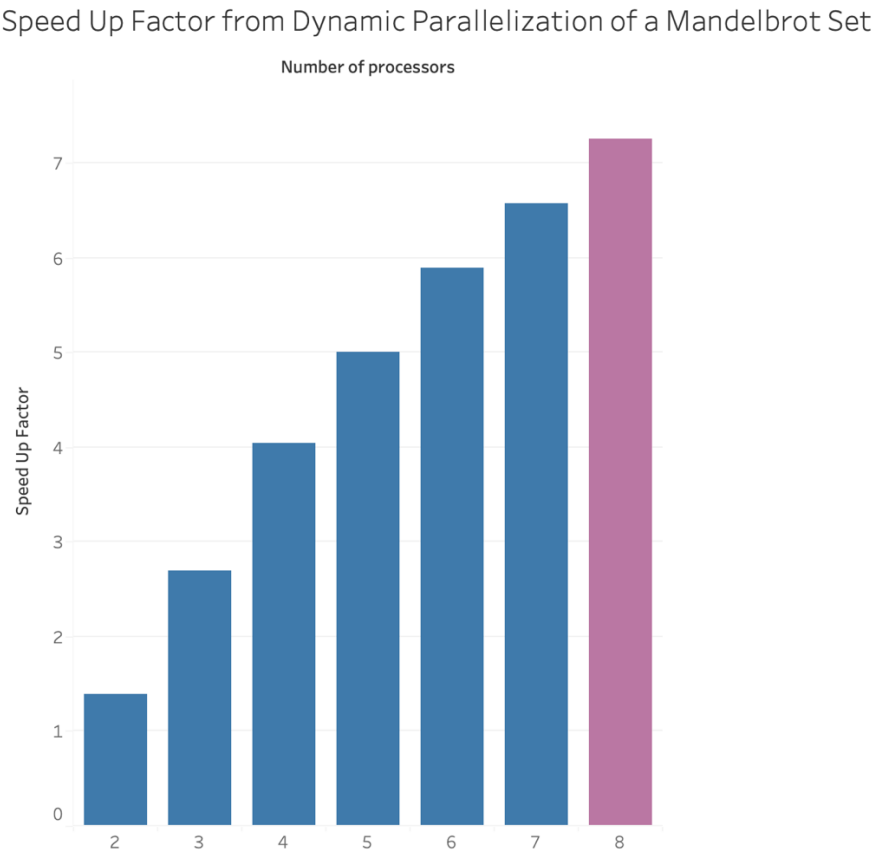


We notice almost equal time between the computation and communication models, this seems to be an optimal behavior as noted by Ismail 2010[1]. We notice interestingly how the computations increase with the iteration number as well as the communications with the number of processors.

Speed Up

By calculating the time to perform the Mandelbrot set in sequential, we can then calculate the speedup factor by the formula $S = T_{seq} / T_{parallelized}$. Thus, with $T_{average} = 3.7523739$, revise the trivial sequential code of finding the Mandelbrot set. Then, by using the speed up formula, we find the below speed up factors. The full datasheet is available on my personal repo linked above.

[1] Empirical Study for Communication Cost of Parallel Conjugate Gradient on a Star-Based Network
https://www.researchgate.net/figure/Communication-Time-and-Computation-Time-for-a-Load_fig5_232628317



It is important to note that increasing does not always yield (a constant) proportional boost. In my case, system call failures started to show up by np 6, and by np 8, shared memory allocation failed and no longer was it possible to take more measurements. *Note this is the average.*

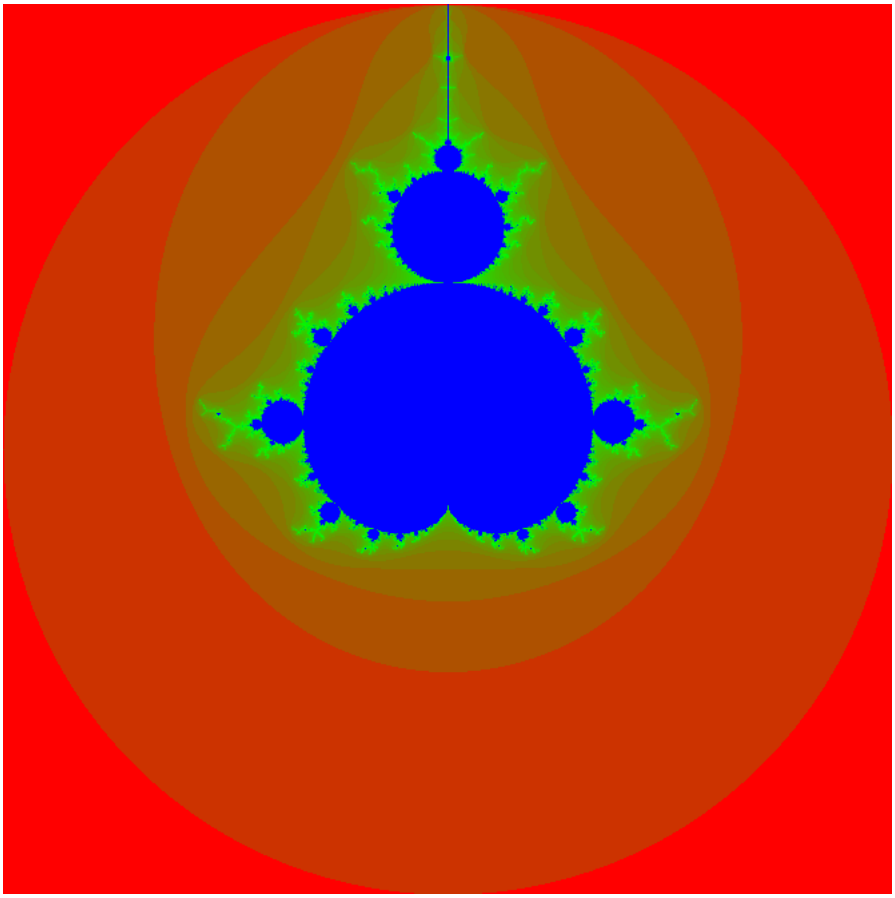
2.1.0 Static Code

In a static method, the pixels must be partitioned in a fixed way between processors. First, we must know "a-priori" information about the number of processors available. Then, we can divide the load in two extremes. For now, there are two methods for distributing pixels to processors: dividing chunks of pixels to processors based on the total number of pixels or distributing each pixel to a processor iteratively. The former approach can result in communication overhead at the end, while the latter approach involves continuous send/receive commands, which also incur in continuous communication overhead.

In the code provided, the Mandelbrot set computation is partitioned statically across the processors. This means that each process is assigned a fixed number of rows of the image to calculate (look at the first method), based on the total number of processes and the size of the image (defined by N and ncpu), Then, the Mandelbrot set is computed in a loop that

iterates over each pixel in the group assigned to the current process. The complex value corresponding to the pixel is computed and iterated until either the maximum number of iterations is reached or the iterate exceeds a certain threshold. The result is stored in the corresponding element of the array x. After the Mandelbrot set is calculated, synchronization occurs, the master calls a MPI_Gather, collecting the results. Then, an image is generated by iterating over each generated pixel in array y.

2.1.1 Static Screenshots and Methodology



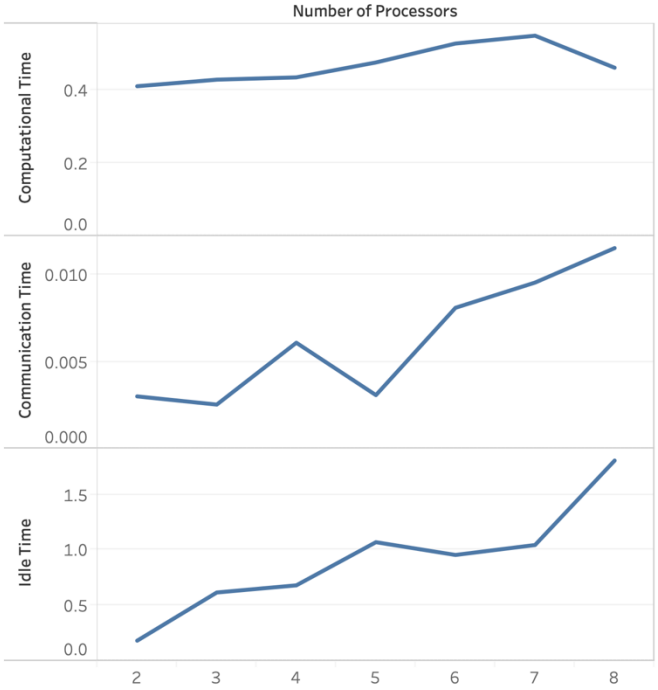
Briefly, we notice the produced image correctly detailing 666x666 image from a 255 iteration (Refer to GH). We perform this process on different processors to generate different computation and communication times which we will then benchmark against the dynamic code. Which we have performed the same steps on. (D=666x666, ITR=2555 N=2 to 8).

Results are shown below.

Osama Shamout
201906271

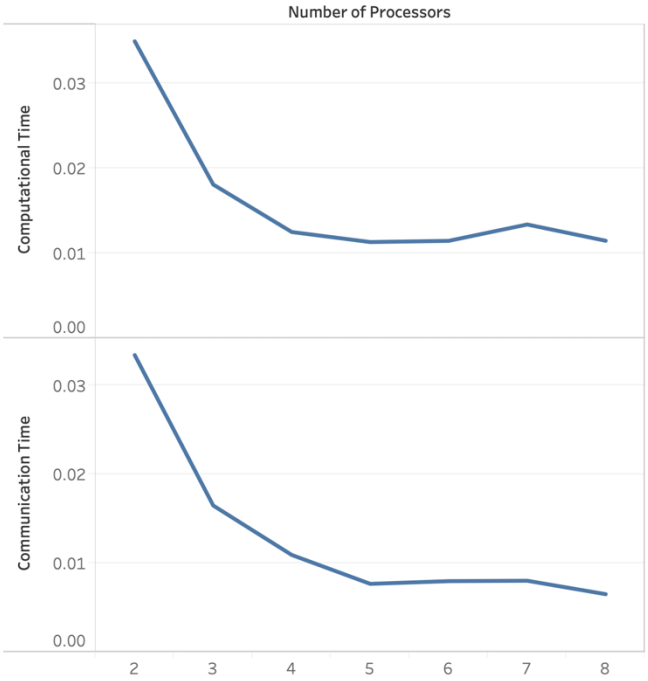
Dr. Hamdan Abdellaef
CSC447 – Parallel Prog.

Static Parallelization (MPI)
Number of Processors vs. Computational, Commmunication, & Idle Time



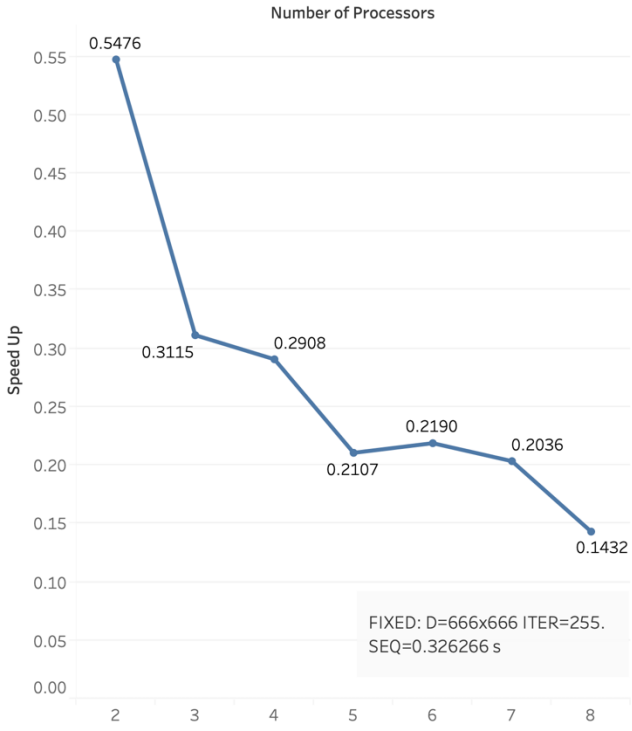
The trends of Computational Time, Communication Time and Idle Time for Number of Processors.

Dynamic Parallelization (MPI)
Number of Processors vs. Computational & Commmunication Time.



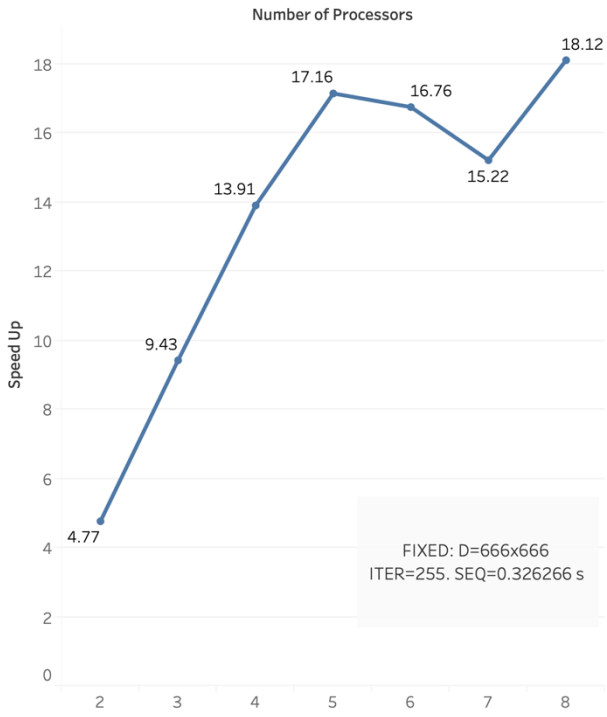
The trends of Computational Time and Communication Time for Number of Processors.

Static Speed Up Factor
S = Old Execution Time/New Execution Time



The trend of Speed Up for Number of Processors.

Dynamic Speed Up Factor
S = Old Execution Time/New Execution Time



The trend of Speed Up for Number of Processors.

Discussion

We notice that the static parallelization did not seem to work. By a close inspection to the output.txt file, we notice that:

$$\exists\ p_c \in \{p1, p2, p3, p4, p5, p6, p7, p8\}: \text{time}(p_c) \geq 0.11$$

Where time(p_c) represents the computational time of processor p.

∴ For any combination of processor p1 to p8, where p1 to p8 are distinct processors, there cannot be a speedup since there exists the condition above, since any additional computation speed on top of processors p_c, speed up fails. Though, I was not able to detect the reason. One theory is that all the computation keeps falling on one processor and the rest is only using minimal processing as the iterations are not very heavy, however, even as the iterations were increased the numbers did not change much.

Regarding the communication time, there did seem a connecting theme in communication time and the increasing number of processers as more message passing would be needed and synchronization is required. However, this is debunked as we see in the dynamic parallelization.

For the dynamic parallelization, the results were perfect of what we expect of parallelization, the speed up factors reached as high as 18x the sequential way. Jumping from 2 to 8 cores can yield 3x to 5x cuts in time.

Yet it is unknown here, if in fact, the parallelization was so good in fact, it handles synchronization that it did not create any bottlenecks and proceeded to maintain almost a balance of 1:1 computation to communication time. Assuming that the system also had low communication overhead and good load balancing, we can conclude that the system is scalable. This is because, with an increase in the problem size (iterations) and the addition of more CPUs, the system showed good performance and efficiency, even achieving speedup factors with dynamic parallelism, which indicates that the system can handle increasing workloads and scale well as more resources are added. However, the system did show weaknesses in resource retention with failed shared allocation as previously mentioned and shown in output texts.