Jake Wheeler

CS415

03/14/2017

PA2 – Mandelbrot Set

Project Description

The goal of this project was to compare the sequential implementation of the Mandelbrot set with the parallel implementation of it. This was accomplished by implementing the calculations in sequential.c and dynamic.c, running them with various image sizes and comparing the results of the timing. For the tests, the following sizes were used: 500x500, 1000x1000, 5000x5000, 10000x10000, and 25000x25000. The output images were the same between sequential and dynamic, but the elapsed time was smaller with the latter, as expected. This comparison can be seen in Figure 1, below.

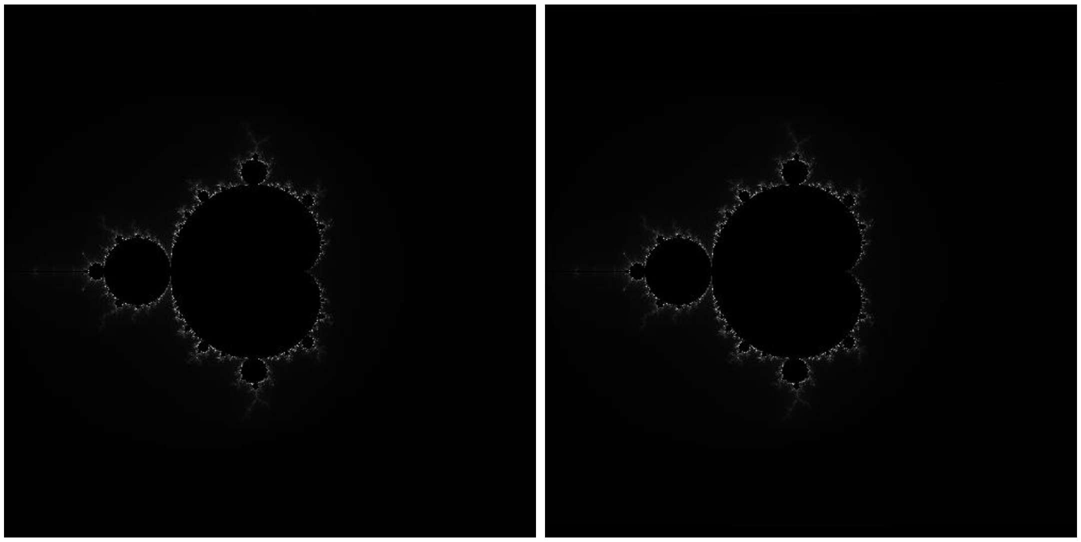

Figure 1, 500x500 images for sequential (left) and dynamic (right).

Sequential

The sequential implementation was straight-forward, because it was designed to be run with only one processor in mind. The program essentially only allocates space for each pixel of the Mandelbrot calculations, calculates each pixel one by one in a loop, and outputs to a file. This sequential design approach led to the time increase being linear, or exponential given the exponential increase in size. This can be seen in Figure 2, below. The tests were run four times to ensure consistent results, but the tests are very difficult to differentiate as the timing was almost exactly the same between tests.

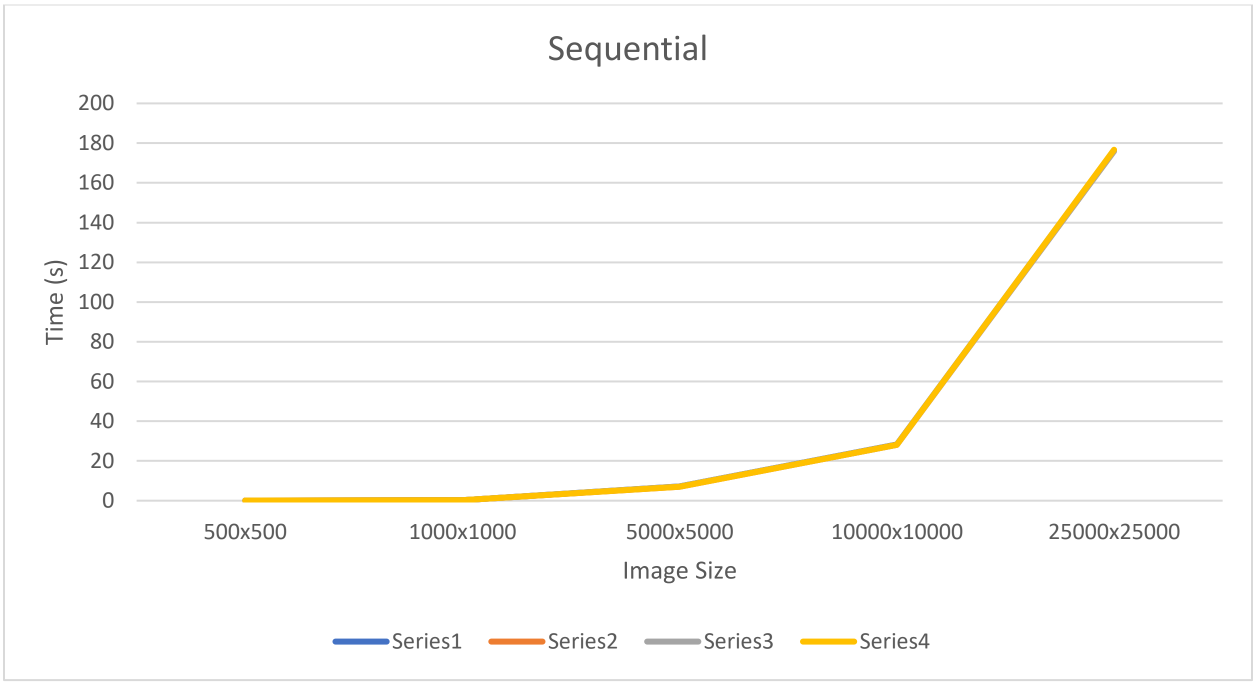

Figure 2, Image Size vs. Time graph for sequential implementation

Dynamic

The dynamic partitioning implementation was slightly more complex than the sequential, but not by much. It works via a basic master/slave structure. The master process spends its time assigning sub tasks to other slave processes and reconstructing the data collected from the slaves. The slaves only do calculations, calculating a set number of rows at a time, determined by the master, and send their finished product back to the master while awaiting a new set of rows to calculate. The major problem I ran into during this implementation was the orchestration of it all, the MPI_Send() and MPI_Recv() calls must be essentially in sync or else a deadlock can take place, which I ran into many times. The dynamic implementation had a significant decrease in time vs the sequential implementation, with the dynamic thriving more under larger image sizes. The graph representing the time when run on 8 cores can be seen below, in Figure 3.

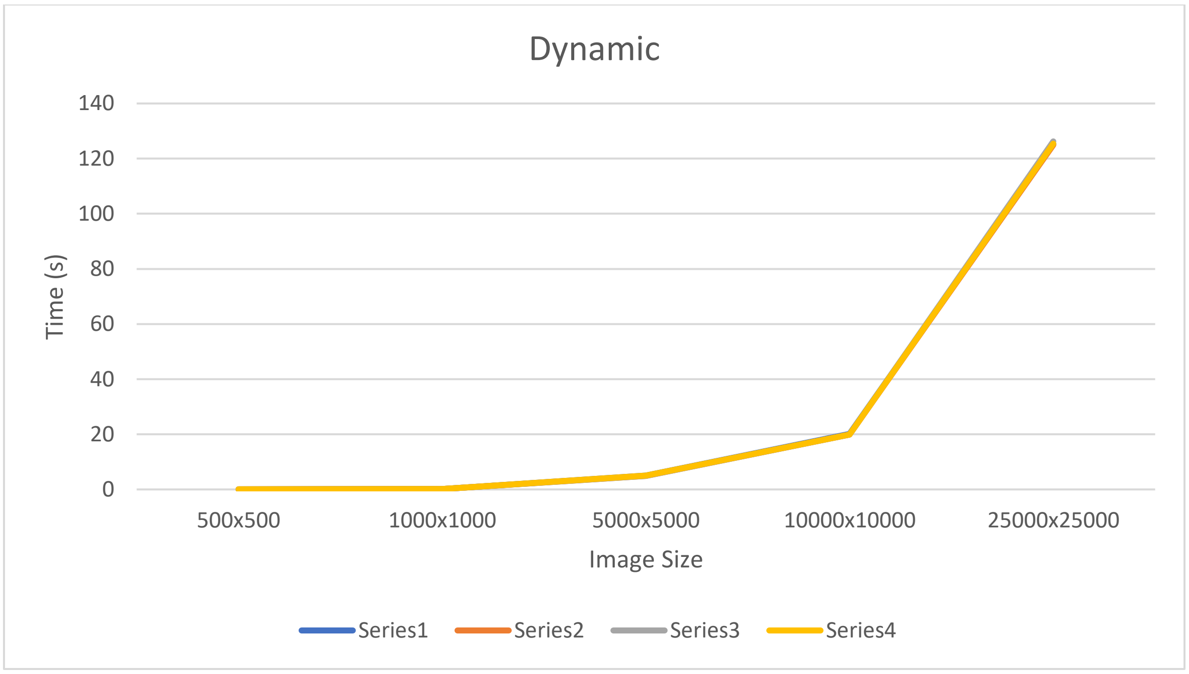

Figure 3, Image Size vs. Time graph for dynamic implementation when run with 8 cores.

Comparison

When comparing the two methods, sequential vs. dynamic, dynamic is faster than sequential in almost every condition. Sequential is faster when run on very small images, where the overhead of the communication outweighs the benefit of the parallelization of the dynamic implementation. The number of cores given to the dynamic program also impacts its performance significantly, as the tests with 4 and eight cores ran much faster than the sequential tests, which can be seen in Figure 4 below.

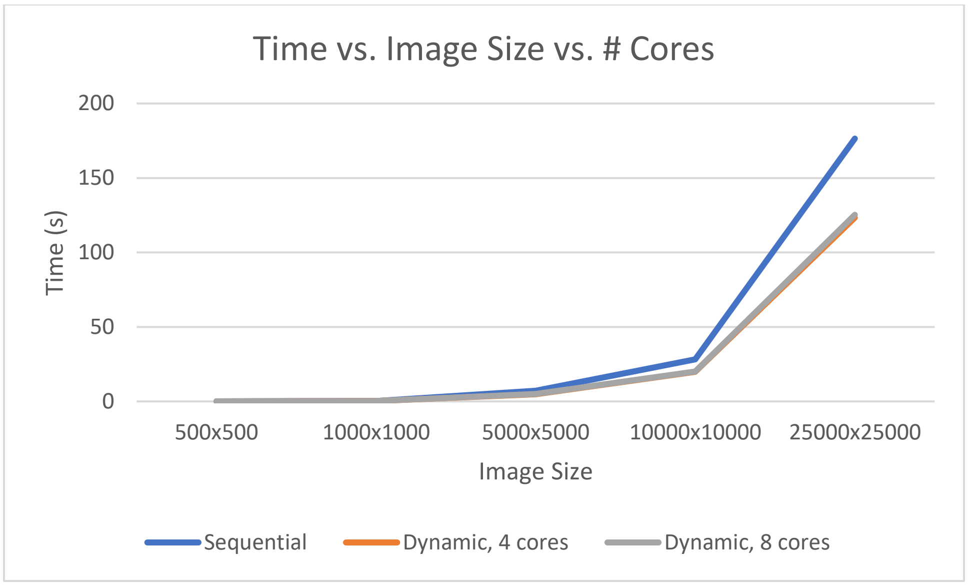

Figure 4, Time vs. Image Size vs. Number of cores (lower is better)

When looking at the speedup factor of the parallelized program it is easy to see that the tests with four and eight cores follow a similar pattern, with the eight core test being slightly higher than the four core, as expected. The graph showing Speedup vs. Size vs. Cores can be seen in Figure 5, below.

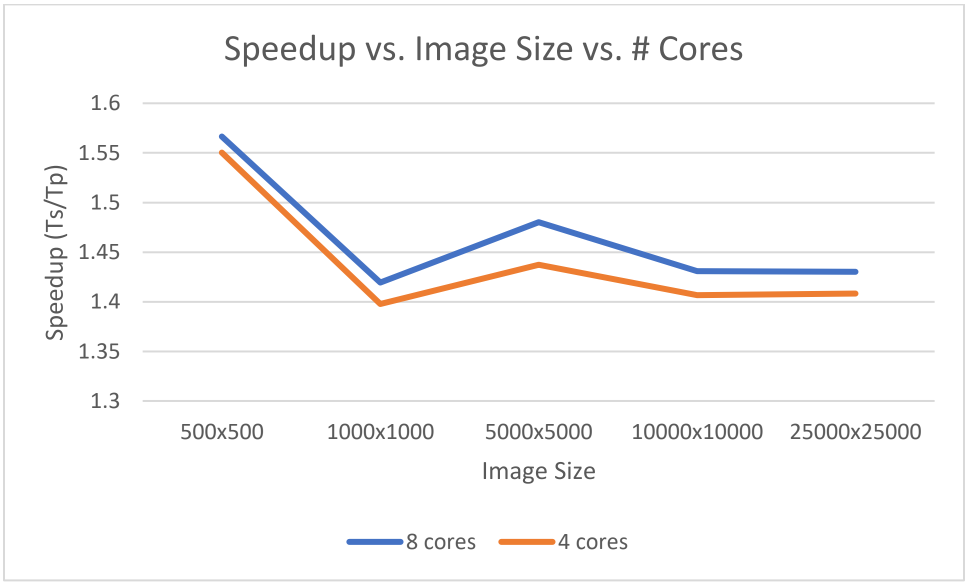

Figure 5, Speedup vs. Image Size vs. Number of Cores

Like the speedup, the efficiency given the number of cores follows a similar pattern between four and eight cores, with eight being consistently slightly higher than four, as seen in Figure 6.

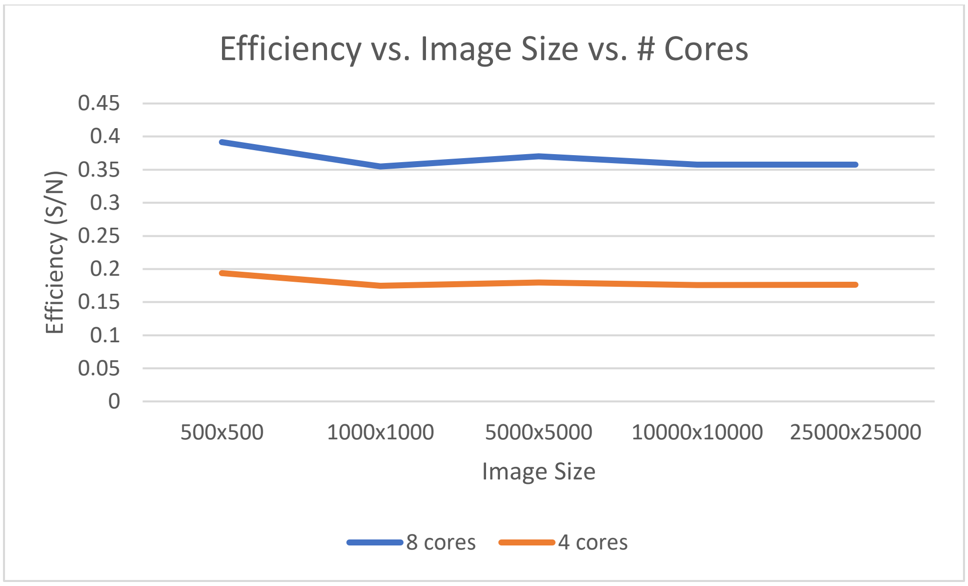

Figure 6, Efficiency vs. Image Size vs. Number of Cores