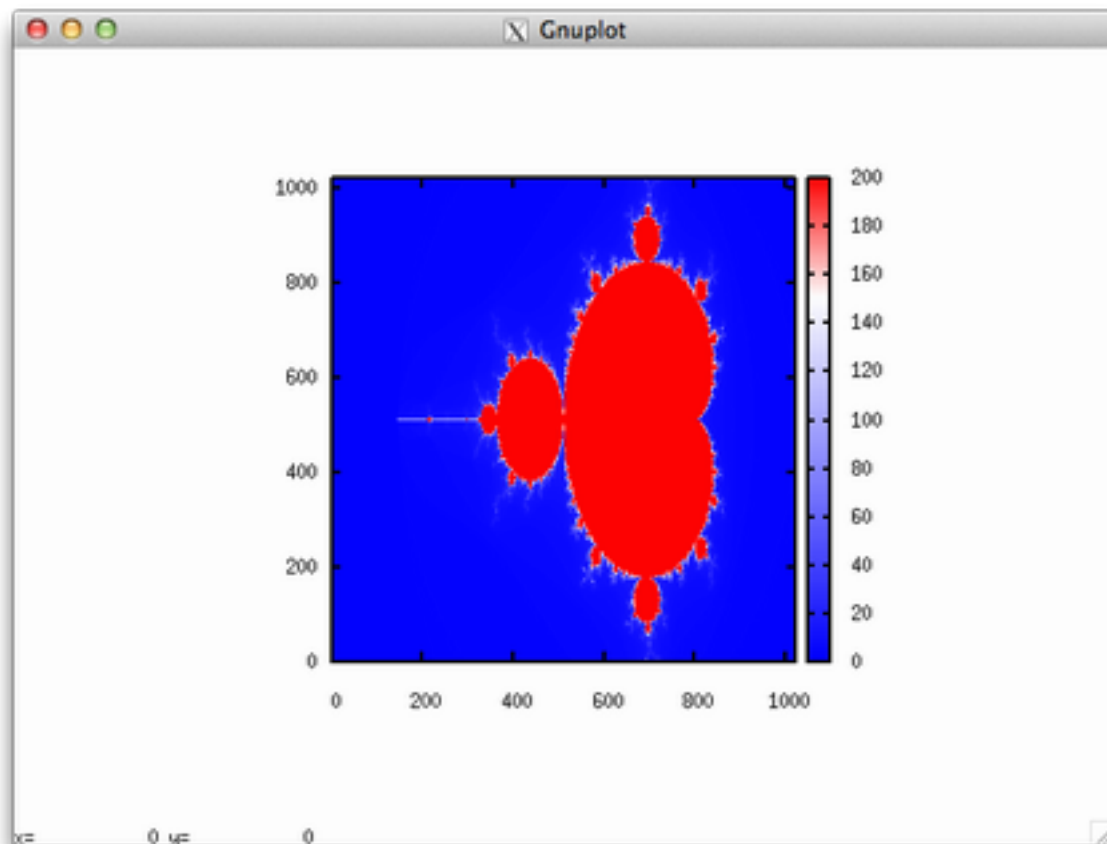


CSE160 Programming Assignment #1

For our program, we created a struct for each chunk that the threads have to work on. If the chunkSize was zero, it means we had to use block partitioning, so we simply divided the given array into rows evenly so that each thread would be given a more or less equal share of the work.

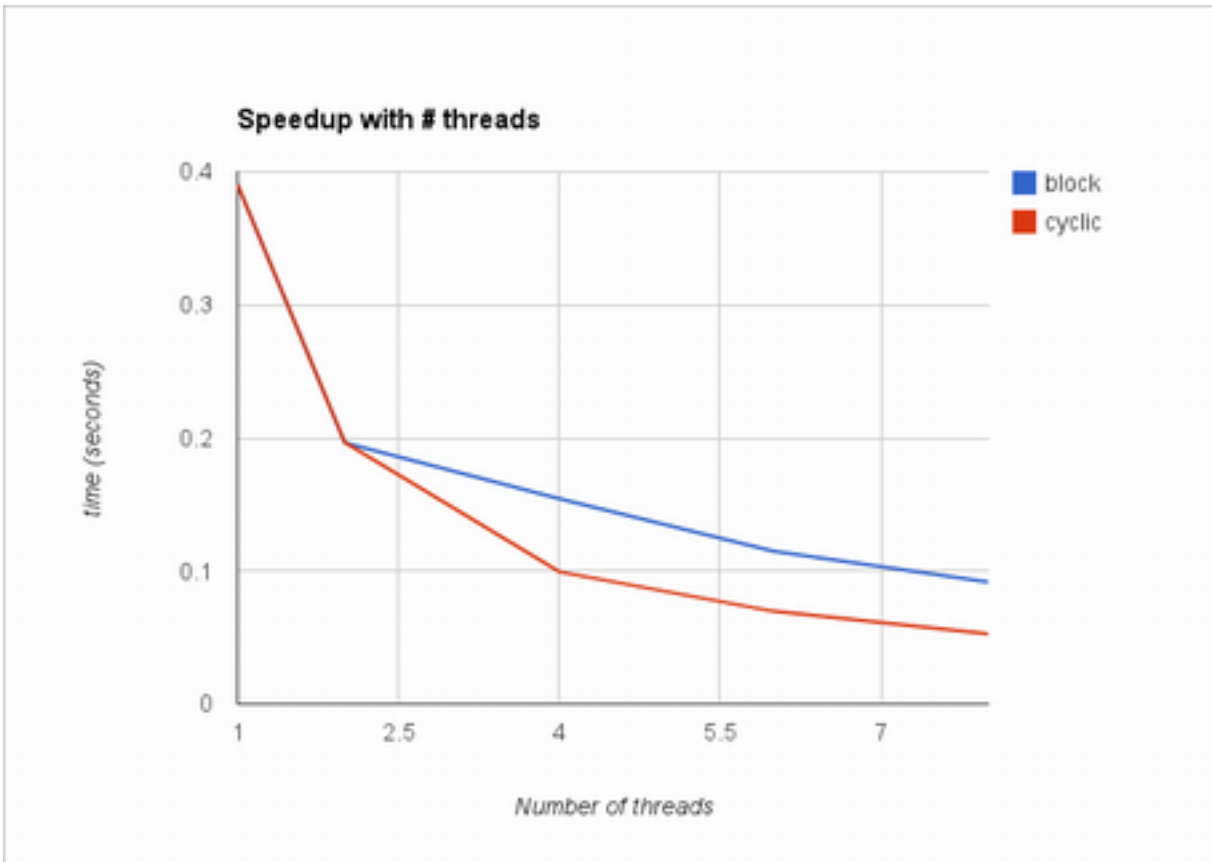
If chunkSize was not zero, it means we had to use cyclic partitioning, so we calculated the number of chunks that each thread would have to work on, and created an array for each with that size. We then calculated the start point of the chunks that each thread has to work on and inserted them into the array. Then, we iterated through the array and computed the data for each element. In the case that the number of chunks cannot be distributed evenly across the threads, we made sure to change the size of the array accordingly and give the beginning threads more chunks than the later ones. We also checked for the case where the chunkSize did not divide evenly into the number of rows. In that case, we marked the thread as “special” and checked to give the thread working on the last chunk the correct size of the chunk.

This is what our plot looks like:



It looks exactly the same for both block and cyclic partition, with 1 2 4 6 and 8 threads, so we decided not to include them all. They were created with default values: dimX and dimY at 1024, with 200 iterations, and with 1, 2, 4, 6, and 8 threads. When using cyclic partitioning, we chose chunkSizes of 2 and 32.

In our implementation, we definitely have a bottleneck before we start letting the threads compute and after, when we join them. Before the creation of the threads, we do a lot of calculations to figure out which parts each thread must work on. This is especially true when we use cyclic partitioning. At the end, when the threads join, there is also a bottleneck. However, as you can see below, we still see quite a good speedup when we use more threads.



The speedup is evidently much better with cyclic partitioning of the data instead of block partitioning, but they both look like they both get pretty good speedup as the number of threads increases. It looks like they are about to hit the point of how much time the overhead takes to run. As a comparison, the serial code took 0.390333 seconds, which is about the same as the time it takes for one thread to compute the whole plot. Since we were using 8 cores, the speedup is $0.390333/0.052762$ or 7.39799. This means that running the program in parallel is over 7x faster than running the program serially. Since we were at it, we also found out that the efficiency of each core is 0.9247, or around 92.5%, which is very good.