GENERATING THE MANDELBROT SET USING THE JUNGREIS FUNCTION

SAM KOTTLER

ABSTRACT. The Mandelbrot is typically drawn by looking at a recursive complex polynomial. However another way is with the Jungreis function. This function is not often used because it takes much long to converge. However, the coefficients of the Jungreis function can be used to precisely calculate the area of the Mandelbrot set. Here I used the Jungreis function to generate the Mandelbrot set. I parallelized the program in eight different configurations to try to increase performance.

1. INTRODUCTION

The Mandelbrot set is defined as the set of points where the polynomial

$$z_{n+1} = z_n^2 + c$$

where $z_0 = 0$ and $c \in \mathbb{C}$ stays finite as n goes to ∞. This is usually drawn by testing if some bound has been exceeded after some maximum number of iterations for every point in the image. This process can only be used to generate the whole interior of the Mandelbrot set. There are a few ways of generating only the boundary of the Mandelbrot set including the Jungreis function.

For this project, I wrote a program that draws the boundary of the Mandelbrot set using the Jungreis function. This function maps the unit circle in the complex plane to the boundary of the Mandelbrot set. The function is given by

$$\Psi(z) = z + \sum_{n=0}^{\infty} b_n z^{-n}$$

where b_n are coefficients of the Laurent series about infinity. These can be found recursively with the following rules:

$$b_n = \begin{cases} -\frac{1}{2} & \text{for } n = 0 \\ -\frac{w_{n,n+1}}{n} & \text{otherwise} \end{cases}$$

$$w_{n,m} = \begin{cases} 0 & \text{for } n = 0 \\ a_{m-1} + w_{n-1,m} + \sum_{j=0}^{m-2} a_j w_{n-1,m-j-1} & \text{otherwise} \end{cases}$$

$$a_j = u_{0,j+1}$$

$$u_{n,k} = \begin{cases} 1 & \text{for } 2^n - 1 = k \\ \sum_{j=0}^{k} u_{n-1,j} u_{n-1,k-j} & \text{for } 2^n - 1 > k \\ 0 & \text{for } 2^{n+1} - 1 > k \\ \frac{1}{2}(u_{n+1,k} - \sum_{j=1}^{k-1} u_{n,j} u_{n,k-j}) & \text{otherwise} \end{cases}$$

There is no known closed form formula for these coefficients. This is not the fastest method for finding the boundary of the Mandelbrot set because the series takes a lot of terms to start to converge. However, it is still interesting and worth

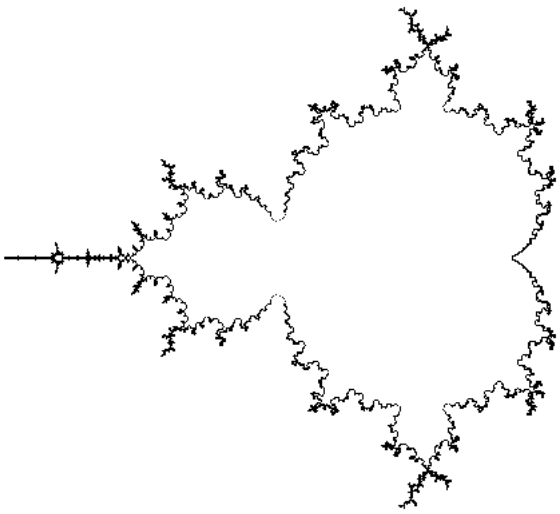


FIGURE 1. An example of the image I want to generate

calculating because the area of the Mandelbrot set is given by

$$A = \pi(1 - \sum_{n=1}^{\infty} n b_n^2)$$

with the same b_n as above.

2. Design

My program outputs a png image of the Boundary every time it runs. This makes it easy to compare the output of the parallel code to the serial code to ensure the parallel versions are calculating the correct numbers. It also made it easier when writing the original serial code to check that I had implemented the algorithm correctly.

I started by implementing the recursion above. However, I discovered that with these recursive rules, there was too much rounding error after only about 50 coefficients so I implemented the following recursive rule to generate the same coefficients.

$$b_n = \beta_{0,n+1}$$

where

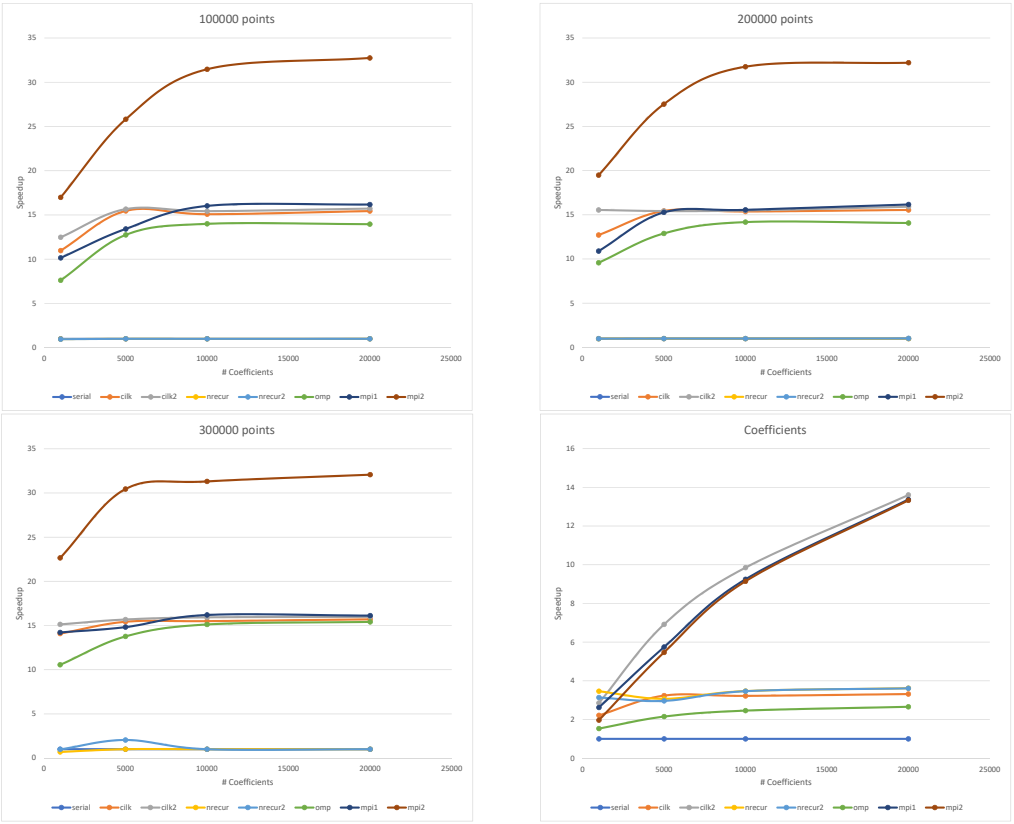
$$\beta_{n,m} = \begin{cases} 1 & \text{for } m = 0 \\ 0 & \text{for } m < 2^{n+1} - 1 \\ \frac{1}{2}(\beta_{n+1,m} - \sum_{k=2^{n+1}-1}^{m-2^{n+1}-1} \beta_{n,k}\beta_{n,m-k} - \beta_{0,m-2^{n+1}-1}) & \text{otherwise} \end{cases}$$

Finally, by using a lookup table, I was able to compute b_n in a loop rather than recursively by figuring out how many dependencies each β has.

This algorithm has two different computationally difficult sections. First is actually calculating the coefficients. The amount of work for this part only depends on how many coefficients I want to calculate. The other part is actually mapping points of the unit circle to points on the boundary of the Mandelbrot set. The work for this part is dependent on both the number of sample points on the unit circle and the number of coefficients calculated. To compare performance of different programs and different problem sizes I timed each section individually.

I ended up having seven different programs. Some of them used the same or similar algorithms for one part but each of the seven is different is some way. First I have a serial implementation which uses recursion for the calculating the coefficients. This is the simplest and is used as a baseline for comparison with the others. Next I made parallel versions using openmp and cilk. The second part is easy to parallelize because there are no data dependencies, so I can run any points at the same time. Both of these made a small attempt to parallelize the recursive version of the coefficient calculation. However, this caused each thread to recalculate some of the β values. Next I made two serial versions using a loop instead of recursion for the coefficient generation. These are almost identical except that the second one is able to be easily parallelized because all β with the same number of dependencies are calculated at the same time. Next I used cilk to parallelize the nonrecursive coefficient version. This used the same parallelization for the point mapping as the first cilk version. Last I made a hybrid MPI and cilk version. This was very similar to second cilk version except it is able to run on multiple compute nodes. Unfortunately the network configuration on mcscn causes a crash when I try to run on more than two nodes. Thus I have eight run configurations: serial, cilk, cilk2, nonrecursive, nonrecursive2, openmp, mpi1, and mpi2.

3. Performance Analysis



To compare performance I timed the two sections individually. I ran each of the eight configurations three times each with 1000, 5000, 10000, and 20000 coefficients and 100000, 200000, and 300000 points. Because I was only timing the parallelizable regions of code I expected to get close to 32 times speedup on one node and 64 times on two nodes. The overhead to set up parallelism is in

the timed section also, so speedup should be a little less than the maximum. The overhead should have less of an effect for larger problem sizes.

Above I have plotted the averages of the three runs on each configuration and each problem size. As expected, the speedup of the parallel code increased as the problem size increased. Also as expected, the MPI version running on two nodes is about twice as fast for point mapping as the versions running on one node. However, each configuration only gets to be about half as fast as expected. One possible explanation for this could be that while each node has 32 logical processors there are only 16 physical processors. I didn't run any of the configurations with a high enough number of coefficients for the curve to start leveling off, but we might expect the same limit of 16 times speedup.

There are some other surprising and interesting things to notice on the plots. First, for coefficient generation, the two nonrecursive serial versions were consistently about the same as the recursive cilk version and better than the recursive openmp version. This is likely just a coincidence. Another interesting thing is that both of the nonrecursive versions were slightly less efficient at coefficient generation for 5000 coefficients. One possible explanation is that there is slightly more loop overhead compared to the amount of time in the loop for 5000 coefficients. It also could be a fluke.

4. Conclusion

The main follow-up I would like to do with this project is get more data with more problem sizes and more runs for each. This would hopefully help me resolve some of the questions I had above. Specifically the questions I'm interested in are why the nonrecursive performance dipped at 5000 and why the performance levels off at about half the speedup I would have expected. I would possibly need to use a profiler like tau to figure out the second question.