# PA2 – Mandelbrot Documentation

Brian Conway

27 Mar. 2017

CS415

Mandelbrot Set

The Mandelbrot Set is a specific set of complex numbers that won't exceed some limit when iterating over the function:

$$z_{k+1} = z_k^2 + c$$

where z is initially 0 and c represents the complex number. We can get a graphical representation of the set by letting c represent a point on the complex plane, and base the intensity of the pixel at that point with the number of times it took the function to reach some limit.

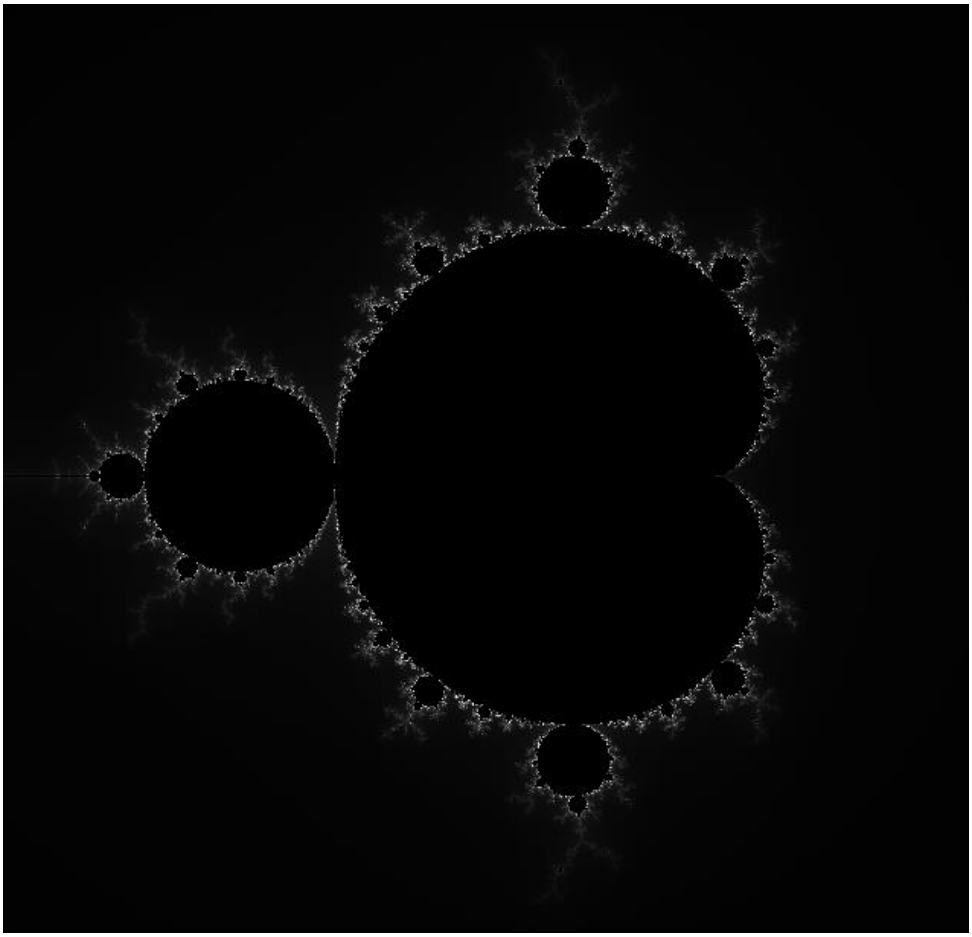

Fig. 1: Graphic representation of the Mandelbrot set calculated sequentially

# Methodology

The file sequential.cpp uses nested for loops to calculate colors of pixels in an image based on the Mandelbrot set 10 times, timing the entire time it takes to calculate the pixel colors. It divides the total time taken for all calculation by 10 to get the time taken for a single calculation of the Mandelbrot set. The dimensions of the image and the overall measurements taken can be adjusted by changing the two constants at the beginning of sequential.cpp.

After the 10 readings are taken, the calcStatistics function finds both the average and the standard deviation of the pixel color calculation times. It outputs this information to the console and then outputs the statistics to a file called “measurements.csv”. The first line of the output file is the average, the second is the standard deviation. The image is output to the file “imageSeq.pim”

The file static.cpp uses static task assignment to calculate colors of pixels in an image based on the Mandelbrot set 10 times, timing the entire time it takes to calculate the pixel colors. It has the master node divide all the rows in the image into sets based off of the amount of tasks, then sends out the row number of the first row to calculate to each corresponding task.

The master node splits the amount of rows that get sent back at once in half repeatedly as long as the amount of pixel colors in one message exceeds 2 million, counting each time the row count was split in half.

The other tasks receive their row numbers and calculate the pixels for that row and the subsequent rows based on how the rows were initially allocated. They calculate enough rows to send back that don’t exceed 2 million pixels and send the calculations to the master as a 1D array, which the master receives and stores it in the corresponding locations in its 2D array of pixel colors. It divides the total time taken for all calculations by 10 to get the time taken for a single calculation of the Mandelbrot set. The dimensions of the image and the overall measurements taken can be adjusted by changing the two constants at the beginning of static.cpp. The image is output to the file “imageStat.pim”

Timing is done using the custom Timer class, which has start, stop, and resume functions similar to a stopwatch. Timer makes use of timevals and the gettimeofday() function in order to get an accurate reading of seconds and milliseconds at certain instants. When the elapsed time is to be given, the Timer takes the difference in seconds and milliseconds between when the timer was started and when the timer was stopped. It then combines the seconds and milliseconds reading into a double precision floating point variable.

Results

Below is a surface graph of the amount of processes, the time taken to calculate all the pixel colors, and the image resolutions tested. The image sizes ranged from 1000x1000 to 40000x40000. In general, as the size of the image increased so did the total calculation time. As the number of processors increased, the execution time generally decreased.

The graph is slightly messed up where the processors axis and the resolution axis meet. That point on the graph represents 33 processors used and a 1000x1000 image resolution. Table 1 at the end of this document has all of the timings.

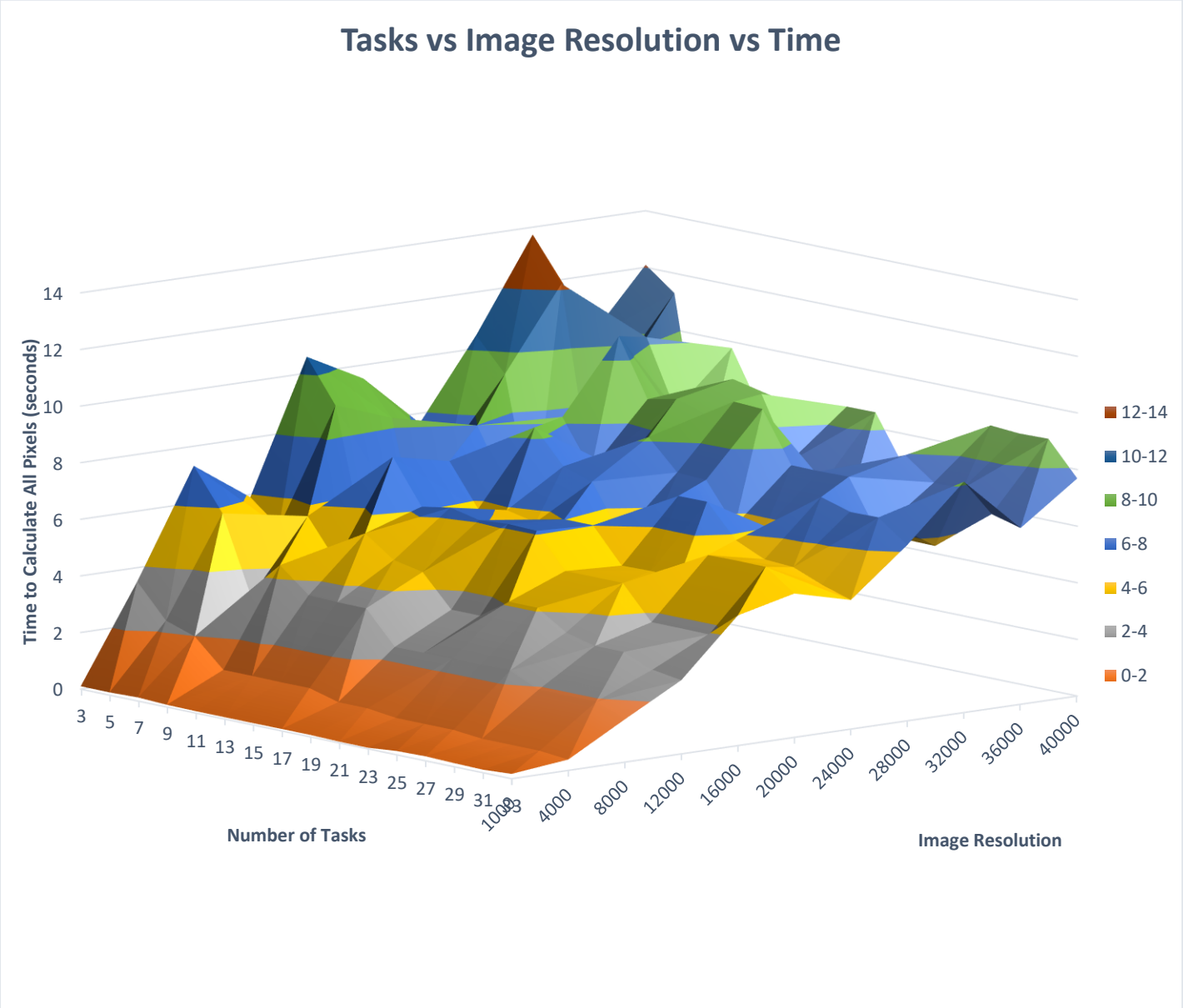


Fig. 2: Graph of runtimes with different amounts of tasks and different image sizes.

Below is the graph of timings for the sequential implementation, ranging from .2 seconds to 5 minutes with images from 1000x1000 to 40000x40000.

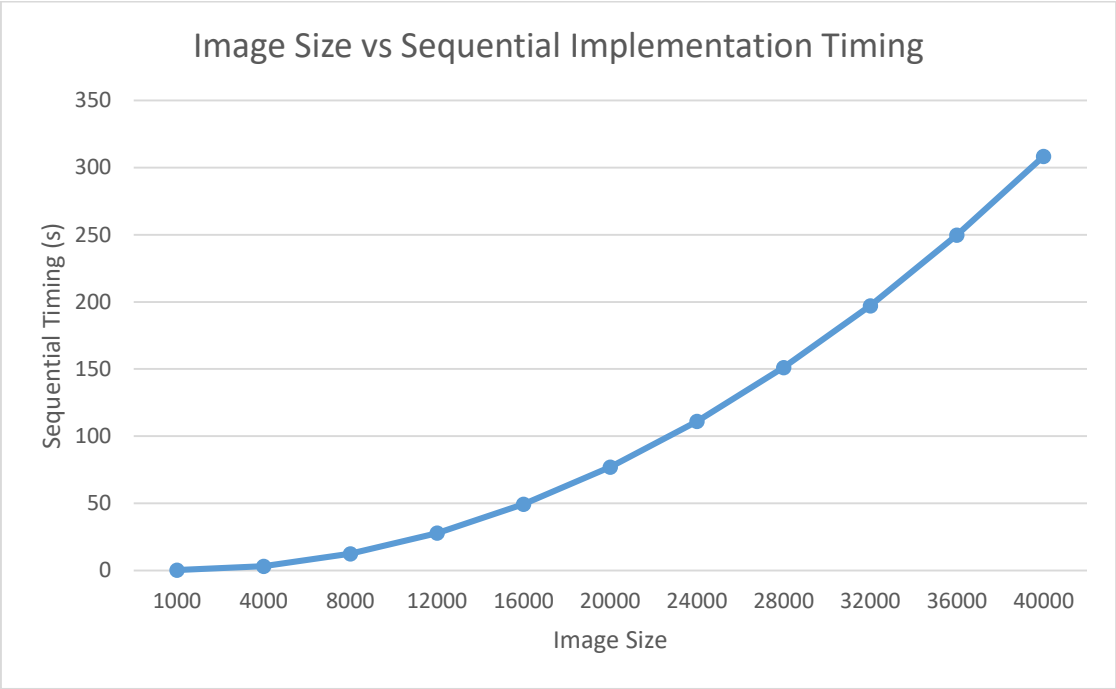


Fig. 2: Timings of the sequential implementation increased as the image size increased.

Speedup of the parallel implementation was calculated by dividing the sequential implementation timing by the parallel implementation timing. My results were very unusual, as I expected to have, at most, linear speedup. However, for image sizes above 12000x12000, I ended up having superlinear speedup, which got larger as the image sizes increased. As I don't think it has much to do with special hardware or special cases of image sizes, I attribute this superlinear speedup to the fact that my sequential implementation was most likely suboptimal, although I can't pinpoint specifically how so.

Another thing I noted was that between 11 and 25 processes there were dips in the speedup graph as the amount of overhead caused execution time to increase, this was especially evident with the larger image sizes.

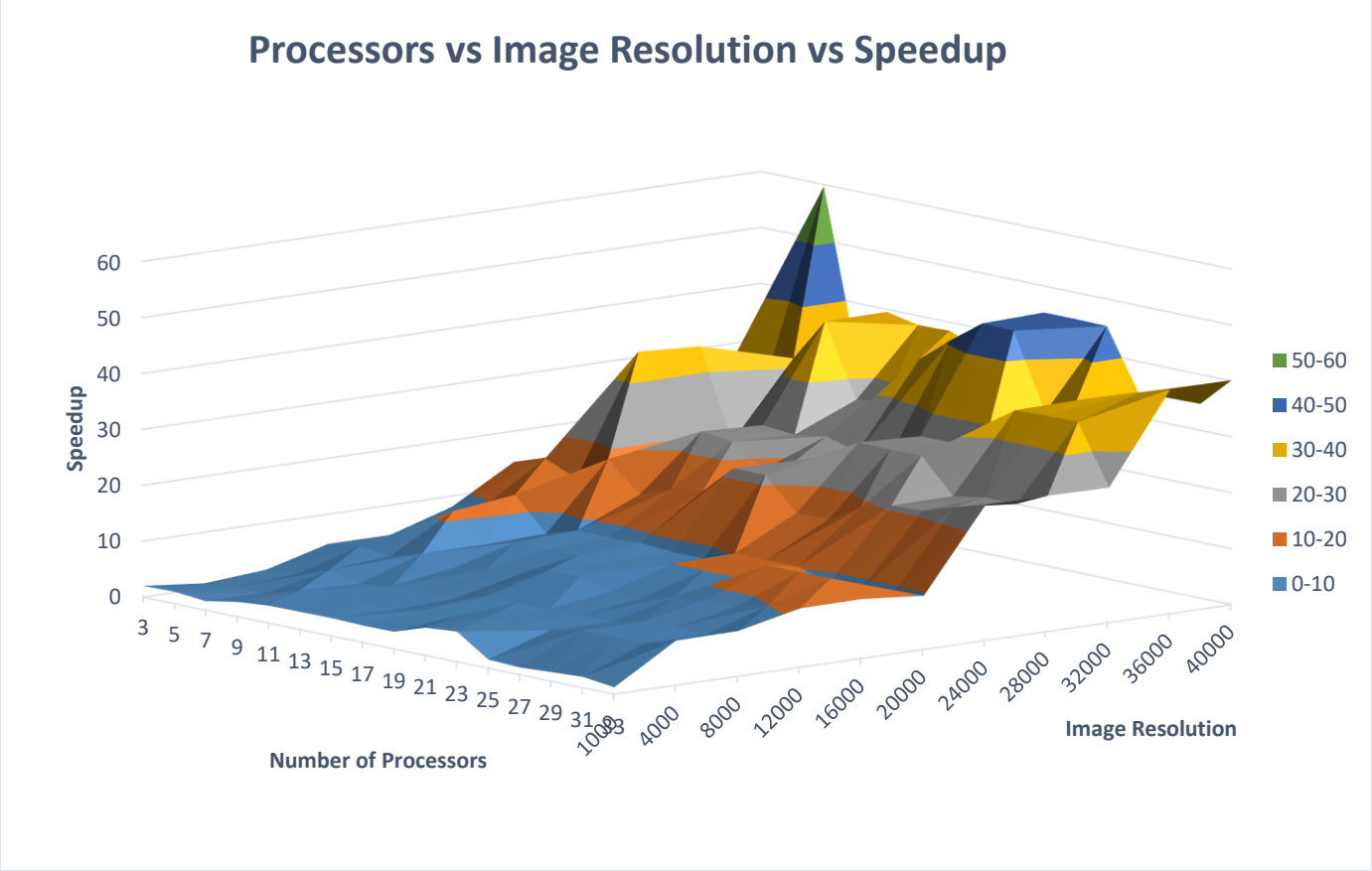


Fig. 3: Graph of speedup that shows unusual superlinear speedup beyond images of size 12000x12000.

Efficiency was calculated by dividing the speedup by the amount of processes used, and it indicates the percentage of utilization of each processor. As I had abnormal results with unusual speedup, I also abnormally had efficiency well over 100%, which shouldn't be possible. Despite this, it is still useful to analyze the overall shape of the graph. The shape indicates that the efficiency tended to decrease as more processes were added, though it decreased less as the image sizes increased.

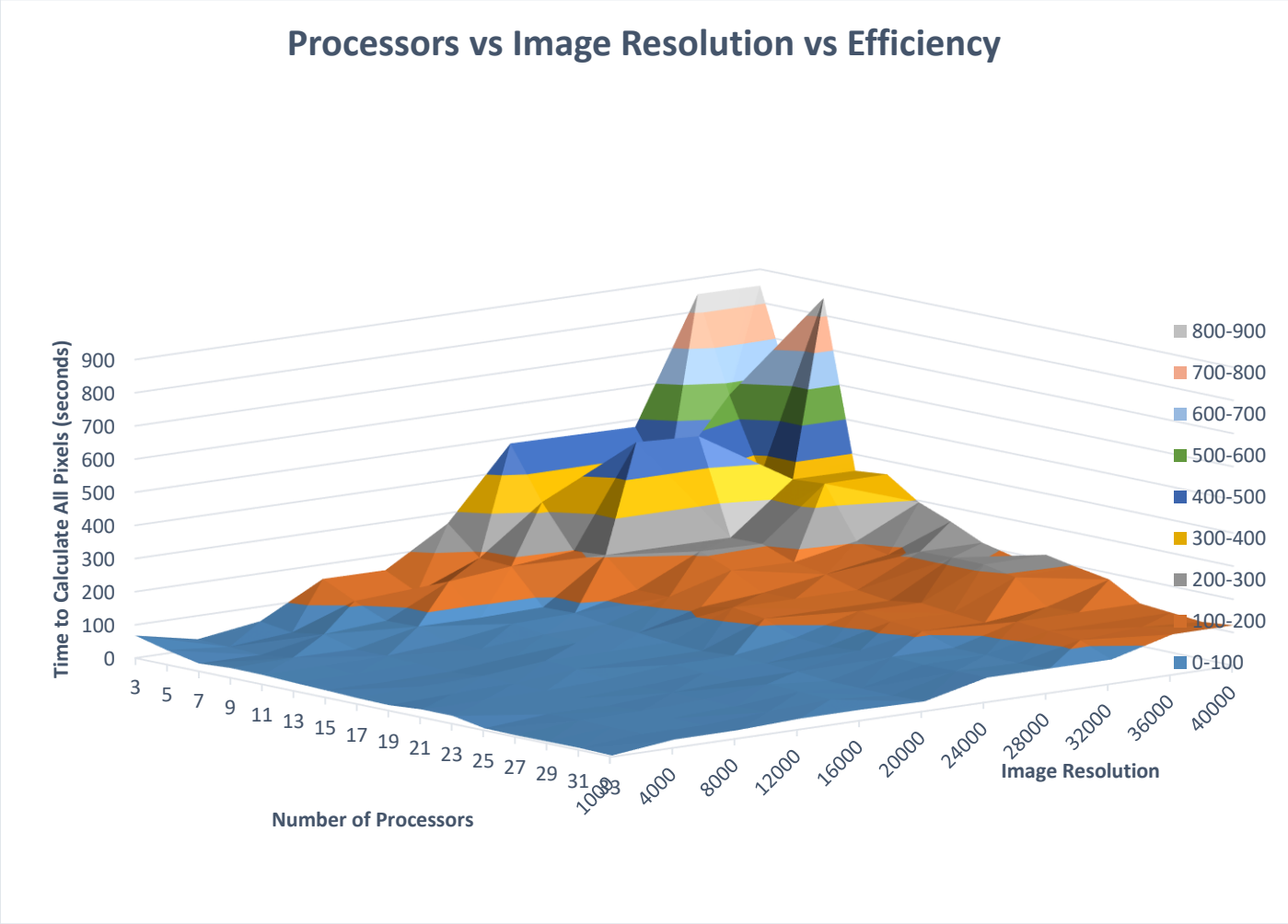


Fig. 3: Graph of Efficiency which shows processor utilization, with abnormal results over 100% but seemingly correct values relative to each other.

Measurements

| | 1000 | 4000 | 8000 | 12000 | 16000 | 20000 | 24000 | 28000 | 32000 | 36000 | 40000 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sequential** | 0.204656 | 3.07554 | 12.3163 | 27.7116 | 49.2843 | 76.9981 | 110.923 | 151.099 | 197.143 | 249.784 | 308.401 |
| **3** | 0.100883 | 3.50963 | 7.30591 | 5.90836 | 10.578 | 9.5223 | 7.65569 | 10.4764 | 13.7404 | 9.77542 | 12.0928 |
| **5** | 0.094027 | 2.341 | 5.83922 | 5.06701 | 9.06932 | 8.33399 | 6.80666 | 9.31076 | 12.2069 | 8.79989 | 11.3013 |
| **7** | 0.125331 | 1.99417 | 3.11341 | 5.65981 | 4.37 | 3.92493 | 7.99378 | 4.32528 | 5.75195 | 10.2378 | 5.16962 |
| **9** | 0.077147 | 1.01914 | 4.42634 | 4.2749 | 7.66898 | 7.23439 | 6.00353 | 8.21122 | 10.776 | 7.88386 | 9.77737 |
| **11** | 0.064736 | 1.01914 | 3.57062 | 5.47158 | 6.12216 | 5.75322 | 8.34423 | 6.56557 | 8.75886 | 6.33331 | 7.79948 |
| **13** | 0.063863 | 1.00606 | 3.5367 | 4.74075 | 6.30066 | 5.92065 | 7.1751 | 6.76406 | 8.97505 | 9.39063 | 8.04216 |
| **15** | 0.063033 | 1.00946 | 3.53166 | 4.52808 | 6.06549 | 5.69814 | 6.84948 | 8.13248 | 8.63504 | 9.07033 | 8.01471 |
| **17** | 0.066755 | 0.721417 | 2.37873 | 3.4073 | 6.14322 | 6.01685 | 5.1462 | 7.09254 | 9.30217 | 6.90333 | 8.54395 |
| **19** | 0.065971 | 0.71833 | 2.38273 | 3.40629 | 6.10796 | 6.01529 | 5.1749 | 7.04065 | 9.24908 | 6.13023 | 8.55445 |
| **21** | 0.040953 | 0.588942 | 2.39546 | 4.10367 | 4.89904 | 4.80013 | 6.96938 | 5.62415 | 7.47572 | 5.56133 | 6.81473 |
| **23** | 0.037378 | 0.58695 | 2.37938 | 4.10512 | 4.89969 | 4.82455 | 6.97889 | 5.65758 | 7.4658 | 5.57771 | 6.81619 |
| **25** | 0.131542 | 0.58779 | 2.38413 | 3.44433 | 4.89888 | 4.79854 | 5.82714 | 5.62727 | 7.48571 | 7.95893 | 6.83025 |
| **27** | 0.155774 | 0.485236 | 1.91937 | 3.17755 | 3.72498 | 5.80287 | 5.36409 | 7.27088 | 5.71502 | 7.20397 | 8.92813 |
| **29** | 0.124959 | 0.483146 | 1.91667 | 3.17708 | 3.72408 | 5.81647 | 5.29778 | 6.95533 | 5.68317 | 7.20295 | 8.85254 |
| **31** | 0.105134 | 0.482727 | 1.91789 | 2.76127 | 3.71395 | 5.80787 | 4.88386 | 6.97222 | 5.67184 | 6.50616 | 8.87683 |
| **33** | 0.167162 | 0.380957 | 1.5058 | 2.59905 | 4.59376 | 5.07578 | 4.55648 | 6.21049 | 8.14965 | 6.22429 | 7.68544 |

**Table 1: Timings of Mandelbrot calculations, with the top row being the image sizes and the leftmost column indicating the amount of processes.**