Parallelizing the Mandelbrot Set

By: Stefan Klaassen

Table Of contents

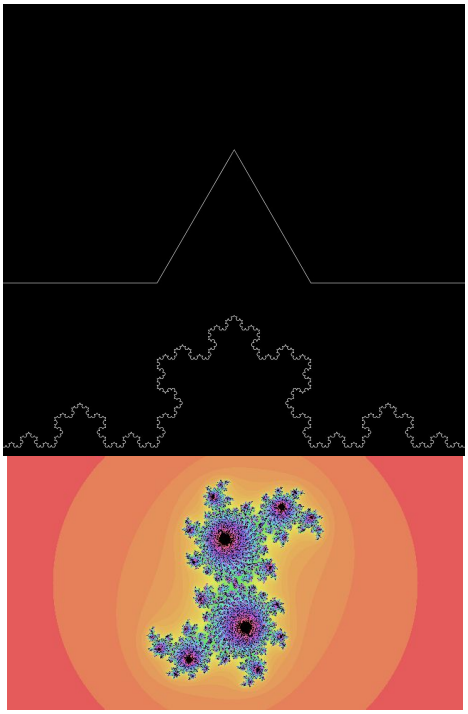
Introduction

Improving programs through the use of parallel computing is very important in today's world. Some problems are easier than others to parallelize. One such embarrassingly parallel problem is the mandelbrot set. It is considered embarrassingly parallel because each component can be computed independently from each other which makes its parallelization simple. Even though it is simple, one must still be careful as it is not necessarily completely embarrassingly parallel. Before we see why, we must first understand the different aspects of the mandelbrot set. The most important aspects of the mandelbrot set are its fractal nature, its calculation using complex numbers and both the generating of the of the image file and how to visualise it in a pleasing way.

Fractals

The first aspect of the mandelbrot set is that it is a fractal. Fractals are a term in mathematics that means that its dimensions are represented by a real number rather than an integer. Common integer dimensional objects would be a line for 1 dimension, a flat plane for two dimensions and a cube for three dimensions. The reason that their dimensions are in real numbers is because it usually has a topological dimension and an added fractal dimension. An example that can help explain this are waves on the surface of a body of water. The surface can be described as a plane which has a 2D topological dimension. The fractal dimension in this example would be used to describe how large and choppy the waves are.

Some other attribute that tend to be associated with fractals are their self similarity. This means that structures that are found in the fractal are made up of smaller versions of themselves. A good example of this would be the famous Von Koch snowflake fractal. The fractal is generated by starting off with a line. Then the middle third is split into two lines of equal length and form a triangular bump. Then to iterate, all straight lines are applied with the same permutation. The term self similar does work well for many fractals but it does not apply to all of them. For example the wave example is not necessarily self similar but still is a fractal because of its fractal dimension.



One last fractal that was important for the Mandelbrot set are all the Julia set fractals. They were the fractal that led Benoit Mandelbrot to find interest in this field and make his own set from Julia's fractals.

Complex Numbers

The patterns that emerges from the mandel brought set comes from the pattern that emerges from iterating an equation that uses complex numbers. These “Special” numbers are made up of two parts. These two parts are the real and imaginary numbers. It can be represented as $Z = a + bi$. both a and b are any real numbers and the $i = \sqrt{-1}$. Since i can not be explained by the real number set, it is considered to be imaginary. Also in the equation b is multiplied by i, this means that the whole imaginary part of a complex number is the value of b. A simple way to look at these numbers is that they are a 2 dimensional. This means that they can be plotted on a graph. The values of a and b are their projection on their respective axis. This also means that the number is also a vector.

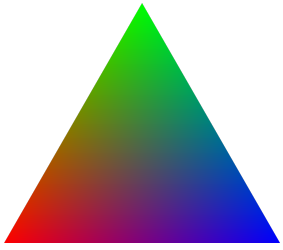
The calculation of the Mandelbrot set uses two calculations using complex numbers. The first is the addition of two numbers $Z_1 + Z_2 = (a_1 + a_2) + (b_1 + b_2)i$. The second and more complicated calculation is the power of 2 of the number $Z^2 = (a^2 - b^2) + 2abi$. As the power function is quite computationally intensive, one simple optimization is to simplify this equation by multiplying the values by themselves to $Z^2 = (a * a - b * b) + 2abi$.

Image File

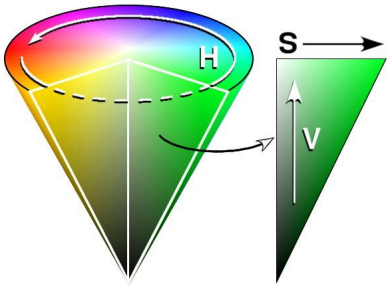
The generation of the image file is important to the parallelization and so for that reason must be addressed. The first step is to pick an image format that is lossless, pixel based, easy to make and can handle multiple color channels. For this project the .ppm file format was chosen. There are two versions of this file format. These two versions are the ASCII (P3) and byte (P6) representations. The byte format is faster and is more efficient on space and is therefore much better. The file consists of a header section and the data section. The header holds the version P3 or P6, the x and y size of the image in pixels and the range of colors for a single color. Normally the range of pixels has a max of 255. The data section is made in such a way that it is done row by row and the pixels consists of 3 channels. The three color channels are red, green and blue in that order. This color model is also known as RGB.

Color model

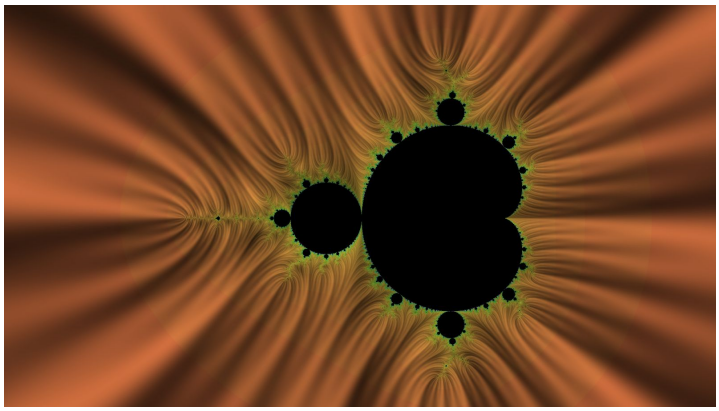
In coloring the Mandelbrot set, it is nice to have a wide range of color. The wide range of colors make the image representation of it more appealing and easier to spot interesting patterns. The simple answer would be to use the RGB model because it is the format that the image file needs. The problem with this type of model is that the part of the mandelbrot that is represented by colors are the number of iterations. This



means that one number needs to be split into three channels. This is difficult as each channel will have to be used evenly and sequentially. For this reason, the model that was picked for this project is the hue, saturation and value scheme or HSV. This model takes into account that all the colors are picked from the hue value. It is done in such a way that it is a degree around a center. The saturation controls how much to include the colors around nearest the degree picked. The value is what sets the max intensity that is allowed. This scheme not only lets the number of iterations be represented by as many colors as possible but also allows the encoding of less important but still visually pleasing patterns on the value intensity.



MandelBrot Set



The generation of the mandelbrot set is given by the equation of $Z_0 = 0 + 0i$ and the iterative part is $Z_{n+1} = Z_n{}^2 + C$. The first equation describes that the initial point working with is always 0. By changing this value, you can change the plane that the mandelbrot slices through. For this project setting it to 0 will give the famous look. The iterative equation is the one that generates the colors. It works by calculating the square of the previous calculation and adding a constant. The constant is unique to each pixel and is calculated by some form of ratio that will condense the pixels to a decimal grid. The x ratio is associated to the real number and the y ratio is associated to the imaginary number. If this equation diverges, it means that that a color will be associated to it with the amount of steps it took to reach infinity. Because infinity takes a long time to reach, it is normally calculated till it reaches an escape criteria. This criteria is calculated on the idea that numbers that are squared that are larger than one grow and numbers that are less than one shrink. Here is one example of the iteration numbers in matlab

```
c=-0.3+ -0.5j;
Z=0+0j;
for i=1:10
   Z=Z^2;
   Z=Z+c;
   fprintf('Iteration %i: %f %f\n', i,real(Z),imag(Z));
end
```

```
Iteration 1: -0.300000 -0.500000
Iteration 2: -0.460000 -0.200000
Iteration 3: -0.128400 -0.316000
Iteration 4: -0.383369 -0.418851
Iteration 5: -0.328464 -0.178851
Iteration 6: -0.224099 -0.382508
Iteration 7: -0.396092 -0.328561
Iteration 8: -0.251063 -0.239719
Iteration 9: -0.294433 -0.379631
Iteration 10: -0.357429 -0.276449
```

From this example, C the constant would be defined by the ratio of the x and y location of the pixel based on the zoom depth. As it iterates through 10 times, it seems like it will repeat forever. This would be a point inside the Mandelbrot set. In this case, the pixel would be set to black. If in another example we set C value to number not in the Mandelbrot set, $C = 1 + 0.5i$ then the Iteration output will look like this:

```
Iteration 1: 1 5.000000e-01
Iteration 2: 1.750000e+00 1.500000e+00
Iteration 3: 1.812500e+00 5.750000e+00
Iteration 4: -2.877734e+01 2.134375e+01
Iteration 5: 3.735798e+02 -1.227933e+03
Iteration 6: -1.368256e+06 -9.174614e+05
Iteration 7: 1.030390e+12 2.510645e+12
Iteration 8: -5.241634e+24 5.173884e+24
Iteration 9: 7.056504e+47 -5.423921e+49
Iteration 10: -2.941394e+99 -7.654785e+97
```

The numbers in this example are growing rapidly and is diverging. In this case the point is not in the mandelbrot set and will be given a color based off the iterations needed to escape. A simple way to decide if a number is diverging is by using the calculation of $a * a + b * b > 16$. In the case of this project, The calculation of streaks needed the value to be larger so the number checked was 10000 before it was considered to have escaped.

The iterative process is important to the parallel calculation because it is the part that takes the longest. The distribution of cpu time is not even across the image because not all pixels iterate the same amount of time. By increasing the maximum number of iterations, the output will be more detailed but will have to do many more calculation and therefore take longer.

Parallelization Styles

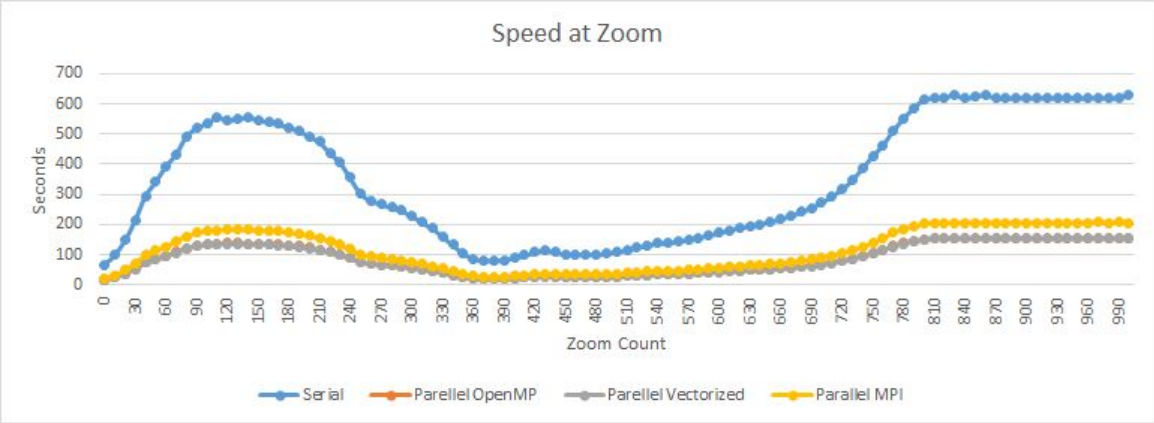
There are a few ways to parallelize this problem but one way is better than the others. The first way and the simplest way would be to spread the amount of pixels to each cpu. Since pixels don't all take the same number of iterations to compute, this will lead to some potently long periods of time where some CPUs are not doing anything. The second way would be to break the image into smaller sections. The problem with this approach is that areas of higher calculation time tend to be closer to others. For this reason cpu's might be given a region with higher calculation while others get patches of quicker calculations. The third option would be to do column by column. The problem with this is the array access for the storage will make it slower as cache lines will have to be moved between CPU's. This unneeded bottleneck is easily avoided by simply going row by row. This means that each CPU will grab one row at a time and calculate all the pixels across. This seems to be the best way because the array storing is within cache lines and the distribution of potential high calculation is spread out as it doesn't calculate

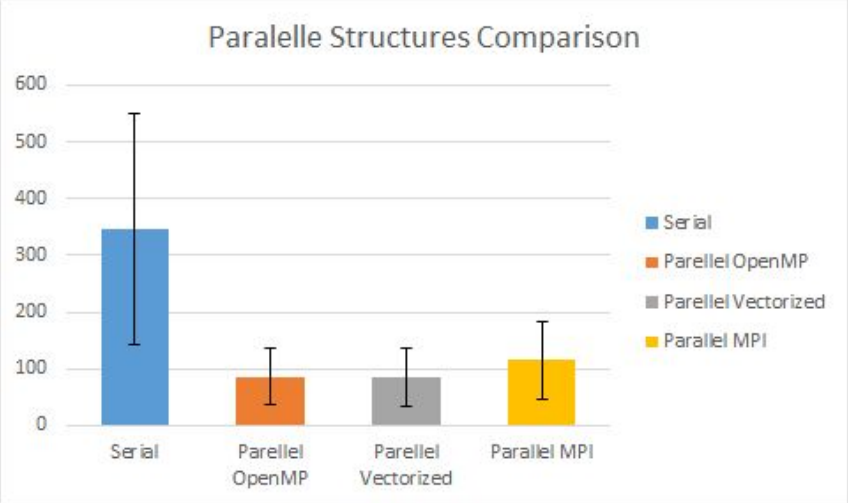
pixels that are grouped. For the purpose of this project, the row by row was picked for all implementations of the parallelization.

One more important part of the parallelization is the I/O operation in the storing of the image. This is important to consider when implementing the MPI code. The reason that this is an important distinction for the MPI code is because the memory is not shared and where the image is stored is important. For this reason, the implemented version uses a master slave approach. This allows all the nodes to do calculation on the rows and the one node will interface with the OS and file storage and store all the rows

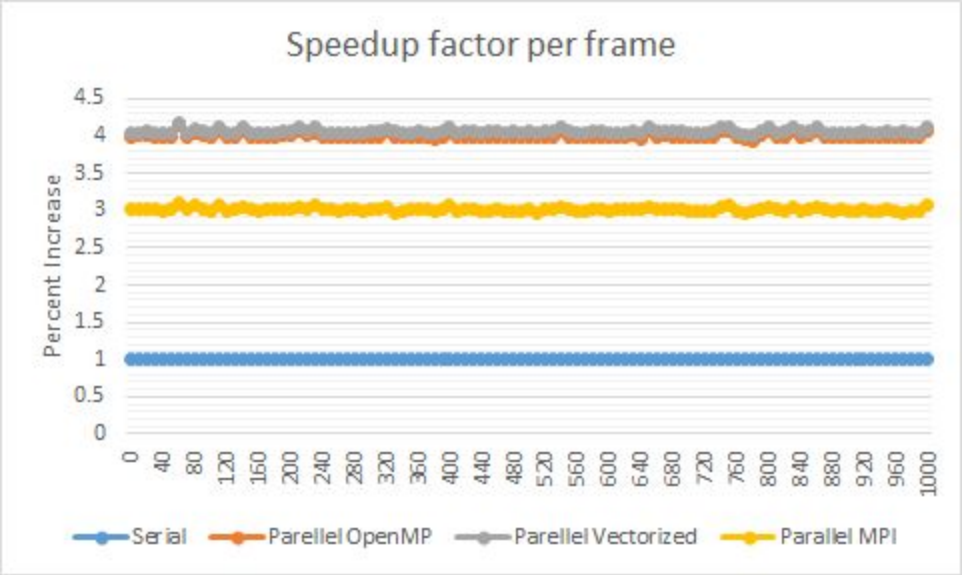
Results

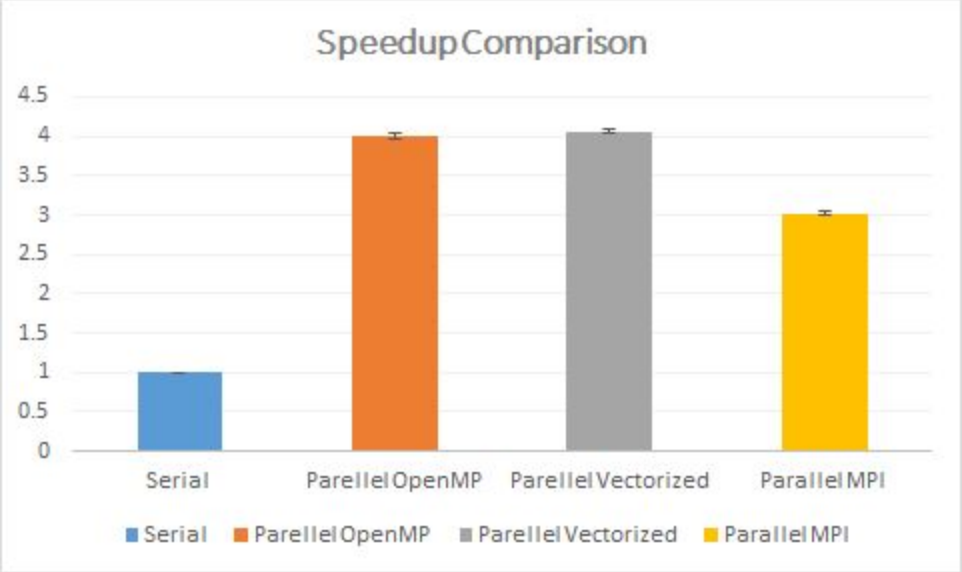
Applying all the parallel techniques styles gave a remarkable boost in performance improvements. All parallel systems use a network of either 4 nodes or 4 cores. To be able to get an average of the time to calculate an image of the mandelbrot set, the testing procedure generated a bunch of images while zooming in at every step. A video of the zoom that was benchmarked can be found online at https://www.youtube.com/watch?v=Lj1rC6ueEDc . This is the time graph that it took to generate every 10th frame:





This graph shows that some frames took longer than others to generate. This is because there were more full iterative calculation needed to be done. The plateau at the end is when the full image is completely black and every point is in the mandelbrot set. This is the worst case scenario. This graph show how variable the generation can be but does not help to see the speed up gain from the parallelisation. For that we need to use this graph:





The first graph shows the distribution of speedup per frame generated. As expected it is very linear. The time it took was almost exactly correlated to how many CPU's were working on it. The second graph shows the average with the standard deviation as the small black bars. The improvement is based almost as much as how many cores are working on it. This means that it linearly speeds up based on how many CPU's or nodes are used to compute it.

Conclusion

The parallelization of the Mandelbrot set was not very hard. For this reason it is a good project to see how difficult it is to implement paralyzation using different architectures. Using OpenMP was easy once the serial version was made. All that needed to be done was to add pragmas to the code and compiling it again. The more complicated code came from the MPI scripts. The reason why the MPI script was more complicated was because a master slave architecture needed to be made so that the file IO was possible. In doing so one of the nodes in the testing was lost as it was acting like a manager, leading to only three nodes being able to do some calculations.As there was no perceivable performance loss from the overhead of the parallelization, the increase in computing network size has the potential to surpass OpenMP performance.

Improvements

Throughout my research I came across many different interesting areas that could lead to visual and performance improvements in the mandelbrot set. As time was a limiting factor in this project, I was not able to perform tests on all these points, but I will share what I found out.

Visuals

The visual improvements where in the form of Anti-Aliasing and colouring techniques. Anti-aliasing is interesting in the mandelbrot set because when calculating a pixel, points around the pixel can be picked rather than its actual neighbour pixels. This could lead to a better looking anti-aliasing and could reuse the calculated between point for the other pixel that it is near to. There are many ways to color the Mandelbrot set. The technique with HSV was a very simple way to get as many colors as possible. Just because something is simple, doesn't mean that it is the best method. Some other coloring techniques that led to some cool results was using tables with sets of colors. This leads to having more control on the colors at set intervals. This type of coloring scheme also leads to be able to have a larger range of color once the generation gets deep.

Performance

Some performance improvements techniques that came across in my research where decimal accuracy, limiting iteration count, bit-shifting and parallelise frame generation in video fabrication.

Decimal accuracy is not not normally a huge issue other than the odd rounding errors in most application, but in the mandelbrot set, it becomes a more real issue if depth or speed is needed. CPU's are quite fast now day with floating point calculation and Double precision floating point precision. But because the mandelbrot set is never ending, zooming into it can be done indefinitely and can lead to needing to use even more precision so as the rounding doesn't mess up the calculations. When starting off implementing this in java, the double precision was only able to go up to 10^{-15} length before the look was completely lost. In the attempt to go deeper, I tried using some libraries that allows 4 times precision. The down side of this was that it took 100 times longer to calculate. This leads me to believe that there might be more optimal ways of doing such calculations with very small numbers.

Limiting the the iteration count was a optimisation that came recommended many times when doing high accuracy generations. The reason that it was recommended was because near the beginning, to get the look of the mandelbrot set you need at best 1000 iterations before it just gets unnecessary as the full detail is already there. More iteration counts are needed to be able to keep up with the high definition feel as the zoom gets deeper. Many videos that showcase mandelbrot sets will show not only the position of the zoom and the zoom depth but will also show the beginning iteration count and the ending iteration count.

Bit shifting is not the most impactful optimization as it takes place outside the iterative part but could lead to some performance improvements. How to do this is that a pixel can be stored in an integer rather than a char array of size 3. By having it in an integer, it would reduce the footprint of passing it around and is much easier to alter it with bit shifting rather than doing arithmetic on it.

The last optimisation comes from the fact that video generation of zooming into the mandelbrot set is very cool and requires recalculating of very similar pixel values. The optimization would be applied in looking at the previous frame and if it sees that there are many max iterations around its new point, it might also be a max iteration point as well. This could potentially lead to some drastic speedups if used well. I was not able to find any examples of this so it is mostly speculation.

Code

Throughout this project I had used many different languages. Each one had their advantages and disadvantages.

C++

All the benchmarking was done using the C++ code. It was used to perform the calculation and generate series of images. The reason that this one was picked to calculate the

speed was because it was the only one that OpenMP and MPI seemed to work on. Also it was the fastest compared to all the other ones.

Processing 3 (Java)

Due to it simplicity in graphics, I chose to do most of my testing using this script. It was useful for finding out bugs quickly and for finding a good place to zoom in.

Matlab Script

Matlab was used to play around with complex numbers and understand how the escape process worked.

Python

The reason why I used python was to generate video files and convert the .ppm files to other images types as they are not universally accepted in all imaging software.

Bash

To be able to Iterate through each script and average output to reduce noise and also run the code over multiple days on the server I had make a bash script to run the code for me. It is not friendly to use with decimal number so it calls awk to do some calculations. It also has a loading bar to debug time lengths. To run this code without me there I used Nohup.