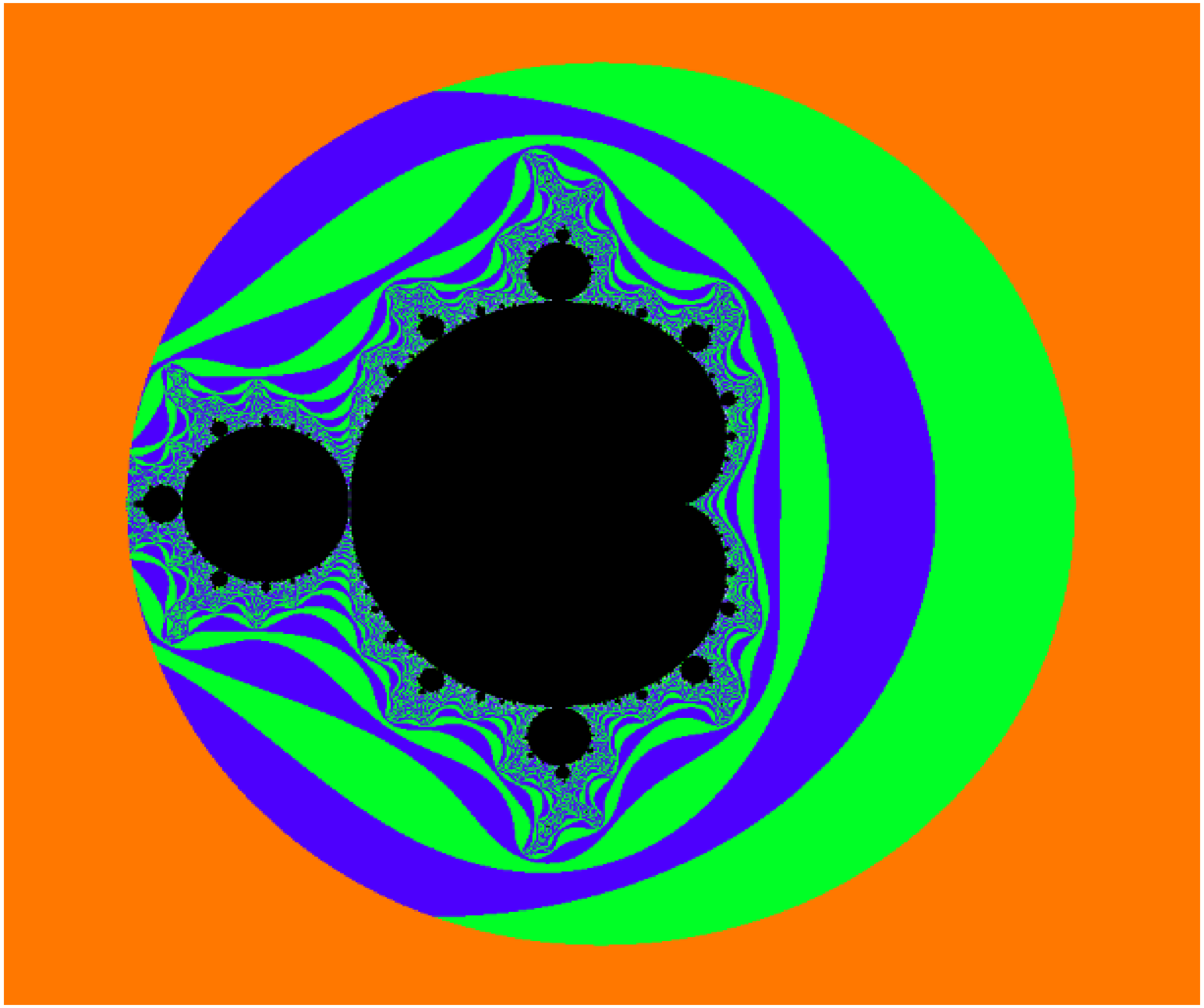
MANDELBROT SET

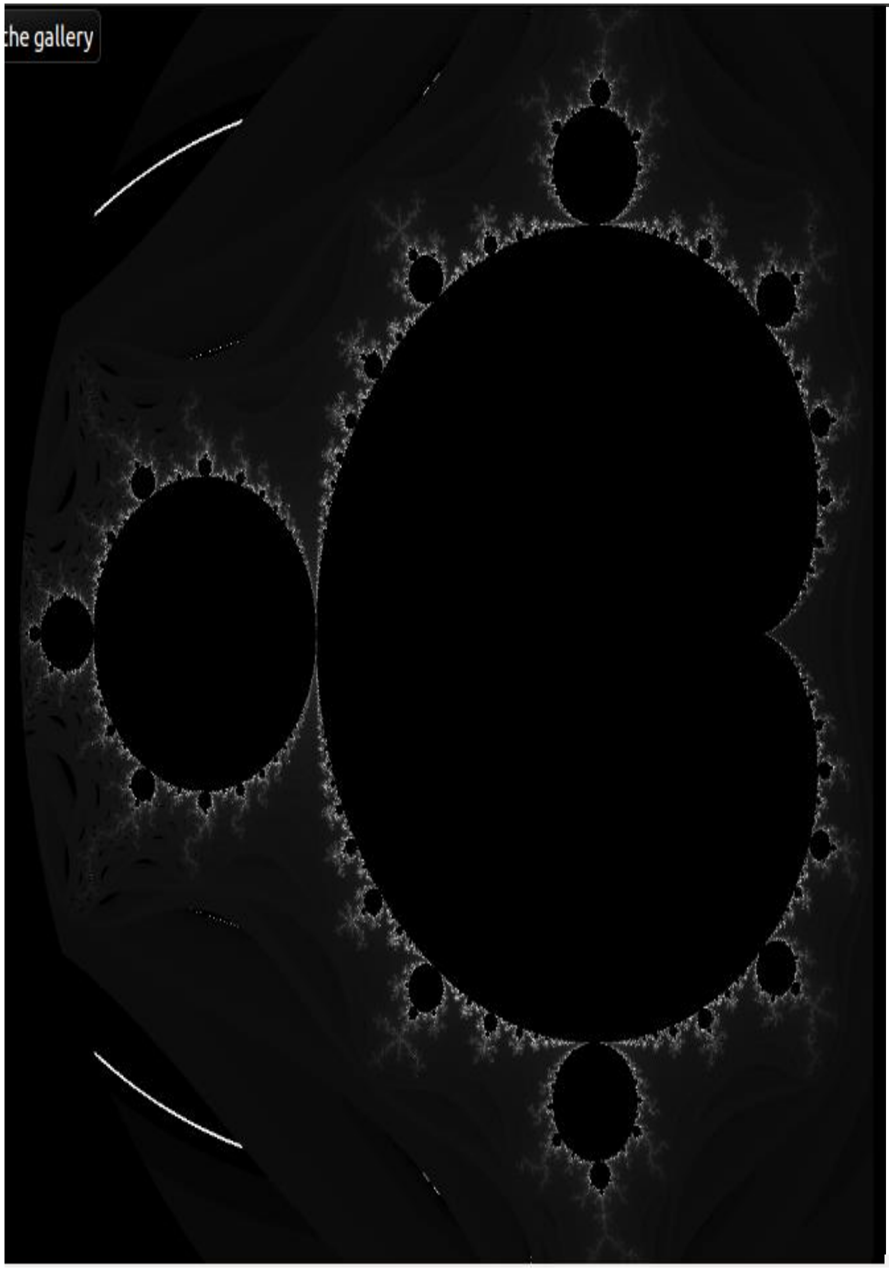
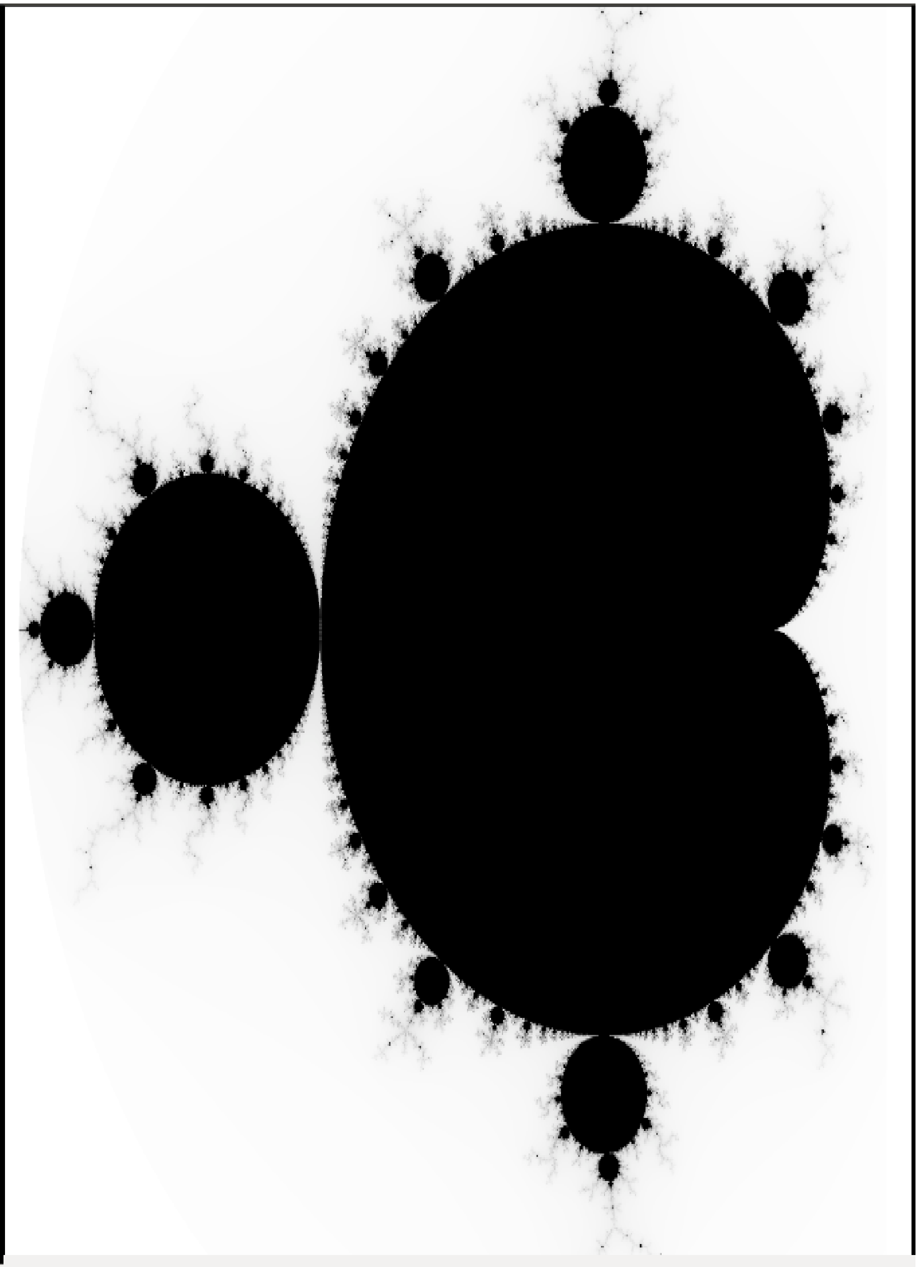


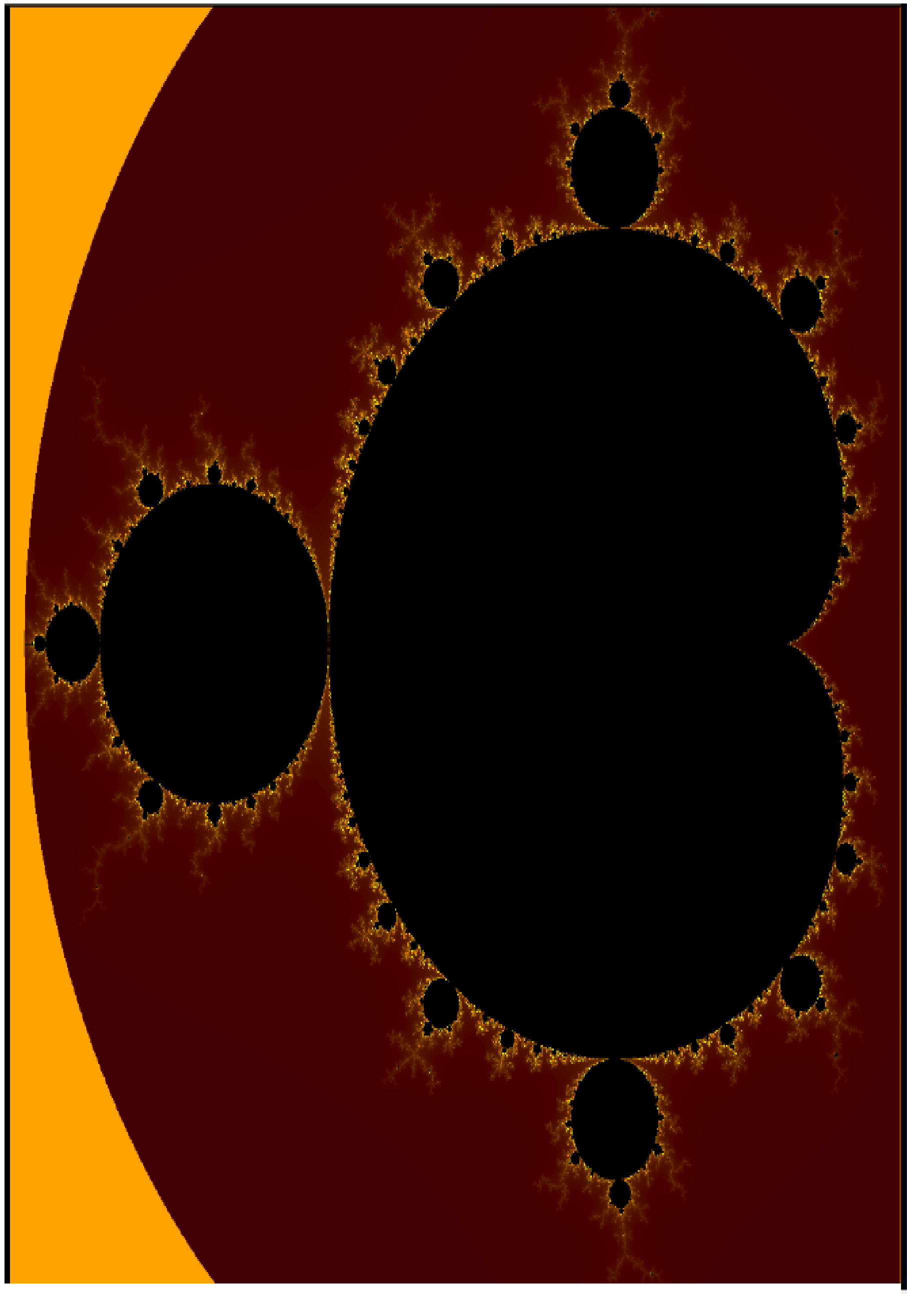
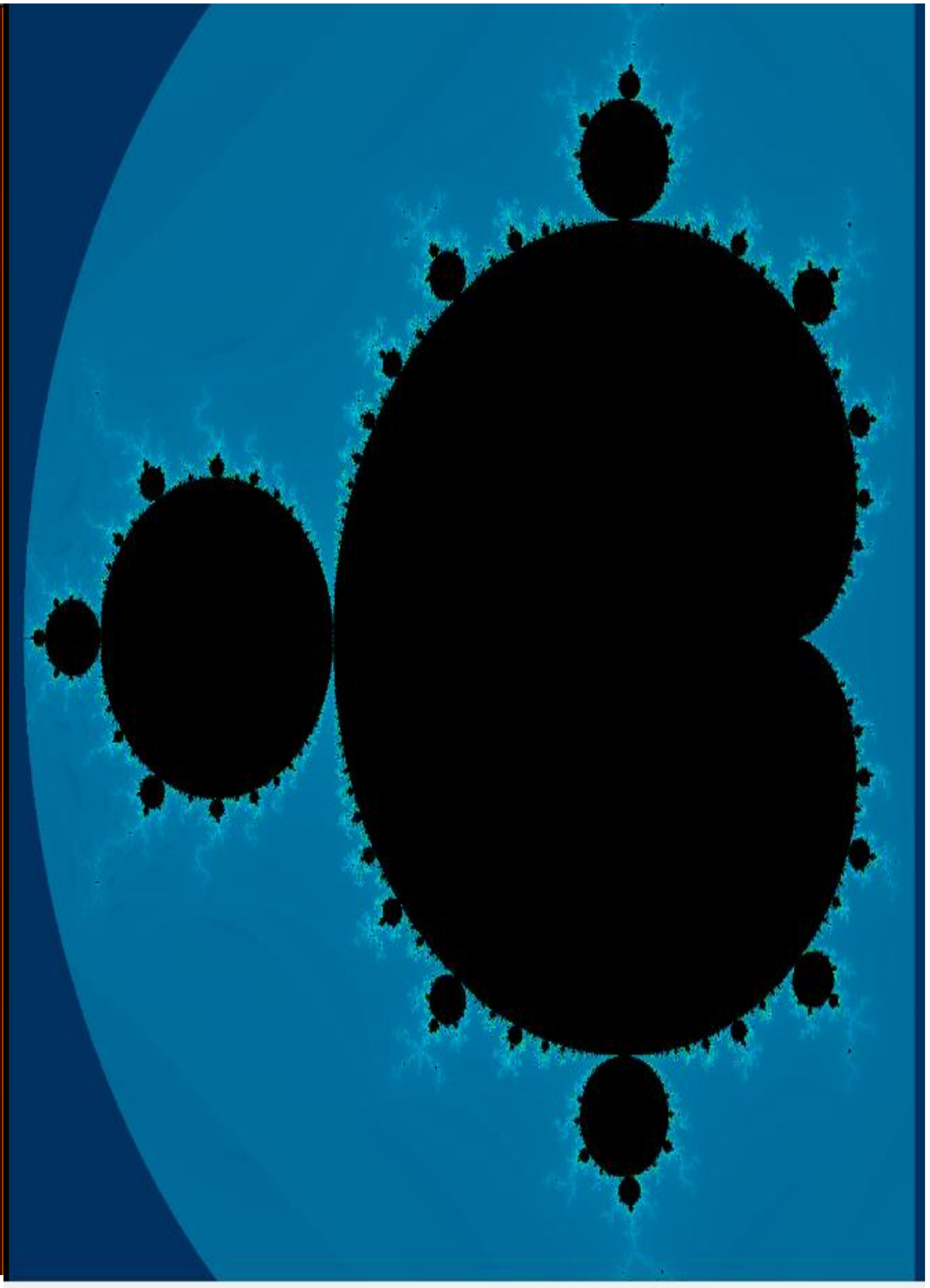
Problem Introduction

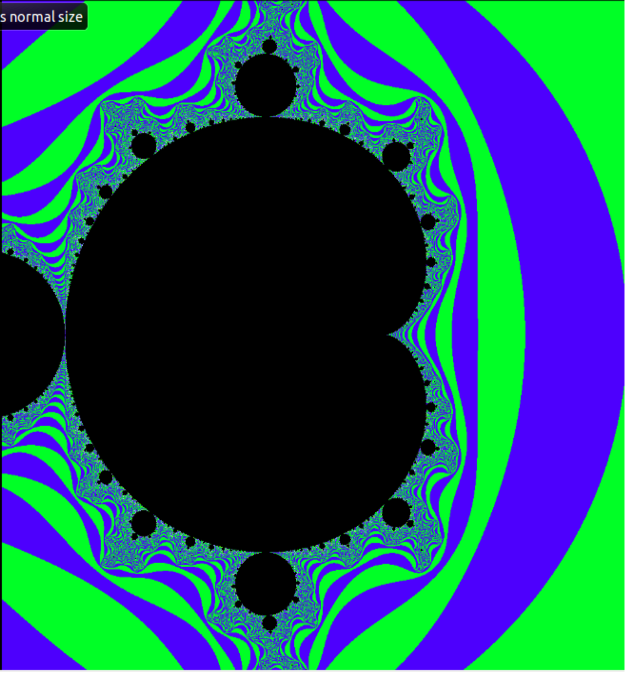
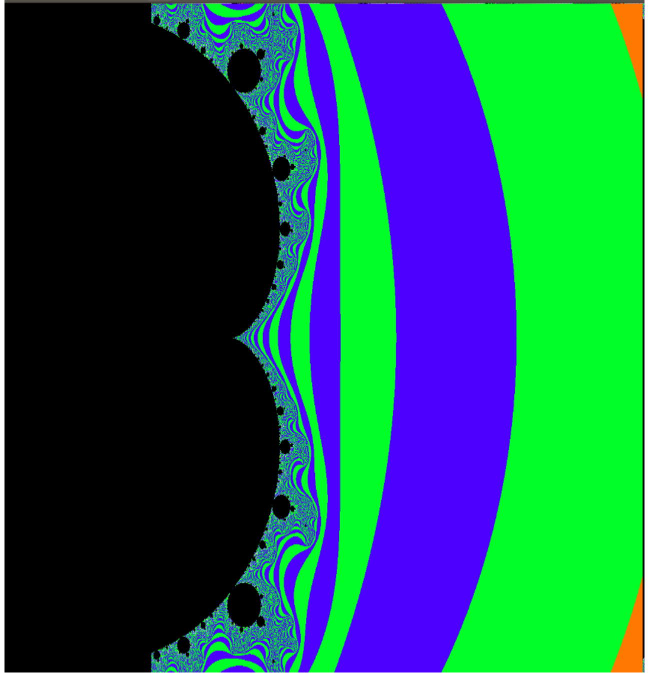
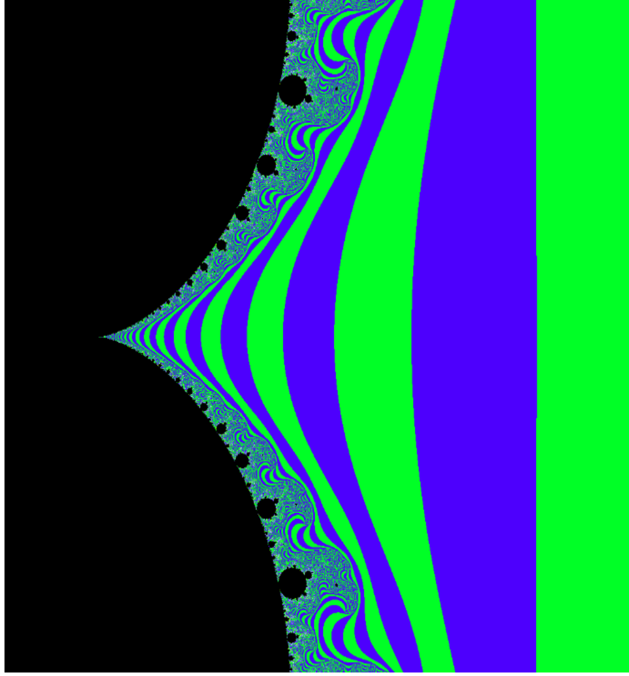
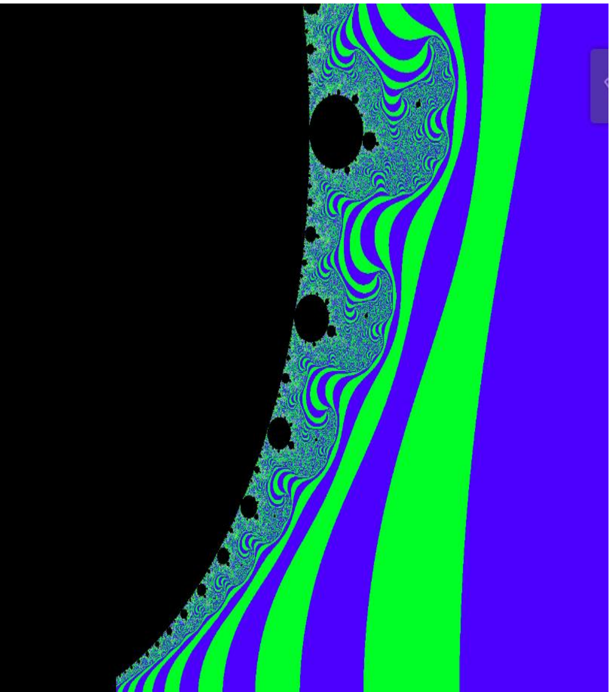
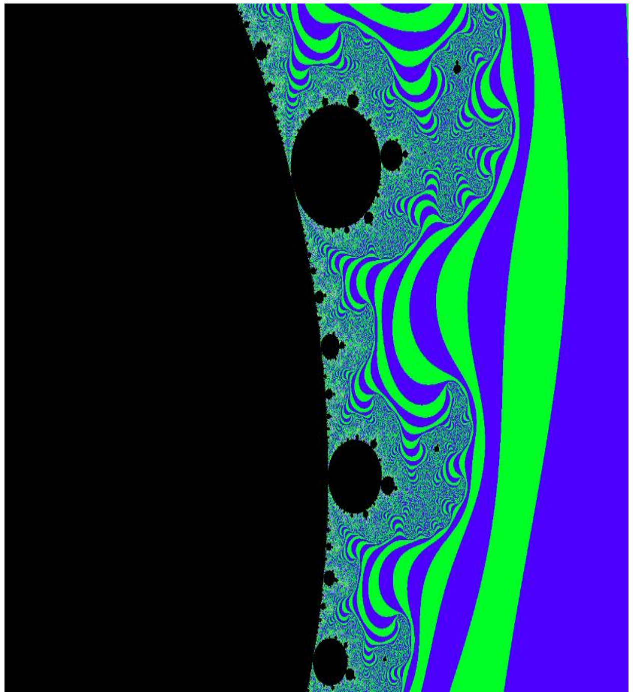
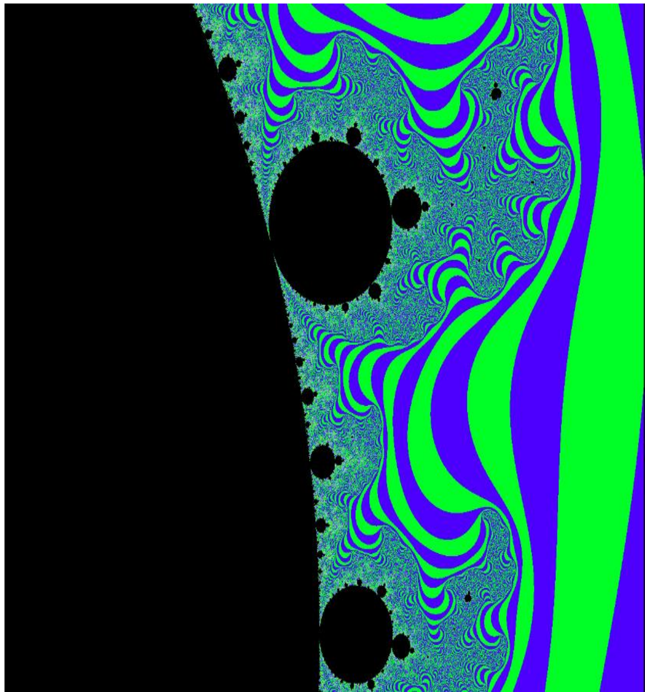
- Mandelbrot set calculation is the set of complex numbers 'c' for which the sequence (c, $c^2 + c$, $(c^2+c)^2 + c$, $((c^2+c)^2+c)^2 + c$, $(((c^2+c)^2+c)^2+c)^2 + c$, …) does not approach infinity and is named after the mathematician Benoit Mandelbrot. The sequence is calculated using the formula :

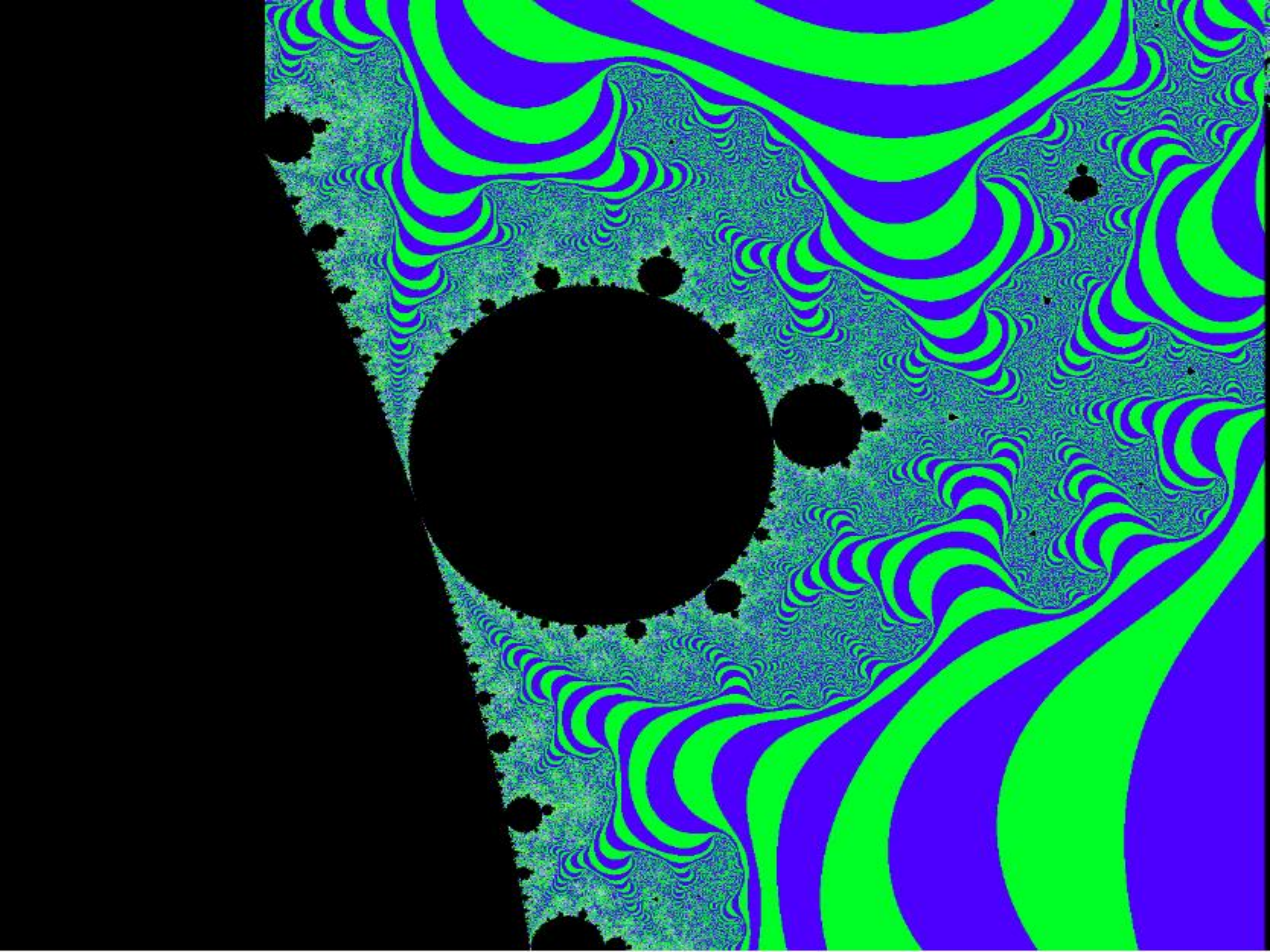
  z(n+1)=z(n)^2+c

That is, a complex number *c is part of the Mandelbrot set if, when starting with Z0 = 0* and applying the iteration repeatedly, the absolute value of *Zn remains bounded (say* less than 2), however large *n gets.* For a 256x256 image, we take 65536 points, divided symmetrically in the plane, bounded by a circle of radius 2, and compute the sequence to decide if they belong to the Mandelbrot set, based on if the sequence converges or diverges. To calculate the sequence, we iterate 2000 times for a point in the plane.


the gallery



s normal size








Input

- Image size, { 256x256, 512x512, 1024x1024 } which decides the number of points in plane to be operated on.

Output

- Output will be a Mandelbrot image, where points belonging in the Mandelbrot set are colored black. Based on the number of iteration, after which a sequence starts diverging, an RGB value is assigned to that coordinate.

Serial Code

- Complexity : O(n*n)

  For an image of size n*n, we consider n*n points in the plane and check if they belong to the Mandelbrot Set or not. For each point, at maximum 2000 iterations are performed.

- **Pseudo Code**

```
For each pixel (Px, Py) DO
{
    x0 = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale
            (-2.5, 1))
    y0 = scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale
            (-1, 1))
    x = 0.0
    y = 0.0
    iteration = 0
    max_iteration = 2000
    while ( x*x + y*y < 2*2 AND iteration < max_iteration )
    {
            xtemp = x*x - y*y + x0
            y = 2*x*y + y0
            x = xtemp
            iteration = iteration + 1
    }
    imageMatrix[Px][Py]=color[iteration]
}
For each pixel (Px, Py)
{
    plot(imageMatrix[Px][Py])
}
```

Optimization Strategy

We are performing 2000 iterations, i.e. calculating 2000 elements of the sequence, to check if the sequence diverges or not. However, if Z(n+1)=Z(n), this means that the sequence has converged already, and will not diverge for further values. So we don't need to check further till 2000 iterations. This optimization strategy is known as **Periodicity** .

Scope of Parallelism

This is an embarrassingly parallel problem, since the value calculated for each pixel is independent of other pixels. The imageMatrix can be computed parallely without any data dependency, using the formula shown previously.

Increasing Problem Size

Problem size can be increased by including more points in the plane, but within the same range. We take 65536, 262144 and 1048576 points in the plane for 256x256, 512x512 and 1024x1024 image sizes.

Parallel Code

OpenMP

- We use the naive parallel method. We are using C, which is row major; so we parallelize the outer loop. To schedule the tasks, we use dynamic scheduling.

- We can also use the naive parallel method, where the matrix can be divided into (problem_size/number_of_threads) parts. This can be achieved using static scheduling. However, size of L1 cache is 32KB. For a problem size of 256x256 pixels, we need to calculate 65536 integers(i.e. 65536 bytes=64KB) which can not be fit all together in the L1 cache. Only half of the matrix can be contained in L1 cache at a time. Typically, the size of cache line is 64 bytes, which means that if we use the above strategy, the possibility of two blocks i.e. two rows which are not consecutive, used by two threads have more chances of being placed in the same cache line, increasing the number of misses; as compared to when two consecutive rows are accessed by two different threads. This might be the reason, we get better speedup for dynamic scheduling, with default chunk size, as compared to static scheduling.

MPICH

- In MPICH, we use 3 strategies to perform the task :

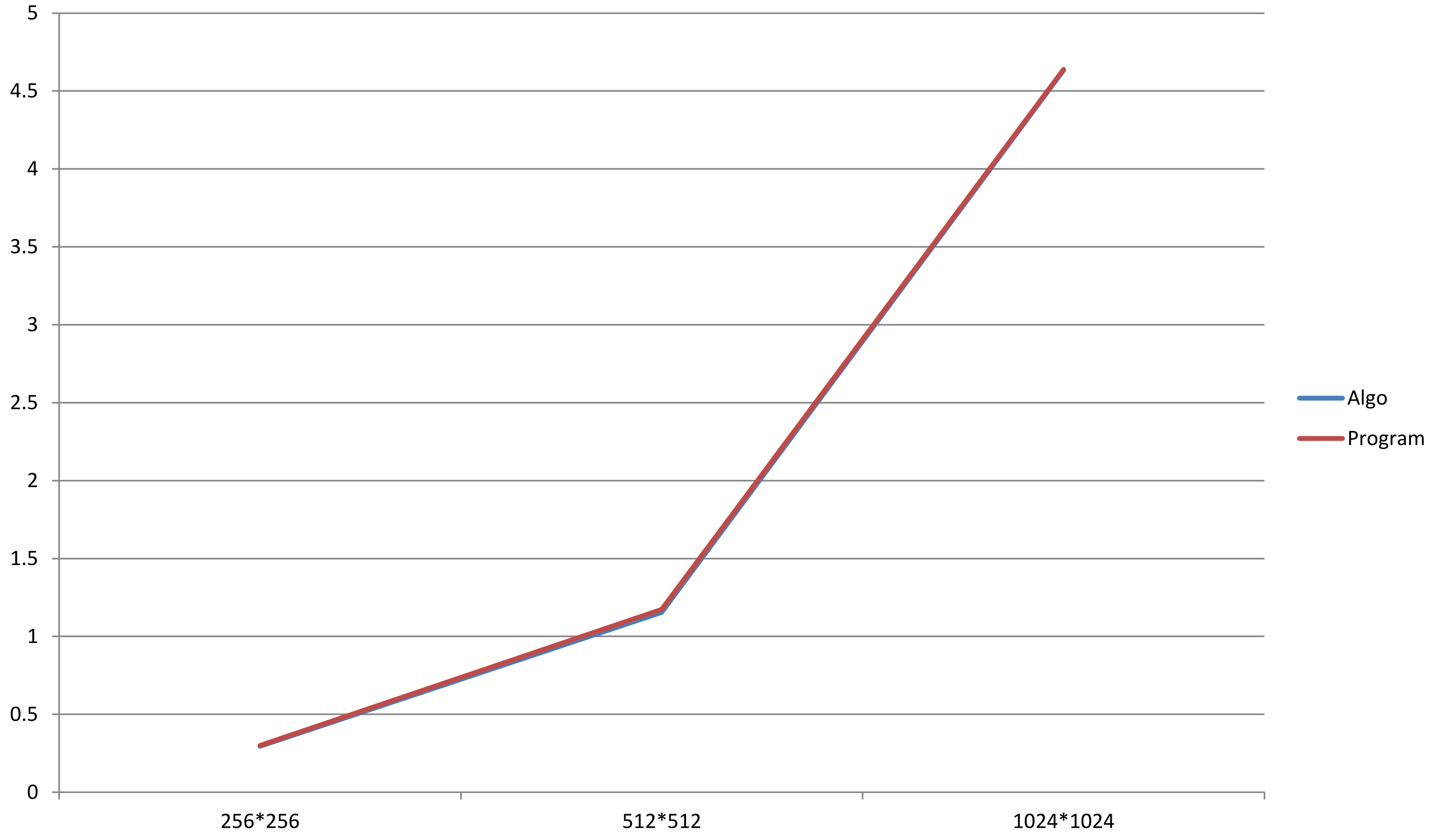
1. We use the naive MPI method, where we have 4 processes and each process calculates the partial image matrix and sends the matrix to process 0, which writes it to the final image. Here, to send the matrix we used **MPI_Send()**, which is blocking.

2. We divide the matrix into 4 parts(1024x256) and use **MPI_Gather** to receive the image matrix from 4 processes to the root process(my_rank=0). After process 0 receives the imageMatrix from all processes, it processes the matrix values and writes the RGB values to generate the bmp image.

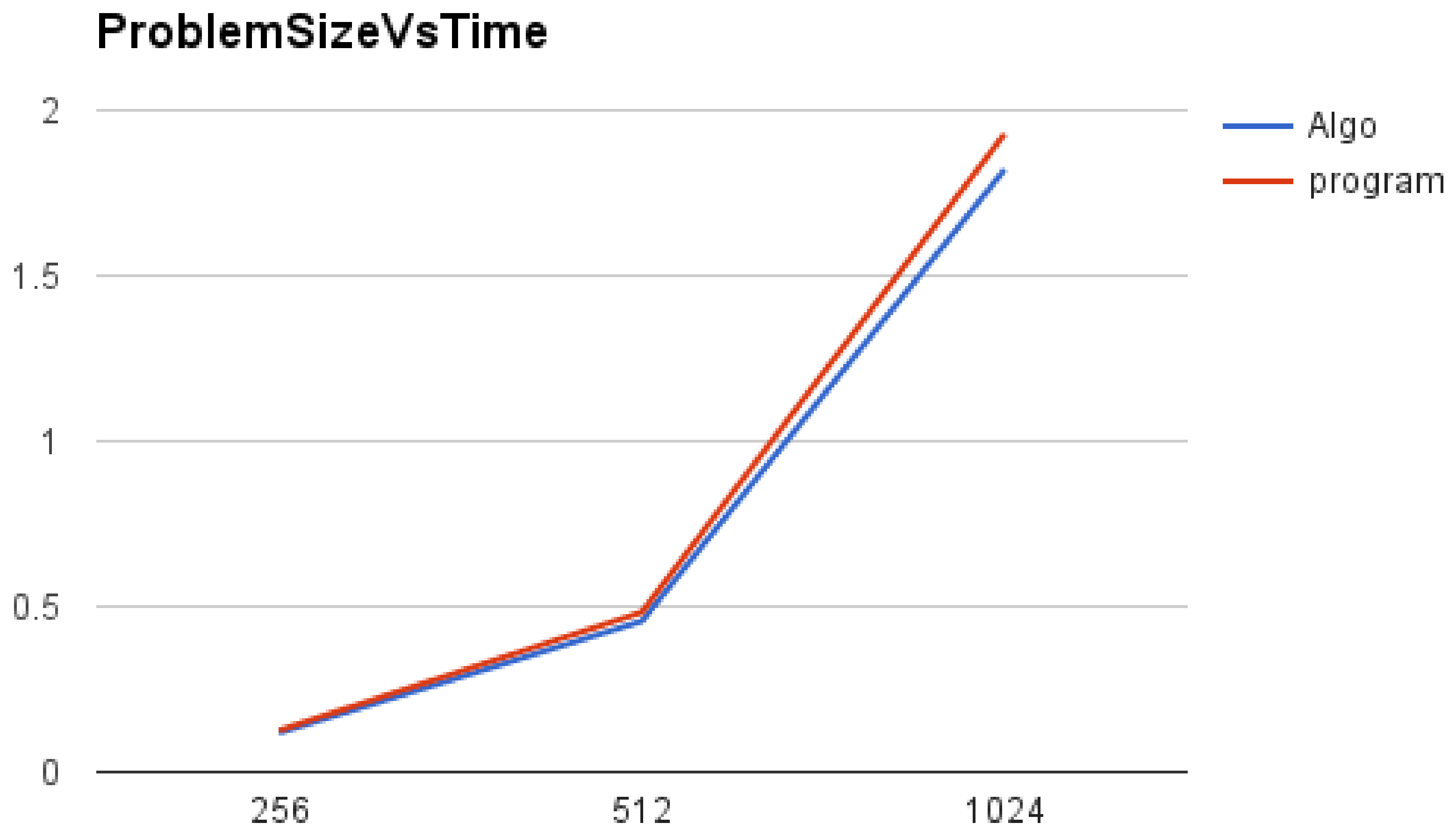
3. We divide the matrix into 3 parts and use the process ranked 0, to perform the write operation to the file. Each of 3 processes operate upon a pixel value and check if it belongs to the mandelbrot set or not. Immediately after checking that, the process sends the x, y coordinates of the pixel and the corresponding color(RGB value, based on if it belongs to mandelbrot set or not) to process 0, using **MPI_Isend**, which is non-blocking and hence allows the same process to perform the same computation for next pixel while sending the data alongside. The process 0, on receiving, writes the corresponding RGB value to the file, by going to that coordinate(pixel) using fseek().

RESULTS

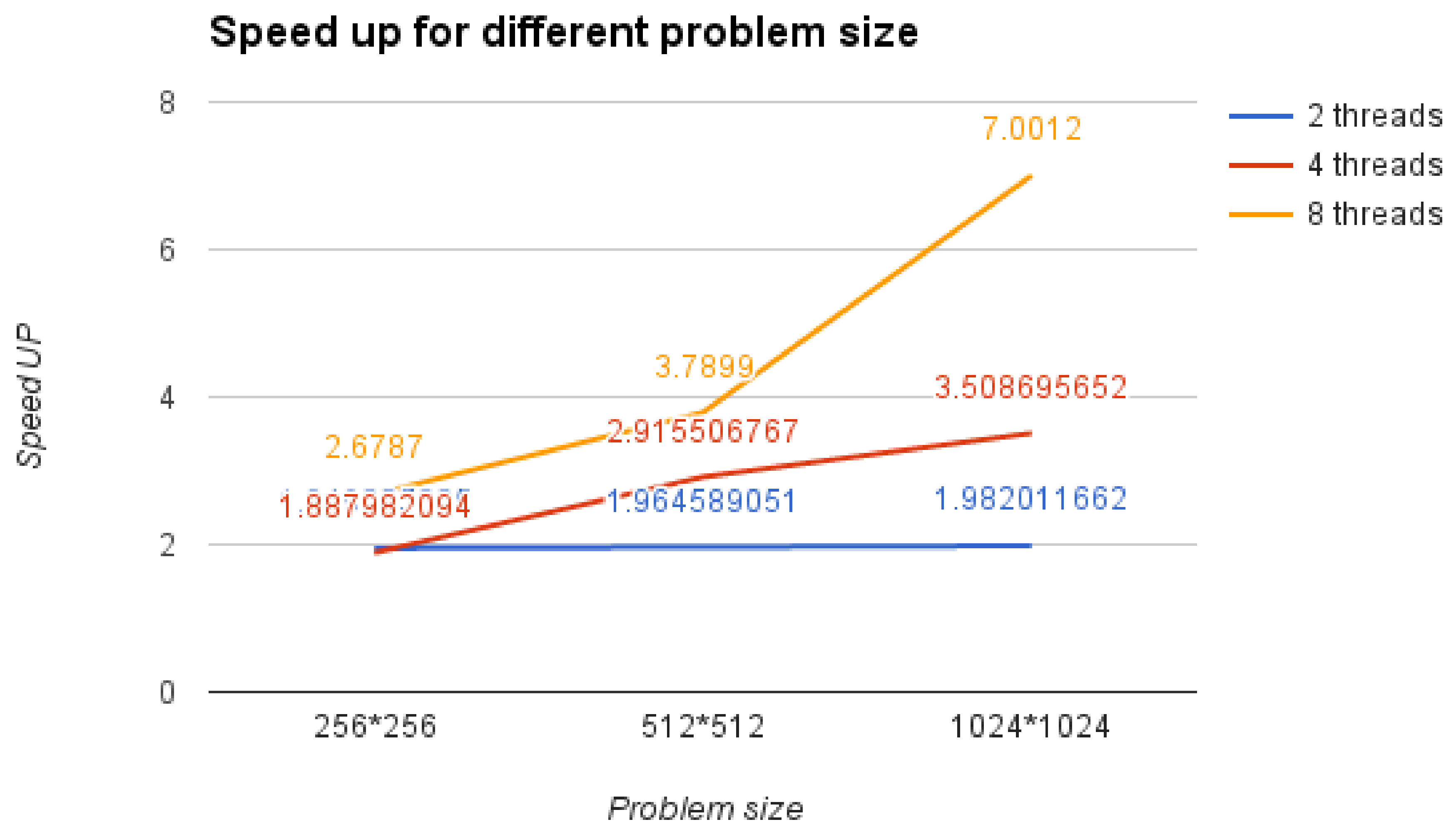
ProbemSize Vs Time for serial implementation



ProblemSize Vs Time for Openmp implementatioin

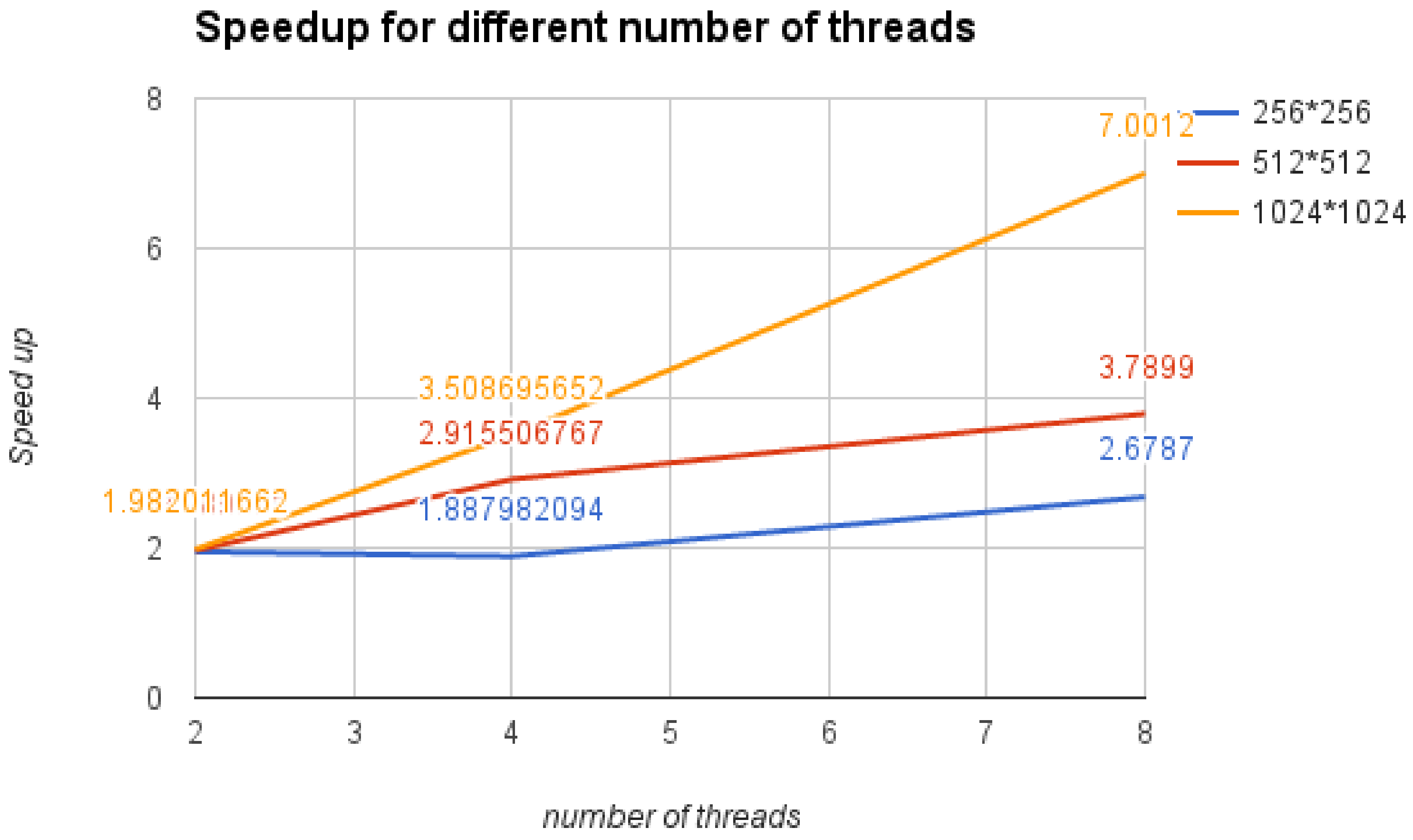


SpeedUp vs ProblemSize for OpenMP implementation



- As we have mentioned above, this is an embarrassingly parallel problem. So we get a high speedup for this problem.
- As problem size increases, speedup increases for a fixed number of threads, which is evident from the above graph. With increasing problem size, the overheads involved are less than the gain in performance due to parallelization.
- However as number of threads increase, the overheads involved in creating, synchronization etc of threads increase, for fixed problem size. The same can be observed from the graph, the maximum speedup for 2 threads is 1.98, however for 4 threads, the maximum speedup is around 3.5 for problem size 1024*1024.

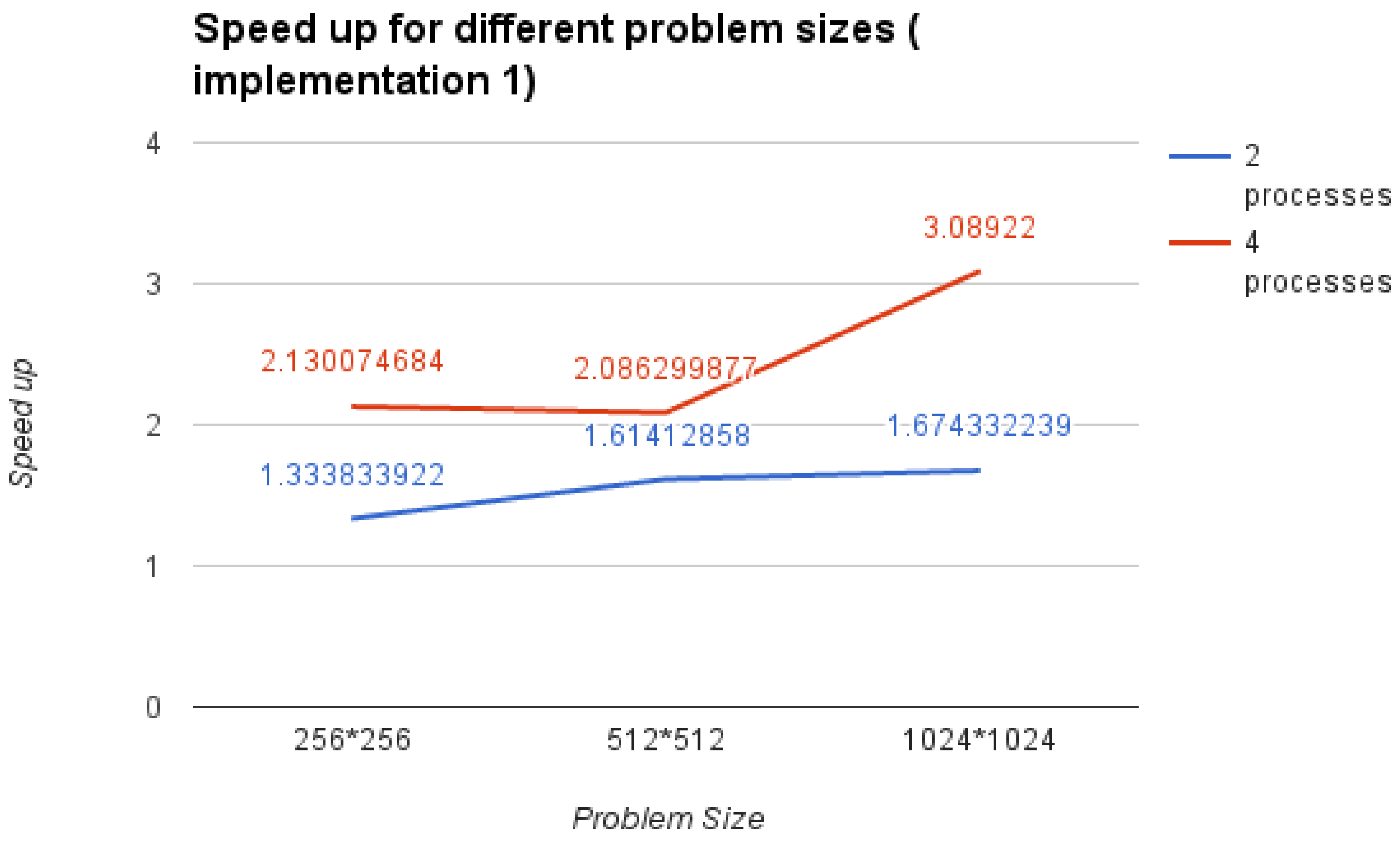
SpeedUp Vs number of threads

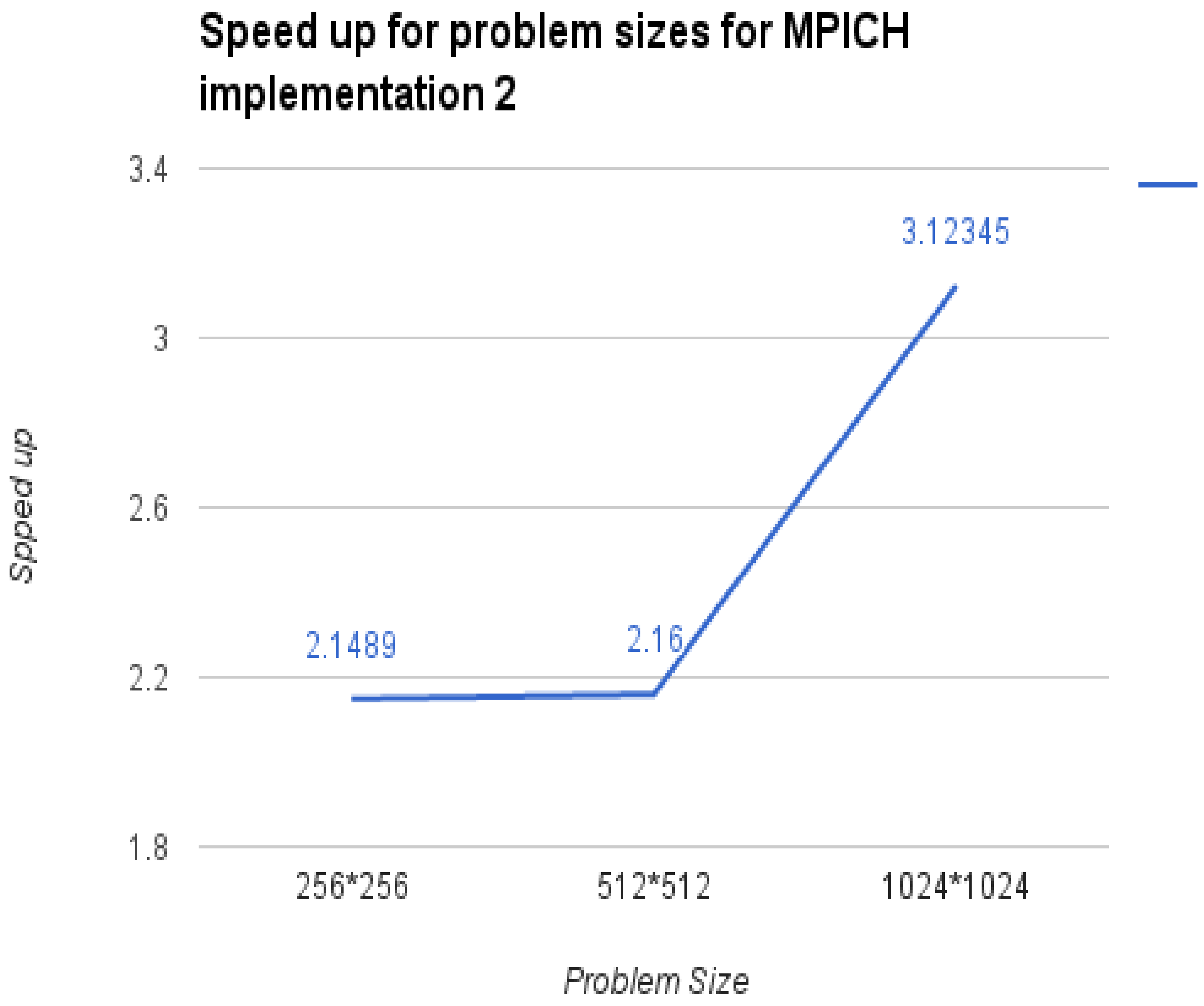


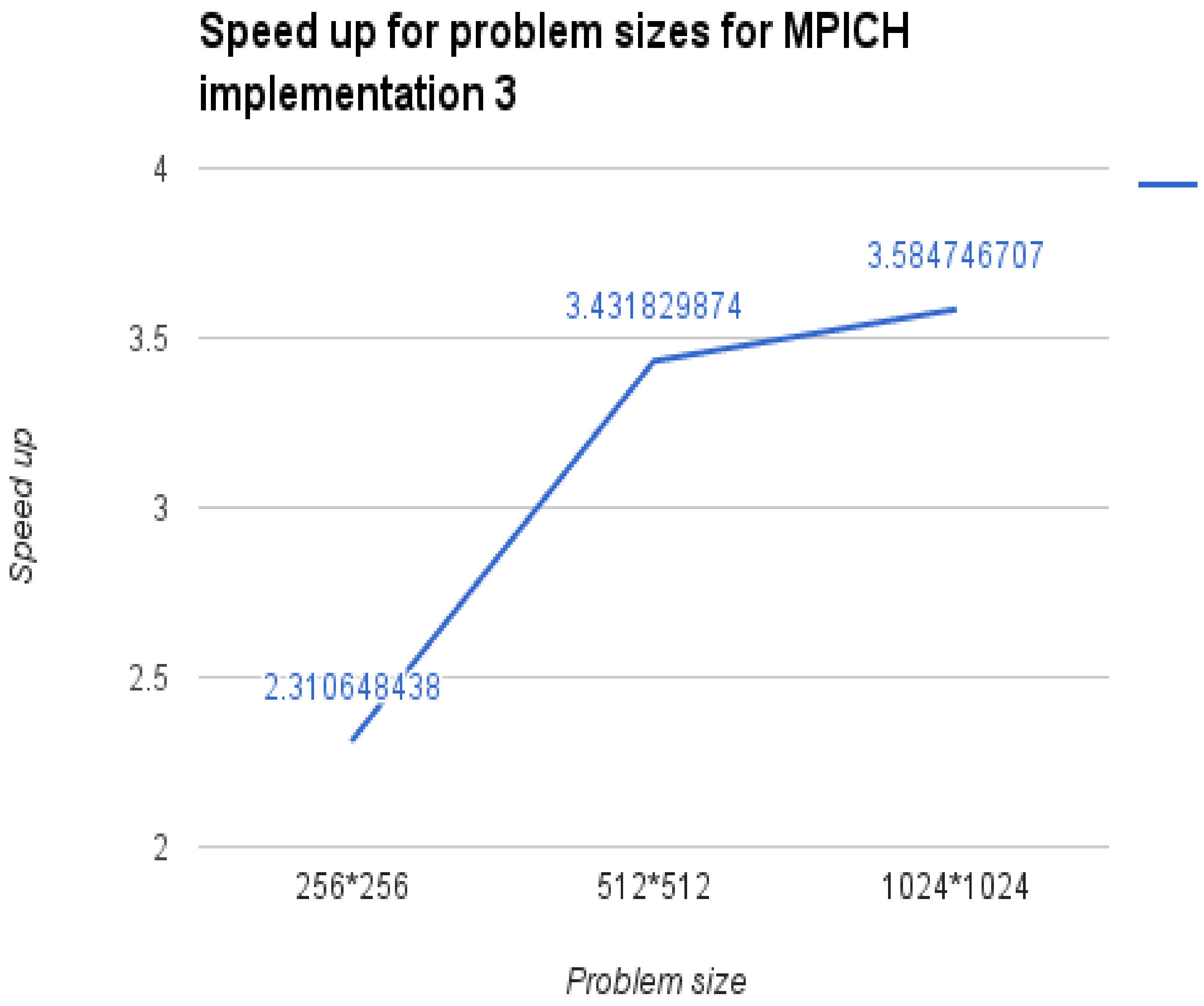
- For a fixed problem size, speedup usually increases as the number of threads increase.

  The same can be seen for the problem sizes 1024 and 512.
- However if the problem size is small, then the overheads due to creating threads dominate the gain in performance due to parallelization; which is the case in problem size 256.

Speedup as a function of problemSize for MPICH implementations




Speed up for problem sizes for MPICH implementation 2
3.4
3
2.6
2.2
1.8
Spped up
2.1489
2.16
3.12345
256*256
512*512
1024*1024
Problem Size


Speed up for problem sizes for MPICH implementation 3
4
3.5
3
2.5
2
Speed up
2.310648438
3.431829874
3.584746707
256*256
512*512
1024*1024
Problem size

- We can see that the speedup increases with increase in problem size in all the three cases.
- We get maximum speed up in implementation 3, where the envelope, i.e. the pixel coordinates and its corresponding RGB value is sent as soon as it is calculated. The speed up in the other two implementations is comparable, which is expected, as the computations are same. The time is consumed in communication. In implementation 3, we used MPI_Isend(), which is non blocking, and hence it allows computation, while communication is happening. This saves time in communication as compared to other two cases where all computation is performed first and then communication takes place.

Karp-Flatt Metric

	2	4	8
Speedup	1.982	3.50	7.001
e	0.0090	0.0466	0.0204

For embarrassingly parallel problems, the serial fraction is so small that it cannot be captured by the Karp-Flatt Metric, which explains the above decreasing values of e. The experimentally determined serial fraction decreases.

CONCLUSION

- From the speedup obtained, we conclude that OpenMP is better than MPI as this problem is computation intensive. OpenMP implementation is faster than MPI, as no communications are involved in it.