HW 6
Parallel Mandelbrot

Philip Nelson

2018 October 5

Introduction

The purpose of this assignment is to write an MPI program that generates an image of the mandelbrot set as described by the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$. My program takes as input the image height, image width, maximum number of iterations, minimum x/real value, maximum x/real value, and minimum y/imaginary value. Then in parallel, it calculates the number of iterations every pixel in the image takes to diverge. The program uses the master-slave architecture to send tasks. The master sends rows to the slaves in a round-robbin style. The slaves calculate the iteration values for the pixels in each row they are assigned and send them back to the master. The master then receives each message and incorporates the message buffers into one array. When every pixel has been calculated, the array is converted to a PNG image, using libpng, based on a color scheme. Functionality to output to a ppm file is included.

Code

The code is broken up into five main files, main.cpp, controller.cpp, calculator.cpp, color.cpp, and writePNG.hpp. The files are included below.

0.1 main.cpp

```
1  #include "controller.hpp"
2  #include <algorithm>
3  #include <fstream>
4  #include <functional>
5  #include <iostream>
6  #include <mpi.h>
7  #include <vector>
8
9  /*
10  *std::vector<int> render_contiguous(int start,
```

```
11  *                                      int end,
12  *                                      double X_MIN,
13  *                                      double X_MAX,
14  *                                      double Y_MIN,
15  *                                      double Y_MAX,
16  *                                      int IMAGE_HIGHT,
17  *                                      int IMAGE_WIDTH,
18  *                                      int MAX_ITERS)
19  *{
20  *  std::vector<int> imagebuf;
21  *  imagebuf.reserve(end - start + 1);
22  *  for (auto i = start / IMAGE_WIDTH; i < IMAGE_HIGHT; ++i)
23  *  {
24  *    for (auto j = start % IMAGE_WIDTH; j < IMAGE_WIDTH; ++j)
25  *    {
26  *      imagebuf.push_back(mandelbrot(
27  *        i, j, IMAGE_WIDTH, IMAGE_HIGHT, MAX_ITERS, X_MIN, X_MAX,
        Y_MIN,
28  *Y_MAX));
29  *    }
30  *  }
31  *
32  *  return imagebuf;
33  *}
34  */
35
36 /**
37  * Write the iteration data as a ppm image
38  *
39  * @param imagebuf The array containing all the iteration data
40  * @param width    The width of the image in pixels
41  * @param height   The height of the image in pixels
42  * @param filename The name of the ppm image
43  */
44 void write_ppm_image(std::vector<int> const& imagebuf,
45                      const int width,
46                      const int height,
47                      std::string fileName)
48 {
49   std::ofstream fout(fileName);
50   fout << "P3\n" << width << " " << height << "\n255\n";
51   std::for_each(begin(imagebuf), end(imagebuf), [&fout](int num) {
52     fout << num << " " << num << " " << num << " ";
53   });
54   fout << std::endl;
55 }
```

```
56
57 int main(int argc, char** argv)
58 {
59   MPI_Init(&argc, &argv);
60
61   int rank;
62   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
63   if (argc < 6)
64   {
65     if (0 == rank)
66     {
67       std::cerr << "incorrect number of arguments\n"
68                 << "mandelbrot height width max_iters x_min x_max y_min
                      \n";
69     }
70     MPI_Finalize();
71     exit(EXIT_FAILURE);
72   }
73
74   if (0 == rank)
75   {
76     master(argv);
77   }
78   else
79   {
80     slave(argv);
81   }
82
83   MPI_Finalize();
84
85   return (EXIT_SUCCESS);
86 }
```

0.2 controller.cpp

```
1  #include "controller.hpp"
2  #include "calculator.hpp"
3  #include "color.hpp"
4  #include "ppmToBmp.hpp"
5  #include "writePNG.hpp"
6  #include <algorithm>
7  #include <mpi.h>
8  #include <vector>
9
10 enum TAG
11 {
12   RENDER,
```

```
13    STOP
14  };
15
16  void master(char** argv)
17  {
18    int rank, world_size;
19    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
21
22    int IMAGE_HIGHT = std::stoi(argv[1]);
23    int IMAGE_WIDTH = std::stoi(argv[2]);
24    int IMAGE_SIZE = IMAGE_WIDTH * IMAGE_HIGHT;
25    const int MAX_ITERS = std::stoi(argv[3]);
26    // const double X_MIN = std::stod(argv[4]);
27    // const double X_MAX = std::stod(argv[5]);
28    // const double Y_MAX = std::stod(argv[6]);
29    // const double Y_MIN =
30    // Y_MAX - (X_MAX - X_MIN) * ((double)(IMAGE_HIGHT)) / IMAGE_WIDTH;
31
32    std::vector<int> imagebuf(IMAGE_SIZE);
33
34    auto tests = 1;
35    auto t1 = MPI_Wtime();
36    for (auto i = 0; i < tests; ++i)
37    {
38
39      auto dest = 1;
40      std::vector<int> buf(1);
41      for (auto i = 0; i < IMAGE_HIGHT; ++i)
42      {
43        buf[0] = i;
44        MPI_Send(buf.data(), 1, MPI_INT, dest++, TAG::RENDER,
              MPI_COMM_WORLD);
45
46        if (dest > world_size - 1) dest = 1;
47      }
48
49      buf.resize(IMAGE_WIDTH);
50      MPI_Status stat;
51      for (auto i = 0; i < IMAGE_HIGHT; ++i)
52      {
53        MPI_Recv(buf.data(),
54                 buf.size(),
55                 MPI_INT,
56                 MPI_ANY_SOURCE,
57                 MPI_ANY_TAG,
```

```
58                MPI_COMM_WORLD,
59                &stat);
60
61        std::copy(
62          begin(buf), end(buf), begin(imagebuf) + IMAGE_WIDTH * stat.
              MPI_TAG);
63      }
64    }
65
66    auto t2 = MPI_Wtime();
67
68    /*
69    ppmToBmp(imagebuf,
70             IMAGE_WIDTH,
71             IMAGE_HIGHT,
72             std::bind(color_scheme_2, std::placeholders::_1, MAX_ITERS),
73             "brot.bmp");
74    */
75
76    std::vector<uint8_t> pixles;
77    pixles.reserve(IMAGE_SIZE * 3);
78    std::for_each(
79      begin(imagebuf), end(imagebuf), [&MAX_ITERS, &pixles](int iter) {
80        auto [r, g, b] = color_scheme_1(iter, MAX_ITERS);
81        pixles.push_back(r);
82        pixles.push_back(g);
83        pixles.push_back(b);
84      });
85
86    std::string filename = "brot" + std::to_string(IMAGE_WIDTH) + ".png";
87    if (save_png_libpng(filename, pixles.data(), IMAGE_WIDTH, IMAGE_HIGHT
        ))
88    {
89      std::cout << "Successfully wrote " << filename << '\n';
90    }
91    else
92    {
93      std::cout << "Failed to write " << filename << '\n';
94    }
95
96    auto t3 = MPI_Wtime();
97
98    std::cout << IMAGE_HIGHT << " x " << IMAGE_WIDTH << '\n'
99              << "Time to compute: " << (t2 - t1) / tests
100             << '\n'
101             //<< "Time to compute: " << (t2 - t1) << '\n'
```

```
102             << "Time to write image: " << t3 - t2 << '\n'
103             << '\n'
104             << '\n';
105   for (auto i = 1; i < world_size; ++i)
106   {
107     std::vector<int> buf(1);
108     MPI_Send(buf.data(), 0, MPI_INT, i, TAG::STOP, MPI_COMM_WORLD);
109   }
110 }
111
112 void slave(char** argv)
113 {
114   int rank, world_size;
115   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
116   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
117
118   int IMAGE_HIGHT = std::stoi(argv[1]);
119   int IMAGE_WIDTH = std::stoi(argv[2]);
120   // int IMAGE_SIZE = IMAGE_WIDTH * IMAGE_HIGHT;
121   const int MAX_ITERS = std::stoi(argv[3]);
122   const double X_MIN = std::stod(argv[4]);
123   const double X_MAX = std::stod(argv[5]);
124   const double Y_MAX = std::stod(argv[6]);
125   const double Y_MIN =
126     Y_MAX - (X_MAX - X_MIN) * ((double)(IMAGE_HIGHT)) / IMAGE_WIDTH;
127
128   MPI_Status stat;
129   std::vector<int> line(1);
130   std::vector<int> imagebuf;
131   imagebuf.resize(IMAGE_WIDTH);
132
133   auto ct = 0;
134   do
135   {
136     ++ct;
137     MPI_Recv(line.data(), 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
          stat);
138     switch (stat.MPI_TAG)
139     {
140     case TAG::RENDER:
141
142       render_row(imagebuf,
143                  line[0],
144                  X_MIN,
145                  X_MAX,
146                  Y_MIN,
```

```
147                   Y_MAX,
148                   IMAGE_HIGHT,
149                   IMAGE_WIDTH,
150                   MAX_ITERS);
151
152       MPI_Send(
153         imagebuf.data(), IMAGE_WIDTH, MPI_INT, 0, line[0],
              MPI_COMM_WORLD);
154
155       break;
156     }
157
158   } while (stat.MPI_TAG != TAG::STOP);
159   std::cout << rank << " finished, completed " << ct << " lines of the
        image\n";
160 }
```

0.3 calculator.cpp

```
1  #include <vector>
2
3  /**
4   * Linear interpolation
5   * Used to determine the complex coordinate from the pixel location
6   *
7   * @param i    The pixel x or y
8   * @param n    The width or height of the image
9   * @param min  The X or Y min
10  * @param max  The X or Y max
11  * @return     The result of the linear interpolation
12  */
13 inline double interpolate(const int i,
14                           const double n,
15                           const double min,
16                           const double max)
17 {
18   return (i * (max - min) / n) + min;
19 }
20
21 /**
22  * Calculates the number of iterations it takes for any pixel
23  * in the image to diverge
24  *
25  * @param i           The x value of the pixel
26  * @param j           The y value of the pixel
27  * @param IMAGE_WIDTH The width in pixels of the image
28  * @param IMAGE_HIGHT The height in pixels of the image
```

```
29   * @param MAX_ITERS   The maximum number of iterations to attempt
30   * @param X_MIN       The minimum real (x) value of the image
31   * @param X_MAX       The maximum real (x) value of the image
32   * @param Y_MIN       The minimum imaginary (y) value of the image
33   * @param Y_MAX       The maximum imaginary (y) value of the image
34   * @return            The number of iterations for the specified pixel
35   */
36  int mandelbrot(const int i,
37                 const int j,
38                 const int IMAGE_WIDTH,
39                 const int IMAGE_HIGHT,
40                 const int MAX_ITERS,
41                 const double X_MIN,
42                 const double X_MAX,
43                 const double Y_MIN,
44                 const double Y_MAX)
45  {
46
47    double xtemp;
48    double x0 = interpolate(j, IMAGE_WIDTH, X_MIN, X_MAX);
49    double y0 = interpolate(i, IMAGE_HIGHT, Y_MIN, Y_MAX);
50    double x = 0.0;
51    double y = 0.0;
52    int iters = 0;
53
54    while (x * x + y * y < 4 && iters < MAX_ITERS)
55    {
56      xtemp = x * x - y * y + x0;
57      y = 2 * x * y + y0;
58      x = xtemp;
59      ++iters;
60    }
61
62    return iters;
63  }
64
65  /**
66   * Renders the Mandelbrot set
67   *
68   * @param X_MIN       The minimum real (x) value of the image
69   * @param X_MAX       The maximum real (x) value of the image
70   * @param Y_MIN       The minimum imaginary (y) value of the image
71   * @param Y_MAX       The maximum imaginary (y) value of the image
72   * @param IMAGE_HIGHT The height in pixels of the image
73   * @param IMAGE_WIDTH The width in pixels of the image
74   * @param MAX_ITERS   The maximum number of iterations to attempt
```

```
 * @return All the iteration data
 */
std::vector<int> render(double X_MIN,
                        double X_MAX,
                        double Y_MIN,
                        double Y_MAX,
                        int IMAGE_HIGHT,
                        int IMAGE_WIDTH,
                        int MAX_ITERS)
{
  std::vector<int> imagebuf;
  imagebuf.reserve(IMAGE_HIGHT * IMAGE_WIDTH);
  for (int i = 0; i < IMAGE_HIGHT; ++i)
  {
    for (int j = 0; j < IMAGE_WIDTH; ++j)
    {
      imagebuf.push_back(mandelbrot(
        i, j, IMAGE_WIDTH, IMAGE_HIGHT, MAX_ITERS, X_MIN, X_MAX, Y_MIN,
            Y_MAX));
    }
  }

  return imagebuf;
}

void render_row(std::vector<int>& imagebuf,
                int row,
                double X_MIN,
                double X_MAX,
                double Y_MIN,
                double Y_MAX,
                int IMAGE_HIGHT,
                int IMAGE_WIDTH,
                int MAX_ITERS)
{
  for (int j = 0; j < IMAGE_WIDTH; ++j)
  {
    imagebuf[j] = (mandelbrot(
      row, j, IMAGE_WIDTH, IMAGE_HIGHT, MAX_ITERS, X_MIN, X_MAX, Y_MIN,
          Y_MAX));
  }
}
```

0.4 color.cpp

```
#include "color.hpp"
#include <cmath>
```

```
3  #include <tuple>
4
5  /**
6   * A simple gray scale color scheme
7   *
8   * @param iters     The number of iterations to turn into a color
9   * @param max_iters The maximum number of iterations
10  * @return          A tuple containing the [R,G,B] triple for that
        iteration
11  */
12 std::tuple<int, int, int> color_scheme_0(int iters, int max_iters)
13 {
14   if (iters == max_iters)
15   {
16     return {0, 0, 0};
17   }
18
19   auto c = log(iters) / log(max_iters) * 255;
20
21   return {c, c, c};
22 }
23
24 /**
25  * A simple linear gradient color scheme
26  *
27  * @param iters     The number of iterations to turn into a color
28  * @param max_iters The maximum number of iterations
29  * @return          A tuple containing the [R,G,B] triple for that
        iteration
30  */
31 std::tuple<int, int, int> color_scheme_1(int iters, int max_iters)
32 {
33   if (iters == max_iters)
34   {
35     return {0, 0, 0};
36   }
37
38   int r, g, b;
39
40   r = 100 + log(max_iters / iters) * 155;
41   g = log(max_iters / iters) * 255;
42   b = 50 + log(max_iters / iters) * 205;
43
44   return {r, g, b};
45 }
46
```

```
47 /**
48  * A simple logarithmic gradient color scheme
49  *
50  * @param iters     The number of iterations to turn into a color
51  * @param max_iters The maximum number of iterations
52  * @return          A tuple containing the [R,G,B] triple for that
        iteration
53  */
54 std::tuple<int, int, int> color_scheme_2(int iters, int max_iters)
55 {
56   if (iters == max_iters)
57   {
58     return {0, 0, 0};
59   }
60
61   int r, g, b;
62
63   r = log(max_iters / iters) * 255;
64   // r = log(iters) / log(max_iters) * 255;
65   g = 0;
66   b = 0;
67
68   return {r, g, b};
69 }
```

0.5 ppmToBmp.hpp

```
1  #ifndef WRITE_PNG_HPP
2  #define WRITE_PNG_HPP
3
4  #include <png.h>
5  #include <string>
6
7  bool save_png_libpng(const std::string filename, uint8_t* pixels, int w
       , int h)
8  {
9    png_structp png =
10     png_create_write_struct(PNG_LIBPNG_VER_STRING, nullptr, nullptr,
           nullptr);
11   if (!png)
12   {
13     return false;
14   }
15
16   png_infop info = png_create_info_struct(png);
17   if (!info)
18   {
```

```
19      png_destroy_write_struct(&png, &info);
20      return false;
21    }
22
23    FILE* fp = fopen(filename.c_str(), "wb");
24    if (!fp)
25    {
26      png_destroy_write_struct(&png, &info);
27      return false;
28    }
29
30    png_init_io(png, fp);
31    png_set_IHDR(png,
32                 info,
33                 w,
34                 h,
35                 8 /* depth */,
36                 PNG_COLOR_TYPE_RGB,
37                 PNG_INTERLACE_NONE,
38                 PNG_COMPRESSION_TYPE_BASE,
39                 PNG_FILTER_TYPE_BASE);
40    png_colorp palette =
41      (png_colorp)png_malloc(png, PNG_MAX_PALETTE_LENGTH * sizeof(
          png_color));
42    if (!palette)
43    {
44      fclose(fp);
45      png_destroy_write_struct(&png, &info);
46      return false;
47    }
48    png_set_PLTE(png, info, palette, PNG_MAX_PALETTE_LENGTH);
49    png_write_info(png, info);
50    png_set_packing(png);
51
52    png_bytepp rows = (png_bytepp)png_malloc(png, h * sizeof(png_bytep));
53    for (int i = 0; i < h; ++i)
54    {
55      rows[i] = (png_bytep)(pixels + (h - i) * w * 3);
56    }
57
58    png_write_image(png, rows);
59    png_write_end(png, info);
60    png_free(png, palette);
61    png_destroy_write_struct(&png, &info);
62
63    fclose(fp);
```

```
64    delete[] rows;
65    return true;
66  }
67
68  #endif
```

Output

```
# mpic++ -std=c++17 -g0 -O3 -Wall -Wextra -Werror main.cpp calculator.
    cpp color.cpp -o mandelbrot.out

# mpiexec -n 4 ./mandelbrot.out 2048 2048 1000 -.760574 -.762574
    -.0837596

Successfully wrote brot2048.png
2048 x 2048
Time to compute: 1.5373
Time to write image: 0.690633


3 finished, completed 683 lines of the image
2 finished, completed 684 lines of the image
1 finished, completed 684 lines of the image
```

Findings

I generated the image below, Figure 5, as a 256x256, 512x512, 1024x2014, 2048x2048, 4096x4096, and 8192x8192 pixel image. I ran each size 10 times and took the average time to calculate the number of iterations for each pixel and the time to write the file to the disk. The results are detailed in Figure 1. The graph shows that the time to generate an image increases with the square of the number of pixels. The same is true for the writing of the file to the disk. Another interesting metric can be seen in Figure: 2 which shows how the pixels per second calculated was not largely affected by the image size. You can however see that it slowly increases with larger images. I believe this is due to caching. The last metric observed is the speedup of the parallel version compared to the serial version. This relationship can be seen in Figure 3. It can be observed that the speedup levels off with the size of the image. I think that the speedup is lower at first because the image is so small that sending messages is more overhead than it is worth. Then you get more speedup with larger images because there are so many more pixels.

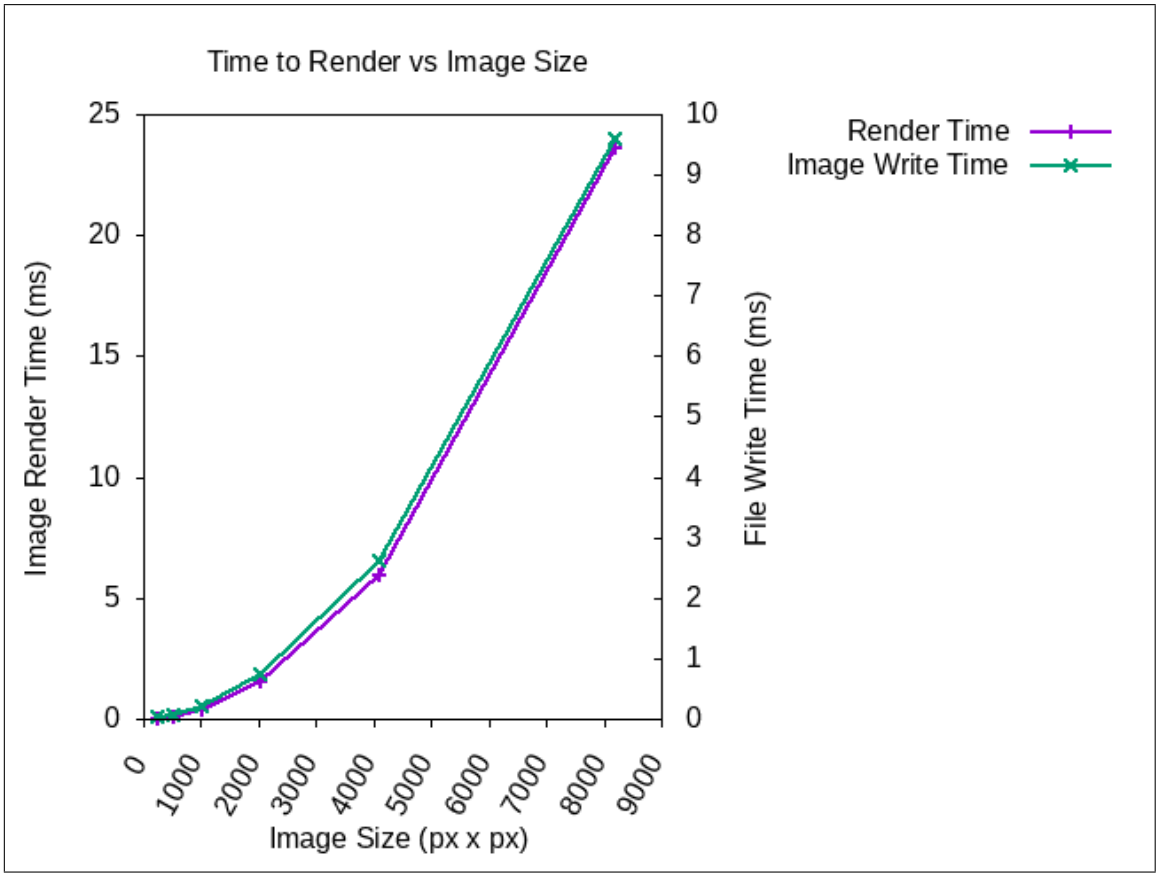
Time to Render vs Image Size
Image Render Time (ms)
File Write Time (ms)
Image Size (px x px)
Render Time
Image Write Time
0
5
10
15
20
25
0
1
2
3
4
5
6
7
8
9
10
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Figure 1:

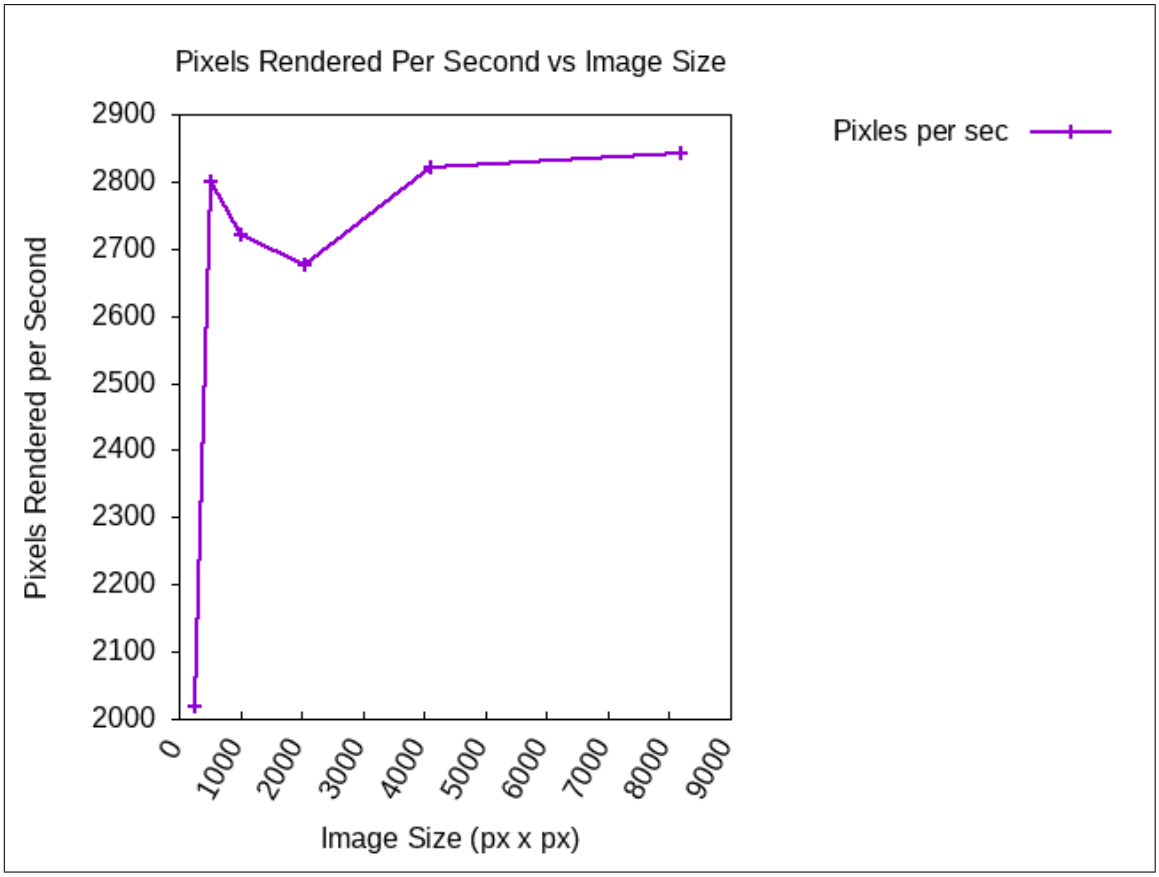
Pixels Rendered Per Second vs Image Size
Pixels Rendered per Second
2900
2800
2700
2600
2500
2400
2300
2200
2100
2000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Image Size (px x px)
Pixles per sec

Figure 2:

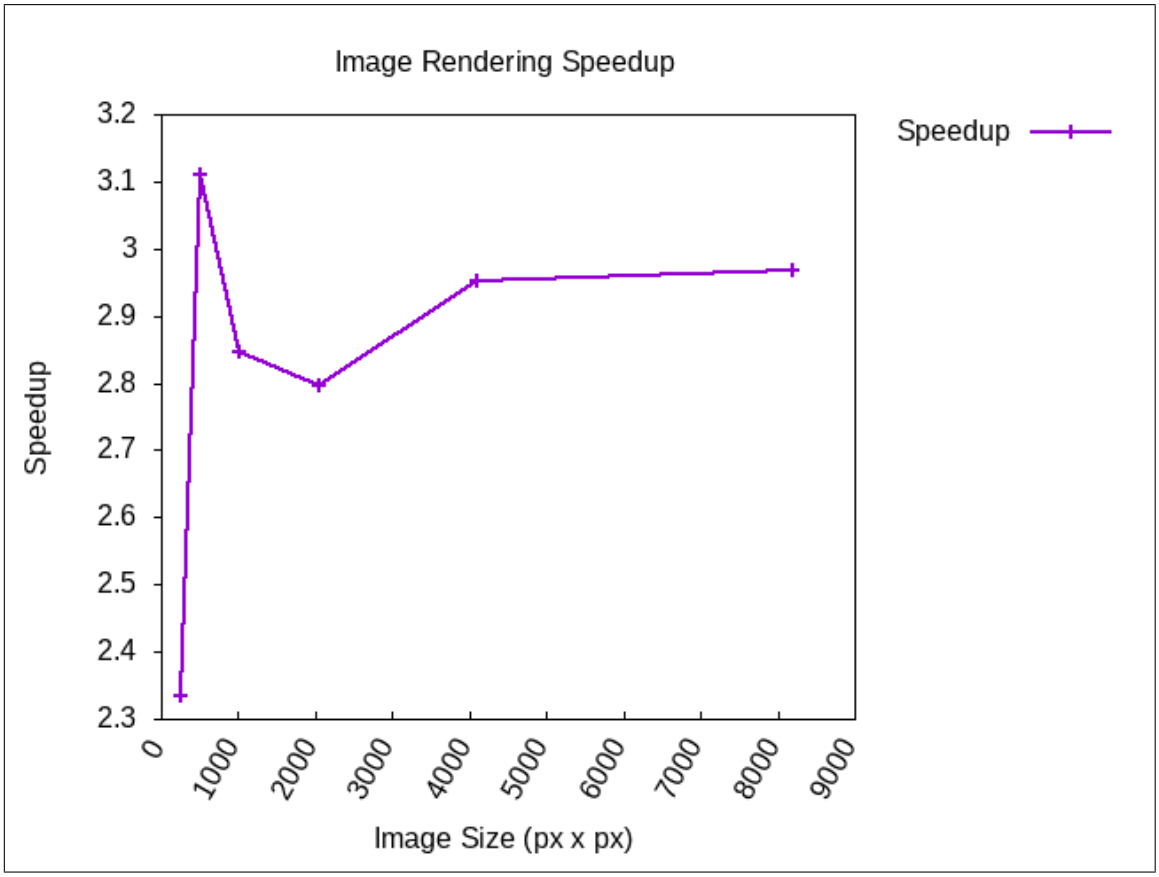
Image Rendering Speedup
Speedup
3.2
3.1
3
2.9
2.8
2.7
2.6
2.5
2.4
2.3
Speedup
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Image Size (px x px)

Figure 3:

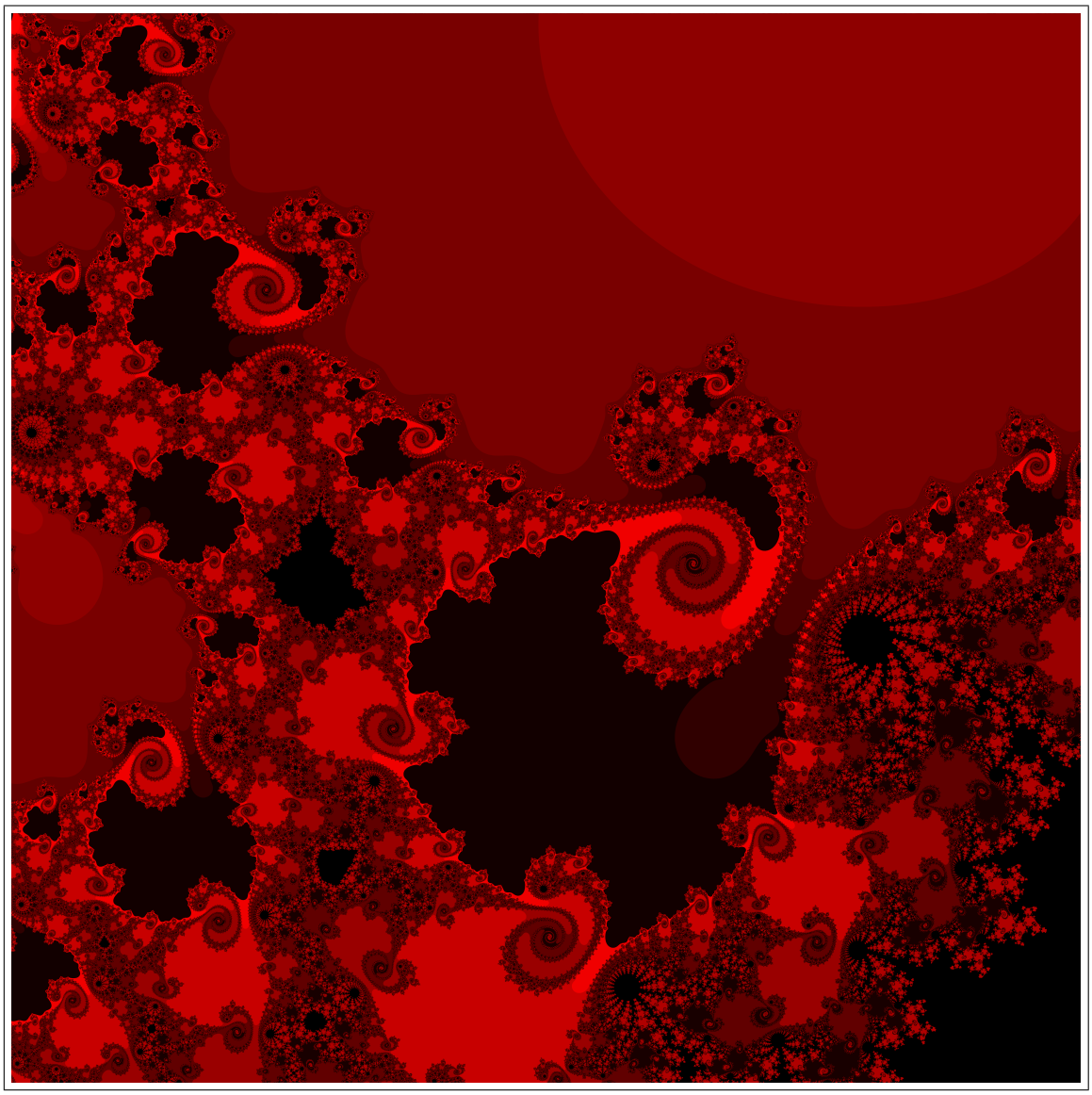

Figure 4:

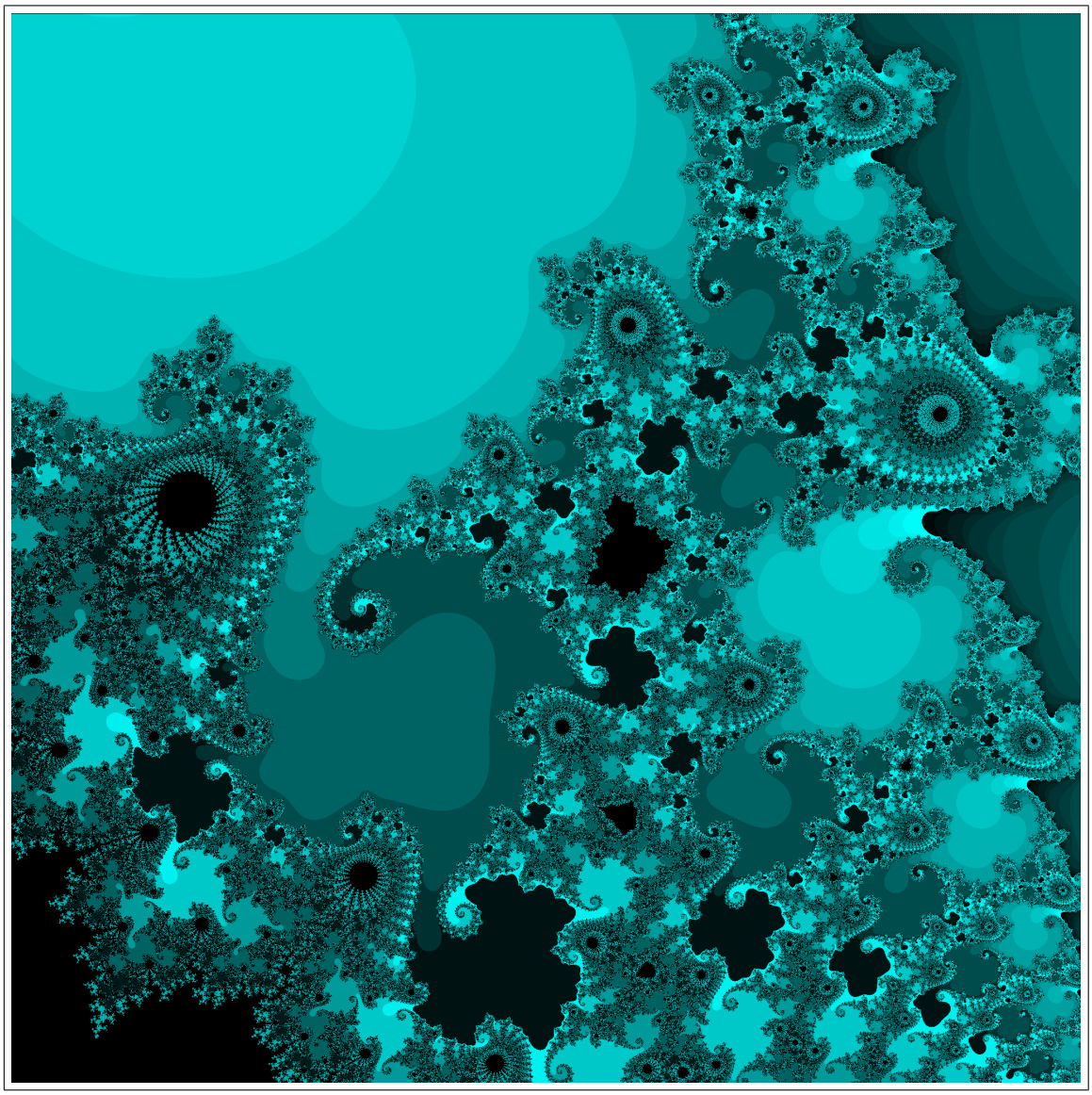

Figure 5:

Figure 6: