HW 5
Serial Mandelbrot

Philip Nelson

2018 October 1

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. It then calculates the number if iterations every pixel in the image takes to diverge and stores those values in an array. When every pixel has been calculated, the array is converted to a bitmap image based on a color scheme. Functionality to output to a ppm file is included.

Code

The code is broken up into four main files, main.cpp, calculator.cpp, color.cpp, and ppmToBmp.hpp. The files are included below.

0.1 main.cpp

```
1  #include "calculator.hpp"
2  #include "color.hpp"
3  #include "ppmToBmp.hpp"
4  #include <algorithm>
5  #include <fstream>
6  #include <functional>
7  #include <iostream>
8  #include <mpi.h>
9  #include <vector>
10
11 /**
12  * Renders the Mandelbrot set
13  *
```

```
14   * @param X_MIN       The minimum real (x) value of the image
15   * @param X_MAX       The maximum real (x) value of the image
16   * @param Y_MIN       The minimum imaginary (y) value of the image
17   * @param Y_MAX       The maximum imaginary (y) value of the image
18   * @param IMAGE_HIGHT The height in pixels of the image
19   * @param IMAGE_WIDTH The width in pixels of the image
20   * @param MAX_ITERS   The maximum number of iterations to attempt
21   * @return All the iteration data
22   */
23  std::vector<int> render(double X_MIN,
24                          double X_MAX,
25                          double Y_MIN,
26                          double Y_MAX,
27                          int IMAGE_HIGHT,
28                          int IMAGE_WIDTH,
29                          int MAX_ITERS)
30  {
31    std::vector<int> imagebuf;
32    imagebuf.reserve(IMAGE_HIGHT * IMAGE_WIDTH);
33    for (int i = 0; i < IMAGE_HIGHT; ++i)
34    {
35      for (int j = 0; j < IMAGE_WIDTH; ++j)
36      {
37        imagebuf.push_back(mandelbrot(
38          i, j, IMAGE_WIDTH, IMAGE_HIGHT, MAX_ITERS, X_MIN, X_MAX, Y_MIN,
                Y_MAX));
39      }
40    }
41
42    return imagebuf;
43  }
44
45  /**
46   * Write the iteration data as a ppm image
47   *
48   * @param imagebuf The array containing all the iteration data
49   * @param width    The width of the image in pixels
50   * @param height   The height of the image in pixels
51   * @param filename The name of the ppm image
52   */
53  void write_ppm_image(std::vector<int> const& imagebuf,
54                       const int width,
55                       const int height,
56                       std::string fileName)
57  {
58    std::ofstream fout(fileName);
```

```
59    fout << "P3\n" << width << " " << height << "\n255\n";
60    std::for_each(begin(imagebuf), end(imagebuf), [&fout](int num) {
61      fout << num << " " << num << " " << num << " ";
62    });
63    fout << std::endl;
64  }
65
66  int main(int argc, char** argv)
67  {
68    MPI_Init(&argc, &argv);
69
70    int rank, size;
71    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
72    MPI_Comm_size(MPI_COMM_WORLD, &size);
73
74    if (0 == rank)
75    {
76      if (argc < 5)
77      {
78        std::cerr << "incorrect number of arguments\n"
79                  << "mandelbrot HEIGHT WIDTH ITERATIONS X_MIN X_MAX
                        Y_MIN\n";
80        MPI_Finalize();
81        exit(EXIT_FAILURE);
82      }
83      int IMAGE_HIGHT = std::stoi(argv[1]);
84      int IMAGE_WIDTH = std::stoi(argv[2]);
85      int MAX_ITERS = std::stoi(argv[3]);
86      double X_MIN = std::stod(argv[4]);
87      double X_MAX = std::stod(argv[5]);
88      double Y_MAX = std::stod(argv[6]);
89      double Y_MIN =
90        Y_MAX - (X_MAX - X_MIN) * ((double)(IMAGE_HIGHT)) / IMAGE_WIDTH;
91
92      std::vector<int> imagebuf;
93
94      auto t1 = MPI_Wtime();
95
96      imagebuf =
97        render(X_MIN, X_MAX, Y_MIN, Y_MAX, IMAGE_HIGHT, IMAGE_WIDTH,
             MAX_ITERS);
98
99      auto t2 = MPI_Wtime();
100
101     ppmToBmp(imagebuf,
102              IMAGE_WIDTH,
```

```
103               IMAGE_HIGHT,
104               std::bind(color_scheme_2, std::placeholders::_1, MAX_ITERS
                     ),
105               "brot.bmp");
106
107     auto t3 = MPI_Wtime();
108
109     std::cout << IMAGE_HIGHT << " x " << IMAGE_WIDTH << '\n'
110               << "Time to compute: " << (t2 - t1) << '\n'
111               << "Time to write image: " << t3 - t2 << '\n'
112               << "Total time: " << t3 - t1 << '\n'
113               << '\n';
114   }
115
116   MPI_Finalize();
117
118   return (EXIT_SUCCESS);
119 }
```

0.2 calculator.cpp

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

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

0.3 color.cpp

```
1  #include "color.hpp"
2  #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 = 55 + (double)iters / max_iters * 200;
41    g = 200 + (double)iters / max_iters * 55;
42    b = 100 + (double)iters / max_iters * 155;
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
```

```
 */
std::tuple<int, int, int> color_scheme_2(int iters, int max_iters)
{
  if (iters == max_iters)
  {
    return {0, 0, 0};
  }

  int r, g, b;

  r = log(max_iters / iters) * 255;
  // r = log(iters) / log(max_iters) * 255;
  g = 0;
  b = 0;

  return {r, g, b};
}
```

0.4 ppmToBmp.hpp

```
#ifndef PPM_TO_BMP_HPP
#define PPM_TO_BMP_HPP

#include <cstdint>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

/**
 * @author Bryan Hansen
 * @author Erik Falor
 * @author Philip Nelson
 * @date 10/9/2017
 *
 * @history
 * 10/16/17  Fixed padding in generated BMP for non-word aligned sizes
 * 10/31/17  Fixed BMP files store rows of pixels from bottom-to-top
 * 09/26/18  Now converts a vector of iteration data
 *             instead of reading from a file. A color_scheme function
    is
 *             passed in to turn the raw iterations into [r,g,b] colors
 */

namespace stayOffMyLawn
{

```

```
27  /**
28   * Writes the standard BMP header to the provided file
29   * @param bmpFile File stream to which the header will be written
30   * @param width   The width of the PPM file in pixels
31   * @param height  The height of the PPM file in pixels
32   */
33  void writeBmpHeader(std::ofstream& bmpFile, int width, int height)
34  {
35    // BMP header (14 bytes)
36    // A two character signature to indicate the file is a bitmap file
37    // (typically    BM   ). A 32bit unsigned little-endian integer
          representing the
38    // size of the file itself. A pair of 16bit unsigned little-endian
          integers
39    // reserved for application specific uses. A 32bit unsigned little-
          endian
40    // integer representing the offset to where the pixel array starts
          in the
41    // file.
42
43    const uint32_t HEADER_SIZE_BYTES = 54;
44    const uint32_t BYTES_PER_PIXEL = 3;
45    uint32_t padBytes = 0;
46    if ((width * BYTES_PER_PIXEL) % sizeof(uint32_t) != 0)
47    {
48      padBytes =
49        sizeof(uint32_t) - ((width * BYTES_PER_PIXEL) % sizeof(uint32_t
              ));
50    }
51
52    const uint32_t paddedWidthBytes = (width * BYTES_PER_PIXEL) +
          padBytes;
53    const uint32_t totalSize = HEADER_SIZE_BYTES + (height *
          paddedWidthBytes);
54    const char sigOne = 'B';
55    const char sigTwo = 'M';
56    const uint16_t reserved = 0;
57    const uint32_t pixelOffset = HEADER_SIZE_BYTES;
58
59    /* clang-format off */
60    bmpFile.write(&sigOne, sizeof(uint8_t));
61    bmpFile.write(&sigTwo, sizeof(uint8_t));
62    bmpFile.write(
63        reinterpret_cast<const char*>(&totalSize), sizeof(uint32_t));
64    bmpFile.write(
65        reinterpret_cast<const char*>(&reserved), sizeof(uint16_t));
```

```
66      bmpFile.write(
67          reinterpret_cast<const char*>(&reserved), sizeof(uint16_t));
68      bmpFile.write(
69        reinterpret_cast<const char*>(&pixelOffset), sizeof(uint32_t));
70      /* clang-format on */
71    }
72
73    /**
74     * Writes the BMP image header to the provided file
75     * @param bmpFile File stream to which image header will be written
76     * @param width   The width of the PPM file in pixels
77     * @param height  The height of the PPM file in pixels
78     */
79    void writeBmpImageHeader(std::ofstream& bmpFile, int width, int
          height)
80    {
81      // Image header (40 bytes)
82      // biSize           4 Header Size - Must be at least 40
83      // biWidth          4 Image width in pixels
84      // biHeight         4 Image height in pixels
85      // biPlanes         2 Must be 1
86      // biBitCount       2 Bits per pixel - 1, 4, 8, 16, 24, or 32
87      // biCompression    4 Compression type (0 = uncompressed)
88      // biSizeImage      4 Image Size - may be zero for uncompressed
            images
89      // biXPelsPerMeter  4 Preferred resolution in pixels per meter
90      // biYPelsPerMeter  4 Preferred resolution in pixels per meter
91      // biClrUsed        4 Number Color Map entries that are actually
            used
92      // biClrImportant   4 Number of significant colors
93
94      const uint32_t headerSizeBytes = 40;
95      const uint16_t planes = 1;
96      const uint16_t bitsPerPixel = 24;
97      const uint32_t compression = 0;
98      const uint32_t imageSize = 0;
99      const uint32_t preferredResolution = 0;
100     const uint32_t colorMapEntries = 0;
101     const uint32_t significantColors = 0;
102
103     /* clang-format off */
104     bmpFile.write(
105       reinterpret_cast<const char*>(&headerSizeBytes), sizeof(uint32_t)
            );
106     bmpFile.write(
107         reinterpret_cast<const char*>(&width), sizeof(uint32_t));
```

```
108       bmpFile.write(
109           reinterpret_cast<const char*>(&height), sizeof(uint32_t));
110       bmpFile.write(
111           reinterpret_cast<const char*>(&planes), sizeof(uint16_t));
112       bmpFile.write(
113         reinterpret_cast<const char*>(&bitsPerPixel), sizeof(uint16_t));
114       bmpFile.write(
115         reinterpret_cast<const char*>(&compression), sizeof(uint32_t));
116       bmpFile.write(
117           reinterpret_cast<const char*>(&imageSize), sizeof(uint32_t));
118       bmpFile.write(
119         reinterpret_cast<const char*>(&preferredResolution), sizeof(
              uint32_t));
120       bmpFile.write(
121         reinterpret_cast<const char*>(&preferredResolution), sizeof(
              uint32_t));
122       bmpFile.write(
123         reinterpret_cast<const char*>(&colorMapEntries), sizeof(uint32_t)
              );
124       bmpFile.write(
125         reinterpret_cast<const char*>(&significantColors), sizeof(
              uint32_t));
126       /* clang-format on */
127     }
128
129     /**
130      * Writes all pixels from the PPM file (ascii) into the BMP file (
           binary)
131      * @param ppmBuffer File stream from which ascii pixels will be read
132      * @param bmpFile   File stream to which binary pixels will be
           written
133      * @param width     The width of the PPM file in pixels
134      * @param height    The height of the PPM file in pixels
135      */
136     template <typename F>
137     bool writePixels(std::vector<int>& ppmBuffer,
138                      std::ofstream& bmpFile,
139                      int width,
140                      int height,
141                      F color_scheme)
142     {
143       // Write pixels to BMP file (24 bits per pixel), padding each row
           to be
144       // 4-byte divisible The BMP image is stored bottom-to-top, so we
           have to
145       // wrote the rows backwards relative to the PPM image
```

```
146
147     char** map = new char*[height];
148     const uint32_t BYTES_PER_PIXEL = 3;
149     uint32_t padBytes = 0;
150     if ((width * BYTES_PER_PIXEL) % sizeof(uint32_t) != 0)
151       padBytes =
152         sizeof(uint32_t) - ((width * BYTES_PER_PIXEL) % sizeof(uint32_t
              ));
153
154     // Copy the top of the PPM into the bottom of the bitmap
155     auto ppmIt = begin(ppmBuffer);
156     for (int row = height - 1; row >= 0; --row)
157     {
158       map[row] = new char[width * BYTES_PER_PIXEL + padBytes];
159       auto col = 0u;
160       for (; col < width * BYTES_PER_PIXEL; col += 3)
161       {
162         auto [red, green, blue] = color_scheme(*ppmIt++);
163
164         map[row][col + 0] = (char)blue;
165         map[row][col + 1] = (char)green;
166         map[row][col + 2] = (char)red;
167       }
168
169       // Pad if needed
170       const uint8_t padData = 0x00;
171       for (auto pad = 0u; pad < padBytes; ++pad)
172         map[row][col++] = (char)padData;
173     }
174
175     // Write the bitmap out to the bmpFile
176     for (int row = 0; row < height; ++row)
177     {
178       bmpFile.write(map[row], width * BYTES_PER_PIXEL + padBytes);
179       delete[] map[row];
180     }
181
182     delete[] map;
183
184     return true;
185   }
186 } // namespace stayOffMyLawn
187
188 /**
189  * Program converts an vector of iteration data into a 24-bit BMP file
190  * @param ppmBuffer    buffer of pixel information
```

```
 * @param ppmWidth     width of the image in pixels
 * @param ppmHeight    height of the image in pixels
 * @param color_scheme std::tuple<int, int, int>(int) function
 * @param bmpFileName  name of the bmp image to write
 * @return true on success, false on failure
 */
template <typename F>
bool ppmToBmp(std::vector<int> ppmBuffer,
              uint32_t ppmWidth,
              uint32_t ppmHeight,
              F color_scheme,
              std::string bmpFileName)
{
  std::cout << "Writing " << bmpFileName << "...\n";

  // Read out PPM header to get size information
  std::ofstream bmpFile(bmpFileName.c_str(), std::ios::binary);

  if (ppmBuffer.size() == ppmWidth * ppmHeight)
  {
    stayOffMyLawn::writeBmpHeader(bmpFile, ppmWidth, ppmHeight);
    stayOffMyLawn::writeBmpImageHeader(bmpFile, ppmWidth, ppmHeight);

    if (stayOffMyLawn::writePixels(
          ppmBuffer, bmpFile, ppmWidth, ppmHeight, color_scheme))
    {
      std::cout << "Success!" << std::endl;
      return true;
    }
  }

  bmpFile.close();

  return false;
}

#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

Writing brot.bmp...
Success!
2048 x 2048
Time to compute: 4.26297
Time to write image: 0.244798
Total time: 4.50777
```

Findings

I generated the image below, Figure 3, 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.

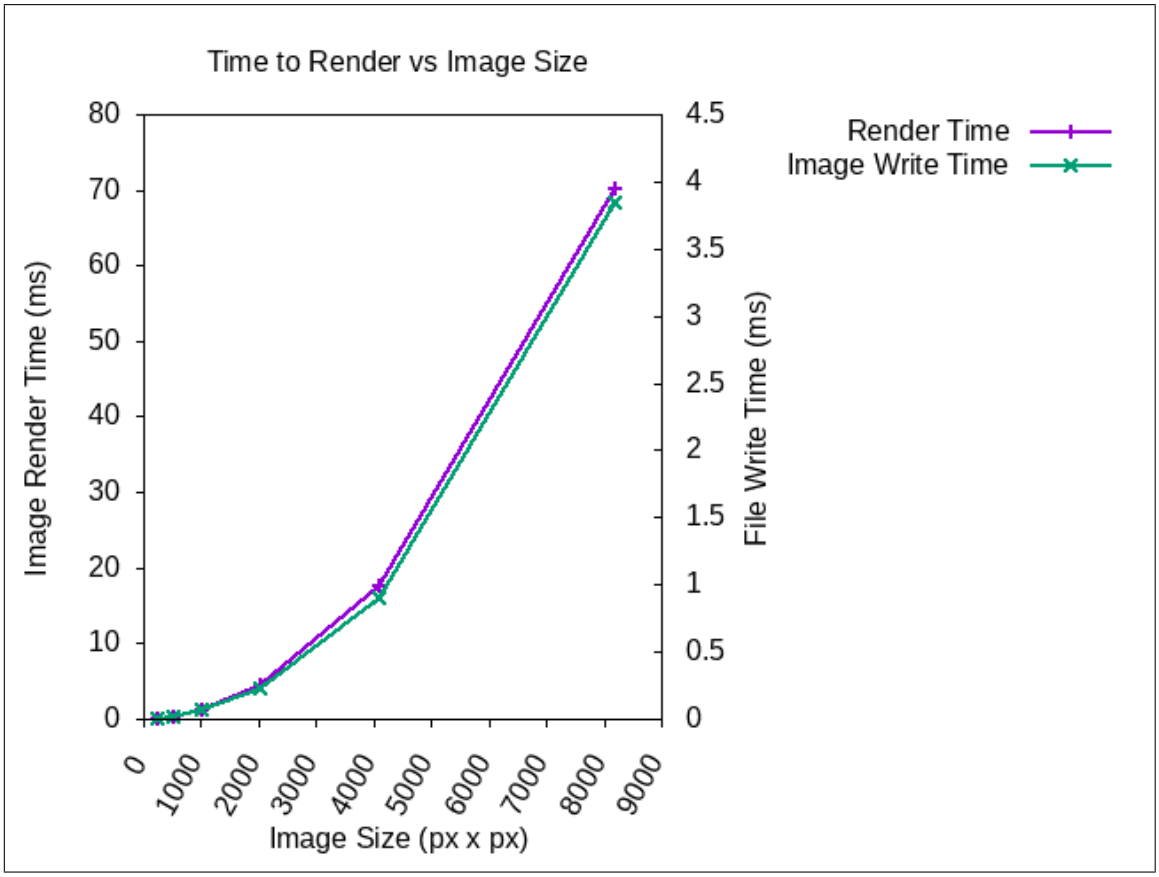
Time to Render vs Image Size
Image Render Time (ms)
File Write Time (ms)
Image Size (px x px)
Render Time
Image Write Time
0
10
20
30
40
50
60
70
80
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Figure 1:

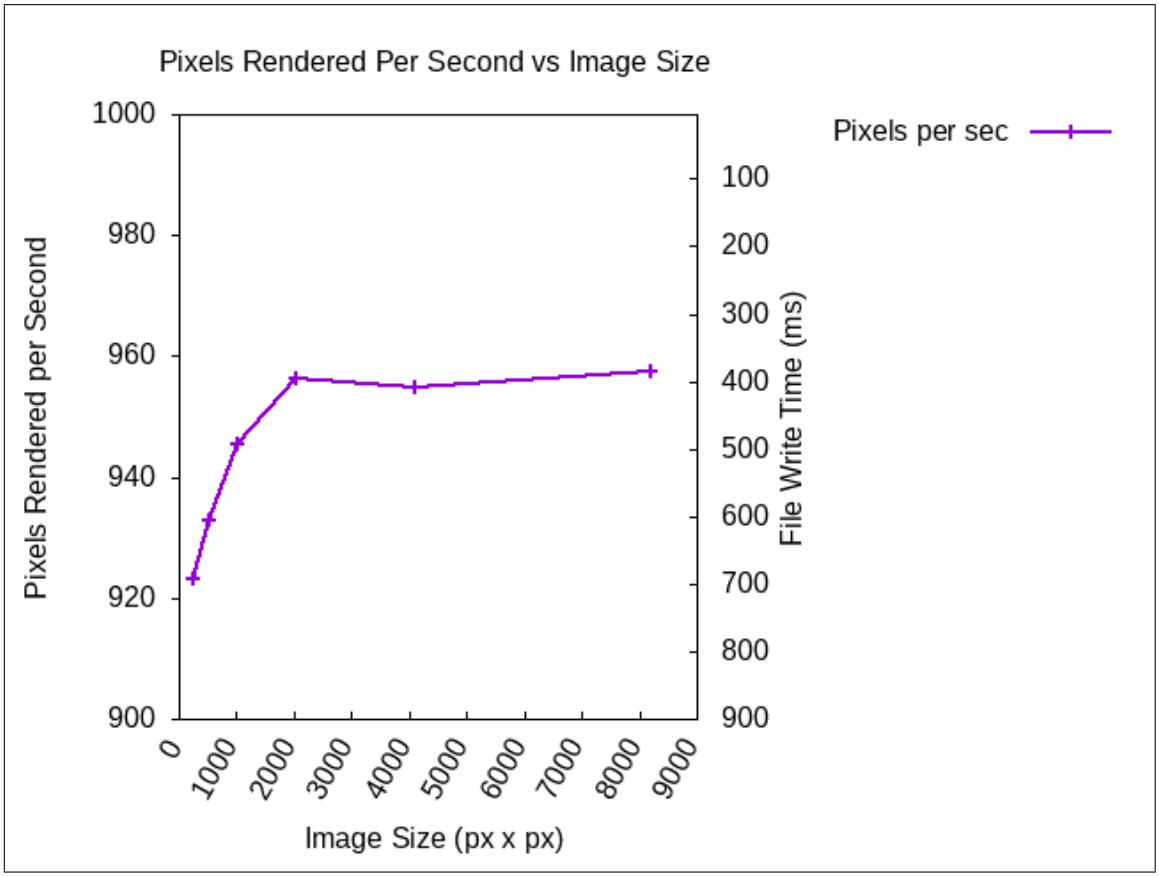
Pixels Rendered Per Second vs Image Size
Pixels per sec
Pixels Rendered per Second
File Write Time (ms)
Image Size (px x px)
1000
980
960
940
920
900
100
200
300
400
500
600
700
800
900
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Figure 2:

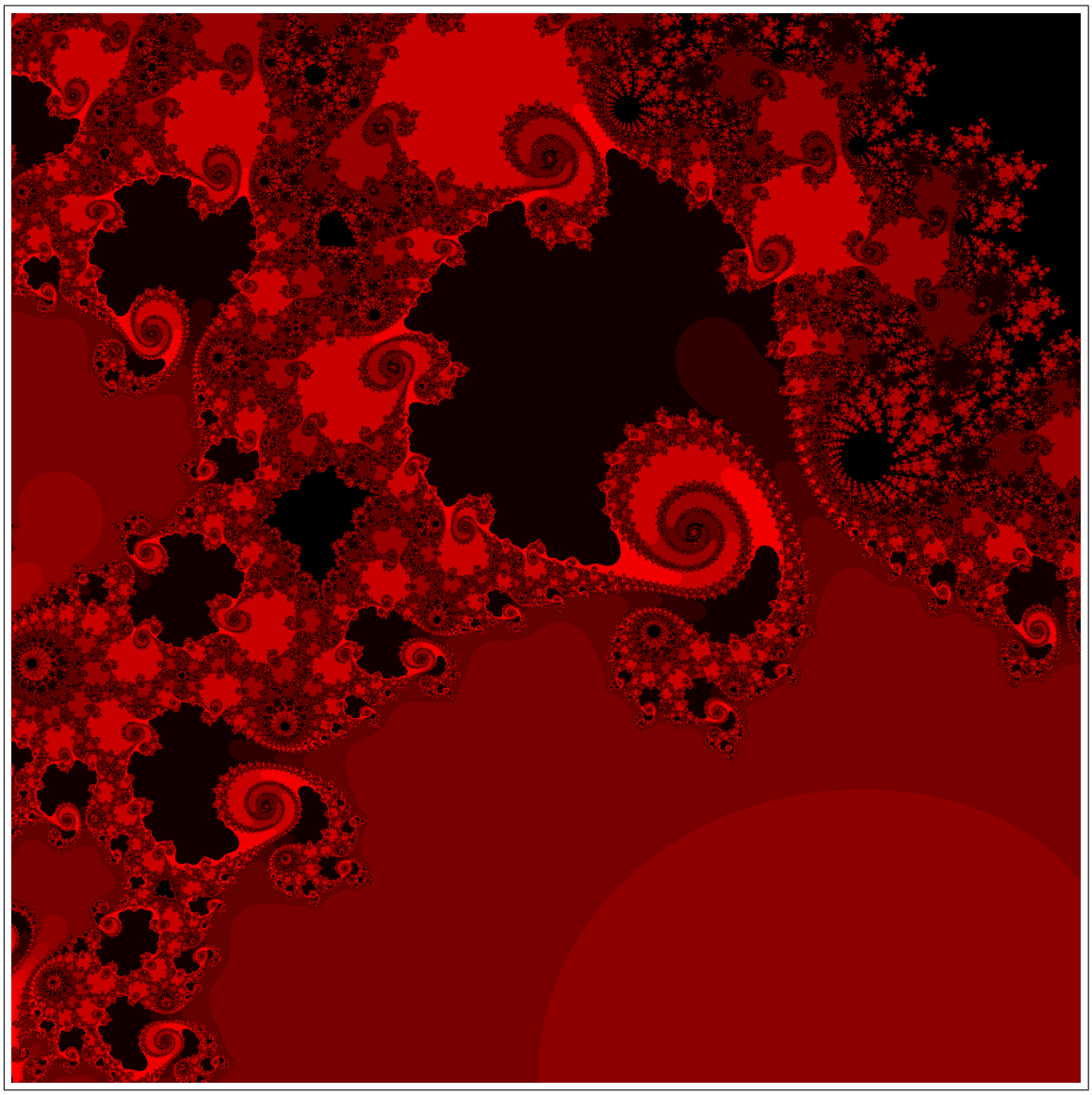

Figure 3: