Threaded Mandelbrot

1. Objective

To extend the Mandelbrot program to use multiple threads to speed up computation.

2. Proposition

The Mandelbrot program in sequential form has been authored by the student. A performance test of the same Mandelbrot program in sequential form will be compared against the parallel form that utilizes multiple threads.

It is proposed that the optimal number of threads is equal to the number of cores on the device CPU. Thus, running more threads than the number of available CPU cores is projected to result in a gradual decrease in performance.

3. Results

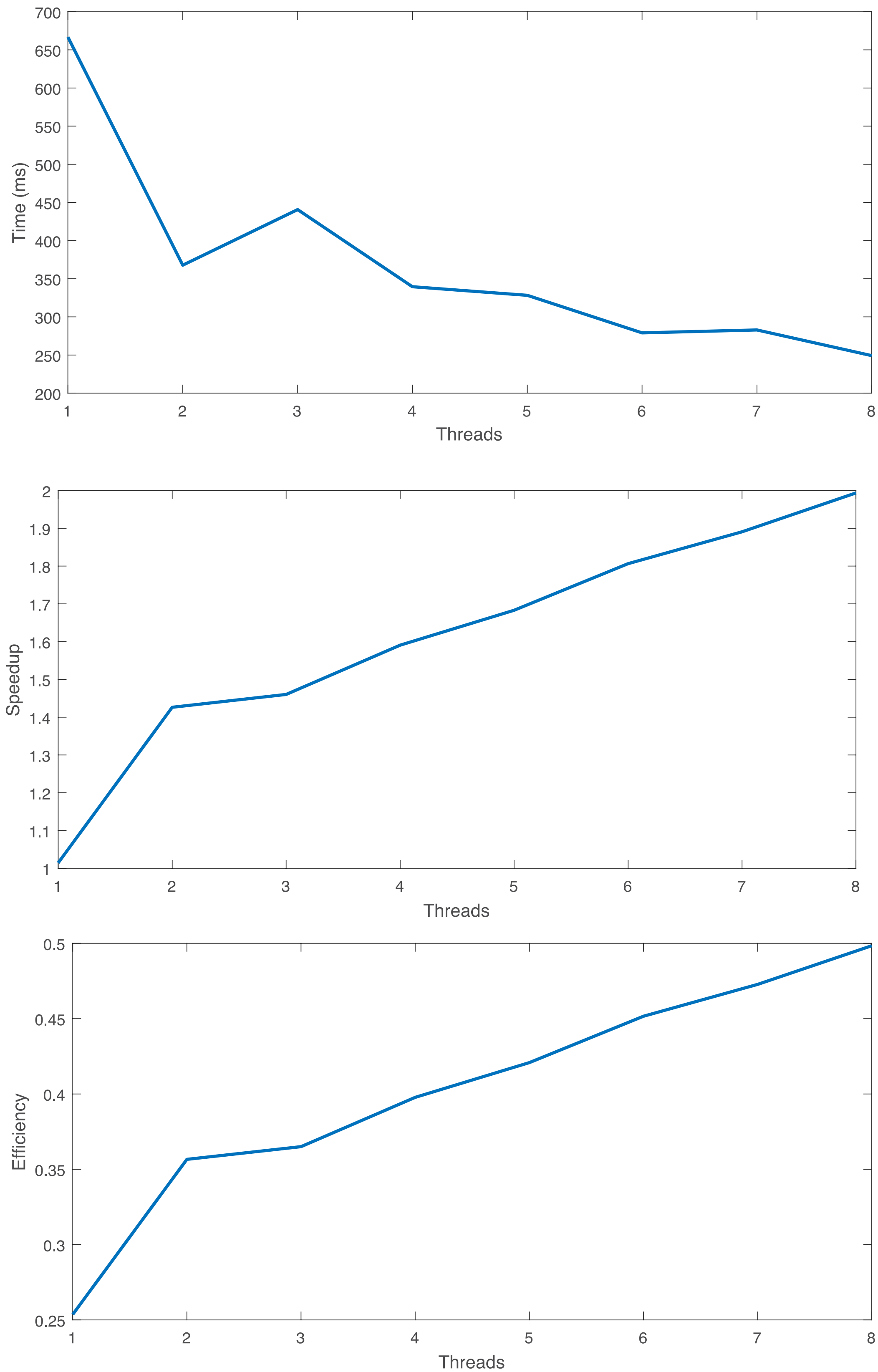
- Device: Raspberry Pi 3 – Model B
- CPU: 1.2GHz 64-bit quad-core ARMv8

- Image Size: 1000 x 571 pixels
- Samples: 10 per thread test

The performance test of the Mandelbrot program was executed for every thread count from 1 to 8 threads. For each thread test, a number of sample program timing runs were completed and average execution times, speedups, and efficiency results were recorded.

$$\text{speedup} = \frac{\text{serial time}}{\text{parallel time}} \qquad \text{efficiency} = \frac{\text{speedup}}{\text{cores}}$$

	Time (ms)		Speedup		Efficiency	
Threads	Average	Std. Dev.	Average	Std. Dev.	Average	Std. Dev.
1	667	2.08	1.01	0.02	0.25	0.00
2	368	22.62	1.43	0.42	0.36	0.11
3	441	15.51	1.46	0.35	0.37	0.09
4	340	12.63	1.59	0.38	0.40	0.09
5	328	14.38	1.68	0.39	0.42	0.10
6	279	21.27	1.81	0.45	0.45	0.11
7	283	25.59	1.89	0.48	0.47	0.12
8	249	18.84	1.99	0.53	0.50	0.13


Time (ms)
Threads
Speedup
Threads
Efficiency
Threads

4. Conclusion

The results show that overall performance in time, speedup, and efficiency substantially improves by nearly fifty-percent the moment that multithreading is used. However, the use of three threads shows a noticeable loss in overall performance. Then the performance mark resumes from the benchmark established by two threads and overall performance steadily improves when the number of threads increases.

Thus, the observed thread behavior is contrary to the proposition statement. The actual results show that overall performance improves substantially once multithreading is used, but continues to gradually improve with increased number of threads rather than gradually decreasing. However, an exception is observed at the mark of three threads that results in a loss of overall performance.

5. Appendix

5.1 Program Code

threaded-mandelbrot.cpp

```
// 01/28/2017 - CS 3100 - Meine, Joel
// Threaded MandelBrot

/* References
[1]: Mandelbrot Set _ https://en.wikipedia.org/wiki/Mandelbrot_set
[2]: C++ Make an Image _ https://www.youtube.com/watch?v=fbH005SzEMc
[3]: C++11 Multithreading Tutorial _ https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
[4]: Amdahl's Law _ http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html
*/

#include "math.h"
#include "time.h"

#include <fstream>
#include <iostream>
#include <thread>
using namespace std;

int scale; // mandelbrot-to-pixel scalar
int samples; // number of function execution samples

int width, height; // image width and height
const int max_iteration = 256; // max number of colors
const double x_min = -2.5, x_max = 1, y_min = -1, y_max = 1; // mandelbrot axis
const int bailout = 2 * 2; // mandelbrot bailout limit
double x_range = x_max - x_min, y_range = y_max - y_min; // mandelbrot axis ranges
double x_unit, y_unit; // pixel-to-mandelbrot point unit
vector<int> m_colors_s = {}; // mandelbrot point colors of image, serial
vector<vector<int>> m_colors_p = {}; // mandelbrot point colors of image, parallel

int t_limit; // maximum number of threads
vector<vector<int>> s_points_start; // top-left points of spliced images
vector<vector<int>> s_points_end; // bottom-right points of spliced images

double time_length_s; // timer sample, serial
double time_length_p; // timer sample, parallel
vector<double> times = {}; // timer samples
vector<double> averages_t = {}; // average times
vector<double> stdevs_t = {}; // standard deviations of times
vector<double> averages_s = {}; // average speedups
vector<double> stdevs_s = {}; // standard deviations of speedups
vector<double> averages_e = {}; // average efficiencies
vector<double> stdevs_e = {}; // standard deviations of efficiencies
vector<double> speedups = {}; // speedup results
vector<double> efficiencies = {}; // efficiency results
const int cores = 4; // number of CPU cores
```

```
// mandelbrot point color [1]
int m_color(double Px, double Py)
{
	double x0 = Px * x_unit - abs(x_min), y0 = Py * y_unit - abs(y_min); // pixel-to-mandelbrot scaled+shifted point
	double x = 0, y = 0; // initalize madelbrot point
	int iteration = 0; // initialize iteration
	double temp;
	// escape time algorithm
	while (x*x + y*y < bailout && iteration < max_iteration)
	{
		temp = x*x - y*y + x0;
		y = 2*x*y + y0;
		x = temp;
		iteration++;
	}
	return(iteration); // return color of mandelbrot point
}

// mandelbrot image, serial [1]
void m_image_s()
{
	for (int y = 0; y < height; y++) // every pixel of height
	{
		for (int x = 0; x < width; x++) // every pixel of width
			m_colors_s.push_back(m_color(x, y)); // save mandelbrot point color
	}
}

// mandelbrot image, parallel [1]
void m_image_p(int x, int y, int x_limit, int y_limit, int thread)
{
	int initial = x;
	vector<int> s_m_colors; // load empty pixel color list of image section
	for (y; y <= y_limit; y++) // every pixel of height
	{
		if (x == x_limit + 2) // is first pixel of next row detected?
			x = initial; // if yes, start from first pixel
		for (initial; x <= x_limit; x++) // every pixel of width
		{
			s_m_colors.push_back(m_color(x, y)); // save mandelbrot point color
			if (x == x_limit) // is last pixel of row detected?
				x = x_limit + 1; // if yes, go to next row
		}
	}
	m_colors_p[thread] = s_m_colors;
}

// write image, serial [2]
void w_image_s()
{
	cout << "writing serial mandelbrot image... ";
	ofstream img("m_image_s.ppm");
	img << "P3" << endl;
	img << width << " " << height << endl;
	img << max_iteration - 1 << endl; // number of colors
	int I = m_colors_s.size();
	int r, g, b; // initialize color channels
	for (int i = 0; i < I; i++) // every pixel in image
	{
		r = m_colors_s[i]; // set red color channel
		g = m_colors_s[i]; // set green color channel
		b = m_colors_s[i]; // set blue color channel
		img << r << " " << g << " " << b << endl;
	}
	cout << "DONE" << "\n" << endl;
}

// write image, parallel [2]
void w_image_p()
{
	cout << "writing parallel mandelbrot image... ";
	ofstream img("m_image_p.ppm");
	img << "P3" << endl;
	img << width << " " << height << endl;
	img << max_iteration - 1 << endl; // number of colors
	int I = m_colors_p.size(); // number of image sections
```

```
	int r, g, b; // initialize color channels
	for (int i = 0; i < I; i++) // every image section
	{
		int J = m_colors_p[i].size(); // number of pixels in image section
		for (int j = 0; j < J; j++) // every pixel in image section
		{
			r = m_colors_p[i][j]; // set red color channel
			g = m_colors_p[i][j]; // set green color channel
			b = m_colors_p[i][j]; // set blue color channel
			img << r << " " << g << " " << b << endl;
		}
	}
	cout << "DONE" << "\n" << endl;
}

// splice image
void s_image(int t_number)
{
	s_points_start = {}; s_points_end = {}; // clear spliced image points
	int s_unit = height / t_number; // height of each spliced image row
	for (int t = 0; t < t_number; t++) // every row of spliced image
	{
		s_points_start.push_back({ 0, t*s_unit }); // save top-left splice point
		if (t != t_number-1)
			s_points_end.push_back({ width-1, ((t+1)*s_unit)-1 }); // save bottom-right splice point
		else
			s_points_end.push_back({ width-1, height-1 }); // save last bottom-right splice point
	}
}

// run serially
void s_run()
{
	time_length_s = timer(true); // start timer
	m_image_s(); // create mandelbrot image serially
	time_length_s = timer(false); // end timer
}

// run threads [3]
void t_run(int t_number)
{
	time_length_p = timer(true); // start timer
	vector<thread> threads = {};
	// start the threads for each image section
	for (int i = 0; i < t_number; i++)
	{
		m_colors_p.push_back({}); // initialize empty colors list
		threads.push_back(thread(m_image_p, s_points_start[i][0], s_points_start[i][1], s_points_end[i][0],
s_points_end[i][1], i));
	}
	// check each thread and hold if not done
	for (int i = 0; i < t_number; i++)
		threads[i].join();
	time_length_p = timer(false); // end timer
}

// speedup formula [4]
double speedup(double time_length_s, double time_length_p)
{
	double T1 = time_length_s;
	double Tj = time_length_p;
	double S = T1 / Tj;
	return(S);
}

// efficiency formula [4]
double efficiency(double S, int C)
{
	double p = C;
	double E = S / p;
	return(E);
}

// run thread test
void t_test(int t_number)
{
	cout << "thread test " << t_number + 1 << " of " << t_limit << endl;
```

```
        cout << "process threads, number of = " << t_number + 1 << endl;
        s_image(t_number + 1); // splice image into sections
        int sample = 0;
        // create madelbrot images
        cout << "creating mandelbrot image samples... ";
        while (sample < samples)
        {
                s_run(); // create the mandelbrot image serially
                t_run(t_number + 1); // create the mandelbrot image using threads
                if (sample != samples - 1)
                {
                        // clear mandelbrot colors if not last sample
                        m_colors_s = {};
                        m_colors_p = {};
                }
                times.push_back(time_length_p); // save time to create image
                double s = speedup(time_length_s, time_length_p); // calculate speedup
                speedups.push_back(s); // save speedup result
                double e = efficiency(s, cores); // calculate efficiency
                efficiencies.push_back(e); // save efficiency result
                sample++; // increment to next sample
        }
        cout << "DONE" << endl;

        cout << "average, time = " << average(times) << " milliseconds" << endl;
        averages_t.push_back(average(times)); // save average times
        cout << "standard deviation, time = " << stdev(times) << " milliseconds" << endl;
        stdevs_t.push_back(stdev(times)); // save standard deviation of times

        cout << "average, speedup = " << average(speedups) << endl;
        averages_s.push_back(average(speedups)); // save average speedups
        cout << "standard deviation, speedup = " << stdev(speedups) << endl;
        stdevs_s.push_back(stdev(speedups)); // save standard deviation of speedups

        cout << "average, efficiency = " << average(efficiencies) << endl;
        averages_e.push_back(average(efficiencies)); // save average efficiencies
        cout << "standard deviation, efficiency = " << stdev(efficiencies) << endl;
        stdevs_e.push_back(stdev(efficiencies)); // save standard deviation of efficiencies

        times = {}; // clear times
        cout << endl;
}

// write performance report
void w_report()
{
        cout << "writing performance report... ";
        ofstream report("report.txt");
        if (report.is_open())
        {
                report << "image width, pixels = " << width << endl;
                report << "image height, pixels = " << height << endl;
                report << "process threads, max number of = " << t_limit << endl;
                report << "image samples, number of per test = " << samples << "\n" << endl;

                report << "t = time, execution (ms)" << endl;
                report << "s = speedup" << endl;
                report << "e = efficiency" << "\n" << endl;

                report << "threads average_t stdev_t average_s stdev_s average_e stdev_e" << "\n" << endl;
                for (int i = 0; i < t_limit; i++)
                        report << i+1 << " " << averages_t[i] << " " << stdevs_t[i] << " " << averages_s[i] << " " << stdevs_s[i]
<< " " << averages_e[i] << " " << stdevs_e[i] << endl;
                report.close();
        }
        cout << "DONE" << "\n" << endl;
}

int main()
{
        // ask user for image width input
        cout << "image width, pixels (min = 28): ";
        cin >> width; cout << endl;
        if (width < 28) // if width is too small
                width = 28; // then set to minimum
        double scale = width / x_range;
        height = y_range * scale;
```

```
        x_unit = x_range / width; y_unit = y_range / height;
        // ask user for number of threads
        cout << "process threads, max number of (min = 1): ";
        cin >> t_limit; cout << endl;
        if (t_limit < 1) // if not enough threads
                t_limit = 1; // then set to minimum
        // ask user for number of image samples
        cout << "image samples, number of per test (min = 1): ";
        cin >> samples; cout << endl;
        if (samples < 1) // if not enough samples
                samples = 1; // then set to minimum

        // run thread test
        for (int i = 0; i < t_limit; i++)
                t_test(i);
        w_report(); // write the performance report
        w_image_s(); // write the mandelbrot image, serial
        w_image_p(); // write the mandelbrot image, parallel

        return 0;
}
```

math.h

```
#include <vector>
#include <numeric>
#include <iostream>
#include <cmath>
using namespace std;

/* References
[1]: How to Find the Mean _ https://www.mathsisfun.com/mean.html
[2]: Standard Deviation Formulas _ https://www.mathsisfun.com/data/standard-deviation-formulas.html
[ ]: Calculate mean and standard deviation... _ http://stackoverflow.com/questions/7616511/calculate-mean-and-standard-deviation-from-
a-vector-of-samples-in-c-using-boos
*/

// average [1]
double average(vector<double> numbers)
{
        double sum = accumulate(numbers.begin(), numbers.end(), 0.0);
        double result = sum / numbers.size();
        return(result);
}

// standard deviation [2]
double stdev(vector<double> numbers)
{
        double mean = average(numbers);
        double minus, squared;
        vector<double> minus_squared = {};
        int I = numbers.size();
        for (int i = 0; i < I; i++)
        {
                minus = numbers[i] - mean;
                squared = pow(minus, 2);
                minus_squared.push_back(squared);
        }
        double sum = accumulate(minus_squared.begin(), minus_squared.end(), 0.0);
        double N = I;
        double result = sqrt((1 / N) * sum);
        return(result);
}
```

time.h

```
#include <chrono>
#include <iostream>
using namespace std;

/* References
[1]: Date and Time Utilities _ http://en.cppreference.com/w/cpp/chrono
*/
```

```
chrono::time_point<chrono::system_clock> timer_initial, timer_final; // initiate timer markers

// function timer [1]
double timer(bool begin)
{
	double result;
	if (begin == true) // start the timer?
	{
		timer_initial = chrono::system_clock::now(); // start timer
		result = 0;
	}
	else // end the timer?
	{
		timer_final = chrono::system_clock::now(); // end timer
		double elapsed_time = chrono::duration<double, milli>(timer_final - timer_initial).count(); // time length
		result = elapsed_time; // save time length in milliseconds
	}
	return(result);
}
```