VPS 5	Parallel and Distributed Software Systems	WT 21/22, Exercise 3

Deadline: 11.01.2022 16:20

Name Maximilian Mitter

Points _____ **Effort in hours** 8h

1. Dish of the Day: "Almondbreads" (4 + 8 + 8 + 4 Points)

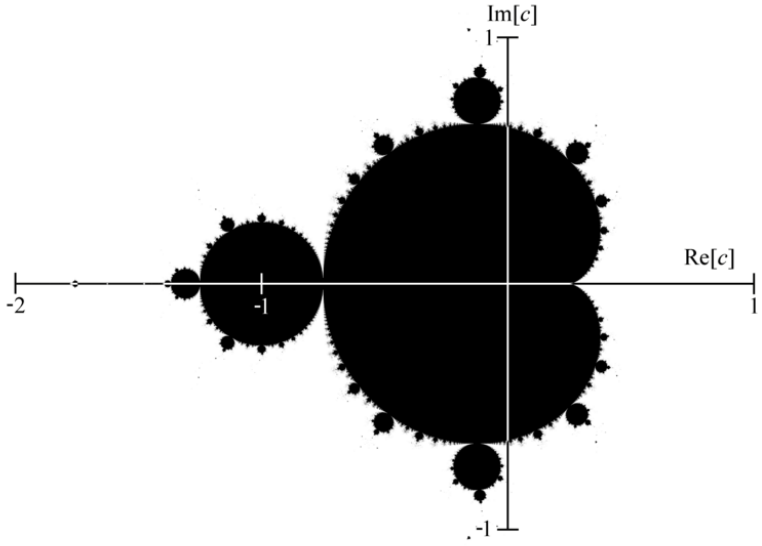
In this exercise, we would like to take a look at a very special form of bread: the "Almondbread" or in other words the *Mandelbrot*. However, the Mandelbrot is not a common form of bread. It is very special (and delicious) and as a consequence, to bake a Mandelbrot we cannot just use normal grains. Instead we need special or complex grains. The recipe is the following:

The Mandelbrot set is the set of complex numbers c, for which the following (recursive) sequence of complex numbers z_n

$$z_0 = 0$$
$$z_{n+1} = z_n^{\,2} + c$$

doesn't diverge towards infinity. If you are not so familiar with complex numbers (anymore), a short introduction can be found at the end of this exercise sheet.

If you mark these points of the Mandelbrot set in the complex plane, you get the very characteristic picture of the set (also called "Apfelmännchen" in German). The set occupies approximately the area from -2-i to 1+i:



To get even more interesting and artistic pictures the points outside the Mandelbrot set can be colored differently depending on how fast the sequence diverges towards infinity. Therefore, just define an upper limit for the absolute value of z_n (usually 4). If z_n grows larger than this upper limit, it can be assumed that z_n will keep growing and will finally diverge. On the other hand if this upper limit is not exceeded in a predefined number of iterations (usually 10.000), it can be assumed that the sequence will not diverge and that the starting point c consequently is element of the Mandelbrot set. Depending on how fast z_n goes beyond the limit (number of iterations) the starting point c can be colored.

a) Write a simple generator in C# using the .NET Windows Forms framework that calculates and displays the Mandelbrot set. Additionally, the generator should have the feature to zoom into the set. Therefore, the user should be able to draw a selection rectangle into the current picture of the set which marks the new section that should be displayed. By clicking on the right mouse button, the original picture (-2-*i* to 1+*i*) should be generated again.

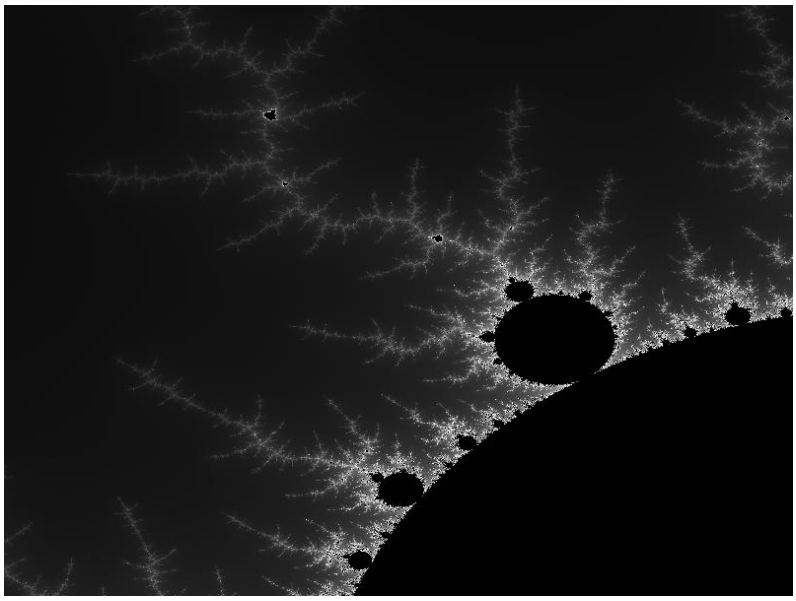
b) Take care that the calculation of the points is computationally expensive. Consequently, it is reasonable to use a separate (worker) thread, so that the user interface stays reactive during the generation of a new picture. However, it can be the case that the user selects a new section before the generation of a previous selection is finished. Furthermore, the time needed for the generation of a picture is variable, depending on how many points of the Mandelbrot set are included in the current selection. It can also happen that the calculation of a latter selected part is finished before an earlier selected one. Therefore, synchronization is necessary to coordinate the different worker threads.

   Implement at least two different ways to create and manage your worker threads (for example you can use BackgroundWorker, threads from the thread pool, plain old thread objects, asynchronous delegates, etc.). Explain how synchronization and management of the worker threads is done in each case.

c) Think about what's the best way to partition the work and to spread it among the workers. Based on these considerations implement a parallel version of the Mandelbrot generator in C# without using the Task Parallel Library (or Parallel.For).

d) Measure the runtime of the sequential and parallel version needed to display the section -1,4-0,1*i* to -1,32-0,02*i* with a resolution of 800 times 600 pixels. Execute 10 independent runs and document also the mean runtimes and the standard deviations.

   For self-control, the generated picture could look like this:



Appendix: Calculations with Complex Numbers

As you all know, it is quite difficult to calculate the square root of negative numbers. However, many applications (electrical engineering, e.g.) require roots of negative numbers leading to the extension of real to complex numbers. So it is necessary to introduce a new number, the imaginary number *i*, which is defined as the square root of -1. A complex number *c* is of the form

$$c = a + b \cdot i$$

where *a* is called the real and *b* the imaginary part. *a* and *b* themselves are normal real numbers.

As a consequence of this special form calculations with complex numbers are a little bit more tricky than in the case of real numbers. The basic arithmetical operations are defined as follows:

$$\begin{aligned}
(a + b \cdot i) + (c + d \cdot i) &= (a + c) + (b + d) \cdot i \\
(a + b \cdot i) - (c + d \cdot i) &= (a - c) + (b - d) \cdot i \\
(a + b \cdot i) \cdot (c + d \cdot i) &= (ac - bd) + (bc + ad) \cdot i \\
\frac{a + b \cdot i}{c + d \cdot i} &= \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i
\end{aligned}$$

Furthermore, we also need the absolute value (distance to 0+0i) of complex numbers which can be calculated easily using the theorem of Pythagoras:

$$\mathrm{Abs}(a + b \cdot i) = \sqrt{a^2 + b^2}$$

Exercise 3

Simple Mandelbrot Generator

The actual generation of the Mandelbrot set was already implemented during class and can be found in the file SyncImageGenerator.

Mandelbrot Generator with Worker Threads

Using Threads

This solution outsources the image generation to a separate thread. The generation can be cancelled via a CancellationToken. After a successful generation, the image is sent to the UI via an Event, similar to the SyncImageGenerator. This was developed during class as well.

Using a BackgroundWorker

A BackgroundWorker is used to wrap the generator thread. It uses the two functions `Run()` and `OnImageGenerated()` to describe, what should happen when the worker runs and what happens after it finished. If a worker is already running and is called again, the running worker will be cancelled and a new worker is initialized. It uses the static function implemented in the SyncImageGenerator to generate the set.

```
public class BackgroundWorkerImageGenerator : IImageGenerator {
    public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>>
ImageGenerated;
    private BackgroundWorker worker;

    public BackgroundWorkerImageGenerator() {
        InitBackgroundWorker();
    }

    private void InitBackgroundWorker() {
        worker = new BackgroundWorker();
        worker.DoWork += Run;
        worker.WorkerSupportsCancellation = true;
        worker.RunWorkerCompleted += OnImageGenerated;
    }

    private void Run(object sender, DoWorkEventArgs e) {
        Area area = (Area)e.Argument;
        Stopwatch sw = new Stopwatch();
        var worker = (BackgroundWorker)sender;
        var token = new CancellationToken(worker.CancellationPending);

        sw.Start();
        var bitmap = SyncImageGenerator.GenerateMandelbrotSet(area, token);
        sw.Stop();

        e.Result = new Tuple<Area, Bitmap, TimeSpan>(area, bitmap, sw.Elapsed);
```

```
    }

    private void OnImageGenerated(object sender, RunWorkerCompletedEventArgs e) {
        if (e.Cancelled) {
            return;
        }

        var handler = ImageGenerated;
        if (handler != null)
            handler(this, new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(
                (Tuple<Area, Bitmap, TimeSpan>)e.Result
                ));
    }

    public void GenerateImage(Area area) {
        if (worker.IsBusy) {
            worker.CancelAsync();
            InitBackgroundWorker();
        }

        worker.RunWorkerAsync(area);
    }
}
```

Partitioning the worker

The Image can be partitioned into columns. The amount of columns depends on the number of workers defined in the settings. Each column is generated by a thread that uses a CancellationToken from the same CancellationTokenSource. Calling the CancellationTokenSource.Cancel function will notify all connected CancellationTokens to stop working. After a column is done rendering it saves the image part in a Bitmap array and checks if the other parts are done already. If all parts are done, they are merged together and sent to the ImageGenerated event.

```
public class ParallelImageGenerator : IImageGenerator {
    public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>>
ImageGenerated;

    private Bitmap[] imageParts;
    private CancellationTokenSource cancellationSource;

    public void GenerateImage(Area area) {
        cancellationSource?.Cancel(false);
        cancellationSource = new CancellationTokenSource();

        var cols = Settings.defaultSettings.Workers;
        var partWidth = area.Width / cols;

        imageParts = new Bitmap[cols];
        int colStart = 0;
        for (int i = 0; i < cols; i++) {
```

```
            int colEnd = colStart + partWidth;

            var t = new Thread(GenerateImagePart);
            t.Start(new Tuple<Area, int, int, int, CancellationToken>(area,
colStart, colEnd, i, cancellationSource.Token));

            colStart += partWidth;
        }
    }

    private void GenerateImagePart(object obj) {
        var tuple = (Tuple<Area, int, int, int, CancellationToken>)obj;
        var sw = new Stopwatch();
        sw.Start();
        var image = GenerateImagePart(tuple.Item1, tuple.Item2, tuple.Item3,
tuple.Item5);
        sw.Stop();

        OnImageGenerated(tuple.Item1, image, sw.Elapsed, tuple.Item4);
    }

    private static Bitmap GenerateImagePart(Area area, int colStart, int colEnd,
CancellationToken token) {
        if (token.IsCancellationRequested) return null;

        var bitmap = new Bitmap(colEnd - colStart, area.Height);
        int maxIterations;
        double zBorder;
        double cReal, cImg, zReal, zImg, zNewReal, zNewImg;

        maxIterations = Settings.DefaultSettings.MaxIterations;
        zBorder = Settings.DefaultSettings.ZBorder *
Settings.DefaultSettings.ZBorder;

        //insert code

        for (int i = colStart; i < colEnd; i++) {
            for (int j = 0; j < area.Height; j++) {
                // extract starting points based on the grid position
                cReal = area.MinReal + i * area.PixelWidth;
                cImg = area.MinImg + j * area.PixelWidth;
                zReal = 0; // sequence variable = current value
                zImg = 0;

                int k = 0;
                while ((zReal * zReal + zImg * zImg < zBorder) && (k <
maxIterations)) {
                    zNewReal = zReal * zReal - zImg * zImg + cReal;
                    zNewImg = 2 * zReal * zImg + cImg;
                    zReal = zNewReal;
                    zImg = zNewImg;
                    k++;
                }
                bitmap.SetPixel(i - colStart, j, ColorSchema.GetColor(k));
```

```
                if (token.IsCancellationRequested) return null;
            }
        }

        //end insert

        return bitmap;
    }

    private void OnImageGenerated(Area area, Bitmap image, TimeSpan elapsed, int
col) {
        imageParts[col] = image;

        if (imageParts.Any(imagepart => imagepart == null)) return;

        var result = MergeBitmaps(area);

        var handler = ImageGenerated;
        if (handler != null)
            handler(this, new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(
                new Tuple<Area, Bitmap, TimeSpan>(area, result, elapsed)
                ));
    }

    private Bitmap MergeBitmaps(Area area) {
        var result = new Bitmap(area.Width, area.Height);

        using (Graphics g = Graphics.FromImage(result)) {
            var colStart = 0;
            foreach (var part in imageParts) {
                g.DrawImage(part, colStart, 0);
                colStart += part.Width;
            }
        }

        return result;
    }
}
```

Measuring performance

Run	SyncImageGenerator	AsyncThreadImageGenerator	BackgroundWorkerImageGenerator	ParallelImageGenerator (4 workers)
1	3.359	3.293	3.276	1.978
2	3.287	3.241	3.273	1.919
3	3.298	3.254	3.278	1.911
4	3.292	3.244	3.273	1.916
5	3.329	3.251	3.280	1.938
6	3.331	3.247	3.292	1.896
7	3.246	3.251	3.277	1.937
8	3.263	3.247	3.284	1.973
9	3.308	3.242	3.271	1.919
10	3.244	3.265	3.286	1.914
Mean	3.296	3.254	3.279	1.930
STDDV	0.03774196	0.015522385	0.006616478	0.026826397

As expected, the first three generators offer very similar performance, as the actual image generation does not really differ much. The asynchronous implementations seem to be more consistent and have a lower standard deviation.

Using 4 workers to generate the image helps it run almost 3x as fast. While one might expect a 4x increase in performance, that would not be realistic, as the parallel generator requires some overhead to create the threads and merge the image in the end.