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

Deadline: 11.01.2022 16:20

Name Andreas Wenzelhuemer

Points _____ **Effort in hours** 5

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

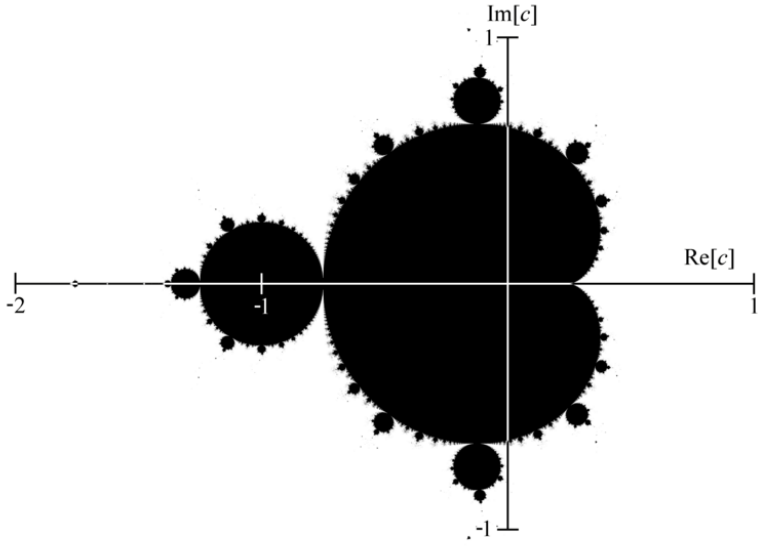
In this exercise, we would like to take a look at a very special form of bread: the "Almondbread" or in other words the *Mandelbrot*. However, the Mandelbrot is not a common form of bread. It is very special (and delicious) and as a consequence, to bake a Mandelbrot we cannot just use normal grains.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

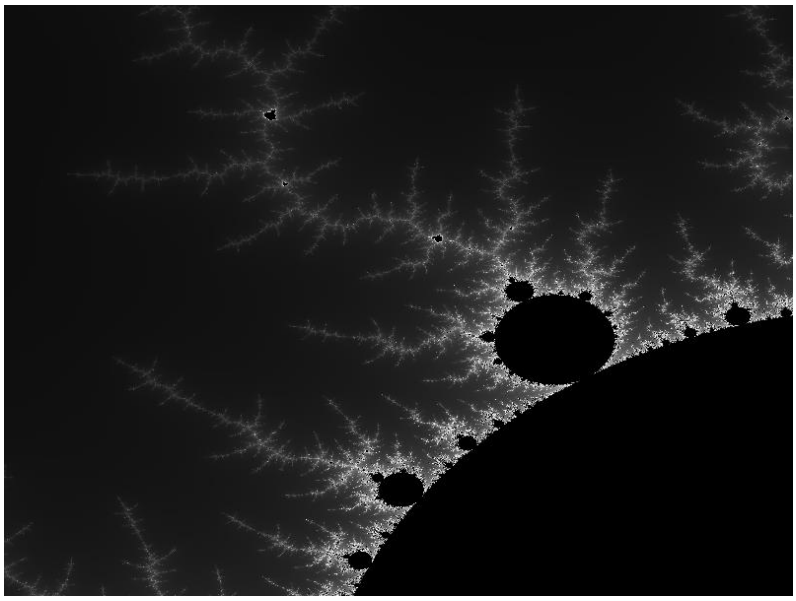
b) Take care that the calculation of the points is computationally expensive. Consequently, it is reasonable to use a separate (worker) thread, so that the user interface stays reactive during the generation of a new picture. However, it can be the case that the user selects a new section before the 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:

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

Übung 3

Table of Contents

1. Simple Mandelbrot generator 2
2. Worker thread. 4
    2.1. Solution with thread 4
    2.2. Solution with background worker. 5
3. Parallelize work 9
4. Runtime measurement 13

1. Simple Mandelbrot generator

Mandelbrot generation was already finished in the lesson. This is the code for the synchronous image generation.

Listing 1. Image generation

```
using System;
using System.Diagnostics;
using System.Drawing;

namespace MandelbrotGenerator.Generators
{
    public class SyncImageGenerator : IImageGenerator
    {
        #region Public Events

        public event EventHandler<ImageGeneratedEventArgs> ImageGenerated;

        #endregion

        #region Public Methods

        public void GenerateImage(Area area)
        {
            Stopwatch stopWatch = Stopwatch.StartNew();

            int maxIterations;
            double zBorder;
            double cReal, cImg, zReal, zImg, zNewReal, zNewImg;

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

            Bitmap bitmap = new Bitmap(area.Width, area.Height);

            for (int x = 0; x < area.Width; x++)
            {
                cReal = area.MinReal + x * area.PixelWidth;
                for (int y = 0; y < area.Height; y++)
                {
                    cImg = area.MinImg + y * area.PixelHeight;
                    zReal = 0.0;
                    zImg = 0.0;
                    int iteration = 0;
                    while (iteration < maxIterations // Check if smaller max iterations
                        && zReal * zReal + zImg * zImg < zBorder) // Check if in border
                    {
                        zNewReal = zReal * zReal - zImg * zImg + cReal;
                        zNewImg = zReal * zImg * 2 + cImg;
                        zReal = zNewReal;
                        zImg = zNewImg;
                        iteration += 1;
                    }
                    bitmap.SetPixel(x, y, ColorSchema.GetColor(iteration));
                }
```

```
            }

            stopWatch.Stop();
            ImageGenerated?.Invoke(this, new ImageGeneratedEventArgs(bitmap, area, stopWatch
.Elapsed));

        }

        #endregion
    }
}
```

2. Worker thread

2.1. Solution with thread

One possible solution would be to create each time a new thread which starts the image execution. If the method is currently executing an image generation, the previous generation gets cancelled via `CancellationTokenSource`. Additionally if the image was generated successfully, an event gets fired where the new image, area and time gets passed with `EventArgs`.

Listing 2. Image generator with thread

```
using System;
using System.Diagnostics;
using System.Drawing;
using System.Threading;

namespace MandelbrotGenerator.Generators
{
    public class AsyncThreadImageGenerator : IImageGenerator
    {
        #region Private Fields

        private CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
        private Thread thread;

        #endregion

        #region Public Events

        public event EventHandler<ImageGeneratedEventArgs> ImageGenerated;

        #endregion

        #region Public Methods

        public static Bitmap GenerateImage(Area area, CancellationToken cancellationToken)
        {
            int maxIterations;
            double zBorder;
            double cReal, cImg, zReal, zImg, zNewReal, zNewImg;

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

            Bitmap bitmap = new Bitmap(area.Width, area.Height);

            for (int x = 0; x < area.Width; x++)
            {
                if (cancellationToken.IsCancellationRequested)
                {
                    return null;
                }
```

```
                cReal = area.MinReal + x * area.PixelWidth;
                for (int y = 0; y < area.Height; y++)
                {
                    cImg = area.MinImg + y * area.PixelHeight;
                    zReal = 0.0;
                    zImg = 0.0;
                    int iteration = 0;
                    while (iteration < maxIterations // Check if smaller max iterations
                        && zReal * zReal + zImg * zImg < zBorder) // Check if in border
                    {
                        zNewReal = zReal * zReal - zImg * zImg + cReal;
                        zNewImg = zReal * zImg * 2 + cImg;
                        zReal = zNewReal;
                        zImg = zNewImg;
                        iteration += 1;
                    }
                    bitmap.SetPixel(x, y, ColorSchema.GetColor(iteration));
                }
            }

            return bitmap;
        }

        public void GenerateImage(Area area)
        {
            cancellationTokenSource.Cancel(); // Cancel previous calculation
            cancellationTokenSource = new CancellationTokenSource(); // Create new cancellation
source
            var token = cancellationTokenSource.Token;
            thread = new Thread(() =>
            {
                var watch = Stopwatch.StartNew();
                var bitmap = GenerateImage(area, token);
                var args = new ImageGeneratedEventArgs(bitmap, area, watch.Elapsed);
                if (!token.IsCancellationRequested)
                {
                    ImageGenerated?.Invoke(this, args);
                }
            });
            thread.Start();
        }

        #endregion
    }
}
```

2.2. Solution with background worker

BackgroundWorker is used for image generation. Previous worker gets cancelled if new worker gets created and started. Two callback methods are used: `DoWork` and `RunWorkerCompleted`. Additionally the flag `WorkerSupportsCancellation` has to be set.

Listing 3. Image generator with background worker

```
using MandelbrotGenerator.Generators;
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Drawing;

namespace MandelbrotGenerator
{
    public class BackgroundWorkerImageGenerator : IImageGenerator
    {
        #region Private Fields

        private BackgroundWorker backgroundWorker;

        #endregion

        #region Public Constructors

        public BackgroundWorkerImageGenerator()
        {
            InitializeBackgroundWorker();
        }

        #endregion

        #region Public Events

        public event EventHandler<ImageGeneratedEventArgs> ImageGenerated;

        #endregion

        #region Private Methods

        private static Bitmap GenerateImage(Area area, BackgroundWorker worker, DoWorkEventArgs
e)
        {
            if (worker.CancellationPending)
            {
                e.Cancel = true;
                return null;
            }

            int maxIterations;
            double zBorder;
            double cReal, cImg, zReal, zImg, zNewReal, zNewImg;

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

            Bitmap bitmap = new Bitmap(area.Width, area.Height);

            for (int x = 0; x < area.Width; x++)
            {
                if (worker.CancellationPending)
                {
                    e.Cancel = true;
                    return null;
                }
                cReal = area.MinReal + x * area.PixelWidth;
                for (int y = 0; y < area.Height; y++)
```

```
            {
                cImg = area.MinImg + y * area.PixelHeight;
                zReal = 0.0;
                zImg = 0.0;
                int iteration = 0;
                while (iteration < maxIterations // Check if smaller max iterations
                    && zReal * zReal + zImg * zImg < zBorder) // Check if in border
                {
                    zNewReal = zReal * zReal - zImg * zImg + cReal;
                    zNewImg = zReal * zImg * 2 + cImg;
                    zReal = zNewReal;
                    zImg = zNewImg;
                    iteration += 1;
                }
                bitmap.SetPixel(x, y, ColorSchema.GetColor(iteration));
            }
        }

        return bitmap;
    }

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

        ImageGenerated?.Invoke(this, (ImageGeneratedEventArgs)e.Result);
    }

    private void InitializeBackgroundWorker()
    {
        backgroundWorker = new BackgroundWorker();

        backgroundWorker.DoWork += Run;
        backgroundWorker.WorkerSupportsCancellation = true;
        backgroundWorker.RunWorkerCompleted += Completed;
    }

    private void Run(object sender, DoWorkEventArgs e)
    {
        var area = (Area)e.Argument;

        var watch = Stopwatch.StartNew();
        var bitmap = GenerateImage(area, sender as BackgroundWorker, e);
        var args = new ImageGeneratedEventArgs(bitmap, area, watch.Elapsed);

        e.Result = args;
    }

    #endregion

    #region Public Methods

    public void GenerateImage(Area area)
    {
        if (backgroundWorker.IsBusy)
        {
            backgroundWorker.CancelAsync();
```

```
                InitializeBackgroundWorker();
            }

            backgroundWorker.RunWorkerAsync(area);
        }

        #endregion
    }
}
```

3. Parallelize work

The whole areal gets splitted into multiple rows and columns. Each part gets calculated separately. That means that for each each part an separate thread gets created. The number of rows and columns can be configured over the setting for the worker count. After the generation of each part they get merged into one bitmap.

Listing 4. Image generator with parallel generator

```
using System;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Threading;

namespace MandelbrotGenerator.Generators
{
    public class MultiAsyncThreadImageGenerator : IImageGenerator
    {
        #region Private Fields

        private Bitmap[] bitmaps;

        private CancellationTokenSource cancellationToken;

        #endregion

        #region Public Events

        public event EventHandler<ImageGeneratedEventArgs> ImageGenerated;

        #endregion

        #region Private Methods

        private static Bitmap GenerateImagePart(Area area, int startWidth, int endWidth, int
startHeight,
            int endHeight, CancellationToken token)
        {
            if (token.IsCancellationRequested)
            {
                return null;
            }

            Bitmap bitmap = new Bitmap(endWidth - startWidth, endHeight - startHeight);
            int maxIterations = Settings.DefaultSettings.MaxIterations;
            double zBorder = Settings.DefaultSettings.ZBorder * Settings.DefaultSettings.
ZBorder;
            double cReal, cImg, zReal, zImg, zNewReal, zNewImg;

            for (int x = startWidth; x < endWidth; x++)
            {
                if (token.IsCancellationRequested)
                {
                    return null;
```

```
                }

                cReal = area.MinReal + x * area.PixelWidth;
                for (int y = startHeight; y < endHeight; y++)
                {
                    cImg = area.MinImg + y * area.PixelHeight;
                    zReal = 0.0;
                    zImg = 0.0;
                    int iteration = 0;
                    while (iteration < maxIterations // Check if smaller max iterations
                           && zReal * zReal + zImg * zImg < zBorder) // Check if in border
                    {
                        zNewReal = zReal * zReal - zImg * zImg + cReal;
                        zNewImg = zReal * zImg * 2 + cImg;
                        zReal = zNewReal;
                        zImg = zNewImg;
                        iteration += 1;
                    }

                    bitmap.SetPixel(x - startWidth, y - startHeight, ColorSchema.GetColor
(iteration));
                }
            }

            return bitmap;
        }

        private void GenerateImagePart(object obj)
        {
            var tuple = (Tuple<Area, int, int, int, int, int, CancellationToken>)obj;
            var area = tuple.Item1;
            var index = tuple.Item6;
            var cancellationToken = tuple.Item7;

            var sw = Stopwatch.StartNew();
            var bitmap = GenerateImagePart(area, tuple.Item2, tuple.Item3, tuple.Item4, tuple
.Item5, cancellationToken);
            sw.Stop();

            OnImageGenerated(area, bitmap, sw.Elapsed, index);
        }

        private Bitmap MergeBitmaps(Area area)
        {
            var result = new Bitmap(area.Width, area.Height);
            using (Graphics graphics = Graphics.FromImage(result))
            {
                var startWidth = 0;
                var startHeight = 0;
                for (var x = 0; x < bitmaps.Length; x++)
                {
                    graphics.DrawImage(bitmaps[x], startWidth, startHeight);
                    startHeight += bitmaps[x].Height;
                    if (startHeight >= area.Height)
                    {
                        startHeight = 0;
                        startWidth += bitmaps[x].Width;
                    }
                }
            }
```

```
            return result;
        }

        private void OnImageGenerated(Area area, Bitmap bitmap, TimeSpan elapsed, int index)
        {
            bitmaps[index] = bitmap;
            if (bitmaps.Any(map => map == null))
            {
                return;
            }

            var resultingBitmap = MergeBitmaps(area);
            var handler = ImageGenerated;
            handler?.Invoke(this, new ImageGeneratedEventArgs(resultingBitmap, area, elapsed));
        }

        #endregion

        #region Public Methods

        public void GenerateImage(Area area)
        {
            // Cancel previous calculations
            cancellationToken?.Cancel(false);
            cancellationToken = new CancellationTokenSource();

            int rows;
            int cols;

            // Calculate how many image parts are needed
            if (Settings.DefaultSettings.Workers > 1)
            {
                rows = 2;
                cols = Settings.DefaultSettings.Workers / 2;
            }
            else
            {
                rows = 1;
                cols = 1;
            }

            var fractionWidth = (int)Math.Floor((double)area.Width / cols);
            var fractionHeight = (int)Math.Floor((double)area.Height / rows);

            bitmaps = new Bitmap[rows * cols];

            int startWidth = 0;
            int startHeight = 0;
            for (int i = 0; i < rows * cols; i++)
            {
                int endWidth = startWidth + fractionWidth >= area.Width - 1 ? area.Width :
startWidth + fractionWidth;
                int endHeight = startHeight + fractionHeight >= area.Height - 1 ? area.Height :
startHeight + fractionHeight;
                var thread = new Thread(GenerateImagePart);
                thread.Start(new Tuple<Area, int, int, int, int, int, CancellationToken>(area,
startWidth, endWidth, startHeight, endHeight, i, cancellationToken.Token));
                startHeight += fractionHeight;
                if (endHeight == area.Height)
```

```
                {
                    startHeight = 0;
                    startWidth += fractionWidth;
                }
            }
        }

        #endregion
    }
}
```

4. Runtime measurement

The performance of the parallel generator is similar to the synchronized one. That's because there is much overhead, for example the merging of the bitmap parts and the calculation for each part. When the number of workers is changed to 8, the generation is twice as fast as the synchronized one.

Run	Syncronized execution	Parallel execution (4 Workers)	Parallel execution (8 workers)
1	643.75680	663.77710	330.16360
2	601.94520	552.82940	328.86840
3	582.95330	564.10520	337.98230
4	583.43150	547.40450	327.58120
5	589.63790	548.44320	337.47930
6	585.59210	580.01350	318.04550
7	595.38610	555.63640	329.68980
8	597.59320	555.60260	328.24720
9	593.22640	556.95190	358.50740
10	588.11790	564.56930	325.68530
STDDEV	17.84348013	34.64295721	10.82525731
AVG	596.16404	568.93331	332.22500