VPS 5	Parallel and Distributed Software Systems	ST 16, Exercise 3

Deadline: 30th of April, 2016 08:00

Name Stefan Kert

Points _____ **Effort in hours** 6

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

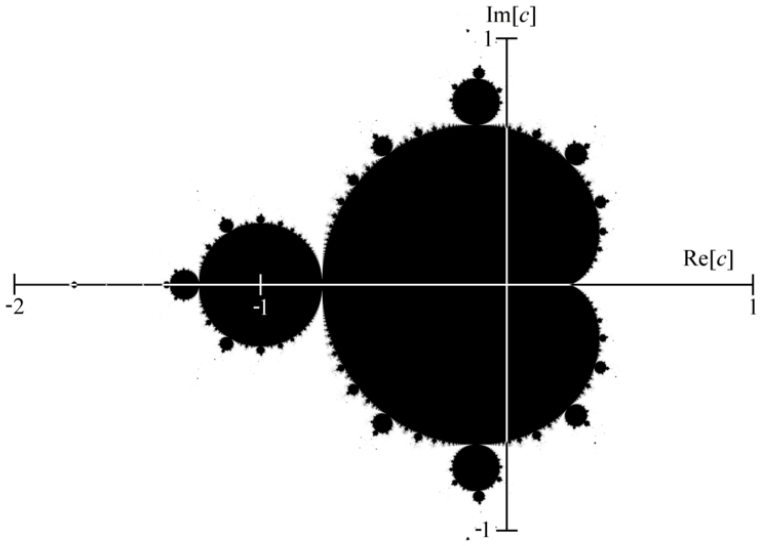
In this exercise we would like to take a look at a very special form of bread: the "Almondbread" or in other words the *Mandelbrot*. However, the Mandelbrot is not a common form of bread. It is very special (and delicious) and as a consequence, to bake a Mandelbrot we cannot just use normal grains. Instead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

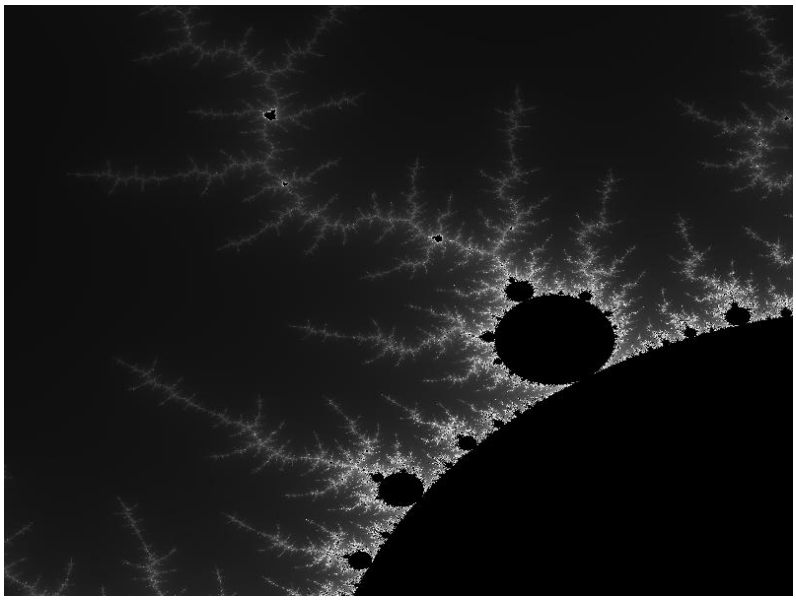
b) Take care that the calculation of the points is computationally expensive. Consequently, it is reasonable to use a separate (worker) thread, so that the user interface stays reactive during the generation of a new picture. However, it can be the case that the user selects a new section before the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

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

VPS5 - UE3

Stefan Kert

28. April 2016

1 Dish of the Day: Almondbreads

1.1 a. Sync - Generator

Der Synchrone Generator wurde bereits in der Übungsstunde implementiert und nur leicht abgeändert, sodass die weiteren Komponenten die für diese Übung entwickelt wurden den Code dieses Generators verwenden können. Dabei enthält der SyncImageGenerator relativ wenig Code, da dort nur die Logik für die Stopwatch bzw. das Aufrufen des Events und der *GenerateBitmap* Methode getätigt wird.

```
public class SyncImageGenerator : AbstractImageGenerator
{
    public override void GenerateImage(Area area) {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        var bm = GenerateBitmap(area, Source.Token);
        stopwatch.Stop();
        OnImageGenerated(area, bm, stopwatch.Elapsed);
    }
}
```

Im *AbstractImageGenerator*, der als Basisklasse für sämtliche Generatoren dient, ist die Logik für das generieren der einzelnen Zeilen und Spalten des Bitmaps, sowie etwaige Events und Hilfsmethoden.

```
 public abstract class AbstractImageGenerator
{
    protected CancellationTokenSource Source;

    protected AbstractImageGenerator(): this(new
        CancellationTokenSource()) {}
    private AbstractImageGenerator(CancellationTokenSource source) {
        Source = source;
    }

    protected virtual Bitmap GenerateBitmap(Area area,
        CancellationToken cancelToken)
    {
        Bitmap bitmap = new Bitmap(area.Width, area.Height);
```

```
            for (int column = 0; column < area.Width; column++) {
                for (int row = 0; row < area.Height; row++) {
                    if (cancelToken.IsCancellationRequested)
                        return null;
                    bitmap.SetPixel(column, row,
                        ImageGeneratorLogic.GetColorForData(area, column,
                        row, cancelToken));
                }
            }
            return bitmap;
        }



        public abstract void GenerateImage(Area area);

        public event EventHandler<EventArgs<Tuple<Area, Bitmap,
            TimeSpan>>> ImageGenerated;

        protected void OnImageGenerated(Area area, Bitmap bitmap,
            TimeSpan timeSpan) {
            if (area != null && bitmap != null)
                ImageGenerated?.Invoke(this, new EventArgs<Tuple<Area,
                    Bitmap, TimeSpan>>(new Tuple<Area, Bitmap,
                    TimeSpan>(area, bitmap, timeSpan)));
        }
    }
```

Um eine bessere Auftrennung der einzelnen Aspekte zu gewährleisten, wurde die Methode für die Generierung der Farbe der einzelnen Felder in eine eigene Klasse *ImageGeneratorLogic* ausgelagert. Dies dient in erster Linie dem Code Reuse, sowie der Testbarkeit, da durch ein herauslösen dieser Methode eine leichtere Testbarkeit gewährleistet ist.

```
    public static class ImageGeneratorLogic
    {
        private static int MaxIterations =>
            Settings.DefaultSettings.MaxIterations;
        private static double Border => Settings.DefaultSettings.ZBorder
            * Settings.DefaultSettings.ZBorder;

        public static Color GetColorForData(Area area, int column, int
            row, CancellationToken cancelToken)
        {
            var cReal = area.MinReal + column * area.PixelWidth;
            var cImg = area.MinImg + row * area.PixelHeight;
            var zReal = 0.0;
            var zImg = 0.0;

            var k = 0;
            while ((zReal * zReal + zImg * zImg < Border) && (k <
                MaxIterations))
            {
                if (cancelToken.IsCancellationRequested)
```

```
                return Color.Empty;
            var zNewReal = zReal * zReal - zImg * zImg + cReal;
            var zNewImg = 2 * zReal * zImg + cImg;

            zReal = zNewReal;
            zImg = zNewImg;

            k++;
        }
        return ColorSchema.GetColor(k);
    }
}
```

1.2 b. Async- Generator

Um Userinteraktionen auch während einer Berechnung zu ermöglichen ist es nötig, die Berechnung des Bildes asynchron auszuführen. Dazu wurden bereits für den Synchronen Generator Vorkehrungen getroffen. Diese Vorkehrungen waren in erster Linie das Hinzufügen eines Events, welches signalisieren sollte, dass die Berechnung abgeschlossen wurde. Diese Event wird über die Methode *OnImageGenerated* aufgerufen. Das Auslösen dieses Events ruft eine Methode im der *MainForm* auf, welche schließlich das Bild lädt und die Informationen über die benötigte Zeit setzt. Hier ist nur wichtig, dass darauf geachtet wird, dass die Aktionen in der GUI im GUI Thread getätigt werden. Hierzu ist eine Synchronisierung notwendig.

In den beiden Implementierungen sollte darauf geachtet werden, dass falls eine erneute Berechnung gestartet wird, eine möglicherweise laufende Berechnung beendet wird. Dies erfolgt jeweils mittels Speichern der jeweiligen Thread, bzw. Backgroundworker Instanz in einer Membervariable und durch die Überprüfung dieser. Nähere Informationen hierzu in den einzelnen Unterpunkten dieser Implementierungen. Da die Asynchrone Variante mit *CancellationTokens* arbeitet, muss hier darauf geachtet werden, dass dieser bei einem Abbrechen gecanceld wird und gewartet wird bis dieses Canceln abgeschlossen wurde. Danach kann die jeweilige Instanz neu erstellt werden und eine neue Berechnung gestartet werden.

1.2.1 AsyncGenerator mit Threads

Die erste Implementierung wurde mit einfachen Threads realisiert. Dazu wird ein Thread mit der Methode gestartet, welche die Berechnung vornimmt. Heir wird überprüft ob die Membervariable ungleich null ist und ob das Flag *IsAlive* true ist. Wenn dies der Fall ist wird über den *Cancellationtoken* die Operation abgebrochen und mit *WaitOne()* gewartet bis dieses Abbrechen abgeschlossen ist. Danach wird eine neue Instanz des *CancellationTokenSource* erstellt und die neue Berechnung gestartet. Wenn die Berechnung abgeschlossen wird, wird die Methode *OnImageGenerated* aufgerufen, welche wie oben beschrieben signalisiert, dass die Operation beendet worden ist.

```
public class AsyncImageGenerator : AbstractImageGenerator
{
```

```
    private Thread _calculationThread;

    public override void GenerateImage(Area area) {
        if (_calculationThread != null &&
            _calculationThread.IsAlive) {
            Source.Cancel();
            Source.Token.WaitHandle.WaitOne();
            Source = new CancellationTokenSource();
        }
        _calculationThread = new Thread(() => BuildBitmap(area));
        _calculationThread.Start();
    }

    protected virtual void BuildBitmap(Area area) {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        var bm = GenerateBitmap(area, Source.Token);
        stopwatch.Stop();
        OnImageGenerated(area, bm, stopwatch.Elapsed);
    }
}
```

1.2.2 AsyncGenerator mit Backgroundworker

Die Implementierung mittels *Backgroundworker* funktioniert sehr ähnlich. Auch hier wird mittels einer Membervariable auf null überprüft, und ob der *Backgroundworker* gerade läuft. Wenn ja wird wiederum über den *CancellationToken* ein *Cancel()* aufgerufen und die Operation somit beendet. Auch hier wird gewartet und nach Abschluss wird der *BackgroundWorker* gecancelt. Es wäre hier auch möglich gewesen, die CancelationEvents über den *BackgroundWorker* zu implementieren, jedoch würde dies zu einer Codeduplizierung führen, da die Logik für die Cancellation erneut implementiert werden müsste. Beim *Backgroundworker* werden die Events *DoWork* und *RunWorkerCompleted* gebunden. Über *DoWork* wird die Methode zum Generieren des Bitmaps gestartet. Nach Abschluss dieser wird der Result Member der *DoWorkEventArgs* gesetzt welcher schließlich in der *RunWorkerCompleted* Methode verwendet wird. In dieser Methode wird die *OnImageGenerated* Methode aufgerufen womit die Berechnung beendet ist.

```
public class AsyncImageGeneratorWithBackgroundWorker :
    AbstractImageGenerator
{
    private BackgroundWorker _bw;

    public override void GenerateImage(Area area) {
        if (_bw != null && _bw.IsBusy) {
            Source.Cancel();
            Source.Token.WaitHandle.WaitOne();
            Source = new CancellationTokenSource();
            _bw.CancelAsync();
        }
```

```
            _bw = new BackgroundWorker {WorkerSupportsCancellation =
                true};
            _bw.DoWork += (sender, args) => BuildBitmap(area, args);
            _bw.RunWorkerCompleted += OnBuildBitmapCompleted;
            _bw.RunWorkerAsync();
        }

        private void BuildBitmap(Area area, DoWorkEventArgs args)
        {
            var stopwatch = new Stopwatch();
            stopwatch.Start();
            var bm = GenerateBitmap(area, Source.Token);
            stopwatch.Stop();
            args.Result = new Tuple<Area, Bitmap, TimeSpan>(area, bm,
                stopwatch.Elapsed);
        }

        private void OnBuildBitmapCompleted(object sender,
            RunWorkerCompletedEventArgs e) {
            var bw = sender as BackgroundWorker;
            if (bw != null)
                bw.RunWorkerCompleted -= OnBuildBitmapCompleted;
            var res = e.Result as Tuple<Area, Bitmap, TimeSpan>;
            OnImageGenerated(res?.Item1, res?.Item2, res?.Item3 ?? new
                TimeSpan());
        }
    }
```

1.3 c. Parallel - Generator

Durch ein Aufteilen der Berechnung in mehrere Threads und ein paralleles ausführen dieser Berechnungen kann ein Speed Up erreicht werden. Dazu werden mehrere Threads erzeugt und jedem dieser Threads wird eine Methode zur Berechnung einer Zeile zugeordnet. Die Nummer der Zeile werden über eine Membervariable ausgelesen, welche auch über einen *lock* synchronisiert wird. Diese Zeilennummer gibt aufschluss darüber, welche Reihe gerade berechnet wird. Nach der Berechnung der aktuellen Farbe wird das bitmap gelockt und die Pixel mit der jeweiligen Farbe gesetzt. Nach dem Initialisieren der Threads wird mittels Schleife auf das Enden der einzelnen Threads gewartet und schließlich, insofern die Operation nicht vorher abgebrochen wurde, die *OnImageCreated* Methode aufgerufen.

```
public class ParallelImageGenerator : AsyncImageGenerator
{
    private readonly object _bitmapLock = new object();
    private readonly object _currentRowLock = new object();
    private int _currentRow;
    private Bitmap _bitmap;

    protected  override void BuildBitmap(Area area)
    {
        var stopwatch = new Stopwatch();
```

```
            stopwatch.Start();
            _currentRow = 0;
            _bitmap = new Bitmap(area.Width, area.Height);
            var threads = new Thread[Settings.DefaultSettings.Workers];

            for (int i = 0; i < threads.Length; i++) {
                Thread t = new Thread(() => BuildBitmap(area,
                    Source.Token));
                threads[i] = t;
                t.Start();
            }

            foreach (var thread in threads) {
                thread.Join();
            }
            stopwatch.Stop();
            if (Source.Token.IsCancellationRequested)
                _bitmap = null;
            OnImageGenerated(area, _bitmap, stopwatch.Elapsed);
        }


        private void BuildBitmap(Area area, CancellationToken
            cancellationToken)
        {
            while (!cancellationToken.IsCancellationRequested)
            {
                var row = GetCurrentRow();
                if (row >= area.Height)
                    return;

                for (int column = 0; column < area.Width; column++) {
                    var color =
                        ImageGeneratorLogic.GetColorForData(area, column,
                        row, cancellationToken);
                    lock (_bitmapLock) {
                        _bitmap.SetPixel(column, row, color);
                    }
                }
            }
        }

        private int GetCurrentRow() {
            int row;
            lock (_currentRowLock) {
                row = _currentRow;
                _currentRow++;
            }
            return row;
        }
    }
```

1.4 d. Performancevergleich

Wie in den Anforderungen definiert wurde der Vergleich mit folgenden Parametern durchgeführt:

- MaxIterations: 10000
- ZBorder: 4
- MaxImg: -0,02
- MaxReal: -1,32
- MinImg: -0,1
- MinReal: -1,4
- Workers: 10

Tabelle 1: Performancevergleich: Ergebnisse

Run	Synchron [ms]	Parallel [ms]	
1	4955	1561	
2	4958	1432	
3	5010	1512	
4	5077	1498	
5	4993	1532	
6	4981	1554	
7	4941	1388	
8	4959	1451	
9	4939	1523	
10	5026	1486	
Mittelwert	4983	1493	
Std. Abw.	41,5	52,6	
Speed Up			3,34

Dieser Vergleich zeigt, dass durch ein Parallelisieren der einzelnen Berechnungen ein enormer Speed Up erreicht werden kann.