VPS 5	Parallel and Distributed Software Systems	ST 16, Exercise 4

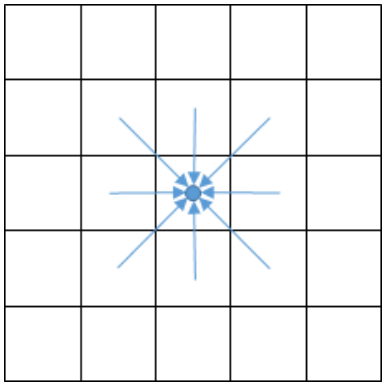
Deadline: 15th of May, 2016 23:55

Name Roman Lumetsberger

Points _____ **Effort in hours** 7

1. Psychedelic Diffusions (4 + 4 + 4 Points)

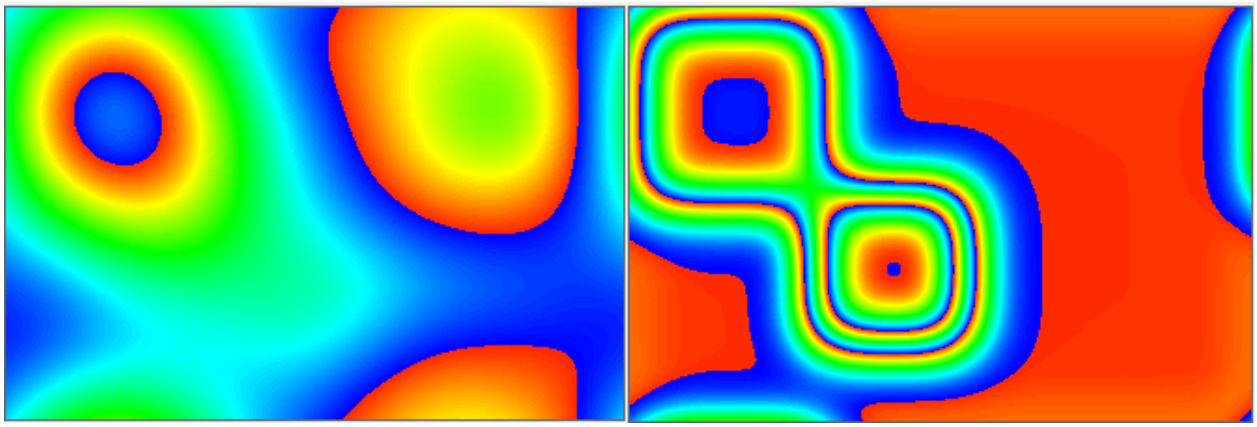
On Moodle you find a template for implementing a diffusion simulation. Simple diffusion simulations work by computing the value of a point by averaging over its neighboring points:



$$f(p) = \frac{1}{8} * \sum neighbor(p)$$

a) Implement a sequential version of the simulation in C# using the provided template. Also implement the mouse event for "reheating" at a point as well as starting and stopping the simulation.
b) Use any of the learned techniques to compute the simulation in the background to provide a responsive UI. Take care of proper cancelation and locking!
c) Implement a parallel version of the simulation with the parallelization technique of your choice. Discuss your design considerations and calculate the speedup.

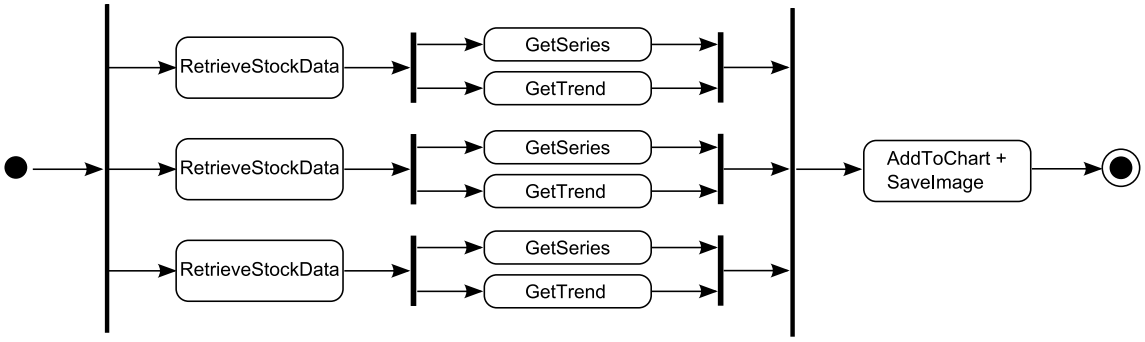
Document each step and also show a screenshot of the application in action.



2. Stock Data Visualization (6 + 6 Points)

Web requests can often be executed asynchronously to improve the responsiveness of an application and to reduce the total response time of multiple requests. On Moodle we provide a sequential implementation of a stock data visualization program that downloads price histories of three stocks from *quandl.com* and shows them in a line chart.

a) Implement an asynchronous version of the stock data visualization program using the .NET Task Parallel Library. Your version should execute tasks as shown in the following activity diagram. Use continuations to create chains of several tasks.



b) Implement a second version of the stock data visualization program which uses the keywords *async* and *await*. Make sure the tasks are again modeled as shown in the activity diagram.

1 Psychedelic Diffusions

1a. Sequenzielle Version

Diese Aufgabe wurde bereits in der Übung programmiert und um das Stoppen und Reheating erweitert. Für die Implementierung selbst wurde eine Basisklasse `ImageGenerator` erstellt, die die Zeitmessung und das Auslösen des Events übernimmt. Weiters wurde das Abbrechen über ein Stop-Flag `StopRequested` implementiert. Das Reheating wird mit Hilfe einer `BlockingQueue` ermöglicht. Diese Queue sammelt alle Reheating-Aufrufe während einer Iteration und wendet sie dann vor Begin der nächsten Iteration auf die aktuelle Matrix an.

Der `SyncImageGenerator` implementiert dann nur noch die Logik zur Erzeugung der nächsten Matrix und Berechnung der Farben.

```
1  using System;
2  using System.Collections.Concurrent;
3  using System.Diagnostics;
4  using System.Drawing;
5  using System.Threading.Tasks;
6
7  namespace Diffusions
8  {
9      public abstract class ImageGenerator : IImageGenerator
10     {
11         protected ConcurrentQueue<ReheatItem> reheatItems;
12
13         public bool StopRequested { get; protected set; }
14
15         public bool Finished { get; protected set; }
16
17         public ImageGenerator()
18         {
19             reheatItems = new ConcurrentQueue<ReheatItem>();
20         }
21
22         public async void GenerateImage(Area area)
23         {
24             Finished = false;
25             StopRequested = false;
26             await Task.Factory.StartNew(() =>
27              {
28                  int maxIt = Settings.DefaultSettings.MaxIterations;
29                  Stopwatch watch = new Stopwatch();
30                  watch.Start();
31                  for (int i = 0; i < maxIt && !StopRequested; i++)
32                  {
33                      //check reheats
34                      while (!reheatItems.IsEmpty)
35                      {
36                          ReheatItem item;
```

```
37                          if (reheatItems.TryDequeue(out item))
38                          {
39                              ReheatMatrix(area.Matrix, item);
40                          }
41                      }
42
43                      Bitmap bitmap = GenerateBitmap(area);
44                      OnImageGenerated(area, bitmap, watch.Elapsed);
45
46                  }
47                  watch.Stop();
48                  Finished = true;
49                  OnImageGenerated(area, null, watch.Elapsed);
50              });
51          }
52
53          protected abstract Bitmap GenerateBitmap(Area are);
54
55          public virtual void ColorBitmap(double[,] array, int width, int height, Bitmap bm)
56          {
57              int maxColorIndex = ColorSchema.Colors.Count - 1;
58
59              for (int i = 0; i < width; i++)
60              {
61                  for (int j = 0; j < height; j++)
62                  {
63                      int colorIndex = (int) array[i, j]%maxColorIndex;
64                      bm.SetPixel(i, j, ColorSchema.Colors[colorIndex]);
65                  }
66              }
67          }
68
69          public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>> ImageGenerated;
70
71          protected void OnImageGenerated(Area area, Bitmap bitmap, TimeSpan timespan)
72          {
73              ImageGenerated?.Invoke(this,
74                  new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(new Tuple<Area, Bitmap,
75                      TimeSpan>(area, bitmap,timespan)));
76          }
77
78          public virtual void Stop()
79          {
80              StopRequested = true;
81          }
82
83          /// <summary>
84          /// Adds a pending reheat to the image generator
85          /// </summary>
```

```
86          /// <param name="x">The x.</param>
87          /// <param name="y">The y.</param>
88          /// <param name="width">The width.</param>
89          /// <param name="height">The height.</param>
90          /// <param name="size">The size.</param>
91          /// <param name="val">The value.</param>
92          public void Reheat(int x, int y, int width, int height, int size, double val)
93          {
94              reheatItems.Enqueue(new ReheatItem
95              {
96                  X = x,
97                  Y = y,
98                  Width = width,
99                  Height = height,
100                 Size = size,
101                 Val = val
102             });
103         }
104
105         /// <summary>
106         /// Reheats the matrix.
107         /// </summary>
108         /// <param name="matrix">The matrix.</param>
109         /// <param name="item">The item.</param>
110         private void ReheatMatrix(double[,] matrix, ReheatItem item)
111         {
112             for (int i = 0; i < item.Size; i++)
113             {
114                 for (int j = 0; j < item.Size; j++)
115                 {
116                     matrix[(item.X + i) % item.Width, (item.Y + j) % item.Height] = item.Val;
117                 }
118             }
119         }
120
121
122
123
124     }
125
126
127 }
```

```
1 using System.Drawing;
2 using System.Resources;
3
4 namespace Diffusions
5 {
```

```
6       class SyncImageGenerator:ImageGenerator
7       {
8           protected override Bitmap GenerateBitmap(Area area)
9           {
10              var matrix = area.Matrix;
11              int height = area.Height;
12              int width = area.Width;
13              var newMatrix = new double[width, height];
14
15              for (int i = 0; i < width; i++)
16              {
17                  for (int j = 0; j < height; j++)
18                  {
19                      int jp, jm, ip, im;
20
21                      jp=(j + height - 1)%height;
22                      jm = (j + 1)%height;
23                      ip= (i + 1)%width;
24                      im = (i + width - 1)%width;
25
26                      newMatrix[i, j] = (matrix[i, jp] +
27                                         matrix[i, jm] +
28                                         matrix[ip, j] +
29                                         matrix[im, j] +
30                                         matrix[ip, jp] +
31                                         matrix[im, jm] +
32                                         matrix[ip, jm] +
33                                         matrix[im, jp]) / 8.0;
34
35                  }
36              }
37              area.Matrix = newMatrix;
38              Bitmap bitmap = new Bitmap(width, height);
39              ColorBitmap(newMatrix, width, height, bitmap);
40              return bitmap;
41          }
42      }
43  }
```

1b. Berechnung im Hintergrund

Um die Berechnung in den Hintergrund zu verlagern, wurden die Statements in der Methode `GenerateImage` in einen Task gekapselt und dann mit `await` auf die Beendigung gewartet. Dadurch wird der UI-Thread nicht blockiert und die GUI bleibt bedienbar. Auf Synchronisierungsprobleme und die Möglichkeit zum Abbrechen wurde bereits bei der Implementierung 1a Rücksicht genommen.

Sourcecode siehe 1.a

1c. Parallele Version

Da die Berechnungen der einzelnen Felder der neuen Matrix vollkommen unabhängig voneinander sind, ist es möglich die Schleifen paralell ablaufen zu lassen. Dabei müssen keine Synchronisierungen durchgeführt werden. Für die Implementierung eignet sich die TPL besonders gut, da hier eine parallele Version der For-Schleife angeboten wird. Da der paralell ausgeführte Code nicht umfangreich ist, wird zusätzlich ein `Partitioner` eingesetzt, da sonst der Overhead zu groß und eventuell die paralelle Version langsamer als die sequenzielle Version sein würde.

Die Klasse `ParallelImageGenerator` ist wieder von `ImageGenerator` abgeleitet und implementiert die Berechnung der neuen Matrix mit Hilfe der TPL.

```
1  using System.Collections.Concurrent;
2  using System.Drawing;
3  using System.Threading.Tasks;
4
5  namespace Diffusions
6  {
7      class ParallelImageGenerator:ImageGenerator
8      {
9          protected override Bitmap GenerateBitmap(Area area)
10         {
11             var matrix = area.Matrix;
12             int height = area.Height;
13             int width = area.Width;
14             var newMatrix = new double[width, height];
15
16             var widthPartitioner = Partitioner.Create(0, width);
17
18             Parallel.ForEach(widthPartitioner,(range, loopstate)=>
19             {
20                 for (int i = range.Item1; i < range.Item2; i++)
21                 {
22
23                     for (int j = 0; j < height; j++)
24                     {
25                         int jp, jm, ip, im;
26
27                         jp = (j + height - 1)%height;
28                         jm = (j + 1)%height;
29                         ip = (i + 1)%width;
30                         im = (i + width - 1)%width;
31
32                         newMatrix[i, j] = (matrix[i, jp] +
33                                            matrix[i, jm] +
34                                            matrix[ip, j] +
35                                            matrix[im, j] +
36                                            matrix[ip, jp] +
37                                            matrix[im, jm] +
```

```
38                                              matrix[ip, jm] +
39                                              matrix[im, jp])/8.0;
40
41                      }
42                  }
43              });
44              area.Matrix = newMatrix;
45              Bitmap bitmap = new Bitmap(width, height);
46              ColorBitmap(newMatrix, width, height, bitmap);
47              return bitmap;
48          }
49      }
50  }
```

Performance

Parameter

- DisplayInterval: 10
- MaxIterations: 200
- Size: 335x224
- kein Reheating

Tabelle 1: Ergebnisse

Simulation		
Lauf	Sync [ms]	Parallel [ms]
1	8566	7870
2	8398	8060
3	8474	7970
4	8469	7900
5	8451	8056
Ø	**8471**	**7971**
Std. Abw.	54,34	77,95

- SpeedUp: 1,06277

Beim Speedup von 1,06277 zeigt, dass Paralellisierung hier nicht viel bringt, da die Berechnung der neuen Matrix nicht wirklich aufwendig ist.

Screenshots

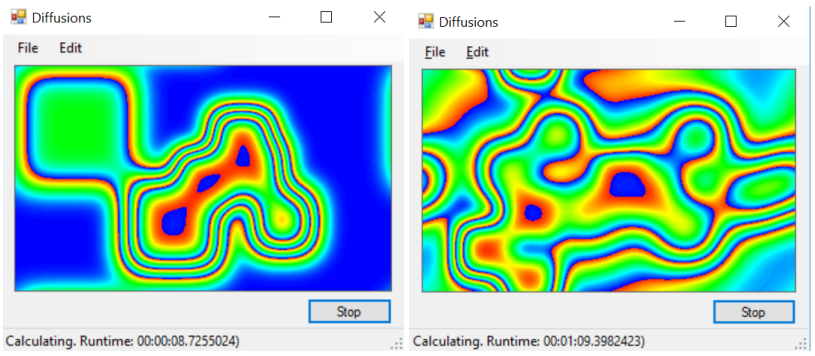


Abbildung 1: Screenshots

2 Stock Data Visualization

2a. Asynchrone Version

Die asynchrone Implementierung verwendet die TPL, um die gewünschte, paralelle Abfolge zu erreichen. Dazu wurden die synchronen Methoden um asynchrone Versionen ergänzt. Diese liefern anstatt des Ergebnisses einen Task zurück, auf den dann gewartet und mit `ContinueWith` weitergearbeitet werden kann. Die asynchrone Implementierung wurde in der Methode `ParallelImplementation` umgesetzt.

Zuerst wird ein Management-Task gestartet, um das GUI nicht zu blockieren. Dieser Task erstellt dann drei weitere Tasks, die paralell die Börsedaten abrufen. Für jeden dieser Tasks werden dann zwei weitere Task gestartet, die paralell die Kurve bzw. den Trend berechnen. Der Management-Task wartet dann auf alle Ergebnisse und zeigt diese dann im GUI an. Dazu wurde dem Task der `Scheduler` des UI-Threads mitgegeben, um die Continuation am UI-Thread auszuführen und somit die GUI-Controls updaten zu können.

Async Versionen der Methoden

```
private Task<StockData> RetrieveStockDataAsync(string name)
{
    return Task.Factory.StartNew(() => RetrieveStockData(name));
}
private Task<Series> GetSeriesAsync(List<StockValue> stockValues, string name)
{
    return Task.Factory.StartNew(() => GetSeries(stockValues, name));
}
private Task<Series> GetTrendAsync(List<StockValue> stockValues, string name)
{
    return Task.Factory.StartNew(() => GetTrend(stockValues, name));
}
```

Paralelle Version

```
private void ParallelImplementation()
{
    displayButton.Enabled = false;
    Task.Factory.StartNew(() =>
    {
        var seriesList = new List<Series>();
        var tasks = new List<Task>();
        foreach (var name in names)
        {
            tasks.Add(RetrieveStockDataAsync(name).ContinueWith(x =>
            {
                var seriesTasks = new Task<Series>[2];
                seriesTasks[0] = GetSeriesAsync(x.Result.GetValues(), x.Result.name);
                seriesTasks[1] = GetTrendAsync(x.Result.GetValues(), x.Result.name);
                Task.WaitAll(seriesTasks);
                seriesList.Add(seriesTasks[0].Result);
                seriesList.Add(seriesTasks[1].Result);
            }));
        }
        Task.WaitAll(tasks.ToArray());
        return seriesList;
    }).ContinueWith(x =>
    {
        DisplayData(x.Result);
        SaveImage("chart");
        displayButton.Enabled = true;
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

2b. Asynchrone Version mit async/await

Die `async/await` Implementierung verwendet wieder die Async-Versionen der Methoden, da diese einen `Task` zurückliefern, auf den mit Hilfe von `await` gewartet werden kann. Die Implementierung ist ähnlich der von 2a, doch anstatt von ContinueWith wird await verwendet. Dazu wird wieder für jede Aktie ein Task erstellt und in eine Liste gespeichert. Jeder dieser Tasks holt sich dann die Börsendaten durch den Aufruf von `RetrieveStockDataAsync`. Mit Hilfe von await wird auf das Ergebnis gewartet. Danach folgen die paralellen Aufrufe von `GetSeriesAsync` und `GetTrendAsync`. Diese werden dann durch den Aufruf von `await` wieder synchronisiert.

Der Aufruf `await Task.WhenAll` wartet schließlich auf den Abschluss aller Haupttasks und danach kann das Ergebnis in der GUI angezeigt werden.

Async/Await Implementierung

```
private async void ParallelAsyncAwaitImplementation()
{
    var seriesList = new List<Series>();
    displayButton.Enabled = false;

    var tasks = names.Select(GetDataAsync).ToList();
    await Task.WhenAll(tasks);
    tasks.ForEach(x => seriesList.AddRange(x.Result));
    DisplayData(seriesList);
    SaveImage("chart");
    displayButton.Enabled = true;
}

private async Task<List<Series>> GetDataAsync(string name)
{
    var seriesList = new List<Series>();
    StockData data = await RetrieveStockDataAsync(name);
    Task<Series> seriesTask = GetSeriesAsync(data.GetValues(),
        data.name);
    Task<Series> trendTask = GetTrendAsync(data.GetValues(),
        data.name);

    seriesList.Add(await seriesTask);
    seriesList.Add(await trendTask);
    return seriesList;
}
```