Übung 4

Table of Contents

1. Setup

Memory size	16,0 GB
CPU type	Intel Core i7-8565U 1.80GHz
Number of cores	4
System	Windows 11 Education N
IDE	Visual Studio 2022

2. Psychedelic Diffusions

2.1. Simulation Logic

Listing 1. SequentialImageGenerator.cs

```
namespace Diffusions.Generators
{
    public class SequentialImageGenerator : ImageGenerator
    {
        protected override void UpdateMatrix(Area area)
        {
            lock (area.Matrix)
            {
                var m = area.Matrix;

                for (int x = 0; x < area.Width; x++)
                {
                    int pX = (x + area.Width - 1) % area.Width;
                    int nX = (x + 1) % area.Width;

                    for (int y = 0; y < area.Height; y++)
                    {
                        int pY = (y + area.Height - 1) % area.Height;
                        int nY = (y + 1) % area.Height;
                        area.NextMatrix[x, y] = (
                            m[pX, pY] + m[pX, y] + m[pX, nY] +
                            m[x, pY] + m[x, nY] +
                            m[nX, pY] + m[nX, y] + m[nX, nY]) / 8;
                    }
                }
                var tmp = area.NextMatrix;
                area.NextMatrix = area.Matrix;
                area.Matrix = tmp;
            }
        }
    }
}
```

2.2. Background Computing

When the method `Start` gets called, a new `Task` gets started. Additionally an `CancellationTokenSource` gets created. When `Stop` gets called, `cancellationTokenSource.Cancel()` gets executed. The loop where the iterations are running, gets canceled and the simulation stops.

Listing 2. ImageGenerator.cs

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

namespace Diffusions.Generators
{
    public abstract class ImageGenerator : IImageGenerator
    {
        public bool Finished { get; protected set; } = false;

        protected CancellationTokenSource cancellationTokenSource;

        public void Start(Area area)
        {
            cancellationTokenSource = new CancellationTokenSource();
            CancellationToken token = cancellationTokenSource.Token;
            Task.Run(() =>
            {
                Finished = false;

                Stopwatch sw = new Stopwatch();
                sw.Start();
                for (int i = 0; i < Settings.Default.MaxIterations && !token
.IsCancellationRequested; i++)
                {
                    UpdateMatrix(area);

                    if (i % Settings.Default.DisplayInterval == 0)
                    {
                        OnImageGenerated(area, ColorSchema.GenerateBitmap(area), sw.Elapsed);
                    }
                }
                sw.Stop();
                Finished = true;
                OnImageGenerated(area, ColorSchema.GenerateBitmap(area), sw.Elapsed);
            }, token);
        }

        public void Stop()
        {
            cancellationTokenSource.Cancel();
        }

        protected abstract void UpdateMatrix(Area area);

        public event EventHandler<EventArgs<Tuple<Area, Bitmap, TimeSpan>>> ImageGenerated;
        protected void OnImageGenerated(Area area, Bitmap bitmap, TimeSpan timespan)
        {
            var handler = ImageGenerated;
            if (handler != null) handler(this, new EventArgs<Tuple<Area, Bitmap, TimeSpan>>(new
Tuple<Area, Bitmap, TimeSpan>(area, bitmap, timespan)));
        }

    }
}
```

2.3. Parallel Version

Fore the parallel execution, `Parallel.forEach` gets used instead of a normal loop. The calculation gets splitted into tiny parts of work. For the separation into separate parts, a `Partitioner` gets used which splits the width for the calculation.

Listing 3. ParallelImageGenerator.cs

```
using System.Collections.Concurrent;
using System.Threading.Tasks;

namespace Diffusions.Generators
{
    public class ParallelImageGenerator : ImageGenerator
    {
        protected override void UpdateMatrix(Area area)
        {
            lock (area.Matrix)
            {
                var m = area.Matrix;
                var partitioner = Partitioner.Create(0, area.Width);

                Parallel.ForEach(partitioner, (partRange, _) =>
                {
                    for (int x = partRange.Item1; x < partRange.Item2; x++)
                    {
                        int pX = (x + area.Width - 1) % area.Width;
                        int nX = (x + 1) % area.Width;

                        for (int y = 0; y < area.Height; y++)
                        {
                            int pY = (y + area.Height - 1) % area.Height;
                            int nY = (y + 1) % area.Height;
                            area.NextMatrix[x, y] = (
                            m[pX, pY] + m[pX, y] + m[pX, nY] +
                            m[x, pY] + m[x, nY] +
                            m[nX, pY] + m[nX, y] + m[nX, nY]) / 8;
                        }
                    }
                });
                var tmp = area.NextMatrix;
                area.NextMatrix = area.Matrix;
                area.Matrix = tmp;
            }
        }
    }
}
```

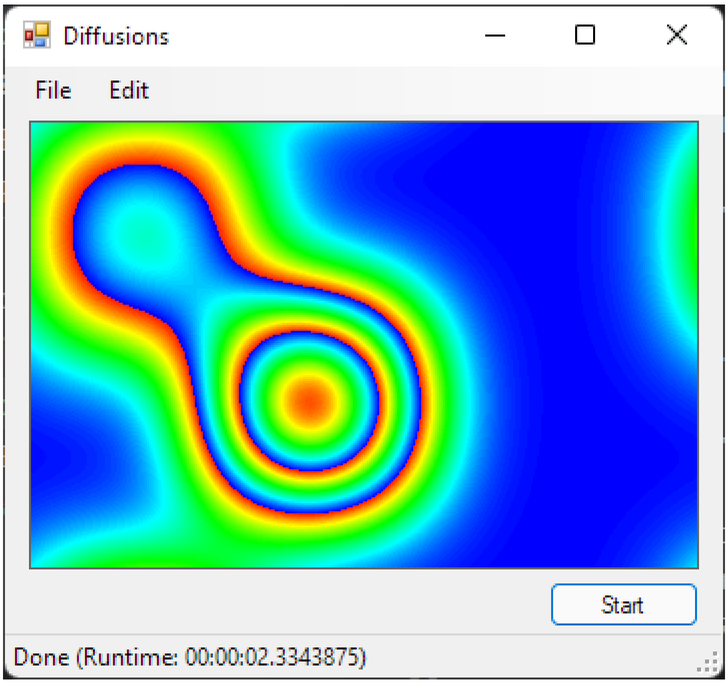

Figure 1. Parallel result

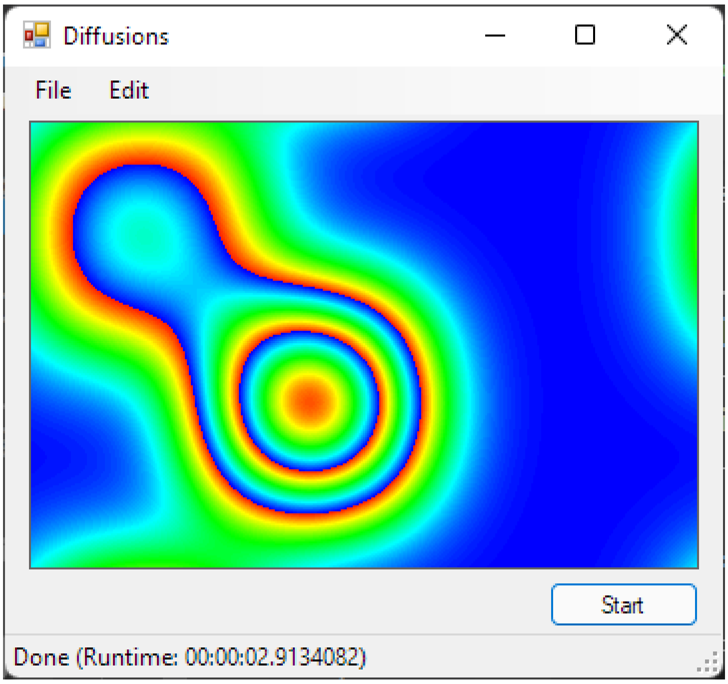

Figure 2. Sequential result

2.3.1. Speedup

Speedup = 2,9134082 / 2,334388 = 1,24804

The parallel version is 24,804 percent faster than the sequential one.

3. Parallel Numerical Integration

3.1. C++ OpenMP

Listing 4. ParallelImageGenerator.cs

```
double integrateOMP(double min, double max, int steps) {
    double stepSize = (max - min) / steps;
    double result = 0.0;

#pragma omp parallel for reduction(+:result)
    for (int i = 0; i < steps; i++) {
        result += f(min + stepSize * i) * stepSize;
    }
    return result;
```

3.2. C# .NET Task Parallel Library

The calculation gets separated into steps with the `Partitioner`, similar to the the OpenMP solution. Additionally `Parallel.ForEach` gets called, the initial state gets set to zero. For the second part all partitioned parts gets calculated separately. The result gets added to the total result. To prevent race conditions, an lock object is needed when a part sum gets added to the total sum.

Listing 5. Program.cs

```
private static double IntegrateParallel(double min, double max, int steps)
{
    double stepSize = (max - min) / steps;
    double totalResult = 0.0;

    object lockObj = new();
    var partitioner = Partitioner.Create(0, steps);
    Parallel.ForEach(partitioner,
        () => 0.0, // Initial state
        (range, _, startValue) =>
        {
            double result = startValue;
            for (int i = range.Item1; i < range.Item2; i++)
            {
                result += F(min + i * stepSize) * stepSize;
            }
            return result;
        },
        result =>
        {
            lock (lockObj) // Lock sum obj
            {
                totalResult += result; // Add to total sum
            }
        });

    return totalResult;
}
```

Both sequential and parallel versions of the C# implementation are slower than the C++ version. With low step size the sequential version are much faster. There is obviously more overhead with the .NET Parallel task library and the C# version. Although with higher results the time for the parallel execution is nearly the same.

C#				
StepSize	ResultSeq	TimeSeq	ResultPar	TimePar
65	3,156937821	222	3,156937821	209
650	3,143130721	2	3,143130721	81
6500	3,141746496	31	3,141746496	105
65000	3,141608038	201	3,141608038	364
650000	3,141594192	2071	3,141594192	2770
6500000	3,141592807	19208	3,141592807	27551
65000000	3,141592669	181979	3,141592669	27968

C++				
StepSize	ResultSeq	TimeSeq	ResultPar	TimePar
65	3,15694	0	3,15694	53
650	3,14313	1	3,14313	12
6500	3,14175	11	3,14175	6
65000	3,14161	111	3,14161	38
650000	3,14159	1103	3,14159	200
6500000	3,14159	11044	3,14159	2155
65000000	3,14159	108094	3,14159	21584

Figure 3. Statistics