VPS 5	Parallel and Distributed Software Systems	WT 20/21, Exercise 5

Deadline: 15.02.2022 00:00

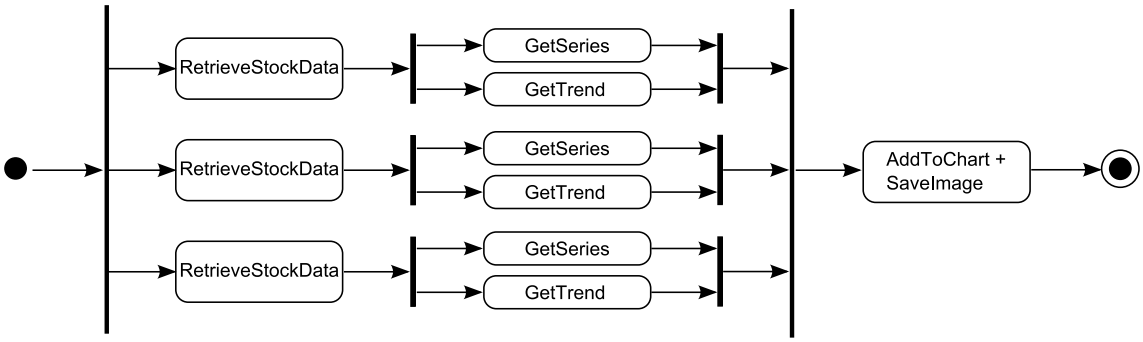
Name Andreas Wenzelhuemer

Points _____ **Effort in hours** 5

1. 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.

2. Parallel Wator (8 + 4 Points)

Implement a parallel version of the Wator simulation from the first exercise using the .NET Task Parallel Library. You can parallelize either your already optimized version or the original version provided on Moodle.

a) Think about how you can separate data and work of the Wator simulation to allow parallel processing and consider appropriate synchronization. Describe and implement your decomposition and parallelization concept and explain your approach and its advantages and maybe also disadvantages.

b) Test your parallel Wator simulation, compare its runtime with your optimized version and the original version from Exercise 1 and calculate its speedup.

Übung 5

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. Stock Data Visualization

Async implementation of the stock visualization. The version with the async and await keywords is much shorter than the other one.

Listing 1. Wrapped methods

```
private Task<StockData> RetrieveStockDataAsync(string name)
{
    return Task.Run(() => RetrieveStockData(name));
}

private Task<Series> GetSeriesAsync(List<StockValue> stockValues, string name)
{
    return Task.Run(() => GetSeries(stockValues, name));
}

private Task<Series> GetTrendAsync(List<StockValue> stockValues, string name)
{
    return Task.Run(() => GetTrend(stockValues, name));
}
```

2.1. Version with .NET Task Parallel Library

Listing 2. Version 1

```
private void TaskImplementation()
{
    var tasks = names.Select(name => RetrieveStockDataAsync(name).ContinueWith((t) =>
    {
        var sd = t.Result;
        var values = sd.GetValues();
        var series = GetSeriesAsync(values, name);
        var trend = GetTrendAsync(values, name);
        return Task.WhenAll(series, trend);
    }));

    Task.WhenAll(tasks).ContinueWith(t =>
    {
        var series = t.Result;
        Task.WhenAll(series).ContinueWith(t1 =>
        {
            var result = t1.Result.SelectMany(x => x).ToList();
            DisplayData(result);
            SaveImage("chart");
        });
    });
}
```

2.2. Version with async and await

Listing 3. Version 2

```
        private async Task AsyncImplementation()
        {
            var tasks = names.Select(async name =>
            {
                var sd = await RetrieveStockDataAsync(name);
                var values = sd.GetValues();
                var series = GetSeriesAsync(values, name);
                var trend = GetTrendAsync(values, name);
                return await Task.WhenAll(series, trend);
            });

            var seriesList = (await Task.WhenAll(tasks)).SelectMany(x => x).ToList();
            DisplayData(seriesList);
            SaveImage("chart");
        }
```

3. Parallel Wator

3.1. Idea and implementation

The simulation can be parallelized by creating parts which get calculated simultaneously. The height of the section has to be at least 4 otherwise there would be race conditions.

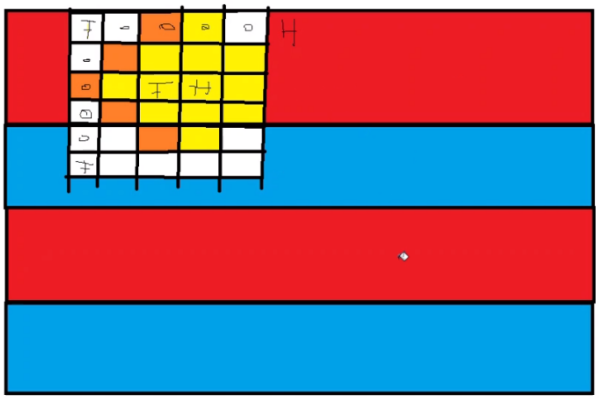


Figure 1. Splitting into Partitions

The separation is done with the help of a partitioner. It is important that the count of the partitions is even because otherwise there would be race conditions with the first and the last rows.

Listing 4. Partitions

```
// Create partitions with size of 4
for (int partitionSize = 4; partitionSize < Height; partitionSize++)
{
    partitioner = Partitioner.Create(0, Height, partitionSize);
    int partitionCount = partitioner.GetDynamicPartitions().ToList().Count();

    // Count of partitions has to be even otherwise there would be a race condition
    if (partitionCount % 2 == 0)
    {
        break;
    }
}
```

The random matrix gets not randomized in the constructor anymore because only parts get randomized not everything.

Listing 5. Matrix initialization

```
            // Don't randomize matrix initially
            randomMatrix = GenerateMatrix(Width, Height, randomize: false);
```

First, all even partitions get executed. After the execution the uneven partitions get executed and the commit gets called. The method itself gets called in `ExecuteStep()` with `Task.Run(ExecuteStepAsync).Wait()`.

Listing 6. Step execution

```
        // Execute one time step of the simulation. Each cell of the world must be executed
once.
        // Animals move around on the grid. To make sure each animal is executed only once we
        // use the moved flag.
        private async Task ExecuteStepAsync()
        {
            var partitions = partitioner.GetDynamicPartitions().ToList();
            ICollection<Task> tasks = new List<Task>();

            // Execute even partitions
            for (int i = 0; i < partitions.Count; i += 2)
            {
                tasks.Add(ExecutePartitionAsync(partitions[i].Item1, partitions[i].Item2));
            }

            await Task.WhenAll(tasks);
            tasks.Clear();

            // Execute uneven partitions
            for (int i = 1; i < partitions.Count; i += 2)
            {
                tasks.Add(ExecutePartitionAsync(partitions[i].Item1, partitions[i].Item2));
            }

            await Task.WhenAll(tasks);

            // Commit all animals in the grid to prepare for the next simulation step
            for (int col = 0; col < Width; col++)
            {
                for (int row = 0; row < Height; row++)
                {
                    Grid[GetGridIndex(row, col)]?.Commit();
                }
            }
        }
```

The execution of the partition is similar to the previous algorithm. The difference is that randomize matrix gets called for each partition and a lower and upper height limit is needed.

Listing 7. Partition execution

```
        private Task ExecutePartitionAsync(int lowerHeight, int upperHeight)
        {
            return Task.Run(() =>
            {
                RandomizeMatrix(randomMatrix, lowerHeight, upperHeight);  // make sure that
order of execution of cells is different and random in each time step

                // process all animals in random order
                int randomRow, randomCol;
                for (int col = 0; col < Width; col++)
                {
                    // Only execute steps in the given bounds
                    for (int row = lowerHeight; row < upperHeight; row++)
                    {
                        // get random position (row/colum) from random matrix
                        randomCol = randomMatrix[GetGridIndex(row, col)] % Width;
                        randomRow = randomMatrix[GetGridIndex(row, col)] / Width;

                        var animal = Grid[GetGridIndex(randomRow, randomCol)];

                        if (animal != null && !animal.Moved)  // process unmoved animals
                            animal.ExecuteStep();
                    }
                }
            });
        }
```

The matrix randomization method got two parameters for the lower and upper height limit because only the part should gets shuffled.

Listing 8. Randomize matrix

```
        private void RandomizeMatrix(int[] array, int lowerHeight, int upperHeight)
        {
            // perform Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
            for (int i = lowerHeight * Width; i < upperHeight * Width; i++)
            {
                int result = random.Next(i, Width * upperHeight);
                int temp = array[result];
                array[result] = array[i];
                array[i] = temp;
            }
        }
    }
}
```

Advantages:

- Nearly twice as fast than the previous version
- Easy implementation and not much to change

Disadvantages:

- No complete randomness because only parts get randomized

3.2. Performance

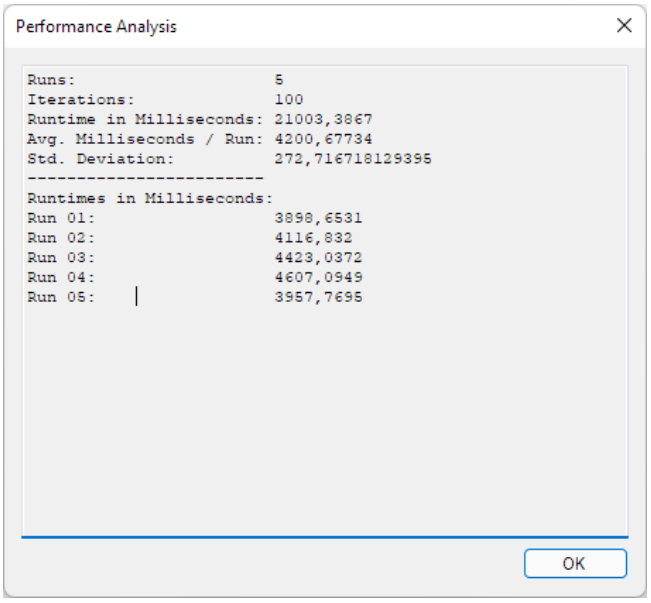


Figure 2. Performance (Original Version)

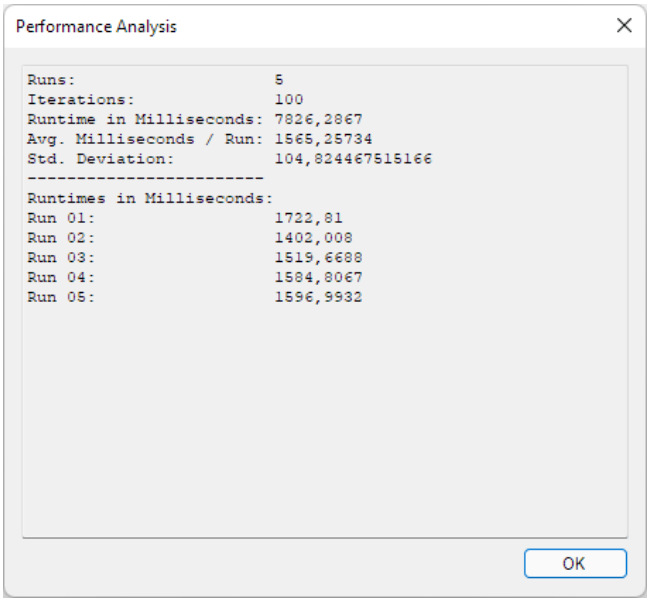


Figure 3. Performance (Async Version)

Version	Total duration	Speedup with Original	Speedup with Previous
Original	19,097	1	1
Improved 1	16,781	1.137991292	1.137991292
Improved 2	15,908	1.200482862	1.054913926
Improved 3	14757.6193	1.294023454	1.07791914
Async	7,826	2.440072314	1.885647672

Figure 4. Speedup

This version is nearly twice as fast than the last optimized.