VPS 5 | **Parallel and Distributed Software Systems** | **WT 21/22, Exercise 1**

Deadline: 15.11.2021 15:00

Name Andreas Wenzelhuemer

Points ________ **Effort in hours** 4

1. Theory – Amuse-Gueule … (2 + 2 + 4 Points)

Assume a given algorithm, which solves a problem of size *p* in parallel. For a problem size of $p = 10$, the algorithm has a relative sequential part $\sigma = 0.2$ (i.e., 20% of the algorithm cannot be parallelized).

a) Calculate and plot speedup and efficiency, which can be achieved for this algorithm with increasing numbers of processors *n* (i.e., cores). What is the upper limit of the speedup?

Further assume, that the sequential part of the algorithm has an asymptotic runtime complexity of $O(p)$ and the parallel part of the algorithm as an asymptotic runtime complexity of $O(p^2)$.

b) Calculate and plot the relative sequential part σ with increasing problem sizes.

c) For a problem size of $p = 100$, 1.000 and 10.000, how many processors can be utilized, if the efficiency has to be above 80%?

2. Wator – Eat or be eaten … (4 + 12 Points)

Wator is the name of a small circular planet, far far away from our galaxy, were no one has ever gone before. On Wator there live two different kinds of species: *sharks* and *fish*. Both species live according to a very old set of rules, which has not been changed for the last thousands of years.

For **fish** the rules are:

- at the beginning of all time there were *f* fish
- each fish has a constant energy E_f
- in each time step a fish moves randomly to one of its four adjacent cells (up, down, left or right), if and only if there is a free cell available
- if all adjacent cells are occupied, the fish doesn't move
- in each time step fish age by one time unit
- if a fish gets older than a specified limit B_f, the fish breeds (i.e., a new fish is born on a free adjacent cell, if such a cell is available)
- after the birth of a new fish the age of the parent fish is reduced by B_f

For **sharks** the rules are:

- at the beginning of all time there were *s* sharks, each with an initial energy of E_s
- in each time step a sharks consumes one energy unit
- in each time step a shark eats a fish, if a fish is on one of its adjacent cells
- if a shark eats a fish, the energy of the shark increases by the energy value of the eaten fish
- if there is no fish adjacent to the shark, the shark moves like a fish to one of its neighbor cells
- if the energy of a shark gets 0, the shark dies
- if the energy of a shark gets larger than a specified limit B_s, the shark breeds and the energy of the parent shark is equally distributed among the parent and the child shark (i.e., a new shark is born on a free adjacent cell, if such a cell is available)

a) On Moodle, you find a ready to use implementation of Wator. Make a critical review of the application and analyze its design, efficiency, clarity, readability, etc. **Document your review results properly.**

b) Change the application gradually to improve its performance. Think of **three concrete improvements** and implement them. For each improvement, document how the runtime changes (in comparison to the prior and to the initial version) and calculate the speedup. Each single optimization should yield a speedup of at least 1.05 compared to the prior version.

For the experiments in Task b) use the following settings:

Fish Settings:	
FishBreedTime	10
InitialFishEnergy	10
InitialFishPopulation	20.000

General Settings:	
DisplayWorld	**False**
Height	500
Iterations	100
Runs	5
Width	500
Workers	1

Shark Settings:	
InitialSharkEnergy	50
InitialSharkPopulation	5.000
SharkBreedEnergy	100

Notes: Improvements must not alter the simulation’s inherent logic (i.e. stick to the listed rules and do not remove simulation logic, such as iteration-wise random execution order).

In this and all upcoming exercises always document your system configuration (i.e., number of cores, memory size, CPU type, etc.) when performing runtime measurements.

Übung 1

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 2019

2. Theory

2.1. Calculate and plot speedup and efficiency

Speedup = 1 / (+ (1 - Sigma) / Cores)

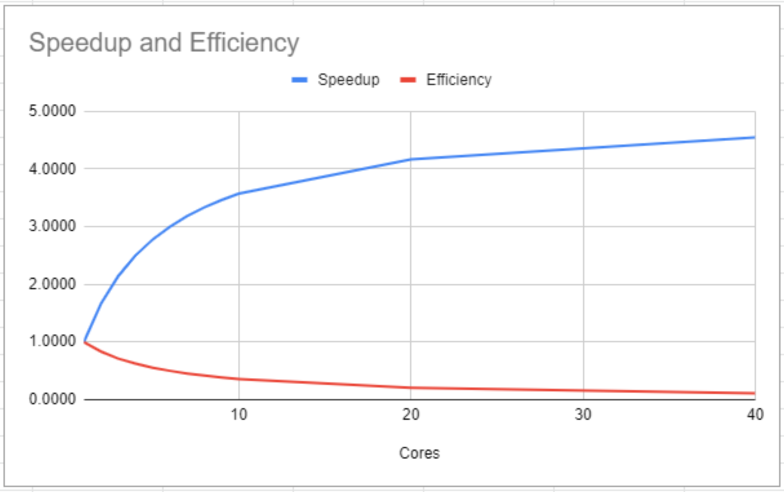
Efficiency = Speedup / Cores

Seq: 0.2

Par: 0.8

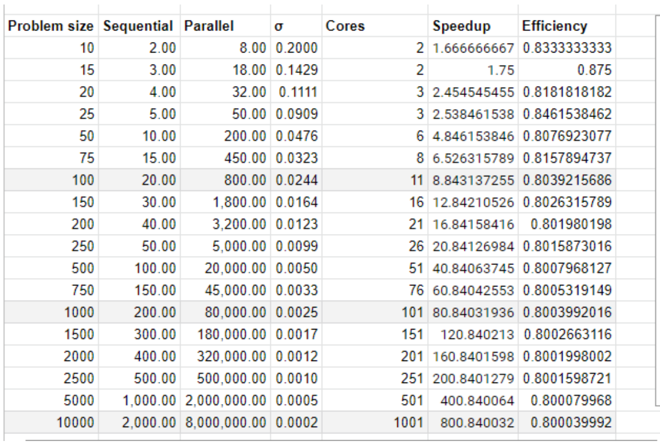
Cores: n

σ	0.2	Sequential part
p	10	
Cores	Speedup	Efficiency
1	1.0000	1.0000
2	1.6667	0.8333
3	2.1429	0.7143
4	2.5000	0.6250
5	2.7778	0.5556
6	3.0000	0.5000
7	3.1818	0.4545
8	3.3333	0.4167
9	3.4615	0.3846
10	3.5714	0.3571
20	4.1667	0.2083
40	4.5455	0.1136
80	4.7619	0.0595
100	4.8077	0.0481
1000	4.9801	0.0050
10000	4.9980	0.0005

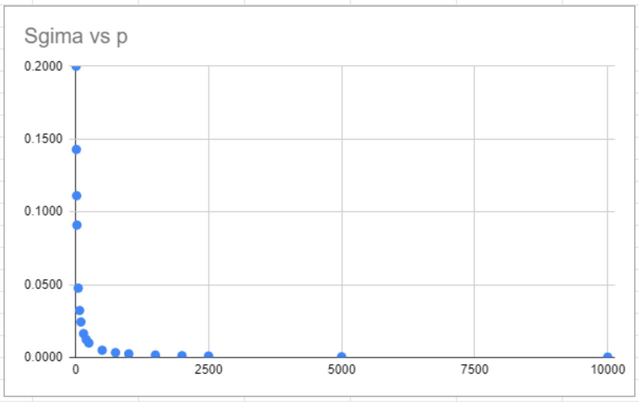


The limit for speed up is 5.

2.2. Calculate and plot Sigma with increasing problem sizes



Problem size	Sequential	Parallel	σ	Cores	Speedup	Efficiency
10	2.00	8.00	0.2000	2	1.666666667	0.8333333333
15	3.00	18.00	0.1429	2	1.75	0.875
20	4.00	32.00	0.1111	3	2.454545455	0.8181818182
25	5.00	50.00	0.0909	3	2.538461538	0.8461538462
50	10.00	200.00	0.0476	6	4.846153846	0.8076923077
75	15.00	450.00	0.0323	8	6.526315789	0.8157894737
100	20.00	800.00	0.0244	11	8.843137255	0.8039215686
150	30.00	1,800.00	0.0164	16	12.84210526	0.8026315789
200	40.00	3,200.00	0.0123	21	16.84158416	0.801980198
250	50.00	5,000.00	0.0099	26	20.84126984	0.8015873016
500	100.00	20,000.00	0.0050	51	40.84063745	0.8007968127
750	150.00	45,000.00	0.0033	76	60.84042553	0.8005319149
1000	200.00	80,000.00	0.0025	101	80.84031936	0.8003992016
1500	300.00	180,000.00	0.0017	151	120.840213	0.8002663116
2000	400.00	320,000.00	0.0012	201	160.8401598	0.8001998002
2500	500.00	500,000.00	0.0010	251	200.8401279	0.8001598721
5000	1,000.00	2,000,000.00	0.0005	501	400.840064	0.800079968
10000	2,000.00	8,000,000.00	0.0002	1001	800.840032	0.800039992



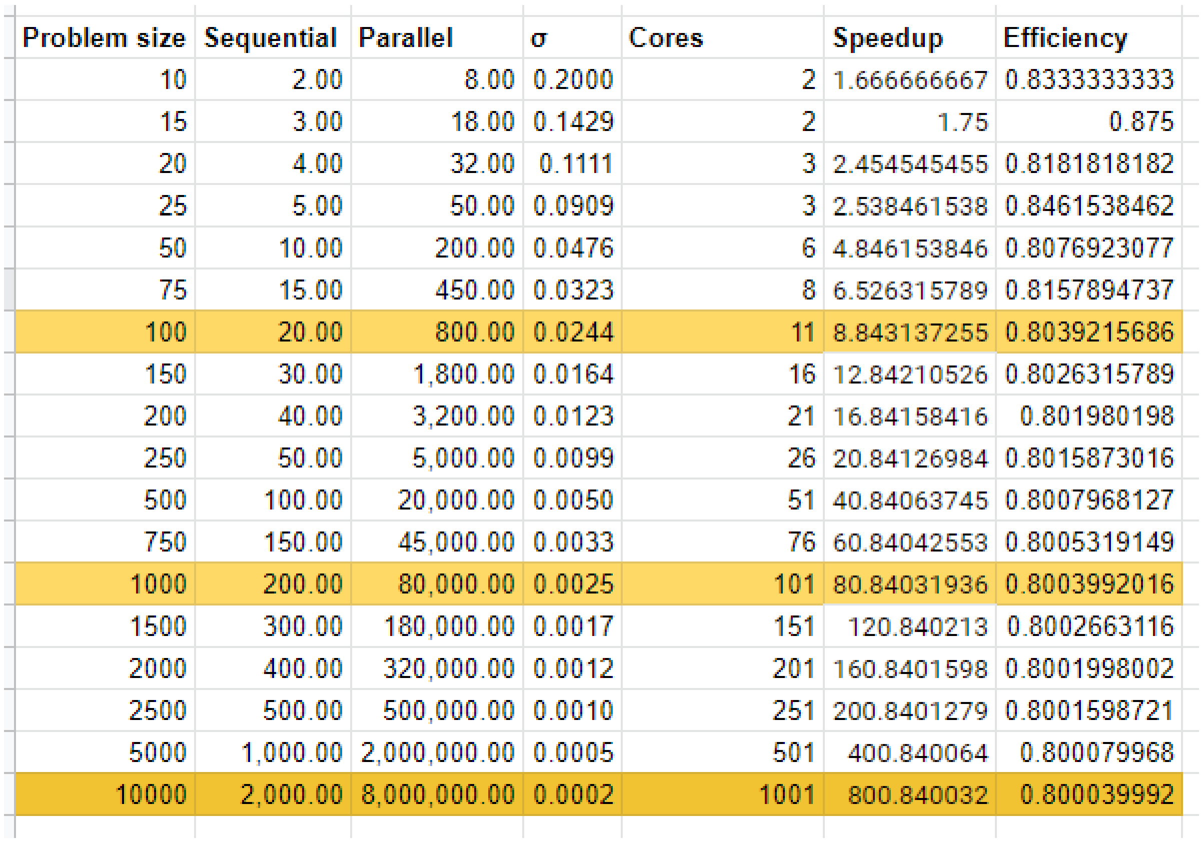
With a problem size of 10000 the sequential part gets extremely small.

2.3. How many processors can be utilized?

p = 100: 11 cores are needed

p = 1000: 100 cores are needed

p = 10000: With 1000 cores an efficiency of 80% can be reached



Problem size	Sequential	Parallel	σ	Cores	Speedup	Efficiency
10	2.00	8.00	0.2000	2	1.666666667	0.8333333333
15	3.00	18.00	0.1429	2	1.75	0.875
20	4.00	32.00	0.1111	3	2.454545455	0.8181818182
25	5.00	50.00	0.0909	3	2.538461538	0.8461538462
50	10.00	200.00	0.0476	6	4.846153846	0.8076923077
75	15.00	450.00	0.0323	8	6.526315789	0.8157894737
100	20.00	800.00	0.0244	11	8.843137255	0.8039215686
150	30.00	1,800.00	0.0164	16	12.84210526	0.8026315789
200	40.00	3,200.00	0.0123	21	16.84158416	0.801980198
250	50.00	5,000.00	0.0099	26	20.84126984	0.8015873016
500	100.00	20,000.00	0.0050	51	40.84063745	0.8007968127
750	150.00	45,000.00	0.0033	76	60.84042553	0.8005319149
1000	200.00	80,000.00	0.0025	101	80.84031936	0.8003992016
1500	300.00	180,000.00	0.0017	151	120.840213	0.8002663116
2000	400.00	320,000.00	0.0012	201	160.8401598	0.8001998002
2500	500.00	500,000.00	0.0010	251	200.8401279	0.8001598721
5000	1,000.00	2,000,000.00	0.0005	501	400.840064	0.800079968
10000	2,000.00	8,000,000.00	0.0002	1001	800.840032	0.800039992

3. Wator

3.1. Review of the application

Methods: The Methods are too long and not very easy to read. For example the GetNeighbour-Method has a lot of redundant operations which could be easily simplified. Also there is the possibility to extract parts of the methods into separate ones to improve readability.

Technology: WinForms is an old microsoft technology from microsoft for desktop development. Nowadays far better technologies exist for such a purpose. Additionally the application works only on windows and doesn't be run on other operation systems.

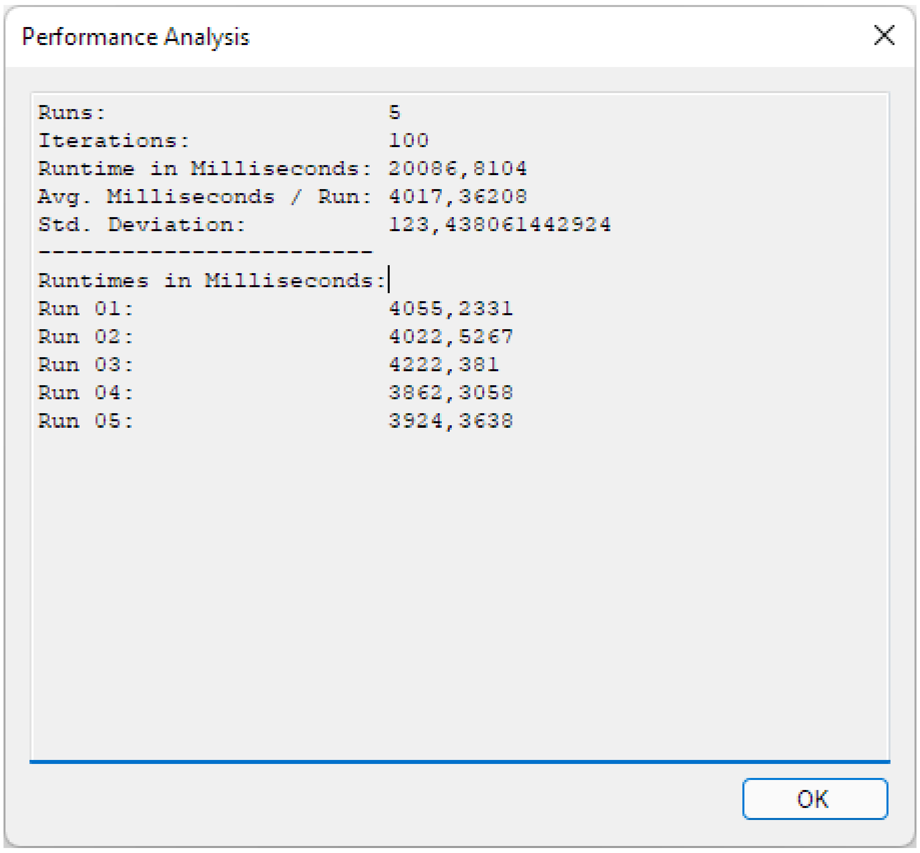
Settings should not be hardcoded, instead they should be available over an general settings file.

The application also has a lot of potential for performance improvements which will be addressed in the next chapter.

3.2. Three performance improvements

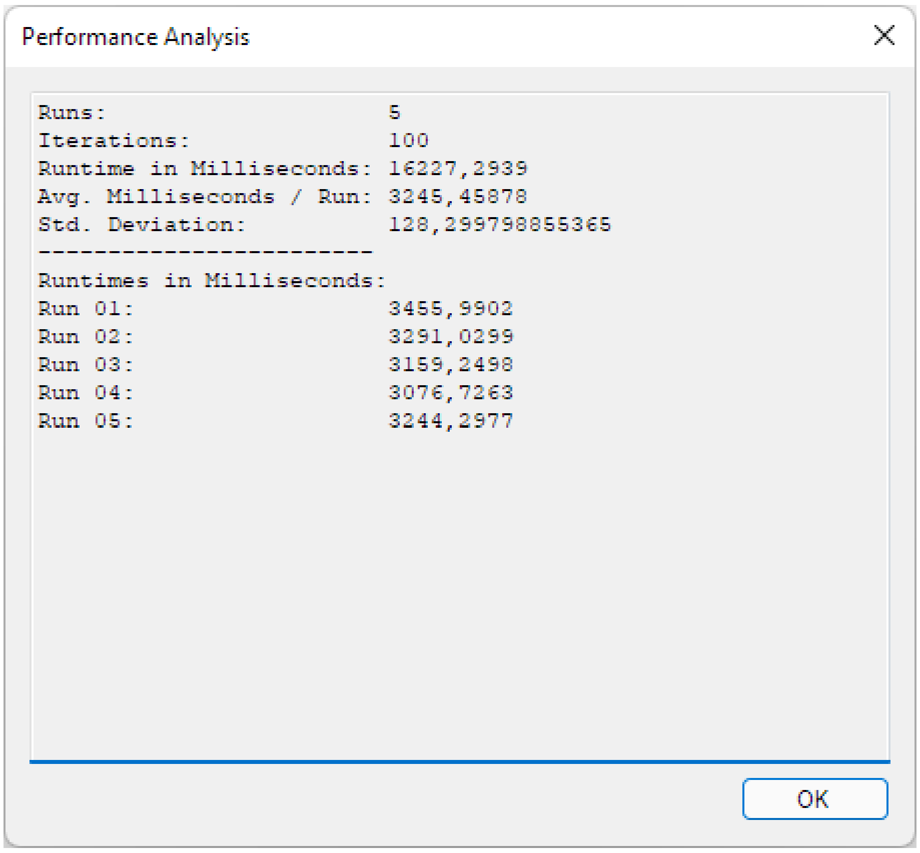
3.2.1. Improvement 1

Changed point array of GetNeighbours to List. Also introduced it globally to reduce the work of the garbage collector. Otherwise each time a new array has to be created. The list gets reused every time.



```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   20086,8104
Avg. Milliseconds / Run:   4017,36208
Std. Deviation:            123,438061442924
------------------------
Runtimes in Milliseconds:
Run 01:                    4055,2331
Run 02:                    4022,5267
Run 03:                    4222,381
Run 04:                    3862,3058
Run 05:                    3924,3638
```

Figure 1. Original



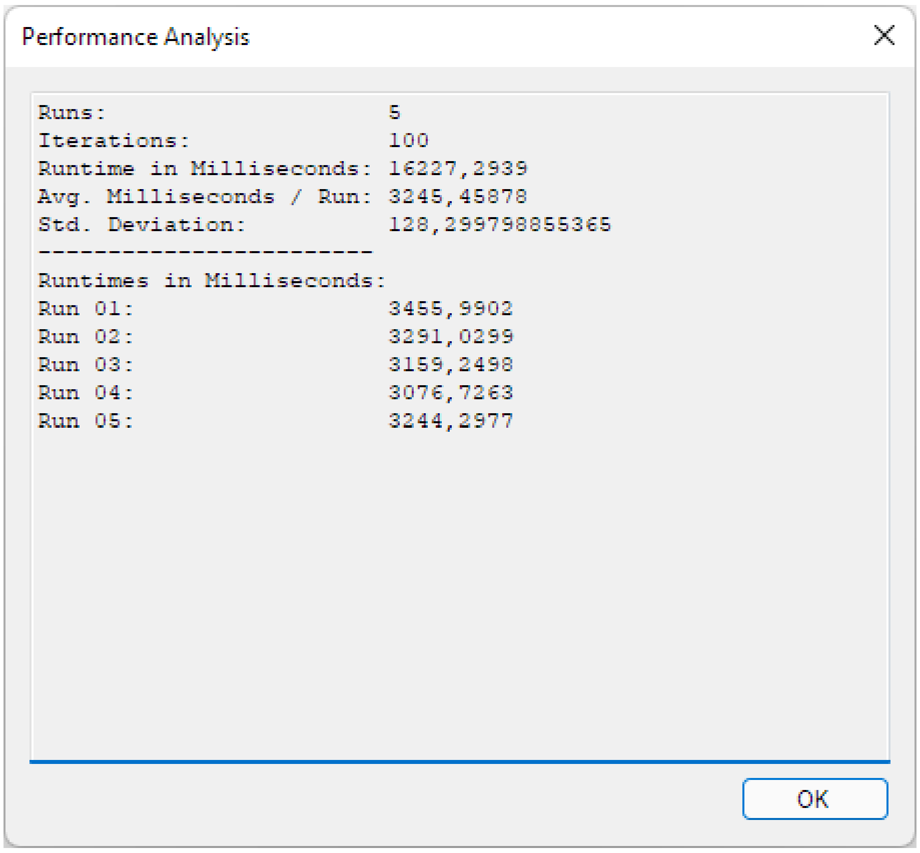
```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   16227,2939
Avg. Milliseconds / Run:   3245,45878
Std. Deviation:            128,299798855365
------------------------
Runtimes in Milliseconds:
Run 01:                    3455,9902
Run 02:                    3291,0299
Run 03:                    3159,2498
Run 04:                    3076,7263
Run 05:                    3244,2977
```

Figure 2. Improved

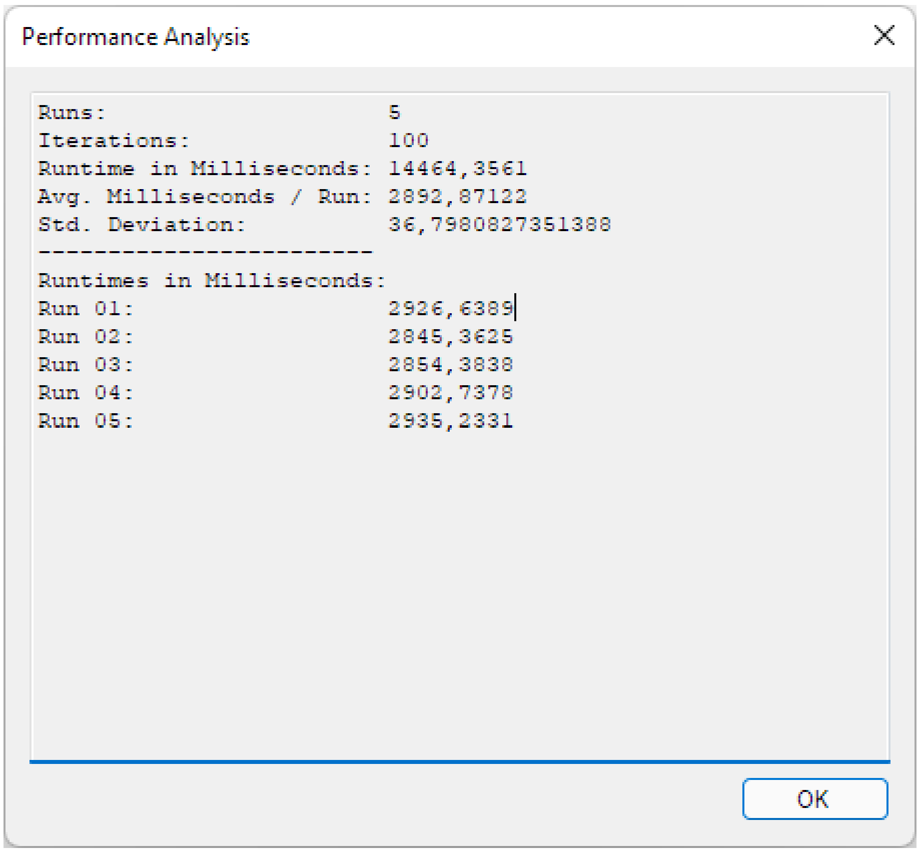
```
public IList<Point> GetNeighbours(Type type, Point position)
{
    points.Clear();
    int i, j;

    // look north
    i = position.X;
    j = (position.Y + Height - 1) % Height;
    if (type == null && Grid[i, j] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[i, j]))
    {
        if (Grid[i, j] != null && !Grid[i, j].Moved)
        {  // ignore animals moved in the current iteration
            points.Add(new Point(i, j));
        }
    }
    // ...
}
```

3.2.2. Improvement 2

Changed all two dimensional matrices do one dimensional. That means that the animal board and the the matrix for the random positioning are only simple arrays where the index get calculated with a special function. Improvements can be seen especially on the ExecuteStep and RandomizeMatrix functions.



```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   16227,2939
Avg. Milliseconds / Run:   3245,45878
Std. Deviation:            128,299798855365
------------------------
Runtimes in Milliseconds:
Run 01:                    3455,9902
Run 02:                    3291,0299
Run 03:                    3159,2498
Run 04:                    3076,7263
Run 05:                    3244,2977
```



```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   14464,3561
Avg. Milliseconds / Run:   2892,87122
Std. Deviation:            36,7980827351388
------------------------
Runtimes in Milliseconds:
Run 01:                    2926,6389
Run 02:                    2845,3625
Run 03:                    2854,3838
Run 04:                    2902,7378
Run 05:                    2935,2331
```

Figure 3. Prior

Listing 1. Improved

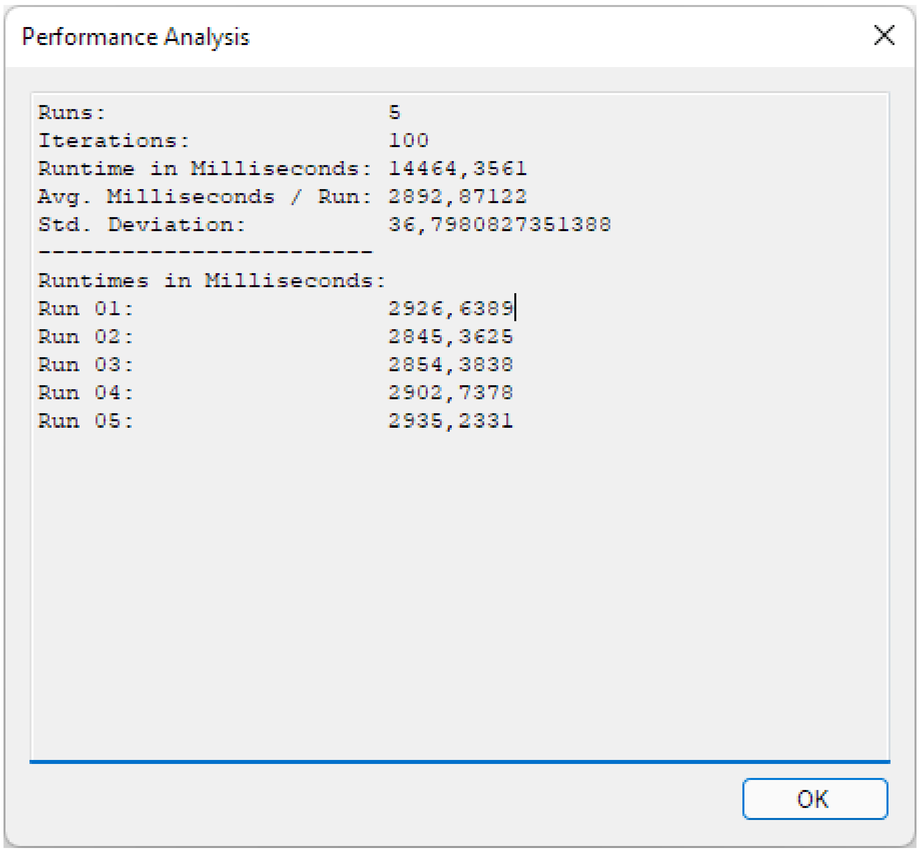
```
private int[] randomMatrix;
```

```
public int GetGridIndex(int row, int column)
{
    return row * Width + column;
}

// shuffle values of the matrix
private void RandomizeMatrix(int[] matrix)
{
    // perform Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
    int temp, selectedRow, selectedCol;

    int row = 0;
    int col = 0;
    for (int i = 0; i < Height * Width; i++)
    {
        temp = matrix[GetGridIndex(row, col)];

        // select random element from remaining elements
        // already processed elements must not be chosen a second time
        selectedRow = random.Next(row, Height);
        if (selectedRow == row) selectedCol = random.Next(col, Width);  // current row selected
-> select from remaining columns
        else selectedCol = random.Next(Width);  // new row selected -> select any column

        // swap
        matrix[GetGridIndex(row, col)] = matrix[GetGridIndex(selectedRow, selectedCol)];
        matrix[GetGridIndex(selectedRow, selectedCol)] = temp;

        // incremet col and row
        col++;
        if (col >= Width) { col = 0; row++; }
    }
}

public IList<Point> GetNeighbors(Type type, Point position)
{
    points.Clear();
    int i, j;

    // look north
    i = position.X;
    j = (position.Y + Height - 1) % Height;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
    {
        if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
        {  // ignore animals moved in the current iteration
            points.Add(new Point(i, j));
        }
    }
    // look east
    i = (position.X + 1) % Width;
    j = position.Y;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
```

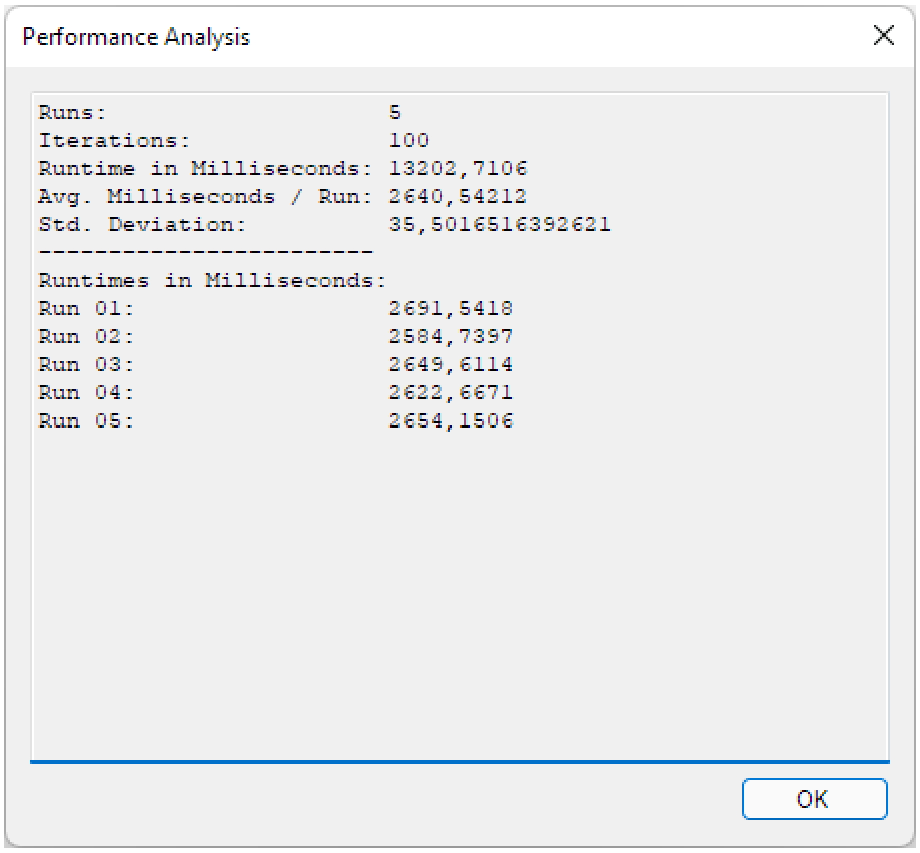
```
        else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
        {
            if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
            {
                points.Add(new Point(i, j));
            }
        }
        // look south
        i = position.X;
        j = (position.Y + 1) % Height;
        if (type == null && Grid[GetGridIndex(j, i)] == null)
        {
            points.Add(new Point(i, j));
        }
        else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
        {
            if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
            {
                points.Add(new Point(i, j));
            }
        }
        // look west
        i = (position.X + Width - 1) % Width;
        j = position.Y;
        if (type == null && Grid[GetGridIndex(j, i)] == null)
        {
            points.Add(new Point(i, j));
        }
        else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
        {
            if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
            {
                points.Add(new Point(i, j));
            }
        }

        return points;
    }
```

3.2.3. Improvement 3

Used more performant version of the Knuth Shuffle. Also improved memory consumption of sharks by removing the second division.



```
Runs:                        5
Iterations:                  100
Runtime in Milliseconds:     14464,3561
Avg. Milliseconds / Run:     2892,87122
Std. Deviation:              36,7980827351388
------------------------
Runtimes in Milliseconds:
Run 01:                      2926,6389
Run 02:                      2845,3625
Run 03:                      2854,3838
Run 04:                      2902,7378
Run 05:                      2935,2331
```



```
Runs:                        5
Iterations:                  100
Runtime in Milliseconds:     13202,7106
Avg. Milliseconds / Run:     2640,54212
Std. Deviation:              35,5016516392621
------------------------
Runtimes in Milliseconds:
Run 01:                      2691,5418
Run 02:                      2584,7397
Run 03:                      2649,6114
Run 04:                      2622,6671
Run 05:                      2654,1506
```

Figure 4. Prior

Listing 2. Improvement

```
private void RandomizeMatrix(int[] array)
{
    // perform Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
    int size = array.Length;
    for (int i = 0; i < (size - 2); i++)
    {
        int result = random.Next(i, size);
        int temp = array[result];
        array[result] = array[i];
        array[i] = temp;
    }
}
public class Shark : Animal
{
    // spawning behaviour of sharks
    protected override void Spawn()
    {
        Point free = World.SelectNeighbor(null, Position);  // find a random empty neighboring
cell
        if (free.X != -1)
        {
            // empty neighboring cell found -> create new shark there and share energy between
parent and child shark
            Energy /= 2;
            new Shark(World, free, Energy);
        }
    }
}
```