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

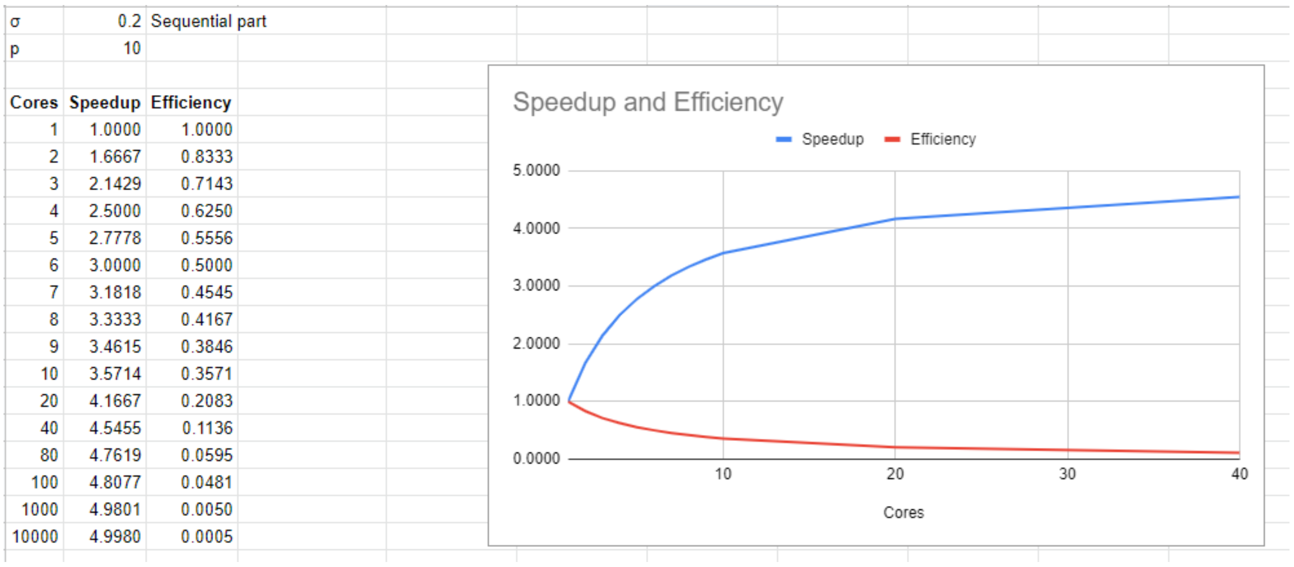
```
Number of cores: 4
Memory size 16,0 GB
CPU type: Intel Core i7-8565U 1.80GHz
System: Windows 11
```

2. Theory

2.1. Calculate and plot speedup and efficiency

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

Seq: 0.2
Par: 0.8
Cores: n

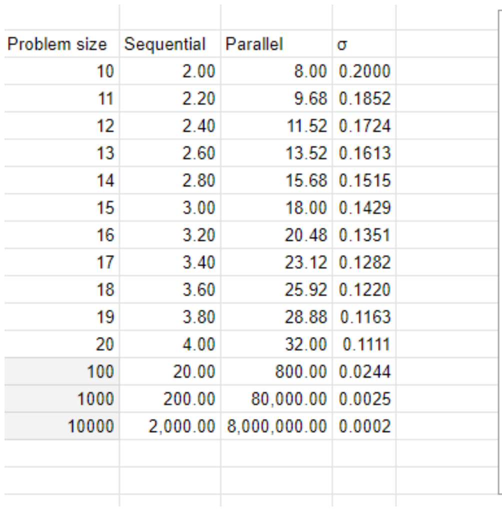


σ	0.2	Sequential part
p	10	

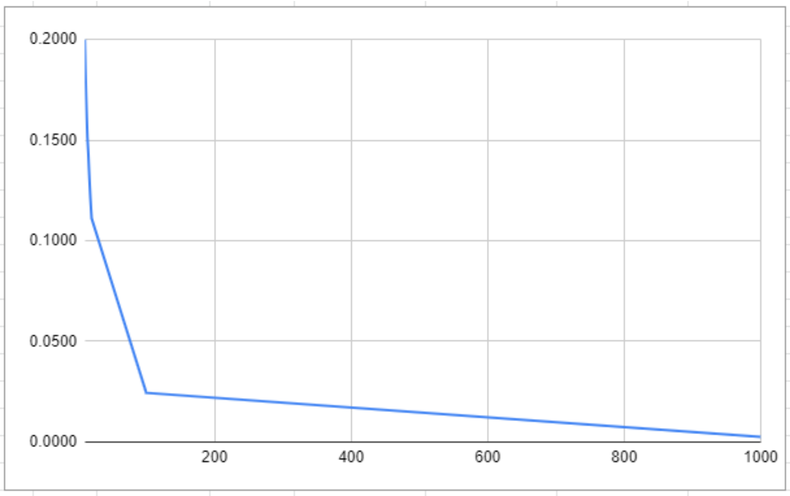
Cores	Speedup	Efficiency
1	1.0000	1.0000
2	1.6667	0.8333
3	2.1429	0.7143
4	2.5000	0.6250
5	2.7778	0.5556
6	3.0000	0.5000
7	3.1818	0.4545
8	3.3333	0.4167
9	3.4615	0.3846
10	3.5714	0.3571
20	4.1667	0.2083
40	4.5455	0.1136
80	4.7619	0.0595
100	4.8077	0.0481
1000	4.9801	0.0050
10000	4.9980	0.0005

The limit for speed up is 5.

2.2. Calculate and plot   with increasing problem sizes



Problem size	Sequential	Parallel	σ
10	2.00	8.00	0.2000
11	2.20	9.68	0.1852
12	2.40	11.52	0.1724
13	2.60	13.52	0.1613
14	2.80	15.68	0.1515
15	3.00	18.00	0.1429
16	3.20	20.48	0.1351
17	3.40	23.12	0.1282
18	3.60	25.92	0.1220
19	3.80	28.88	0.1163
20	4.00	32.00	0.1111
100	20.00	800.00	0.0244
1000	200.00	80,000.00	0.0025
10000	2,000.00	8,000,000.00	0.0002



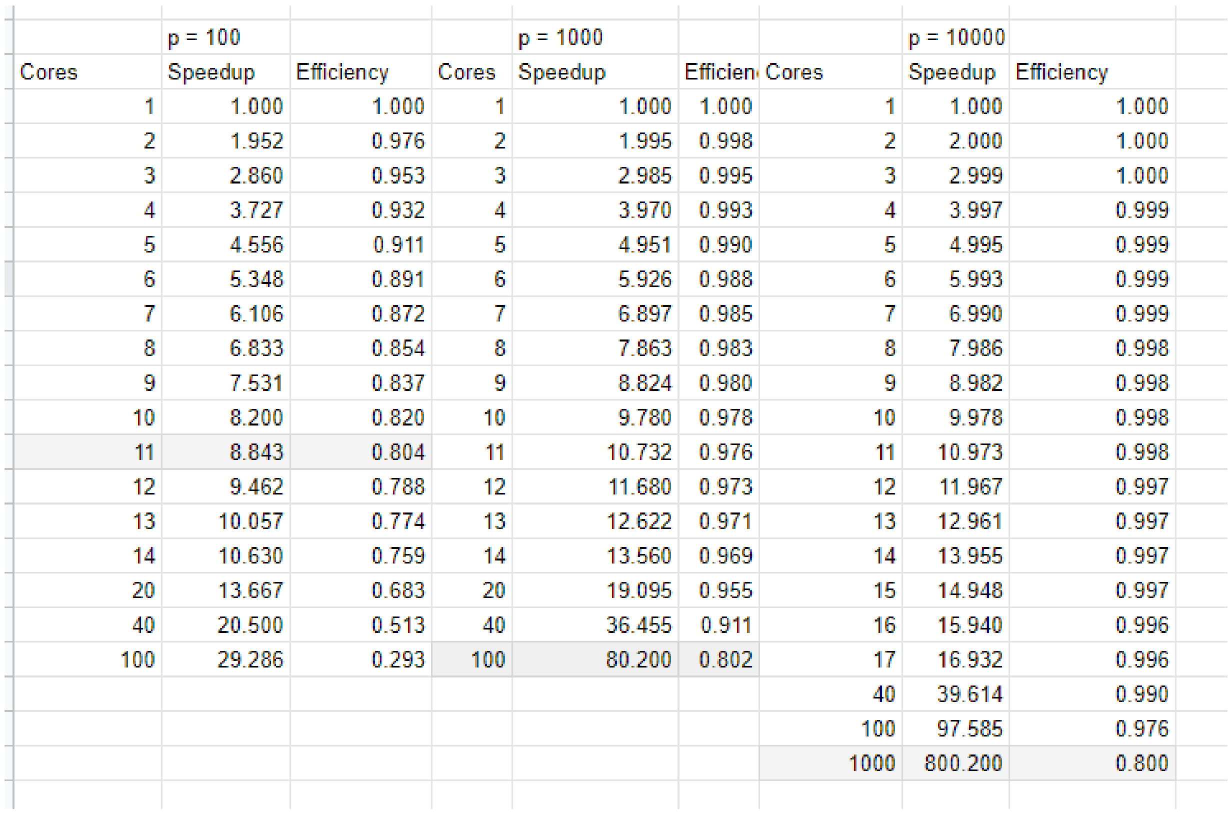
With a problem size of 10000 the sequential part gets extremely small.

2.3. How many processors can be utilized?

p = 100: 11 cores are needed

p = 1000: 100 cores are necessary

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



	p = 100			p = 1000			p = 10000	
Cores	Speedup	Efficiency	Cores	Speedup	Efficien	Cores	Speedup	Efficiency
1	1.000	1.000	1	1.000	1.000	1	1.000	1.000
2	1.952	0.976	2	1.995	0.998	2	2.000	1.000
3	2.860	0.953	3	2.985	0.995	3	2.999	1.000
4	3.727	0.932	4	3.970	0.993	4	3.997	0.999
5	4.556	0.911	5	4.951	0.990	5	4.995	0.999
6	5.348	0.891	6	5.926	0.988	6	5.993	0.999
7	6.106	0.872	7	6.897	0.985	7	6.990	0.999
8	6.833	0.854	8	7.863	0.983	8	7.986	0.998
9	7.531	0.837	9	8.824	0.980	9	8.982	0.998
10	8.200	0.820	10	9.780	0.978	10	9.978	0.998
11	8.843	0.804	11	10.732	0.976	11	10.973	0.998
12	9.462	0.788	12	11.680	0.973	12	11.967	0.997
13	10.057	0.774	13	12.622	0.971	13	12.961	0.997
14	10.630	0.759	14	13.560	0.969	14	13.955	0.997
20	13.667	0.683	20	19.095	0.955	15	14.948	0.997
40	20.500	0.513	40	36.455	0.911	16	15.940	0.996
100	29.286	0.293	100	80.200	0.802	17	16.932	0.996
						40	39.614	0.990
						100	97.585	0.976
						1000	800.200	0.800

3. Wator

3.1. Review of the application

3.1.1. Design

The methods are too long and could often be simplified. Also some methods could be created in separate classes instead they are all in OriginalWatorWorld.

3.1.2. Efficiency

The program in case of efficiency has definitely potential for improvement. For example the use of two dimensional arrays or the creation of an position array each time the GetNeighbors function is called is not ideal.

3.1.3. Clarity

With the GetNeighbors function, the if else constructs are not very ideal to read and could be improved or separate methods could be created.

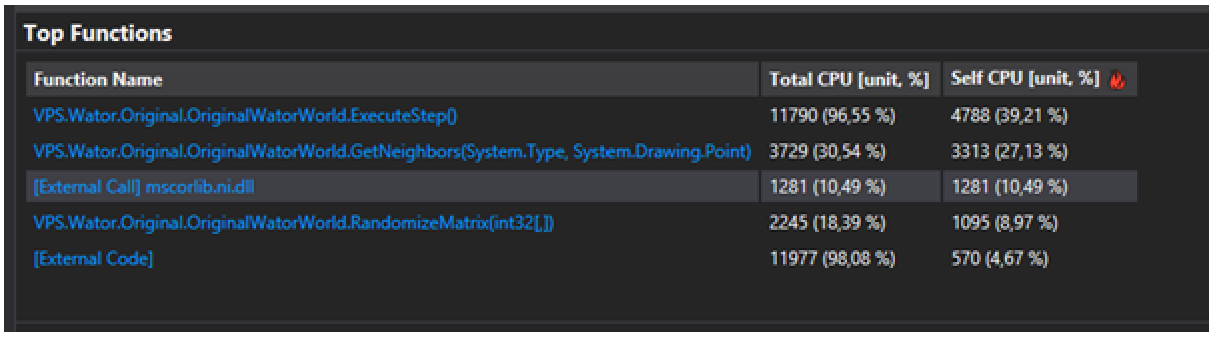
3.1.4. Readability

Readability is definitely not the best. For example the GetNeighbours function is too long with lots of with lots of code duplications.

3.2. Three improvements

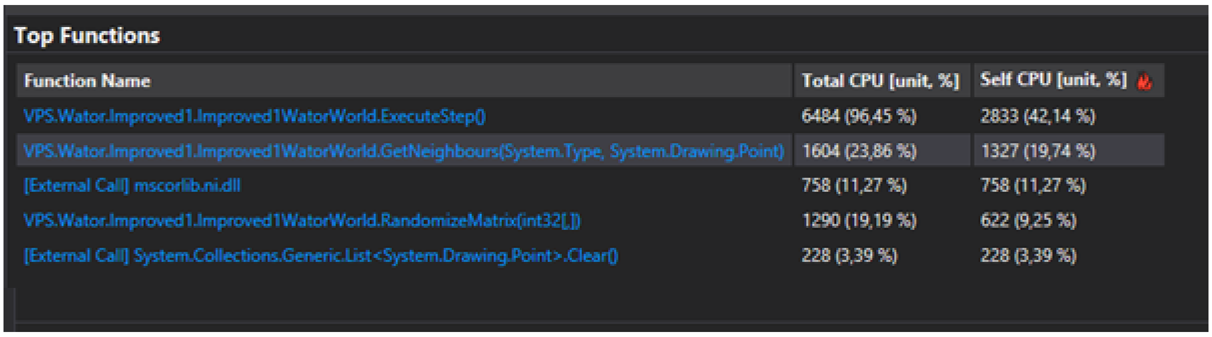
3.2.1. Improvement 1

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.



Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Original.OriginalWatorWorld.ExecuteStep()	11790 (96,55 %)	4788 (39,21 %)
VPS.Wator.Original.OriginalWatorWorld.GetNeighbors(System.Type, System.Drawing.Point)	3729 (30,54 %)	3313 (27,13 %)
[External Call] mscorlib.ni.dll	1281 (10,49 %)	1281 (10,49 %)
VPS.Wator.Original.OriginalWatorWorld.RandomizeMatrix(int32[,])	2245 (18,39 %)	1095 (8,97 %)
[External Code]	11977 (98,08 %)	570 (4,67 %)



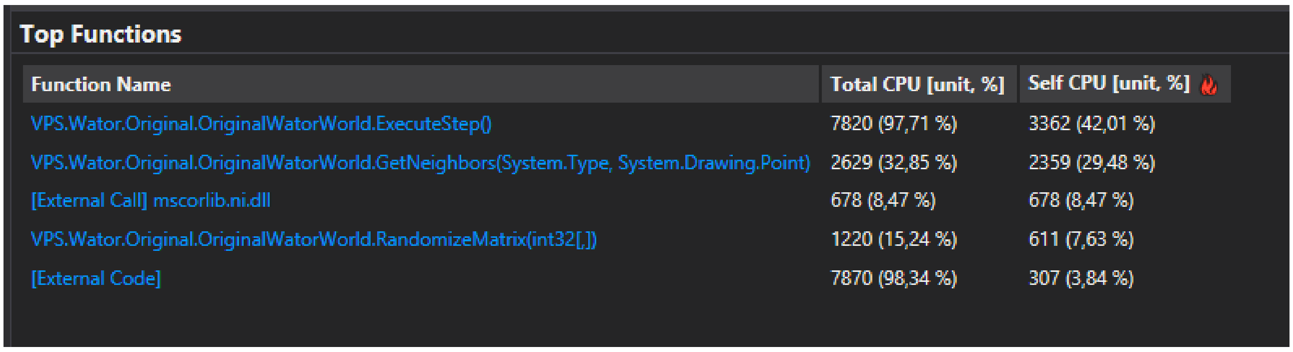
Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Improved1.Improved1WatorWorld.ExecuteStep()	6484 (96,45 %)	2833 (42,14 %)
VPS.Wator.Improved1.Improved1WatorWorld.GetNeighbours(System.Type, System.Drawing.Point)	1604 (23,86 %)	1327 (19,74 %)
[External Call] mscorlib.ni.dll	758 (11,27 %)	758 (11,27 %)
VPS.Wator.Improved1.Improved1WatorWorld.RandomizeMatrix(int32[,])	1290 (19,19 %)	622 (9,25 %)
[External Call] System.Collections.Generic.List<System.Drawing.Point>.Clear()	228 (3,39 %)	228 (3,39 %)

```
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));
        }
    }
    // ...
}
```

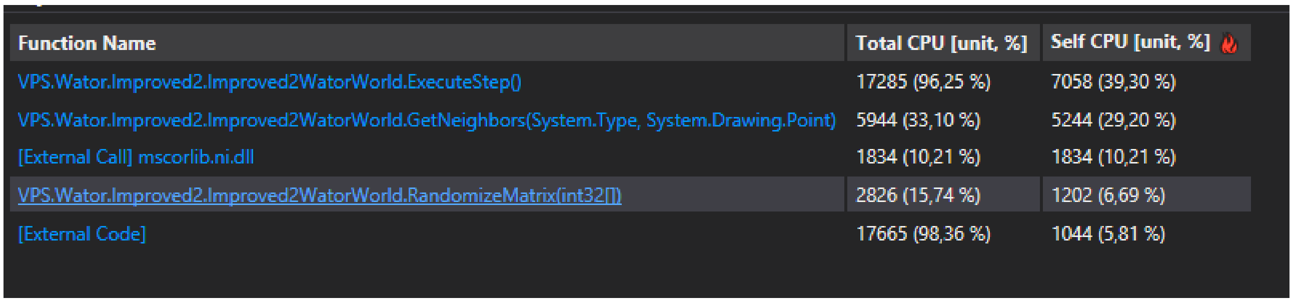
3.2.2. Improvement 2

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.



Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Original.OriginalWatorWorld.ExecuteStep()	7820 (97,71 %)	3362 (42,01 %)
VPS.Wator.Original.OriginalWatorWorld.GetNeighbors(System.Type, System.Drawing.Point)	2629 (32,85 %)	2359 (29,48 %)
[External Call] mscorlib.ni.dll	678 (8,47 %)	678 (8,47 %)
VPS.Wator.Original.OriginalWatorWorld.RandomizeMatrix(int32[,])	1220 (15,24 %)	611 (7,63 %)
[External Code]	7870 (98,34 %)	307 (3,84 %)

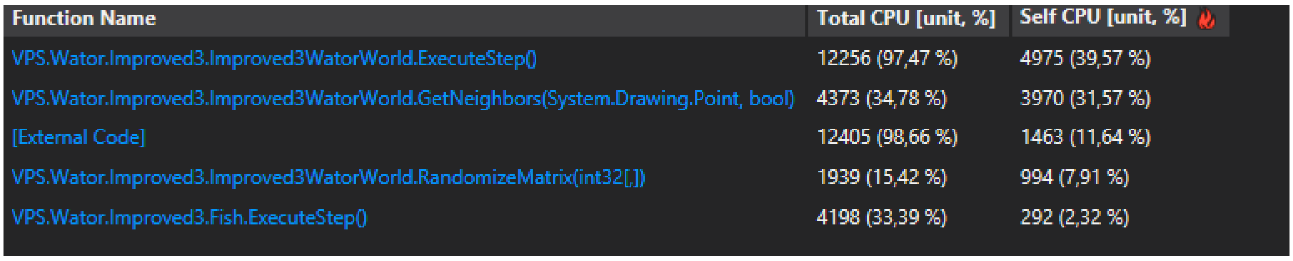


Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Improved2.Improved2WatorWorld.ExecuteStep()	17285 (96,25 %)	7058 (39,30 %)
VPS.Wator.Improved2.Improved2WatorWorld.GetNeighbors(System.Type, System.Drawing.Point)	5944 (33,10 %)	5244 (29,20 %)
[External Call] mscorlib.ni.dll	1834 (10,21 %)	1834 (10,21 %)
VPS.Wator.Improved2.Improved2WatorWorld.RandomizeMatrix(int32[])	2826 (15,74 %)	1202 (6,69 %)
[External Code]	17665 (98,36 %)	1044 (5,81 %)

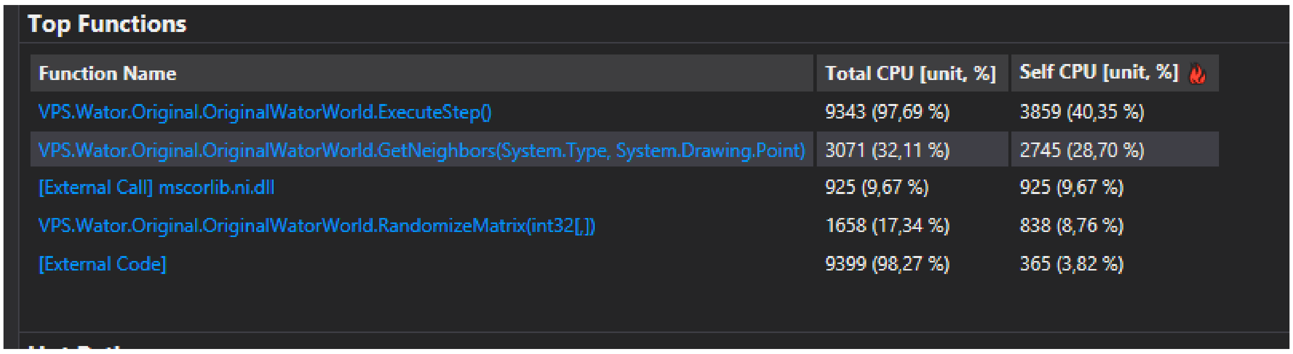
```
private int[] randomMatrix;

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

public Improved2WatorWorld(Settings settings)
{
    // ...
    for (int col = 0; col < Width; col++)
    {
        for (int row = 0; row < Height; row++)
        {
            int value = randomMatrix[GetGridIndex(row, col)];
            if (value < InitialFishPopulation)
            {
                Grid[GetGridIndex(row, col)] = new Fish(this, new Point(col, row), random.Next(
0, FishBreedTime));
            }
            else if (value < InitialFishPopulation + InitialSharkPopulation)
            {
                Grid[GetGridIndex(row, col)] = new Shark(this, new Point(col, row), random.Next
(0, SharkBreedEnergy));
            }
            else
            {
                Grid[GetGridIndex(row, col)] = null;
            }
        }
    }
}
```

3.2.3. Improvement 3

Changed instance of to check if current neighbor is fish by creating an additional property IsFish for the shark class. That means no instance of is necessary. Additionally the Neighbor positions are stored globally with offsets instead of absolute positions and the conditions got simplified.



Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Improved3.Improved3WatorWorld.ExecuteStep()	12256 (97,47 %)	4975 (39,57 %)
VPS.Wator.Improved3.Improved3WatorWorld.GetNeighbors(System.Drawing.Point, bool)	4373 (34,78 %)	3970 (31,57 %)
[External Code]	12405 (98,66 %)	1463 (11,64 %)
VPS.Wator.Improved3.Improved3WatorWorld.RandomizeMatrix(int32[,])	1939 (15,42 %)	994 (7,91 %)
VPS.Wator.Improved3.Fish.ExecuteStep()	4198 (33,39 %)	292 (2,32 %)



Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
VPS.Wator.Original.OriginalWatorWorld.ExecuteStep()	9343 (97,69 %)	3859 (40,35 %)
VPS.Wator.Original.OriginalWatorWorld.GetNeighbors(System.Type, System.Drawing.Point)	3071 (32,11 %)	2745 (28,70 %)
[External Call] mscorlib.ni.dll	925 (9,67 %)	925 (9,67 %)
VPS.Wator.Original.OriginalWatorWorld.RandomizeMatrix(int32[,])	1658 (17,34 %)	838 (8,76 %)
[External Code]	9399 (98,27 %)	365 (3,82 %)

```
// find all neighboring cells of the given position and type
public Point[] GetNeighbors(Point position, bool findNeighborFishes)
{
    Point[] neighbors = new Point[4];
    int neighborIndex = 0;

    foreach (var neighborPos in this.neighborsWithOffset)
    {
        var newPos = new Point((position.X + neighborPos.X) % Width, (position.Y + neighborPos.
Y) % Height);
        var animal = Grid[newPos.X, newPos.Y];
        if ((findNeighborFishes && animal != null && !animal.Moved && animal.IsFish) // Find
neighbor fish which was not moved
            || (!findNeighborFishes && animal == null)) // Find empty neighbor
        {
            // ignore animals moved in the current iteration
            neighbors[neighborIndex] = newPos;
            neighborIndex++;
        }
    }

    // create result array that only contains found cells
    Point[] result = new Point[neighborIndex];
    Array.Copy(neighbors, result, neighborIndex);
    return result;
}
```