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

Deadline: 15.11.2021 15:00

Name Maximilian Mitter

Points ________ **Effort in hours** 5h

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.

Exercise 1

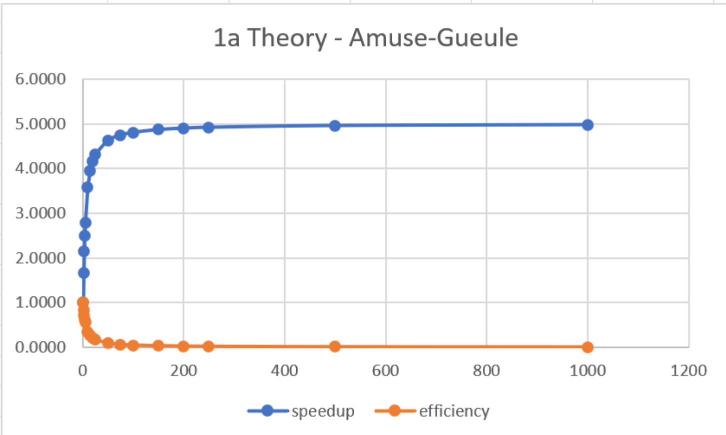
System Configuration

CPU	**Intel Core i7-7700K**
CPU Base Speed	4.20 GHz
CPU Boost Speed	4.50 GHz
Memory Size	16 GB
Memory Speed	3000 MHz
Hard Drive	1 TB M.2 SSD
Operating System	Windows 10 Pro
IDE	Jetbrains Rider V2021.2.2

Theory - Amuse-Gueule

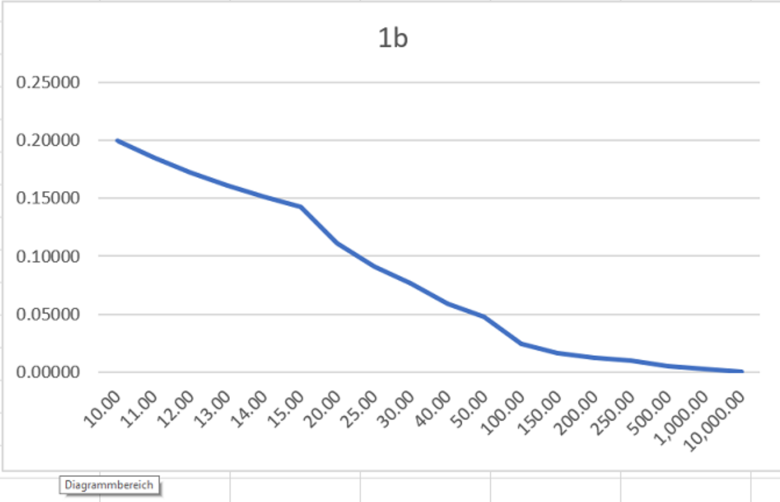
Speedup and Efficiency

p	sigma	n	speedup	efficiency
10	0.2	1	1.0000	1.0000
10	0.2	2	1.6667	0.8333
10	0.2	3	2.1429	0.7143
10	0.2	4	2.5000	0.6250
10	0.2	5	2.7778	0.5556
10	0.2	10	3.5714	0.3571
10	0.2	15	3.9474	0.2632
10	0.2	20	4.1667	0.2083
10	0.2	25	4.3103	0.1724
10	0.2	50	4.6296	0.0926
10	0.2	75	4.7468	0.0633
10	0.2	100	4.8077	0.0481
10	0.2	150	4.8701	0.0325
10	0.2	200	4.9020	0.0245
10	0.2	250	4.9213	0.0197
10	0.2	500	4.9603	0.0099
10	0.2	1000	4.9801	0.0050



Sigma with increasing problem size

p	seq	par	sigma
10.00	2.00	8.00	0.20000
11.00	2.20	9.68	0.18519
12.00	2.40	11.52	0.17241
13.00	2.60	13.52	0.16129
14.00	2.80	15.68	0.15152
15.00	3.00	18.00	0.14286
20.00	4.00	32.00	0.11111
25.00	5.00	50.00	0.09091
30.00	6.00	72.00	0.07692
40.00	8.00	128.00	0.05882
50.00	10.00	200.00	0.04762
100.00	20.00	800.00	0.02439
150.00	30.00	1,800.00	0.01639
200.00	40.00	3,200.00	0.01235
250.00	50.00	5,000.00	0.00990
500.00	100.00	20,000.00	0.00498
1,000.00	200.00	80,000.00	0.00249
10,000.00	2,000.00	8,000,000.00	0.00025



Efficiency with more processors

p	seq	par	sigma	n	speedup	efficiency
10.00	2.00	8.00	0.20000	1	1	1
11.00	2.20	9.68	0.18519	2	1.6875	0.84375
12.00	2.40	11.52	0.17241	2	1.70588235	0.85294118
13.00	2.60	13.52	0.16129	2	1.72222222	0.86111111
14.00	2.80	15.68	0.15152	2	1.73684211	0.86842105
15.00	3.00	18.00	0.14286	2	1.75	0.875
20.00	4.00	32.00	0.11111	3	2.45454545	0.81818182
25.00	5.00	50.00	0.09091	3	2.53846154	0.84615385
30.00	6.00	72.00	0.07692	4	3.25	0.8125
40.00	8.00	128.00	0.05882	5	4.04761905	0.80952381
50.00	10.00	200.00	0.04762	6	4.84615385	0.80769231
100.00	**20.00**	**800.00**	**0.02439**	**11**	**8.84313725**	**0.80392157**
150.00	30.00	1,800.00	0.01639	16	12.8421053	0.80263158
200.00	40.00	3,200.00	0.01235	21	16.8415842	0.8019802
250.00	50.00	5,000.00	0.00990	26	20.8412698	0.8015873
500.00	100.00	20,000.00	0.00498	51	40.8406375	0.80079681
1,000.00	**200.00**	**80,000.00**	**0.00249**	**101**	**80.8403194**	**0.8003992**
10,000.00	**2,000.00**	**8,000,000.00**	**0.00025**	**1001**	**800.840032**	**0.80003999**

Wator World

Review

While WinForms is still supported by Microsoft, it is quite old technology and should not be used anymore. It also makes it harder to port the Application to other operating systems.

The Code suffers from issues with readability and efficiency in quite a few spots. The GetNeighbour-Method is one of the worst offenders for that. The method should extract some functionality (like looking at neighbouring cells) into another function.

General Settings should not be hardcoded, they could be imported from a settings file.

Improvements

Baseline Performance:

Performance Analysis

```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   18233.0054
Avg. Milliseconds / Run:   3646.60108
Std. Deviation:            16.6409777197036
------------------------
Runtimes in Milliseconds:
Run 01:                    3675.1448
Run 02:                    3626.0803
Run 03:                    3641.4022
Run 04:                    3652.862
Run 05:                    3637.5161
```

Optimization 1

Points required for the GetNeighbour-Method are allocated with every call. This has been reworked to only allocate the array once and reuse it for every call. **Speedup: 1.1**.

Performance Analysis

```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   16500.218
Avg. Milliseconds / Run:   3300.0436
Std. Deviation:            17.3198099336057
------------------------
Runtimes in Milliseconds:
Run 01:                    3331.5905
Run 02:                    3302.5733
Run 03:                    3296.7283
Run 04:                    3282.0028
Run 05:                    3287.3231
```

```
private Point[] neighbors = new Point[4];

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

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

```
    } else if ((type != null) && (type.IsInstanceOfType(Grid[j * Width + i]))) {
        if ((Grid[j * Width + i] != null) && (!Grid[j * Width + i].Moved)) {
            neighbors[neighborIndex] = new Point(i, j);
            neighborIndex++;
        }
    }

    // create result array that only contains found cells
    Point[] result = new Point[neighborIndex];
    for (int x = 0; x < neighborIndex; x++)
        result[x] = neighbors[x];
    return result;
}
```

Optimization 2

Operations on a 2-Dimensional Array can be quite performance hungry, so both matrixes are replaced with an array. The index can be calculated using the width of the world. **Speedup: 1.06**.

Performance Analysis

```
Runs:                       5
Iterations:                 100
Runtime in Milliseconds:    15527.309
Avg. Milliseconds / Run:    3105.4618
Std. Deviation:             42.409367845947
------------------------
Runtimes in Milliseconds:
Run 01:                     3174.7234
Run 02:                     3070.7694
Run 03:                     3080.6307
Run 04:                     3066.475
Run 05:                     3134.7105
```

```
private int[] randomMatrix;
public Animal[] Grid { get; private set; }

Grid = new Animal[Width * Height];

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[row * Width + 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[row * Width + col] = matrix[selectedRow * Width + selectedCol];
            matrix[selectedRow * Width + selectedCol] = temp;

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

Optimization 3

The RandomizeMatrix method uses an inefficient version of the Knuth shuffle. This has been replaced with the more effiecent version which brings the asymptotic runtime complexity from O(n^2) to O(n). **Speedup: 1.1**.

Performance Analysis

```
Runs:                       5
Iterations:                 100
Runtime in Milliseconds: 14097.3747
Avg. Milliseconds / Run: 2819.47494
Std. Deviation:             22.4575668697784
------------------------
Runtimes in Milliseconds:
Run 01:                     2856.6924
Run 02:                     2806.3503
Run 03:                     2799.5477
Run 04:                     2834.1036
Run 05:                     2800.6807
```

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

    for (int i = 0; i < Height * Width; i++) {
        int j = random.Next(i, Height * Width);
        temp = matrix[i];
        matrix[i] = matrix[j];
        matrix[j] = temp;
    }
}
```