VPS 5	Parallel and Distributed Software Systems	ST 16, Exercise 1

Deadline: 1st of April 2016 17:00

Name Roman Lumetsberger

Points _____ **Effort in hours** 8

1. Wator – Eat or be eaten … (8 Points)

Wator is the Name of a small circular planet, far far away from our galaxy, were no man 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 hasn't 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)

In the Moodle course you find a ready to use implementation of Wator. Make a critical review of the code and analyze its design, performance, readability, etc. **Write a short report** which outlines the results of your review.

To get a fair comparison of the application's performance, **analyze** a Wator world of 500 x 500 cells. How long takes a run of 100 iterations on average with deactivated graphical output? Execute several independent test runs and **document the results in a table** (also calculate the mean value and the standard deviation). Then **answer the following questions**: Where and what for is most of the runtime consumed? What can be done to improve performance? What are the most performance-critical aspects?

2. Wator – Optimization (16 Points)

Based on your analysis, change the application step by step to improve performance. Think of at least **three concrete improvements** and implement them. Document for each improvement 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. Test your improvements with the settings given in the previous task.

1 1. Wator – Eat or be eaten

1.1 Analyse

Der Sourcecode ist grundsätzlich gut dokumentiert und lesbar. Die Architektur der Anwendung ist loose gekoppelt und nachvollziehbar.

Zur Implementierung gibt es zu erwähnen, dass in der selben Methode oft auf die gleichen Feldelemente zugegriffen wird. Hier wäre es vielleicht besser, wenn der Wert einmal zwischengespeichert werden würde, da sonst bei jedem Zugriff Laufzeitüberprüfungen durchgeführt werden.

Weiters fällt auf, dass sehr viele Objekte der Klasse `Point` angelegt werden.

Bei der Methode `GetNeighbors` gibt es zu erwähnen, dass hier für alle vier Richtungen eigentlich der selbe Code verwendet wird und dieser vielleicht besser in eine Methode ausgelagert werden könnte (bringt warscheinlich keine Performancesteigerung aber der Code wäre besser lesbar).

1.2 Performance

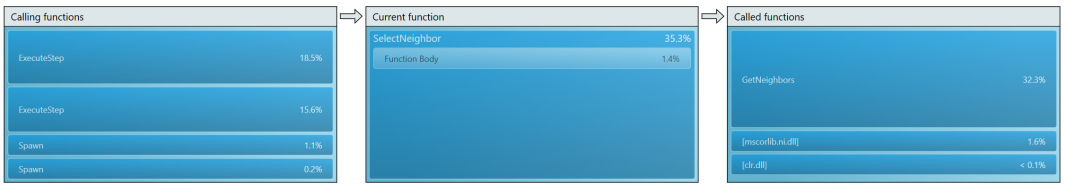
Zur Bestimmung der Performance wurden folgende Parameter verwendet:

- 5 Durchläufe
- Größe der Welt: 500x500
- 100 Iterationen
- Grafikausgabe deaktiviert

Performance der originalen Anwendung

```
Runs:                    5
Iterations:              100
Runtime in Milliseconds: 30907,1241
Avg. Milliseconds / Run: 6181,42482
Std. Deviation:          220,870269363494
------------------------
Runtimes in Milliseconds:
Run 01:                  6275,0363
Run 02:                  6573,7986
Run 03:                  6059,9067
Run 04:                  5996,6139
Run 05:                  6001,7686
```

1.3 Wo wird die meiste Rechneleistung verbraucht?



Grundsätzlich wird die meiste CPU-Zeit in der Methode `ExecuteStep` und in weiterer Folge dann in `GetNeighbors` verbraucht. Diese Methode ist für das Suchen der Nachbarn eines Feldes zuständig und wird daher sehr oft aufgerufen. Die Analsyse der Methode zeigt, dass sehr viele `Point` Objekte angelegt und dann sogar nochmals kopiert werden.

Die Methode `RandomizeMatrix` ist eine weitere Methode die sehr viel CPU-Zeit benötigt. Diese ist für das Mischen der Durchlauf-Matrix zuständig.

1.4 Wie kann die Performance verbessert werden?

- Umgang mit den Koordinaten in der Methode `GetNeighbors` optimieren, damit nicht mehr so viele Objekte angelegt werden müssen.

  Verwenden eines statischen Arrays.

  Kopieren des Arrays am Ende der Methode vermeiden.

- Umstellen der zweidimensionalen Felder auf eindimensionales Felder.
- Ändern `Moved` Eigenschaft um die zusätzliche Schleife über alle Elemente zu sparen.

  Eine Möglichkeit wäre hier die Iterationsnummer zu verwenden.

2 Wator – Optimierung

2.1 Version 1

Für diese Optimierung wurde versucht die Anzahl der benötigten `Point` Objekte zu verringern, indem die Methode `GetNeighbors` geändert wurde. Die lokale Variable `neighbors` wurde als private Datenkomponente angelgt und wird so für jeden Aufruf wiederverwendet. Damit braucht sie nicht jedesmal neu angelegt werden.

Weiters wurde die Logik der zufälligen Auswahl einer Richtung direkt in die Methode `GetNeighbors` verlagert, damit braucht am Ende das Array nicht kopiert werden, sondern es wird direkt der Punkt zurückgegeben.

Zur besseren Lesbarkeit des Quellcodes wurde die Prüfung ob ein Nachbarelement gültig ist, in eine eigene Methode `CheckNeighbor` ausgelagert.

2.1.1 Performance

```
Runs:                    5
Iterations:              100
Runtime in Milliseconds: 24739,7561
Avg. Milliseconds / Run: 4947,95122
Std. Deviation:          31,7208145277168
------------------------
Runtimes in Milliseconds:
Run 01:                  4960,0705
Run 02:                  4968,9271
Run 03:                  4884,7747
Run 04:                  4963,4762
Run 05:                  4962,5076
```

Speedup zur vorherigen Version: 1,24726873
Speedup zur original Version: 1,24726873

2.1.2 Codeänderungen

```
 /// <summary>
/// Neighbors static data array which is reused
/// </summary>
private readonly Point[] neighbors = new Point[4];

// find all neighbouring cells of the given position that contain an animal of the given type
public Point SelectNeighbor(Type type, Point position)
{
    int neighborIndex;
    int i, j;

    // counter for the number of cells of the correct type
    neighborIndex = 0;
    // look up
    i = position.X;
    j = (position.Y + Height - 1) % Height;
    if (CheckNeighbor(type, i,j ))
    {
        neighbors[neighborIndex].X = i;
        neighbors[neighborIndex].Y = j;
        neighborIndex++;
    }

    i = (position.X + 1) % Width;
    j = position.Y;
    if (CheckNeighbor(type,i, j))
    {
        neighbors[neighborIndex].X = i;
        neighbors[neighborIndex].Y = j;
        neighborIndex++;
    }

    // look down
    i = position.X;
    j = (position.Y + 1) % Height;
    if (CheckNeighbor(type, i, j))
    {
        neighbors[neighborIndex].X = i;
        neighbors[neighborIndex].Y = j;
        neighborIndex++;
    }

    // look left
    i = (position.X + Width - 1) % Width;
    j = position.Y;
    if (CheckNeighbor(type, i, j))
    {
        neighbors[neighborIndex].X = i;
        neighbors[neighborIndex].Y = j;
        neighborIndex++;
    }

    if (neighborIndex > 1)
    {
        // if more than one cell has been found => return a randomly selected cell
        return neighbors[random.Next(neighborIndex)];
    }
    else if (neighborIndex == 1)
    {
```

```
        // if only a single cell contains an animal of the given type we can save the call to random
        return neighbors[0];
    }
    else
    {
        // return a point with negative coordinates to indicate
        // that no neighbouring cell has found
        // return value must be checked by the caller
        return new Point(-1, -1);
    }
}
/// <summary>
/// Checks if a neighbor is from the given type
/// </summary>
private bool CheckNeighbor(Type type, int xCoord, int yCoord)
{
    var value = Grid[xCoord, yCoord];
    if ((type == null) && (value  == null))
    {
        return true;
    }
    else if ((type != null) && (type.IsInstanceOfType(value)))
    {
        if ((value != null) && (!value.Moved))
        {
            return true;
        }
    }
    return false;
}
```

2.2 Version 2

In dieser Version wurde das zweidimensionale Feld `randomMatrix` in ein eindimensionales Feld geändert. Damit ist können die Methoden `GenerateRandomMatrix` und `RandomizeMatrix` wesentlich einfacher implmentiert werden. Weiters benötigt der die Methode `RandomizeMatrix` nur mehr einen Aufruf von `random.Next`.

2.3 Performance

```
Runs:                    5
Iterations:              100
Runtime in Milliseconds: 20419,0235
Avg. Milliseconds / Run: 4083,8047
Std. Deviation:          41,3133004982238
------------------------
Runtimes in Milliseconds:
Run 01:                  4047,6108
Run 02:                  4114,0845
Run 03:                  4033,2422
Run 04:                  4144,8737
Run 05:                  4079,2123
```

Speedup zur vorherigen Version: 1,21160329
Speedup zur original Version: 1,513643593

2.3.1 Codeänderungen

```
private int[] randomMatrix;


// create a 2D array containing all numbers in the range 0 .. width * height
// the numbers are shuffled to create a random ordering
private int[] GenerateRandomMatrix(int width, int height)
{
    int[] matrix = new int[width * height];

    for (int i = 0; i < matrix.Length; i++)
    {
       matrix[i]=i;
    }
    // shuffle matrix
    RandomizeMatrix(matrix);
    return matrix;
}

// shuffle the values of the 2D array in a random fashion
private void RandomizeMatrix(int[] matrix)
{
    //Knuth shuffle for arrays
    //https://www.rosettacode.org/wiki/Knuth_shuffle#C.23
    for (int i = 0; i < matrix.Length; i++)
    {
        int j = random.Next(i, matrix.Length); // Don't select from the entire array on subsequent loops
        int temp = matrix[i];
        matrix[i] = matrix[j];
        matrix[j] = temp;
    }
```

```
}

public void ExecuteStep()
{
...
// go over all cells of the random matrix
    int row, col;
    for (int i = 0; i < Width*Height; i++)
    {
        // determine row and col of the grid cell by manipulating the value
        var value = randomMatrix[i];
        col = value % Width;
        row = value / Width;

        // if there is an animal on this cell that has not been moved in this simulation step
        // then we execute it
        if (Grid[col, row] != null && !Grid[col, row].Moved)
            Grid[col, row].ExecuteStep();

    }
...
```

2.4 Version 3

In der dritten Optimierung wurde die Notwendigkeit des zusätzlichen Schleifendurchlaufs um das `Commit` auszuführen. Dazu wurde eine neue Datenkomponente `CurrentIteration` eingeführt und beim Property `Moved` wird dann der Iterationszähler verglichen.

2.5 Performance

```
Runs:                       5
Iterations:                 100
Runtime in Milliseconds:    18814,3484
Avg. Milliseconds / Run:    3762,86968
Std. Deviation:             28,985930215203
------------------------
Runtimes in Milliseconds:
Run 01:                     3742,9105
Run 02:                     3813,2592
Run 03:                     3747,6104
Run 04:                     3733,8888
Run 05:                     3776,6795
```

Speedup zur vorherigen Version: 1,0852900
Speedup zur original Version: 1,6427422

2.5.1 Codeänderungen

```
public abstract class Animal
{
...
  //iteration in which the animal was moved
  private long movedIteration;
```

```
  // ctor: create a new animal on the specified position of the given world
  public Animal(OriginalWatorWorld world, Point position)
  {
...
      movedIteration = World.CurrentIteration;
...
  }

  // move the animal to a given position
  // does not check if the position can be reached by the animal
  protected void Move(Point destination)
  {
    World.Grid[Position.X, Position.Y] = null;
    World.Grid[destination.X, destination.Y] = this;
    Position = destination;
    movedIteration = World.CurrentIteration;
  }


  // boolean flag that indicates wether an animal has moved in the current iteration
  public bool Moved
  {
      get { return movedIteration == World.CurrentIteration; }
  }
```

```
public class OriginalWatorWorld : IWatorWorld
{
  ...
    /// <summary>
    /// Gets the current iteration of the world
    /// </summary>
    public long CurrentIteration { get; private set; }

  public void ExecuteStep()
  {
      CurrentIteration++;
      ...

  }
  ...
}
```

3 Zusammenfassung

Parameter

- 5 Durchläufe
- Größe der Welt: 500x500
- 100 Iterationen
- Grafikausgabe deaktiviert

Lauf	Original	Version 1	Version 2	Version 3
1 Lauf	6275,0363	4960,0705	4047,6108	3742,9105
2 Lauf	6573,7986	4968,9271	4114,0845	3813,2592
3 Lauf	6059,9067	4884,7747	4033,2422	3747,6104
4 Lauf	5996,6139	4963,4762	4144,8737	3733,8888
5 Lauf	6001,7686	4962,5076	4079,2123	3776,6795
Durchschnitt	6181,4248	4947,9512	4083,8047	3762,8697
Standardabweichung	220,8703	31,7208	41,3133	28,9859
Speedup zur vorherigen Version		1,2473	1,2116	1,0853
Speedup zur originalen Version		1,2473	1,5136	**1,6427**