Politechnika Wrocławska

Platformy programistyczne .NET, Java

Zadanie 3, Wielowątkowość

Hanna Grzebieluch, 264209
Termin zajęć: Środa, $17^{05} - 18^{45}$
GitHub:

`https://github.com/perigrins/net_threads`

Wydział Informatyki i Telekomunikacji
Informatyczne Systemy Automatyki

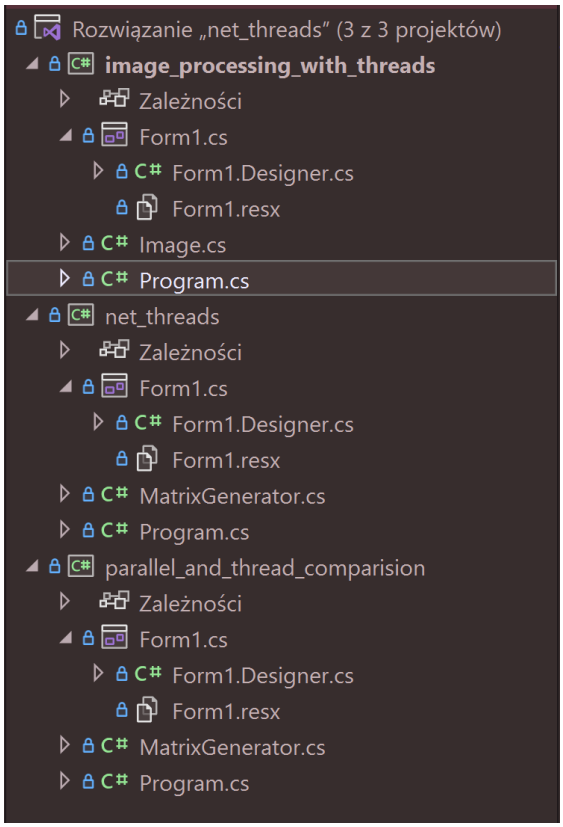
17 kwietnia 2024

1 Cel zadania

Celem zadania trzeciego było zapoznanie się z wielowątkowością w języku C#. Składały się na to trzy podzadania - przyspieszenie działania kodu poprzez zrównoleglenie obliczeń z użyciem funkcji zarówno niskiego jak i wysokiego poziomu oraz zastosowanie wielowątkowości do przetwarzania obrazów.

2 Wykonanie

2.1 Drzewo projektu



Rysunek 1: Drzewo projektu

2.2 Zadanie 1

Zadanie pierwsze polegało na wielowątkowym mnożeniu macierzy z wykorzystaniem klasy Thread. Rozwiązania zadania pierwszego znajdują się w namespace **net_threads**.

Pierwszym krokiem było zaimplementowanie klasy do generowania losowej macierzy (w programie jest to macierz kwadratowa) **MatrixGenerator**.

```
Odwołania: 2
internal class MatrixGenerator
{
    Odwołania: 2
    public int[,] Generate(int r, int c)
    {
        Random random = new Random();
        int[,] matrix = new int[r, c];
        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < c; j++)
            {
                matrix[i, j] = random.Next(0, 10);
            }
        }
        return matrix;
    }
}
```

Rysunek 2: Klasa MatrixGenerator

Następnie w klasie **Form1** tworzone są 2 zmienne odpowiedzialne na liczbę kolumn i rzędów macierzy. Zaimplementowana została funkcja button_start_Click, w której następuje utworzenie nowego obiektu klasy MatrixGenerator oraz deklaracja zmiennych użytych do obliczeń.

```
MatrixGenerator matrix = new MatrixGenerator();
int[,] matrix1 = new int[r, c];
int[,] matrix2 = new int[r, c];
int[,] matrixR = new int[r, c];
int[,] matrixS = new int[r, c];     // for sequential time measurement

matrix1 = matrix.Generate(r, c);
matrix2 = matrix.Generate(r, c);
```

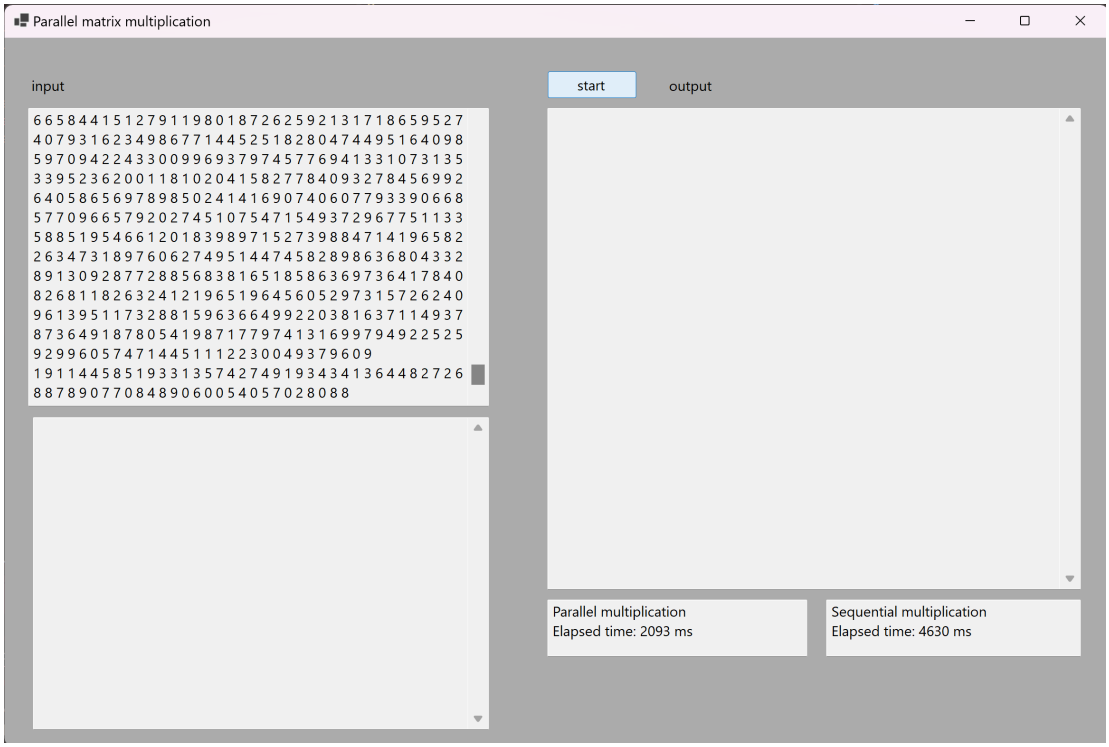
Rysunek 3: Deklaracja zmiennych

Kolejnym krokiem było napisanie funkcji Mult, której zadaniem było odpowiednie tworzenie i łączenie wątków. Funkcja jako argument przyjmowała liczbę wątków.

```
int[,] Mult(int thread_nr)
{
    double x = (double)r / thread_nr;
    double elementsPerThread_d = Math.Ceiling(x);
    int elementsPerThread = (int)elementsPerThread_d;

    Thread[] threads = new Thread[thread_nr];

    for (int t = 0; t < thread_nr; t++)
    {
        int start = t * elementsPerThread;
        int end;
        if (t == thread_nr - 1)
        {
            end = r;
        }
        else
        {
            end = start + elementsPerThread - 1;
        }

        threads[t] = new Thread(() =>
        {
            for (int i = start; i < end; i++)
            {
                for (int j = 0; j < c; j++)
                {
                    matrixR[i, j] = 0;
                    for (int k = 0; k < r; k++)
                    {
                        matrixR[i, j] += matrix1[i, k] * matrix2[k, j];
                    }
                }
            }
        });
        threads[t].Start();
    }
    foreach (Thread thread in threads)
    {
        thread.Join();
    }
    return matrixR;
}
```

Rysunek 4: Funkcja służąca do mnożenia macierzy

Ostatnim krokiem było wywołanie funkcji oraz zmierzenie czasu jej działania oraz późniejsze porównanie z obliczaniem sekwencyjnym.



Rysunek 5: Aplikacja okienkowa do zadania 1

Poprawność działań została przetestowana na macierzach mniejszych rozmiarów.

2.3 Zadanie 2

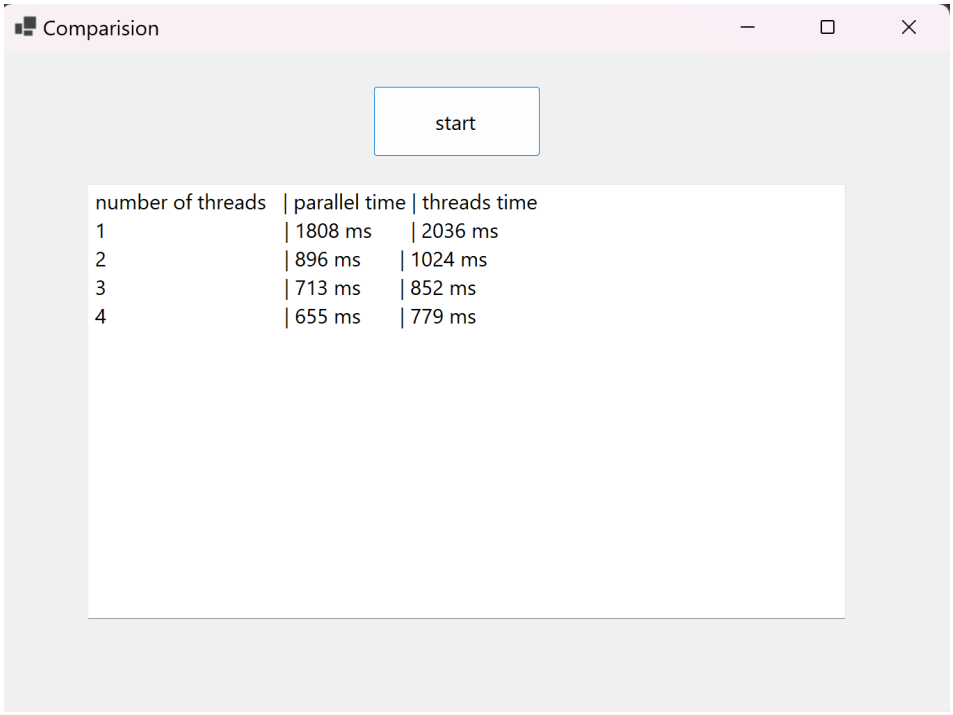
Zadanie drugie polegało na zrównolegleniu obliczeń przy pomocy biblioteki Parallel oraz na analizie przyspieszenia względem klasy Threads. Do generowania macierzy użyta została identyczna klasa **MatrixGenerator**.

W celu poprawnego zmierzenia czasów działania wszystkie obliczenia zostały powtórzone 5 razy, a wynik został uśredniony. Obliczenia przeprowadzono dla liczby wątków $< 1, 4 >$.

```
void Mult_Parallel(int x)
{
    ParallelOptions opt = new ParallelOptions()
    {
        MaxDegreeOfParallelism = x
    };

    Parallel.For(0, rows, opt, i =>
    {
        for (int j = 0; j < cols; j++)
        {
            matrixP[i, j] = 0;
            for (int k = 0; k < rows; k++)
            {
                matrixP[i, j] += matrix1[i, k] * matrix2[k, j];
            }
        }
    });
}
```

Rysunek 6: Zastosowanie biblioteki Parallel

W powyższym przykładzie x jest liczbą wątków.



Rysunek 7: Wyniki dla macierzy 600 x 600

Poniżej znajduje się porównanie czasów dla różnych wielkości macierzy i różnych ilości wątków.

rozmiar macierzy	czas (parallel) [ms]	czas (threads) [ms]	czas (parallel) [ms]	czas (threads) [ms]	czas (parallel) [ms]	czas (threads) [ms]
	przy ilości wątków n=4		przy ilości wątków n=2		przy ilości wątków n=1	
100 X 100	4	54	4	14	14	21
200 X 200	19	84	34	55	64	78
300 X 300	77	147	141	159	305	302
400 X 400	202	237	242	308	402	476
500 X 500	403	471	499	586	1218	1524
600 X 600	680	844	1220	1622	1688	1824

Rysunek 8: Porównanie czasów

Z zestawienia można wysnuć wniosek, że im więcej wątków, tym program działa efektywniej. Dotyczy to jednak tylko sytuacji, gdy pod uwagę bierzemy również liczbę rdzeni procesora - przy bardzo dużych ilościach wątków (np. 16) wyniki będą odwrotne od zakładanych.

2.4 Zadanie 3

Zadanie trzecie polegało na napisaniu 4 różnych filtrów przetwarzających obrazy. Filtry napisane zostały sekwencyjnie, a zrównolegleniu poddana została ich liczba.

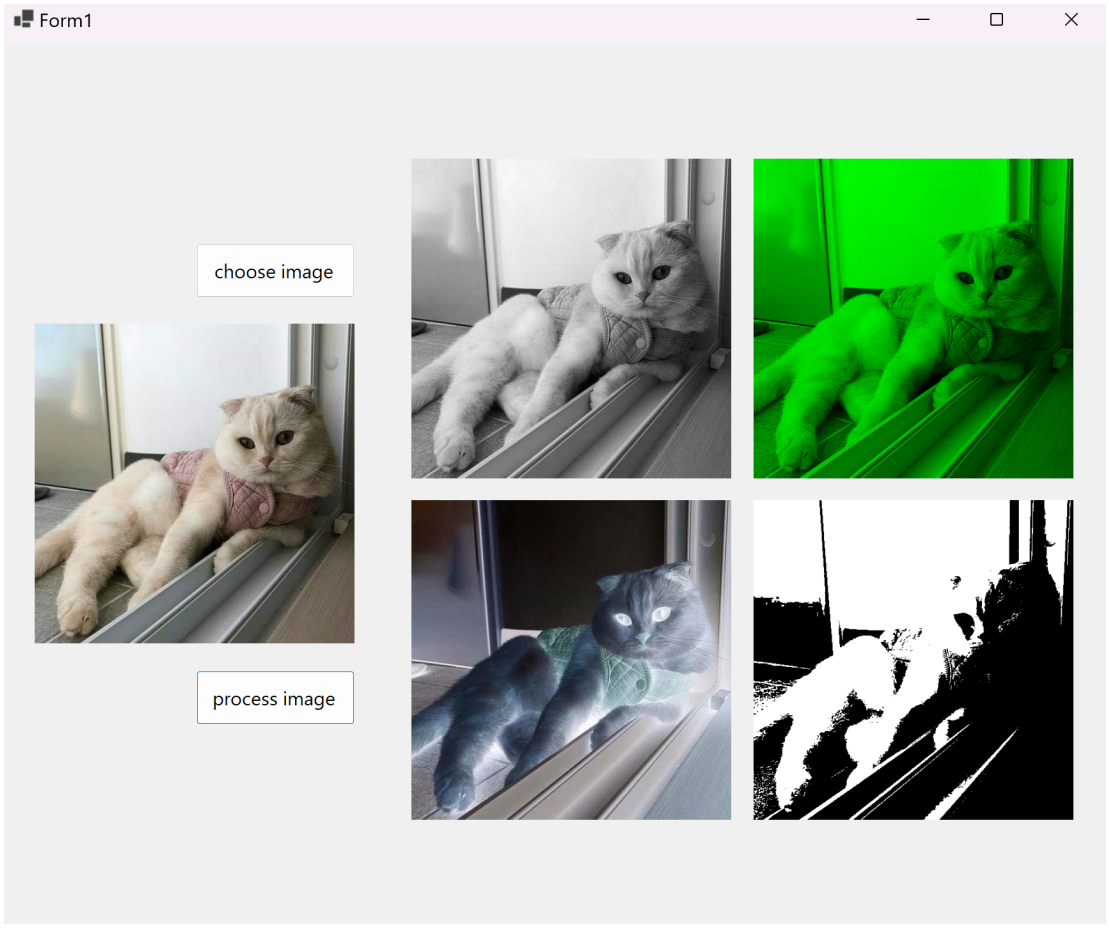
Klasa **Image** zawiera 4 funkcje odpowiedzialne za progowanie, grayscale, greenscale oraz negatyw.
Użytkownik ma możliwość wybrania dowolnego zdjęcia do przetworzenia, lecz musi być ono w formacie .jpg.

```
1 odwołanie
private void button_choose_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Title = "Open Image";
    dlg.Filter = "jpg files (*.jpg)|*.jpg";
    dlg.ShowDialog();

    /*if (dlg.ShowDialog() == DialogResult.OK)
    {
        img = new Bitmap(dlg.FileName);
        pictureBox1.Image = img;
    }*/

    img = new Bitmap(dlg.FileName);
    pictureBox1.Image = img;
    dlg.Dispose();
}
```

Rysunek 9: Wybór zdjęcia do przetworzenia

```
1 odwołanie
private void button_process_Click(object sender, EventArgs e)
{
    Image image = new Image();

    Bitmap b1 = new Bitmap(img);
    Bitmap b2 = new Bitmap(img);
    Bitmap b3 = new Bitmap(img);
    Bitmap b4 = new Bitmap(img);

    int thread_nr = 4;
    Thread[] threads = new Thread[thread_nr];
    threads[0] = new Thread(() => image.Grayscale(b1));
    threads[1] = new Thread(() => image.Negative(b2));
    threads[2] = new Thread(() => image.Green(b3));
    threads[3] = new Thread(() => image.Threshold(b4));

    foreach (Thread x in threads)
    {
        x.Start();
    }

    foreach (Thread x in threads)
    {
        x.Join();
    }

    pictureBox2.Image = b1;
    pictureBox3.Image = b2;
    pictureBox4.Image = b3;
    pictureBox5.Image = b4;

    img.Dispose();
}
```

Rysunek 10: Funkcja odpowiedzialna za wielowątkowość



Rysunek 11: Aplikacja okienkowa dla zadania 3