



SPRAWOZDANIE PLATFORMY PROGRAMISTYCZNE .NET I JAVA – LAB3



POLITECHNIKA WROCŁAWSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

INFORMATYCZNE SYSTEMY AUTOMATYKI

MICHAŁ WYRZYKOWSKI

INDEKS 264228

1. Kod C# oraz aplikacja okienkowa

```
1 odwołanie
private int[,] MultiplyMatricesParallelLibrary(int[,] matrixA, int[,] matrixB, int threadCount)
{
    int rows = matrixA.GetLength(0);
    int columns = matrixB.GetLength(1);
    int[,] resultMatrix = new int[rows, columns];

    int elementsPerThread = rows * columns / threadCount;

    Parallel.For(0, threadCount, (threadIndex) =>
    {
        int start = threadIndex * elementsPerThread;
        int end = (threadIndex == threadCount - 1) ? rows * columns : (threadIndex + 1) * elementsPerThread;

        for (int index = start; index < end; index++)
        {
            int row = index / columns;
            int col = index % columns;

            for (int k = 0; k < matrixA.GetLength(1); k++)
            {
                resultMatrix[row, col] += matrixA[row, k] * matrixB[k, col];
            }
        }
    });

    return resultMatrix;
}
```

```
1 odwołanie
private void ApplyThreshold(Bitmap inputImage, Bitmap processedImage, int threshold)
{
    for (int y = 0; y < inputImage.Height; y++)
    {
        for (int x = 0; x < inputImage.Width; x++)
        {
            Color pixelColor = inputImage.GetPixel(x, y);
            int grayValue = (int)(0.299 * pixelColor.R + 0.587 * pixelColor.G + 0.114 * pixelColor.B);
            Color newColor = grayValue > threshold ? Color.White : Color.Black;
            processedImage.SetPixel(x, y, newColor);
        }
    }
}
```

```
1 odwołanie
private void ApplyNegative(Bitmap inputImage, Bitmap processedImage)
{
    for (int y = 0; y < inputImage.Height; y++)
    {
        for (int x = 0; x < inputImage.Width; x++)
        {
            Color pixelColor = inputImage.GetPixel(x, y);
            Color newColor = Color.FromArgb(pixelColor.A, 255 - pixelColor.R, 255 - pixelColor.G, 255 - pixelColor.B);
            processedImage.SetPixel(x, y, newColor);
        }
    }
}
```

1 odwołanie

```
private int[,] MultiplyMatricesParallelThreads(int[,] matrixA, int[,] matrixB, int threadCount)
{
    int rows = matrixA.GetLength(0);
    int columns = matrixB.GetLength(1);
    int[,] resultMatrix = new int[rows, columns];
    int elementsPerThread = rows * columns / threadCount;

    Thread[] threads = new Thread[threadCount];

    for (int i = 0; i < threadCount; i++)
    {
        int start = i * elementsPerThread;
        int end = (i == threadCount - 1) ? rows * columns : (i + 1) * elementsPerThread;

        threads[i] = new Thread(() =>
        {
            for (int index = start; index < end; index++)
            {
                int row = index / columns;
                int col = index % columns;

                for (int k = 0; k < matrixA.GetLength(1); k++)
                {
                    resultMatrix[row, col] += matrixA[row, k] * matrixB[k, col];
                }
            }
        });
        threads[i].Start();
    }

    foreach (var thread in threads)
    {
        thread.Join();
    }

    return resultMatrix;
}
```

1 odwołanie

```
private void ApplyFiltersParallel()
{
    if (originalImage == null)
    {
        MessageBox.Show("Wczytaj obraz przed zastosowaniem filtrów!", "Błąd", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    Bitmap processedImageGray = new Bitmap(originalImage.Width, originalImage.Height);
    Bitmap processedImageMirror = new Bitmap(originalImage.Width, originalImage.Height);
    Bitmap processedImageThreshold = new Bitmap(originalImage.Width, originalImage.Height);
    Bitmap processedImageNegative = new Bitmap(originalImage.Width, originalImage.Height);

    int threshold = 128;

    /* ThreadPool.QueueUserWorkItem((state) => ApplyGrayscale(originalImage, processedImageGray));
    ThreadPool.QueueUserWorkItem((state) => ApplyMirror(originalImage, processedImageMirror));
    ThreadPool.QueueUserWorkItem((state) => ApplyThreshold(originalImage, processedImageThreshold, threshold));
    ThreadPool.QueueUserWorkItem((state) => ApplyNegative(originalImage, processedImageNegative));

    while (ThreadPool.PendingWorkItemCount > 0) { } */

    ApplyGrayscale(originalImage, processedImageGray);
    ApplyMirror(originalImage, processedImageMirror);
    ApplyThreshold(originalImage, processedImageThreshold, threshold);
    ApplyNegative(originalImage, processedImageNegative);

    pictureBox2.Image = processedImageGray;
    pictureBox3.Image = processedImageMirror;
    pictureBox4.Image = processedImageThreshold;
    pictureBox5.Image = processedImageNegative;
}
```

```

1 odwołanie
private void button2_Click(object sender, EventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Image Files (*.jpg; *.jpeg; *.png; *.bmp)|*.jpg; *.jpeg; *.png; *.bmp|All files (*.*)|*.*";
    openFileDialog.FilterIndex = 1;

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        originalImage = new Bitmap(openFileDialog.FileName);
        pictureBox1.Image = originalImage;
    }
}

```

```

Odwołania: 4
public class MatrixGenerator
{
    private Random random;

    Odwołania: 2
    public MatrixGenerator(int seed)
    {
        random = new Random(seed);
    }

    Odwołania: 2
    public int[,] GenerateRandomMatrix(int rows, int columns)
    {
        int[,] matrix = new int[rows, columns];
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                matrix[i, j] = random.Next(100);
            }
        }
        return matrix;
    }

    Odwołania: 0
    public void DisplayMatrix(int[,] matrix, string title)
    {
        Console.WriteLine(title);
        int rows = matrix.GetLength(0);
        int columns = matrix.GetLength(1);
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                Console.Write(matrix[i, j] + "\t");
            }
            Console.WriteLine();
        }
        Console.WriteLine();
    }
}

```

```

1 odwołanie
private void ApplyGrayscale(Bitmap inputImage, Bitmap processedImage)
{
    for (int y = 0; y < inputImage.Height; y++)
    {
        for (int x = 0; x < inputImage.Width; x++)
        {
            Color pixelColor = inputImage.GetPixel(x, y);
            int grayValue = (int)(0.299 * pixelColor.R + 0.587 * pixelColor.G + 0.114 * pixelColor.B);
            Color newColor = Color.FromArgb(pixelColor.A, grayValue, grayValue, grayValue);
            processedImage.SetPixel(x, y, newColor);
        }
    }
}

1 odwołanie
private void ApplyMirror(Bitmap inputImage, Bitmap processedImage)
{
    for (int y = 0; y < inputImage.Height; y++)
    {
        for (int x = 0; x < inputImage.Width; x++)
        {
            Color pixelColor = inputImage.GetPixel(inputImage.Width - x - 1, y);
            processedImage.SetPixel(x, y, pixelColor);
        }
    }
}

```

Form1

Czas obliczeń

Rozmiar macierzy	Threads	Parallel
500	367 ms	284 ms
Ilość wątków	Seed macierzy 1	Seed macierzy 2
4	123	123

Pomnóż

Wczytaj

Przetwórz

Macierz wejściowa 1

Macierz wejściowa 2

Macierz wynikowa

2. Opis działania

- **Form1 class:** Jest to klasa okna głównego aplikacji, dziedzicząca po klasie `Form` z biblioteki `Windows Forms`.
- **button1_Click:** Obsługuje kliknięcie przycisku "Oblicz" w interfejsie użytkownika. W tej metodzie następuje wczytanie danych z pól tekstowych, generowanie dwóch losowych macierzy, a następnie obliczanie iloczynu tych macierzy na dwa sposoby: za pomocą wielowątkowości wątków oraz za pomocą wielowątkowości biblioteki `Parallel`.
- **MultiplyMatricesParallelThreads:** Metoda służąca do obliczania iloczynu dwóch macierzy przy użyciu wielowątkowości wątków. Macierze są dzielone na części i każda część jest obliczana przez osobny wątek.
- **MultiplyMatricesParallelLibrary:** Metoda służąca do obliczania iloczynu dwóch macierzy przy użyciu wielowątkowości z biblioteki `Parallel`. Podobnie jak w poprzedniej metodzie, macierze są dzielone na części, ale obliczenia są wykonywane asynchronicznie przez równoległe wykonanie.
- **DisplayMatrix:** Metoda do wyświetlania macierzy w oknie tekstowym interfejsu użytkownika.
- **ApplyFiltersParallel:** Obsługuje przycisk "Zastosuj filtry" w interfejsie użytkownika. Ta metoda służy do aplikowania filtrów na wczytanym obrazie, takich jak przekształcenie w odcienie szarości, lustrzane odbicie, progowanie i negatyw. Filtry są stosowane równoległe, ale nie są uruchamiane na osobnych wątkach, tylko synchronicznie w tym samym wątku interfejsu użytkownika.
- **ApplyGrayscale:** Metoda do przekształcania obrazu na odcienie szarości.
- **ApplyMirror:** Metoda do wykonania lustrzanego odbicia obrazu.
- **ApplyThreshold:** Metoda do progowania obrazu (tj. przekształcenia obrazu na obraz binarny w oparciu o zadany próg jasności).
- **ApplyNegative:** Metoda do uzyskania negatywu obrazu.
- **button3_Click:** Obsługuje kliknięcie przycisku "Wczytaj obraz" w interfejsie użytkownika. Pozwala użytkownikowi wybrać obraz do wczytania i wyświetlenia w interfejsie użytkownika.

3. Charakterystyka funkcji używanych w pliku `Form1.cs`

`ApplyGrayscale(Bitmap inputImage, Bitmap processedImage):`

- Metoda ta przyjmuje dwa obrazy jako argumenty: `inputImage`, który jest oryginalnym obrazem, i `processedImage`, który będzie zawierał przetworzony obraz w odcieniach szarości.
- Następnie iteruje przez każdy piksel obrazu wejściowego (`inputImage`).
- Dla każdego piksela oblicza nową wartość piksela w odcieniach szarości, korzystając z formuły: $\text{grayValue} = 0.299 * R + 0.587 * G + 0.114 * B$, gdzie `R`, `G` i `B` są wartościami składowych koloru Red, Green i Blue piksela.
- Tworzy nowy kolor na podstawie obliczonej wartości odcienia szarości i przypisuje go do odpowiedniego piksela w obrazie przetworzonym (`processedImage`).

`ApplyMirror(Bitmap inputImage, Bitmap processedImage):`

- Ta funkcja przyjmuje obraz wejściowy `inputImage` i obraz przetworzony `processedImage`.
- Następnie iteruje przez każdy piksel obrazu wejściowego.
- Piklele są kopiowane z lewej strony obrazu wejściowego do prawej strony obrazu przetworzonego, co prowadzi do efektu lustrzanego odbicia.

`ApplyThreshold(Bitmap inputImage, Bitmap processedImage, int threshold):`

- Metoda ta przyjmuje trzy argumenty: obraz wejściowy `inputImage`, obraz przetworzony `processedImage` oraz wartość progu `threshold`.
- Przetwarza każdy piksel obrazu wejściowego.

- Oblicza wartość odcienia szarości dla każdego piksela podobnie jak w funkcji ApplyGrayscale.
- Porównuje wartość odcienia szarości z wartością progu. Jeśli wartość odcienia szarości jest większa niż próg, ustawia piksel na biały, w przeciwnym razie na czarny.
- Ustawia przetworzone piksele w obrazie wynikowym (processedImage).

ApplyNegative(Bitmap inputImage, Bitmap processedImage):

- Ta funkcja działa podobnie do funkcji ApplyGrayscale, przyjmując obrazy wejściowy i przetworzony.
- Iteruje przez każdy piksel obrazu wejściowego.
- Dla każdego piksela oblicza negatyw, zmieniając wartości kanałów kolorów na przeciwne wartości (255 - wartość kanału).
- Ustawia przetworzone piksele w obrazie wynikowym.

MultiplyMatricesParallelThreads(int[,] matrixA, int[,] matrixB, int threadCount):

- Ta metoda wykonuje mnożenie dwóch macierzy równoległe za pomocą wątków.
- Najpierw pobiera wymiary macierzy: liczba wierszy i kolumn.
- Następnie tworzy nową macierz, która będzie przechowywała wynik mnożenia.
- Liczba elementów w macierzy wynikowej jest dzielona przez liczbę wątków, aby określić ile elementów każdy wątek będzie obliczał.
- Następnie tworzone są wątki, gdzie każdy wątek jest odpowiedzialny za obliczenie określonego zakresu elementów macierzy wynikowej.
- Każdy wątek iteruje przez odpowiedni zakres elementów macierzy wynikowej i oblicza ich wartości poprzez odpowiednie przemnożenie i sumowanie elementów macierzy wejściowych.
- Po zakończeniu obliczeń wątki łączą się z głównym wątkiem za pomocą metody Join().
- Ostatecznie funkcja zwraca macierz wynikową.

MultiplyMatricesParallelLibrary(int[,] matrixA, int[,] matrixB, int threadCount):

- Ta funkcja wykonuje mnożenie dwóch macierzy równoległe za pomocą biblioteki Parallel.
- Podobnie jak w poprzedniej metodzie, najpierw pobierane są wymiary macierzy i tworzona jest macierz wynikowa.
- Następnie liczba elementów macierzy wynikowej jest dzielona przez liczbę wątków, aby określić, ile elementów każdy wątek będzie obliczał.
- Za pomocą metody Parallel.For wykonuje się pętla równoległa, gdzie każda iteracja odpowiada jednemu wątkowi.
- W każdej iteracji obliczane są elementy macierzy wynikowej dla określonego zakresu elementów.
- Obliczenia są rozproszone pomiędzy wątki, a biblioteka Parallel automatycznie zarządza przypisaniem zadań do wątków.
- Ostatecznie funkcja zwraca macierz wynikową.