Wrocław, 7 maja 2024

Platformy Programistyczne `.NET` i `Java`

Laboratorium 3

Obliczenia wielowątkowe w technologii `.NET`

prowadzący: *Dr inż. Radosław Idzikowski, mgr inż. Michał Jaroszczuk*

1 Cel laboratorium

Celem laboratorium jest zapoznanie się z podstawami przetwarzania wielowątkowego w technologii `.NET`. W ramach zajęć należy stworzyć program w języku `C#`, który pozwoli na przyspieszenie obliczeń poprzez zrównoleglenie kodu. Będzie ono wykonane z użyciem funkcji zarówno niskiego jak i wysokiego poziomu, a następnie zostanie zbadane uzyskane przyspieszenie.

Praca będzie oceniania na bieżąco podczas zajęć. **Ukończenie każdego etapu powinno być zgłoszone prowadzącemu w celu akceptacji i odnotowania postępów**. Sam program należy umieścić na repozytorium `github` i wysłać zaproszenie do prowadzącego. Czas na wykonanie zadania to dwa zajęcia laboratoryjne. Podczas pierwszego spotkania trzeba wykonać co najmniej jedno zadanie zdefiniowane w Rozdziale 3.

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Wielowątkowe mnożenie macierzy z wykorzystaniem klasy `Thread`.
2. Biblioteka `Parallel` oraz analiza przyśpieszenia.
3. Przetwarzanie obrazów.

Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Link do repozytorium należy przesłać dopiero po oddaniu i ocenieniu pracy na laboratorium.

3 Opis Zadań

W tej sekcji zostaną kolejno omówione wszystkie zadania do wykonania podczas laboratorium. Komputery w laboratorium zostały przygotowane do przeprowadzenia zajęć.

3.1 Zadanie 1

W ramach zadania należy wykonać aplikację w technologii `.NET 8.0`, która pozwoli na sprawdzenie przyśpieszenia obliczeń równoległych względem sekwencyjnych, na przykładzie mnożenia macierzy. Mnożenie macierzy może być rozbite na n x m niezależnych etapów, gdzie n jest liczbą wierszy, a m liczbą kolumn macierzy wynikowej. Na tym etapie aplikacja może działać w konsoli, jednak zalecane jest tworzenie docelowo aplikacji okienkowej (np.: `Windows Forms`). Poniżej wypisano założenia, które powinien spełniać program:

- Macierze do obliczeń powinny być wygenerowane losowo (mogą być kwadratowe), z uwzględnieniem możliwości wprowadzania rozmiaru macierzy jako parametr (może być to robione w kodzie).
- Napisany program powinien przyjmować jako parametr liczbę wątków, która zostanie użyta do obliczeń.
- Napisany algorytm powinien być skalowalny w zależności od liczby użytych wątków. Oznacza to, że nie ma potrzeby pisać osobno algorytmu sekwencyjnego, tylko należy uruchomić algorytm równoległy na jednym wątku.
- Napisany program powinien zwracać czas potrzebny na pomnożenie macierzy o danym rozmiarze.
- Program powinien być napisany z uwzględnieniem paradygmatów programowania obiektowego.
- Napisany program powinien mieć możliwość wyświetlenia macierzy wejściowych i macierzy wynikowej tak aby sprawdzić, czy obliczenia są poprawne (przy złej implementacji blokad wątków, mogą się popsuć). Do badania czasów obliczeń wypisywanie należy zakomentować, gdyż zaburza ono wyniki.
- Przy oddaniu programu należy pokazać przyspieszenie obliczeń przy użyciu wielu wątków w porównaniu do obliczania sekwencyjnego. Najlepsze efekty widoczne są przy macierzach o większych rozmiarach ($\geqslant 100$).

Wprowadzenie do wątków

Podstawową klasą do zarządzania wątkami niskopoziomowo jest klasa `Thread`. Podczas tworzenia obiektu wątka jako parametr możemy przekazać funkcję w formie delegata. Funkcja musi być statyczna oraz nie może zwracać żadnej wartości. Może ona natomiast posiadać parametry, które należy przekazać z użyciem wyrażenia lambda, bądź metody `ParameterizedThreadStart()`. Każdy nowo utworzony wątek musi zostać uruchomiony metodą `Start`. Poniżej przedstawiono prosty przykład z wyświetleniem powitania.

```
internal class Program
{
    static void Main(string[] args)
    {
        Thread thread = new Thread(Welcome);
        thread.Start();
        Console.WriteLine("Main: Hello!");
    }
    static void Welcome()
    {
        Console.WriteLine("Th: Hello!");
    }
}
```

Dla przypomnienia, funkcja `Main` również jest uruchomiana jako wątek, ale główny naszego programu (stąd jej podobna konstrukcja). Dla przedstawionego poniżej programu z większą liczbą wątków, warto zwrócić uwagę, że kolejność wyświetlania komunikatów jest różna dla kolejnych uruchomień programu, w zależności który wątek pierwszy uzyskał dostęp do zasobu współdzielonego w tym wypadku do konsoli (efekt tzw. wyścigu). Efekt jest lepiej widoczny przy większej liczbie wątków. W celu zarządzania większą liczbą wątków, można utworzyć tablicę obiektów o rozmiarze `n`. Dla poprawienia czytelności, każdemu wątkowi warto nadać unikatową nazwą i ją dodać w komunikacie. Uwaga! każdy wątek należy osobno uruchomić.

```
internal class Program
{
    static void Main(string[] args)
    {
        int n = 10;
        Thread[] threads = new Thread[n];
        for (int i = 0; i < n; i++)
        {
            threads[i] = new Thread(Welcome);
            threads[i].Name = String.Format("Thread: {0}", i+1);
        }
        foreach (Thread x in threads)
            x.Start();
        Console.WriteLine("Main: Hello!");
    }
    static void Welcome()
    {
        Console.WriteLine($"{Thread.CurrentThread.Name} : Hello!!");
    }
}
```

Synchronizacja wątków

Jak można było zaobserwować na poprzednich przykładach, wątki mogą otrzymywać dostęp do pamięci w różnym czasie oraz różnią się czasem "*życia*". Przez opisane zachowanie może więc dojść do utraty części danych, podczas gdy wątki będą korzystały ze wspólnej pamięci w tym samym czasie. Jeśli chcemy zagwarantować, aby wątek nadrzędny zaczekał na zakończenie wszystkich wątków, to czekanie na zakończenie każdego wątka z osobna wymusimy metodą `Join`. Jeśli jakieś pole w klasie może być modyfikowane przez inne wątki, należy je oznaczyć słowem kluczowym `volatile`. W poniższym przykładzie pole tego typu posłuży do zliczenia ile wątków zostało zakończonych. Uwaga! Odczyt wartości i jej zapis może nie być natychmiastowy (w przedstawionym fragmencie kodu wymuszono to metodą `Sleep`), więc krytyczny fragment kody warto zabezpieczyć mechanizmem blokady. Blokada może być realizowana zarówno, tak jak pokazano w przykładzie z użyciem słowa kluczowego `lock`, ale także przy użyciu obiektu klasy `Mutex`. W momencie nałożenia blokady na obiekt, żaden inny wątek nie może uzyskać do niego dostępu. W przykładzie posłużono się obiektem pomocniczym służącym tylko do odczytu, gdyby funkcja odpowiedzialna za obliczenia wewnątrz wątku była otoczona klasą to można by zablokować dostęp do jej obiektu. Usunięcie mechanizmu blokady spowoduje, że finalnie `count` $\neq$ `n`.

```
internal class Program
{
    public static volatile int count = 0;
    public static readonly object locker = new object();
    static void Main(string[] args)
    {
        int n = 4;
        Thread[] threads = new Thread[n];
        for (int i = 0; i < n; i++) threads[i] = new Thread(Counting);
        var watch = System.Diagnostics.Stopwatch.StartNew();
        foreach (Thread x in threads) x.Start();
        foreach (Thread x in threads) x.Join();
        watch.Stop();
        Console.WriteLine($"{count} threads ended in {watch.ElapsedMilliseconds} ms."
    );
    }
    static void Counting()
    {
        Thread.Sleep(500);
        lock (locker)
        {
```

```
            var tmp = count;
            Thread.Sleep(5);
            count = tmp + 1;
        }
    }
}
```

3.2 Zadanie 2

W tym zadaniu należy ponownie napisać program do mnożenia macierzy, ale tym razem z wykorzystaniem biblioteki wysokiego poziomu `Parallel`. Niniejsza biblioteka pozwoli znacząco zredukować objętość kodu, ale prawdopodobnie uzyskane przyśpieszenie będzie mniejsze niż dla zrównoleglenia niskiego poziomu. Dlatego **w ramach zadania należy zmierzyć i przeanalizować przyśpieszenie** między wersją z wykorzystaniem klasy `Thread` a biblioteki `Parallel` w zależności od liczby użytych rdzeni. Wyniki wystarczy wypisać w formie uproszczonej tabeli. Należy pamiętać, iż wyniki muszą być uśrednioną wartością z kilku prób (pojedyncze czasy mogą przekłamywać).

Biblioteka `Parallel` dostarcza nam odpowiednik klasycznej pętli `for`, ale w formie metody przyjmującej następujące argumenty: (1) indeks początkowy, włącznie. (2) indeks końcowy, wyłączny, (3) obiekt opcji (nieobowiązkowo) oraz (4) funkcję lambda. W ramach opcji `ParallelOptions` możemy wymusić stopień zrównoleglenia (liczbę użytych wątków). Poniżej przedstawiono fragment kodu do równoległego uzupełnienia tablicy typu `int`. Ponadto dodano tablicę pomocniczą `threadUsed`, która zliczy wywołanie każdego wątku. Proszę zwrócić uwagę, że tablica jest długości maksymalnej liczby wątków w systemie, mimo wymuszenia liczenia tylko z wykorzystaniem czterech wątków, ponieważ nie mamy gwarancji jakie wątki zostaną użyte i na pewno nie będą to cztery pierwsze. Jednak obliczenia powinny zostać rozłożone równomiernie między cztery różne wątki.

```
internal class Program
{
    static void Main(string[] args)
    {

        int[] numbers = new int[100];
        int maxThreads = 4;
        ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism =
    maxThreads };
        int[] threadUesed = new int[Environment.ProcessorCount];
        Random random = new Random();
        Parallel.For(1, 100, opt , x => {
            numbers[x] = random.Next(1, 5000);
            threadUesed[Thread.CurrentThread.ManagedThreadId]++;
        });
        Console.WriteLine(string.Join(" ", threadUesed));
    }
}
```

W bibliotece `Parallel` poza odpowiednikiem pętli `for` jest również odpowiednik `foreach`. Działa analogicznie przyjmując następujące argumentu: (1) kolekcję danych, (2) opcje oraz (3) funkcję lambda. Poniżej przedstawiono kompletny kod dla grupy studentów, którzy mają przystąpić do egzaminu. Przy założeniu, że metoda przystąpienia do egzaminu jest najbardziej obciążająca system (symulowane ponownie metodą `Sleep`), to właśnie ta metoda zostanie wywołane równolegle dla każdego studenta przy użyciu biblioteki `Parallel`.

```
internal class Program
{
    private static  Random Random = new Random();
    private static  float [] Grades = new float[] {2.0f, 3.0f, 3.5f, 4.0f, 4.5f, 5.0f
     };
    class Student
    {
```

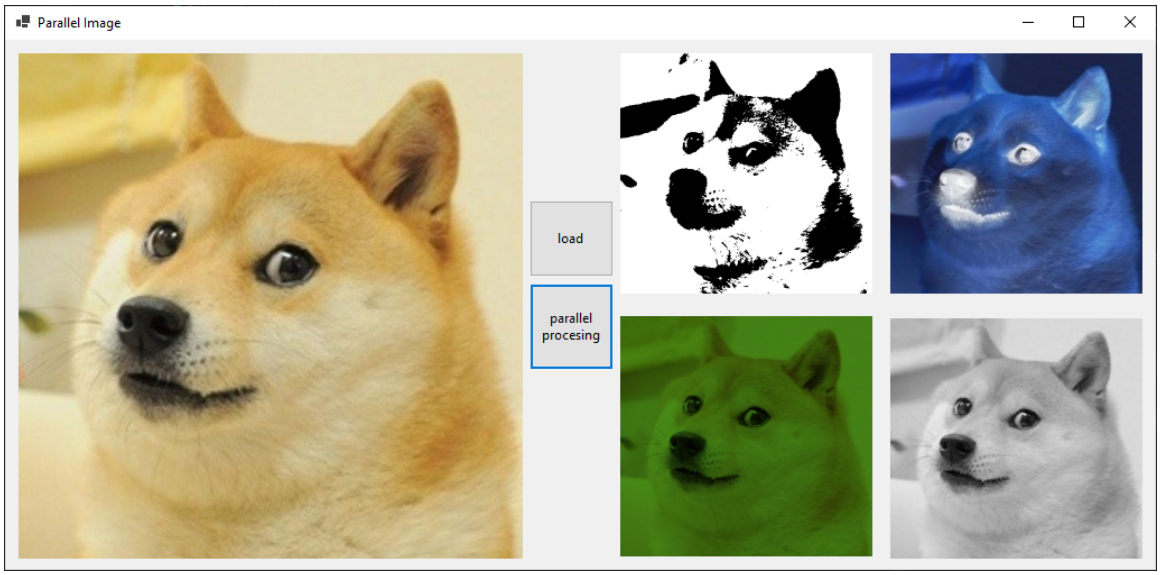
```
7         public int Id { get; set; }
8         public float? Grade { get; set; }
9         public void examination ()
10        {
11            Thread.Sleep (500);
12            Grade = Grades[Random.Next(Grades.Length)];
13        }
14        public override string ToString()
15        {
16            return $"Student {Id}, grade: {Grade}";
17        }
18    }
19    static void Main(string[] args)
20    {
21        int n = 10;
22        ParallelOptions opt = new ParallelOptions() { MaxDegreeOfParallelism = n };
23        List<Student> students = new List<Student>();
24        var watch = System.Diagnostics.Stopwatch.StartNew();
25        for (int i = 0; i < 10; i++) students.Add(new Student() { Id= i });
26        Parallel.ForEach(students, opt, student => student.examination());
27        foreach (Student student in students) Console.WriteLine(student.ToString());
28        watch.Stop();
29        Console.WriteLine($"{n} threads ended in {watch.ElapsedMilliseconds} ms.");
30    }
31}
```

3.3 Zadanie 3

Podstawowym celem zadania jest zaprojektowanie prostej aplikacji okienkowej do przetwarzania obrazów, która pozwoli zastosować wiele filtrów na jednym obrazie jednocześnie. Niezbędne będzie wyświetlenie obrazu przed jego przetworzeniem i zestawu obrazów po przetworzeniu. Pojedynczy filtr powinien być zrobiony sekwencyjnie, natomiast zrównolegleniu podlegać będzie liczba filtrów, tzn. jedną filtr powinien wykonywać jeden wątek. Można użyć wybranej przez siebie technologii (`Thread` lub `Parallel`). Należy zaimplementować co najmniej cztery różne operacje przetwarzania obrazów, np:. progowanie, odcienie szarości, negatyw, wykrywanie krawędzi, odbicie lustrzane itp.

W celu osiągnięcia maksymalnej oceny za to zadanie, zalecenie jest użycie `Windows Forms` lub innej technologi pozwalającej tworzyć `GUI`. W przypadku `Windows Forms` przydatne będą dwie nowe kontrolki: (1) `openFileDialog` oraz `PictureBox`. Kontrolka `openFileDialog` służyć będzie do otwarcia okienka pozwalającego na wybór pliku obrazka, który chcemy wyświetlić. W przypadku `PictureBox` nasze zadanie organiczy się do ustawienia odpowiednich wymiarów, żeby wczytany obrazek był czytelny. Ponadto warto zmienić tryb wyświetlania obrazu `SizeMode`, który pozwoli wyświetlić przeskalowany lub rozciągnięty obraz. W przypadku `openFileDialog` możemy wprowadzić filtr, żeby ograniczyć wyświetlanie pliku po rozszerzeniu, np: `jpg files (*.jpg)|*.jpg|All files (*.*)|*.*`, warto ustawić jeszcze `FilterIndex` na wartość 1, aby domyślnie były wyświetlanie wyłącznie pliki `.jpg`.

```
1 private Bitmap? img;
2
3 private void buttonLoad_Click(object sender, EventArgs e)
4 {
5     openFileDialog1.ShowDialog();
6     var file = openFileDialog1.FileName;
7     if (file != null)
8     {
9         img = new Bitmap(file);
10        pictureBox1.Image = img;
11    }
12}
```



Rysunek 1: Poglądowe zdjęcie aplikacji

Z `openFileDialog` można wyciągnąć `string` ze ścieżką do pliku. Następnie można skorzystać z klasy `Bitmap`, aby utworzyć obiekt z wczytanym obrazkiem na podstawie ścieżki. Obiekt typu `Bitmap` można bezpośrednio przypisać do pola `Image` w naszym `PictureBox`, aby wyświetlić obrazek. W ramach obiektu `Bitmap` mamy dostęp do metadanych obrazu, m. in. wysokości – `img.Height` czy szerokości – `img.Width`. Do pojedynczego pixela dostajemy się metodą `img.GetPixel(x, y)`, która zwraca `Color`, który ma następujące atrybuty

- `R` – wartość czerwonego kanału (`RGB`),
- `G` – wartość zielonego kanału (`RGB`),
- `B` – wartość niebieskiego kanału (`RGB`),
- `GetHue()` – wartość barwy (`HSL`),
- `GetSaturation()` – wartość nasycenia (`HSL`),
- `GetBrightness()` – wartość światła białego (`HSL`),

Kolor konkretnego pixela możemy ustawić z użyciem metody `img.SetPixel(x, y, Color.Black)`, która jako trzeci parametr przyjmuje `Color`. Można zastosować jeden z wielu już zdefiniowanych kolorów lub na podstawie wartości `Color.FromArgb(alpha, red, green, blue)`.