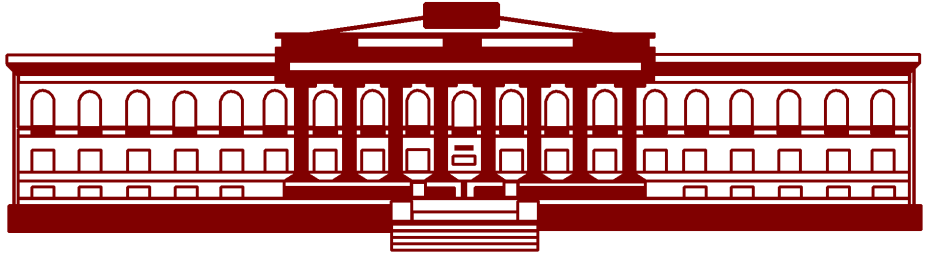
**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**імені ТАРАСА ШЕВЧЕНКА**

Факультет інформаційних технологій

**Кафедра прикладних інформаційних систем**

****

**Звіт**

до лабораторної роботи №4

з дисципліни «**Паралельні та розподілені обчислення**»

ВАРІАНТ №16

**Виконано:**

студ. групи ПП- 31, підгрупа 2

Селецький Віктор

**Київ – 2023**

Тема: Технології паралельного та розподіленого обчислення в різних мовах програмування

Ціль: освоїти реалізацію паралельних обчислень за допомогою

різних мов та технологій програмування.

Завдання: Обрати одну технологію паралельних та розподілених обчислень

обраної мови програмування та підготувати короткі теоретичні відомості про

застосування цієї технології та практичну реалізацію. Порівняти результати

роботи практичної реалізації обраної технології із результатами роботи

програми з послідовним виконанням. Зробити висновок.

Порівняти результати практичної реалізації з результатами однієї з

розглянутих технологій практичної роботи 2-3. Зробити висновок.

Виконання:

Task Parallel Library (TPL) є однією з ключових бібліотек в C#, яка дозволяє створювати багатопотокові застосунки та ефективно використовувати доступні ресурси системи. Основним класом TPL є Task, який може бути використаний для запуску асинхронних операцій.

Основні концепції TPL включають в себе:

Task - представлення асинхронної операції, яку можна запустити та отримати результат.

Task Scheduler - це механізм, який визначає, який потік повинен виконувати задачу та коли вона має бути виконана.

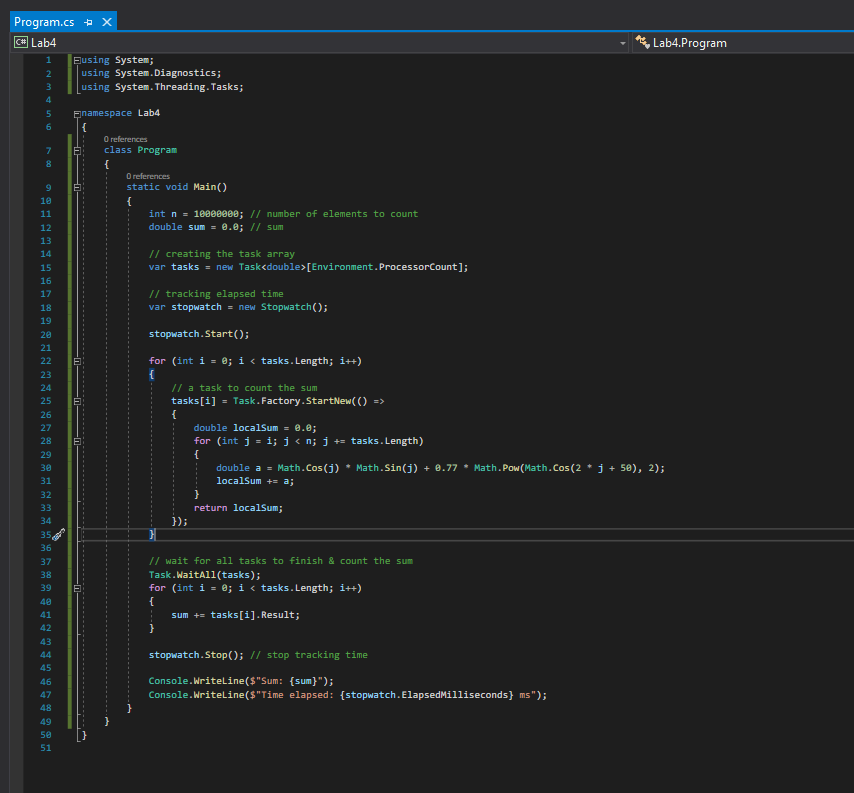
Parallel.ForEach - це метод, який дозволяє ітерувати колекцію елементів та виконувати на них операції паралельно.

PLINQ - це бібліотека, яка дозволяє виконувати запити до колекцій даних паралельно.

Також TPL містить багато інших корисних класів та методів, наприклад, TaskFactory, TaskCompletionSource, TaskContinuationOptions та ін., які дозволяють розширити функціональність та забезпечити кращу продуктивність застосунків, що використовують багатопотоковість.

Фактично TPL пропонує вищий рівень абстракції. В її основі лежить завдання (task), що є еквівалентом потоку (thread), але більш легким (в сенсі ресурсів) та таким, що не потребує створення потоку ОС. Коротко кажучи, завдання є легшим способом виконати щось асинхронно, ніж потік.

Код:



using System;

using System.Diagnostics;

using System.Threading.Tasks;

namespace Lab4

{

class Program

{

static void Main()

{

int n = 10000000; // number of elements to count

double sum = 0.0; // sum

// creating the task array

var tasks = new Task<double>[Environment.ProcessorCount];

// tracking elapsed time

var stopwatch = new Stopwatch();

stopwatch.Start();

for (int i = 0; i < tasks.Length; i++)

{

// a task to count the sum

tasks[i] = Task.Factory.StartNew(() =>

{

double localSum = 0.0;

for (int j = i; j < n; j += tasks.Length)

{

double a = Math.Cos(j) \* Math.Sin(j) + 0.77 \* Math.Pow(Math.Cos(2 \* j + 50), 2);

localSum += a;

}

return localSum;

});

}

// wait for all tasks to finish & count the sum

Task.WaitAll(tasks);

for (int i = 0; i < tasks.Length; i++)

{

sum += tasks[i].Result;

}

stopwatch.Stop(); // stop tracking time

Console.WriteLine($"Sum: {sum}");

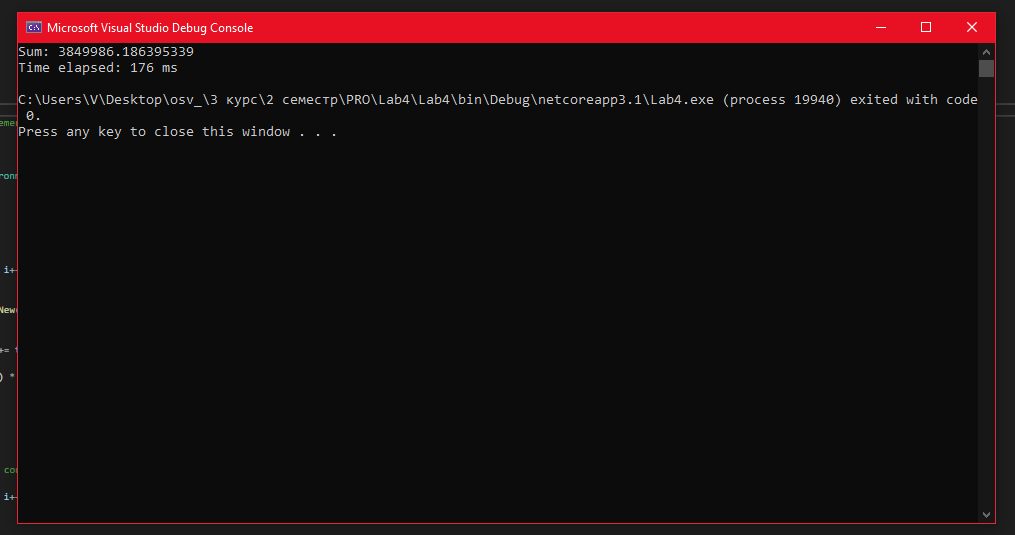
Console.WriteLine($"Time elapsed: {stopwatch.ElapsedMilliseconds} ms");

}

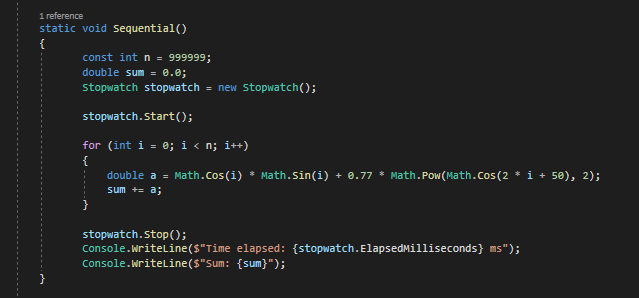
}

}

Результат виконання:



Послідовно:



static void Sequential()

{

const int n = 999999;

double sum = 0.0;

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

for (int i = 0; i < n; i++)

{

double a = Math.Cos(i) \* Math.Sin(i) + 0.77 \* Math.Pow(Math.Cos(2 \* i + 50), 2);

sum += a;

}

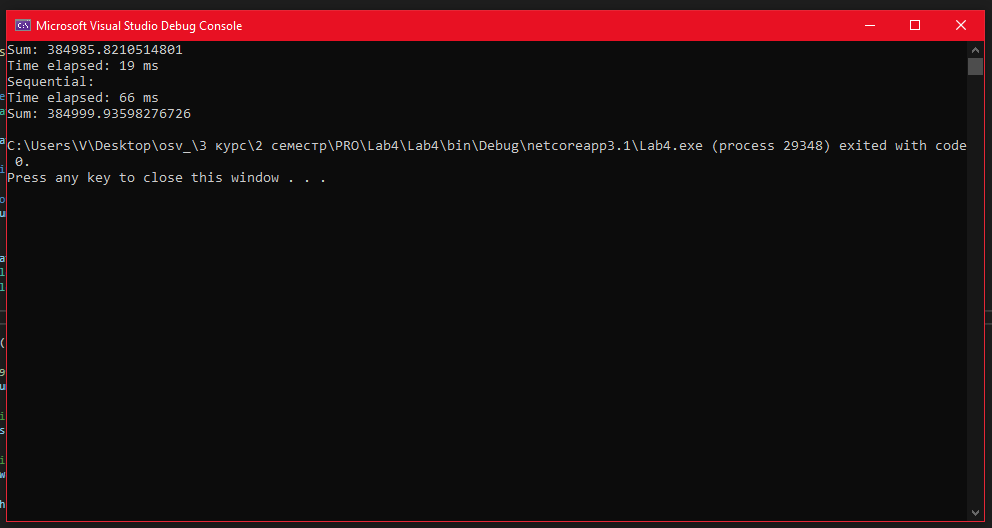
stopwatch.Stop();

Console.WriteLine($"Time elapsed: {stopwatch.ElapsedMilliseconds} ms");

Console.WriteLine($"Sum: {sum}");

}

Результат виконання:



Таблиця для порівняння:

| **Кількість членів** | **C++ (без мультипоточності)** | **C++ (з мультипоточністю)** | **Python (без мультипоточності)** | **Python (з мультипоточністю)** | **C++ (OpenMP)** | **C# (TPL)** | **C# (Послідовно)** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 100 | 0.000008 сек | 0.000033 сек | 0.001820 сек | 0.000340 сек | 0.000298 сек | 0.007 сек | 0 cек |
| 1000 | 0.000033 сек | 0.000048 сек | 0.000042 сек | 0.000076 сек | 0.001325 сек | 0.007 сек | 0 сек |
| 10000 | 0.000310 сек | 0.000218 сек | 0.000571 сек | 0.000295 сек | 0.001058 сек | 0.007 сек | 0 сек |
| 999999 | 0.025255 сек | 0.004491 сек | 0.056196 сек | 0.013505 сек | 0.076945 сек | 0.019 сек | 0.065 сек |

Висновки:

Як можна побачити, код з використанням паралельних задач (Task Parallel Library) сповільнює виконання задачі за умови малої кількості ітерацій циклу. Але якщо збільшити кількість необхідних ітерацій циклу, то розпаралелювання пришвидшує виконання в ~3.5 рази. Також можна побачити, що код на C++ з використанням мультипоточності швидший за всі інші наведені тут програми за умови великої кількості ітерацій, але він також сповільнюється на малій кількості ітерацій. З цього можна зробити висновок, що для задач, які не потребують великої кількості операцій, доцільніше використовувати послідовне виконання, оскільки використання мультипоточності буде в таких випадках цифровим аналогом стрільби з гармати по горобцях і лише сповільнить код.