**1. Вступ**

У даній роботі буде розглянуто способи паралелізації програм у мові програмування С# на платформі .NET, зокрема – бібліотеку паралельних задач (TPL). З використанням даної бібліотеки буде виконана спроба паралізації алгоритму на рядках «Ахо карасик», результати буде порівняно з аналогічною реалізацією написаною мові С++ та OpenMP.

У мові С# є можливість розпаралелювати програми створюючи потоки напряму (використовуючи клас System.Threading.Thread). Але з починаючи з .NET 4.0, рекомендується використовувати TPL, адже вона неймовірно зручна у використанні та має низький поріг входження. TPL можна використовувати для усіх класів паралельних задач (однак, як, буде показано далі, не завжди це взагалі має сенс).

**2. Вимоги для початку використання**

Для того, щоб почати використовувати TPL – необхідний лише .NET 4.0 (власне, у цій версії і з’явилась дана бібліотека) або вище.

**3. Загальна теорія**

Варто зазначити, що у своїй реалізації клас Task все одно використовує звичайні потоки (Thread), тож TPL можна використовувати разом із засобами розпаралелювання програм, що були до неї (наприклад, конструкція lock(object) { }).

3.1 Створення та запуск задач

Основний функціонал TPL сконцентровано у класі System.Threading.Tasks.Task. Даний клас описую окрему задачу, котра буде запущена асинхронно в одному за потоків пула потоків (однак, є можливість також запустити її синхронно в головному потоці).

Аби запустити деяку задачу необхідно створити екземпляр типу Task проініціалізувавши його делегатом (це може в тому числі і лямбда вираз), та викликати його метод Start. Наприклад:

Task task = new Task(() => Console.WriteLine(Thread.CurrentThread.ManagedThreadId));

task.Start();

Аналогічного результату можна досягнути скориставшись або статичним методом Run класу Task або статичним полем Factory та його статичним методом StartNew. І Run і StartNew в якості аргументу приймають делегат та повертають екземпляр типу Task, що був створений.

Task task = Task.Run(() => Console.WriteLine(Thread.CurrentThread.ManagedThreadId));

Task task = Task.Factory.StartNew(() => Console.WriteLine(Thread.CurrentThread.ManagedThreadId));

На практиці найчастіше використовується метод Run.

3.2 Очікування завершення виконання задач

Варто сказати, що, оскільки задачі виконуються асинхронно, основний потік, після запуску задачі, продовжить своє виконання.

Наприклад, у наступному прикладі, напис «Hello world» не буде виведено на екран, адже основний потік завершиться раніше за той, у якому буде виконуватись задача, тож виконання задачі буде зупинено, а программу – завершено.

static void Main(string[] args)

{

Task.Run(() =>

{

Thread.Sleep(50000);

Console.WriteLine(“Hello world”));

}

}

У разі, якщо для подальшого виконання якогось потоку необхідно дочекатись завершення якоїсь задачі – необхідно викликати метод Wait на екземплярі даної задачі. Можна змінити попередній приклад наступним чином, аби напис «Hello world» будо виведено.

static void Main(string[] args)

{

Task task = Task.Run(() =>

{

Thread.Sleep(50000);

Console.WriteLine(“Hello world”));

}

task.Wait();

}

Якщо необхідно дочекатись завершення деякого набору задач, рекомендується використовувати статичний WaitAll метод класу Task, в який необхідно передати ті задачі (або склавши їх попередньо в масив, або передавши як параметри, через кому).

Task task1 = Task.Run(() => Console.WriteLine("1"));

Task task2 = Task.Run(() => Console.WriteLine("2"));

Task task3 = Task.Run(() => Console.WriteLine("3"));

Task.WaitAll(task1, task2, task3);

// або - Task.WaitAll(new Task[] { task1, task2, task3 });

Console.WriteLine("all tasks ended")

Якщо необхідно дочекатись завершення хоча б однієї з задач – можна використовувати метод WaitAny метод класу Task, який викликається аналогічно методу WaitAll.

Task task1 = Task.Run(() => Console.WriteLine("1"));

Task task2 = Task.Run(() => Console.WriteLine("2"));

Task task3 = Task.Run(() => Console.WriteLine("3"));

Task.WaitAny(task1, task2, task3);

// або - Task.WaitAny(new Task[] { task1, task2, task3 });

Console.WriteLine("one of tasks ended")

3.3 Передача параметрів у функцію, що буде виконуватись паралельно

Існує два основних способи передати параметри у функцію, що буде виконуватись паралельно.

Перший – це скористатись замиканням. Наприклад:

static void Print(int intField, string stringField, bool boolField)

{

Console.WriteLine($"int: {intField} | string: {stringField} | bool: {boolField}");

}

static void Main(string[] args)

{

Task task = Task.Run(() => Print(5, "Hello", true));

task.Wait();

}

У даному прикладі на екран буде виведено стрічку «int: 5 | string: Hello | bool: True».

Другий спосіб – скористатися однією за перегрузок методу StartNew, про який говорилося раніше (варто зазначити, що інщі способи запуску задачі – не підходять, оскільки відповідні їм методи не мають необхідних перегрузок). У такий спосіб можна запустити лише функцію, що приймає один параметр типу object. Тобто для передачі декількох параметрів у функцію необхідно буде створити окрему структуру для збереження цих параметрів. Приклад використання:

struct MyStruct

{

public int intField;

public string stringField;

public bool boolField;

public MyStruct(int intField, string stringField, bool boolField)

{

this.intField = intField;

this.stringField = stringField;

this.boolField = boolField;

}

}

static void Print(object state)

{

MyStruct myStruct = (MyStruct)state;

Console.WriteLine($"int: {myStruct.intField} | string: {myStruct.stringField} | bool: {myStruct.boolField}");

}

static void Main(string[] args)

{

Task task = Task.Factory.StartNew(Print, new MyStruct(5, "Hello", true));

task.Wait();

}

Результат виконання буде таким же, як і у попередньому прикладі.

Очевидно, що другий варіант – громіздкий, важкий для використання і розуміння, тож рекомендується використовувати перший спосіб.

3.4 Отримання результату роботи паралельної функції

У типу Task є generic «близнюк» тип Task<T>, тож його можна ініціалізувати різними типами. Task<T> ініціалізується типом значення, що повертає функція, яка буде виконуватись в задачі. У випадку просто типу таск Task – можна вважати, що він проініціалізован типом void (хоча це, звичайно ж не так). Наприклад:

Task<int> tint = Task.Run(() => { return 1; });

Task<string> tstr = Task.Run(() => { return "Hello"; });

Task<object> tobj = Task.Run(() => { return new object(); });

У екземплярів типу Task<T> наявні властивість Result, через яку можна отримати результат виконання функції, що була запущена в задачі. У випадку, якщо функція ще не була завершена, потік, що викликав Result буде призупинено до її завершення (аналогічно методу Wait).

static void Main(string[] args)

{

Task<int> task = Task.Run(() => { Thread.Sleep(5000); return 42; });

Console.WriteLine(task.Result);

}

У прикладі буде виведено «42», а потім програма завершиться.

**4. Розпаралелювання алгоритму «Ахо карасик».**

Розпаралелений та звичайний варіанти алгоритму (як на С# так і на C++), а також код тестування їх ефективності можна знайти на github репозиторії <https://github.com/gotthit/ParallelProgrammingProject> .

Усі тести проводились на машині за наступними характеристиками:

* Процесор: Intel Core i5-5200U
* Частота процесору: 2.2 ГГц
* Кількість ядер процесору: 2
* Об’єм оперативної пам’яті: 8 Гб

На машинах з іншими характеристиками результати тестування можуть різнитися.