Логвін Андрій ПІ-15-2

«TPL та ідеї розпаралелювання Ахо карасика»

**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 програму – 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 інші 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 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> .

Тестування 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.

**4.1 Основні ідеї**

4.1.1 Додавання стрічок у бор

Розпаралелити операцію додавання однієї стрічки в бор неможливо, оскільки виконання кожної ітерації циклу залежить від результату виконання попередньої операції (перехід до сина вузла).

З іншого боку, можна достатньо просто розпаралелити додавання масиву стрічок у бор. Для цього треба просто запустити операції додавання однієї стрічки паралельно. Однак варто зазначити, що два потоки не можуть одночасно працювати з одним вузлом бору (додавати йому синів, перевіряти наявність сина по ключу). Тому, в кожному вузлі будемо зберігати поле з об’єктом синхронізації для цього вузла, та помістимо роботу з вузлом у критичну секцію з даним об’єктом синхронізації. Таким чином в кожен момент часу з вузлом зможе працювати лише один потік, але декілька потоків зможуть виконувати одну і ту саму частину коду, якщо вони працюють з різними вузлами.

4.1.2 Попередній підрахунок суфіксних посилань та переходів

Більшість реалізацій Ахо карасика використовують ліниву рекурсію для знаходження суфіксних посилань та переходів. Проводити їх попередній підрахунок має сенс лише у випадку, коли проводиться багаторазовий пошук по різних текстах. А приріст швидкості для рішень з попереднім підрахунком, зазвичай дуже малий. Однак, попередній підрахунок суфіксних посилань та переходів представляє інтерес з точки зору його паралелизації. Тож написаний Ахо карасик підтримує як підрахунок лінивою рекурсією, так і попередній підрахунок.

Попередній підрахунок можна здійснити якщо запустити пошук в ширину від кореня бору. Таким чином для кожного вузла гарантується, що уся необхідна інформація вже була підрахована раніше.

Можна помітити, що усі вершини можна згрупувати по їх відстані від кореня на шари. При чому, для попереднього підрахунку у кожному з вузлів слою номер N, необхідна лише інформація зі шару номер N-1та N-2. Таким чином, інформацію для усіх вузлів на кожному шарі можна рахувати паралельно.

4.1.3 Пошук входження хоча б однієї стрічки з набору в тексті

Для даної операції в коді є два варіанти виконання – якщо попередній підрахунок був проведений, і якщо ні. Ці два варіанти дуже схожу в реалізації. Єдина відмінність – в другому випадку необхідно находу знаходити суфіксні посилання та переходи.

Аналогічно з додаванням стрічки в бор, розпаралелити підрахунок одного суфіксного посилання – неможливо. Однак, можна виконувати підрахунок декількох суфіксних посилань, у випадку якщо вони рахуються для різних вузлів, і жоден вузол не є потомком іншого. Реалізовувати будемо аналогічно додаванню стручки в бор, однак, потрібно бути обережним, аби потік не заблокував сам себе.

Сам же процес пошуку входження однієї зі стрічок у тесті для обох варіантів однаковий.

В процесі пошуку кожна наступна ітерація циклу залежить від попередньої (оскільки ми блукаємо по вершинах бору). Але, помітимо, що N та операція не залежить від (N-maxL) тої операції (де maxL – максимальна довжина слова, що було додане до бору). Таким чином, ми можемо розбити текст на блоки, таким чином, щоб вони накладалися одне на одного на maxL символів, та виконувати пошук паралельно для кожного з цих блоків. Кількість блоків, на яку треба розбити текст – було вирішено рахувати як sqrt(textSize/maxL), по аналогії з sqrt-декомпозицією. Таким чином, нами буде виконано maxL \* (sqrt(textSize/maxL) - 1) операцій.

**4.2 Тестування ефективності рішень**

Для виміру часу роботи, алгоритм запускався n разів. Для кожного разу замірявся час роботи. Для результуючого списку рахувалося усечене середнє. Алгоритми тестувалися на чотирьох наборах даних. Час вимірювався в секундах.

1) Кількість букв в алфавіті: 26. Кількість букв в тексті: 10^7. Кількість букв у кожному слові: 50. Кількість слів: 10^4.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | С# | | | | | C++ | | | | |
| Лінива рекурсія | | Предрахунок | | | Лінива рекурсія | | Предрахунок | | |
| add | find | add | recount | find | add | find | add | recount | find |
| Послідовно | 1.09 | 6.22 | 1.5 | 13.68 | 6.53 | 1 | 17.8 | 0.85 | 8.1 | 12 |
| Паралельно | 7.26 | 5.729 | 7.24 | 63.64 | 3 | 0.25 | 3.2 | 0.45 | 5.56 | 2.84 |
| Прискорення (посл/пар) | 0.15 | 1.086 | 0.2 | 0.215 | 2.18 | 4 | 5.5 | 1.89 | 1.457 | 6 |

2) Кількість букв в алфавіті: 256. Кількість букв в тексті: 10^7. Кількість букв у кожному слові: 20. Кількість слів: 10^3.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | С# | | | | | C++ | | | | |
| Лінива рекурсія | | Предрахунок | | | Лінива рекурсія | | Предрахунок | | |
| add | find | add | recount | find | add | find | add | recount | find |
| Послідовно | 0.024 | 3.69 | 0.03 | 2.77 | 3.17 | 0.037 | 15.24 | 0.036 | 3.567 | 10.17 |
| Паралельно | 0.18 | 3 | 0.25 | 6.62 | 1.91 | 0.0085 | 2.369 | 0.02 | 1.49 | 2.34 |
| Прискорення (посл/пар) | 0.133 | 1.23 | 0.12 | 0.42 | 1.66 | 4.35 | 6.34 | 1.8 | 2.39 | 4.35 |

3) Кількість букв в алфавіті: 26. Кількість букв в тексті: 10^7. Кількість букв у кожному слові: 10. Кількість слів: 5\*10^4.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | С# | | | | | C++ | | | | |
| Лінива рекурсія | | Предрахунок | | | Лінива рекурсія | | Предрахунок | | |
| add | find | add | recount | find | add | find | add | recount | find |
| Послідовно | 1.27 | 11.54 | 1.49 | 11.257 | 9.16 | 0.91 | 24.6 | 0.91 | 6.04 | 15.18 |
| Паралельно | 6.56 | 12.95 | 6.91 | 26.9 | 4.07 | 0.41 | 5.54 | 0.4 | 3.16 | 2.75 |
| Прискорення (посл/пар) | 1.19 | 0.89 | 0.216 | 0.419 | 2.25 | 2.22 | 4.44 | 2.11 | 1.91 | 5.52 |

4) Кількість букв в алфавіті: 26. Кількість букв в тексті: 10^7. Кількість букв у кожному слові: 10^3. Кількість слів: 50.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | С# | | | | | C++ | | | | |
| Лінива рекурсія | | Предрахунок | | | Лінива рекурсія | | Предрахунок | | |
| add | find | add | recount | find | add | find | add | recount | find |
| Послідовно | 0.1 | 1.41 | 0.09 | 0.91 | 0.91 | 0.095 | 3.68 | 0.1 | 0.596 | 1.68 |
| Паралельно | 0.38 | 1.67 | 0.72 | 3.75 | 0.66 | 0.019 | 0.87 | 0.017 | 0.1 | 0.16 |
| Прискорення (посл/пар) | 0.26 | 0.844 | 0.125 | 0.24 | 1.28 | 5 | 4.22 | 5.59 | 5.96 | 10.5 |

**5. Висновок**

TPL – це дуже зручний інструмент для розпаралелювання програм, що має дуже низький поріг входження. Однак накладні витрати на створення потоків у такий спосіб є дуже великими (хоча це стосується не стільки TPL, скільки C# та .NET в цілому), тому не рекомендується створювати та видаляти велику кількість задач. У випадку з Ахо карасиком паралельна версія працювала значно довше за звичайну, тоді як у випадку з С++ та OpenMP розпаралелювання давало прискорення у 4-5 разів.

TPL дуже зручно використовувати для програмування додатків з графічним ітерфейсом користувача та інших аналогічних ситуацій, але для оптимізації алгоритмів TPL не підходить. Її можна використовувати тільки якщо ви впевнені, що буде створена невелика кількість задач, а операції, що будуть виконуватись паралельно, займають багато часу (так, наприклад, в операціях пошуку розпаралелювання інколи давало приріст в швидкості).