Паралельне програмування з використанням бібліотеки TPL та PLINQ

Клас ParallelEnumerable. Розпаралелювання запиту методом AsParallel (). Збереження порядку в PLINQ. Застосування методу AsOrdered (). Злиття результатів. Відміна запиту PLINQ. Виклик блокуючих функцій або функцій з інтенсивним введенням/виведенням. Ефективність використання PLINQ.

Лекція 4. Паралелізм даних з використанням PLINQ

PLINQ (Parallel Language - Integrated Query) - паралельна інтегрована мова запитів, тобто, це паралельна реалізація LINQ for Object. Відмінність PLINQ від LINQ полягає в тому, що запити виконуються паралельно, використовуючи усі доступні ядра і процесори. Таким чином, PLINQ застосовується для досягнення паралелізму даних усередині запиту.

Бібліотека PLINQ автоматизує усі етапи розпаралелювання обчислень, включаючи розподіл роботи на завдання, виконання цих завдань різними потоками і об'єднання результатів в одну вихідну послідовність.

При використанні методів бібліотеки, при виконанні запиту вхідна послідовність розбивається на частини, кожна з яких обробляється різними потоками, об'єднуючи потім результати в одну вихідну послідовність (рис. 4.1).

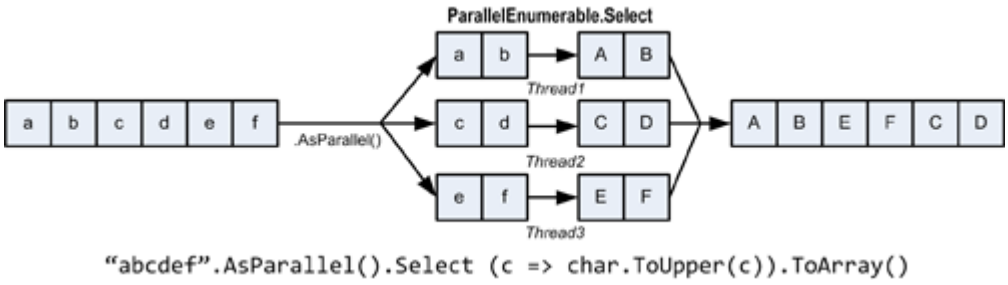


Рис. 4.1. Схема розпаралелювання запиту

Основною відмінністю є те, що PLINQ намагається повністю використовувати можливості усіх процесорів в системі. Це досягається шляхом розподілу джерела даних на сегменти і паралельного виконання запиту кожного сегменту в окремому робочому потоці на декількох процесорах. У багатьох випадках паралельне виконання означає, що запит виконується значно швидше.

1. Клас ParallelEnumerable

Основа PLINQ - це клас ParallelEnumerable, який визначений в просторі імен System.Linq. Це статичний клас, в якому визначено багато методів розширення, що підтримують паралельне виконання операцій. Ці методи наведено в таблиці 4.1.

http://msdn.microsoft.com/ru-ru/library/dd997425.aspx

Таблиця 4.1. – Методи класу ParallelEnumerable

Методи класу ParallelEnumerable	Опис
AsParallel	Точка входу для PLINQ. Вказує на необхідність розпаралелювання решти запиту, якщо це можливо.
AsSequential(Of TSource)	Вказує на необхідність послідовного виконання іншої частини запиту як непаралельного запиту LINQ.
AsOrdered	Вказує, що PLINQ повинен зберігати порядок початкової послідовності для іншої частини запиту або до тих пір, поки порядок не буде змінений, наприклад за допомогою оператора orderby.
AsUnordered(Of TSource)	Вказує, що PLINQ не повинен зберігати порядок початкової послідовності для іншої частини запиту.
WithCancellation(Of TSource)	Вказує, що PLINQ повинен періодично відстежувати стан наданого токена відміни і відміняти виконання при запиті.
WithDegreeOfParallelism(Of TSource)	Вказує максимальну кількість процесорів, яку повинен використовувати PLINQ для розпаралелювання запиту.
WithMergeOptions(Of TSource)	Надає підказку про те, як PLINQ повинен, якщо це можливо, виконувати злиття паралельних результатів в одну послідовність в потоці-споживачі.
WithExecutionMode(Of TSource)	Вказує, чи повинен PLINQ виконувати розпаралелювання запиту, навіть якщо згідно з поведінкою за замовчанням він виконуватиметься послідовно.
ForAll(Of TSource)	Багатопотоковий метод перелічування, який на відміну від ітерації результатів запиту дозволяє обробляти результати паралельно без попереднього злиття в потік-споживач.
Aggregate	Унікальне для PLINQ перевантаження, що забезпечує проміжну агрегацію локальних частин потоку, і надає функцію остаточної агрегації для об'єднання результатів усіх частин.

Далі розглянемо приклади застосування деяких з цих методів.

2. Розпаралелювання запиту методом AsParallel ()

Найбільш простим і зручним способом є виклик методу AsParallel () для джерела даних (наприклад, масиву).

Метод AsParallel() повертає джерело даних, інкапсульоване в екземплярі об'єкту типу ParallelQuery. Це дає можливість підтримувати методи розширення паралельних запитів. Після виклику цього методу запит розділяє джерело даних на частини і оперує з кожною з них так, щоб отримати максимальну перевагу від розпаралелювання. (Якщо розпаралелювання виявляється неможливим або неприйнятним, то запит виконується послідовно.) Таким чином, додавання в початковий код єдиного виклику методу AsParallel() виявляється достатньо для того, щоб перетворити послідовний запит LINQ на паралельний запит LINQ.

Для простих запитів це єдина необхідна умова.
Наступний приклад (http://msdn.microsoft.com/ru-ru/library/dd460714.aspx) демонструє різні способи реалізації розпаралелювання запиту для вибору з масиву чисел, які діляться на 10.

Приклад 4.1.

```
var source = Enumerable.Range(100, 20000);

        // Result sequence might be out of order.
        var parallelQuery = from num in source.AsParallel()
                            where num % 10 == 0
                            select num;

        // Process result sequence in parallel
        parallelQuery.ForAll((e) => DoSomething(e));

        // Or use foreach to merge results first.
        foreach (var n in parallelQuery)
        {
            Console.WriteLine(n);
        }

        // You can also use ToArray, ToList, etc
        // as with LINQ to Objects.
        var parallelQuery2 = (from num in source.AsParallel()
                              where num % 10 == 0
                              select num).ToArray();
```

Таким чином, для розпаралелювання запиту потрібно просто викликати метод AsParallel() для джерела даних.

В наступному прикладі наведена послідовна реалізація запиту для вибору парних чисел.

Приклад 4.2.

```
using System;
using System.Linq;
    class Example
    {
        static void Main()
        {
            int[] myArray = {-18, 15, -4, 69, 36, 11, 98, -7, -8};
            var criteria = from n in myArray
                           where n % 2 == 0
                           select n;
            Console.Write("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            Console.WriteLine("Нажмите любую кнопку!");
            Console.ReadKey();
        }
    }
```

Проста паралельна версія може бути такою:

```
using System;
using System.Linq;

    class Example
    {
        static void Main()
        {
            int[] myArray = {-18, 15, -4, 69, 36, 11, 98, -7, -8};
            var criteria = from n in myArray.AsParallel()
                           where n % 2 == 0
                           select n;
            Console.Write("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            Console.WriteLine("Нажмите любую кнопку!");
            Console.ReadKey();
        }
    }
```

Як видно з прикладів, метод AsParallel() викликається тільки до джерела даних. Решта запиту залишається без змін.

В наступному прикладі обидва запити виконуються в одному проекті.

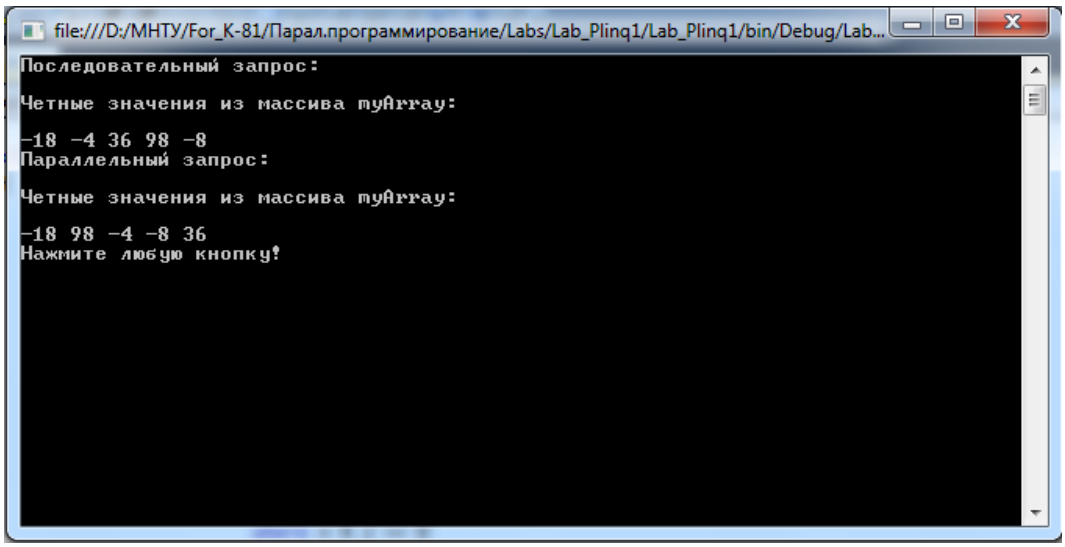
Приклад 4.3.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace Lab_Plinq1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] myArray = { -18, 15, -4, 69, 36, 11, 98, -7, -8 };
           //последовательный запрос
            var criteria = from n in myArray
                           where n % 2 == 0
                           select n;
            Console.WriteLine("Последовательный запрос: \n");
            Console.WriteLine("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            //параллельный запрос
            criteria = from n in myArray.AsParallel()
                           where n % 2 == 0
                           select n;
            Console.WriteLine("Параллельный запрос: \n");
            Console.WriteLine("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            Console.WriteLine("Нажмите любую кнопку!");
            Console.ReadKey();

        }
    }
}
```

Зверніть увагу, що результати відрізняються. При виконанні паралельного запиту результати відображаються не в тому порядку, в якому вони розташовані в масиві.



Наступний запит обчислює прості числа від 3 до 100000, використовуючи усі ядра процесора. В цьому прикладі використовується запис у формі лямбда виразу.

Приклад 4.4.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lab_Plinq1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Знаходимо прості числа за допомогою простого (неоптимізованого) алгоритму.
            IEnumerable<int> numbers = Enumerable.Range(3, 100000);
            var parallelQuery =
              from n in numbers.AsParallel()
              where Enumerable.Range(2, (int)Math.Sqrt(n)).All(i => n % i > 0)
              select n;

            int[] primes = parallelQuery.ToArray();
            foreach (int s in primes) Console.Write(s + " ");
            Console.WriteLine();
            Console.WriteLine("Нажмите любую кнопку!");
            Console.ReadKey();
        }
    }
}
```

В багатьох випадках цикл For можна замінити на запит Linq (PLINQ).

Приклад генерації відкритих/закритих ключів шифрування з лекції 3 можна замінити запитом PLINQ.

Приклад демонструє використання Parallel.For для генерації шести пар відкритих/закритих ключів паралельно.

Приклад 4.5.

```
namespace Lab4_Parallel
{
    class Program
    {
        static void Main(string[] args)
        {
            var keyPairs = new string[6];

            Parallel.For(0, keyPairs.Length,
                i => keyPairs[i] = RSA.Create().ToXmlString(true));
            for (int i = 0; i < 6; i++)
            {
                Console.WriteLine(keyPairs[i]);
                Console.ReadKey();
            }

        }
    }
}
```

Приклад 4.6.

Генерація ключів за допомогою LINQ:

```
string[] keyPairs =
  ParallelEnumerable.Range (0, 6)
  .Select (i => RSA.Create().ToXmlString (true))
```

```
  .ToArray();
```

3. Збереження порядку в PLINQ. Застосування методу AsOrdered ()

Ми бачили з прикладу 4.2, що результати паралельного запиту можуть бути не впорядкованими. Це відбувається через те, що при виконанні розпаралелювання масив розбивається на частини.

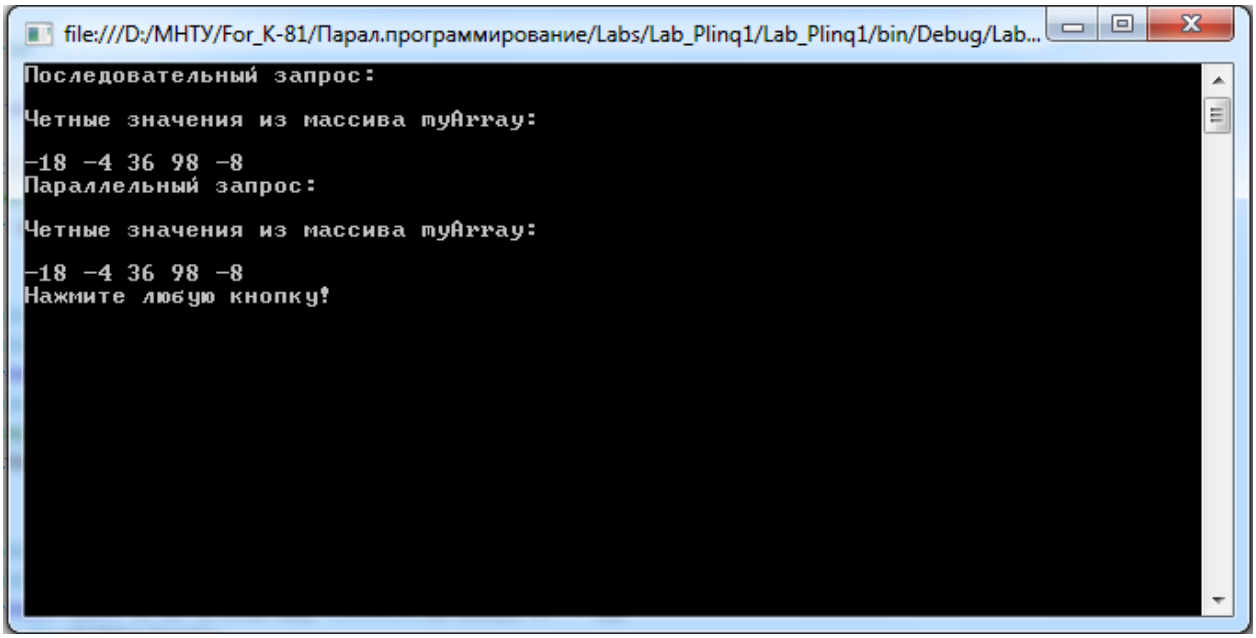
Більше того, результуючу послідовність слід розглядати як практично невпорядковану. Якщо ж результат повинен відбивати порядок організації джерела даних, то його треба запросити спеціально за допомогою методу AsOrdered (), визначеного в класі ParallelEnumerable.

Приклад 4.7.

```
namespace Lab_Plinq1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] myArray = { -18, 15, -4, 69, 36, 11, 98, -7, -8 };
            //последовательный запрос
            var criteria = from n in myArray
                           where n % 2 == 0
                           select n;
            Console.WriteLine("Последовательный запрос: \n");
            Console.WriteLine("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            //параллельный запрос
            criteria = from n in myArray.AsParallel().AsOrdered()
                       where n % 2 == 0
                       select n;
            Console.WriteLine("Параллельный запрос: \n");
            Console.WriteLine("Четные значения из массива myArray: \n");
            foreach (int i in criteria) Console.Write(i + " ");
            Console.WriteLine();
            Console.WriteLine("Нажмите любую кнопку!");
            Console.ReadKey();

        }
    }
}
```

Після виконання програми порядок вибору елементів в результуючій послідовності такий як і в початковій послідовності.



У наступному прикладі показаний неврегульований паралельний запит, який виконує фільтрацію для усіх елементів, відповідних умові, не намагаючись упорядкувати результати яким-небудь чином.

Приклад 4.8.

```
var cityQuery = (from city in cities.AsParallel()
                 where city.Population > 10000
                 select city)
                   .Take(1000);
```

У відповідь на цей запит не обов'язково видаються перші 1000 міст в початковій послідовності, які задовольняють умові, замість цього видається деякий набір з 1000 міст, що задовольняють умові. Оператори запиту PLINQ розділяють початкову послідовність на декілька підпослідовностей, які обробляються як одночасні завдання. Якщо збереження порядку не задане, результати з кожної частини передаються на наступний етап запиту в довільному порядку. Крім того, розподіл може призвести до створення підмножини результатів до того, як почнеться обробка елементів, що залишилися. Отриманий порядок може кожного разу відрізнятися.

У наступному прикладі перевизначається поведінка за замовчанням в початковій послідовності за допомогою оператора AsOrdered. Це гарантує повернення методом Take(Of TSource) перших 10 міст в початковій послідовності, що відповідають умові.

```
Var orderedCities = (from city in cities.AsParallel().AsOrdered()
                                  where city.Population > 10000
                                  select city)
                                 .Take(10);
```

Проте цей запит, можливо, не виконається так само швидко, як неврегульована версія, оскільки в ньому повинен відстежуватися початковий

порядок в усіх частинах початкової послідовності і під час злиття забезпечуватися відповідний порядок. Отже, метод AsOrdered рекомендується використовувати тільки при необхідності і тільки для тих частин запиту, які його вимагають. Якщо збереження порядку більше не потрібно, використовуйте метод **AsUnordered(Of Tsource)** для його відключення. У наступному прикладі це досягається шляхом створення двох запитів.

Приклад 4.9.

Перший запит.

```
var orderedCities2 = (from city in cities.AsParallel().AsOrdered()
                      where city.Population > 10000
                      select city)
                        .Take(1000);
```

Другий запит

```
var finalResult = from city in orderedCities2.AsUnordered()
                  join p in people.AsParallel() on city.Name equals p.CityName into
details
                  from c in details
                  select new { Name = city.Name, Pop = city.Population, Mayor =
c.Mayor };

foreach (var city in finalResult) { /*...*/ }
```

У наступному прикладі порядок початкової послідовності зберігається. Це іноді необхідно, наприклад, якщо деякі оператори запиту вимагають впорядкованої початкової послідовності для отримання правильних результатів.

Приклад 4.10.

```
namespace Lab_Plinq1
{
    class Program
    {
        static void Main(string[] args)
        {
            var source = Enumerable.Range(9, 10000);

            // Source is ordered; let's preserve it.
            var parallelQuery = from num in source.AsParallel().AsOrdered()
                                where num % 3 == 0
                                select num;

            // Use foreach to preserve order at execution time.
            foreach (var v in parallelQuery)
                Console.Write("{0} ", v);

            // Some operators expect an ordered source sequence.
            var lowValues = parallelQuery.Take(10);

        }
    }
}
```

4. Злиття результатів

В наступному прикладі показано, як вказати параметри злиття, яке застосовуватиметься до усіх наступних операторів в запиті PLINQ. Немає необхідності задавати параметри злиття явно, але це може підвищити продуктивність.

Приклад 4.11.

```
namespace MergeOptions
{
    using System;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading;

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

            var nums = Enumerable.Range(1, 10000);

            // Replace NotBuffered with AutoBuffered
            // or FullyBuffered to compare behavior.
            var scanLines = from n in nums.AsParallel()
                            .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                            where n % 2 == 0
                            select ExpensiveFunc(n);

            Stopwatch sw = Stopwatch.StartNew();
            foreach (var line in scanLines)
            {
                Console.WriteLine(line);
            }

            Console.WriteLine("Elapsed time: {0} ms. Press any key to exit.",
                              sw.ElapsedMilliseconds);
            Console.ReadKey();
        }

        // A function that demonstrates what a fly
        // sees when it watches television :-)
        static string ExpensiveFunc(int i)
        {
            Thread.SpinWait(2000000);
            return String.Format("{0} ************************************", i);
        }
    }
}
```

5. Відміна запиту PLINQ

Паралельний запит відміняється так само, як і задача. І в тому і в іншому випадку відміна спирається на структуру CancellationToken, що отримується з класу CancellationTokenSource. Ознака відміни, що отримується у результаті, передається запиту з допомогою методу WithCancellation (). Відміна паралельного запиту виконується методом Cancel(), який викликається для джерела ознак відміни. Головна різниця між

відміною паралельного запиту і відміною задачі полягає у видачі різних виключень. Коли відміняється паралельний запит, він генерує виключення OperationCanceledException, а не AggregateException.

У наступних прикладах показано два способи відміни запиту PLINQ. У першому прикладі показано, як відмінити запит, що складається в основному з обходу даних.У другому прикладі показано, як відмінити запит, який містить функцію користувача, що вимагає великих витрат комп'ютерних ресурсів.

Приклад 4.12.

```
namespace PLINQCancellation_1
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;

    class Program
    {
        static void Main(string[] args)
        {
            int[] source = Enumerable.Range(1, 10000000).ToArray();
            CancellationTokenSource cs = new CancellationTokenSource();

            // Start a new asynchronous task that will cancel the
            // operation from another thread. Typically you would call
            // Cancel() in response to a button click or some other
            // user interface event.
            Task.Factory.StartNew(() =>
            {
                UserClicksTheCancelButton(cs);
            });

            int[] results = null;
            try
            {
                results = (from num in source.AsParallel().WithCancellation(cs.Token)
                           where num % 3 == 0
                           orderby num descending
                           select num).ToArray();

            }

            catch (OperationCanceledException e)
            {
                Console.WriteLine(e.Message);
            }

            catch (AggregateException ae)
            {
                if (ae.InnerExceptions != null)
                {
                    foreach (Exception e in ae.InnerExceptions)
                        Console.WriteLine(e.Message);
                }
            }
```

```
            if (results != null)
            {
                foreach (var v in results)
                    Console.WriteLine(v);
            }
            Console.WriteLine();
            Console.ReadKey();

        }

static void UserClicksTheCancelButton(CancellationTokenSource cs)
        {
            // Wait between 150 and 500 ms, then cancel.
            // Adjust these values if necessary to make
            // cancellation fire while query is still executing.
            Random rand = new Random();
            Thread.Sleep(rand.Next(150, 350));
            cs.Cancel();
        }
    }
}
```

6. Виклик блокуючих функцій або функцій з інтенсивним введенням/виведенням

Іноді запити виконуються тривалий час не через інтенсивне використання центрального процесора, а тому що процесор чекає чогось, наприклад, завантаження web –сторінок або відповіді від апаратури. PLINQ може ефективно розпаралелювати такі запити, якщо вказати це шляхом виклику методу **WithDegreeOfParallelism** після виклику методу AsParallel.

Приклад 4.13

http://www.rsdn.ru/article/dotnet/Threading_In_C_Sharp_Part_3.xml#EKLAC

Як приклад давайте припустимо, що ми пишемо систему відеоспостереження і хочемо постійно об'єднувати зображення з чотирьох камер спостереження в одне складене зображення для відображення його на моніторі.

Ми можемо представити камеру у вигляді наступного класу:

```
class Camera
{
  public readonly int CameraID;
  public Camera (int cameraID) { CameraID = cameraID; }

  // Получить изображение с камеры:
  // возвращаем простую строку, а не изображение
  public string GetNextFrame()
  {
       Thread.Sleep (123); // Симулируем время, необходимое для получения
снимка
    return "Frame from camera " + CameraID;
  }
}
```

Для отримання складеного зображення ми повинні викликати метод **GetNextFrame** у кожної з чотирьох об'єктів-камер. Припускаючи, що для цього потрібно більша кількість операцій введення/виведення, можна в 4 рази збільшити частоту кадрів за допомогою паралелізму - навіть на одноядерному ПК. PLINQ дозволяє добитися цього з мінімальними зусиллями з боку розробника:

```
Camera[] cameras = Enumerable.Range(0, 4) //Создаем 4 объекта камеры.
  .Select(i => new Camera(i))
  .ToArray();

while (true)
{
  string[] data = cameras
    .AsParallel().AsOrdered().WithDegreeOfParallelism (4)
    .Select(c => c.GetNextFrame()).ToArray();

  Console.WriteLine(string.Join(", ", data)); // Отображаем данные...
}
```

Метод GetNextFrame блокує хід виконання програми, тому ми викликаємо **WithDegreeOfParallelism** для отримання необхідного рівня паралелізму. У нашому прикладі блокування відбувається при виклику методу Sleep; у реальному житті блокування відбуватиметься через те, що отримання зображення з камери вимагає скоріше інтенсивного введення/виведення, ніж інтенсивного завантаження процесора.

Виклик методу AsOrdered гарантує, що зображення будуть відображені в потрібній послідовності. Оскільки в послідовності всього чотири елементи, це не зробить ніякого впливу на продуктивність.

Ще один приклад застосування блокуючих функцій.

Приклад 4.14.

В цьому прикладі одночасно виконується перевірка наявності шести сайтів.

```
from site in new[]
{
  "www.albahari.com",
  "www.linqpad.net",
  "www.oreilly.com",
  "www.google.com",
  "www.takeonit.com",
  "stackoverflow.com"
}
.AsParallel().WithDegreeOfParallelism(6)
let p = new Ping().Send (site)
select new
{
  site,
  Result = p.Status,
  Time = p.RoundtripTime
}
```

Виклик методу **WithDegreeOfParallelism** примушує PLINQ запускати вказану кількість завдань одночасно. Це необхідно, коли ви викликаєте блокуючі функції, такі як **Ping.Send**, оскільки інакше PLINQ припускає, що запит виконує завдання з високим навантаженням на центральний процесор, і створює завдання відповідним чином. Наприклад, на двоядерному комп'ютері PLINQ за замовчанням може запускати тільки два завдання одночасно, що в даному випадку є небажаним.

7. Приклад. Перебір каталогів з файлами за допомогою plinq

http://msdn.microsoft.com/ru-ru/library/ff 462679.aspx

У прикладі показано два прості способи паралельного виконання операцій в каталогах з файлами. Перший запит використовує метод GetFiles для заповнення масиву імен файлів в каталозі і усіх підкаталогах. Цей метод не повертає результат, поки увесь масив не буде заповнений, тому він може викликати затримку на початку операції. Проте після заповнення масиву PLINQ може дуже швидко виконувати паралельну обробку.

Другий запит використовує статичні методи EnumerateDirectories і EnumerateFiles, які відразу ж починають повертати результати. Завдяки такому підходу швидкість обробки може бути вище при переборі по великих деревах каталогів, хоча час обробки порівняно з першим прикладом може залежати від багатьох чинників.

У наступному прикладі показаний метод перебору в каталогах з файлами в простих сценаріях, коли є доступ до усіх каталогів в дереві, файли не дуже великого розміру і час доступу не має значення. Цей підхід має на увазі затримку на початку операції, поки створюється масив імен файлів.

```
struct FileResult
{
    public string Text;
    public string FileName;
}
// Use Directory.GetFiles to get the source sequence of file names.
public static void FileIteration_1(string path)
{
    var sw = Stopwatch.StartNew();
    int count = 0;
    string[] files = null;
    try
    {
        files = Directory.GetFiles(path, "*.*", SearchOption.AllDirectories);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine("You do not have permission to access one or more folders
in this directory tree.");
        return;
    }

    catch (FileNotFoundException)
    {
```

```
        Console.WriteLine("The specified directory {0} was not found.", path);
    }

    var fileContents = from file in files.AsParallel()
            let extension = Path.GetExtension(file)
            where extension == ".txt" || extension == ".htm"
            let text = File.ReadAllText(file)
            select new FileResult { Text = text , FileName = file }; //Or
ReadAllBytes, ReadAllLines, etc.

    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine(Path.GetFileName(item.FileName) + ":" +
item.Text.Length);
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle((ex) =>
            {
                if (ex is UnauthorizedAccessException)
                {
                   Console.WriteLine(ex.Message);
                   return true;
                }
                return false;
            });
    }

    Console.WriteLine("FileIteration_1 processed {0} files in {1} milliseconds",
count, sw.ElapsedMilliseconds);
    }
```

У наступному прикладі показаний метод перебору в каталогах з файлами в простих сценаріях, коли є доступ до усіх каталогів в дереві, файли не дуже великого розміру і час доступу не має значення. При такому підході результати з'являються швидше, ніж у попередньому прикладі.

```
struct FileResult
{
    public string Text;
    public string FileName;
}

// Use Directory.EnumerateDirectories and EnumerateFiles to get the source sequence
of file names.
public static void FileIteration_2(string path) //225512 ms
{
    var count = 0;
    var sw = Stopwatch.StartNew();
    var fileNames = from dir in Directory.EnumerateFiles(path, "*.*",
SearchOption.AllDirectories)
                    select dir;
```

```
    var fileContents = from file in fileNames.AsParallel() // Use AsOrdered to
preserve source ordering
                       let extension = Path.GetExtension(file)
                       where extension == ".txt" || extension == ".htm"
                       let Text = File.ReadAllText(file)
                       select new { Text, FileName = file }; //Or ReadAllBytes,
ReadAllLines, etc.
    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine(Path.GetFileName(item.FileName) + ":" +
item.Text.Length);
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle((ex) =>
            {
                if (ex is UnauthorizedAccessException)
                {
                   Console.WriteLine(ex.Message);
                   return true;
                }
                return false;
            });
    }

    Console.WriteLine("FileIteration_2 processed {0} files in {1} milliseconds",
count, sw.ElapsedMilliseconds);
}
```

8. Ефективність використання PLINQ

Основне призначення мови PLINQ - це збільшення швидкості виконання запитів LINQ to Objects за допомогою паралельного виконання делегатів запиту на багатоядерних комп'ютерах. Мова PLINQ має кращу продуктивність, коли обробка кожного елементу в початковій колекції є незалежною, без стану спільного доступу, який пов'язаний з окремими делегатами. Такі операції є спільними в мовах LINQ to Objects і PLINQ, і часто називаються *абсолютно паралельними*, оскільки вони легко надаються для планування в декількох потоках. Проте не усі запити повністю складаються з таких паралельних операцій. В більшості випадків запит містить деякі оператори, які не можна розпаралелити, або вони уповільнюють паралельне виконання. Навіть для запитів, які повністю є паралельними, мова PLINQ повинна як і раніше розділяти джерела даних і планувати роботу з потоками, а також, як правило, виконувати злиття результатів при завершенні запиту. Усі такі операції збільшують обчислювальну вартість паралелізму. Такі витрати на додавання паралелізму називаються витратами. Метою досягнення оптимальної продуктивності запиту PLINQ є максимізація абсолютно паралельних частин і мінімізація частин, які вимагають витрат. Далі перелічені чинники, які впливають на продуктивність запитів PLINQ.

1. Обчислювальна вартість загальної роботи

Для збільшення швидкості запит PLINQ повинен містити достатній об'єм паралельних операцій для компенсації витрат. Робота може бути виражена як обчислювальна вартість кожного делегата, помножена на кількість елементів в початковій колекції. Припустимо, що операція може бути зроблена паралельною. Це потребує більше обчислювальних ресурсів і дасть можливість для збільшення швидкості. Наприклад, якщо функція вимагає для виконання однієї мілісекунди, послідовний запит з більше, ніж 1000 елементами потребує для виконання такої операції однієї секунди, в той час, як паралельний запит на комп'ютері з чотирма ядрами може бути виконаний за 250 мілісекунд. Це дає прискорення на 750 мілісекунд. Якщо функція вимагає однієї секунди на виконання кожного елементу, то прискорення складе 750 секунд. Якщо делегат вимагає великих обчислювальних витрат, PLINQ може запропонувати значне прискорення вже при декількох елементах в початковій колекції. Навпаки, невеликі колекції з простими делегатами в загальному випадку не доцільно розпаралелювати за допомогою PLINQ.

У наступному прикладі запит queryA є потенційно хорошим кандидатом для PLINQ, якщо його функція Select містить великий об'єм роботи. Запит queryB не є таким кандидатом, оскільки в інструкції Select не міститься великий об'єм роботи і витрати паралелізму компенсують велику частину або усе прискорення.

```
var queryA = from num in numberList.AsParallel()
             select ExpensiveFunction(num); //good for PLINQ

var queryB = from num in numberList.AsParallel()
             where num % 2 > 0
             select num; //not as good for PLINQ
```

2. Кількість логічних ядер в системі (міра паралелізму).

Цей пункт є очевидним наслідком, що витікає з попереднього розділу. Абсолютно паралельні запити виконуються швидше на комп'ютерах, у яких велика кількість ядер, оскільки робота може бути розділена серед великої кількості паралельних потоків.Загальне значення прискорення залежить від відсотка від загального об'єму роботи, яка може бути зроблена паралельно. Проте не треба думати, що усі запити виконуються в два рази швидше на комп'ютерах з вісьмома ядрами, ніж на комп'ютері з чотирма ядрами. При налаштуванні запитів на оптимальну продуктивність важливо оцінювати фактичні результати на комп'ютерах з різною кількістю ядер.

3. Кількість і тип операцій.

Для ситуацій, в яких необхідно підтримувати порядок елементів в початковій послідовності, призначений оператор AsOrdered. Існують витрати, які пов'язані з впорядковуванням, але вони як правило невеликі. Операції GroupBy і Join також призводять до витрат. Продуктивність мови PLINQ вища, коли дозволено довільний порядок обробки елементів в початковій колекції і передача їх наступному операторові у міру їх готовності.

4. Форма виконання запиту.

При збереженні результатів запиту за допомогою виклику ToArray або ToList, результати з усіх паралельних потоків мають бути об'єднані в одну структуру даних. Це призводить до неминучих обчислювальних витрат. Також при ітерації результатів за допомогою циклу foreach, результати з робочих потоків мають бути серіалізовані в потік-нумератор. Проте якщо необхідно виконати окремі дії на основі результатів з кожного потоку, для виконання цієї роботи в декількох потоках можна використовувати метод ForAll.

5. Тип варіантів злиття.

Мову PLINQ можна налаштувати на буферизацію вихідних даних, видача яких здійснюється у вигляді блоків або у вигляді відразу усього результуючого набору, після його створення, або налаштувати на потік окремих результатів у міру їх отримання. Перший варіант зменшує загальний час виконання, а останній призводить до зменшення затримки між отриманими елементами. Варіанти злиття не завжди чинять основний вплив на загальну продуктивність запиту. Вони можуть вплинути на сприйману продуктивність, оскільки вони управляють часом, протягом якого користувач повинен побачити результати.

Висновки

Мова PLINQ – це паралельна реалізація LINQ for Object. Вона дозволяє спростити паралельні алгоритми обчислень завдяки тому, що бере на себе всю роботу по розбиттю колекцій об’єктів на окремі потоки і злиття результатів. Наведені в цій лекції приклади демонструють деяки можливості реалізації паралельних запитів. Разом з цим, при внесенні паралелізму в програму слід враховувати чинники, які впливають на ефективність використання PLINQ.

Контрольні запитання і завдання для самостійного виконання

1. Яке призначення PLINQ?
2. Яке призначення ParallelEnumerable?
3. В якому класі реалізовані методи PLINQ?
4. Який метод виконує розпаралелювання запиту?
5. Який метод виконує сортування результату запиту?
6. Яким чином можна впливати на параметри злиття результатів запиту?
7. Як можна відмінити запит? В яких випадках це потрібно робити?
8. Що означає цей оператор:

```
CancellationTokenSource cs = new CancellationTokenSource();
```

9. Яке призначення методу WithDegreeOfParallelism() і коли слід використовувати цей метод?
10. Коли недоцільно використовувати в програмі методи мови PLINQ?