Паралельне програмування з використанням бібліотеки TPL та PLINQ

Лекція 3. Клас Parallel

Розпаралелювання задач методом Invoke (). Неявне створення паралельних задач. Паралелізм даних. Паралельні цикли: методи Parallel.For() і Parallel.ForEach().

Раніше ми говорили, що паралелізм в програму можна ввести двома основними способами. Це

1. Паралелізм задач. При цьому підході дві або більше операції виконуються паралельно. Прикладами такого паралелізму є багатопоточність, реалізована в класі **System.Thread** та багатозадачність, реалізована в бібліотеці TPL. Це класи **System.Task** та **System.Task.Factory.** Отже, паралелізмом задач є різновидом паралелізму, який досягався у минулому засобами класу **System.Thread.**

Все що ми вивчали до цих пір стосувалося паралелізму задач.

2. Паралелізм даних. При такому підході одна операція над сукупністю даних розбивається на два або більше паралельно виконуваних потоки, в кожному з яких обробляється частина даних (декомпозиція за даними). Так, якщо змінюється кожен елемент масиву, то, застосовуючи паралелізм даних, можна організувати паралельну обробку різних областей масиву в двох або більше потоках.

Не дивлячись на те, що паралелізм даних був завжди можливий і за допомогою класу Thread, побудова масштабованих рішень засобами цього класу вимагали чимало зусиль і часу. Використання бібліотеки TPL спрощує внесення такого паралелізму даних в програму.

В .Net Framework паралелізм даних реалізований за допомогою:

- статичного класу **Parallel**;
- **PLINQ** – паралельної реалізації мови структурованих запитів.

В цій лекції ми розглянемо засоби розпаралелювання даних, які надає клас **Parallel**.

Клас **Parallel** є статичним, і в ньому визначені методи **For(), ForEach ()** і **Invoke ().** У кожного з цих методів є різні форми. Зокрема, метод **For ()** виконує цикл розпаралелювання операцій в циклі for, а метод **ForEach () -** розпаралелювання операцій в циклі foreach, і обидва методи підтримують *паралелізм даних*. А метод **Invoke** () підтримує паралельне виконання двох або більше методів. Тобто, цей метод спрощує реалізацію *паралелізму задач*. Використання цих методів спрощує реалізацію на практиці поширених методик паралельного програмування, не удаючись до управління завданнями або потоками явним чином.

1. Розпаралелювання задач методом Invoke ()

Метод **Invoke (),** визначений в класі Parallel, дозволяє виконувати один або декілька методів, що вказуються у вигляді його аргументів. Він також масштабує виконання коду, використовуючи доступні процесори, якщо є така можливість. Нижче наведена проста форма його оголошення.

public static void Invoke(params Action[] actions)

Виконувані методи мають бути сумісні з описаним раніше делегатом Action. Нагадаємо, що делегат Action оголошується таким чином.

public delegate void Action()

Отже, кожен метод, що передається методу **Invoke ()** як аргумент, не повинен ні приймати параметрів, ні повертати значення. Завдяки тому, що параметр actions цього методу відноситься до типу params, виконувані методи можуть бути вказані у вигляді змінного списку аргументів. Для цієї мети можна також скористатися масивом об'єктів типу Action, але часто виявляється простіше вказати список аргументів.

Метод **Invoke ()** спочатку ініціює виконання, а потім чекає завершення усіх переданих йому методів. Це, зокрема, позбавляє від необхідності (та і не дозволяє) викликати метод Wait (). І хоча це не гарантує, що методи дійсно виконуватимуться паралельно, проте, саме таке їх виконання передбачається, якщо система підтримує декілька процесорів. Крім того, відсутня можливість вказати порядок виконання методів від першого і до останнього, і цей порядок може не бути таким самим, як і в списку аргументів.

У наведеному нижче прикладі програми демонструється застосування методу **Invoke()**. У цій програмі два методи MyMeth1() і MyMeth2() виконуються паралельно за допомогою виклику методу Invoke ().

Приклад 3.1.

// Застосування методу Parallel.Invoke() для паралельного виконання двох методів.

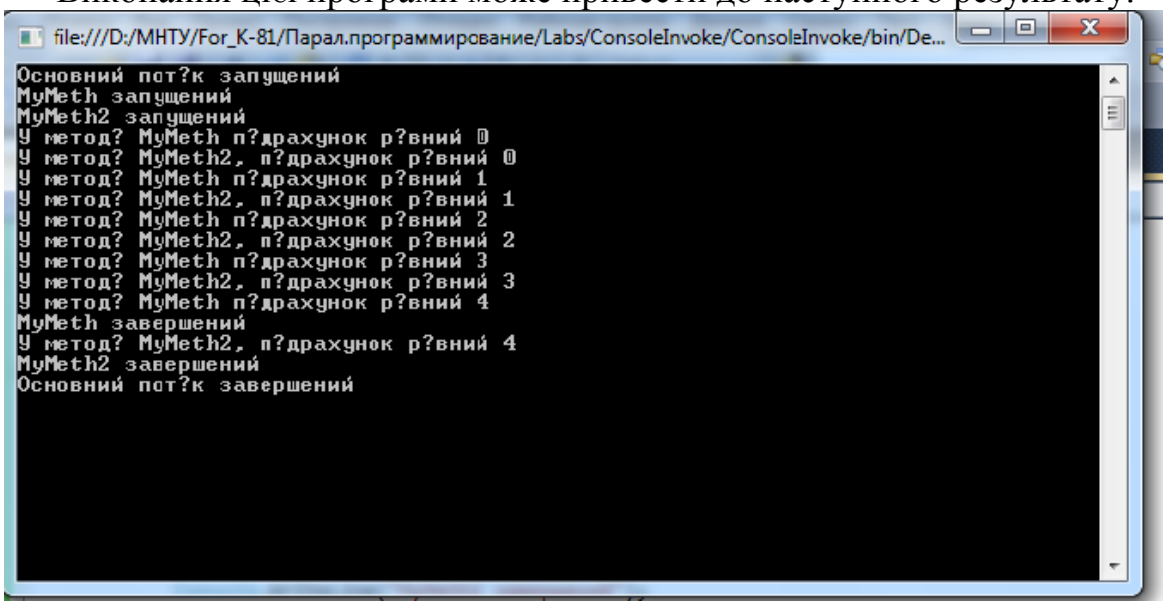
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace ConsoleInvoke
{
    class Program
    {
        // Метод, що виконується як задача
        static void MyMeth1()
        {
            Console.WriteLine("MyMeth1 запущений");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(100);
                Console.WriteLine("У методі MyMeth1 підрахунок рівний " + count);
            }
            Console.WriteLine("MyMeth1 завершений");
```

```
        }
        // Метод, що виконується як задача
        static void MyMeth2()
        {
            Console.WriteLine("MyMeth2 запущений");
            for (int count = 0; count < 5; count++)
            {
                Thread.Sleep(100);
                Console.WriteLine("У методі MyMeth2, підрахунок рівний " + count);
            }
            Console.WriteLine("MyMeth2 завершений");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущений");
            // Виконати паралельно два іменовані методи.
            Parallel.Invoke(MyMeth1, MyMeth2);
            Console.WriteLine("Основний потік завершений");
            Console.ReadKey();
        }
    }

}
```

Виконання цієї програми може привести до наступного результату.



У цьому прикладі слід звернути увагу на те, що виконання методу Main () призупиняється до тих пір, поки не завершиться виконання методу Invoke (). Отже, метод Main (), на відміну від методів MyMeth1 () і MyMeth2 (), не виконується паралельно. Тому застосовувати метод Invoke () таким способом не можна у тому випадку, якщо треба, щоб виконання головного потоку продовжувалося. Тобто, це не асинхронне виконання.

У наведеному вище прикладі використовувалися *іменовані методи*, але для виклику методу Invoke () ця умова не є обов'язковою. Нижче наведено перероблений варіант тієї самої програми, але з використанням лямбда-виразу.

Приклад 3.2.

// Застосувати метод Parallel.Invoke() для паралельного виконання двох методів.

//У цій версії програми застосовуються лямбда-вирази.

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

namespace ConsoleInvoke
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Основной поток запущен.");
            // Выполнить два анонимных метода, указываемых в лямбда-выражениях.
            Parallel.Invoke(() =>
            {
                Console.WriteLine("Выражение #1 запущено");
                for (int count = 0; count < 5; count++)
                {
                    Thread.Sleep(100);
                    Console.WriteLine("В выражении #1 подсчет равен " + count);
                }
                Console.WriteLine("Выражение #1 завершено");
            },
           () =>
            {
                Console.WriteLine("Выражение #2 запущено");
                for (int count = 0; count < 5; count++)
                {
                    Thread.Sleep(100);
                    Console.WriteLine("В выражении #2 подсчет равен " + count);
                }
                Console.WriteLine("Выражение #2 завершено");
            }
            );
            Console.WriteLine("Основной поток завершен.");
            Console.ReadKey();
        }

    }
}
```

Ця програма дає результат, схожий на результат виконання попередньої програми. В ній код методів розміщений в самому методі Parallel.Invoke.

Таким чином, статичний метод **Invoke** дозволяє розпаралелювати виконання блоків операторів. Ці блоки можуть бути оформлені у вигляді методів чи лямбда-виразів.

Часто в програмі існують такі послідовності операторів, для яких не має значення порядок виконання операторів усередині них. У таких випадках замість послідовного виконання операторів одного за іншим, можливо їх паралельне виконання, що дозволяє скоротити час рішення задачі.

Приклад 3.3. (http://msdn.microsoft.com/ru-ru/library/dd460705.aspx)

Застосування Parallel.Invoke. Для виклику паралельних методів використовуються лямбда-вирази.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Net;

namespace ConsoleInvoke
{
    class ParallelInvoke
    {
        static void Main()
        {
            // Retrieve Darwin's "Origin of the Species" from Gutenberg.org.
            string[] words =
CreateWordArray(@"http://www.gutenberg.org/files/2009/2009.txt");

            #region ParallelTasks
            // Perform three tasks in parallel on the source array
            Parallel.Invoke(() =>
                            {
                                Console.WriteLine("Begin first task...");
                                GetLongestWord(words);
                            },  // close first Action

                            () =>
                            {
                                Console.WriteLine("Begin second task...");
                                GetMostCommonWords(words);
                            }, //close second Action

                            () =>
                            {
                                Console.WriteLine("Begin third task...");
                                GetCountForWord(words, "species");
                            } //close third Action
                        ); //close parallel.invoke

            Console.WriteLine("Returned from Parallel.Invoke");
            #endregion

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        #region HelperMethods
        private static void GetCountForWord(string[] words, string term)
        {
            var findWord = from word in words
                           where word.ToUpper().Contains(term.ToUpper())
                           select word;

            Console.WriteLine(@"Task 3 -- The word ""{0}"" occurs {1} times.",
                term, findWord.Count());
        }

        private static void GetMostCommonWords(string[] words)
        {
```

```
            var frequencyOrder = from word in words
                                 where word.Length > 6
                                 group word by word into g
                                 orderby g.Count() descending
                                 select g.Key;

            var commonWords = frequencyOrder.Take(10);

            StringBuilder sb = new StringBuilder();
            sb.AppendLine("Task 2 -- The most common words are:");
            foreach (var v in commonWords)
            {
                sb.AppendLine("  " + v);
            }
            Console.WriteLine(sb.ToString());
        }

        private static string GetLongestWord(string[] words)
        {
            var longestWord = (from w in words
                               orderby w.Length descending
                               select w).First();

            Console.WriteLine("Task 1 -- The longest word is {0}", longestWord);
            return longestWord;
        }


        // An http request performed synchronously for simplicity.
        static string[] CreateWordArray(string uri)
        {
            Console.WriteLine("Retrieving from {0}", uri);

            // Download a web page the easy way.
            string s = new WebClient().DownloadString(uri);

            // Separate string into an array of words, removing some common punctuation.
            return s.Split(
                new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '_', '/' },
                StringSplitOptions.RemoveEmptyEntries);
        }
        #endregion
    }
}

/* Output (May vary on each execution):
        Retrieving from http://www.gutenberg.org/dirs/etext99/otoos610.txt
        Response stream received.
        Begin first task...
        Begin second task...
        Task 2 -- The most common words are:
          species
          selection
          varieties
          natural
          animals
          between
          different
          distinct
          several
          conditions

        Begin third task...
```

```
        Task 1 -- The longest word is characteristically
        Task 3 -- The word "species" occurs 1927 times.
        Returned from Parallel.Invoke
        Press any key to exit
    */
```

2. Паралелізм даних. Паралельні цикли. Метод Parallel.For()

У TPL паралелізм даних підтримується за допомогою методів **For (), ForEach(),** які визначені в класі Parallel.

Розглянемо спочатку метод For (). Цей метод існує в декількох формах. Його розгляд ми почнемо з найпростішої форми, наведеної нижче:

public static ParallelLoopResult

For (int fromlnclusive, int toExclusive, Action<int> body)

де fromlnclusive означає початкове значення того, що відповідає змінній управління циклом; воно називається також ітераційним, або індексним значенням; a toExclusive - значення, на одиницю більше за кінцеве. На кожному кроці циклу змінна управління циклом збільшується на одиницю. Отже, цикл поступово просувається від початкового значення fromlnclusive до кінцевого значення toExclusive мінус одиниця. Циклічно виконуваний код вказується методом, що передається через параметр body. Цей метод має бути сумісний з делегатом Action<int>, оголошуваним таким чином.

public delegate void Action<int T>(T obj)

для методу For () узагальнений параметр T має бути, звичайно, типу int.

Значення, що передається через параметр obj, буде наступним значенням змінної управління циклом. А метод, що передається через параметр body, може бути іменованим або анонімним. Метод For () повертає екземпляр об'єкту типу ParallelLoopResult, що описує стан завершення циклу. Для простих циклів цим значенням можна нехтувати.

Головна особливість методу For () полягає в тому, що він дозволяє, коли така можливість є, розпаралелювати виконання коду в циклі. А це, у свою чергу, може привести до підвищення продуктивності. Наприклад, процес перетворення масиву в циклі може бути розділений на частини так, щоб різні частини масиву перетворювалися одночасно.

Розглянемо наступний послідовний цикл на C#:

```
for (int i = 0; i < N; i++)
            {
               results[i] = Compute(i);
            }
```

З допомогою Parallel.For можна розпаралелити цей цикл таким чином:

```
  // паралельний цикл
            Parallel.For(0, N, delegate(int i)
```

```
            {
                results[i] = Compute(i);
                Console.WriteLine("results[i] = " + results[i]);
            });
```

Для спрощення коду можна використовувати синтаксис лямбда-виразів:

```
 // паралельний цикл через лямбда-вираз
            Parallel.For(0, N, i =>
            {
                results[i] = Compute(i);
                Console.WriteLine("results[i] = " + results[i]);
            });
```

Приклад 10.4. Повний код програми

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

namespace ParaellFor
{
    class Program
    {
        static long Compute(int i)
        {
         return i*i;
        }

        static void Main(string[] args)
        {
            int N = 10;
            long[] results = new long[N];
            //лінійний цикл
            for (int i = 0; i < N; i++)
            {
               results[i] = Compute(i);
            }
           // паралельний цикл
            Parallel.For(0, N, delegate(int i)
            {
                results[i] = Compute(i);
  //            Console.WriteLine("results[i] = " + results[i]);
            });

            // паралельний цикл через лямбда-вираз
            Parallel.For(0, N, i =>
            {
                results[i] = Compute(i);
                Console.WriteLine("results[i] = " + results[i]);
            });

             Console.ReadKey();
        }
    }
}
```

Тепер ітерації цього циклу можуть бути виконані паралельно.

Приклад 10.5. Паралельна генерація відкритих/закритих ключів

Класи для роботи з ключами знаходяться у просторі імен System.Security.Cryptography.

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

```
namespace Lab3_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();
            }

        }
    }
}
```

Розглянемо ще приклад використання циклу for.

Приклад 3.6.

В програмі створюється масив data, що складається з 10000 цілих чисел. Потім викликається метод For (), якому як "тіло" циклу передається метод MyTransform(). Цей метод складається з ряду операторів, що виконують довільні перетворення в масиві data. Його призначення - зімітувати конкретну операцію. Виконувана операція має бути нетривіальною, щоб паралелізм даних приніс якийсь позитивний ефект. Інакше послідовне виконання циклу може завершитися швидше.

```
 using System;
    using System.Collections.Generic;
    using System.Collections.Concurrent;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
namespace Parallel_For
{
    class Program
    {
        static int[] data;
        static void MyTransform(int i) {
        data[i] = data[i] / 10;
        if(data[i] < 10000) data[i] = 0;
        if(data[i] > 10000 & data[i] < 20000) data[i]=100;
        if (data[i] > 20000 & data[i] < 30000) data[i]=200;
        if(data[i] > 30000) data[i] = 300;
}

        static void Main()
        {
```

```
            Console.WriteLine("Основной поток запущен.");
            data = new int[10000];
            // Инициализировать данные в обычном цикле for.
            for (int i = 0; i < data.Length; i++) data[i] = i;
            // Распараллелить цикл методом For().
            Parallel.For(0, data.Length, MyTransform);
            Console.WriteLine("Основной поток завершен.");
            Console.ReadKey();
        }
    }
}
```

Ця програма складається з двох циклів. У першому, стандартному, циклі for ініціалізувався масив data. А в другому циклі, що виконується паралельно методом For (), над кожним елементом масиву data виконуються перетворення. Метод For () автоматично розбиває виклики методу MyTransform () на частини для паралельної обробки окремих порцій даних, що зберігаються в масиві.

Слід, проте, мати на увазі, що далеко не всі цикли можуть виконуватися ефективно, коли вони розпаралелюються. Як правило, дрібні цикли, а також цикли, що складаються з дуже простих операцій, виконуються швидше послідовним способом, ніж паралельним. Саме тому цикл for ініціалізації масиву даних не розпаралелюється методом For () в програмі, що розглядається тут. Розпаралелювання дрібних і дуже простих циклів може виявитися неефективним тому, що час, що вимагається для організації паралельних завдань, а також час, що витрачається на перемикання контексту, перевищує час, що економиться завдяки паралелізму.

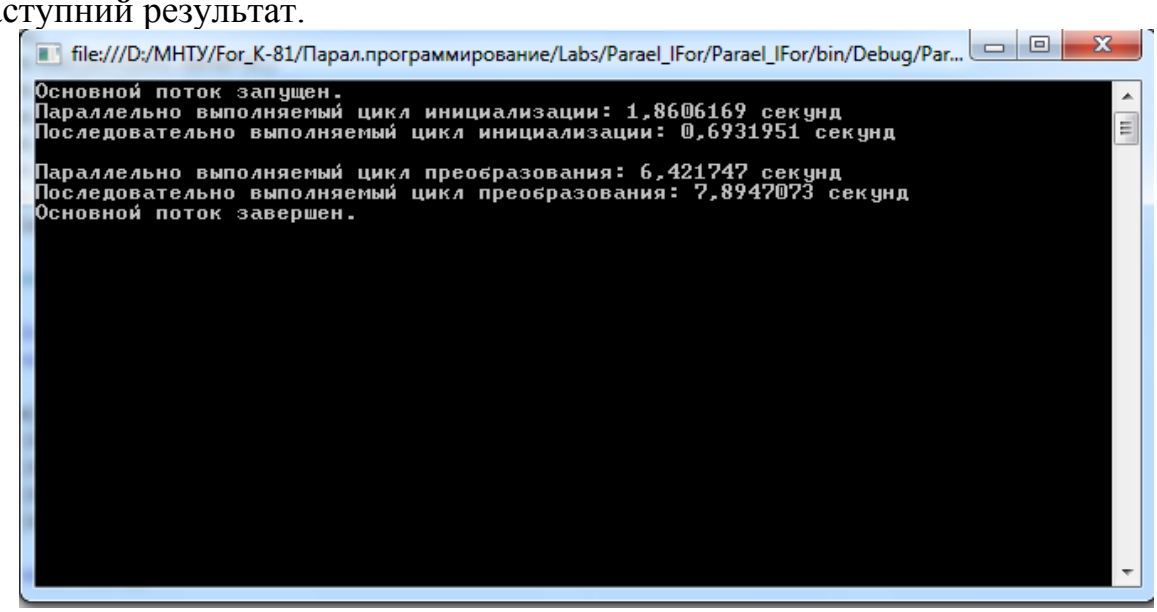
Це демонструється в наступному прикладі.

Приклад 3.7.

```
    using System;
    using System.Collections.Generic;
    using System.Collections.Concurrent;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
namespace Parallel_For
{
    class Program
    {
        static int[] data;
        static void MyTransform(int i)
        {
            data[i] = data[i] / 10;
            if (data[i] < 1000) data[i] = 0;
            if (data[i] > 1000 & data[i] < 2000) data[i] = 100;
            if (data[i] > 2000 & data[i] < 3000) data[i] = 200;
            if (data[i] > 3000) data[i] = 300;
        }
        static void Main()
        {
            Console.WriteLine("Основной поток запущен.");
            // Create экземпляр объекта типа Stopwatch
            // для хранения времени выполнения цикла.
            Stopwatch sw = new Stopwatch();
            data = new int[100000000];
```

```
            // Инициализировать данные,
            sw.Start();
            // Параллельный вариант инициализации массива в цикле.
            Parallel.For(0, data.Length, (i) => data[i] = i);
            sw.Stop();
            Console.WriteLine("Параллельно выполняемый цикл инициализации: " +
            "{0} секунд", sw.Elapsed.TotalSeconds);
            sw.Reset();
            sw.Start();
            // Последовательный вариант инициализации массива в цикле,
            for (int i = 0; i < data.Length; i++) data[i] = i;
            sw.Stop();
            Console.WriteLine("Последовательно выполняемый цикл инициализации: " +
            "{0} секунд", sw.Elapsed.TotalSeconds);
            Console.WriteLine();
            // Выполнить преобразования,
            sw.Start();
            // Параллельный вариант преобразования данных в цикле.
            Parallel.For(0, data.Length, MyTransform);
            sw.Stop();
            Console.WriteLine("Параллельно выполняемый цикл преобразования: " +
            "{0} секунд", sw.Elapsed.TotalSeconds);
            sw.Reset();
            sw.Start();
            // Последовательный вариант преобразования данных в цикле.
            for (int i = 0; i < data.Length; i++) MyTransform(i);
            sw.Stop();
            Console.WriteLine("Последовательно выполняемый цикл преобразования: " +
            "{0} секунд", sw.Elapsed.TotalSeconds);
            Console.WriteLine("Основной поток завершен.");
            Console.ReadKey();
        }
    }
}
```

При виконанні цієї програми на двоядерному комп'ютері виходить наступний результат.



Передусім, зверніть увагу на те, що паралельний варіант циклу ініціалізації масиву даних виконується приблизно в три рази повільніше, ніж послідовний. Це тому, що в даному випадку на операцію ініціалізації

витрачається так мало часу, що витрати на додатково організовуване розпаралелювання перевищують економію, яку воно дає.

А от паралельний варіант циклу перетворення даних виконується швидше, ніж послідовний. В цьому випадку економія від розпаралелювання покриває витрати на його додаткову організацію.

В цій програмі для обчислення часу виконання циклу використовується клас Stopwatch. Цей клас знаходиться в просторі імен System.Diagnostics. Для того, щоб скористатися ним, досить створити екземпляр його об'єкту, а потім викликати метод Start(), який розпочинає обчислення часу, і далі - метод Stop(), який завершує обчислення часу. А за допомогою методу Reset () відлік часу скидається в початковий стан. Тривалість виконання можна отримати різними способами. У програмі, що розглядається тут, для цієї мети використана властивість Elapsed, що повертає об'єкт типу TimeSpan. За допомогою цього об'єкту і властивості TotalSeconds час відображується в секундах, включаючи і частки секунди. Як показує приклад програми, що розглядається тут, клас Stopwatch виявляється дуже корисним при розробці паралельно виконуваного коду.

3. Паралелізм даних. Метод Parallel.ForEach()

Оператор циклу **foreach** використовується для перебору елементів в спеціальним чином організованій групі даних, яка називається колекцією. Масив є саме такою групою (колекцією). Зручність цього вигляду циклу полягає в тому, що нам не потрібно визначати кількість елементів в групі і виконувати їх перебір за індексом: ми просто вказуємо на необхідність перебрати всі елементи групи. Синтаксис оператора:

```
foreach ( тип ім'я in вираз ) тіло_циклу
```

Ім'я задає локальну по відношенню до циклу змінну, яка по черзі набуватиме всіх значень з масиву виразу (у якості виразу найчастіше застосовується ім'я масиву або іншої колекції даних). У простому або складеному операторі, що є тілом циклу, виконуються дії зі змінною циклу. Тип змінної повинен відповідати типу елементу масиву.

Використовуючи метод ForEach (), можна створити паралельний варіант циклу foreach. Існує декілька форм методу ForEach (). Нижче наведена проста форма його оголошення:

public static ParallelLoopResult
ForEach<TSource>(IEnumerable<TSource> source,
Action<TSource> body)

де source означає колекцію даних, що обробляються в циклі, a body - метод, який виконуватиметься на кожному кроці циклу. Метод, що передається через параметр body, приймає у якості аргумента значення або посилання на кожен оброблюваний в циклі елемент масиву, але не його індекс. А у результаті повертаються відомості про стан циклу.

Аналогічно методу For (), паралельне виконання циклу методом ForEach () можна зупинити, викликавши метод Break () для екземпляра об'єкту типу ParallelLoopState, що передається через параметр body, за умови, що використовується наведена нижче форма методу ForEach ().

public static ParallelLoopResult
ForEach<TSource>(IEnumerable<TSource> source,
ActionKTSource, ParallelLoopState> body)

Розпаралелювання циклів foreach виконується аналогічним чином. Розглянемо цикл на C#:

```
foreach(MyClass c in data)
{
  Compute(c);
}
```

З допомогою Parallel.ForEach можна розпаралелити цей цикл таким чином:

```
Parallel.Foreach(data, delegate(MyClass c))
{
  Compute(c);
}
```

А спрощений код з використанням лямбда-виразів виглядає так:

```
Parallel.Foreach(data, c =>
{
  Compute(c);
});
```

Цикл ForEach працює так само як цикл For. Початкова колекція розділяється, і робота планується для декількох потоків залежно від системного середовища. Чим більше процесорів в системі, тим швидше виконується паралельний метод. У разі деяких початкових колекцій послідовний цикл може виконуватися швидше залежно від розміру джерела і типу виконуваних робіт.

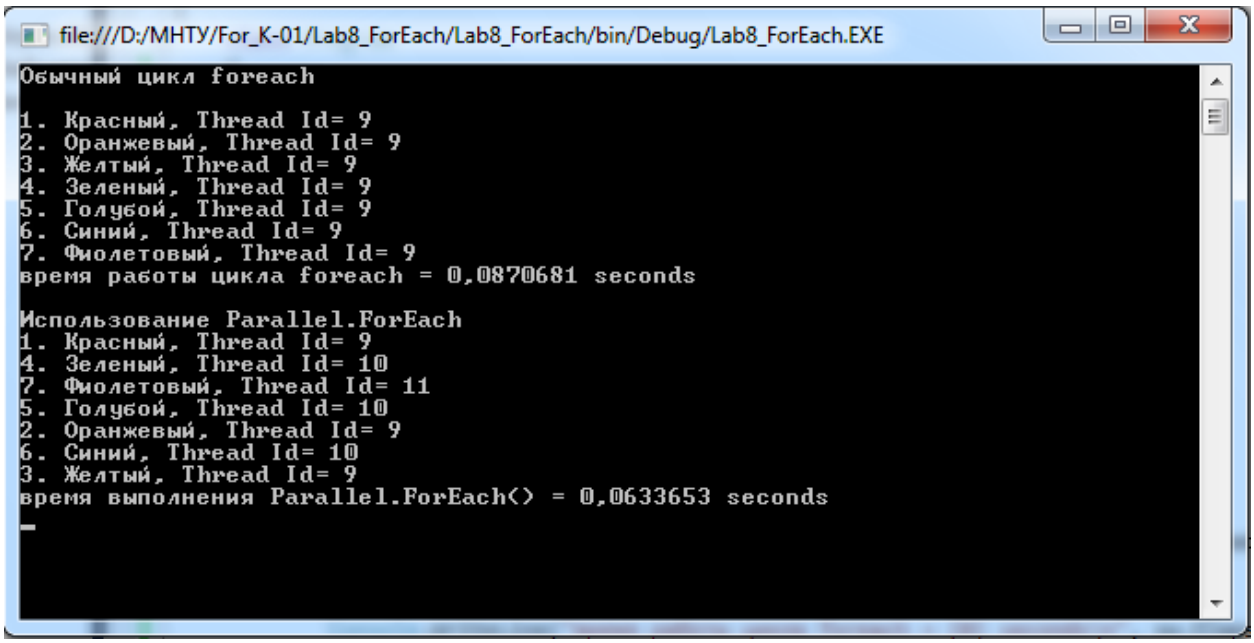
Приклад 3.8. Використання Parallel.ForEach

В цьому прикладі створюється масив кольорів спектру. Далі демонструється використання звичайного циклу foreach і паралельного для виведення на консоль значень кольорів і поточних потоків виконання. Також обчислюється час виконання.

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

```
using System.Diagnostics;

namespace Lab3_ForEach
{
    class Program
    {
        static void Main(string[] args)
        {
         string[] colors = {
             "1. Красный",
             "2. Оранжевый",
             "3. Желтый",
             "4. Зеленый",
             "5. Голубой",
             "6. Синий",
             "7. Фиолетовый" };
            Console.WriteLine("Обычный цикл foreach \n");
            //запуск stopwatch для цикла foreach
            var sw = Stopwatch.StartNew();
            foreach (string color in colors)
            {
                Console.WriteLine("{0}, Thread Id= {1}", color,
Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(10);
            }
            Console.WriteLine("время работы цикла foreach = {0} seconds\n",
sw.Elapsed.TotalSeconds);
            //парал.версия цикла
            Console.WriteLine("Использование Parallel.ForEach");
            //запуск stopwatch для "Parallel.ForEach"
            sw = Stopwatch.StartNew();
            Parallel.ForEach(colors, color =>
            {
                Console.WriteLine("{0}, Thread Id= {1}", color,
Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(10);
            }
            );
            Console.WriteLine("время выполнения Parallel.ForEach() = {0} seconds",
sw.Elapsed.TotalSeconds);
            Console.ReadKey();
        }
    }
}
```



Приклад 3.9 Робота з одновимірним масивом з використанням циклу foreach. Послідовна версія.

В цьому прикладі спочатку на консоль виводяться всі елементи масиву. В другому циклі обчислюється сума від'ємних елементів і їхня кількість. В третьому циклі шукається максимальний елемент.

```
using System;
namespace Lab3_9
{
    class Program
    {
        static void Main()
        {
           int[] a = { 3, 12, 5, -9, 8, -4 };
            Console.WriteLine( "Початковий  масив:" );
            foreach ( int elem in a )
                Console.Write( "\t" + elem );
            Console.WriteLine();

            long sum = 0;              // сума від'ємних елементів
            int  num = 0;              // кількість від'ємних елементів
            foreach ( int elem in a )
                if ( elem < 0 )
                {
                    sum += elem;
                    ++num;
                }
            Console.WriteLine( "sum = " + sum );
            Console.WriteLine( "num = " + num );

            int max = a[0];            // максимальний елемент
            foreach ( int elem in a )
                if ( elem > max ) max = elem;

            Console.WriteLine( "max = " + max );

            Console.ReadKey();
        }
    }
}
```

Приклад 3.9_1 Робота з одновимірним масивом з використанням циклу `Parallel.ForEach`. Паралельна версія.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;

namespace Lab3.9_1
{
    class Program
    {
        static void Main()
        {
           int[] mas = { 3, 12, 5, -9, 8, -4 };
            Console.WriteLine( "Початковий  масив:" );
     /*       foreach ( int elem in mas )
              Console.Write( "\t" + elem );
            Console.WriteLine();
*/
            Parallel.ForEach(mas, elem => {
                Console.Write( "\t" + elem );
                  });
            Console.WriteLine();

            long sum = 0;              // сума від'ємних елементів
            int  num = 0;              // кількість від'ємних елементів
 /*           foreach ( int elem in mas )
              if ( elem < 0 )
              {
                  sum += elem;
                  ++num;
              }
*/
            Parallel.ForEach(mas, elem =>
            {
                if (elem < 0)
                {
                    sum += elem;
                    ++num;
                }
            });
            Console.WriteLine( "sum = " + sum );
            Console.WriteLine( "num = " + num );

            int max = mas[0];            // максимальний елемент
      /*      foreach ( int elem in mas )
              if ( elem > max ) max = elem;
*/
            Parallel.ForEach(mas, elem =>
            {
                 if ( elem > max ) max = elem;
            });
            Console.WriteLine( "max = " + max );

            Console.ReadKey();
        }
    }
}
```

Висновки

В цій лекції ми розглянули нові можливості внесення паралелізму в програму, реалізовані в класі Parallel. Метод Parallel.Invoke () забезпечує розпаралелювання незалежних ділянок коду програми. При його виконанні основний потік блокується. Тому застосовувати метод Invoke () таким способом не можна у тому випадку, якщо треба, щоб виконання головного потоку продовжувалося. Тобто, це не асинхронне виконання. Використання Parallel.Invoke () значно спрощує реалізацію паралелізму задач, ніж багатопоточність.

Іншим способом внесення паралелізму в програму є паралелізм даних. Для розпаралелювання циклів в класі **Parallel** визначено два методи: **Parallel.For()** та **Parallel.ForEach**(). Вони значно спрощують реалізацію паралельних алгоритмів, не потребують розбиття великих масивів на сегменти, обчислення часткових результатів в зведення їх в одне рішення.

Контрольні запитання і завдання для самостійного виконання

1. Які є види паралелізму і яка між ними різниця?
2. Який клас реалізує паралелізм даних?
3. Яка різниця між синхронним і асинхронним паралельним виконанням і за допомогою яких класів воно реалізується?
4. Що дає використання лямбда виразу замість використання іменованих методів?
5. Коли доцільно використовувати метод **Parallel.Invoke(),** а коли краще **TaskFactory()**?
6. Яке призначення методу **Parallel.For()**?
7. Яке призначення методу **Parallel.ForEach()**?
8. Коли доцільно розпаралелювати обчислення у циклах, а коли ні?
9. Яке призначення класу Stopwatch?
10. Яким чином можна зупинити виконання циклів в методах Parallel.For() і Parallel.ForEach()?