**Parallel LINQ (PLINQ) in C# with Examples**

##### **Parallel LINQ (PLINQ) in C#**

If we have a collection and if we want to use parallelism to process it, we have the option of using Parallel LINQ or PLINQ. Parallel LINQ (PLINQ) is basically the same as we have in LINQ. But with parallel functionality, we can define the maximum degree of parallelism and we can also use a cancellation token to cancel the operation and so on.

One difference LINQ has from parallel for each is that LINQ has a nice and compact syntax for performing operations on collections. To be able to process a sequence with LINQ, we just use the AsParallel method.

##### **Example to Understand Parallel LINQ in C#:**

Let us understand this with an example. In the below example we are creating a collection of integer numbers from 1 to 20 using Enumerable.Range method. Then using the LINQ method, we are filtering the list of even numbers from the numbers collection. In the below example, we are not using PLINQ, we are simply using LINQ.

using System;

using System.Linq;

namespace ParallelLINQDemo

{

class Program

{

static void Main(string[] args)

{

//Creating a Collection of integer numbers

var numbers = Enumerable.Range(1, 20);

//Fetching the List of Even Numbers using LINQ

var evenNumbers = numbers.Where(x => x % 2 == 0).ToList();

Console.WriteLine("Even Numbers Between 1 and 20");

foreach (var number in evenNumbers)

{

Console.WriteLine(number);

}

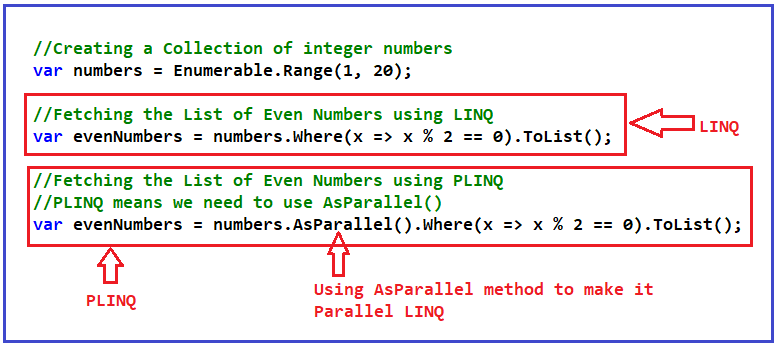
Console.ReadKey();

}

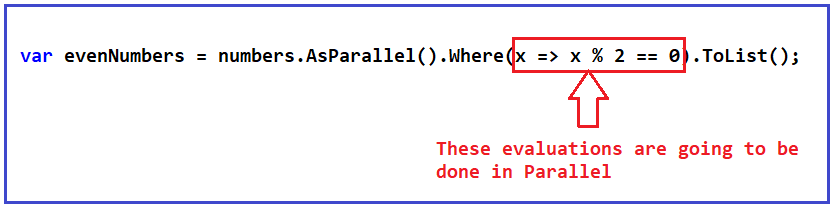
}

}

Now, let us see how to use PLINQ in C# with the same example. As discussed earlier we need to use the AsParallel method. For a better understanding, please have a look at the below image which shows both LINQ and PLINQ syntaxes to get the even numbers from the numbers collection.



So, this is as simple as it is. The following code uses parallelism. Now, the evaluations (i.e. **x => x % 2 == 0**) are going to be done in parallel.



Now, let us iterate over the evenNumbers collection and see the output. The following is the complete code example.

using System;

using System.Linq;

namespace ParallelLINQDemo

{

class Program

{

static void Main(string[] args)

{

//Creating a Collection of integer numbers

var numbers = Enumerable.Range(1, 20);

//Fetching the List of Even Numbers using LINQ

//var evenNumbers = numbers.Where(x => x % 2 == 0).ToList();

//Fetching the List of Even Numbers using PLINQ

//PLINQ means we need to use AsParallel()

var evenNumbers = numbers.AsParallel().Where(x => x % 2 == 0).ToList();

Console.WriteLine("Even Numbers Between 1 and 20");

foreach (var number in evenNumbers)

{

Console.WriteLine(number);

}

Console.ReadKey();

}

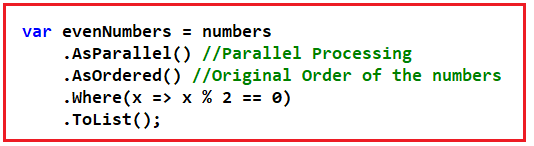
}

}

You can observe the order of the numbers. They are in random order. This is because we have already seen in the past that when we use parallelism, we typically cannot control the order of the operations. Now, if you run the code multiple times, each time you might get a different order of the numbers.

##### **How to Maintain the Original Order in PLINQ?**

If you want the output to be in order, then you need to use AsOrdered method after AsParallel which means that after doing the operations in parallel, it will maintain the original order of the elements. For a better understanding, please have a look at the following image which shows how to use the AsOrdered method.



The order will be the original order in which the elements are stored in the numbers collections. The following is the complete code.

using System;

using System.Collections.Generic;

using System.Linq;

namespace ParallelLINQDemo

{

class Program

{

static void Main(string[] args)

{

//Creating a Collection of integer numbers

var numbers = Enumerable.Range(1, 20);

//Fetching the List of Even Numbers using PLINQ

//PLINQ means we need to use AsParallel()

var evenNumbers = numbers

.AsParallel() //Parallel Processing

.AsOrdered() //Original Order of the numbers

.Where(x => x % 2 == 0)

.ToList();

Console.WriteLine("Even Numbers Between 1 and 20");

foreach (var number in evenNumbers)

{

Console.WriteLine(number);

}

Console.ReadKey();

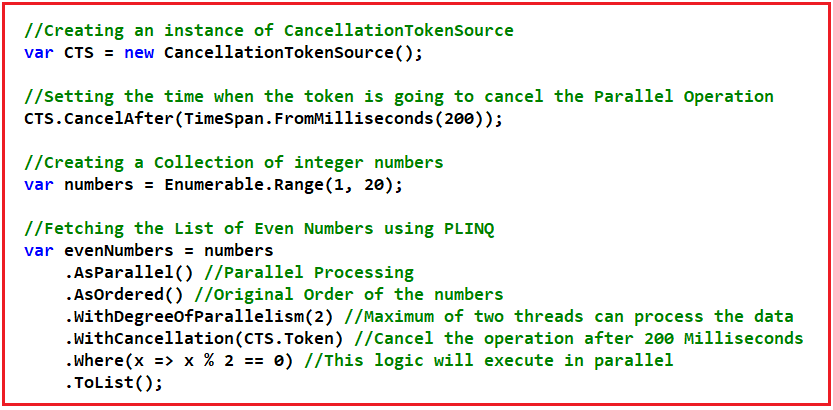
}

}

}

##### **Maximum Degree of Parallelism and Cancellation Token in PLINQ:**

As we said, we can have the same functionality here as a Parallel For Each. For example, you can define the maximum degree of parallelism. You can also define and pass a cancellation token that will cancel the execution of the PLINQ operation. For a better understanding, please have a look at the below image.



That means with Parallel LINQ, we can achieve the same functionality as a parallel for each with the difference is that we have this nice syntax that comes from PLINQ. The complete example code is given below.

using System;

using System.Linq;

using System.Threading;

namespace ParallelLINQDemo

{

class Program

{

static void Main(string[] args)

{

//Creating an instance of CancellationTokenSource

var CTS = new CancellationTokenSource();

//Setting the time when the token is going to cancel the Parallel Operation

CTS.CancelAfter(TimeSpan.FromMilliseconds(200));

//Creating a Collection of integer numbers

var numbers = Enumerable.Range(1, 20);

//Fetching the List of Even Numbers using PLINQ

var evenNumbers = numbers

.AsParallel() //Parallel Processing

.AsOrdered() //Original Order of the numbers

.WithDegreeOfParallelism(2) //Maximum of two threads can process the data

.WithCancellation(CTS.Token) //Cancel the operation after 200 Milliseconds

.Where(x => x % 2 == 0) //This logic will execute in parallel

.ToList();

Console.WriteLine("Even Numbers Between 1 and 20");

foreach (var number in evenNumbers)

{

Console.WriteLine(number);

}

Console.ReadKey();

}

}

}

##### **Doing Aggregates in PLINQ**

Something we can do is aggregate the elements of an enumeration. For example, we can add them all into a single value or we can calculate the average of the elements of a collection, again, producing a single value. Let us see an example where we will calculate the Sum, Max, Min, and Average of an enumeration using Parallel LINQ in C#.

using System;

using System.Linq;

namespace ParallelLINQDemo

{

class Program

{

static void Main()

{

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

//Sum, Min, Max and Average LINQ extension methods

Console.WriteLine("Sum, Min, Max and Average with LINQ");

var Sum = numbers.AsParallel().Sum();

var Min = numbers.AsParallel().Min();

var Max = numbers.AsParallel().Max();

var Average = numbers.AsParallel().Average();

Console.WriteLine($"Sum:{Sum}\nMin: {Min}\nMax: {Max}\nAverage:{Average}");

Console.ReadKey();

}

}

}

##### **Is really Parallel LINQ improving the Performance of an Application?**

Let us see an example using both LINQ and Parallel LINQ to do the same task and then see the performance benchmark. Please have a look at the below example. In the below example we are comparing the performance of LINQ and PLINQ Min, Max, and Average methods. The Min, Max, and Average methods are going to return a single scalar value or you can say aggregate value.

using System;

using System.Diagnostics;

using System.Linq;

namespace ParallelLINQDemo

{

class Program

{

static void Main()

{

var random = new Random();

int[] values = Enumerable.Range(1, 99999999)

.Select(x => random.Next(1, 1000))

.ToArray();

//Min, Max and Average LINQ extension methods

Console.WriteLine("Min, Max and Average with LINQ");

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

// var linqStart = DateTime.Now;

var linqMin = values.Min();

var linqMax = values.Max();

var linqAverage = values.Average();

stopwatch.Stop();

var linqTimeMS = stopwatch.ElapsedMilliseconds;

DisplayResults(linqMin, linqMax, linqAverage, linqTimeMS);

//Min, Max and Average PLINQ extension methods

Console.WriteLine("\nMin, Max and Average with PLINQ");

stopwatch.Restart();

var plinqMin = values.AsParallel().Min();

var plinqMax = values.AsParallel().Max();

var plinqAverage = values.AsParallel().Average();

stopwatch.Stop();

var plinqTimeMS = stopwatch.ElapsedMilliseconds;

DisplayResults(plinqMin, plinqMax, plinqAverage, plinqTimeMS);

Console.ReadKey();

}

static void DisplayResults(int min, int max, double average, double time)

{

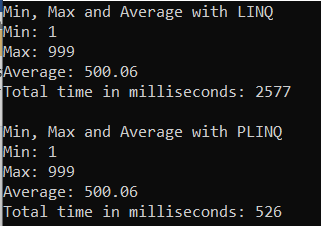
Console.WriteLine($"Min: {min}\nMax: {max}\n" + $"Average: {average:F}\nTotal time in milliseconds: {time}");

}

}

}

###### **Output:**



##### **Summary of Parallel Programming:**

1. In this Parallel Programming Section, we saw that with parallelism, we can perform several actions at the same time on our computer. This helps with the speed of our program to solve certain problems.
2. We can use Task.WhenAll for IO-bound operations and a Parallel class for CPU-bound operations.
3. With Parallel For and Parallel Foreach, we can execute a loop in parallel where we cannot guarantee a defined order of executions.
4. We saw that it is not always convenient to use parallelism and this depends on the amount of work to be done. If it is very little work, the cost of parallelism is greater than not using it.
5. We can cancel operations in parallel and we can also define the number of threads to use by defining the maximum degree of parallelism.
6. We saw that atomic methods guarantee that there is no data corruption where multiple threads invoke the method concurrently.
7. A race condition is where multiple threads try to modify a variable at the same time causing unpredictable results.
8. Interlocked is a class that allows us to perform certain operations in an atomic way, such as adding variables which help us avoid race conditions.
9. Look allows us to create a block of code that can only be accessed by one thread at a time. In this way, we can avoid race conditions between multiple operations.
10. PLINQ allows us to use link syntax to process collections in parallel.