**Task Parallel Library in C#**

**Task Parallel Library in C# with Examples (TPL in C#)**

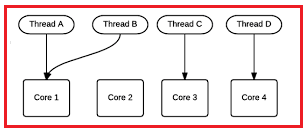
In this article, I am going to give you an overview of the **Task Parallel Library in C#**with Examples. Please read our previous section articles where we discussed [**Task-Based Asynchronous Programming**](https://dotnettutorials.net/lesson/asynchronous-programming-in-csharp/) in C#. The Task Parallel Library is also referred to as TPL in C#**.**At the end of this article, you will understand What is Task Parallel Library is and why do we need it in C# applications.

**Why do we need Task Parallel Library in C#?**

We can’t expect our sequential program to run faster on the new processors as we know the processor technology advances means the focus is on Multicore-processors. Today’s desktop typically has 4 cores but the latest experimental multi-core chips have up to 1000 cores.

So in simple words, we can say that the multicore processor machines are now becoming standard and the aim is to improve the performance by running a program on multiple processors in parallel. So by considering the above scenario, the .NET Framework 4 introduces **Task Parallel Library (TPL)** that makes it easier for developers to write parallel programs that target multi-core machines (automatically use multiple processors) which improves the performance.

Using the Task Parallel Library (TPL), we can express the parallelism in the existing sequential code, which means we can express the code as the Parallel task, which will be run concurrently on all the available processors.



**What is Parallel Programming in C#?**

Parallel Programming is a type of programming in which many calculations or the execution of processes are carried out simultaneously. The Points to Remember while working with Parallel Programming:

1. The Tasks must be independent.
2. The order of the execution does not matter

**C# Supports Two Types of Parallelism:**

**Data parallelism:**In the case of Data Parallelism, the operation is applied to each element of a collection. This means each process does the same work on unique and independent pieces of data.

**Example:**

1. [**Parallel.For**](https://dotnettutorials.net/lesson/parallel-for-method-csharp/)
2. [**Parallel.ForEach**](https://dotnettutorials.net/lesson/parallel-foreach-method-csharp/)

**Task parallelism:**In the case of Task Parallelism independent computations are executed in parallel. This means each process performs a different function or executes different code sections that are independent.

# Parallel For in C#

## ****Parallel For in C# with Examples****

In this article, I am going to discuss the static**Parallel For in C#** with Examples. Please read our previous article before proceeding to this article where we discussed the basic concepts of [**Parallel Programming**](https://dotnettutorials.net/lesson/task-parallel-library-overview/) in C#. As part of this article, we will discuss the need and use of the Parallel For loop compared with the C# for loop. So, let’s start the discussion with one of the most frequently asked interview questions.

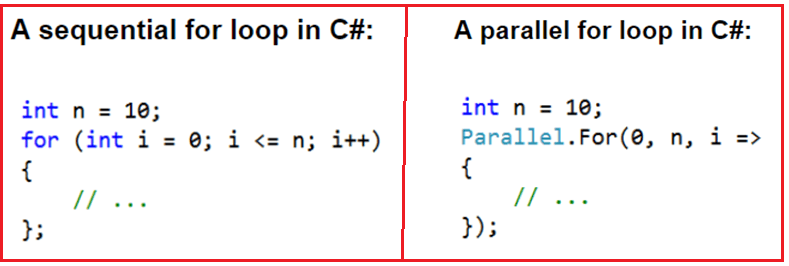
##### ****What is the difference between the Parallel For loop and Standard C# for loop?****

The main difference between the Parallel For loop and the standard C# for loop is as follows

1. In the case of the standard C# for loop, the loop is going to run using a single thread whereas, in the case of Parallel For loop, the loop is going to execute using multiple threads.
2. The second difference is that, in the case of the standard C# for loop, the loop is iterated in sequential order whereas, in the case of Parallel For loop, the order of the iteration is not going to be in sequential order.

**Note:** When the iterations are independent of each other, means, subsequent iterations do not need the state updates made by previous iterations, then in such cases, we need to use Task Parallel Library (TPL) to run each iteration in parallel on all the available cores. Moreover, the iteration should be an expensive iteration otherwise we will get negative performance, which we will also discuss as part of this article.

###### **Syntax:**



###### **Let us see an example for a better understanding of the above two types of for loop in C#:**

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"C# For Loop"**)**;

**int** number = 10;

**for** **(int** count = 0; count **<** number; count++**)**

**{**

//Thread.CurrentThread.ManagedThreadId returns an integer that

//represents a unique identifier for the current managed thread.

Console.WriteLine**(**$"value of count = {count}, thread = {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep the loop for 10 miliseconds

Thread.Sleep**(**10**)**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Parallel For Loop"**)**;

Parallel.For**(**0, number, count =**>**

**{**

Console.WriteLine**(**$"value of count = {count}, thread = {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep the loop for 10 miliseconds

Thread.Sleep**(**10**)**;

**})**;

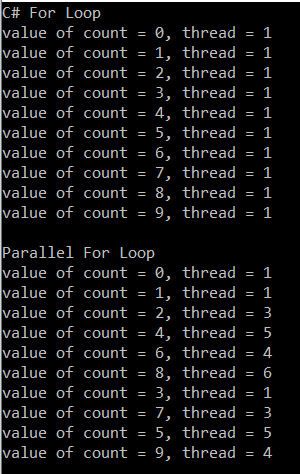
Console.ReadLine**()**;

**}**

**}**

**}**

As you can see in the above example, the static “For” method of the static “Parallel” class is defined as **public static ParallelLoopResult For(int fromInclusive, int toExclusive, Action<int> body);**. Here the first parameter (**i.e. int fromInclusive**) is the start index. The second parameter (**i.e. int toExclusive**) is the end index and the third parameter (**i.e. Action<int> body**) is the delegate which is invoked once per iteration. You can find many overloaded versions of this method in the Parallel class. Once you run the above code, you will get the following output.



As you can see in the above output, the standard C# for loop iterates sequentially using a single thread as a result, the results are printed sequentially. On the other hand, you can see with the Parallel for loop the results are not printed in sequential order. This is because it uses multiple threads to iterate over the collection. You can see that in our example it uses five threads to execute the code. It may vary in your system.

##### ****Let’s consider another example for a better understanding from a performance point of view.****

First, we will write the example using C# for loop and will see how much time it will take to complete the execution. Then we will write the same example using the Parallel For method and will see how much time it will take to complete the execution.

In the below example, we create a sequential loop. The loop iterates ten times, and the loop control variable increasing from zero to nine. In each iteration, the **DoSomeIndependentTask** method is called. The **DoSomeIndependentTask** method performs a calculation that is included to generate a long enough pause to see the performance improvement of the parallel version.

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

DateTime StartDateTime = DateTime.Now;

Console.WriteLine**(**@"For Loop Execution start at : {0}", StartDateTime**)**;

**for** **(int** i = 0; i **<** 10; i++**)**

**{**

**long** total = DoSomeIndependentTask**()**;

Console.WriteLine**(**"{0} - {1}", i, total**)**;

**}**

DateTime EndDateTime = DateTime.Now;

Console.WriteLine**(**@"For Loop Execution end at : {0}", EndDateTime**)**;

TimeSpan span = EndDateTime - StartDateTime;

**int** ms = **(int)**span.TotalMilliseconds;

Console.WriteLine**(**@"Time Taken to Execute the For Loop in miliseconds {0}", ms**)**;

Console.WriteLine**(**"Press any key to exist"**)**;

Console.ReadLine**()**;

**}**

**static** **long** DoSomeIndependentTask**()**

**{**

//Do Some Time Consuming Task here

//Most Probably some calculation or DB related activity

**long** total = 0;

**for** **(int** i = 1; i **<** 100000000; i++**)**

**{**

total += i;

**}**

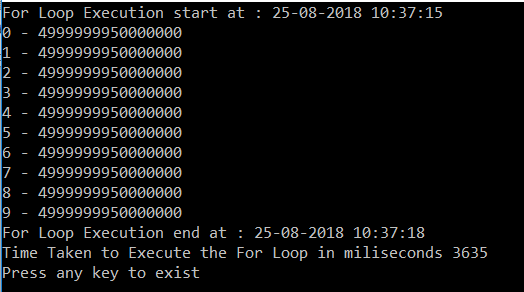
**return** total;

**}**

**}**

**}**

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



As you can see from the above output the for loop statement took approximately 3635 milliseconds to complete the execution. **Let’s rewrite the same example using the Parallel For method.**

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

DateTime StartDateTime = DateTime.Now;

Console.WriteLine**(**@"Parallel For Loop Execution start at : {0}", StartDateTime**)**;

Parallel.For**(**0, 10, i =**>** **{**

**long** total = DoSomeIndependentTask**()**;

Console.WriteLine**(**"{0} - {1}", i, total**)**;

**})**;

DateTime EndDateTime = DateTime.Now;

Console.WriteLine**(**@"Parallel For Loop Execution end at : {0}", EndDateTime**)**;

TimeSpan span = EndDateTime - StartDateTime;

**int** ms = **(int)**span.TotalMilliseconds;

Console.WriteLine**(**@"Time Taken to Execute the Loop in miliseconds {0}", ms**)**;

Console.WriteLine**(**"Press any key to exist"**)**;

Console.ReadLine**()**;

**}**

**static** **long** DoSomeIndependentTask**()**

**{**

//Do Some Time Consuming Task here

//Most Probably some calculation or DB related activity

**long** total = 0;

**for** **(int** i = 1; i **<** 100000000; i++**)**

**{**

total += i;

**}**

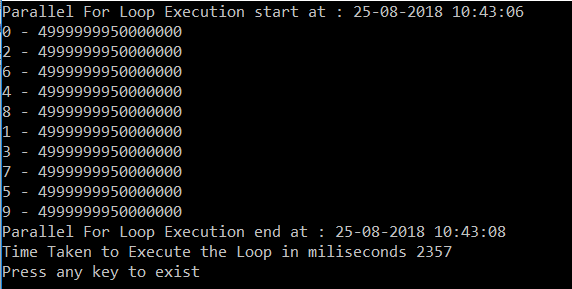
**return** total;

**}**

**}**

**}**

**OUTPUT:**



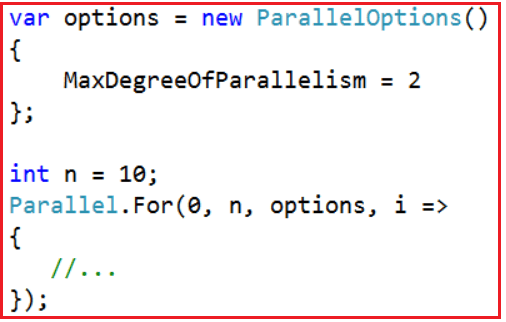
As shown in the above output, the Parallel For method took 2357 milliseconds to complete the execution.

##### ****ParallelOptions class in C#****

The ParallelOptions class is one of the most useful classes when working with multithreading. This class provides options to limit the number of concurrently executing loop methods.

##### ****The Degree of parallelism:****

Using the Degree of parallelism we can specify the maximum number of threads to be used to execute the program. Following is the syntax to use ParallelOptions class with Degree of parallelism.



The MaxDegreeOfParallelism property affects the number of concurrent operations run by Parallel method calls that are passed this ParallelOptions instance. A positive property value limits the number of concurrent operations to the set value. If it is -1, there is no limit on the number of concurrently running operations.

**Let us see an example for a better understanding of the MaxDegreeOfParallelism.**

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Limiting the maximum degree of parallelism to 2

var options = new ParallelOptions**()**

**{**

MaxDegreeOfParallelism = 2

**}**;

**int** n = 10;

Parallel.For**(**0, n, options, i =**>**

**{**

Console.WriteLine**(**@"value of i = {0}, thread = {1}",

i, Thread.CurrentThread.ManagedThreadId**)**;

Thread.Sleep**(**10**)**;

**})**;

Console.WriteLine**(**"Press any key to exist"**)**;

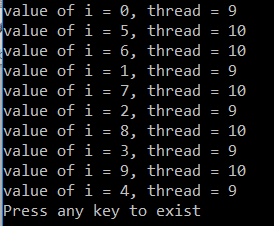
Console.ReadLine**()**;

**}**

**}**

**}**

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



As we set the degree of parallelism to 2. So, a maximum of 2 threads is used to execute the code that we can see from the above output.

##### ****Terminating a Parallel Loop:****

The below example shows how to break out of a For loop and also how to stop a loop. In this context, “break” means complete all iterations on all threads that are prior to the current iteration on the current thread, and then exit the loop. “Stop” means to stop all iterations as soon as convenient.

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

var BreakSource = Enumerable.Range**(**0, 1000**)**.ToList**()**;

**int** BreakData = 0;

Console.WriteLine**(**"Using loopstate Break Method"**)**;

Parallel.For**(**0, BreakSource.Count, **(**i, BreakLoopState**)** =**>**

**{**

BreakData += i;

**if** **(**BreakData **>** 100**)**

**{**

BreakLoopState.Break**()**;

Console.WriteLine**(**"Break called iteration {0}. data = {1} ", i, BreakData**)**;

**}**

**})**;

Console.WriteLine**(**"Break called data = {0} ", BreakData**)**;

var StopSource = Enumerable.Range**(**0, 1000**)**.ToList**()**;

**int** StopData = 0;

Console.WriteLine**(**"Using loopstate Stop Method"**)**;

Parallel.For**(**0, StopSource.Count, **(**i, StopLoopState**)** =**>**

**{**

StopData += i;

**if** **(**StopData **>** 100**)**

**{**

StopLoopState.Stop**()**;

Console.WriteLine**(**"Stop called iteration {0}. data = {1} ", i, StopData**)**;

**}**

**})**;

Console.WriteLine**(**"Stop called data = {0} ", StopData**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

In a Parallel.For or Parallel.ForEach loop, you cannot use the same break or Exit statement that is used in a sequential loop because those language constructs are valid for loops, and a parallel “loop” is actually a method, not a loop. Instead, you use either the Stop or Break method.

# Parallel Foreach in C#

## ****Parallel Foreach in C#****

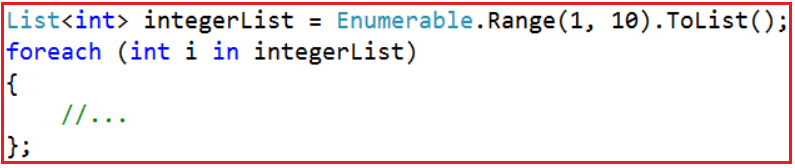
In this article, I am going to discuss the **Parallel Foreach in C#** with examples.  As we already discussed in our previous article that the **Task Parallel Library (TPL)** provides two methods (i.e. **Parallel.For** and **Parallel.Foreach**) which are conceptually the “for” and “for each” loops, except that, they use multiple threads to execute multiple iterations at the same time on a machine with multiple cores. In our previous article, we already discussed the [**Parallel for Method in C#**](https://dotnettutorials.net/lesson/parallel-for-method-csharp/)with examples. Here, in this article, I am going to keep the focus on the Parallel Foreach method in C#.

##### ****Parallel.ForEach in C#****

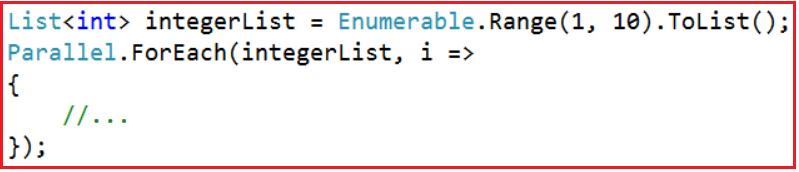
The **Parallel ForEach in C#** provides a parallel version of the standard, sequential Foreach loop. In a standard Foreach loop, each iteration processes a single item from the collection and will process all the items one by one only. However, the Parallel Foreach method executes multiple iterations at the same time on different processors or processor cores. This may open the possibility of synchronization problems. So, the loop is ideally suited to processes where each iteration is independent of the others.

**Note:** We need to use parallel loops such as Parallel.For and Parallel.ForEach method to speed up operations where an expensive, independent operation needs to be performed for each input of a sequence.

**A sequential Foreach loop in C#:**



**A parallel Foreach loop in C#:**



The parallel version of the loop uses the static **ForEach**method of the **Parallel**class. There are many overloaded versions available for this method. This is the simplest overloaded version that accepts two arguments. The first one is the collection of objects that will be enumerated. This can be any collection that implements **IEnumerable<T>**.

The second parameter accepts an Action delegate, usually expressed as a lambda expression that determines the action to take for each item in the collection. The delegate’s parameter contains the item from the collection that is to be processed during the iteration.

##### ****C# Parallel Foreach Method Example.****

Let us understand Parallel Foreach with an example. First, we will write an example using the standard sequential Foreach loop and will see how much time it will take to complete the execution. Then we will write the same example using the Parallel ForEach method and will see how much time it will take to complete the execution of the same example.

In the below example, we create a sequential Foreach loop that performs a long-running task once for each item in the collection. The code below loops through a list of ten integers generated using the **Enumerable.Range** method. In each iteration, the **DoSomeIndependentTimeconsumingTask** method is called. The **DoSomeIndependentTimeconsumingTask** method performs a calculation that is included to generate a long enough pause to see the performance improvement of the parallel version.

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

DateTime StartDateTime = DateTime.Now;

Console.WriteLine**(**@"foreach Loop start at : {0}", StartDateTime**)**;

List**<int>** integerList = Enumerable.Range**(**1, 10**)**.ToList**()**;

**foreach** **(int** i in integerList**)**

**{**

**long** total = DoSomeIndependentTimeconsumingTask**()**;

Console.WriteLine**(**"{0} - {1}", i, total**)**;

**}**;

DateTime EndDateTime = DateTime.Now;

Console.WriteLine**(**@"foreach Loop end at : {0}", EndDateTime**)**;

TimeSpan span = EndDateTime - StartDateTime;

**int** ms = **(int)**span.TotalMilliseconds;

Console.WriteLine**(**@"Time Taken by foreach Loop in miliseconds {0}", ms**)**;

Console.WriteLine**(**"Press any key to exist"**)**;

Console.ReadLine**()**;

**}**

**static** **long** DoSomeIndependentTimeconsumingTask**()**

**{**

//Do Some Time Consuming Task here

//Most Probably some calculation or DB related activity

**long** total = 0;

**for** **(int** i = 1; i **<** 100000000; i++**)**

**{**

total += i;

**}**

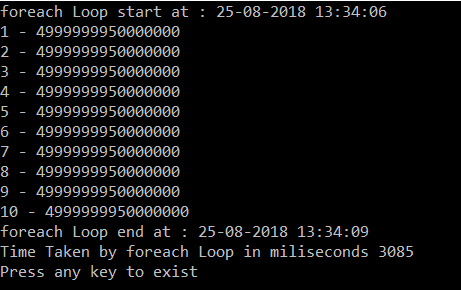
**return** total;

**}**

**}**

**}**

**Now run the application and observe the output.**



As you can see from the above output screen the “**Foreach**” loop statement took approximately 3035 milliseconds to complete the execution.

##### ****Let’s rewrite the same example using the C# Parallel ForEach method.****

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

DateTime StartDateTime = DateTime.Now;

Console.WriteLine**(**@"Parallel foreach method start at : {0}", StartDateTime**)**;

List**<int>** integerList = Enumerable.Range**(**1, 10**)**.ToList**()**;

Parallel.ForEach**(**integerList, i =**>**

**{**

**long** total = DoSomeIndependentTimeconsumingTask**()**;

Console.WriteLine**(**"{0} - {1}", i, total**)**;

**})**;

DateTime EndDateTime = DateTime.Now;

Console.WriteLine**(**@"Parallel foreach method end at : {0}", EndDateTime**)**;

TimeSpan span = EndDateTime - StartDateTime;

**int** ms = **(int)**span.TotalMilliseconds;

Console.WriteLine**(**@"Time Taken by Parallel foreach method in miliseconds {0}", ms**)**;

Console.WriteLine**(**"Press any key to exist"**)**;

Console.ReadLine**()**;

**}**

**static** **long** DoSomeIndependentTimeconsumingTask**()**

**{**

//Do Some Time Consuming Task here

//Most Probably some calculation or DB related activity

**long** total = 0;

**for** **(int** i = 1; i **<** 100000000; i++**)**

**{**

total += i;

**}**

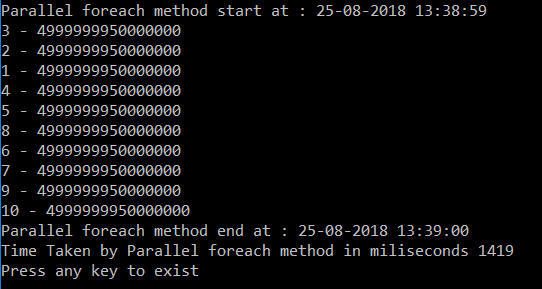
**return** total;

**}**

**}**

**}**

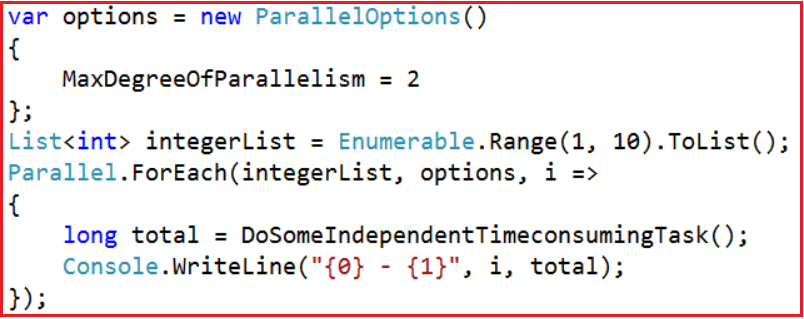
Now, run the application and see the output as shown below. The time may vary on your machine.



As you can see in the above output, the Parallel.ForEach method took 1419 milliseconds to complete the execution.

##### ****The Degree of Parallelism:****

Using the Degree of Parallelism we can specify the maximum number of threads to be used to execute the program. The syntax to use the Degree of Parallelism is given below.



The **MaxDegreeOfParallelism** property affects the number of concurrent operations run by Parallel method calls that are passed this ParallelOptions instance. A positive property value limits the number of concurrent operations to the set value. If it is -1, there is no limit on the number of concurrently running operations.

By default, For and ForEach will utilize however many threads the underlying scheduler provides, so changing MaxDegreeOfParallelism from the default only limits how many concurrent tasks will be used.

###### **Let us see an example for a better understanding.**

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

List**<int>** integerList = Enumerable.Range**(**0,10**)**.ToList**()**;

Parallel.ForEach**(**integerList, i =**>**

**{**

Console.WriteLine**(**@"value of i = {0}, thread = {1}",

i, Thread.CurrentThread.ManagedThreadId**)**;

**})**;

Console.WriteLine**(**"Press any key to exist"**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Now run the above code multiple times, and definitely, you will get different output. You will also observe that the number of threads is created is not in our control. Now, let us see how to restrict the number of threads to be created.

##### ****How to control the degree of concurrency i.e. How to restrict the number of threads to be created?****

We can restrict the number of concurrent threads created during the execution of parallel loops by using the MaxDegreeOfParallelism property. By assigning some value to **MaxDegreeOfParallelism**, we can restrict the degree of this concurrency and can restrict the number of processor cores to be used by our loops. The default value of this property is -1, which means there is no restriction on concurrently running operations.

**Let’s see the example.**

**namespace** *ParallelProgrammingDemo*

**{**

**class** Program

**{**

**static** **void** Main**()**

**{**

var options = new ParallelOptions**()**

**{**

MaxDegreeOfParallelism = 2

**}**;

List**<int>** integerList = Enumerable.Range**(**0,10**)**.ToList**()**;

Parallel.ForEach**(**integerList, options, i =**>**

**{**

Console.WriteLine**(**@"value of i = {0}, thread = {1}",

i, Thread.CurrentThread.ManagedThreadId**)**;

**})**;

Console.WriteLine**(**"Press any key to exist"**)**;

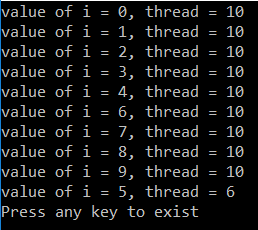
Console.ReadLine**()**;

**}**

**}**

**}**

In the above code, we are setting the **MaxDegreeOfParallelism** to 2, which means that only a maximum of 2 threads will be created, which will, in turn, use fewer cores, which is 2 here. Now run the application and see the output as shown below.



Whatever number of times we execute the above code, the number of threads will never go above 2.

# Parallel Invoke in C#

## ****Parallel Invoke in C# with Examples****

In this article, I am going to discuss the **Parallel Invoke in C#** with examples. The Parallel Invoke Method in C# is one of the most frequently used static method of the Parallel class. This Parallel Invoke method is used to launch multiple tasks that are going to be executed in parallel. Please read the following articles before proceeding to this article.

[**Overview of Task Parallel Library**](https://dotnettutorials.net/lesson/task-parallel-library-overview/)

[**Parallel For Method in C#**](https://dotnettutorials.net/lesson/parallel-for-method-csharp/)

[**Parallel ForEach Method in C#**](https://dotnettutorials.net/lesson/parallel-foreach-method-csharp/)

##### ****Let us understand Parallel Invoke Method in C# with an example.****

The following example demonstrates how to use Parallel Invoke method in C# with other methods, anonymous methods (delegates), and lambda expressions.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *ParallelProgrammingDemo*

**{**

**public** **class** Program

**{**

**static** **void** Main**()**

**{**

Parallel.Invoke**(**

NormalAction, // Invoking Normal Method

**delegate** **()** // Invoking an inline delegate

**{**

Console.WriteLine**(**$"Method 2, Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**,

**()** =**>** // Invoking a lambda expression

**{**

Console.WriteLine**(**$"Method 3, Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadKey**()**;

**}**

**static** **void** NormalAction**()**

**{**

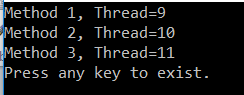
Console.WriteLine**(**$"Method 1, Thread={Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

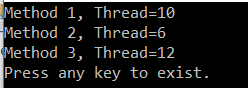
**}**

**}**

**First run OUTPUT:**



**2nd Run OUTPUT:**



The Parallel Invoke method is used to execute a set of operations (actions) in parallel. As you can see in the above output three threads are created to execute three actions which prove that this parallel Invoke Method executes the actions in parallel.

**Note:** The Parallel Invoke method in C# does not give you any guarantees about the order in which the actions are executed. Each time you execute the code, you may get a different order of output. Another important point that you need to remember is, this method is going to return when all the actions invoked by this method complete their execution.

##### ****ParallelOptions class****

As we already discussed, using the **ParallelOptions** class instance, we can limit the number of concurrently executing loop methods. The same thing can also be done with the Invoke method. So, using the Degree of parallelism we can specify the maximum number of threads to be used to execute the program.

##### ****Let’s understand this with an example.****

In the following example, we are creating seven actions without specifying a limit to the number of parallel tasks. So, in this example, it may be possible that all seven actions can be executed concurrently.

As you can see in the below example, we are calling the **DoSomeTask** method seven times using the Parallel Invoke method. As part of the **DoSomeTask** method, we are just printing two messages with a 5000 milliseconds pause between them. The messages showing when the task started and ended and by which thread so that you will understand the order of execution.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *ParallelProgrammingDemo*

**{**

**public** **class** ParallelInvoke

**{**

**static** **void** Main**()**

**{**

Parallel.Invoke**(**

**()** =**>** DoSomeTask**(**1**)**,

**()** =**>** DoSomeTask**(**2**)**,

**()** =**>** DoSomeTask**(**3**)**,

**()** =**>** DoSomeTask**(**4**)**,

**()** =**>** DoSomeTask**(**5**)**,

**()** =**>** DoSomeTask**(**6**)**,

**()** =**>** DoSomeTask**(**7**)**

**)**;

Console.ReadKey**()**;

**}**

**static** **void** DoSomeTask**(int** number**)**

**{**

Console.WriteLine**(**$"DoSomeTask {number} started by Thread {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep for 5000 milliseconds

Thread.Sleep**(**5000**)**;

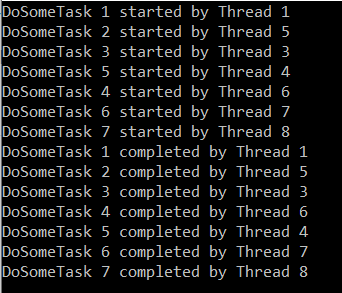
Console.WriteLine**(**$"DoSomeTask {number} completed by Thread {Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**}**

**}**

**Now run the application and see the output as shown below. The output may vary in your machine.**



You can see in the above output that each of the seven tasks started before any other completed which proofs that all the seven tasks run concurrently. In order to limit the parallelism i.e. to limit the number of threads to execute concurrently, we need to use the ParallelOptions class. We need to pass the object of ParallelOptions to the first parameter of the Invoke method.

###### **In the below example the code is limited to three concurrent tasks.**

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *ParallelProgrammingDemo*

**{**

**public** **class** ParallelInvoke

**{**

**static** **void** Main**()**

**{**

//Allowing three task to execute at a time

ParallelOptions parallelOptions = new ParallelOptions

**{**

MaxDegreeOfParallelism = 3

**}**;

//parallelOptions.MaxDegreeOfParallelism = System.Environment.ProcessorCount - 1;

//Passing ParallelOptions as the first parameter

Parallel.Invoke**(**

parallelOptions,

**()** =**>** DoSomeTask**(**1**)**,

**()** =**>** DoSomeTask**(**2**)**,

**()** =**>** DoSomeTask**(**3**)**,

**()** =**>** DoSomeTask**(**4**)**,

**()** =**>** DoSomeTask**(**5**)**,

**()** =**>** DoSomeTask**(**6**)**,

**()** =**>** DoSomeTask**(**7**)**

**)**;

Console.ReadKey**()**;

**}**

**static** **void** DoSomeTask**(int** number**)**

**{**

Console.WriteLine**(**$"DoSomeTask {number} started by Thread {Thread.CurrentThread.ManagedThreadId}"**)**;

//Sleep for 500 milliseconds

Thread.Sleep**(**5000**)**;

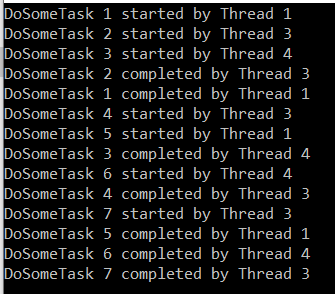
Console.WriteLine**(**$"DoSomeTask {number} completed by Thread {Thread.CurrentThread.ManagedThreadId}"**)**;

**}**

**}**

**}**

**OUTPUT:**



As you can see from the above output that, first three tasks have started concurrently as we set the degree of parallelism to 3. When one of the tasks completes its execution then another task started. This process will continue until all of the actions have completed their work. But the most important point that you need to remember is at any given point of time, no more than three tasks are running.