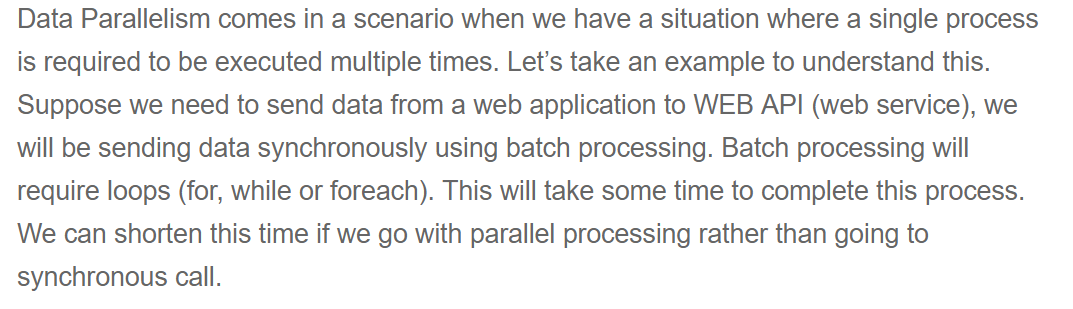
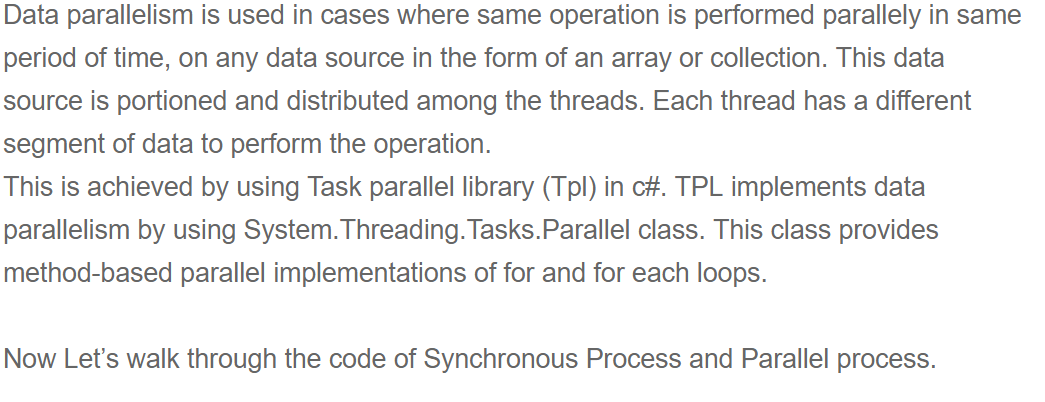
* **Task parallelism** is the simultaneous execution on multiple cores of many different functions across the same or different datasets.
* **Data parallelism** is the simultaneous execution on multiple cores of the same function across the elements of a dataset.

Data Parallelism in C#.NET





Synchronous Process using Loop:

namespace Data\_Parallelism

{

class Program

{

static void Main(string[] args)

{

synchronousProcess();

}

static void PostData()

{

Thread.Sleep(6000);

}

public static void synchronousProcess()

{

Console.WriteLine("Process started");

DateTime starttime = DateTime.Now;

List<int> batchList = new List<int> { 1, 2, 3, 4, 5 };

Console.WriteLine($"Start Time : {starttime}");

foreach (int item in batchList)

{

Console.WriteLine($" Begin Task : {item}");

PostData();

Console.WriteLine($"End Task :{item}");

}

DateTime endtime = DateTime.Now;

Console.WriteLine($" End Time: {endtime}" );

TimeSpan executiontime = endtime - starttime;

Console.WriteLine($" Total Timefor Execution : {executiontime}");

Console.WriteLine(" Synchronous Process End");

Console.ReadLine();

}

}

}

Now Let see how parallel Processing Works:

class Parallel\_ForEach\_Demo

{

static void Main(string[] args)

{

ParallelProcess();

}

static void postData()

{

Thread.Sleep(6000);

}

public static void ParallelProcess()

{

DateTime starttime = DateTime.Now;

Console.WriteLine("Parallel Process started ");

Console.WriteLine($" Process start Time : {starttime}" );

List<int> batchList = new List<int> { 1, 2, 3, 4, 5 };

Parallel.ForEach(batchList, item =>

{

Console.WriteLine($" Begin Task : {item}");

postData();

Console.WriteLine($" End Task :{ item} ");

});

DateTime endtime = DateTime.Now;

TimeSpan executiontime = endtime - starttime;

Console.WriteLine($" End Time : {endtime}");

Console.WriteLine(" Total Execution time " +executiontime);

Console.WriteLine(" Parallel process End ");

Console.ReadKey();

}

}

}

Parallel.For Loop in .NET 4

A lot of developers ask me about the difference between the C# for loop statement and the Parallel.For. The difference is that with the C# for statement, the loop is run from a single thread. However the Parallel class uses multiple threads. Moreover the order of the iteration in the parallel version is not necessarily in order.

Let us see an example:

using System;

using System.Threading;

using System.Threading.Tasks;

namespace ParallelFor

{

class Program

{

    static void Main(string[] args)

    {

        Console.WriteLine("Using C# For Loop \n");

        for(int i=0; i <=10; i++){

            Console.WriteLine("i = {0}, thread = {1}",

                i, Thread.CurrentThread.ManagedThreadId);

            Thread.Sleep(10);

        }

        Console.WriteLine("\nUsing Parallel.For \n");

        Parallel.For(0, 10, i =>

        {

            Console.WriteLine("i = {0}, thread = {1}", i,

            Thread.CurrentThread.ManagedThreadId);

            Thread.Sleep(10);

        });

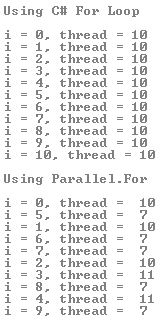
        Console.ReadLine();

    }

}

}

As you can see, the Parallel.For method is defined as **Parallel.For Method (Int32, Int32, Action(Of Int32))**. Here the first param is the start index (inclusive), the second param is the end index (exclusive) and the third param is the Action<int> delegate that is invoked once per iteration. Here’s the output of running the code:



As you can see, with the C# for loop statement, the results are printed sequentially and the loop is run from a single thread. However with the Parallel.For method uses multiple threads and the order of the iteration is not in order.

The Parallel.For() construct is useful if you have a set of data that is to be processed independently. The construct splits the task over multiple processor.

# How to: Write a Parallel.For Loop with Thread-Local Variables

This example shows how to use thread-local variables to store and retrieve state in each separate task that is created by a [For](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for) loop. By using thread-local data, you can avoid the overhead of synchronizing a large number of accesses to shared state. Instead of writing to a shared resource on each iteration, you compute and store the value until all iterations for the task are complete. You can then write the final result once to the shared resource, or pass it to another method.

## Example

The following example calls the [For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>)](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for#System_Threading_Tasks_Parallel_For__1_System_Int32_System_Int32_System_Func___0__System_Func_System_Int32_System_Threading_Tasks_ParallelLoopState___0___0__System_Action___0__) method to calculate the sum of the values in an array that contains one million elements. The value of each element is equal to its index.

namespace Data\_Parallelism

{

class For\_Local\_variables\_Demo

{

static void Main(string[] args)

{

int [] nums = Enumerable.Range(0, 5).ToArray();

long total = 0;

Parallel.For(0, nums.Length, () => 0, (j, loop, subtotal) =>

{

subtotal += nums[j];

return subtotal;

},

(x) => Interlocked.Add(ref total, x)

);

Console.WriteLine("The Total : " +total);

Console.ReadKey();

}

}

}

# How to: Write a Parallel.ForEach loop with partition-local variables

The following example shows how to write a [ForEach](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach) method that uses partition-local variables. When a [ForEach](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach) loop executes, it divides its source collection into multiple partitions. Each partition has its own copy of the partition-local variable. A partition-local variable is similar to a [thread-local variable](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadlocal-1), except that multiple partitions can run on a single thread.

static void Main()

{

int[] nums = Enumerable.Range(0, 1000000).ToArray();

long total = 0;

// First type parameter is the type of the source elements

// Second type parameter is the type of the thread-local variable (partition subtotal)

Parallel.ForEach<int, long>(nums, // source collection

() => 0, // method to initialize the local variable

(j, loop, subtotal) => // method invoked by the loop on each iteration

{

subtotal += j; //modify local variable

return subtotal; // value to be passed to next iteration

},

// Method to be executed when each partition has completed.

// finalResult is the final value of subtotal for a particular partition.

(finalResult) => Interlocked.Add(ref total, finalResult)

);

Console.WriteLine("The total from Parallel.ForEach is {0:N0}", total);

}

}

// The example displays the following output:

// The total from Parallel.ForEach is 499,999,500,000

# How to: Cancel a Parallel.For or ForEach Loop

## **AggregateException**

The AggregateException class is a [subclass](http://www.blackwasp.co.uk/Inheritance.aspx) of Exception so provides all of the standard exception functionality. In addition, it has a [property](http://www.blackwasp.co.uk/CSharpClassProperties.aspx) that holds a collection of inner exceptions. When thrown by a parallel loop, all of the encountered exceptions are included in the property, ensuring that no exception details are lost. It's important to note that an AggregateException will be thrown even if only one exception is encountered during the processing of the loop. Other exceptions can only be generated if there is a problem with the loop command itself, such as the [action delegate](http://www.blackwasp.co.uk/FuncAction.aspx) that defines the loop body being null.

In the following sections we'll demonstrate exception handling for parallel loops. To provide simplified access to the loop methods add the following [using directive](http://www.blackwasp.co.uk/Namespaces.aspx):

### **Catching Aggregated Exceptions**

For the first demonstration we'll create a parallel loop that throws an exception. As we know that the exception will be wrapped within an AggregateException, we'll only catch AggregrateExceptions. The For loop below iterates through the values between -10 and 9. Each value is used as the divisor in a simple arithmetic operation. When the value is zero a division by zero error occurs and an exception is thrown.

NB: When you run the sample code, Visual Studio may break on the division by zero exception to allow debugging. If it does, press F5 to continue execution. You can avoid the problem by pressing Ctrl-F5 to run the program without debugging.

|  |
| --- |
| try  {      Parallel.For(-10, 10, i =>      {          Console.WriteLine("100/{0}={1}", i, 100 / i);      });  }  catch (AggregateException ex)  {      Console.WriteLine(ex.Message);  }    /\* OUTPUT    100/-10=-10  100/-5=-20  100/-9=-11  100/-6=-16  100/-2=-50  100/-1=-100  100/1=100  100/2=50  100/3=33  100/4=25  100/6=16  100/7=14  100/8=12  100/9=11  100/5=20  100/-8=-12  100/-7=-14  100/-4=-25  100/-3=-33  One or more errors occurred.    \*/ |

The results shown in the above [comments](http://www.blackwasp.co.uk/CSharpComments.aspx) were generated on a quad-core processor. You can see that at some point in the process the division by zero exception was encountered and, after the loop stopped, the AggregateException was caught and its *Message* property outputted. The message is simple, indicating that the AggregateException contains one or more inner exceptions. *NB: If we had attempted to catch a DivideByZeroException the exception would have been unhandled.*

It is impossible to look at the above results and understand when the exception actually occurred. In a sequential loop we could assume that it was during the last processed iteration but in a parallel loop this may be incorrect. To understand what is actually happening, we can add an extra line to the code that tells us when we are about to divide by zero.

The sample below is modified to show a message just before the exception:

|  |
| --- |
| try  {      Parallel.For(-10, 10, i =>      {          if (i == 0) Console.WriteLine("About to divide by zero.");          Console.WriteLine("100/{0}={1}", i, 100 / i);      });  }  catch (AggregateException ex)  {      Console.WriteLine(ex.Message);  }    /\* OUTPUT    100/-10=-10  100/5=20  About to divide by zero.  100/-5=-20  100/-4=-25  100/-3=-33  100/-2=-50  100/-1=-100  100/1=100  100/2=50  100/3=33  100/4=25  100/8=12  100/9=11  100/-6=-16  100/-9=-11  100/-8=-12  100/-7=-14  100/6=16  100/7=14  One or more errors occurred.    \*/ |

With the new message we can see that the exception happened very early in the process, after only two calculation results had been shown. The remaining calculations happened after the initial exception in previously scheduled loop iterations. You should always consider this possibility when writing parallel loop code.

### **Accessing the Exception Details**

Once you have caught an AggregateException you can examine each of the contained exceptions by reading the InnerExceptions property. This returns a read-only collection that may be addressed by index or enumerated.

The sample code below demonstrates the use of the InnerExceptions property by introducing two changes. First, the division has been modified to give the possibility that two exceptions will be thrown, rather than one. These occur when the loop control value is -10 or zero. The second change is that the catch block now loops through the InnerExceptions property of the AggregateException and outputs all of the error messages.

|  |
| --- |
| try  {      Parallel.For(-10, 10, i =>      {          Console.WriteLine("100/{0}={1}", i, 100 / (i % 10));      });  }  catch (AggregateException ex)  {      foreach (Exception inner in ex.InnerExceptions)      {          Console.WriteLine(inner.Message);      }  }    /\* OUTPUT    100/-5=-20  100/5=20  100/6=16  100/7=14  100/8=12  100/9=11  100/-8=-12  100/-7=-14  100/-6=-16  100/-2=-50  100/-1=-100  100/1=100  100/-9=-11  100/2=50  100/3=33  100/4=25  100/-4=-25  100/-3=-33  Attempted to divide by zero.  Attempted to divide by zero.    \*/ |

The [Parallel.For](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for) and [Parallel.ForEach](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach) methods support cancellation through the use of cancellation tokens. For more information about cancellation in general, see [Cancellation](https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads). In a parallel loop, you supply the [CancellationToken](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken) to the method in the [ParallelOptions](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions) parameter and then enclose the parallel call in a try-catch block.

namespace CancelParallelLoops

{

using System;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

class Program

{

static void Main()

{

int[] nums = Enumerable.Range(0, 10000000).ToArray();

CancellationTokenSource cts = new CancellationTokenSource();

// Use ParallelOptions instance to store the CancellationToken

ParallelOptions po = new ParallelOptions();

po.CancellationToken = cts.Token;

po.MaxDegreeOfParallelism = System.Environment.ProcessorCount;

Console.WriteLine("Press any key to start. Press 'c' to cancel.");

Console.ReadKey();

// Run a task so that we can cancel from another thread.

Task.Factory.StartNew(() =>

{

if (Console.ReadKey().KeyChar == 'c')

cts.Cancel();

Console.WriteLine("press any key to exit");

});

try

{

Parallel.ForEach(nums, po, (num) =>

{

double d = Math.Sqrt(num);

Console.WriteLine("{0} on {1}", d, Thread.CurrentThread.ManagedThreadId);

po.CancellationToken.ThrowIfCancellationRequested();

});

}

catch (OperationCanceledException e)

{

Console.WriteLine(e.Message);

}

finally

{

cts.Dispose();

}

Console.ReadKey();

}

}

}

If the token that signals the cancellation is the same token that is specified in the [ParallelOptions](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.paralleloptions)instance, then the parallel loop will throw a single [OperationCanceledException](https://docs.microsoft.com/en-us/dotnet/api/system.operationcanceledexception) on cancellation. If some other token causes cancellation, the loop will throw an [AggregateException](https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception) with an [OperationCanceledException](https://docs.microsoft.com/en-us/dotnet/api/system.operationcanceledexception) as an InnerException.