**Parallal Programming**

**Task Parallal Library**

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently.

The Task Parallel Library (TPL) supports data parallelism through the [System.Threading.Tasks.Parallel](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel) class. This class provides method-based parallel implementations of [for](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/for) and [foreach](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in) loops (For and For Each in Visual Basic). You write the loop logic for a [Parallel.For](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for) or [Parallel.ForEach](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach) loop much as you would write a sequential loop. You do not have to create threads or queue work items. In basic loops, you do not have to take locks.

// Sequential version

foreach (var item in sourceCollection)

{

Process(item);

}

// Parallel equivalent

Parallel.ForEach(sourceCollection, item => Process(item));

When a parallel loop runs, the TPL partitions the data source so that the loop can operate on multiple parts concurrently.

Program can be found managing a note of Time Difference between Parallal.For Loop and Parallal.ForEach loop

**Task-based Asynchronous Programming**

## **Creating and running tasks explicitly**

The task object handles the infrastructure details and provides methods and properties that are accessible from the calling thread throughout the lifetime of the task. For example, you can access the [Status](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.status) property of a task at any time to determine whether it has started running, ran to completion, was canceled, or has thrown an exception. The status is represented by a [TaskStatus](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskstatus) enumeration.

[Task.Wait](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.wait) method to ensure that the task completes execution before the console mode application ends.

[Task.Run](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run) methods to create and start a task in one operation. To manage the task, the [Run](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run) methods use the default task scheduler, regardless of which task scheduler is associated with the current thread. The [Run](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run) methods are the preferred way to create and start tasks when more control over the creation and scheduling of the task is not needed.

You can also use the [TaskFactory.StartNew](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskfactory.startnew) method to create and start a task in one operation. Use this method when creation and scheduling do not have to be separated and you require additional task creation options or the use of a specific scheduler, or when you need to pass additional state into the task that you can retrieve through its [Task.AsyncState](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.asyncstate) property.

Each thread has an associated culture and UI culture, which is defined by the [Thread.CurrentCulture](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.currentculture) and [Thread.CurrentUICulture](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.currentuiculture) properties, respectively. A thread's culture is used in such operations as formatting, parsing, sorting, and string comparison. The calling thread's culture is inherited by each task, even if the task runs asynchronously on a thread pool thread.

## **Creating task continuations**

The [Task.ContinueWith](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.continuewith) and [Task<TResult>.ContinueWith](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1.continuewith) methods let you specify a task to start when the antecedent task finishes.

var processData = getData.ContinueWith((x) => {

int n = x.Result.Length;

long sum = 0;

double mean;

for (int ctr = 0; ctr <= x.Result.GetUpperBound(0); ctr++)

sum += x.Result[ctr];

mean = sum / (double) n;

return Tuple.Create(n, sum, mean);

} );

## **Creating detached child tasks**

When user code that is running in a task creates a new task and does not specify the [AttachedToParent](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions" \l "System_Threading_Tasks_TaskCreationOptions_AttachedToParent) option, the new task is not synchronized with the parent task in any special way. This type of non-synchronized task is called a detached nested task or detached child task.

(Parent Task does not wait for the child task to get completed.)

## **Creating child tasks**

When user code that is running in a task creates a task with the [AttachedToParent](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions" \l "System_Threading_Tasks_TaskCreationOptions_AttachedToParent) option, the new task is known as a attached child task of the parent task. You can use the [AttachedToParent](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskcreationoptions" \l "System_Threading_Tasks_TaskCreationOptions_AttachedToParent) option to express structured task parallelism, because the parent task implicitly waits for all attached child tasks to finish.

(Parent Task wait for the child task to get completed.)

## **Exception Handling (Task Parallel Library)**

To propagate all the exceptions back to the calling thread, the Task infrastructure wraps them in an [AggregateException](https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception) instance. The [AggregateException](https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception) exception has an [InnerExceptions](https://docs.microsoft.com/en-us/dotnet/api/system.aggregateexception.innerexceptions) property that can be enumerated to examine all the original exceptions that were thrown, and handle (or not handle) each one individually.