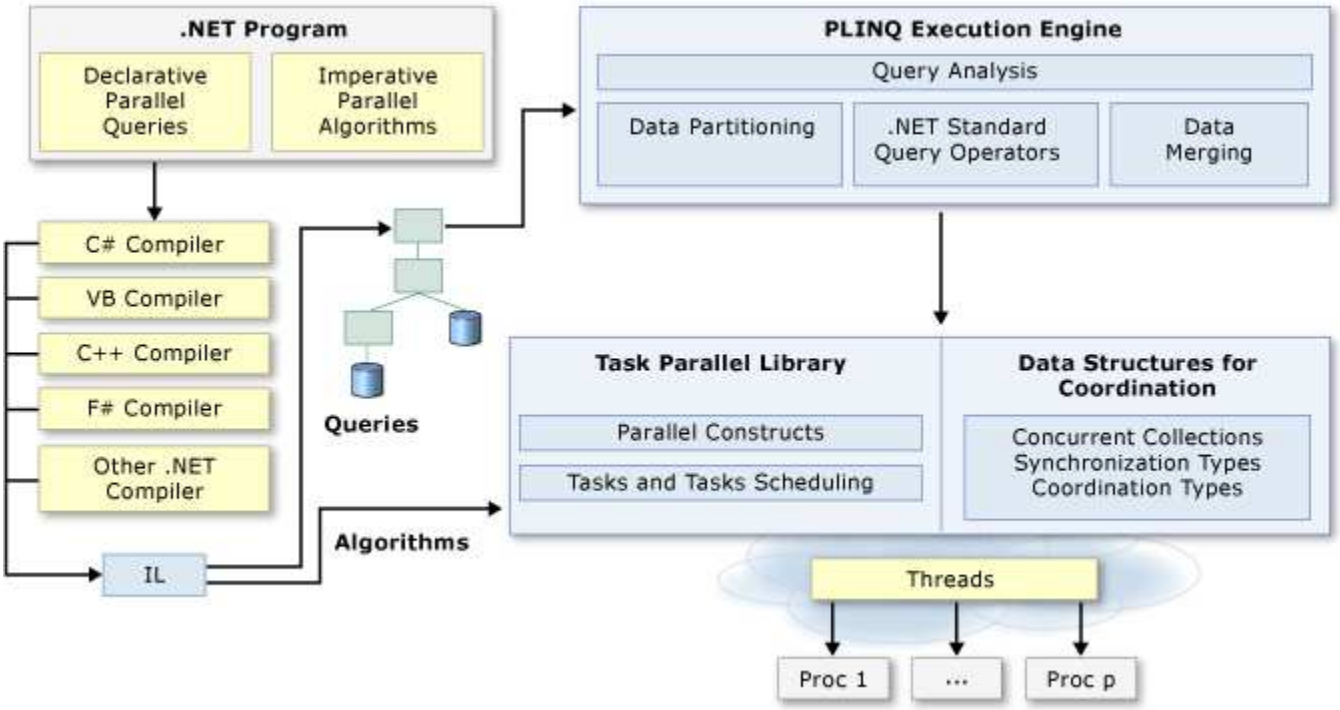
Parallel Programming in the .NET Framework

.NET Framework (current version)

Many personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously. Computers in the near future are expected to have significantly more cores. To take advantage of the hardware of today and tomorrow, you can parallelize your code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio 2010 and the .NET Framework 4 enhance support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools. These features simplify parallel development so that you can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. The following illustration provides a high-level overview of the parallel programming architecture in the .NET Framework 4.



Related Topics

Technology	Description
Task Parallel Library (TPL)	Provides documentation for the System.Threading.Tasks.Parallel class, which includes parallel versions of **For** and **ForEach** loops, and also for the System.Threading.Tasks.Task class, which represents the preferred way to express asynchronous operations.
Parallel LINQ (PLINQ)	A parallel implementation of LINQ to Objects that significantly improves performance in many scenarios.

Data Structures for Parallel Programming	Provides links to documentation for thread-safe collection classes, lightweight synchronization types, and types for lazy initialization.
Parallel Diagnostic Tools	Provides links to documentation for Visual Studio debugger windows for tasks and parallel stacks, and the Concurrency Visualizer, which consists of a set of views in the Visual Studio Application Lifecycle Management Profiler that you can use to debug and to tune the performance of parallel code.
Custom Partitioners for PLINQ and TPL	Describes how partitioners work and how to configure the default partitioners or create a new partitioner.
Task Schedulers	Describes how schedulers work and how the default schedulers may be configured.
Lambda Expressions in PLINQ and TPL	Provides a brief overview of lambda expressions in C# and Visual Basic, and shows how they are used in PLINQ and the Task Parallel Library.
For Further Reading (Parallel Programming)	Provides links to additional documentation and sample resources for parallel programming in the .NET Framework.

See Also

Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4
Samples for Parallel Programming with the .NET Framework

Task Parallel Library (TPL)

.NET Framework (current version)

The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.

Starting with the .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization; for example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multithreaded code adds complexity to your program execution. Although the TPL simplifies multithreaded scenarios, we recommend that you have a basic understanding of threading concepts, for example, locks, deadlocks, and race conditions, so that you can use the TPL effectively.

Related Topics

Title	Description
Data Parallelism (Task Parallel Library)	Describes how to create parallel **for** and **foreach** loops (**For** and **For Each** in Visual Basic).
Task Parallelism (Task Parallel Library)	Describes how to create and run tasks implicitly by using Parallel.Invoke or explicitly by using Task objects directly.
Dataflow (Task Parallel Library)	Describes how to use the dataflow components in the TPL Dataflow Library to handle multiple operations that must communicate with one another or to process data as it becomes available.
Using TPL with Other Asynchronous Patterns	Describes how to use TPL with other asynchronous patterns in .NET
Potential Pitfalls in Data and Task Parallelism	Describes some common pitfalls and how to avoid them.
Parallel LINQ (PLINQ)	Describes how to achieve data parallelism with LINQ queries.
Parallel Programming in the .NET Framework	Top level node for .NET parallel programming.

See Also

Samples for Parallel Programming with the .NET Framework

Data Parallelism (Task Parallel Library)

.NET Framework (current version)

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 class. This class provides method-based parallel implementations of for and foreach loops (**For** and **For Each** in Visual Basic). You write the loop logic for a Parallel.For or 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. The TPL handles all the low-level work for you. For in-depth information about the use of Parallel.For and Parallel.ForEach, download the document Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4. The following code example shows a simple **foreach** loop and its parallel equivalent.

Note

This documentation uses lambda expressions to define delegates in TPL. If you are not familiar with lambda expressions in C# or Visual Basic, see Lambda Expressions in PLINQ and TPL.

VB

```
' Sequential version
For Each item In sourceCollection
    Process(item)
Next

' Parallel equivalent
Parallel.ForEach(sourceCollection, Sub(item) Process(item))
```

When a parallel loop runs, the TPL partitions the data source so that the loop can operate on multiple parts concurrently. Behind the scenes, the Task Scheduler partitions the task based on system resources and workload. When possible, the scheduler redistributes work among multiple threads and processors if the workload becomes unbalanced.

Note

You can also supply your own custom partitioner or scheduler. For more information, see Custom Partitioners for PLINQ and TPL and Task Schedulers.

Both the Parallel.For and Parallel.ForEach methods have several overloads that let you stop or break loop execution, monitor the state of the loop on other threads, maintain thread-local state, finalize thread-local objects, control the degree of concurrency, and so on. The helper types that enable this functionality include ParallelLoopState, ParallelOptions,

ParallelLoopResult, CancellationToken, and CancellationTokenSource.

For more information, see Patterns of Parallel Programming.

Data parallelism with declarative, or query-like, syntax is supported by PLINQ. For more information, see Parallel LINQ (PLINQ).

Related Topics

Title	Description
How to: Write a Simple Parallel.For Loop	Describes how to write a For loop over any array or indexable IEnumerable(Of T) source collection.
How to: Write a Simple Parallel.ForEach Loop	Describes how to write a ForEach loop over any IEnumerable(Of T) source collection.
How to: Stop or Break from a Parallel.For Loop	Describes how to stop or break from a parallel loop so that all threads are informed of the action.
How to: Write a Parallel.For Loop with Thread-Local Variables	Describes how to write a For loop in which each thread maintains a private variable that is not visible to any other threads, and how to synchronize the results from all threads when the loop completes.
How to: Write a Parallel.ForEach Loop with Thread-Local Variables	Describes how to write a ForEach loop in which each thread maintains a private variable that is not visible to any other threads, and how to synchronize the results from all threads when the loop completes.
How to: Cancel a Parallel.For or ForEach Loop	Describes how to cancel a parallel loop by using a System.Threading.CancellationToken
How to: Speed Up Small Loop Bodies	Describes one way to speed up execution when a loop body is very small.
Task Parallel Library (TPL)	Provides an overview of the Task Parallel Library.
Parallel Programming in the .NET Framework	Introduces Parallel Programming in the .NET Framework.

See Also

Parallel Programming in the .NET Framework

Task Parallelism (Task Parallel Library)

.NET Framework (current version)

The Task Parallel Library (TPL) is based on the concept of a *task*, which represents an asynchronous operation. In some ways, a task resembles a thread or ThreadPool work item, but at a higher level of abstraction. The term *task parallelism* refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:

- More efficient and more scalable use of system resources.

  Behind the scenes, tasks are queued to the ThreadPool, which has been enhanced with algorithms that determine and adjust to the number of threads and that provide load balancing to maximize throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism.

- More programmatic control than is possible with a thread or work item.

  Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

For both of these reasons, in the .NET Framework, TPL is the preferred API for writing multi-threaded, asynchronous, and parallel code.

Creating and running tasks implicitly

The Parallel.Invoke method provides a convenient way to run any number of arbitrary statements concurrently. Just pass in an Action delegate for each item of work. The easiest way to create these delegates is to use lambda expressions. The lambda expression can either call a named method or provide the code inline. The following example shows a basic Invoke call that creates and starts two tasks that run concurrently. The first task is represented by a lambda expression that calls a method named `DoSomeWork`, and the second task is represented by a lambda expression that calls a method named `DoSomeOtherWork`.

> **Note**
>
> This documentation uses lambda expressions to define delegates in TPL. If you are not familiar with lambda expressions in C# or Visual Basic, see Lambda Expressions in PLINQ and TPL.

VB

```
Parallel.Invoke(Sub() DoSomeWork(), Sub() DoSomeOtherWork())
```

> **Note**

The number of Task instances that are created behind the scenes by Invoke is not necessarily equal to the number of delegates that are provided. The TPL may employ various optimizations, especially with large numbers of delegates.

For more information, see How to: Use Parallel.Invoke to Execute Parallel Operations.

For greater control over task execution or to return a value from the task, you have to work with Task objects more explicitly.

Creating and running tasks explicitly

A task that does not return a value is represented by the System.Threading.Tasks.Task class. A task that returns a value is represented by the System.Threading.Tasks.Task(Of TResult) class, which inherits from Task. 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 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 enumeration.

When you create a task, you give it a user delegate that encapsulates the code that the task will execute. The delegate can be expressed as a named delegate, an anonymous method, or a lambda expression. Lambda expressions can contain a call to a named method, as shown in the following example. Note that the example includes a call to the Task.Wait method to ensure that the task completes execution before the console mode application ends.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        ' Create a task and supply a user delegate by using a lambda expression.
        Dim taskA = New Task(Sub() Console.WriteLine("Hello from taskA."))
        ' Start the task.
        taskA.Start()

        ' Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name)
        taskA.Wait()
   End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

You can also use the Task.Run methods to create and start a task in one operation. To manage the task, the Run methods use the default task scheduler, regardless of which task scheduler is associated with the current thread. The 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.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        Dim taskA As Task = Task.Run(Sub() Console.WriteLine("Hello from taskA."))

        ' Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name)
        taskA.Wait()
   End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

You can also use the 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 through its AsyncState property, as shown in the following example.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Thread.CurrentThread.Name = "Main"

      ' Better: Create and start the task in one operation.
      Dim taskA = Task.Factory.StartNew(Sub() Console.WriteLine("Hello from taskA."))

      ' Output a message from the calling thread.
      Console.WriteLine("Hello from thread '{0}'.",
                        Thread.CurrentThread.Name)

      taskA.Wait()
   End Sub
End Module
' The example displays output like the following:
'       Hello from thread 'Main'.
'       Hello from taskA.
```

Task and Task(Of TResult) each expose a static Factory property that returns a default instance of TaskFactory, so that you

can call the method as `Task.Factory.StartNew()`. Also, in the following example, because the tasks are of type System.Threading.Tasks.Task(Of TResult), they each have a public Task(Of TResult).Result property that contains the result of the computation. The tasks run asynchronously and may complete in any order. If the Result property is accessed before the computation finishes, the property blocks the calling thread until the value is available.

VB

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
        Dim taskArray() = { Task(Of Double).Factory.StartNew(Function()
DoComputation(1.0)),
                            Task(Of Double).Factory.StartNew(Function()
DoComputation(100.0)),
                            Task(Of Double).Factory.StartNew(Function()
DoComputation(1000.0)) }

        Dim results(taskArray.Length - 1) As Double
        Dim sum As Double

        For i As Integer = 0 To taskArray.Length - 1
            results(i) = taskArray(i).Result
            Console.Write("{0:N1} {1}", results(i),
                          If(i = taskArray.Length - 1, "= ", "+ "))
            sum += results(i)
        Next
        Console.WriteLine("{0:N1}", sum)
   End Sub

   Private Function DoComputation(start As Double) As Double
      Dim sum As Double
      For value As Double = start To start + 10 Step .1
         sum += value
      Next
      Return sum
   End Function
End Module
' The example displays the following output:
'       606.0 + 10,605.0 + 100,495.0 = 111,706.0
```

For more information, see How to: Return a Value from a Task.

When you use a lambda expression to create a delegate, you have access to all the variables that are visible at that point in your source code. However, in some cases, most notably within loops, a lambda doesn't capture the variable as expected. It only captures the final value, not the value as it mutates after each iteration. The following example illustrates the problem. It passes a loop counter to a lambda expression that instantiates a `CustomData` object and uses the loop counter as the object's identifier. As the output from the example shows, each `CustomData` object has an identical identifier.

VB

```
Imports System.Threading
```

```
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
   Public Name As Integer
   Public ThreadNum As Integer
End Class

Module Example
   Public Sub Main()
      ' Create the task object by using an Action(Of Object) to pass in the loop
      ' counter. This produces an unexpected result.
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As New CustomData With {.Name
= i, .CreationTime = DateTime.Now.Ticks}
                                                 data.ThreadNum =
Thread.CurrentThread.ManagedThreadId
                                                 Console.WriteLine("Task #{0} created
at {1} on thread #{2}.",
                                                                   data.Name,
data.CreationTime, data.ThreadNum)
                                              End Sub,
                                              i )
      Next
      Task.WaitAll(taskArray)
   End Sub
End Module
' The example displays output like the following:
'       Task #10 created at 635116418427727841 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427727841 on thread #3.
'       Task #10 created at 635116418427747843 on thread #3.
'       Task #10 created at 635116418427747843 on thread #3.
'       Task #10 created at 635116418427737842 on thread #4.
```

You can access the value on each iteration by providing a state object to a task through its constructor. The following example modifies the previous example by using the loop counter when creating the `CustomData` object, which, in turn, is passed to the lambda expression. As the output from the example shows, each `CustomData` object now has a unique identifier based on the value of the loop counter at the time the object was instantiated.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
```

```
   Public Name As Integer
   Public ThreadNum As Integer
End Class

Module Example
   Public Sub Main()
      ' Create the task object by using an Action(Of Object) to pass in custom data
      ' to the Task constructor. This is useful when you need to capture outer variables
      ' from within a loop.
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As CustomData = TryCast(obj,
CustomData)
                                                 If data Is Nothing Then Return

                                                 data.ThreadNum =
Thread.CurrentThread.ManagedThreadId
                                                 Console.WriteLine("Task #{0} created
at {1} on thread #{2}.",
                                                                   data.Name,
data.CreationTime, data.ThreadNum)
                                              End Sub,
                                              New CustomData With {.Name = i,
.CreationTime = DateTime.Now.Ticks} )
      Next
      Task.WaitAll(taskArray)
   End Sub
End Module
' The example displays output like the following:
'       Task #0 created at 635116412924597583 on thread #3.
'       Task #1 created at 635116412924607584 on thread #4.
'       Task #3 created at 635116412924607584 on thread #4.
'       Task #4 created at 635116412924607584 on thread #4.
'       Task #2 created at 635116412924607584 on thread #3.
'       Task #6 created at 635116412924607584 on thread #3.
'       Task #5 created at 635116412924607584 on thread #4.
'       Task #8 created at 635116412924607584 on thread #4.
'       Task #7 created at 635116412924607584 on thread #3.
'       Task #9 created at 635116412924607584 on thread #4.
```

This state is passed as an argument to the task delegate, and it can be accessed from the task object by using the Task.AsyncState property. The following example is a variation on the previous example. It uses the AsyncState property to display information about the `CustomData` objects passed to the lambda expression.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
   Public CreationTime As Long
   Public Name As Integer
```

```
    Public ThreadNum As Integer
End Class

Module Example
   Public Sub Main()
      Dim taskArray(9) As Task
      For i As Integer = 0 To taskArray.Length - 1
         taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                 Dim data As CustomData = TryCast(obj,
CustomData)
                                                 If data Is Nothing Then Return

                                                 data.ThreadNum =
Thread.CurrentThread.ManagedThreadId
                                              End Sub,
                                              New CustomData With {.Name = i,
.CreationTime = DateTime.Now.Ticks} )
      Next
      Task.WaitAll(taskArray)

      For Each task In taskArray
         Dim data = TryCast(task.AsyncState, CustomData)
         If data IsNot Nothing Then
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum)
         End If
      Next
   End Sub
End Module
' The example displays output like the following:
'    Task #0 created at 635116451245250515, ran on thread #3, RanToCompletion
'    Task #1 created at 635116451245270515, ran on thread #4, RanToCompletion
'    Task #2 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #3 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #4 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #5 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #6 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #7 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #8 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #9 created at 635116451245270515, ran on thread #3, RanToCompletion
```

Task ID

Every task receives an integer ID that uniquely identifies it in an application domain and can be accessed by using the Task.Id property. The ID is useful for viewing task information in the Visual Studio debugger **Parallel Stacks** and **Tasks** windows. The ID is lazily created, which means that it isn't created until it is requested; therefore, a task may have a different ID every time the program is run. For more information about how to view task IDs in the debugger, see Using the Tasks Window and Using the Parallel Stacks Window.

Task Creation Options

Most APIs that create tasks provide overloads that accept a TaskCreationOptions parameter. By specifying one of these options, you tell the task scheduler how to schedule the task on the thread pool. The following table lists the various task creation options.

TaskCreationOptions parameter value	**Description**
None	The default when no option is specified. The scheduler uses its default heuristics to schedule the task.
PreferFairness	Specifies that the task should be scheduled so that tasks created sooner will be more likely to be executed sooner, and tasks created later will be more likely to execute later.
LongRunning	Specifies that the task represents a long-running operation.
AttachedToParent	Specifies that a task should be created as an attached child of the current task, if one exists. For more information, see Attached and Detached Child Tasks.
DenyChildAttach	Specifies that if an inner task specifies the **AttachedToParent** option, that task will not become an attached child task.
HideScheduler	Specifies that the task scheduler for tasks created by calling methods like TaskFactory.StartNew or Task(Of TResult).ContinueWith from within a particular task is the default scheduler instead of the scheduler on which this task is running.

The options may be combined by using a bitwise **OR** operation. The following example shows a task that has the LongRunning and PreferFairness option.

VB

```
Dim task3 = New Task(Sub() MyLongRunningMethod(),
                         TaskCreationOptions.LongRunning Or
TaskCreationOptions.PreferFairness)
task3.Start()
```

Tasks, threads, and culture

Each thread has an associated culture and UI culture, which is defined by the Thread.CurrentCulture and Thread.CurrentUICulture properties, respectively. A thread's culture is used in such operations as formatting, parsing, sorting, and string comparison. A thread's UI culture is used in resource lookup. Ordinarily, unless you specify a default culture for all the threads in an application domain by using the CultureInfo.DefaultThreadCurrentCulture and CultureInfo.DefaultThreadCurrentUICulture properties, the default culture and UI culture of a thread is defined by the

system culture. If you explicitly set a thread's culture and launch a new thread, the new thread does not inherit the culture of the calling thread; instead, its culture is the default system culture. The task-based programming model for apps that target versions of the .NET Framework prior to .NET Framework 4.6 adhere to this practice.

Important

Note that the calling thread's culture as part of a task's context applies to apps that *target* the .NET Framework 4.6, not apps that *run under* the .NET Framework 4.6. You can target a particular version of the .NET Framework when you create your project in Visual Studio by selecting that version from the dropdown list at the top of the **New Project** dialog box, or outside of Visual Studio you can use the TargetFrameworkAttribute attribute. For apps that target versions of the .NET Framework prior to the .NET Framework 4.6, or that do not target a specific version of the .NET Framework, a task's culture continues to be determined by the culture of the thread on which it runs.

Starting with apps that target the .NET Framework 4.6, the calling thread's culture is inherited by each task, even if the task runs asynchronously on a thread pool thread.

The following example provides a simple illustration. It uses the TargetFrameworkAttribute attribute to target the .NET Framework 4.6 and changes the app's current culture to either French (France) or, if French (France) is already the current culture, English (United States). It then invokes a delegate named `formatDelegate` that returns some numbers formatted as currency values in the new culture. Note that whether the delegate as a task either synchronously or asynchronously, it returns the expected result because the culture of the calling thread is inherited by the asynchronous task.

VB

```
Imports System.Globalization
Imports System.Runtime.Versioning
Imports System.Threading
Imports System.Threading.Tasks

<Assembly:TargetFramework(".NETFramework,Version=v4.6")>

Module Example
   Public Sub Main()
       Dim values() As Decimal = { 163025412.32d, 18905365.59d }
       Dim formatString As String = "C2"
       Dim formatDelegate As Func(Of String) = Function()
                                                  Dim output As String =
String.Format("Formatting using the {0} culture on thread {1}.",

CultureInfo.CurrentCulture.Name,

Thread.CurrentThread.ManagedThreadId)
                                                  output += Environment.NewLine
                                                  For Each value In values
                                                     output += String.Format("{0}   ",
value.ToString(formatString))
                                                  Next
                                                  output += Environment.NewLine
                                                  Return output
                                               End Function
```

```
        Console.WriteLine("The example is running on thread {0}",
                          Thread.CurrentThread.ManagedThreadId)
        ' Make the current culture different from the system culture.
        Console.WriteLine("The current culture is {0}",
                          CultureInfo.CurrentCulture.Name)
        If CultureInfo.CurrentCulture.Name = "fr-FR" Then
           Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US")
        Else
           Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR")
        End If
        Console.WriteLine("Changed the current culture to {0}.",
                          CultureInfo.CurrentCulture.Name)
        Console.WriteLine()

        ' Execute the delegate synchronously.
        Console.WriteLine("Executing the delegate synchronously:")
        Console.WriteLine(formatDelegate())

        ' Call an async delegate to format the values using one format string.
        Console.WriteLine("Executing a task asynchronously:")
        Dim t1 = Task.Run(formatDelegate)
        Console.WriteLine(t1.Result)

        Console.WriteLine("Executing a task synchronously:")
        Dim t2 = New Task(Of String)(formatDelegate)
        t2.RunSynchronously()
        Console.WriteLine(t2.Result)
    End Sub
End Module
' The example displays the following output:
'          The example is running on thread 1
'          The current culture is en-US
'          Changed the current culture to fr-FR.
'
'          Executing the delegate synchronously:
'          Formatting Imports the fr-FR culture on thread 1.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task asynchronously:
'          Formatting Imports the fr-FR culture on thread 3.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task synchronously:
'          Formatting Imports the fr-FR culture on thread 1.
'          163 025 412,32 €   18 905 365,59 €
' If the TargetFrameworkAttribute statement is removed, the example
' displays the following output:
'          The example is running on thread 1
'          The current culture is en-US
'          Changed the current culture to fr-FR.
'
'          Executing the delegate synchronously:
'          Formatting using the fr-FR culture on thread 1.
'          163 025 412,32 ?   18 905 365,59 ?
```

```
'
'          Executing a task asynchronously:
'          Formatting using the en-US culture on thread 3.
'          $163,025,412.32   $18,905,365.59
'
'          Executing a task synchronously:
'          Formatting using the fr-FR culture on thread 1.
'          163 025 412,32 ?   18 905 365,59 ?
```

If you are using Visual Studio, you can omit the TargetFrameworkAttribute attribute and instead select the .NET Framework 4.6 as the target when you create the project in the **New Project** dialog.

For output that reflects the behavior of apps the target versions of the .NET Framework prior to .NET Framework 4.6, remove the TargetFrameworkAttribute attribute from the source code. The output will reflect the formatting conventions of the default system culture, not the culture of the calling thread.

For more information on asynchronous tasks and culture, see the "Culture and asynchronous task-based operations" section in the CultureInfo topic.

Creating task continuations

The Task.ContinueWith and Task(Of TResult).ContinueWith methods let you specify a task to start when the *antecedent task* finishes. The delegate of the continuation task is passed a reference to the antecedent task so that it can examine the antecedent task's status and, by retrieving the value of the Task(Of TResult).Result property, can use the output of the antecedent as input for the continuation.

In the following example, the `getData` task is started by a call to the TaskFactory.StartNew(Of TResult)(Func(Of TResult)) method. The `processData` task is started automatically when `getData` finishes, and `displayData` is started when `processData` finishes. `getData` produces an integer array, which is accessible to the `processData` task through the `getData` task's Task(Of TResult).Result property. The `processData` task processes that array and returns a result whose type is inferred from the return type of the lambda expression passed to the Task(Of TResult).ContinueWith(Of TNewResult)(Func(Of Task(Of TResult), TNewResult)) method. The `displayData` task executes automatically when `processData` finishes, and the Tuple(Of T1, T2, T3) object returned by the `processData` lambda expression is accessible to the `displayData` task through the `processData` task's Task(Of TResult).Result property. The `displayData` task takes the result of the `processData` task and produces a result whose type is inferred in a similar manner and which is made available to the program in the Result property.

VB

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim getData = Task.Factory.StartNew(Function()
                                             Dim rnd As New Random()
                                             Dim values(99) As Integer
                                             For ctr = 0 To values.GetUpperBound(0)
                                                values(ctr) = rnd.Next()
                                             Next
                                             Return values
```

```
                                            End Function)
      Dim processData = getData.ContinueWith(Function(x)
                                                Dim n As Integer = x.Result.Length
                                                Dim sum As Long
                                                Dim mean As Double

                                                For ctr = 0 To x.Result.GetUpperBound(0)
                                                   sum += x.Result(ctr)
                                                Next
                                                mean = sum / n
                                                Return Tuple.Create(n, sum, mean)
                                             End Function)
      Dim displayData = processData.ContinueWith(Function(x)
                                                    Return String.Format("N={0:N0},
Total = {1:N0}, Mean = {2:N2}",

x.Result.Item1, x.Result.Item2,
                                                                         x.Result.Item3)
                                                 End Function)
      Console.WriteLine(displayData.Result)
   End Sub
End Module
' The example displays output like the following:
'    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

Because Task.ContinueWith is an instance method, you can chain method calls together instead of instantiating a Task(Of TResult) object for each antecedent task. The following example is functionally identical to the previous example, except that it chains together calls to the Task.ContinueWith method. Note that the Task(Of TResult) object returned by the chain of method calls is the final continuation task.

VB

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim displayData = Task.Factory.StartNew(Function()
                                                 Dim rnd As New Random()
                                                 Dim values(99) As Integer
                                                 For ctr = 0 To values.GetUpperBound(0)
                                                    values(ctr) = rnd.Next()
                                                 Next
                                                 Return values
                                              End Function). _
                        ContinueWith(Function(x)
                                        Dim n As Integer = x.Result.Length
                                        Dim sum As Long
                                        Dim mean As Double

                                        For ctr = 0 To x.Result.GetUpperBound(0)
                                           sum += x.Result(ctr)
                                        Next
                                        mean = sum / n
```

```
                                        Return Tuple.Create(n, sum, mean)
                                    End Function). _
                          ContinueWith(Function(x)
                                        Return String.Format("N={0:N0}, Total = {1:N0},
Mean = {2:N2}",
                                                             x.Result.Item1,
x.Result.Item2,
                                                             x.Result.Item3)
                                    End Function)
      Console.WriteLine(displayData.Result)
   End Sub
End Module
' The example displays output like the following:
'    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

The ContinueWhenAll and ContinueWhenAny methods enable you to continue from multiple tasks.

For more information, see Chaining Tasks by Using Continuation Tasks.

Creating detached child tasks

When user code that is running in a task creates a new task and does not specify the 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*. The following example shows a task that creates one detached child task.

VB

```
Dim outer = Task.Factory.StartNew(Sub()
                                      Console.WriteLine("Outer task beginning.")
                                      Dim child = Task.Factory.StartNew(Sub()

Thread.SpinWait(5000000)

Console.WriteLine("Detached task completed.")
                                                                          End Sub)
                                  End Sub)
outer.Wait()
Console.WriteLine("Outer task completed.")
' The example displays the following output:
'      Outer task beginning.
'      Outer task completed.
'     Detached child completed.
```

Note that the parent task does not wait for the detached child task to finish.

Creating child tasks

When user code that is running in a task creates a task with the AttachedToParent option, the new task is known as a

attached child task of the parent task. You can use the AttachedToParent option to express structured task parallelism, because the parent task implicitly waits for all attached child tasks to finish. The following example shows a parent task that creates ten attached child tasks. Note that although the example calls the Task.Wait method to wait for the parent task to finish, it does not have to explicitly wait for the attached child tasks to complete.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim parent = Task.Factory.StartNew(Sub()
                                            Console.WriteLine("Parent task beginning.")
                                            For ctr As Integer = 0 To 9
                                               Dim taskNo As Integer = ctr
                                               Task.Factory.StartNew(Sub(x)

Thread.SpinWait(5000000)

Console.WriteLine("Attached child #{0} completed.",

x)
                                                                     End Sub,
                                                                     taskNo,
TaskCreationOptions.AttachedToParent)
                                            Next
                                         End Sub)
      parent.Wait()
      Console.WriteLine("Parent task completed.")
   End Sub
End Module
' The example displays output like the following:
'       Parent task beginning.
'       Attached child #9 completed.
'       Attached child #0 completed.
'       Attached child #8 completed.
'       Attached child #1 completed.
'       Attached child #7 completed.
'       Attached child #2 completed.
'       Attached child #6 completed.
'       Attached child #3 completed.
'       Attached child #5 completed.
'       Attached child #4 completed.
'       Parent task completed.
```

A parent task can use the TaskCreationOptions.DenyChildAttach option to prevent other tasks from attaching to the parent task. For more information, see Attached and Detached Child Tasks.

Waiting for tasks to finish

The System.Threading.Tasks.Task and System.Threading.Tasks.Task(Of TResult) types provide several overloads of the Task.Wait and Task(Of TResult).Wait methods that enable you to wait for a task to finish. In addition, overloads of the static Task.WaitAll and Task.WaitAny methods let you wait for any or all of an array of tasks to finish.

Typically, you would wait for a task for one of these reasons:

- The main thread depends on the final result computed by a task.
- You have to handle exceptions that might be thrown from the task.
- The application may terminate before all tasks have completed execution. For example, console applications will terminate as soon as all synchronous code in `Main` (the application entry point) has executed.

The following example shows the basic pattern that does not involve exception handling.

VB

```
Dim tasks() =
{
    Task.Factory.StartNew(Sub() MethodA()),
    Task.Factory.StartNew(Sub() MethodB()),
    Task.Factory.StartNew(Sub() MethodC())
}

' Block until all tasks complete.
Task.WaitAll(tasks)

' Continue on this thread...
```

For an example that shows exception handling, see Exception Handling (Task Parallel Library).

Some overloads let you specify a time-out, and others take an additional CancellationToken as an input parameter, so that the wait itself can be canceled either programmatically or in response to user input.

When you wait for a task, you implicitly wait for all children of that task that were created by using the TaskCreationOptions.AttachedToParent option. Task.Wait returns immediately if the task has already completed. Any exceptions raised by a task will be thrown by a Task.Wait method, even if the Task.Wait method was called after the task completed.

Composing tasks

The Task and Task(Of TResult) classes provide several methods that can help you compose multiple tasks to implement common patterns and to better use the asynchronous language features that are provided by C#, Visual Basic, and F#. This section describes the WhenAll, WhenAny, Delay, and FromResult(Of TResult) methods.

Task.WhenAll

The Task.WhenAll method asynchronously waits for multiple Task or Task(Of TResult) objects to finish. It provides overloaded versions that enable you to wait for non-uniform sets of tasks. For example, you can wait for multiple Task

and Task(Of TResult) objects to complete from one method call.

Task.WhenAny

The Task.WhenAny method asynchronously waits for one of multiple Task or Task(Of TResult) objects to finish. As in the Task.WhenAll method, this method provides overloaded versions that enable you to wait for non-uniform sets of tasks. The WhenAny method is especially useful in the following scenarios.

- Redundant operations. Consider an algorithm or operation that can be performed in many ways. You can use the WhenAny method to select the operation that finishes first and then cancel the remaining operations.
- Interleaved operations. You can start multiple operations that must all finish and use the WhenAny method to process results as each operation finishes. After one operation finishes, you can start one or more additional tasks.
- Throttled operations. You can use the WhenAny method to extend the previous scenario by limiting the number of concurrent operations.
- Expired operations. You can use the WhenAny method to select between one or more tasks and a task that finishes after a specific time, such as a task that is returned by the Delay method. The Delay method is described in the following section.

Task.Delay

The Task.Delay method produces a Task object that finishes after the specified time. You can use this method to build loops that occasionally poll for data, introduce time-outs, delay the handling of user input for a predetermined time, and so on.

Task(T).FromResult

By using the Task.FromResult(Of TResult) method, you can create a Task(Of TResult) object that holds a pre-computed result. This method is useful when you perform an asynchronous operation that returns a Task(Of TResult) object, and the result of that Task(Of TResult) object is already computed. For an example that uses FromResult(Of TResult) to retrieve the results of asynchronous download operations that are held in a cache, see How to: Create Pre-Computed Tasks.

Handling exceptions in tasks

When a task throws one or more exceptions, the exceptions are wrapped in an AggregateException exception. That exception is propagated back to the thread that joins with the task, which is typically the thread that is waiting for the task to finish or the thread that accesses the Result property. This behavior serves to enforce the .NET Framework policy that all unhandled exceptions by default should terminate the process. The calling code can handle the exceptions by using any of the following in a **try**/**catch** block:

- The Wait method
- The WaitAll method
- The WaitAny method
- The Result property

The joining thread can also handle exceptions by accessing the Exception property before the task is garbage-collected. By accessing this property, you prevent the unhandled exception from triggering the exception propagation behavior that terminates the process when the object is finalized.

For more information about exceptions and tasks, see Exception Handling (Task Parallel Library).

Canceling tasks

The **Task** class supports cooperative cancellation and is fully integrated with the System.Threading.CancellationTokenSource and System.Threading.CancellationToken classes, which were introduced in the .NET Framework 4. Many of the constructors in the System.Threading.Tasks.Task class take a CancellationToken object as an input parameter. Many of the StartNew and Run overloads also include a CancellationToken parameter.

You can create the token, and issue the cancellation request at some later time, by using the CancellationTokenSource class. Pass the token to the Task as an argument, and also reference the same token in your user delegate, which does the work of responding to a cancellation request.

For more information, see Task Cancellation and How to: Cancel a Task and Its Children.

The TaskFactory class

The TaskFactory class provides static methods that encapsulate some common patterns for creating and starting tasks and continuation tasks.

- The most common pattern is StartNew, which creates and starts a task in one statement.
- When you create continuation tasks from multiple antecedents, use the ContinueWhenAll method orContinueWhenAny method or their equivalents in the Task(Of TResult) class. For more information, see Chaining Tasks by Using Continuation Tasks.
- To encapsulate Asynchronous Programming Model `BeginX` and `EndX` methods in a Task or Task(Of TResult) instance, use the FromAsync methods. For more information, see TPL and Traditional .NET Framework Asynchronous Programming.

The default TaskFactory can be accessed as a static property on the Task class or Task(Of TResult) class. You can also instantiate a TaskFactory directly and specify various options that include a CancellationToken, a TaskCreationOptions option, a TaskContinuationOptions option, or a TaskScheduler. Whatever options are specified when you create the task factory will be applied to all tasks that it creates, unless the Task is created by using the TaskCreationOptions enumeration, in which case the task's options override those of the task factory.

Tasks without delegates

In some cases, you may want to use a Task to encapsulate some asynchronous operation that is performed by an external component instead of your own user delegate. If the operation is based on the Asynchronous Programming Model Begin/End pattern, you can use the FromAsync methods. If that is not the case, you can use the TaskCompletionSource(Of TResult) object to wrap the operation in a task and thereby gain some of the benefits of Task programmability, for example, support for exception propagation and continuations. For more information, see TaskCompletionSource(Of TResult).

Custom schedulers

Most application or library developers do not care which processor the task runs on, how it synchronizes its work with other tasks, or how it is scheduled on the System.Threading.ThreadPool. They only require that it execute as efficiently as possible on the host computer. If you require more fine-grained control over the scheduling details, the Task Parallel Library lets you configure some settings on the default task scheduler, and even lets you supply a custom scheduler. For more information, see TaskScheduler.

Related data structures

The TPL has several new public types that are useful in both parallel and sequential scenarios. These include several thread-safe, fast and scalable collection classes in the System.Collections.Concurrent namespace, and several new synchronization types, for example, System.Threading.Semaphore and System.Threading.ManualResetEventSlim, which are more efficient than their predecessors for specific kinds of workloads. Other new types in the .NET Framework 4, for example, System.Threading.Barrier and System.Threading.SpinLock, provide functionality that was not available in earlier releases. For more information, see Data Structures for Parallel Programming.

Custom task types

We recommend that you do not inherit from System.Threading.Tasks.Task or System.Threading.Tasks.Task(Of TResult). Instead, we recommend that you use the AsyncState property to associate additional data or state with a Task or Task(Of TResult) object. You can also use extension methods to extend the functionality of the Task and Task(Of TResult) classes. For more information about extension methods, see Extension Methods (C# Programming Guide) and Extension Methods (Visual Basic).

If you must inherit from Task or Task(Of TResult), you cannot use Run, Run(Of TResult), or the System.Threading.Tasks.TaskFactory, System.Threading.Tasks.TaskFactory(Of TResult), or System.Threading.Tasks.TaskCompletionSource(Of TResult) classes to create instances of your custom task type because these mechanisms create only Task and Task(Of TResult) objects. In addition, you cannot use the task continuation mechanisms that are provided by Task, Task(Of TResult), TaskFactory, and TaskFactory(Of TResult) to create instances of your custom task type because these mechanisms also create only Task and Task(Of TResult) objects.

Related topics

Title	Description
Chaining Tasks by Using Continuation Tasks	Describes how continuations work.
Attached and Detached Child Tasks	Describes the difference between attached and detached child tasks.
Task Cancellation	Describes the cancellation support that is built into the Task object.
Exception Handling (Task Parallel Library)	Describes how exceptions on concurrent threads are handled.
How to: Use Parallel.Invoke to Execute Parallel Operations	Describes how to use Invoke.
How to: Return a Value from a Task	Describes how to return values from tasks.
How to: Cancel a Task and Its Children	Describes how to cancel tasks.
How to: Create Pre-Computed Tasks	Describes how to use the Task.FromResult(Of TResult) method to retrieve the results of asynchronous download operations that are held in a cache.
How to: Traverse a Binary Tree with Parallel Tasks	Describes how to use tasks to traverse a binary tree.
How to: Unwrap a Nested Task	Demonstrates how to use the Unwrap extension method.

Data Parallelism (Task Parallel Library)	Describes how to use For and ForEach(Of TSource, TLocal) to create parallel loops over data.
Parallel Programming in the .NET Framework	Top level node for .NET Framework parallel programming.

See Also

Parallel Programming in the .NET Framework
Samples for Parallel Programming with the .NET Framework

Dataflow (Task Parallel Library)

.NET Framework (current version)

The Task Parallel Library (TPL) provides dataflow components to help increase the robustness of concurrency-enabled applications. These dataflow components are collectively referred to as the *TPL Dataflow Library*. This dataflow model promotes actor-based programming by providing in-process message passing for coarse-grained dataflow and pipelining tasks. The dataflow components build on the types and scheduling infrastructure of the TPL and integrate with the C#, Visual Basic, and F# language support for asynchronous programming. These dataflow components are useful when you have multiple operations that must communicate with one another asynchronously or when you want to process data as it becomes available. For example, consider an application that processes image data from a web camera. By using the dataflow model, the application can process image frames as they become available. If the application enhances image frames, for example, by performing light correction or red-eye reduction, you can create a *pipeline* of dataflow components. Each stage of the pipeline might use more coarse-grained parallelism functionality, such as the functionality that is provided by the TPL, to transform the image.

This document provides an overview of the TPL Dataflow Library. It describes the programming model, the predefined dataflow block types, and how to configure dataflow blocks to meet the specific requirements of your applications.

Tip

The TPL Dataflow Library (System.Threading.Tasks.Dataflow namespace) is not distributed with the .NET Framework 4.5. To install the System.Threading.Tasks.Dataflow namespace, open your project in Visual Studio 2012, choose **Manage NuGet Packages** from the Project menu, and search online for the **Microsoft.Tpl.Dataflow** package.

This document contains the following sections:

Programming Model

The TPL Dataflow Library provides a foundation for message passing and parallelizing CPU-intensive and I/O-intensive applications that have high throughput and low latency. It also gives you explicit control over how data is buffered and moves around the system. To better understand the dataflow programming model, consider an application that asynchronously loads images from disk and creates a composite of those images. Traditional programming models typically require that you use callbacks and synchronization objects, such as locks, to coordinate tasks and access to shared data. By using the dataflow programming model, you can create dataflow objects that process images as they are

read from disk. Under the dataflow model, you declare how data is handled when it becomes available, and also any dependencies between data. Because the runtime manages dependencies between data, you can often avoid the requirement to synchronize access to shared data. In addition, because the runtime schedules work based on the asynchronous arrival of data, dataflow can improve responsiveness and throughput by efficiently managing the underlying threads. For an example that uses the dataflow programming model to implement image processing in a Windows Forms application, see Walkthrough: Using Dataflow in a Windows Forms Application.

Sources and Targets

The TPL Dataflow Library consists of *dataflow blocks*, which are data structures that buffer and process data. The TPL defines three kinds of dataflow blocks: *source blocks*, *target blocks*, and *propagator blocks*. A source block acts as a source of data and can be read from. A target block acts as a receiver of data and can be written to. A propagator block acts as both a source block and a target block, and can be read from and written to. The TPL defines the System.Threading.Tasks.Dataflow.ISourceBlock(Of TOutput) interface to represent sources, System.Threading.Tasks.Dataflow.ITargetBlock(Of TInput) to represent targets, and System.Threading.Tasks.Dataflow.IPropagatorBlock(Of TInput, TOutput) to represent propagators. IPropagatorBlock(Of TInput, TOutput) inherits from both ISourceBlock(Of TOutput), and ITargetBlock(Of TInput).

The TPL Dataflow Library provides several predefined dataflow block types that implement the ISourceBlock(Of TOutput), ITargetBlock(Of TInput), and IPropagatorBlock(Of TInput, TOutput) interfaces. These dataflow block types are described in this document in the section Predefined Dataflow Block Types.

Connecting Blocks

You can connect dataflow blocks to form *pipelines*, which are linear sequences of dataflow blocks, or *networks*, which are graphs of dataflow blocks. A pipeline is one form of network. In a pipeline or network, sources asynchronously propagate data to targets as that data becomes available. The ISourceBlock(Of TOutput).LinkTo method links a source dataflow block to a target block. A source can be linked to zero or more targets; targets can be linked from zero or more sources. You can add or remove dataflow blocks to or from a pipeline or network concurrently. The predefined dataflow block types handle all thread-safety aspects of linking and unlinking.

For an example that connects dataflow blocks to form a basic pipeline, see Walkthrough: Creating a Dataflow Pipeline. For an example that connects dataflow blocks to form a more complex network, see Walkthrough: Using Dataflow in a Windows Forms Application. For an example that unlinks a target from a source after the source offers the target a message, see How to: Unlink Dataflow Blocks.

Filtering

When you call the ISourceBlock(Of TOutput).LinkTo method to link a source to a target, you can supply a delegate that determines whether the target block accepts or rejects a message based on the value of that message. This filtering mechanism is a useful way to guarantee that a dataflow block receives only certain values. For most of the predefined dataflow block types, if a source block is connected to multiple target blocks, when a target block rejects a message, the source offers that message to the next target. The order in which a source offers messages to targets is defined by the source and can vary according to the type of the source. Most source block types stop offering a message after one target accepts that message. One exception to this rule is the BroadcastBlock(Of T) class, which offers each message to all targets, even if some targets reject the message. For an example that uses filtering to process only certain messages, see Walkthrough: Using Dataflow in a Windows Forms Application.

Important

Because each predefined source dataflow block type guarantees that messages are propagated out in the order in which they are received, every message must be read from the source block before the source block can process the next message. Therefore, when you use filtering to connect multiple targets to a source, make sure that at least one target block receives each message. Otherwise, your application might deadlock.

Message Passing

The dataflow programming model is related to the concept of *message passing*, where independent components of a program communicate with one another by sending messages. One way to propagate messages among application components is to call the Post(Of TInput) and DataflowBlock.SendAsync(Of TInput) methods to send messages to target dataflow blocks post (Post(Of TInput) acts synchronously; SendAsync(Of TInput) acts asynchronously) and the Receive(Of TOutput), ReceiveAsync(Of TOutput), and TryReceive(Of TOutput) methods to receive messages from source blocks. You can combine these methods with dataflow pipelines or networks by sending input data to the head node (a target block), and receiving output data from the terminal node of the pipeline or the terminal nodes of the network (one or more source blocks). You can also use the Choose method to read from the first of the provided sources that has data available and perform action on that data.

Source blocks offer data to target blocks by calling the ITargetBlock(Of TInput).OfferMessage method. The target block responds to an offered message in one of three ways: it can accept the message, decline the message, or postpone the message. When the target accepts the message, the OfferMessage method returns Accepted. When the target declines the message, the OfferMessage method returns Declined. When the target requires that it no longer receives any messages from the source, OfferMessage returns DecliningPermanently. The predefined source block types do not offer messages to linked targets after such a return value is received, and they automatically unlink from such targets.

When a target block postpones the message for later use, the OfferMessage method returns Postponed. A target block that postpones a message can later calls the ISourceBlock(Of TOutput).ReserveMessage method to try to reserve the offered message. At this point, the message is either still available and can be used by the target block, or the message has been taken by another target. When the target block later requires the message or no longer needs the message, it calls the ISourceBlock(Of TOutput).ConsumeMessage or ReleaseReservation method, respectively. Message reservation is typically used by the dataflow block types that operate in non-greedy mode. Non-greedy mode is explained later in this document. Instead of reserving a postponed message, a target block can also use the ISourceBlock(Of TOutput).ConsumeMessage method to attempt to directly consume the postponed message.

Dataflow Block Completion

Dataflow blocks also support the concept of *completion*. A dataflow block that is in the completed state does not perform any further work. Each dataflow block has an associated System.Threading.Tasks.Task object, known as a *completion task*, that represents the completion status of the block. Because you can wait for a Task object to finish, by using completion tasks, you can wait for one or more terminal nodes of a dataflow network to finish. The IDataflowBlock interface defines the Complete method, which informs the dataflow block of a request for it to complete, and the Completion property, which returns the completion task for the dataflow block. Both ISourceBlock(Of TOutput) and ITargetBlock(Of TInput) inherit the IDataflowBlock interface.

There are two ways to determine whether a dataflow block completed without error, encountered one or more errors, or was canceled. The first way is to call the Task.Wait method on the completion task in a **try**-**catch** block (**Try**-**Catch** in Visual Basic). The following example creates an ActionBlock(Of TInput) object that throws

ArgumentOutOfRangeException if its input value is less than zero. AggregateException is thrown when this example calls Wait on the completion task. The ArgumentOutOfRangeException is accessed through the InnerExceptions property of the AggregateException object.

VB

```
            ' Create an ActionBlock<int> object that prints its input
            ' and throws ArgumentOutOfRangeException if the input
            ' is less than zero.
            Dim throwIfNegative = New ActionBlock(Of Integer)(Sub(n)
                Console.WriteLine("n = {0}", n)
                If n < 0 Then
                    Throw New ArgumentOutOfRangeException()
                End If
            End Sub)

            ' Post values to the block.
            throwIfNegative.Post(0)
            throwIfNegative.Post(-1)
            throwIfNegative.Post(1)
            throwIfNegative.Post(-2)
            throwIfNegative.Complete()

            ' Wait for completion in a try/catch block.
            Try
               throwIfNegative.Completion.Wait()
            Catch ae As AggregateException
               ' If an unhandled exception occurs during dataflow processing, all
               ' exceptions are propagated through an AggregateException object.
               ae.Handle(Function(e)
                   Console.WriteLine("Encountered {0}: {1}", e.GetType().Name, e.Message)
                   Return True
               End Function)
            End Try

'           Output:
'          n = 0
'          n = -1
'          Encountered ArgumentOutOfRangeException: Specified argument was out of the
range
'           of valid values.
'
```

This example demonstrates the case in which an exception goes unhandled in the delegate of an execution dataflow block. We recommend that you handle exceptions in the bodies of such blocks. However, if you are unable to do so, the block behaves as though it was canceled and does not process incoming messages.

When a dataflow block is canceled explicitly, the AggregateException object contains OperationCanceledException in the InnerExceptions property. For more information about dataflow cancellation, see Enabling Cancellation later in this document.

The second way to determine the completion status of a dataflow block is to use a continuation off of the completion task, or to use the asynchronous language features of C# and Visual Basic to asynchronously wait for the completion

task. The delegate that you provide to the Task.ContinueWith method takes a Task object that represents the antecedent task. In the case of the Completion property, the delegate for the continuation takes the completion task itself. The following example resembles the previous one, except that it also uses the ContinueWith method to create a completion task that prints the status of the overall dataflow operation.

VB

```
          ' Create an ActionBlock<int> object that prints its input
          ' and throws ArgumentOutOfRangeException if the input
          ' is less than zero.
          Dim throwIfNegative = New ActionBlock(Of Integer)(Sub(n)
              Console.WriteLine("n = {0}", n)
              If n < 0 Then
                  Throw New ArgumentOutOfRangeException()
              End If
          End Sub)

          ' Create a continuation task that prints the overall
          ' task status to the console when the block finishes.
          throwIfNegative.Completion.ContinueWith(Sub(task) Console.WriteLine("The
status of the completion task is '{0}'.", task.Status))

          ' Post values to the block.
          throwIfNegative.Post(0)
          throwIfNegative.Post(-1)
          throwIfNegative.Post(1)
          throwIfNegative.Post(-2)
          throwIfNegative.Complete()

          ' Wait for completion in a try/catch block.
          Try
             throwIfNegative.Completion.Wait()
          Catch ae As AggregateException
             ' If an unhandled exception occurs during dataflow processing, all
             ' exceptions are propagated through an AggregateException object.
             ae.Handle(Function(e)
                 Console.WriteLine("Encountered {0}: {1}", e.GetType().Name, e.Message)
                 Return True
             End Function)
          End Try

'          Output:
'         n = 0
'         n = -1
'         The status of the completion task is 'Faulted'.
'         Encountered ArgumentOutOfRangeException: Specified argument was out of the
range
'          of valid values.
'
```

You can also use properties such as IsCanceled in the body of the continuation task to determine additional information about the completion status of a dataflow block. For more information about continuation tasks and how they relate to cancellation and error handling, see Chaining Tasks by Using Continuation Tasks, Task Cancellation,

Exception Handling (Task Parallel Library), and NIB: How to: Handle Exceptions Thrown by Tasks.

[go to top]

Predefined Dataflow Block Types

The TPL Dataflow Library provides several predefined dataflow block types. These types are divided into three categories: *buffering blocks*, *execution blocks*, and *grouping blocks*. The following sections describe the block types that make up these categories.

Buffering Blocks

Buffering blocks hold data for use by data consumers. The TPL Dataflow Library provides three buffering block types: System.Threading.Tasks.Dataflow.BufferBlock(Of T), System.Threading.Tasks.Dataflow.BroadcastBlock(Of T), and System.Threading.Tasks.Dataflow.WriteOnceBlock(Of T).

BufferBlock(T)

The BufferBlock(Of T) class represents a general-purpose asynchronous messaging structure. This class stores a first in, first out (FIFO) queue of messages that can be written to by multiple sources or read from by multiple targets. When a target receives a message from a BufferBlock(Of T) object, that message is removed from the message queue. Therefore, although a BufferBlock(Of T) object can have multiple targets, only one target will receive each message. The BufferBlock(Of T) class is useful when you want to pass multiple messages to another component, and that component must receive each message.

The following basic example posts several Int32 values to a BufferBlock(Of T) object and then reads those values back from that object.

VB

```
' Create a BufferBlock<int> object.
Dim bufferBlock = New BufferBlock(Of Integer)()

' Post several messages to the block.
For i As Integer = 0 To 2
   bufferBlock.Post(i)
Next i

' Receive the messages back from the block.
For i As Integer = 0 To 2
   Console.WriteLine(bufferBlock.Receive())
Next i

'  Output:
'    0
'    1
'    2
'
```

For a complete example that demonstrates how to write messages to and read messages from a BufferBlock(Of T) object, see How to: Write Messages to and Read Messages from a Dataflow Block.

BroadcastBlock(T)

The BroadcastBlock(Of T) class is useful when you must pass multiple messages to another component, but that component needs only the most recent value. This class is also useful when you want to broadcast a message to multiple components.

The following basic example posts a Double value to a BroadcastBlock(Of T) object and then reads that value back from that object several times. Because values are not removed from BroadcastBlock(Of T) objects after they are read, the same value is available every time.

VB

```
' Create a BroadcastBlock<double> object.
Dim broadcastBlock = New BroadcastBlock(Of Double)(Nothing)

' Post a message to the block.
broadcastBlock.Post(Math.PI)

' Receive the messages back from the block several times.
For i As Integer = 0 To 2
   Console.WriteLine(broadcastBlock.Receive())
Next i

'          Output:
'            3.14159265358979
'            3.14159265358979
'            3.14159265358979
'
```

For a complete example that demonstrates how to use BroadcastBlock(Of T) to broadcast a message to multiple target blocks, see How to: Specify a Task Scheduler in a Dataflow Block.

WriteOnceBlock(T)

The WriteOnceBlock(Of T) class resembles the BroadcastBlock(Of T) class, except that a WriteOnceBlock(Of T) object can be written to one time only. You can think of WriteOnceBlock(Of T) as being similar to the C# readonly (ReadOnly in Visual Basic) keyword, except that a WriteOnceBlock(Of T) object becomes immutable after it receives a value instead of at construction. Like the BroadcastBlock(Of T) class, when a target receives a message from a WriteOnceBlock(Of T) object, that message is not removed from that object. Therefore, multiple targets receive a copy of the message. The WriteOnceBlock(Of T) class is useful when you want to propagate only the first of multiple messages.

The following basic example posts multiple String values to a WriteOnceBlock(Of T) object and then reads the value back from that object. Because a WriteOnceBlock(Of T) object can be written to one time only, after a WriteOnceBlock(Of T) object receives a message, it discards subsequent messages.

VB

```
        ' Create a WriteOnceBlock<string> object.
        Dim writeOnceBlock = New WriteOnceBlock(Of String)(Nothing)

        ' Post several messages to the block in parallel. The first
        ' message to be received is written to the block.
        ' Subsequent messages are discarded.
        Parallel.Invoke(Function() writeOnceBlock.Post("Message 1"), Function()
writeOnceBlock.Post("Message 2"), Function() writeOnceBlock.Post("Message 3"))

        ' Receive the message from the block.
        Console.WriteLine(writeOnceBlock.Receive())

'          Sample output:
'            Message 2
'
```

For a complete example that demonstrates how to use WriteOnceBlock(Of T) to receive the value of the first operation that finishes, see How to: Unlink Dataflow Blocks.

Execution Blocks

Execution blocks call a user-provided delegate for each piece of received data. The TPL Dataflow Library provides three execution block types: ActionBlock(Of TInput), System.Threading.Tasks.Dataflow.TransformBlock(Of TInput, TOutput), and System.Threading.Tasks.Dataflow.TransformManyBlock(Of TInput, TOutput).

ActionBlock(T)

The ActionBlock(Of TInput) class is a target block that calls a delegate when it receives data. Think of a ActionBlock(Of TInput) object as a delegate that runs asynchronously when data becomes available. The delegate that you provide to an ActionBlock(Of TInput) object can be of type Action or type **System.Func<TInput, Task>**. When you use an ActionBlock(Of TInput) object with Action, processing of each input element is considered completed when the delegate returns. When you use an ActionBlock(Of TInput) object with **System.Func<TInput, Task>**, processing of each input element is considered completed only when the returned Task object is completed. By using these two mechanisms, you can use ActionBlock(Of TInput) for both synchronous and asynchronous processing of each input element.

The following basic example posts multiple Int32 values to an ActionBlock(Of TInput) object. The ActionBlock(Of TInput) object prints those values to the console. This example then sets the block to the completed state and waits for all dataflow tasks to finish.

VB

```
        ' Create an ActionBlock<int> object that prints values
        ' to the console.
        Dim actionBlock = New ActionBlock(Of Integer)(Function(n) WriteLine(n))

        ' Post several messages to the block.
        For i As Integer = 0 To 2
           actionBlock.Post(i * 10)
```

```
            Next i

            ' Set the block to the completed state and wait for all
            ' tasks to finish.
            actionBlock.Complete()
            actionBlock.Completion.Wait()

'           Output:
'             0
'             10
'             20
'
```

For complete examples that demonstrate how to use delegates with the ActionBlock(Of TInput) class, see How to: Perform Action When a Dataflow Block Receives Data.

TransformBlock(TInput, TOutput)

The TransformBlock(Of TInput, TOutput) class resembles the ActionBlock(Of TInput) class, except that it acts as both a source and as a target. The delegate that you pass to a TransformBlock(Of TInput, TOutput) object returns a value of type *TOutput*. The delegate that you provide to a TransformBlock(Of TInput, TOutput) object can be of type **System.Func<TInput, TOutput>** or type **System.Func<TInput, Task>**. When you use a TransformBlock(Of TInput, TOutput) object with **System.Func<TInput, TOutput>**, processing of each input element is considered completed when the delegate returns. When you use a TransformBlock(Of TInput, TOutput) object used with **System.Func<TInput, Task<TOutput>>**, processing of each input element is considered completed only when the returned Task object is completed. As with ActionBlock(Of TInput), by using these two mechanisms, you can use TransformBlock(Of TInput, TOutput) for both synchronous and asynchronous processing of each input element.

The following basic example creates a TransformBlock(Of TInput, TOutput) object that computes the square root of its input. The TransformBlock(Of TInput, TOutput) object takes Int32 values as input and produces Double values as output.

VB

```
            ' Create a TransformBlock<int, double> object that
            ' computes the square root of its input.
            Dim transformBlock = New TransformBlock(Of Integer, Double)(Function(n)
  Math.Sqrt(n))

            ' Post several messages to the block.
            transformBlock.Post(10)
            transformBlock.Post(20)
            transformBlock.Post(30)

            ' Read the output messages from the block.
            For i As Integer = 0 To 2
               Console.WriteLine(transformBlock.Receive())
            Next i

'           Output:
'             3.16227766016838
'             4.47213595499958
```

```
'            5.47722557505166
'
```

For complete examples that uses TransformBlock(Of TInput, TOutput) in a network of dataflow blocks that performs image processing in a Windows Forms application, see Walkthrough: Using Dataflow in a Windows Forms Application.

TransformManyBlock(TInput, TOutput)

The TransformManyBlock(Of TInput, TOutput) class resembles the TransformBlock(Of TInput, TOutput) class, except that TransformManyBlock(Of TInput, TOutput) produces zero or more output values for each input value, instead of only one output value for each input value. The delegate that you provide to a TransformManyBlock(Of TInput, TOutput) object can be of type **System.Func<TInput, IEnumerable<TOutput>>** or **type System.Func<TInput, Task<IEnumerable<TOutput>>>**. When you use a TransformManyBlock(Of TInput, TOutput) object with **System.Func<TInput, IEnumerable<TOutput>>**, processing of each input element is considered completed when the delegate returns. When you use a TransformManyBlock(Of TInput, TOutput) object with **System.Func<TInput, Task<IEnumerable<TOutput>>>**, processing of each input element is considered complete only when the returned **System.Threading.Tasks.Task<IEnumerable<TOutput>>** object is completed.

The following basic example creates a TransformManyBlock(Of TInput, TOutput) object that splits strings into their individual character sequences. The TransformManyBlock(Of TInput, TOutput) object takes String values as input and produces Char values as output.

VB

```
         ' Create a TransformManyBlock<string, char> object that splits
         ' a string into its individual characters.
         Dim transformManyBlock = New TransformManyBlock(Of String,
Char)(Function(s) s.ToCharArray())

         ' Post two messages to the first block.
         transformManyBlock.Post("Hello")
         transformManyBlock.Post("World")

         ' Receive all output values from the block.
         For i As Integer = 0 To ("Hello" & "World").Length - 1
            Console.WriteLine(transformManyBlock.Receive())
         Next i

'          Output:
'            H
'            e
'            l
'            l
'            o
'            W
'            o
'            r
'            l
'            d
'
```

For complete examples that use TransformManyBlock(Of TInput, TOutput) to produce multiple independent outputs for each input in a dataflow pipeline, see Walkthrough: Creating a Dataflow Pipeline.

Degree of Parallelism

Every ActionBlock(Of TInput), TransformBlock(Of TInput, TOutput), and TransformManyBlock(Of TInput, TOutput) object buffers input messages until the block is ready to process them. By default, these classes process messages in the order in which they are received, one message at a time. You can also specify the degree of parallelism to enable ActionBlock(Of TInput), TransformBlock(Of TInput, TOutput) and TransformManyBlock(Of TInput, TOutput) objects to process multiple messages concurrently. For more information about concurrent execution, see the section Specifying the Degree of Parallelism later in this document. For an example that sets the degree of parallelism to enable an execution dataflow block to process more than one message at a time, see How to: Specify the Degree of Parallelism in a Dataflow Block.

Summary of Delegate Types

The following table summarizes the delegate types that you can provide to ActionBlock(Of TInput), TransformBlock(Of TInput, TOutput), and TransformManyBlock(Of TInput, TOutput) objects. This table also specifies whether the delegate type operates synchronously or asynchronously.

Type	Synchronous Delegate Type	Asynchronous Delegate Type
ActionBlock(Of TInput)	**System.Action**	**System.Func<TInput, Task>**
TransformBlock(Of TInput, TOutput)	**System.Func<TInput, TOutput>`2**	**System.Func<TInput, Task<TOutput>>**
TransformManyBlock(Of TInput, TOutput)	**System.Func<TInput, IEnumerable<TOutput>>**	**System.Func<TInput, Task<IEnumerable<TOutput>>>**

You can also use lambda expressions when you work with execution block types. For an example that shows how to use a lambda expression with an execution block, see How to: Perform Action When a Dataflow Block Receives Data.

Grouping Blocks

Grouping blocks combine data from one or more sources and under various constraints. The TPL Dataflow Library provides three join block types: BatchBlock(Of T), JoinBlock(Of T1, T2), and BatchedJoinBlock(Of T1, T2).

BatchBlock(T)

The BatchBlock(Of T) class combines sets of input data, which are known as batches, into arrays of output data. You specify the size of each batch when you create a BatchBlock(Of T) object. When the BatchBlock(Of T) object receives the specified count of input elements, it asynchronously propagates out an array that contains those elements. If a BatchBlock(Of T) object is set to the completed state but does not contain enough elements to form a batch, it

propagates out a final array that contains the remaining input elements.

The BatchBlock(Of T) class operates in either *greedy* or *non-greedy* mode. In greedy mode, which is the default, a BatchBlock(Of T) object accepts every message that it is offered and propagates out an array after it receives the specified count of elements. In non-greedy mode, a BatchBlock(Of T) object postpones all incoming messages until enough sources have offered messages to the block to form a batch. Greedy mode typically performs better than non-greedy mode because it requires less processing overhead. However, you can use non-greedy mode when you must coordinate consumption from multiple sources in an atomic fashion. Specify non-greedy mode by setting Greedy to **False** in the *dataflowBlockOptions* parameter in the BatchBlock(Of T) constructor.

The following basic example posts several Int32 values to a BatchBlock(Of T) object that holds ten elements in a batch. To guarantee that all values propagate out of the BatchBlock(Of T), this example calls the Complete method. The Complete method sets the BatchBlock(Of T) object to the completed state, and therefore, the BatchBlock(Of T) object propagates out any remaining elements as a final batch.

VB

```
          ' Create a BatchBlock<int> object that holds ten
          ' elements per batch.
          Dim batchBlock = New BatchBlock(Of Integer)(10)

          ' Post several values to the block.
          For i As Integer = 0 To 12
             batchBlock.Post(i)
          Next i
          ' Set the block to the completed state. This causes
          ' the block to propagate out any any remaining
          ' values as a final batch.
          batchBlock.Complete()

          ' Print the sum of both batches.

          Console.WriteLine("The sum of the elements in batch 1 is {0}.",
batchBlock.Receive().Sum())

          Console.WriteLine("The sum of the elements in batch 2 is {0}.",
batchBlock.Receive().Sum())

'           Output:
'             The sum of the elements in batch 1 is 45.
'             The sum of the elements in batch 2 is 33.
'
```

For a complete example that uses BatchBlock(Of T) to improve the efficiency of database insert operations, see Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency.

JoinBlock(T1, T2, ...)

The JoinBlock(Of T1, T2) and JoinBlock(Of T1, T2, T3) classes collect input elements and propagate out System.Tuple(Of T1, T2) or System.Tuple(Of T1, T2, T3) objects that contain those elements. The JoinBlock(Of T1, T2) and JoinBlock(Of T1, T2, T3) classes do not inherit from ITargetBlock(Of TInput). Instead, they provide

properties, Target1, Target2, and Target3, that implement ITargetBlock(Of TInput).

Like BatchBlock(Of T), JoinBlock(Of T1, T2) and JoinBlock(Of T1, T2, T3) operate in either greedy or non-greedy mode. In greedy mode, which is the default, a JoinBlock(Of T1, T2) or JoinBlock(Of T1, T2, T3) object accepts every message that it is offered and propagates out a tuple after each of its targets receives at least one message. In non-greedy mode, a JoinBlock(Of T1, T2) or JoinBlock(Of T1, T2, T3) object postpones all incoming messages until all targets have been offered the data that is required to create a tuple. At this point, the block engages in a two-phase commit protocol to atomically retrieve all required items from the sources. This postponement makes it possible for another entity to consume the data in the meantime, to allow the overall system to make forward progress.

The following basic example demonstrates a case in which a JoinBlock(Of T1, T2, T3) object requires multiple data to compute a value. This example creates a JoinBlock(Of T1, T2, T3) object that requires two Int32 values and a Char value to perform an arithmetic operation.

VB

```
          ' Create a JoinBlock<int, int, char> object that requires
          ' two numbers and an operator.
          Dim joinBlock = New JoinBlock(Of Integer, Integer, Char)()

          ' Post two values to each target of the join.

          joinBlock.Target1.Post(3)
          joinBlock.Target1.Post(6)

          joinBlock.Target2.Post(5)
          joinBlock.Target2.Post(4)

          joinBlock.Target3.Post("+"c)
          joinBlock.Target3.Post("-"c)

          ' Receive each group of values and apply the operator part
          ' to the number parts.

          For i As Integer = 0 To 1
             Dim data = joinBlock.Receive()
             Select Case data.Item3
                Case "+"c
                   Console.WriteLine("{0} + {1} = {2}", data.Item1, data.Item2,
data.Item1 + data.Item2)
                Case "-"c
                   Console.WriteLine("{0} - {1} = {2}", data.Item1, data.Item2,
data.Item1 - data.Item2)
                Case Else
                   Console.WriteLine("Unknown operator '{0}'.", data.Item3)
             End Select
          Next i

'          Output:
'            3 + 5 = 8
'            6 - 4 = 2
'
```

For a complete example that uses JoinBlock(Of T1, T2) objects in non-greedy mode to cooperatively share a resource, see How to: Use JoinBlock to Read Data From Multiple Sources.

BatchedJoinBlock(T1, T2, ...)

The BatchedJoinBlock(Of T1, T2) and BatchedJoinBlock(Of T1, T2, T3) classes collect batches of input elements and propagate out **System.Tuple(IList(T1), IList(T2))** or **System.Tuple(IList(T1), IList(T2), IList(T3))** objects that contain those elements. Think of BatchedJoinBlock(Of T1, T2) as a combination of BatchBlock(Of T) and JoinBlock(Of T1, T2). Specify the size of each batch when you create a BatchedJoinBlock(Of T1, T2) object. BatchedJoinBlock(Of T1, T2) also provides properties, Target1 and Target2, that implement ITargetBlock(Of TInput). When the specified count of input elements are received from across all targets, the BatchedJoinBlock(Of T1, T2) object asynchronously propagates out a **System.Tuple(IList(T1), IList(T2))** object that contains those elements.

The following basic example creates a BatchedJoinBlock(Of T1, T2) object that holds results, Int32 values, and errors that are Exception objects. This example performs multiple operations and writes results to the Target1 property, and errors to the Target2 property, of the BatchedJoinBlock(Of T1, T2) object. Because the count of successful and failed operations is unknown in advance, the IList(Of T) objects enable each target to receive zero or more values.

VB

```
' For demonstration, create a Func<int, int> that
' returns its argument, or throws ArgumentOutOfRangeException
' if the argument is less than zero.
Dim DoWork As Func(Of Integer, Integer) = Function(n)
   If n < 0 Then
      Throw New ArgumentOutOfRangeException()
   End If
   Return n
End Function

' Create a BatchedJoinBlock<int, Exception> object that holds
' seven elements per batch.
Dim batchedJoinBlock = New BatchedJoinBlock(Of Integer, Exception)(7)

' Post several items to the block.
For Each i As Integer In New Integer() { 5, 6, -7, -22, 13, 55, 0 }
   Try
      ' Post the result of the worker to the
      ' first target of the block.
      batchedJoinBlock.Target1.Post(DoWork(i))
   Catch e As ArgumentOutOfRangeException
      ' If an error occurred, post the Exception to the
      ' second target of the block.
      batchedJoinBlock.Target2.Post(e)
   End Try
Next i

' Read the results from the block.
Dim results = batchedJoinBlock.Receive()

' Print the results to the console.
```

```
            ' Print the results.
            For Each n As Integer In results.Item1
               Console.WriteLine(n)
            Next n
            ' Print failures.
            For Each e As Exception In results.Item2
               Console.WriteLine(e.Message)
            Next e

'           Output:
'             5
'             6
'             13
'             55
'             0
'             Specified argument was out of the range of valid values.
'             Specified argument was out of the range of valid values.
'
```

For a complete example that uses BatchedJoinBlock(Of T1, T2) to capture both the results and any exceptions that occur while the program reads from a database, see Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency.

[go to top]

Configuring Dataflow Block Behavior

You can enable additional options by providing a System.Threading.Tasks.Dataflow.DataflowBlockOptions object to the constructor of dataflow block types. These options control behavior such the scheduler that manages the underlying task and the degree of parallelism. The DataflowBlockOptions also has derived types that specify behavior that is specific to certain dataflow block types. The following table summarizes which options type is associated with each dataflow block type.

Dataflow Block Type	**DataflowBlockOptions type**
BufferBlock(Of T)	DataflowBlockOptions
BroadcastBlock(Of T)	DataflowBlockOptions
WriteOnceBlock(Of T)	DataflowBlockOptions
ActionBlock(Of TInput)	ExecutionDataflowBlockOptions
TransformBlock(Of TInput, TOutput)	ExecutionDataflowBlockOptions
TransformManyBlock(Of TInput, TOutput)	ExecutionDataflowBlockOptions

BatchBlock(Of T)	GroupingDataflowBlockOptions
JoinBlock(Of T1, T2)	GroupingDataflowBlockOptions
BatchedJoinBlock(Of T1, T2)	GroupingDataflowBlockOptions

The following sections provide additional information about the important kinds of dataflow block options that are available through the System.Threading.Tasks.Dataflow.DataflowBlockOptions, System.Threading.Tasks.Dataflow.ExecutionDataflowBlockOptions, and System.Threading.Tasks.Dataflow.GroupingDataflowBlockOptions classes.

Specifying the Task Scheduler

Every predefined dataflow block uses the TPL task scheduling mechanism to perform activities such as propagating data to a target, receiving data from a source, and running user-defined delegates when data becomes available. TaskScheduler is an abstract class that represents a task scheduler that queues tasks onto threads. The default task scheduler, Default, uses the ThreadPool class to queue and execute work. You can override the default task scheduler by setting the TaskScheduler property when you construct a dataflow block object.

When the same task scheduler manages multiple dataflow blocks, it can enforce policies across them. For example, if multiple dataflow blocks are each configured to target the exclusive scheduler of the same ConcurrentExclusiveSchedulerPair object, all work that runs across these blocks is serialized. Similarly, if these blocks are configured to target the concurrent scheduler of the same ConcurrentExclusiveSchedulerPair object, and that scheduler is configured to have a maximum concurrency level, all work from these blocks is limited to that number of concurrent operations. For an example that uses the ConcurrentExclusiveSchedulerPair class to enable read operations to occur in parallel, but write operations to occur exclusively of all other operations, see How to: Specify a Task Scheduler in a Dataflow Block. For more information about task schedulers in the TPL, see the TaskScheduler class topic.

Specifying the Degree of Parallelism

By default, the three execution block types that the TPL Dataflow Library provides, ActionBlock(Of TInput), TransformBlock(Of TInput, TOutput), and TransformManyBlock(Of TInput, TOutput), process one message at a time. These dataflow block types also process messages in the order in which they are received. To enable these dataflow blocks to process messages concurrently, set the ExecutionDataflowBlockOptions.MaxDegreeOfParallelism property when you construct the dataflow block object.

The default value of MaxDegreeOfParallelism is 1, which guarantees that the dataflow block processes one message at a time. Setting this property to a value that is larger than 1 enables the dataflow block to process multiple messages concurrently. Setting this property to DataflowBlockOptions.Unbounded enables the underlying task scheduler to manage the maximum degree of concurrency.

Important

When you specify a maximum degree of parallelism that is larger than 1, multiple messages are processed simultaneously, and therefore, messages might not be processed in the order in which they are received. The order in which the messages are output from the block will, however, be correctly ordered.

Because the MaxDegreeOfParallelism property represents the maximum degree of parallelism, the dataflow block might execute with a lesser degree of parallelism than you specify. The dataflow block might use a lesser degree of parallelism to meet its functional requirements or because there is a lack of available system resources. A dataflow block never chooses more parallelism than you specify.

The value of the MaxDegreeOfParallelism property is exclusive to each dataflow block object. For example, if four dataflow block objects each specify 1 for the maximum degree of parallelism, all four dataflow block objects can potentially run in parallel.

For an example that sets the maximum degree of parallelism to enable lengthy operations to occur in parallel, see How to: Specify the Degree of Parallelism in a Dataflow Block.

Specifying the Number of Messages per Task

The predefined dataflow block types use tasks to process multiple input elements. This helps minimize the number of task objects that are required to process data, which enables applications to run more efficiently. However, when the tasks from one set of dataflow blocks are processing data, the tasks from other dataflow blocks might need to wait for processing time by queuing messages. To enable better fairness among dataflow tasks, set the MaxMessagesPerTask property. When MaxMessagesPerTask is set to DataflowBlockOptions.Unbounded, which is the default, the task used by a dataflow block processes as many messages as are available. When MaxMessagesPerTask is set to a value other than Unbounded, the dataflow block processes at most this number of messages per Task object. Although setting the MaxMessagesPerTask property can increase fairness among tasks, it can cause the system to create more tasks than are necessary, which can decrease performance.

Enabling Cancellation

The TPL provides a mechanism that enables tasks to coordinate cancellation in a cooperative manner. To enable dataflow blocks to participate in this cancellation mechanism, set the CancellationToken property. When this CancellationToken object is set to the canceled state, all dataflow blocks that monitor this token finish execution of their current item but do not start processing subsequent items. These dataflow blocks also clear any buffered messages, release connections to any source and target blocks, and transition to the canceled state. By transitioning to the canceled state, the Completion property has the Status property set to Canceled, unless an exception occurred during processing. In that case, Status is set to Faulted.

For an example that demonstrates how to use cancellation in a Windows Forms application, see How to: Cancel a Dataflow Block. For more information about cancellation in the TPL, see Task Cancellation.

Specifying Greedy Versus Non-Greedy Behavior

Several grouping dataflow block types can operate in either *greedy* or *non-greedy* mode. By default, the predefined dataflow block types operate in greedy mode.

For join block types such as JoinBlock(Of T1, T2), greedy mode means that the block immediately accepts data even if the corresponding data with which to join is not yet available. Non-greedy mode means that the block postpones all incoming messages until one is available on each of its targets to complete the join. If any of the postponed messages are no longer available, the join block releases all postponed messages and restarts the process. For the BatchBlock(Of T) class, greedy and non-greedy behavior is similar, except that under non-greedy mode, a BatchBlock(Of T) object postpones all incoming messages until enough are available from distinct sources to complete a batch.

To specify non-greedy mode for a dataflow block, set Greedy to **False**. For an example that demonstrates how to use non-greedy mode to enable multiple join blocks to share a data source more efficiently, see How to: Use JoinBlock to Read Data From Multiple Sources.

[go to top]

Custom Dataflow Blocks

Although the TPL Dataflow Library provides many predefined block types, you can create additional block types that perform custom behavior. Implement the ISourceBlock(Of TOutput) or ITargetBlock(Of TInput) interfaces directly or use the Encapsulate(Of TInput, TOutput) method to build a complex block that encapsulates the behavior of existing block types. For examples that show how to implement custom dataflow block functionality, see Walkthrough: Creating a Custom Dataflow Block Type.

[go to top]

Related Topics

Title	Description
How to: Write Messages to and Read Messages from a Dataflow Block	Demonstrates how to write messages to and read messages from a BufferBlock(Of T) object.
How to: Implement a Producer-Consumer Dataflow Pattern	Describes how to use the dataflow model to implement a producer-consumer pattern, where the producer sends messages to a dataflow block, and the consumer reads messages from that block.
How to: Perform Action When a Dataflow Block Receives Data	Describes how to provide delegates to the execution dataflow block types, ActionBlock(Of TInput), TransformBlock(Of TInput, TOutput), and TransformManyBlock(Of TInput, TOutput).
Walkthrough: Creating a Dataflow Pipeline	Describes how to create a dataflow pipeline that downloads text from the web and performs operations on that text.
How to: Unlink Dataflow Blocks	Demonstrates how to use the LinkTo method to unlink a target block from its source after the source offers a message to the target.
Walkthrough: Using Dataflow in a Windows Forms Application	Demonstrates how to create a network of dataflow blocks that perform image processing in a Windows Forms application.
How to: Cancel a Dataflow Block	Demonstrates how to use cancellation in a Windows Forms application.

How to: Use JoinBlock to Read Data From Multiple Sources	Explains how to use the JoinBlock(Of T1, T2) class to perform an operation when data is available from multiple sources, and how to use non-greedy mode to enable multiple join blocks to share a data source more efficiently.
How to: Specify the Degree of Parallelism in a Dataflow Block	Describes how to set the MaxDegreeOfParallelism property to enable an execution dataflow block to process more than one message at a time.
How to: Specify a Task Scheduler in a Dataflow Block	Demonstrates how to associate a specific task scheduler when you use dataflow in your application.
Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency	Describes how to use the BatchBlock(Of T) class to improve the efficiency of database insert operations, and how to use the BatchedJoinBlock(Of T1, T2) class to capture both the results and any exceptions that occur while the program reads from a database.
Walkthrough: Creating a Custom Dataflow Block Type	Demonstrates two ways to create a dataflow block type that implements custom behavior.
Task Parallel Library (TPL)	Introduces the TPL, a library that simplifies parallel and concurrent programming in .NET Framework applications.

Using TPL with Other Asynchronous Patterns

.NET Framework (current version)

The Task Parallel Library can be used with traditional .NET Framework asynchronous programming patterns in various ways.

In This Section

TPL and Traditional .NET Framework Asynchronous Programming
: Describes how Task objects may be used in conjunction with the Asynchronous Programming Model (APM) and the Event-based Asynchronous Pattern (EAP).

How to: Wrap EAP Patterns in a Task
: Shows how to use Task objects to encapsulate EAP patterns.

See Also

Task Parallel Library (TPL)

TPL and Traditional .NET Framework Asynchronous Programming

.NET Framework (current version)

The .NET Framework provides the following two standard patterns for performing I/O-bound and compute-bound asynchronous operations:

- Asynchronous Programming Model (APM), in which asynchronous operations are represented by a pair of Begin/End methods such as FileStream.BeginRead and Stream.EndRead.
- Event-based asynchronous pattern (EAP), in which asynchronous operations are represented by a method/event pair that are named *OperationName*Async and *OperationName*Completed, for example, WebClient.DownloadStringAsync and WebClient.DownloadStringCompleted. (EAP was introduced in the .NET Framework version 2.0.)

The Task Parallel Library (TPL) can be used in various ways in conjunction with either of the asynchronous patterns. You can expose both APM and EAP operations as Tasks to library consumers, or you can expose the APM patterns but use Task objects to implement them internally. In both scenarios, by using Task objects, you can simplify the code and take advantage of the following useful functionality:

- Register callbacks, in the form of task continuations, at any time after the task has started.
- Coordinate multiple operations that execute in response to a **Begin_** method, by using the ContinueWhenAll and ContinueWhenAny methods, or the WaitAll method or the WaitAny method.
- Encapsulate asynchronous I/O-bound and compute-bound operations in the same Task object.
- Monitor the status of the Task object.
- Marshal the status of an operation to a Task object by using TaskCompletionSource(Of TResult).

Wrapping APM Operations in a Task

Both the System.Threading.Tasks.TaskFactory and System.Threading.Tasks.TaskFactory(Of TResult) classes provide several overloads of the TaskFactory.FromAsync and TaskFactory(Of TResult).FromAsync methods that let you encapsulate an APM Begin/End method pair in one Task or Task(Of TResult) instance. The various overloads accommodate any Begin/End method pair that have from zero to three input parameters.

For pairs that have **End** methods that return a value (**Function** in Visual Basic), use the methods in TaskFactory(Of TResult) that create a Task(Of TResult). For **End** methods that return void (**Sub** in Visual Basic), use the methods in TaskFactory that create a Task.

For those few cases in which the **Begin** method has more than three parameters or contains *ref* or *out* parameters, additional **FromAsync** overloads that encapsulate only the **End** method are provided.

The following example shows the signature for the **FromAsync** overload that matches the FileStream.BeginRead and

FileStream.EndRead methods. This overload takes three input parameters, as follows.

VB

```
Public Function FromAsync(Of TArg1, TArg2, TArg3)(
                ByVal beginMethod As Func(Of TArg1, TArg2, TArg3, AsyncCallback,
Object, IAsyncResult),
                ByVal endMethod As Func(Of IAsyncResult, TResult),
                ByVal dataBuffer As TArg1,
                ByVal byteOffsetToStartAt As TArg2,
                ByVal maxBytesToRead As TArg3,
                ByVal stateInfo As Object)
```

The first parameter is a Func(Of T1, T2, T3, T4, T5, TResult) delegate that matches the signature of the FileStream.BeginRead method. The second parameter is a Func(Of T, TResult) delegate that takes an IAsyncResult and returns a *TResult*. Because EndRead returns an integer, the compiler infers the type of **TResult** as Int32 and the type of the task as Task. The last four parameters are identical to those in the FileStream.BeginRead method:

- The buffer in which to store the file data.
- The offset in the buffer at which to begin writing data.
- The maximum amount of data to read from the file.
- An optional object that stores user-defined state data to pass to the callback.

Using ContinueWith for the Callback Functionality

If you require access to the data in the file, as opposed to just the number of bytes, the FromAsync method is not sufficient. Instead, use Task, whose **Result** property contains the file data. You can do this by adding a continuation to the original task. The continuation performs the work that would typically be performed by the AsyncCallback delegate. It is invoked when the antecedent completes, and the data buffer has been filled. (The FileStream object should be closed before returning.)

The following example shows how to return a Task that encapsulates the BeginRead/EndRead pair of the FileStream class.

VB

```
Const MAX_FILE_SIZE As Integer = 14000000
Shared Function GetFileStringAsync(ByVal path As String) As Task(Of String)
    Dim fi As New FileInfo(path)
    Dim data(fi.Length) As Byte

    Dim fs As FileStream = New FileStream(path, FileMode.Open, FileAccess.Read,
FileShare.Read, data.Length, True)

    ' Task(Of Integer) returns the number of bytes read
    Dim myTask As Task(Of Integer) = Task(Of Integer).Factory.FromAsync(
        AddressOf fs.BeginRead, AddressOf fs.EndRead, data, 0, data.Length, Nothing)

    ' It is possible to do other work here while waiting
```

```
    ' for the antecedent task to complete.
    ' ...

    ' Add the continuation, which returns a Task<string>.
    Return myTask.ContinueWith(Function(antecedent)
                                   fs.Close()
                                   If (antecedent.Result < 100) Then
                                       Return "Data is too small to bother with."
                                   End If
                                   ' If we did not receive the entire file, the end
of the
                                   ' data buffer will contain garbage.
                                   If (antecedent.Result < data.Length) Then
                                       Array.Resize(data, antecedent.Result)
                                   End If

                                   ' Will be returned in the Result property of the
Task<string>
                                   ' at some future point after the asynchronous file
I/O operation completes.
                                   Return New UTF8Encoding().GetString(data)
                               End Function)

End Function
```

The method can then be called, as follows.

VB

```
Dim myTask As Task(Of String) = GetFileStringAsync(path)

' Do some other work
' ...

Try
    Console.WriteLine(myTask.Result.Substring(0, 500))
Catch ex As AggregateException
    Console.WriteLine(ex.InnerException.Message)
End Try
```

Providing Custom State Data

In typical IAsyncResult operations, if your AsyncCallback delegate requires some custom state data, you have to pass it in through the last parameter in the **Begin** method, so that the data can be packaged into the IAsyncResult object that is eventually passed to the callback method. This is typically not required when the **FromAsync** methods are used. If the custom data is known to the continuation, then it can be captured directly in the continuation delegate. The following example resembles the previous example, but instead of examining the **Result** property of the antecedent, the continuation examines the custom state data that is directly accessible to the user delegate of the continuation.

VB

```
Public Function GetFileStringAsync2(ByVal path As String) As Task(Of String)
    Dim fi = New FileInfo(path)
    Dim data(fi.Length) As Byte
    Dim state As New MyCustomState()

    Dim fs As New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read,
data.Length, True)
    ' We still pass null for the last parameter because
    ' the state variable is visible to the continuation delegate.
    Dim myTask As Task(Of Integer) = Task(Of Integer).Factory.FromAsync(
            AddressOf fs.BeginRead, AddressOf fs.EndRead, data, 0, data.Length,
Nothing)

    Return myTask.ContinueWith(Function(antecedent)
                                   fs.Close()
                                   ' Capture custom state data directly in the user
delegate.
                                   ' No need to pass it through the FromAsync method.
                                   If (state.StateData.Contains("New York, New
York")) Then
                                       Return "Start spreading the news!"
                                   End If

                                   ' If we did not receive the entire file, the end
of the
                                   ' data buffer will contain garbage.
                                   If (antecedent.Result < data.Length) Then
                                       Array.Resize(data, antecedent.Result)
                                   End If
                                   '/ Will be returned in the Result property of the
Task<string>
                                   '/ at some future point after the asynchronous
file I/O operation completes.
                                   Return New UTF8Encoding().GetString(data)
                               End Function)

End Function
```

Synchronizing Multiple FromAsync Tasks

The static ContinueWhenAll and ContinueWhenAny methods provide added flexibility when used in conjunction with the **FromAsync** methods. The following example shows how to initiate multiple asynchronous I/O operations, and then wait for all of them to complete before you execute the continuation.

VB

```
Public Function GetMultiFileData(ByVal filesToRead As String()) As Task(Of String)
    Dim fs As FileStream
    Dim tasks(filesToRead.Length) As Task(Of String)
    Dim fileData() As Byte = Nothing
    For i As Integer = 0 To filesToRead.Length
```

```
        fileData(&H1000) = New Byte()
        fs = New FileStream(filesToRead(i), FileMode.Open, FileAccess.Read,
FileShare.Read, fileData.Length, True)

        ' By adding the continuation here, the
        ' Result of each task will be a string.
        tasks(i) = Task(Of Integer).Factory.FromAsync(AddressOf fs.BeginRead,
                                                      AddressOf fs.EndRead,
                                                      fileData,
                                                      0,
                                                      fileData.Length,
                                                      Nothing).
                                                  ContinueWith(Function(antecedent)
                                                                   fs.Close()
                                                                   'If we did not
receive the entire file, the end of the
                                                                   ' data buffer will
contain garbage.
                                                                   If
(antecedent.Result < fileData.Length) Then
                                                                       ReDim Preserve
fileData(antecedent.Result)
                                                                   End If

                                                                   'Will be returned
in the Result property of the Task<string>
                                                                   ' at some future
point after the asynchronous file I/O operation completes.
                                                                   Return New
UTF8Encoding().GetString(fileData)
                                                               End Function)
    Next

    Return Task(Of String).Factory.ContinueWhenAll(tasks, Function(data)

                                                           ' Propagate all
exceptions and mark all faulted tasks as observed.
                                                           Task.WaitAll(data)


                                                           ' Combine the results
from all tasks.
                                                           Dim sb As New
StringBuilder()
                                                           For Each t As Task(Of
String) In data
                                                               sb.Append(t.Result)
                                                           Next
                                                           ' Final result to be
returned eventually on the calling thread.
                                                           Return sb.ToString()
                                                       End Function)
End Function
```

FromAsync Tasks For Only the End Method

For those few cases in which the **Begin** method requires more than three input parameters, or has *ref* or *out* parameters, you can use the **FromAsync** overloads, for example, TaskFactory(Of TResult).FromAsync(IAsyncResult, Func(Of IAsyncResult, TResult)), that represent only the **End** method. These methods can also be used in any scenario in which you are passed an IAsyncResult and want to encapsulate it in a Task.

VB

```
Shared Function ReturnTaskFromAsyncResult() As Task(Of String)
    Dim ar As IAsyncResult = DoSomethingAsynchronously()
    Dim t As Task(Of String) = Task(Of String).Factory.FromAsync(ar, Function(res)
CStr(res.AsyncState))
    Return t
End Function
```

Starting and Canceling FromAsync Tasks

The task returned by a **FromAsync** method has a status of WaitingForActivation and will be started by the system at some point after the task is created. If you attempt to call Start on such a task, an exception will be raised.

You cannot cancel a **FromAsync** task, because the underlying .NET Framework APIs currently do not support in-progress cancellation of file or network I/O. You can add cancellation functionality to a method that encapsulates a **FromAsync** call, but you can only respond to the cancellation before **FromAsync** is called or after it completed (for example, in a continuation task).

Some classes that support EAP, for example, WebClient, do support cancellation, and you can integrate that native cancellation functionality by using cancellation tokens.

Exposing Complex EAP Operations As Tasks

The TPL does not provide any methods that are specifically designed to encapsulate an event-based asynchronous operation in the same way that the **FromAsync** family of methods wrap the IAsyncResult pattern. However, the TPL does provide the System.Threading.Tasks.TaskCompletionSource(Of TResult) class, which can be used to represent any arbitrary set of operations as a Task(Of TResult). The operations may be synchronous or asynchronous, and may be I/O bound or compute-bound, or both.

The following example shows how to use a TaskCompletionSource(Of TResult) to expose a set of asynchronous WebClient operations to client code as a basic Task(Of TResult). The method lets you enter an array of Web URLs, and a term or name to search for, and then returns the number of times the search term occurs on each site.

VB

```
Imports System.Collections.Generic
Imports System.Net
Imports System.Threading
Imports System.Threading.Tasks
```

```
Public Class SimpleWebExample
   Dim tcs As New TaskCompletionSource(Of String())
   Dim token As CancellationToken
   Dim results As New List(Of String)
   Dim m_lock As New Object()
   Dim count As Integer
   Dim addresses() As String
   Dim nameToSearch As String

   Public Function GetWordCountsSimplified(ByVal urls() As String, ByVal str As String,
                                           ByVal token As CancellationToken) As Task(Of
String())
      addresses = urls
      nameToSearch = str

      Dim webClients(urls.Length - 1) As WebClient

      ' If the user cancels the CancellationToken, then we can use the
      ' WebClient's ability to cancel its own async operations.
      token.Register(Sub()
                        For Each wc As WebClient In webClients
                           If wc IsNot Nothing Then
                              wc.CancelAsync()
                           End If
                        Next
                     End Sub)

      For i As Integer = 0 To urls.Length - 1
         webClients(i) = New WebClient()

         ' Specify the callback for the DownloadStringCompleted
         ' event that will be raised by this WebClient instance.
         AddHandler webClients(i).DownloadStringCompleted, AddressOf WebEventHandler

         Dim address As New Uri(urls(i))
         ' Pass the address, and also use it for the userToken
         ' to identify the page when the delegate is invoked.
         webClients(i).DownloadStringAsync(address, address)
      Next

      ' Return the underlying Task. The client code
      ' waits on the Result property, and handles exceptions
      ' in the try-catch block there.
      Return tcs.Task
   End Function

   Public Sub WebEventHandler(ByVal sender As Object, ByVal args As
DownloadStringCompletedEventArgs)

      If args.Cancelled = True Then
         tcs.TrySetCanceled()
         Return
      ElseIf args.Error IsNot Nothing Then
```

```
            tcs.TrySetException(args.Error)
            Return
        Else
            ' Split the string into an array of words,
            ' then count the number of elements that match
            ' the search term.
            Dim words() As String = args.Result.Split(" "c)

            Dim name As String = nameToSearch.ToUpper()
            Dim nameCount = (From word In words.AsParallel()
                             Where word.ToUpper().Contains(name)
                             Select word).Count()

            ' Associate the results with the url, and add new string to the array that
            ' the underlying Task object will return in its Result property.
            SyncLock (m_lock)
               results.Add(String.Format("{0} has {1} instances of {2}", args.UserState,
nameCount, nameToSearch))
               count = count + 1
               If (count = addresses.Length) Then
                  tcs.TrySetResult(results.ToArray())
               End If
            End SyncLock
        End If
    End Sub
End Class
```

For a more complete example, which includes additional exception handling and shows how to call the method from client code, see How to: Wrap EAP Patterns in a Task.

Remember that any task that is created by a TaskCompletionSource(Of TResult) will be started by that TaskCompletionSource and, therefore, user code should not call the Start method on that task.

Implementing the APM Pattern By Using Tasks

In some scenarios, it may be desirable to directly expose the IAsyncResult pattern by using Begin/End method pairs in an API. For example, you may want to maintain consistency with existing APIs, or you may have automated tools that require this pattern. In such cases, you can use Tasks to simplify how the APM pattern is implemented internally.

The following example shows how to use tasks to implement an APM Begin/End method pair for a long-running compute-bound method.

VB

```
Class Calculator
    Public Function BeginCalculate(ByVal decimalPlaces As Integer, ByVal ac As
AsyncCallback, ByVal state As Object) As IAsyncResult
        Console.WriteLine("Calling BeginCalculate on thread {0}",
Thread.CurrentThread.ManagedThreadId)
        Dim myTask = Task(Of String).Factory.StartNew(Function(obj)
Compute(decimalPlaces), state)
```

```
            myTask.ContinueWith(Sub(antedecent) ac(myTask))

        End Function
        Private Function Compute(ByVal decimalPlaces As Integer)
            Console.WriteLine("Calling compute on thread {0}",
    Thread.CurrentThread.ManagedThreadId)

            ' Simulating some heavy work.
            Thread.SpinWait(500000000)

            ' Actual implemenation left as exercise for the reader.
            ' Several examples are available on the Web.
            Return "3.14159265358979323846264338327950288"
        End Function

        Public Function EndCalculate(ByVal ar As IAsyncResult) As String
            Console.WriteLine("Calling EndCalculate on thread {0}",
    Thread.CurrentThread.ManagedThreadId)
            Return CType(ar, Task(Of String)).Result
        End Function
    End Class

    Class CalculatorClient
        Shared decimalPlaces As Integer
        Shared Sub Main()
            Dim calc As New Calculator
            Dim places As Integer = 35
            Dim callback As New AsyncCallback(AddressOf PrintResult)
            Dim ar As IAsyncResult = calc.BeginCalculate(places, callback, calc)

            ' Do some work on this thread while the calulator is busy.
            Console.WriteLine("Working...")
            Thread.SpinWait(500000)
            Console.ReadLine()
        End Sub

        Public Shared Sub PrintResult(ByVal result As IAsyncResult)
            Dim c As Calculator = CType(result.AsyncState, Calculator)
            Dim piString As String = c.EndCalculate(result)
            Console.WriteLine("Calling PrintResult on thread {0}; result = {1}",
                      Thread.CurrentThread.ManagedThreadId, piString)
        End Sub

    End Class
```

Using the StreamExtensions Sample Code

The Streamextensions.cs file, in Samples for Parallel Programming with the .NET Framework 4 on the MSDN Web site, contains several reference implementations that use Task objects for asynchronous file and network I/O.

See Also

Task Parallel Library (TPL)

How to: Wrap EAP Patterns in a Task

.NET Framework (current version)

The following example shows how to expose an arbitrary sequence of Event-Based Asynchronous Pattern (EAP) operations as one task by using a TaskCompletionSource(Of TResult). The example also shows how to use a CancellationToken to invoke the built-in cancellation methods on the WebClient objects.

Example

VB

```
Class WebDataDownLoader

    Dim tcs As New TaskCompletionSource(Of String())
    Dim nameToSearch As String
    Dim token As CancellationToken
    Dim results As New List(Of String)
    Dim m_lock As Object
    Dim count As Integer
    Dim addresses() As String

    Shared Sub Main()

        Dim downloader As New WebDataDownLoader()
        downloader.addresses = {"http://www.msnbc.com", "http://www.yahoo.com", _
                                "http://www.nytimes.com",
"http://www.washingtonpost.com", _
                                "http://www.latimes.com", "http://www.newsday.com"}
        Dim cts As New CancellationTokenSource()

        ' Create a UI thread from which to cancel the entire operation
        Task.Factory.StartNew(Sub()
                                  Console.WriteLine("Press c to cancel")
                                  If Console.ReadKey().KeyChar = "c"c Then
                                      cts.Cancel()
                                  End If
                              End Sub)

        ' Using a neutral search term that is sure to get some hits on English web sites.
        ' Please substitute your favorite search term.
        downloader.nameToSearch = "the"
        Dim webTask As Task(Of String()) = downloader.GetWordCounts(downloader.addresses,
downloader.nameToSearch, cts.Token)

        ' Do some other work here unless the method has already completed.
        If (webTask.IsCompleted = False) Then
            ' Simulate some work
            Thread.SpinWait(5000000)
```

```
        End If

        Dim results As String() = Nothing
        Try
            results = webTask.Result
        Catch ae As AggregateException
            For Each ex As Exception In ae.InnerExceptions
                If (TypeOf (ex) Is OperationCanceledException) Then
                    Dim oce As OperationCanceledException = CType(ex,
OperationCanceledException)
                    If oce.CancellationToken = cts.Token Then
                        Console.WriteLine("Operation canceled by user.")
                    End If
                Else
                    Console.WriteLine(ex.Message)
                End If

            Next
        Finally
           cts.Dispose()
        End Try

        If (Not results Is Nothing) Then
            For Each item As String In results
                Console.WriteLine(item)
            Next
        End If

        Console.WriteLine("Press any key to exit")
        Console.ReadKey()
    End Sub

    Public Function GetWordCounts(ByVal urls() As String, ByVal str As String, ByVal
token As CancellationToken) As Task(Of String())

        Dim webClients() As WebClient
        ReDim webClients(urls.Length)
        m_lock = New Object()

        ' If the user cancels the CancellationToken, then we can use the
        ' WebClient's ability to cancel its own async operations.
        token.Register(Sub()
                           For Each wc As WebClient In webClients
                               If Not wc Is Nothing Then
                                   wc.CancelAsync()
                               End If
                           Next
                       End Sub)


        For i As Integer = 0 To urls.Length - 1
            webClients(i) = New WebClient()

            ' Specify the callback for the DownloadStringCompleted
```

```
            ' event that will be raised by this WebClient instance.
            AddHandler webClients(i).DownloadStringCompleted, AddressOf WebEventHandler

            Dim address As Uri = Nothing
            Try
                address = New Uri(urls(i))
                ' Pass the address, and also use it for the userToken
                ' to identify the page when the delegate is invoked.
                webClients(i).DownloadStringAsync(address, address)
            Catch ex As UriFormatException
                tcs.TrySetException(ex)
                Return tcs.Task
            End Try

        Next

        ' Return the underlying Task. The client code
        ' waits on the Result property, and handles exceptions
        ' in the try-catch block there.
        Return tcs.Task
    End Function



    Public Sub WebEventHandler(ByVal sender As Object, ByVal args As
DownloadStringCompletedEventArgs)

        If args.Cancelled = True Then
            tcs.TrySetCanceled()
            Return
        ElseIf Not args.Error Is Nothing Then
            tcs.TrySetException(args.Error)
            Return
        Else
            ' Split the string into an array of words,
            ' then count the number of elements that match
            ' the search term.
            Dim words() As String = args.Result.Split(" "c)
            Dim NAME As String = nameToSearch.ToUpper()
            Dim nameCount = (From word In words.AsParallel()
                             Where word.ToUpper().Contains(NAME)
                             Select word).Count()

            ' Associate the results with the url, and add new string to the array that
            ' the underlying Task object will return in its Result property.
            results.Add(String.Format("{0} has {1} instances of {2}", args.UserState,
nameCount, nameToSearch))
        End If

        SyncLock (m_lock)
            count = count + 1
            If (count = addresses.Length) Then
                tcs.TrySetResult(results.ToArray())
            End If
```

```
            End SyncLock
        End Sub
```

See Also

TPL and Traditional .NET Framework Asynchronous Programming

Potential Pitfalls in Data and Task Parallelism

.NET Framework (current version)

In many cases, Parallel.For and Parallel.ForEach can provide significant performance improvements over ordinary sequential loops. However, the work of parallelizing the loop introduces complexity that can lead to problems that, in sequential code, are not as common or are not encountered at all. This topic lists some practices to avoid when you write parallel loops.

Do Not Assume That Parallel Is Always Faster

In certain cases a parallel loop might run slower than its sequential equivalent. The basic rule of thumb is that parallel loops that have few iterations and fast user delegates are unlikely to speedup much. However, because many factors are involved in performance, we recommend that you always measure actual results.

Avoid Writing to Shared Memory Locations

In sequential code, it is not uncommon to read from or write to static variables or class fields. However, whenever multiple threads are accessing such variables concurrently, there is a big potential for race conditions. Even though you can use locks to synchronize access to the variable, the cost of synchronization can hurt performance. Therefore, we recommend that you avoid, or at least limit, access to shared state in a parallel loop as much as possible. The best way to do this is to use the overloads of Parallel.For and Parallel.ForEach that use a System.Threading.ThreadLocal(Of T) variable to store thread-local state during loop execution. For more information, see How to: Write a Parallel.For Loop with Thread-Local Variables and How to: Write a Parallel.ForEach Loop with Thread-Local Variables.

Avoid Over-Parallelization

By using parallel loops, you incur the overhead costs of partitioning the source collection and synchronizing the worker threads. The benefits of parallelization are further limited by the number of processors on the computer. There is no speedup to be gained by running multiple compute-bound threads on just one processor. Therefore, you must be careful not to over-parallelize a loop.

The most common scenario in which over-parallelization can occur is in nested loops. In most cases, it is best to parallelize only the outer loop unless one or more of the following conditions apply:

- The inner loop is known to be very long.
- You are performing an expensive computation on each order. (The operation shown in the example is not expensive.)
- The target system is known to have enough processors to handle the number of threads that will be produced by parallelizing the query on `cust.Orders`.

In all cases, the best way to determine the optimum query shape is to test and measure.

Avoid Calls to Non-Thread-Safe Methods

Writing to non-thread-safe instance methods from a parallel loop can lead to data corruption which may or may not go undetected in your program. It can also lead to exceptions. In the following example, multiple threads would be attempting to call the FileStream.WriteByte method simultaneously, which is not supported by the class.

VB

```
Dim fs As FileStream = File.OpenWrite(filepath)
Dim bytes() As Byte
ReDim bytes(1000000)
' ...init byte array
Parallel.For(0, bytes.Length, Sub(n) fs.WriteByte(bytes(n)))
```

Limit Calls to Thread-Safe Methods

Most static methods in the .NET Framework are thread-safe and can be called from multiple threads concurrently. However, even in these cases, the synchronization involved can lead to significant slowdown in the query.

Note

You can test for this yourself by inserting some calls to WriteLine in your queries. Although this method is used in the documentation examples for demonstration purposes, do not use it in parallel loops unless necessary.

Be Aware of Thread Affinity Issues

Some technologies, for example, COM interoperability for Single-Threaded Apartment (STA) components, Windows Forms, and Windows Presentation Foundation (WPF), impose thread affinity restrictions that require code to run on a specific thread. For example, in both Windows Forms and WPF, a control can only be accessed on the thread on which it was created. This means, for example, that you cannot update a list control from a parallel loop unless you configure the thread scheduler to schedule work only on the UI thread. For more information, see How to: Schedule Work on the User Interface (UI) Thread.

Use Caution When Waiting in Delegates That Are Called by Parallel.Invoke

In certain circumstances, the Task Parallel Library will inline a task, which means it runs on the task on the currently executing thread. (For more information, see Task Schedulers.) This performance optimization can lead to deadlock in

certain cases. For example, two tasks might run the same delegate code, which signals when an event occurs, and then waits for the other task to signal. If the second task is inlined on the same thread as the first, and the first goes into a Wait state, the second task will never be able to signal its event. To avoid such an occurrence, you can specify a timeout on the Wait operation, or use explicit thread constructors to help ensure that one task cannot block the other.

Do Not Assume that Iterations of ForEach, For and ForAll Always Execute in Parallel

It is important to keep in mind that individual iterations in a For, ForEach or ForAll(Of TSource) loop may but do not have to execute in parallel. Therefore, you should avoid writing any code that depends for correctness on parallel execution of iterations or on the execution of iterations in any particular order. For example, this code is likely to deadlock:

VB

```
Dim mres = New ManualResetEventSlim()
Enumerable.Range(0, Environment.ProcessorCount * 100) _
.AsParallel() _
.ForAll(Sub(j)

            If j = Environment.ProcessorCount Then
                Console.WriteLine("Set on {0} with value of {1}",
                                  Thread.CurrentThread.ManagedThreadId, j)
                mres.Set()
            Else
                Console.WriteLine("Waiting on {0} with value of {1}",
                                  Thread.CurrentThread.ManagedThreadId, j)
                mres.Wait()
            End If
        End Sub) ' deadlocks
```

In this example, one iteration sets an event, and all other iterations wait on the event. None of the waiting iterations can complete until the event-setting iteration has completed. However, it is possible that the waiting iterations block all threads that are used to execute the parallel loop, before the event-setting iteration has had a chance to execute. This results in a deadlock – the event-setting iteration will never execute, and the waiting iterations will never wake up.

In particular, one iteration of a parallel loop should never wait on another iteration of the loop to make progress. If the parallel loop decides to schedule the iterations sequentially but in the opposite order, a deadlock will occur.

Avoid Executing Parallel Loops on the UI Thread

It is important to keep your application's user interface (UI) responsive. If an operation contains enough work to warrant parallelization, then it likely should not be run that operation on the UI thread. Instead, it should offload that operation to be run on a background thread. For example, if you want to use a parallel loop to compute some data that should then be rendered into a UI control, you should consider executing the loop within a task instance rather than directly in a UI event handler. Only when the core computation has completed should you then marshal the UI update back to the UI thread.

If you do run parallel loops on the UI thread, be careful to avoid updating UI controls from within the loop. Attempting to update UI controls from within a parallel loop that is executing on the UI thread can lead to state corruption, exceptions, delayed updates, and even deadlocks, depending on how the UI update is invoked. In the following example, the parallel loop blocks the UI thread on which it's executing until all iterations are complete. However, if an iteration of the loop is running on a background thread (as For may do), the call to Invoke causes a message to be submitted to the UI thread and blocks waiting for that message to be processed. Since the UI thread is blocked running the For, the message can never be processed, and the UI thread deadlocks.

VB

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click

    Dim iterations As Integer = 20
    Parallel.For(0, iterations, Sub(x)
                                    Button1.Invoke(Sub()
                                                       DisplayProgress(x)
                                                   End Sub)
                                End Sub)
End Sub
```

The following example shows how to avoid the deadlock, by running the loop inside a task instance. The UI thread is not blocked by the loop, and the message can be processed.

VB

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click

    Dim iterations As Integer = 20
    Task.Factory.StartNew(Sub() Parallel.For(0, iterations, Sub(x)
                                                                Button1.Invoke(Sub()

DisplayProgress(x)
                                                                                   End Sub)
                                                            End Sub))
End Sub
```

See Also

Parallel Programming in the .NET Framework
Potential Pitfalls with PLINQ
Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4

Parallel LINQ (PLINQ)

.NET Framework (current version)

Parallel LINQ (PLINQ) is a parallel implementation of LINQ to Objects. PLINQ implements the full set of LINQ standard query operators as extension methods for the System.Linq namespace and has additional operators for parallel operations. PLINQ combines the simplicity and readability of LINQ syntax with the power of parallel programming. Just like code that targets the Task Parallel Library, PLINQ queries scale in the degree of concurrency based on the capabilities of the host computer.

In many scenarios, PLINQ can significantly increase the speed of LINQ to Objects queries by using all available cores on the host computer more efficiently. This increased performance brings high performance computing power onto the desktop.

In This Section

See Also

LINQ (Language-Integrated Query)

Introduction to PLINQ

.NET Framework (current version)

What is a Parallel Query?

Language-Integrated Query (LINQ) was introduced in the .NET Framework 3.5. It features a unified model for querying any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable(Of T) data source in a type-safe manner. LINQ to Objects is the name for LINQ queries that are run against in-memory collections such as List(Of T) and arrays. This article assumes that you have a basic understand of LINQ. For more information, see LINQ (Language-Integrated Query).

Parallel LINQ (PLINQ) is a parallel implementation of the LINQ pattern. A PLINQ query in many ways resembles a non-parallel LINQ to Objects query. PLINQ queries, just like sequential LINQ queries, operate on any in-memory IEnumerable or IEnumerable(Of T) data source, and have deferred execution, which means they do not begin executing until the query is enumerated. The primary difference is that PLINQ attempts to make full use of all the processors on the system. It does this by partitioning the data source into segments, and then executing the query on each segment on separate worker threads in parallel on multiple processors. In many cases, parallel execution means that the query runs significantly faster.

Through parallel execution, PLINQ can achieve significant performance improvements over legacy code for certain kinds of queries, often just by adding the AsParallelquery operation to the data source. However, parallelism can introduce its own complexities, and not all query operations run faster in PLINQ. In fact, parallelization actually slows down certain queries. Therefore, you should understand how issues such as ordering affect parallel queries. For more information, see Understanding Speedup in PLINQ.

> **Note**
>
> This documentation uses lambda expressions to define delegates in PLINQ. If you are not familiar with lambda expressions in C# or Visual Basic, see Lambda Expressions in PLINQ and TPL.

The remainder of this article gives an overview of the main PLINQ classes, and discusses how to create PLINQ queries. Each section contains links to more detailed information and code examples.

The ParallelEnumerable Class

The System.Linq.ParallelEnumerable class exposes almost all of PLINQ's functionality. It and the rest of the System.Linq namespace types are compiled into the System.Core.dll assembly. The default C# and Visual Basic projects in Visual Studio both reference the assembly and import the namespace.

ParallelEnumerable includes implementations of all the standard query operators that LINQ to Objects supports, although it does not attempt to parallelize each one. If you are not familiar with LINQ, see Introduction to LINQ.

In addition to the standard query operators, the ParallelEnumerable class contains a set of methods that enable behaviors

specific to parallel execution. These PLINQ-specific methods are listed in the following table.

ParallelEnumerable Operator	Description
AsParallel(Of TSource)	The entry point for PLINQ. Specifies that the rest of the query should be parallelized, if it is possible.
AsSequential(Of TSource)	Specifies that the rest of the query should be run sequentially, as a non-parallel LINQ query.
AsOrdered(Of TSource)	Specifies that PLINQ should preserve the ordering of the source sequence for the rest of the query, or until the ordering is changed, for example by the use of an orderby (Order By in Vlsual Basic) clause.
AsUnordered(Of TSource)	Specifies that PLINQ for the rest of the query is not required to preserve the ordering of the source sequence.
WithCancellation(Of TSource)	Specifies that PLINQ should periodically monitor the state of the provided cancellation token and cancel execution if it is requested.
WithDegreeOfParallelism(Of TSource)	Specifies the maximum number of processors that PLINQ should use to parallelize the query.
WithMergeOptions(Of TSource)	Provides a hint about how PLINQ should, if it is possible, merge parallel results back into just one sequence on the consuming thread.
WithExecutionMode(Of TSource)	Specifies whether PLINQ should parallelize the query even when the default behavior would be to run it sequentially.
ForAll(Of TSource)	A multithreaded enumeration method that, unlike iterating over the results of the query, enables results to be processed in parallel without first merging back to the consumer thread.
Aggregate(Of TSource) overload	An overload that is unique to PLINQ and enables intermediate aggregation over thread-local partitions, plus a final aggregation function to combine the results of all partitions.

The Opt-in Model

When you write a query, opt in to PLINQ by invoking the ParallelEnumerable.AsParallel(Of TSource) extension method on the data source, as shown in the following example.

VB

```
Dim source = Enumerable.Range(1, 10000)

' Opt in to PLINQ with AsParallel
```

```
Dim evenNums = From num In source.AsParallel()
               Where num Mod 2 = 0
               Select num
Console.WriteLine("{0} even numbers out of {1} total",
                  evenNums.Count(), source.Count())
' The example displays the following output:
'       5000 even numbers out of 10000 total
```

The AsParallel(Of TSource) extension method binds the subsequent query operators, in this case, **where** and **select**, to the System.Linq.ParallelEnumerable implementations.

Execution Modes

By default, PLINQ is conservative. At run time, the PLINQ infrastructure analyzes the overall structure of the query. If the query is likely to yield speedups by parallelization, PLINQ partitions the source sequence into tasks that can be run concurrently. If it is not safe to parallelize a query, PLINQ just runs the query sequentially. If PLINQ has a choice between a potentially expensive parallel algorithm or an inexpensive sequential algorithm, it chooses the sequential algorithm by default. You can use the WithExecutionMode(Of TSource) method and the System.Linq.ParallelExecutionMode enumeration to instruct PLINQ to select the parallel algorithm. This is useful when you know by testing and measurement that a particular query executes faster in parallel. For more information, see How to: Specify the Execution Mode in PLINQ.

Degree of Parallelism

By default, PLINQ uses all of the processors on the host computer up to a maximum of 64. You can instruct PLINQ to use no more than a specified number of processors by using the WithDegreeOfParallelism(Of TSource) method. This is useful when you want to make sure that other processes running on the computer receive a certain amount of CPU time. The following snippet limits the query to utilizing a maximum of two processors.

VB

```
Dim query = From item In source.AsParallel().WithDegreeOfParallelism(2)
            Where Compute(item) > 42
            Select item
```

In cases where a query is performing a significant amount of non-compute-bound work such as File I/O, it might be beneficial to specify a degree of parallelism greater than the number of cores on the machine.

Ordered Versus Unordered Parallel Queries

In some queries, a query operator must produce results that preserve the ordering of the source sequence. PLINQ provides the AsOrdered(Of TSource) operator for this purpose. AsOrdered(Of TSource) is distinct from AsSequential(Of TSource). An AsOrdered(Of TSource) sequence is still processed in parallel, but its results are buffered and sorted. Because order preservation typically involves extra work, an AsOrdered(Of TSource) sequence might be processed more slowly than the default AsUnordered(Of TSource) sequence. Whether a particular ordered parallel operation is faster than a sequential version of the operation depends on many factors.

The following code example shows how to opt in to order preservation.

VB

```
Dim evenNums = From num In numbers.AsParallel().AsOrdered()
               Where num Mod 2 = 0
               Select num
```

For more information, see Order Preservation in PLINQ.

Parallel vs. Sequential Queries

Some operations require that the source data be delivered in a sequential manner. The ParallelEnumerable query operators revert to sequential mode automatically when it is required. For user-defined query operators and user delegates that require sequential execution, PLINQ provides the AsSequential(Of TSource) method. When you use AsSequential(Of TSource), all subsequent operators in the query are executed sequentially until AsParallel(Of TSource) is called again. For more information, see How to: Combine Parallel and Sequential LINQ Queries.

Options for Merging Query Results

When a PLINQ query executes in parallel, its results from each worker thread must be merged back onto the main thread for consumption by a **foreach** loop (**For Each** in Visual Basic), or insertion into a list or array. In some cases, it might be beneficial to specify a particular kind of merge operation, for example, to begin producing results more quickly. For this purpose, PLINQ supports the WithMergeOptions(Of TSource) method, and the ParallelMergeOptions enumeration. For more information, see Merge Options in PLINQ.

The ForAll Operator

In sequential LINQ queries, execution is deferred until the query is enumerated either in a **foreach** (**For Each** in Visual Basic) loop or by invoking a method such as ToList(Of TSource) , ToArray(Of TSource) , or ToDictionary(Of TSource, TKey). In PLINQ, you can also use **foreach** to execute the query and iterate through the results. However, **foreach** itself does not run in parallel, and therefore, it requires that the output from all parallel tasks be merged back into the thread on which the loop is running. In PLINQ, you can use **foreach** when you must preserve the final ordering of the query results, and also whenever you are processing the results in a serial manner, for example when you are calling **Console.WriteLine** for each element. For faster query execution when order preservation is not required and when the processing of the results can itself be parallelized, use the ForAll(Of TSource) method to execute a PLINQ query. ForAll(Of TSource) does not perform this final merge step. The following code example shows how to use the ForAll(Of TSource) method. System.Collections.Concurrent.ConcurrentBag(Of T) is used here because it is optimized for multiple threads adding concurrently without attempting to remove any items.

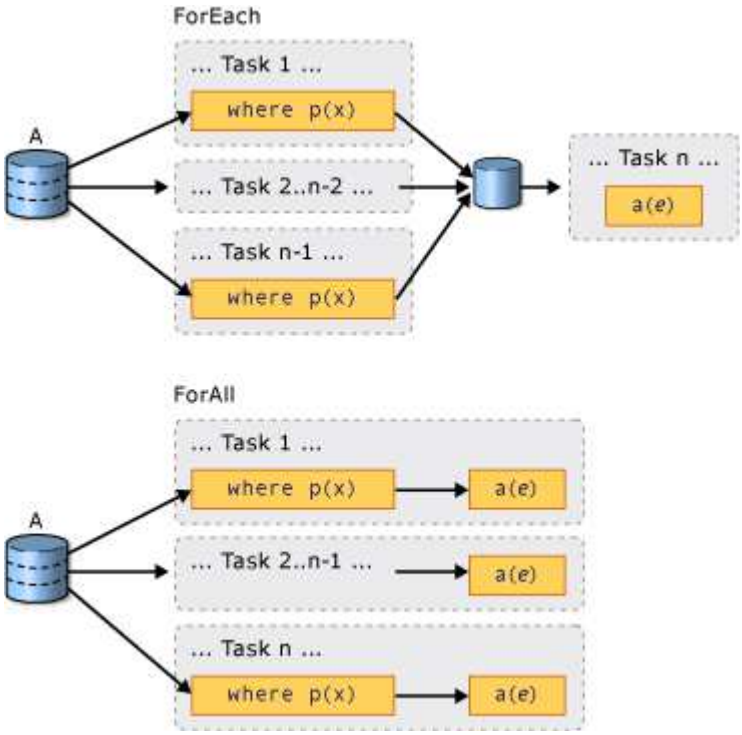
VB

```
Dim nums = Enumerable.Range(10, 10000)
Dim query = From num In nums.AsParallel()
```

```
            Where num Mod 10 = 0
            Select num

' Process the results as each thread completes
' and add them to a System.Collections.Concurrent.ConcurrentBag(Of Int)
' which can safely accept concurrent add operations
query.ForAll(Sub(e) concurrentBag.Add(Compute(e)))
```

The following illustration shows the difference between **foreach** and ForAll(Of TSource) with regard to query execution.



Cancellation

PLINQ is integrated with the cancellation types in .NET Framework 4. (For more information, see Cancellation in Managed Threads.) Therefore, unlike sequential LINQ to Objects queries, PLINQ queries can be canceled. To create a cancelable PLINQ query, use the WithCancellation(Of TSource) operator on the query and provide a CancellationToken instance as the argument. When the IsCancellationRequested property on the token is set to true, PLINQ will notice it, stop processing on all threads, and throw an OperationCanceledException.

It is possible that a PLINQ query might continue to process some elements after the cancellation token is set.

For greater responsiveness, you can also respond to cancellation requests in long-running user delegates. For more information, see How to: Cancel a PLINQ Query.

Exceptions

When a PLINQ query executes, multiple exceptions might be thrown from different threads simultaneously. Also, the code to handle the exception might be on a different thread than the code that threw the exception. PLINQ uses the AggregateException type to encapsulate all the exceptions that were thrown by a query, and marshal those exceptions

back to the calling thread. On the calling thread, only one try-catch block is required. However, you can iterate through all of the exceptions that are encapsulated in the AggregateException and catch any that you can safely recover from. In rare cases, some exceptions may be thrown that are not wrapped in an AggregateException, and ThreadAbortExceptions are also not wrapped.

When exceptions are allowed to bubble up back to the joining thread, then it is possible that a query may continue to process some items after the exception is raised.

For more information, see How to: Handle Exceptions in a PLINQ Query.

Custom Partitioners

In some cases, you can improve query performance by writing a custom partitioner that takes advantage of some characteristic of the source data. In the query, the custom partitioner itself is the enumerable object that is queried.

VB

```
Dim arr(10000) As Integer
Dim partitioner As Partitioner(Of Integer) = New MyArrayPartitioner(Of Integer)(arr)
Dim query = partitioner.AsParallel().Select(Function(x) SomeFunction(x))
```

PLINQ supports a fixed number of partitions (although data may be dynamically reassigned to those partitions during run time for load balancing.). For and ForEach support only dynamic partitioning, which means that the number of partitions changes at run time. For more information, see Custom Partitioners for PLINQ and TPL.

Measuring PLINQ Performance

In many cases, a query can be parallelized, but the overhead of setting up the parallel query outweighs the performance benefit gained. If a query does not perform much computation or if the data source is small, a PLINQ query may be slower than a sequential LINQ to Objects query. You can use the Parallel Performance Analyzer in Visual Studio Team Server to compare the performance of various queries, to locate processing bottlenecks, and to determine whether your query is running in parallel or sequentially. For more information, see Concurrency Visualizer and How to: Measure PLINQ Query Performance.

See Also

Parallel LINQ (PLINQ)
Understanding Speedup in PLINQ

Custom Partitioners for PLINQ and TPL

.NET Framework (current version)

To parallelize an operation on a data source, one of the essential steps is to *partition* the source into multiple sections that can be accessed concurrently by multiple threads. PLINQ and the Task Parallel Library (TPL) provide default partitioners that work transparently when you write a parallel query or ForEach loop. For more advanced scenarios, you can plug in your own partitioner.

Kinds of Partitioning

There are many ways to partition a data source. In the most efficient approaches, multiple threads cooperate to process the original source sequence, rather than physically separating the source into multiple subsequences. For arrays and other indexed sources such as IList collections where the length is known in advance, *range partitioning* is the simplest kind of partitioning. Every thread receives unique beginning and ending indexes, so that it can process its range of the source without overwriting or being overwritten by any other thread. The only overhead involved in range partitioning is the initial work of creating the ranges; no additional synchronization is required after that. Therefore, it can provide good performance as long as the workload is divided evenly. A disadvantage of range partitioning is that if one thread finishes early, it cannot help the other threads finish their work.

For linked lists or other collections whose length is not known, you can use *chunk partitioning*. In chunk partitioning, every thread or task in a parallel loop or query consumes some number of source elements in one chunk, processes them, and then comes back to retrieve additional elements. The partitioner ensures that all elements are distributed and that there are no duplicates. A chunk may be any size. For example, the partitioner that is demonstrated in How to: Implement Dynamic Partitions creates chunks that contain just one element. As long as the chunks are not too large, this kind of partitioning is inherently load-balancing because the assignment of elements to threads is not pre-determined. However, the partitioner does incur the synchronization overhead each time the thread needs to get another chunk. The amount of synchronization incurred in these cases is inversely proportional to the size of the chunks.

In general, range partitioning is only faster when the execution time of the delegate is small to moderate, and the source has a large number of elements, and the total work of each partition is roughly equivalent. Chunk partitioning is therefore generally faster in most cases. On sources with a small number of elements or longer execution times for the delegate, then the performance of chunk and range partitioning is about equal.

The TPL partitioners also support a dynamic number of partitions. This means they can create partitions on-the-fly, for example, when the ForEach loop spawns a new task. This feature enables the partitioner to scale together with the loop itself. Dynamic partitioners are also inherently load-balancing. When you create a custom partitioner, you must support dynamic partitioning to be consumable from a ForEach loop.

Configuring Load Balancing Partitioners for PLINQ

Some overloads of the Partitioner.Create method let you create a partitioner for an array or IList source and specify whether it should attempt to balance the workload among the threads. When the partitioner is configured to load-balance, chunk partitioning is used, and the elements are handed off to each partition in small chunks as they are requested. This approach helps ensure that all partitions have elements to process until the entire loop or query is completed. An additional overload can be used to provide load-balancing partitioning of any IEnumerable source.

In general, load balancing requires the partitions to request elements relatively frequently from the partitioner. By

contrast, a partitioner that does static partitioning can assign the elements to each partitioner all at once by using either range or chunk partitioning. This requires less overhead than load balancing, but it might take longer to execute if one thread ends up with significantly more work than the others. By default when it is passed an IList or an array, PLINQ always uses range partitioning without load balancing. To enable load balancing for PLINQ, use the **Partitioner.Create** method, as shown in the following example.

VB

```
' Static number of partitions requires indexable source.
Dim nums = Enumerable.Range(0, 100000000).ToArray()

' Create a load-balancing partitioner. Or specify false For  Shared partitioning.
Dim customPartitioner = Partitioner.Create(nums, True)

' The partitioner is the query's data source.
Dim q = From x In customPartitioner.AsParallel()
        Select x * Math.PI

q.ForAll(Sub(x) ProcessData(x))
```

The best way to determine whether to use load balancing in any given scenario is to experiment and measure how long it takes operations to complete under representative loads and computer configurations. For example, static partitioning might provide significant speedup on a multi-core computer that has only a few cores, but it might result in slowdowns on computers that have relatively many cores.

The following table lists the available overloads of the Create method. These partitioners are not limited to use only with PLINQ or Task. They can also be used with any custom parallel construct.

Overload	Uses load balancing
Create(Of TSource)(IEnumerable(Of TSource))	Always
Create(Of TSource)(TSource(), Boolean)	When the Boolean argument is specified as true
Create(Of TSource)(IList(Of TSource), Boolean)	When the Boolean argument is specified as true
Create(Int32, Int32)	Never
Create(Int32, Int32, Int32)	Never
Create(Int64, Int64)	Never
Create(Int64, Int64, Int64)	Never

Configuring Static Range Partitioners for Parallel.ForEach

In a For loop, the body of the loop is provided to the method as a delegate. The cost of invoking that delegate is about

the same as a virtual method call. In some scenarios, the body of a parallel loop might be small enough that the cost of the delegate invocation on each loop iteration becomes significant. In such situations, you can use one of the Create overloads to create an IEnumerable(Of T) of range partitions over the source elements. Then, you can pass this collection of ranges to a ForEach method whose body consists of a regular **for** loop. The benefit of this approach is that the delegate invocation cost is incurred only once per range, rather than once per element. The following example demonstrates the basic pattern.

VB

```
Imports System.Threading.Tasks
Imports System.Collections.Concurrent

Module PartitionDemo

    Sub Main()
        ' Source must be array or IList.
        Dim source = Enumerable.Range(0, 100000).ToArray()

        ' Partition the entire source array.
        ' Let the partitioner size the ranges.
        Dim rangePartitioner = Partitioner.Create(0, source.Length)

        Dim results(source.Length - 1) As Double

        ' Loop over the partitions in parallel. The Sub is invoked
        ' once per partition.
        Parallel.ForEach(rangePartitioner, Sub(range, loopState)

                                               ' Loop over each range element without
a delegate invocation.
                                               For i As Integer = range.Item1 To
range.Item2 - 1
                                                   results(i) = source(i) * Math.PI
                                               Next
                                           End Sub)
        Console.WriteLine("Operation complete. Print results? y/n")
        Dim input As Char = Console.ReadKey().KeyChar
        If input = "y"c Or input = "Y"c Then
            For Each d As Double In results
                Console.Write("{0} ", d)
            Next
        End If

    End Sub
End Module
```

Every thread in the loop receives its own Tuple(Of T1, T2) that contains the starting and ending index values in the specified sub-range. The inner **for** loop uses the `fromInclusive` and `toExclusive` values to loop over the array or the IList directly.

One of the Create overloads lets you specify the size of the partitions, and the number of partitions. This overload can be used in scenarios where the work per element is so low that even one virtual method call per element has a

noticeable impact on performance.

Custom Partitioners

In some scenarios, it might be worthwhile or even required to implement your own partitioner. For example, you might have a custom collection class that you can partition more efficiently than the default partitioners can, based on your knowledge of the internal structure of the class. Or, you may want to create range partitions of varying sizes based on your knowledge of how long it will take to process elements at different locations in the source collection.

To create a basic custom partitioner, derive a class from System.Collections.Concurrent.Partitioner(Of TSource) and override the virtual methods, as described in the following table.

GetPartitions	This method is called once by the main thread and returns an IList(IEnumerator(TSource)). Each worker thread in the loop or query can call **GetEnumerator** on the list to retrieve a IEnumerator(Of T) over a distinct partition.
SupportsDynamicPartitions	Return **true** if you implement GetDynamicPartitions, otherwise, **false**.
GetDynamicPartitions	If SupportsDynamicPartitions is **true**, this method can optionally be called instead of GetPartitions.

If the results must be sortable or you require indexed access into the elements, then derive from System.Collections.Concurrent.OrderablePartitioner(Of TSource) and override its virtual methods as described in the following table.

GetPartitions	This method is called once by the main thread and returns an **IList(IEnumerator(TSource))**. Each worker thread in the loop or query can call **GetEnumerator** on the list to retrieve a IEnumerator(Of T) over a distinct partition.
SupportsDynamicPartitions	Return **true** if you implement GetDynamicPartitions; otherwise, false.
GetDynamicPartitions	Typically, this just calls GetOrderableDynamicPartitions.
GetOrderableDynamicPartitions	If SupportsDynamicPartitions is **true**, this method can optionally be called instead of GetPartitions.

The following table provides additional details about how the three kinds of load-balancing partitioners implement the OrderablePartitioner(Of TSource) class.

Method/Property	**IList / Array without Load Balancing**	**IList / Array with Load Balancing**	**IEnumerable**

GetOrderablePartitions	Uses range partitioning	Uses chunk partitioning optimized for Lists for the partitionCount specified	Uses chunk partitioning by creating a static number of partitions.
OrderablePartitioner(Of TSource).GetOrderableDynamicPartitions	Throws not-supported exception	Uses chunk partitioning optimized for Lists and dynamic partitions	Uses chunk partitioning by creating a dynamic number of partitions.
KeysOrderedInEachPartition	Returns **true**	Returns **true**	Returns **true**
KeysOrderedAcrossPartitions	Returns **true**	Returns **false**	Returns **false**
KeysNormalized	Returns **true**	Returns **true**	Returns **true**
SupportsDynamicPartitions	Returns **false**	Returns **true**	Returns **true**

Dynamic Partitions

If you intend the partitioner to be used in a ForEach method, you must be able to return a dynamic number of partitions. This means that the partitioner can supply an enumerator for a new partition on-demand at any time during loop execution. Basically, whenever the loop adds a new parallel task, it requests a new partition for that task. If you require the data to be orderable, then derive from System.Collections.Concurrent.OrderablePartitioner(Of TSource) so that each item in each partition is assigned a unique index.

For more information, and an example, see How to: Implement Dynamic Partitions.

Contract for Partitioners

When you implement a custom partitioner, follow these guidelines to help ensure correct interaction with PLINQ and ForEach in the TPL:

- If GetPartitions is called with an argument of zero or less for **partitionsCount**, throw ArgumentOutOfRangeException. Although PLINQ and TPL will never pass in a **partitionCount** equal to 0, we nevertheless recommend that you guard against the possibility.
- GetPartitions and GetOrderablePartitions should always return **partitionsCount** number of partitions. If the partitioner runs out of data and cannot create as many partitions as requested, then the method should return an empty enumerator for each of the remaining partitions. Otherwise, both PLINQ and TPL will throw an InvalidOperationException.
- GetPartitions, GetOrderablePartitions, GetDynamicPartitions, and GetOrderableDynamicPartitions should never return **null** (**Nothing** in Visual Basic). If they do, PLINQ / TPL will throw an InvalidOperationException.
- Methods that return partitions should always return partitions that can fully and uniquely enumerate the data source. There should be no duplication in the data source or skipped items unless specifically required by the design of the partitioner. If this rule is not followed, then the output order may be scrambled.

- The following Boolean getters must always accurately return the following values so that the output order is not scrambled:
  - **KeysOrderedInEachPartition**: Each partition returns elements with increasing key indices.
  - **KeysOrderedAcrossPartitions**: For all partitions that are returned, the key indices in partition *i* are higher than the key indices in partition *i*-1.
  - **KeysNormalized**: All key indices are monotonically increasing without gaps, starting from zero.
- All indices must be unique. There may not be duplicate indices. If this rule is not followed, then the output order may be scrambled.
- All indices must be nonnegative. If this rule is not followed, then PLINQ/TPL may throw exceptions.

See Also

Parallel Programming in the .NET Framework
How to: Implement Dynamic Partitions
How to: Implement a Partitioner for Static Partitioning

Lambda Expressions in PLINQ and TPL

.NET Framework (current version)

The Task Parallel Library (TPL) contains many methods that take one of the System.Func(Of TResult) or System.Action family of delegates as input parameters. You use these delegates to pass in your custom program logic to the parallel loop, task or query. The code examples for TPL as well as PLINQ use lambda expressions to create instances of those delegates as inline code blocks. This topic provides a brief introduction to Func and Action and shows you how to use lambda expressions in the Task Parallel Library and PLINQ.

Note For more information about delegates in general, see Delegates (C# Programming Guide) and Delegates (Visual Basic). For more information about lambda expressions in C# and Visual Basic, see Lambda Expressions (C# Programming Guide) and Lambda Expressions (Visual Basic).

Func Delegate

A **Func** delegate encapsulates a method that returns a value. In a Func signature, the last or rightmost type parameter always specifies the return type. One common cause of compiler errors is to attempt to pass in two input parameters to a System.Func(Of T, TResult); in fact this type takes only one input parameter. The Framework Class Library defines 17 versions of **Func**: System.Func(Of TResult), System.Func(Of T, TResult), System.Func(Of T1, T2, TResult), and so on up through System.Func(Of T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, TResult).

Action Delegate

A System.Action delegate encapsulates a method (Sub in Visual Basic) that does not return a value, or returns void. In an Action type signature, the type parameters represent only input parameters. Like Func, the Framework Class Library defines 17 versions of Action, from a version that has no type parameters up through a version that has 16 type parameters.

Example

The following example for the Parallel.ForEach(Of TSource, TLocal)(IEnumerable(Of TSource), Func(Of TLocal), Func(Of TSource, ParallelLoopState, TLocal, TLocal), Action(Of TLocal)) method shows how to express both Func and Action delegates by using lambda expressions.

VB

```
Imports System.Threading
Imports System.Threading.Tasks
Module ForEachDemo

    ' Demonstrated features:
    '   Parallel.ForEach()
    '   Thread-local state
```

```
    ' Expected results:
    '   This example sums up the elements of an int[] in parallel.
    '   Each thread maintains a local sum. When a thread is initialized, that local sum
is set to 0.
    '   On every iteration the current element is added to the local sum.
    '   When a thread is done, it safely adds its local sum to the global sum.
    '   After the loop is complete, the global sum is printed out.
    ' Documentation:
    '   http://msdn.microsoft.com/en-us/library/dd990270(VS.100).aspx
    Private Sub ForEachDemo()
        ' The sum of these elements is 40.
        Dim input As Integer() = {4, 1, 6, 2, 9, 5, _
        10, 3}
        Dim sum As Integer = 0

        Try
            ' source collection
            Parallel.ForEach(input,
                             Function()
                                 ' thread local initializer
                                 Return 0
                             End Function,
                             Function(n, loopState, localSum)
                                 ' body
                                 localSum += n
                                 Console.WriteLine("Thread={0}, n={1}, localSum={2}",
Thread.CurrentThread.ManagedThreadId, n, localSum)
                                 Return localSum
                             End Function,
                             Sub(localSum)
                                 ' thread local aggregator
                                 Interlocked.Add(sum, localSum)
                             End Sub)

            Console.WriteLine(vbLf & "Sum={0}", sum)
        Catch e As AggregateException
            ' No exception is expected in this example, but if one is still thrown from
a task,
            ' it will be wrapped in AggregateException and propagated to the main
thread.
            Console.WriteLine("Parallel.ForEach has thrown an exception. THIS WAS NOT
EXPECTED." & vbLf & "{0}", e)
        End Try
    End Sub


End Module
```

See Also

Parallel Programming in the .NET Framework