Contents

Parallel Programming
- Task Parallel Library (TPL)
  - Data Parallelism
    - How to: Write a Simple Parallel.For Loop
    - How to: Write a Simple Parallel.ForEach Loop
    - How to: Write a Parallel.For Loop with Thread-Local Variables
    - How to: Write a Parallel.ForEach Loop with Partition-Local Variables
    - How to: Cancel a Parallel.For or ForEach Loop
    - How to: Handle Exceptions in Parallel Loops
    - How to: Speed Up Small Loop Bodies
    - How to: Iterate File Directories with the Parallel Class
  - Task-based Asynchronous Programming
    - Chaining Tasks by Using Continuation Tasks
    - Attached and Detached Child Tasks
    - Task Cancellation
    - Exception Handling
    - How to: Use Parallel.Invoke to Execute Parallel Operations
    - How to: Return a Value from a Task
    - How to: Cancel a Task and Its Children
    - How to: Create Pre-Computed Tasks
    - How to: Traverse a Binary Tree with Parallel Tasks
    - How to: Unwrap a Nested Task
    - How to: Prevent a Child Task from Attaching to its Parent
  - Dataflow
    - How to: Write Messages to and Read Messages from a Dataflow Block
    - How to: Implement a Producer-Consumer Dataflow Pattern
    - How to: Perform Action When a Dataflow Block Receives Data
    - Walkthrough: Creating a Dataflow Pipeline
    - How to: Unlink Dataflow Blocks

- Walkthrough: Using Dataflow in a Windows Forms Application
- How to: Cancel a Dataflow Block
- Walkthrough: Creating a Custom Dataflow Block Type
- How to: Use JoinBlock to Read Data From Multiple Sources
- How to: Specify the Degree of Parallelism in a Dataflow Block
- How to: Specify a Task Scheduler in a Dataflow Block
- Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency

Using TPL with Other Asynchronous Patterns
- TPL and Traditional .NET Framework Asynchronous Programming
- How to: Wrap EAP Patterns in a Task

Potential Pitfalls in Data and Task Parallelism

Parallel LINQ (PLINQ)
- Introduction to PLINQ
- Understanding Speedup in PLINQ
- Order Preservation in PLINQ
- Merge Options in PLINQ
- Potential Pitfalls with PLINQ
- How to: Create and Execute a Simple PLINQ Query
- How to: Control Ordering in a PLINQ Query
- How to: Combine Parallel and Sequential LINQ Queries
- How to: Handle Exceptions in a PLINQ Query
- How to: Cancel a PLINQ Query
- How to: Write a Custom PLINQ Aggregate Function
- How to: Specify the Execution Mode in PLINQ
- How to: Specify Merge Options in PLINQ
- How to: Iterate File Directories with PLINQ
- How to: Measure PLINQ Query Performance
- PLINQ Data Sample

Data Structures for Parallel Programming

Parallel Diagnostic Tools

Custom Partitioners for PLINQ and TPL
- How to: Implement Dynamic Partitions

How to: Implement a Partitioner for Static Partitioning
Lambda Expressions in PLINQ and TPL
For Further Reading

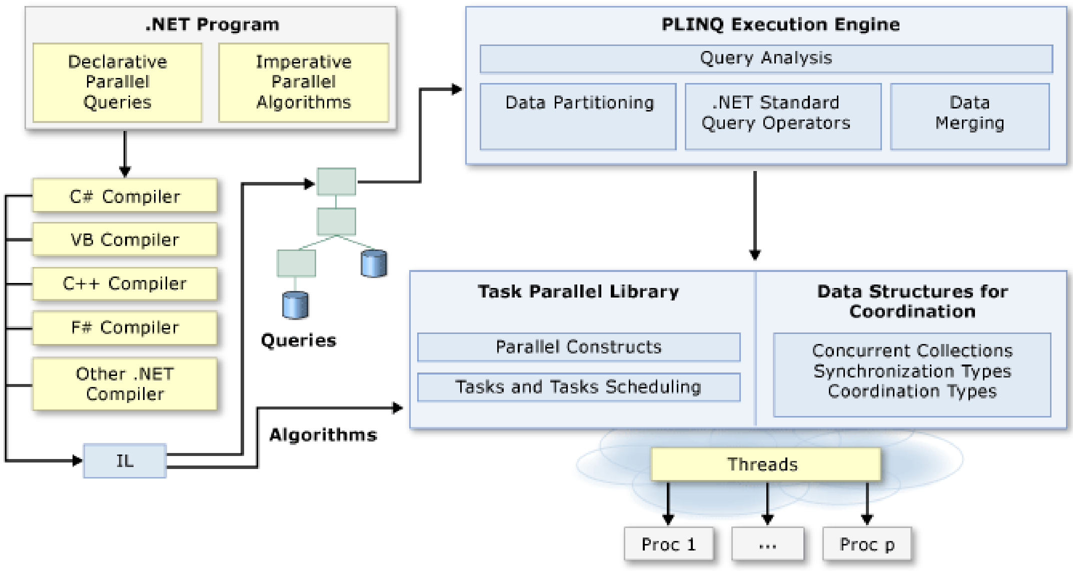
Parallel Programming in .NET

2/28/2019 • 2 minutes to read • Edit Online

Many personal computers and workstations have multiple CPU cores that enable multiple threads to be executed simultaneously. To take advantage of the hardware, 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 and the .NET Framework enhance support for parallel programming by providing a runtime, class library types, and diagnostic tools. These features, which were introduced with the .NET Framework 4, simplify parallel development. 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:



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.

TECHNOLOGY	DESCRIPTION
Parallel Diagnostic Tools	Provides links to documentation for Visual Studio debugger windows for tasks and parallel stacks, and for the Concurrency Visualizer.
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	Provides links to additional information and sample resources for parallel programming in .NET.

See also

- Async Overview
- Managed Threading

Task Parallel Library (TPL)

6/1/2019 • 2 minutes to read • Edit Online

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	Describes how to create parallel `for` and `foreach` loops (`For` and `For Each` in Visual Basic).
Task-based Asynchronous Programming	Describes how to create and run tasks implicitly by using Parallel.Invoke or explicitly by using Task objects directly.
Dataflow	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	Top level node for .NET parallel programming.

See also

- Samples for Parallel Programming with the .NET Framework

Data Parallelism (Task Parallel Library)

8/22/2019 • 3 minutes to read • Edit Online

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.

```
// Sequential version
foreach (var item in sourceCollection)
{
    Process(item);
}

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

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

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<T> source collection.
How to: Write a Simple Parallel.ForEach Loop	Describes how to write a ForEach loop over any IEnumerable<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 Partition-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	Introduces Parallel Programming in the .NET Framework.

See also

- Parallel Programming

How to: Write a Simple Parallel.For Loop

1/23/2019 • 9 minutes to read • Edit Online

This topic contains two examples that illustrate the Parallel.For method. The first uses the Parallel.For(Int64, Int64, Action<Int64>) method overload, and the second uses the Parallel.For(Int32, Int32, Action<Int32>) overload, the two simplest overloads of the Parallel.For method. You can use these two overloads of the Parallel.For method when you do not need to cancel the loop, break out of the loop iterations, or maintain any thread-local state.

> **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.

The first example calculates the size of files in a single directory. The second computes the product of two matrices.

Directory size example

This example is a simple command-line utility that calculates the total size of files in a directory. It expects a single directory path as an argument, and reports the number and total size of the files in that directory. After verifying that the directory exists, it uses the Parallel.For method to enumerate the files in the directory and determine their file sizes. Each file size is then added to the `totalSize` variable. Note that the addition is performed by calling the Interlocked.Add so that the addition is performed as an atomic operation. Otherwise, multiple tasks could try to update the `totalSize` variable simultaneously.

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      long totalSize = 0;

      String[] args = Environment.GetCommandLineArgs();
      if (args.Length == 1) {
         Console.WriteLine("There are no command line arguments.");
         return;
      }
      if (! Directory.Exists(args[1])) {
         Console.WriteLine("The directory does not exist.");
         return;
      }

      String[] files = Directory.GetFiles(args[1]);
      Parallel.For(0, files.Length,
                   index => { FileInfo fi = new FileInfo(files[index]);
                              long size = fi.Length;
                              Interlocked.Add(ref totalSize, size);
                   } );
      Console.WriteLine("Directory '{0}':", args[1]);
      Console.WriteLine("{0:N0} files, {1:N0} bytes", files.Length, totalSize);
   }
}
// The example displaysoutput like the following:
//       Directory 'c:\windows\':
//       32 files, 6,587,222 bytes
```

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

Module Example
   Public Sub Main()
      Dim totalSize As Long = 0

      Dim args() As String = Environment.GetCommandLineArgs()
      If args.Length = 1 Then
         Console.WriteLine("There are no command line arguments.")
         Return
      End If
      If Not Directory.Exists(args(1))
         Console.WriteLine("The directory does not exist.")
         Return
      End If

      Dim files() As String = Directory.GetFiles(args(1))
      Parallel.For(0, files.Length,
                   Sub(index As Integer)
                      Dim fi As New FileInfo(files(index))
                      Dim size As Long = fi.Length
                      Interlocked.Add(totalSize, size)
                   End Sub)
      Console.WriteLine("Directory '{0}':", args(1))
      Console.WriteLine("{0:N0} files, {1:N0} bytes", files.Length, totalSize)
   End Sub
End Module
' The example displays output like the following:
'       Directory 'c:\windows\':
'       32 files, 6,587,222 bytes
```

Matrix and stopwatch example

This example uses the Parallel.For method to compute the product of two matrices. It also shows how to use the System.Diagnostics.Stopwatch class to compare the performance of a parallel loop with a non-parallel loop. Note that, because it can generate a large volume of output, the example allows output to be redirected to a file.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

class MultiplyMatrices
{
    #region Sequential_Loop
    static void MultiplyMatricesSequential(double[,] matA, double[,] matB,
                                            double[,] result)
    {
        int matACols = matA.GetLength(1);
        int matBCols = matB.GetLength(1);
        int matARows = matA.GetLength(0);

        for (int i = 0; i < matARows; i++)
        {
            for (int j = 0; j < matBCols; j++)
            {
                double temp = 0;
                for (int k = 0; k < matACols; k++)
                {
                   temp += matA[i, k] * matB[k, j];
                }
                result[i, j] += temp;
            }
        }
    }
    #endregion

    #region Parallel_Loop
    static void MultiplyMatricesParallel(double[,] matA, double[,] matB, double[,] result)
    {
        int matACols = matA.GetLength(1);
        int matBCols = matB.GetLength(1);
        int matARows = matA.GetLength(0);

        // A basic matrix multiplication.
        // Parallelize the outer loop to partition the source array by rows.
        Parallel.For(0, matARows, i =>
        {
            for (int j = 0; j < matBCols; j++)
            {
                double temp = 0;
                for (int k = 0; k < matACols; k++)
                {
                    temp += matA[i, k] * matB[k, j];
                }
                result[i, j] = temp;
            }
        }); // Parallel.For
    }
    #endregion


    #region Main
    static void Main(string[] args)
    {
        // Set up matrices. Use small values to better view
        // result matrix. Increase the counts to see greater
        // speedup in the parallel loop vs. the sequential loop.
        int colCount = 180;
        int rowCount = 2000;
        int colCount2 = 270;
        double[,] m1 = InitializeMatrix(rowCount, colCount);
        double[,] m2 = InitializeMatrix(colCount, colCount2);
        double[,] result = new double[rowCount, colCount2];

        // First do the sequential version.
        Console.Error.WriteLine("Executing sequential loop...");
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        MultiplyMatricesSequential(m1, m2, result);
        stopwatch.Stop();
        Console.Error.WriteLine("Sequential loop time in milliseconds: {0}",
                                stopwatch.ElapsedMilliseconds);

        // For the skeptics.
        OfferToPrint(rowCount, colCount2, result);

        // Reset timer and results matrix.
        stopwatch.Reset();
        result = new double[rowCount, colCount2];

        // Do the parallel loop.
        Console.Error.WriteLine("Executing parallel loop...");
        stopwatch.Start();
        MultiplyMatricesParallel(m1, m2, result);
        stopwatch.Stop();
        Console.Error.WriteLine("Parallel loop time in milliseconds: {0}",
                                stopwatch.ElapsedMilliseconds);
        OfferToPrint(rowCount, colCount2, result);

        // Keep the console window open in debug mode.
        Console.Error.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    #endregion

    #region Helper_Methods
    static double[,] InitializeMatrix(int rows, int cols)
    {
        double[,] matrix = new double[rows, cols];

        Random r = new Random();
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i, j] = r.Next(100);
            }
        }
        return matrix;
    }

    private static void OfferToPrint(int rowCount, int colCount, double[,] matrix)
    {
        Console.Error.Write("Computation complete. Print results (y/n)? ");
        char c = Console.ReadKey(true).KeyChar;
        Console.Error.WriteLine(c);
        if (Char.ToUpperInvariant(c) == 'Y')
        {
            if (! Console.IsOutputRedirected) Console.WindowWidth = 180;
            Console.WriteLine();
            for (int x = 0; x < rowCount; x++)
            {
                Console.WriteLine("ROW {0}: ", x);
                for (int y = 0; y < colCount; y++)
                {
                    Console.Write("{0:#.##} ", matrix[x, y]);
                }
                Console.WriteLine();
            }
        }
    }
    #endregion
}
```

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

Module MultiplyMatrices
#Region "Sequential_Loop"
    Sub MultiplyMatricesSequential(ByVal matA As Double(,), ByVal matB As Double(,), ByVal result As
Double(,))
        Dim matACols As Integer = matA.GetLength(1)
        Dim matBCols As Integer = matB.GetLength(1)
        Dim matARows As Integer = matA.GetLength(0)

        For i As Integer = 0 To matARows - 1
            For j As Integer = 0 To matBCols - 1
                Dim temp As Double = 0
                For k As Integer = 0 To matACols - 1
                   temp += matA(i, k) * matB(k, j)
                Next
                result(i, j) += temp
            Next
        Next
    End Sub
#End Region
```

```
#Region "Parallel_Loop"
    Private Sub MultiplyMatricesParallel(ByVal matA As Double(,), ByVal matB As Double(,), ByVal result As
Double(,))
        Dim matACols As Integer = matA.GetLength(1)
        Dim matBCols As Integer = matB.GetLength(1)
        Dim matARows As Integer = matA.GetLength(0)

        ' A basic matrix multiplication.
        ' Parallelize the outer loop to partition the source array by rows.
        Parallel.For(0, matARows, Sub(i)
                                      For j As Integer = 0 To matBCols - 1
                                          Dim temp As Double = 0
                                          For k As Integer = 0 To matACols - 1
                                              temp += matA(i, k) * matB(k, j)
                                          Next
                                          result(i, j) += temp
                                      Next
                                  End Sub)
    End Sub
#End Region

#Region "Main"
    Sub Main(ByVal args As String())
        ' Set up matrices. Use small values to better view
        ' result matrix. Increase the counts to see greater
        ' speedup in the parallel loop vs. the sequential loop.
        Dim colCount As Integer = 180
        Dim rowCount As Integer = 2000
        Dim colCount2 As Integer = 270
        Dim m1 As Double(,) = InitializeMatrix(rowCount, colCount)
        Dim m2 As Double(,) = InitializeMatrix(colCount, colCount2)
        Dim result As Double(,) = New Double(rowCount - 1, colCount2 - 1) {}

        ' First do the sequential version.
        Console.Error.WriteLine("Executing sequential loop...")
        Dim stopwatch As New Stopwatch()
        stopwatch.Start()

        MultiplyMatricesSequential(m1, m2, result)
        stopwatch.[Stop]()
        Console.Error.WriteLine("Sequential loop time in milliseconds: {0}", stopwatch.ElapsedMilliseconds)

        ' For the skeptics.
        OfferToPrint(rowCount, colCount2, result)

        ' Reset timer and results matrix.
        stopwatch.Reset()
        result = New Double(rowCount - 1, colCount2 - 1) {}

        ' Do the parallel loop.
        Console.Error.WriteLine("Executing parallel loop...")
        stopwatch.Start()
        MultiplyMatricesParallel(m1, m2, result)
        stopwatch.[Stop]()
        Console.Error.WriteLine("Parallel loop time in milliseconds: {0}", stopwatch.ElapsedMilliseconds)
        OfferToPrint(rowCount, colCount2, result)

        ' Keep the console window open in debug mode.
        Console.Error.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub
#End Region

#Region "Helper_Methods"
    Function InitializeMatrix(ByVal rows As Integer, ByVal cols As Integer) As Double(,)
        Dim matrix As Double(,) = New Double(rows - 1, cols - 1) {}

        Dim r As New Random()
        For i As Integer = 0 To rows - 1
```

```
            For j As Integer = 0 To cols - 1
                matrix(i, j) = r.[Next](100)
            Next
        Next
        Return matrix
    End Function

    Sub OfferToPrint(ByVal rowCount As Integer, ByVal colCount As Integer, ByVal matrix As Double(,))
        Console.Error.Write("Computation complete. Display results (y/n)? ")
        Dim c As Char = Console.ReadKey(True).KeyChar
        Console.Error.WriteLine(c)
        If Char.ToUpperInvariant(c) = "Y"c Then
            If Not Console.IsOutputRedirected Then Console.WindowWidth = 168
            Console.WriteLine()
            For x As Integer = 0 To rowCount - 1
                Console.WriteLine("ROW {0}: ", x)
                For y As Integer = 0 To colCount - 1
                    Console.Write("{0:#.##} ", matrix(x, y))
                Next
                Console.WriteLine()
            Next
        End If
    End Sub
#End Region
End Module
```

When parallelizing any code, including loops, one important goal is to utilize the processors as much as possible without over parallelizing to the point where the overhead for parallel processing negates any performance benefits. In this particular example, only the outer loop is parallelized because there is not very much work performed in the inner loop. The combination of a small amount of work and undesirable cache effects can result in performance degradation in nested parallel loops. Therefore, parallelizing the outer loop only is the best way to maximize the benefits of concurrency on most systems.

The Delegate

The third parameter of this overload of For is a delegate of type `Action<int>` in C# or `Action(Of Integer)` in Visual Basic. An `Action` delegate, whether it has zero, one or sixteen type parameters, always returns void. In Visual Basic, the behavior of an `Action` is defined with a `Sub`. The example uses a lambda expression to create the delegate, but you can create the delegate in other ways as well. For more information, see Lambda Expressions in PLINQ and TPL.

The Iteration Value

The delegate takes a single input parameter whose value is the current iteration. This iteration value is supplied by the runtime and its starting value is the index of the first element on the segment (partition) of the source that is being processed on the current thread.

If you require more control over the concurrency level, use one of the overloads that takes a System.Threading.Tasks.ParallelOptions input parameter, such as: Parallel.For(Int32, Int32, ParallelOptions, Action<Int32,ParallelLoopState>).

Return Value and Exception Handling

For returns a System.Threading.Tasks.ParallelLoopResult object when all threads have completed. This return value is useful when you are stopping or breaking loop iteration manually, because the ParallelLoopResult stores information such as the last iteration that ran to completion. If one or more exceptions occur on one of the threads, a System.AggregateException will be thrown.

In the code in this example, the return value of For is not used.

Analysis and Performance

You can use the Performance Wizard to view CPU usage on your computer. As an experiment, increase the number of columns and rows in the matrices. The larger the matrices, the greater the performance difference between the parallel and sequential versions of the computation. When the matrix is small, the sequential version will run faster because of the overhead in setting up the parallel loop.

Synchronous calls to shared resources, like the Console or the File System, will significantly degrade the performance of a parallel loop. When measuring performance, try to avoid calls such as Console.WriteLine within the loop.

Compile the Code

Copy and paste this code into a Visual Studio project.

See also

- For
- ForEach
- Data Parallelism
- Parallel Programming

How to: Write a simple Parallel.ForEach loop

10/3/2019 • 3 minutes to read • Edit Online

This example shows how to use a Parallel.ForEach loop to enable data parallelism over any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> data source.

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.

Example

This example assumes you have several .jpg files in a *C:\Users\Public\Pictures\Sample Pictures* folder and creates a new sub-folder named *Modified*. When you run the example, it rotates each .jpg image in *Sample Pictures* and saves it to *Modified*. You can modify the two paths as necessary.

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Drawing;

public class Example
{
    public static void Main()
    {
        // A simple source for demonstration purposes. Modify this path as necessary.
        string[] files = Directory.GetFiles(@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg");
        string newDir = @"C:\Users\Public\Pictures\Sample Pictures\Modified";
        Directory.CreateDirectory(newDir);

        // Method signature: Parallel.ForEach(IEnumerable<TSource> source, Action<TSource> body)
        Parallel.ForEach(files, (currentFile) =>
                                {
                                    // The more computational work you do here, the greater
                                    // the speedup compared to a sequential foreach loop.
                                    string filename = Path.GetFileName(currentFile);
                                    var bitmap = new Bitmap(currentFile);

                                    bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
                                    bitmap.Save(Path.Combine(newDir, filename));

                                    // Peek behind the scenes to see how work is parallelized.
                                    // But be aware: Thread contention for the Console slows down parallel
loops!!!

                                    Console.WriteLine($"Processing {filename} on thread
{Thread.CurrentThread.ManagedThreadId}");
                                    //close lambda expression and method invocation
                                });


        // Keep the console window open in debug mode.
        Console.WriteLine("Processing complete. Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
Imports System.IO
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Drawing

Module ForEachDemo

    Sub Main()
        ' A simple source for demonstration purposes. Modify this path as necessary.
        Dim files As String() = Directory.GetFiles("C:\Users\Public\Pictures\Sample Pictures", "*.jpg")
        Dim newDir As String = "C:\Users\Public\Pictures\Sample Pictures\Modified"
        Directory.CreateDirectory(newDir)

        Parallel.ForEach(files, Sub(currentFile)
                                    ' The more computational work you do here, the greater
                                    ' the speedup compared to a sequential foreach loop.
                                    Dim filename As String = Path.GetFileName(currentFile)
                                    Dim bitmap As New Bitmap(currentFile)

                                    bitmap.RotateFlip(System.Drawing.RotateFlipType.Rotate180FlipNone)
                                    bitmap.Save(Path.Combine(newDir, filename))

                                    ' Peek behind the scenes to see how work is parallelized.
                                    ' But be aware: Thread contention for the Console slows down parallel
loops!!!

                                    Console.WriteLine($"Processing {filename} on thread
{Thread.CurrentThread.ManagedThreadId}")
                                    'close lambda expression and method invocation
                                End Sub)


        ' Keep the console window open in debug mode.
        Console.WriteLine("Processing complete. Press any key to exit.")
        Console.ReadKey()
    End Sub
End Module
```

A Parallel.ForEach loop works like a Parallel.For loop. The loop partitions the source collection and schedules the work on multiple threads based on the system environment. The more processors on the system, the faster the parallel method runs. For some source collections, a sequential loop may be faster, depending on the size of the source and the kind of work the loop performs. For more information about performance, see Potential pitfalls in data and task parallelism.

For more information about parallel loops, see How to: Write a simple Parallel.For loop.

To use Parallel.ForEach with a non-generic collection, you can use the Enumerable.Cast extension method to convert the collection to a generic collection, as shown in the following example:

```
Parallel.ForEach(nonGenericCollection.Cast<object>(),
    currentElement =>
    {
    });
```

```
Parallel.ForEach(nonGenericCollection.Cast(Of Object), _
                 Sub(currentElement)
                     ' ... work with currentElement
                 End Sub)
```

You can also use Parallel LINQ (PLINQ) to parallelize processing of IEnumerable<T> data sources. PLINQ enables you to use declarative query syntax to express the loop behavior. For more information, see Parallel LINQ

(PLINQ).

Compile and run the code

You can compile the code as a console application for .NET Framework or as a console application for .NET Core.

In Visual Studio, there are Visual Basic and C# console application templates for Windows Desktop and .NET Core.

From the command line, you can use either .NET Core and its CLI tools (for example, `dotnet new console` or `dotnet new console -lang vb`), or you can create the file and use the command-line compiler for a .NET Framework application.

For a .NET Core project, you must reference the **System.Drawing.Common** NuGet package. In Visual Studio, use the NuGet Package Manager to install the package. Alternatively, you can add a reference to the package in your *.csproj or *.vbproj file:

```
<ItemGroup>
     <PackageReference Include="System.Drawing.Common" Version="4.5.1" />
</ItemGroup>
```

To run a .NET Core console application from the command line, use `dotnet run` from the folder that contains your application.

To run your console application from Visual Studio, press **F5**.

See also

- Data parallelism
- Parallel programming
- Parallel LINQ (PLINQ)

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

1/23/2019 • 3 minutes to read • Edit Online

This example shows how to use thread-local variables to store and retrieve state in each separate task that is created by a For loop. By using thread-local data, you can avoid the overhead of synchronizing a large number of accesses to shared state. Instead of writing to a shared resource on each iteration, you compute and store the value until all iterations for the task are complete. You can then write the final result once to the shared resource, or pass it to another method.

Example

The following example calls the For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) method to calculate the sum of the values in an array that contains one million elements. The value of each element is equal to its index.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;


class Test
{
    static void Main()
    {
        int[] nums = Enumerable.Range(0, 1000000).ToArray();
        long total = 0;

        // Use type parameter to make subtotal a long, not an int
        Parallel.For<long>(0, nums.Length, () => 0, (j, loop, subtotal) =>
        {
            subtotal += nums[j];
            return subtotal;
        },
            (x) => Interlocked.Add(ref total, x)
        );

        Console.WriteLine("The total is {0:N0}", total);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

```
'How to: Write a Parallel.For Loop That Has Thread-Local Variables

Imports System.Threading
Imports System.Threading.Tasks

Module ForWithThreadLocal

    Sub Main()
        Dim nums As Integer() = Enumerable.Range(0, 1000000).ToArray()
        Dim total As Long = 0

        ' Use type parameter to make subtotal a Long type. Function will overflow otherwise.
        Parallel.For(Of Long)(0, nums.Length, Function() 0, Function(j, [loop], subtotal)
                                                                subtotal += nums(j)
                                                                Return subtotal
                                                            End Function, Function(x) Interlocked.Add(total,
x))

        Console.WriteLine("The total is {0:N0}", total)
        Console.WriteLine("Press any key to exit")
        Console.ReadKey()
    End Sub

End Module
```

The first two parameters of every For method specify the beginning and ending iteration values. In this overload of the method, the third parameter is where you initialize your local state. In this context, local state means a variable whose lifetime extends from just before the first iteration of the loop on the current thread, to just after the last iteration.

The type of the third parameter is a Func<TResult> where `TResult` is the type of the variable that will store the thread-local state. Its type is defined by the generic type argument supplied when calling the generic For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) method, which in this case is Int64. The type argument tells the compiler the type of the temporary variable that will be used to store the thread-local state. In this example, the expression `() => 0` (or `Function() 0` in Visual Basic) initializes the thread-local variable to zero. If the generic type argument is a reference type or user-defined value type, the expression would look like this:

```
() => new MyClass()
```

```
Function() new MyClass()
```

The fourth parameter defines the loop logic. It must be a delegate or lambda expression whose signature is `Func<int, ParallelLoopState, long, long>` in C# or `Func(Of Integer, ParallelLoopState, Long, Long)` in Visual Basic. The first parameter is the value of the loop counter for that iteration of the loop. The second is a ParallelLoopState object that can be used to break out of the loop; this object is provided by the Parallel class to each occurrence of the loop. The third parameter is the thread-local variable. The last parameter is the return type. In this case, the type is Int64 because that is the type we specified in the For type argument. That variable is named `subtotal` and is returned by the lambda expression. The return value is used to initialize `subtotal` on each subsequent iteration of the loop. You can also think of this last parameter as a value that is passed to each iteration, and then passed to the `localFinally` delegate when the last iteration is complete.

The fifth parameter defines the method that is called once, after all the iterations on a particular thread have completed. The type of the input argument again corresponds to the type argument of the For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) method and the type returned by the body lambda expression. In this example, the value is added to a variable at class scope in a thread safe way by calling the Interlocked.Add method. By using a thread-local variable, we have avoided writing to this class variable on every iteration of the loop.

For more information about how to use lambda expressions, see Lambda Expressions in PLINQ and TPL.

See also

- Data Parallelism
- Parallel Programming
- Task Parallel Library (TPL)
- Lambda Expressions in PLINQ and TPL

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

7/9/2019 • 3 minutes to read • Edit Online

The following example shows how to write a ForEach method that uses partition-local variables. When a ForEach loop executes, it divides its source collection into multiple partitions. Each partition has its own copy of the partition-local variable. A partition-local variable is similar to a thread-local variable, except that multiple partitions can run on a single thread.

The code and parameters in this example closely resemble the corresponding For method. For more information, see How to: Write a Parallel.For Loop with Thread-Local Variables.

To use a partition-local variable in a ForEach loop, you must call one of the method overloads that takes two type parameters. The first type parameter, `TSource`, specifies the type of the source element, and the second type parameter, `TLocal`, specifies the type of the partition-local variable.

Example

The following example calls the Parallel.ForEach<TSource,TLocal>(IEnumerable<TSource>, Func<TLocal>, Func<TSource,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) overload to compute the sum of an array of one million elements. This overload has four parameters:

- `source`, which is the data source. It must implement IEnumerable<T>. The data source in our example is the one million member `IEnumerable<Int32>` object returned by the Enumerable.Range method.
- `localInit`, or the function that initializes the partition-local variable. This function is called once for each partition in which the Parallel.ForEach operation executes. Our example initializes the partition-local variable to zero.
- `body`, a Func<T1,T2,T3,TResult> that is invoked by the parallel loop on each iteration of the loop. Its signature is `Func\<TSource, ParallelLoopState, TLocal, TLocal>`. You supply the code for the delegate, and the loop passes in the input parameters, which are:
  - The current element of the IEnumerable<T>.
  - A ParallelLoopState variable that you can use in your delegate's code to examine the state of the loop.
  - The partition-local variable.

  Your delegate returns the partition-local variable, which is then passed to the next iteration of the loop that executes in that particular partition. Each loop partition maintains a separate instance of this variable.

  In the example, the delegate adds the value of each integer to the partition-local variable, which maintains a running total of the values of the integer elements in that partition.
- `localFinally`, an `Action<TLocal>` delegate that the Parallel.ForEach invokes when the looping operations in each partition have completed. The Parallel.ForEach method passes your `Action<TLocal>` delegate the final value of the partition-local variable for this loop partition, and you provide the code that performs the required action for combining the result from this partition with the results from the other partitions. This delegate can be invoked concurrently by multiple tasks. Because of this, the example uses the Interlocked.Add(Int32, Int32) method to synchronize access to the `total` variable. Because the delegate type is an Action<T>, there is no return value.

```
 using System;
 using System.Linq;
 using System.Threading;
 using System.Threading.Tasks;

 class Test
 {
     static void Main()
     {
         int[] nums = Enumerable.Range(0, 1000000).ToArray();
         long total = 0;

         // First type parameter is the type of the source elements
         // Second type parameter is the type of the thread-local variable (partition subtotal)
         Parallel.ForEach<int, long>(nums, // source collection
                                     () => 0, // method to initialize the local variable
                                     (j, loop, subtotal) => // method invoked by the loop on each iteration
                                     {
                                         subtotal += j; //modify local variable
                                         return subtotal; // value to be passed to next iteration
                                     },
             // Method to be executed when each partition has completed.
             // finalResult is the final value of subtotal for a particular partition.
                                     (finalResult) => Interlocked.Add(ref total, finalResult)
                                     );

         Console.WriteLine("The total from Parallel.ForEach is {0:N0}", total);
    }
}
// The example displays the following output:
//        The total from Parallel.ForEach is 499,999,500,000
```

```
' How to: Write a Parallel.ForEach Loop That Has Thread-Local Variables

Imports System.Threading
Imports System.Threading.Tasks

Module ForEachThreadLocal
    Sub Main()

        Dim nums() As Integer = Enumerable.Range(0, 1000000).ToArray()
        Dim total As Long = 0

        ' First type paramemter is the type of the source elements
        ' Second type parameter is the type of the thread-local variable (partition subtotal)
        Parallel.ForEach(Of Integer, Long)(nums, Function() 0,
                                           Function(elem, loopState, subtotal)
                                               subtotal += elem
                                               Return subtotal
                                           End Function,
                                            Sub(finalResult)
                                                Interlocked.Add(total, finalResult)
                                            End Sub)

        Console.WriteLine("The result of Parallel.ForEach is {0:N0}", total)
    End Sub
End Module
' The example displays the following output:
'       The result of Parallel.ForEach is 499,999,500,000
```

See also

- Data Parallelism

- How to: Write a Parallel.For Loop with Thread-Local Variables
- Lambda Expressions in PLINQ and TPL

How to: Cancel a Parallel.For or ForEach Loop

1/23/2019 • 2 minutes to read • Edit Online

The Parallel.For and Parallel.ForEach methods support cancellation through the use of cancellation tokens. For more information about cancellation in general, see Cancellation. In a parallel loop, you supply the CancellationToken to the method in the ParallelOptions parameter and then enclose the parallel call in a try-catch block.

Example

The following example shows how to cancel a call to Parallel.ForEach. You can apply the same approach to a Parallel.For call.

```
namespace CancelParallelLoops
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;

    class Program
    {
        static void Main()
        {
            int[] nums = Enumerable.Range(0, 10000000).ToArray();
            CancellationTokenSource cts = new CancellationTokenSource();

           // Use ParallelOptions instance to store the CancellationToken
            ParallelOptions po = new ParallelOptions();
            po.CancellationToken = cts.Token;
            po.MaxDegreeOfParallelism = System.Environment.ProcessorCount;
            Console.WriteLine("Press any key to start. Press 'c' to cancel.");
            Console.ReadKey();

            // Run a task so that we can cancel from another thread.
            Task.Factory.StartNew(() =>
            {
                if (Console.ReadKey().KeyChar == 'c')
                    cts.Cancel();
                Console.WriteLine("press any key to exit");
            });

            try
            {
                Parallel.ForEach(nums, po, (num) =>
                {
                    double d = Math.Sqrt(num);
                    Console.WriteLine("{0} on {1}", d, Thread.CurrentThread.ManagedThreadId);
                    po.CancellationToken.ThrowIfCancellationRequested();
                });
            }
            catch (OperationCanceledException e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                cts.Dispose();
            }

            Console.ReadKey();
        }
    }
}
```

```
' How to: Cancel a Parallel.For or ForEach Loop

Imports System.Threading
Imports System.Threading.Tasks

Module CancelParallelLoops
    Sub Main()
        Dim nums() As Integer = Enumerable.Range(0, 10000000).ToArray()
        Dim cts As New CancellationTokenSource

         ' Use ParallelOptions instance to store the CancellationToken
        Dim po As New ParallelOptions
        po.CancellationToken = cts.Token
        po.MaxDegreeOfParallelism = System.Environment.ProcessorCount
        Console.WriteLine("Press any key to start. Press 'c' to cancel.")
        Console.ReadKey()

        ' Run a task so that we can cancel from another thread.
        Dim t As Task = Task.Factory.StartNew(Sub()
                                                  If Console.ReadKey().KeyChar = "c"c Then
                                                      cts.Cancel()
                                                  End If
                                                  Console.WriteLine(vbCrLf & "Press any key to exit.")
                                              End Sub)

        Try

            ' The error "Exception is unhandled by user code" will appear if "Just My Code"
            ' is enabled. This error is benign. You can press F5 to continue, or disable Just My Code.
            Parallel.ForEach(nums, po, Sub(num)
                                           Dim d As Double = Math.Sqrt(num)
                                           Console.CursorLeft = 0
                                           Console.Write("{0:##.##} on {1}", d,
Thread.CurrentThread.ManagedThreadId)
                                           po.CancellationToken.ThrowIfCancellationRequested()
                                       End Sub)

        Catch e As OperationCanceledException
            Console.WriteLine(e.Message)
        Finally
            cts.Dispose()
        End Try

        Console.ReadKey()

    End Sub
End Module
```

If the token that signals the cancellation is the same token that is specified in the ParallelOptions instance, then the parallel loop will throw a single OperationCanceledException on cancellation. If some other token causes cancellation, the loop will throw an AggregateException with an OperationCanceledException as an InnerException.

See also

- Data Parallelism
- Lambda Expressions in PLINQ and TPL

How to: Handle Exceptions in Parallel Loops

8/22/2019 • 3 minutes to read • Edit Online

The Parallel.For and Parallel.ForEach overloads do not have any special mechanism to handle exceptions that might be thrown. In this respect, they resemble regular `for` and `foreach` loops (`For` and `For Each` in Visual Basic); an unhandled exception causes the loop to terminate immediately.

When you add your own exception-handling logic to parallel loops, handle the case in which similar exceptions might be thrown on multiple threads concurrently, and the case in which an exception thrown on one thread causes another exception to be thrown on another thread. You can handle both cases by wrapping all exceptions from the loop in a System.AggregateException. The following example shows one possible approach.

NOTE

When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the example below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

Example

In this example, all exceptions are caught and then wrapped in an System.AggregateException which is thrown. The caller can decide which exceptions to handle.

```
class ExceptionDemo2
{
    static void Main(string[] args)
    {
        // Create some random data to process in parallel.
        // There is a good probability this data will cause some exceptions to be thrown.
        byte[] data = new byte[5000];
        Random r = new Random();
        r.NextBytes(data);

        try
        {
            ProcessDataInParallel(data);
        }
        catch (AggregateException ae)
        {
            var ignoredExceptions = new List<Exception>();
            // This is where you can choose which exceptions to handle.
            foreach (var ex in ae.Flatten().InnerExceptions)
            {
                if (ex is ArgumentException)
                    Console.WriteLine(ex.Message);
                else
                    ignoredExceptions.Add(ex);
            }
            if (ignoredExceptions.Count > 0) throw new AggregateException(ignoredExceptions);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    private static void ProcessDataInParallel(byte[] data)
    {
        // Use ConcurrentQueue to enable safe enqueueing from multiple threads.
        var exceptions = new ConcurrentQueue<Exception>();

        // Execute the complete loop and capture all exceptions.
        Parallel.ForEach(data, d =>
        {
            try
            {
                // Cause a few exceptions, but not too many.
                if (d < 3)
                    throw new ArgumentException($"Value is {d}. Value must be greater than or equal to 3.");
                else
                    Console.Write(d + " ");
            }
            // Store the exception and continue with the loop.
            catch (Exception e)
            {
                exceptions.Enqueue(e);
            }
        });
        Console.WriteLine();

        // Throw the exceptions here after the loop completes.
        if (exceptions.Count > 0) throw new AggregateException(exceptions);
    }
}
```

```
' How to: Handle Exceptions in Parallel Loops

Imports System.Collections.Concurrent
Imports System.Collections.Generic
Imports System.Threading.Tasks

Module ExceptionsInLoops

    Sub Main()

        ' Create some random data to process in parallel.
        ' There is a good probability this data will cause some exceptions to be thrown.
        Dim data(1000) As Byte
        Dim r As New Random()
        r.NextBytes(data)

        Try
            ProcessDataInParallel(data)
        Catch ae As AggregateException
            Dim ignoredExceptions As New List(Of Exception)
            ' This is where you can choose which exceptions to handle.
            For Each ex As Exception In ae.Flatten().InnerExceptions
                If (TypeOf (ex) Is ArgumentException) Then
                    Console.WriteLine(ex.Message)
                Else
                    ignoredExceptions.Add(ex)
                End If
            Next
            If ignoredExceptions.Count > 0 Then
                Throw New AggregateException(ignoredExceptions)
            End If
        End Try
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub
    Sub ProcessDataInParallel(ByVal data As Byte())

        ' Use ConcurrentQueue to enable safe enqueueing from multiple threads.
        Dim exceptions As New ConcurrentQueue(Of Exception)

        ' Execute the complete loop and capture all exceptions.
        Parallel.ForEach(Of Byte)(data, Sub(d)
                                            Try
                                                ' Cause a few exceptions, but not too many.
                                                If d < 3 Then
                                                    Throw New ArgumentException($"Value is {d}. Value must be
greater than or equal to 3")
                                                Else
                                                    Console.Write(d & " ")
                                                End If
                                            Catch ex As Exception
                                                ' Store the exception and continue with the loop.
                                                exceptions.Enqueue(ex)
                                            End Try
                                        End Sub)
        Console.WriteLine()
        ' Throw the exceptions here after the loop completes.
        If exceptions.Count > 0 Then
            Throw New AggregateException(exceptions)
        End If
    End Sub
End Module
```

See also

- Data Parallelism
- Lambda Expressions in PLINQ and TPL

How to: Speed Up Small Loop Bodies

2/28/2019 • 2 minutes to read • Edit Online

When a Parallel.For loop has a small body, it might perform more slowly than the equivalent sequential loop, such as the for loop in C# and the For loop in Visual Basic. Slower performance is caused by the overhead involved in partitioning the data and the cost of invoking a delegate on each loop iteration. To address such scenarios, the Partitioner class provides the Partitioner.Create method, which enables you to provide a sequential loop for the delegate body, so that the delegate is invoked only once per partition, instead of once per iteration. For more information, see Custom Partitioners for PLINQ and TPL.

Example

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {

        // Source must be array or IList.
        var source = Enumerable.Range(0, 100000).ToArray();

        // Partition the entire source array.
        var rangePartitioner = Partitioner.Create(0, source.Length);

        double[] results = new double[source.Length];

        // Loop over the partitions in parallel.
        Parallel.ForEach(rangePartitioner, (range, loopState) =>
        {
            // Loop over each range element without a delegate invocation.
            for (int i = range.Item1; i < range.Item2; i++)
            {
                results[i] = source[i] * Math.PI;
            }
        });

        Console.WriteLine("Operation complete. Print results? y/n");
        char input = Console.ReadKey().KeyChar;
        if (input == 'y' || input == 'Y')
        {
            foreach(double d in results)
            {
                Console.Write("{0} ", d);
            }
        }
    }
}
```

```
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
```

The approach demonstrated in this example is useful when the loop performs a minimal amount of work. As the work becomes more computationally expensive, you will probably get the same or better performance by using a For or ForEach loop with the default partitioner.

See also

- Data Parallelism
- Custom Partitioners for PLINQ and TPL
- Iterators (C#)
- Iterators (Visual Basic)
- Lambda Expressions in PLINQ and TPL

How to: Iterate File Directories with the Parallel Class

2/28/2019 • 5 minutes to read • Edit Online

In many cases, file iteration is an operation that can be easily parallelized. The topic How to: Iterate File Directories with PLINQ shows the easiest way to perform this task for many scenarios. However, complications can arise when your code has to deal with the many types of exceptions that can arise when accessing the file system. The following example shows one approach to the problem. It uses a stack-based iteration to traverse all files and folders under a specified directory, and it enables your code to catch and handle various exceptions. Of course, the way that you handle the exceptions is up to you.

Example

The following example iterates the directories sequentially, but processes the files in parallel. This is probably the best approach when you have a large file-to-directory ratio. It is also possible to parallelize the directory iteration, and access each file sequentially. It is probably not efficient to parallelize both loops unless you are specifically targeting a machine with a large number of processors. However, as in all cases, you should test your application thoroughly to determine the best approach.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Security;
using System.Threading;
using System.Threading.Tasks;

class Program
{
   static void Main()
   {
      try {
         TraverseTreeParallelForEach(@"C:\Program Files", (f) =>
         {
            // Exceptions are no-ops.
            try {
               // Do nothing with the data except read it.
               byte[] data = File.ReadAllBytes(f);
            }
            catch (FileNotFoundException) {}
            catch (IOException) {}
            catch (UnauthorizedAccessException) {}
            catch (SecurityException) {}
            // Display the filename.
            Console.WriteLine(f);
         });
      }
      catch (ArgumentException) {
         Console.WriteLine(@"The directory 'C:\Program Files' does not exist.");
      }

      // Keep the console window open.
      Console.ReadKey();
   }

   public static void TraverseTreeParallelForEach(string root, Action<string> action)
   {
      //Count of files traversed and timer for diagnostic output
      int fileCount = 0;
      var sw = Stopwatch.StartNew();

      // Determine whether to parallelize file processing on each folder based on processor count.
      int procCount = System.Environment.ProcessorCount;

      // Data structure to hold names of subfolders to be examined for files.
      Stack<string> dirs = new Stack<string>();

      if (!Directory.Exists(root)) {
             throw new ArgumentException();
      }
      dirs.Push(root);

      while (dirs.Count > 0) {
         string currentDir = dirs.Pop();
         string[] subDirs = {};
         string[] files = {};

         try {
            subDirs = Directory.GetDirectories(currentDir);
         }
         // Thrown if we do not have discovery permission on the directory.
         catch (UnauthorizedAccessException e) {
            Console.WriteLine(e.Message);
            continue;
         }
         // Thrown if another process has deleted the directory after we retrieved its name.
         catch (DirectoryNotFoundException e) {
            Console.WriteLine(e.Message);
            continue;
         }

         try {
            files = Directory.GetFiles(currentDir);
         }
         catch (UnauthorizedAccessException e) {
            Console.WriteLine(e.Message);
            continue;
         }
         catch (DirectoryNotFoundException e) {
            Console.WriteLine(e.Message);
            continue;
         }
         catch (IOException e) {
            Console.WriteLine(e.Message);
            continue;
         }

         // Execute in parallel if there are enough files in the directory.
         // Otherwise, execute sequentially.Files are opened and processed
         // synchronously but this could be modified to perform async I/O.
         try {
            if (files.Length < procCount) {
               foreach (var file in files) {
                  action(file);
                  fileCount++;
               }
            }
            else {
               Parallel.ForEach(files, () => 0, (file, loopState, localCount) =>
                                            { action(file);
                                              return (int) ++localCount;
                                            },
                                 (c) => {
                                           Interlocked.Add(ref fileCount, c);
                                 });
            }
         }
         catch (AggregateException ae) {
            ae.Handle((ex) => {
                            if (ex is UnauthorizedAccessException) {
                               // Here we just output a message and go on.
                               Console.WriteLine(ex.Message);
                               return true;
                            }
                            // Handle other exceptions here if necessary...

                            return false;
            });
         }

         // Push the subdirectories onto the stack for traversal.
         // This could also be done before handing the files.
         foreach (string str in subDirs)
            dirs.Push(str);
      }

      // For diagnostic purposes.
      Console.WriteLine("Processed {0} files in {1} milliseconds", fileCount, sw.ElapsedMilliseconds);
   }
}
```

```
Imports System.Collections.Generic
Imports System.Diagnostics
Imports System.IO
Imports System.Security
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Sub Main()
      Try
         TraverseTreeParallelForEach("C:\Program Files",
                                     Sub(f)
                                        ' Exceptions are No-ops.
                                        Try
                                           ' Do nothing with the data except read it.
                                           Dim data() As Byte = File.ReadAllBytes(f)
                                        ' In the event the file has been deleted.
                                        Catch e As FileNotFoundException

                                        ' General I/O exception, especially if the file is in use.
                                        Catch e As IOException

                                        ' Lack of adequate permissions.
                                        Catch e As UnauthorizedAccessException

                                        ' Lack of adequate permissions.
                                        Catch e As SecurityException

                                        End Try
                                        ' Display the filename.
                                        Console.WriteLine(f)
                                     End Sub)
      Catch e As ArgumentException
         Console.WriteLine("The directory 'C:\Program Files' does not exist.")
      End Try
      ' Keep the console window open.
      Console.ReadKey()
   End Sub

   Public Sub TraverseTreeParallelForEach(ByVal root As String, ByVal action As Action(Of String))
      'Count of files traversed and timer for diagnostic output
      Dim fileCount As Integer = 0
      Dim sw As Stopwatch = Stopwatch.StartNew()
      ' Determine whether to parallelize file processing on each folder based on processor count.
      Dim procCount As Integer = System.Environment.ProcessorCount

      ' Data structure to hold names of subfolders to be examined for files.
      Dim dirs As New Stack(Of String)

      If Not Directory.Exists(root) Then Throw New ArgumentException()

      dirs.Push(root)

      While (dirs.Count > 0)
         Dim currentDir As String = dirs.Pop()
         Dim subDirs() As String = Nothing
         Dim files() As String = Nothing

         Try
            subDirs = Directory.GetDirectories(currentDir)
         ' Thrown if we do not have discovery permission on the directory.
         Catch e As UnauthorizedAccessException
            Console.WriteLine(e.Message)
            Continue While
         ' Thrown if another process has deleted the directory after we retrieved its name.
         Catch e As DirectoryNotFoundException
            Console.WriteLine(e.Message)
            Continue While
         End Try

         Try
            files = Directory.GetFiles(currentDir)
         Catch e As UnauthorizedAccessException
            Console.WriteLine(e.Message)
            Continue While
         Catch e As DirectoryNotFoundException
            Console.WriteLine(e.Message)
            Continue While
         Catch e As IOException
            Console.WriteLine(e.Message)
            Continue While
         End Try

         ' Execute in parallel if there are enough files in the directory.
         ' Otherwise, execute sequentially.Files are opened and processed
         ' synchronously but this could be modified to perform async I/O.
         Try
            If files.Length < procCount Then
               For Each file In files
                  action(file)
                  fileCount += 1
               Next
            Else
               Parallel.ForEach(files, Function() 0, Function(file, loopState, localCount)
                                                        action(file)
                                                        localCount = localCount + 1
                                                        Return localCount
                                                     End Function,
                                 Sub(c)
                                    Interlocked.Add(fileCount, c)
                                 End Sub)
            End If
         Catch ae As AggregateException
            ae.Handle(Function(ex)

                         If TypeOf (ex) Is UnauthorizedAccessException Then

                            ' Here we just output a message and go on.
                            Console.WriteLine(ex.Message)
                            Return True
                         End If
                         ' Handle other exceptions here if necessary...
```

```
                                Return False
                            End Function)
            End Try
            ' Push the subdirectories onto the stack for traversal.
            ' This could also be done before handing the files.
            For Each str As String In subDirs
               dirs.Push(str)
            Next

            ' For diagnostic purposes.
            Console.WriteLine("Processed {0} files in {1} milliseconds", fileCount, sw.ElapsedMilliseconds)
         End While
      End Sub
   End Module
```

In this example, the file I/O is performed synchronously. When dealing with large files or slow network connections, it might be preferable to access the files asynchronously. You can combine asynchronous I/O techniques with parallel iteration. For more information, see TPL and Traditional .NET Framework Asynchronous Programming.

The example uses the local `fileCount` variable to maintain a count of the total number of files processed. Because the variable might be accessed concurrently by multiple tasks, access to it is synchronized by calling the Interlocked.Add method.

Note that if an exception is thrown on the main thread, the threads that are started by the ForEach method might continue to run. To stop these threads, you can set a Boolean variable in your exception handlers, and check its value on each iteration of the parallel loop. If the value indicates that an exception has been thrown, use the ParallelLoopState variable to stop or break from the loop. For more information, see How to: Stop or Break from a Parallel.For Loop.

See also

- Data Parallelism

Task-based asynchronous programming

8/20/2019 • 37 minutes to read • Edit Online

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.

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

```
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<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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.Name = "Main";

      // Create a task and supply a user delegate by using a lambda expression.
      Task taskA = new Task( () => 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();
   }
}
// The example displays output like the following:
//       Hello from thread 'Main'.
//       Hello from taskA.
```

```
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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.Name = "Main";

      // Define and run the task.
      Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));

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

```
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 that you can retrieve through its Task.AsyncState property, as shown in the following example.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                               },
                                               new CustomData() {Name = i, CreationTime = DateTime.Now.Ticks}
);
      }
      Task.WaitAll(taskArray);
      foreach (var task in taskArray) {
         var data = task.AsyncState as CustomData;
         if (data != null)
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum);
      }
   }
}
// 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.
```

```
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 and Task<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<TResult>, they each have a public Task<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.

```csharp
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
        Task<Double>[] taskArray = { Task<Double>.Factory.StartNew(() => DoComputation(1.0)),
                                     Task<Double>.Factory.StartNew(() => DoComputation(100.0)),
                                     Task<Double>.Factory.StartNew(() => DoComputation(1000.0)) };

        var results = new Double[taskArray.Length];
        Double sum = 0;

        for (int i = 0; i < taskArray.Length; i++) {
            results[i] = taskArray[i].Result;
            Console.Write("{0:N1} {1}", results[i],
                              i == taskArray.Length - 1 ? "= " : "+ ");
            sum += results[i];
        }
        Console.WriteLine("{0:N1}", sum);
   }

   private static Double DoComputation(Double start)
   {
      Double sum = 0;
      for (var value = start; value <= start + 10; value += .1)
         sum += value;

      return sum;
   }
}
// The example displays the following output:
//        606.0 + 10,605.0 + 100,495.0 = 111,706.0
```

```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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      // Create the task object by using an Action(Of Object) to pass in the loop
      // counter. This produces an unexpected result.
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj) => {
                                                 var data = new CustomData() {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);
                                               },
                                              i );
      }
      Task.WaitAll(taskArray);
   }
}
// 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.
```

```
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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void 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.
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                                  Console.WriteLine("Task #{0} created at {1} on thread #
{2}.",
                                                                   data.Name, data.CreationTime,
data.ThreadNum);
                                               },
                                               new CustomData() {Name = i, CreationTime = DateTime.Now.Ticks}
);
      }
      Task.WaitAll(taskArray);
   }
}
// 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.
```

```
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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                               },
                                               new CustomData() {Name = i, CreationTime = DateTime.Now.Ticks}
);
      }
      Task.WaitAll(taskArray);
      foreach (var task in taskArray) {
         var data = task.AsyncState as CustomData;
         if (data != null)
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum);
      }
   }
}
// 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.
```

```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<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.

```
var task3 = new Task(() => MyLongRunningMethod(),
                    TaskCreationOptions.LongRunning | TaskCreationOptions.PreferFairness);
task3.Start();
```

```
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.

```
using System;
using System.Globalization;
using System.Runtime.Versioning;
using System.Threading;
using System.Threading.Tasks;

[assembly:TargetFramework(".NETFramework,Version=v4.6")]

public class Example
{

   public static void Main()
   {
       decimal[] values = { 163025412.32m, 18905365.59m };
       string formatString = "C2";
       Func<String> formatDelegate = () => { string output = String.Format("Formatting using the {0} culture
on thread {1}.\n",
                                                                            CultureInfo.CurrentCulture.Name,

Thread.CurrentThread.ManagedThreadId);
                                             foreach (var value in values)
                                                output += String.Format("{0}   ",
value.ToString(formatString));

                                             output += Environment.NewLine;
                                             return output;
                                           };

       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")
          Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
       else
          Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

       Console.WriteLine("Changed the current culture to {0}.\n",
                         CultureInfo.CurrentCulture.Name);

       // 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:");
        var t1 = Task.Run(formatDelegate);
        Console.WriteLine(t1.Result);

        Console.WriteLine("Executing a task synchronously:");
        var t2 = new Task<String>(formatDelegate);
        t2.RunSynchronously();
        Console.WriteLine(t2.Result);
    }
}
// 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 fr-FR 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 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 €
```

```
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<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<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<TResult>(Func<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<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<TResult>.ContinueWith<TNewResult>(Func<Task<TResult>,TNewResult>) method. The `displayData` task executes automatically when `processData` finishes, and the Tuple<T1,T2,T3> object returned by the `processData` lambda expression is accessible to the `displayData` task through the `processData` task's Task<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.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var getData = Task.Factory.StartNew(() => {
                                             Random rnd = new Random();
                                             int[] values = new int[100];
                                             for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
                                                values[ctr] = rnd.Next();

                                             return values;
                                          } );
      var processData = getData.ContinueWith((x) => {
                                                int n = x.Result.Length;
                                                long sum = 0;
                                                double mean;

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

                                                mean = sum / (double) n;
                                                return Tuple.Create(n, sum, mean);
                                             } );
      var displayData = processData.ContinueWith((x) => {
                                                    return String.Format("N={0:N0}, Total = {1:N0}, Mean =
{2:N2}",
                                                                         x.Result.Item1, x.Result.Item2,
                                                                         x.Result.Item3);
                                                 } );
      Console.WriteLine(displayData.Result);
   }
}
// The example displays output similar to the following:
//    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

```
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<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<TResult> object returned by the chain of method calls is the final continuation task.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var displayData = Task.Factory.StartNew(() => {
                                                 Random rnd = new Random();
                                                 int[] values = new int[100];
                                                 for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
                                                    values[ctr] = rnd.Next();

                                                 return values;
                                              } ).
                        ContinueWith((x) => {
                                        int n = x.Result.Length;
                                        long sum = 0;
                                        double mean;

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

                                        mean = sum / (double) n;
                                        return Tuple.Create(n, sum, mean);
                                     } ).
                        ContinueWith((x) => {
                                        return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}",
                                                             x.Result.Item1, x.Result.Item2,
                                                             x.Result.Item3);
                                     } );
      Console.WriteLine(displayData.Result);
   }
}
// The example displays output similar to the following:
//    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

```
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.

```
var outer = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Outer task beginning.");

    var child = Task.Factory.StartNew(() =>
    {
        Thread.SpinWait(5000000);
        Console.WriteLine("Detached task completed.");
    });

});

outer.Wait();
Console.WriteLine("Outer task completed.");
// The example displays the following output:
//    Outer task beginning.
//    Outer task completed.
//    Detached task completed.
```

```
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.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
                      Console.WriteLine("Parent task beginning.");
                      for (int ctr = 0; ctr < 10; ctr++) {
                         int taskNo = ctr;
                         Task.Factory.StartNew((x) => {
                                                 Thread.SpinWait(5000000);
                                                 Console.WriteLine("Attached child #{0} completed.",
                                                                   x);
                                              },
                                              taskNo, TaskCreationOptions.AttachedToParent);
                      }
                   });

      parent.Wait();
      Console.WriteLine("Parent task completed.");
   }
}
// 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.
```

```
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<TResult> types provide several overloads of the Task.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.

```
Task[] tasks = new Task[3]
{
    Task.Factory.StartNew(() => MethodA()),
    Task.Factory.StartNew(() => MethodB()),
    Task.Factory.StartNew(() => MethodC())
};

//Block until all tasks complete.
Task.WaitAll(tasks);

// Continue on this thread...
```

```
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.

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<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 methods.

Task.WhenAll

The Task.WhenAll method asynchronously waits for multiple Task or Task<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<TResult> objects to complete from one method call.

Task.WhenAny

The Task.WhenAny method asynchronously waits for one of multiple Task or Task<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 method, you can create a Task<TResult> object that holds a pre-computed result. This method is useful when you perform an asynchronous operation that returns a Task<TResult> object, and the result of that Task<TResult> object is already computed. For an example that uses FromResult 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.

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 or ContinueWhenAny method or their equivalents in the Task<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<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<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<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<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<TResult>. Instead, we recommend that you use the AsyncState property to associate additional data or state with a Task or Task<TResult> object. You can also use extension methods to extend the functionality of the Task and Task<TResult> classes. For more information about extension methods, see Extension Methods and Extension Methods.

If you must inherit from Task or Task<TResult>, you cannot use Run, Run, or the System.Threading.Tasks.TaskFactory, System.Threading.Tasks.TaskFactory<TResult>, or

System.Threading.Tasks.TaskCompletionSource<TResult> classes to create instances of your custom task type because these mechanisms create only Task and Task<TResult> objects. In addition, you cannot use the task continuation mechanisms that are provided by Task, Task<TResult>, TaskFactory, and TaskFactory<TResult> to create instances of your custom task type because these mechanisms also create only Task and Task<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	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 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	Describes how to use For and ForEach to create parallel loops over data.
Parallel Programming	Top level node for .NET Framework parallel programming.

See also

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

Chaining Tasks by Using Continuation Tasks

4/28/2019 • 28 minutes to read • Edit Online

In asynchronous programming, it is common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. Traditionally, continuations have been done by using callback methods. In the Task Parallel Library, the same functionality is provided by *continuation tasks*. A continuation task (also known just as a continuation) is an asynchronous task that is invoked by another task, which is known as the *antecedent*, when the antecedent finishes.

Continuations are relatively easy to use, but are nevertheless powerful and flexible. For example, you can:

- Pass data from the antecedent to the continuation.
- Specify the precise conditions under which the continuation will be invoked or not invoked.
- Cancel a continuation either before it starts or cooperatively as it is running.
- Provide hints about how the continuation should be scheduled.
- Invoke multiple continuations from the same antecedent.
- Invoke one continuation when all or any one of multiple antecedents complete.
- Chain continuations one after another to any arbitrary length.
- Use a continuation to handle exceptions thrown by the antecedent.

About continuations

A continuation is a task that is created in the WaitingForActivation state. It is activated automatically when its antecedent task or tasks complete. Calling Task.Start on a continuation in user code throws an System.InvalidOperationException exception.

A continuation is itself a Task and does not block the thread on which it is started. Call the Task.Wait method to block until the continuation task finishes.

Creating a continuation for a single antecedent

You create a continuation that executes when its antecedent has completed by calling the Task.ContinueWith method. The following example shows the basic pattern (for clarity, exception handling is omitted). It executes an antecedent task, `taskA`, that returns a DayOfWeek object that indicates the name of the current day of the week. When the antecedent completes, the continuation task, `continuation`, is passed the antecedent and displays a string that includes its result.

NOTE

The C# samples in this article make use of the `async` modifier on the `Main` method. That feature is available in C# 7.1 and later. Previous versions generate `CS5001` when compiling this sample code. You'll need to set the language version to C# 7.1 or newer. You can learn how to configure the language version in the article on configure language version.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static async Task Main()
   {
      // Execute the antecedent.
      Task<DayOfWeek> taskA = Task.Run( () => DateTime.Today.DayOfWeek );

      // Execute the continuation when the antecedent finishes.
      await taskA.ContinueWith( antecedent => Console.WriteLine("Today is {0}.", antecedent.Result) );
   }
}
// The example displays output like the following output:
//       Today is Monday.
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      ' Execute the antecedent.
      Dim taskA As Task(Of DayOfWeek) = Task.Run(Function() DateTime.Today.DayOfWeek )

      ' Execute the continuation when the antecedent finishes.
      Dim continuation As Task = taskA.ContinueWith(Sub(antecedent)
                                                       Console.WriteLine("Today is {0}.", antecedent.Result)
                                                    End Sub)
      continuation.Wait()
   End Sub
End Module
' The example displays output like the following output:
'       Today is Monday.
```

Creating a continuation for multiple antecedents

You can also create a continuation that will run when any or all of a group of tasks has completed. To execute a continuation when all antecedent tasks have completed, you call the static (`Shared` in Visual Basic) Task.WhenAll method or the instance TaskFactory.ContinueWhenAll method. To execute a continuation when any of the antecedent tasks has completed, you call the static (`Shared` in Visual Basic) Task.WhenAny method or the instance TaskFactory.ContinueWhenAny method.

Note that calls to the Task.WhenAll and Task.WhenAny overloads do not block the calling thread. However, you typically call all but the Task.WhenAll(IEnumerable<Task>) and Task.WhenAll(Task[]) methods to retrieve the returned Task<TResult>.Result property, which does block the calling thread.

The following example calls the Task.WhenAll(IEnumerable<Task>) method to create a continuation task that reflects the results of its 10 antecedent tasks. Each antecedent task squares an index value that ranges from one to 10. If the antecedents complete successfully (their Task.Status property is TaskStatus.RanToCompletion), the Task<TResult>.Result property of the continuation is an array of the Task<TResult>.Result values returned by each antecedent. The example adds them to compute the sum of squares for all numbers between one and 10.

```
using System.Collections.Generic;
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      List<Task<int>> tasks = new List<Task<int>>();
      for (int ctr = 1; ctr <= 10; ctr++) {
         int baseValue = ctr;
         tasks.Add(Task.Factory.StartNew( (b) => { int i = (int) b;
                                                    return i * i; }, baseValue));
      }
      var continuation = Task.WhenAll(tasks);

      long sum = 0;
      for (int ctr = 0; ctr <= continuation.Result.Length - 1; ctr++) {
         Console.Write("{0} {1} ", continuation.Result[ctr],
                       ctr == continuation.Result.Length - 1 ? "=" : "+");
         sum += continuation.Result[ctr];
      }
      Console.WriteLine(sum);
   }
}
// The example displays the following output:
//    1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

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

Module Example
   Public Sub Main()
      Dim tasks As New List(Of Task(Of Integer))()
      For ctr As Integer = 1 To 10
         Dim baseValue As Integer = ctr
         tasks.Add(Task.Factory.StartNew( Function(b)
                                             Dim i As Integer = CInt(b)
                                             Return i * i
                                          End Function, baseValue))
      Next
      Dim continuation = Task.WhenAll(tasks)

      Dim sum As Long = 0
      For ctr As Integer = 0 To continuation.Result.Length - 1
         Console.Write("{0} {1} ", continuation.Result(ctr),
                       If (ctr = continuation.Result.Length - 1, "=", "+"))
         sum += continuation.Result(ctr)
      Next
      Console.WriteLine(sum)
   End Sub
End Module
' The example displays the following output:
'       1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

Continuation options

When you create a single-task continuation, you can use a ContinueWith overload that takes a System.Threading.Tasks.TaskContinuationOptions enumeration value to specify the conditions under which the continuation starts. For example, you can specify that the continuation is to run only if the antecedent completes successfully, or only if it completes in a faulted state. If the condition is not true when the antecedent is ready to invoke the continuation, the continuation transitions directly to the TaskStatus.Canceled state and subsequently cannot be started.

A number of multi-task continuation methods, such as overloads of the TaskFactory.ContinueWhenAll method, also include a System.Threading.Tasks.TaskContinuationOptions parameter. Only a subset of all System.Threading.Tasks.TaskContinuationOptions enumeration members are valid, however. You can specify System.Threading.Tasks.TaskContinuationOptions values that have counterparts in the System.Threading.Tasks.TaskCreationOptions enumeration, such as TaskContinuationOptions.AttachedToParent, TaskContinuationOptions.LongRunning, and TaskContinuationOptions.PreferFairness. If you specify any of the `NotOn` or `OnlyOn` options with a multi-task continuation, an ArgumentOutOfRangeException exception will be thrown at run time.

For more information on task continuation options, see the TaskContinuationOptions topic.

Passing Data to a Continuation

The Task.ContinueWith method passes a reference to the antecedent to the user delegate of the continuation as an argument. If the antecedent is a System.Threading.Tasks.Task<TResult> object, and the task ran until it was completed, then the continuation can access the Task<TResult>.Result property of the task.

The Task<TResult>.Result property blocks until the task has completed. However, if the task was canceled or faulted, attempting to access the Result property throws an AggregateException exception. You can avoid this problem by using the OnlyOnRanToCompletion option, as shown in the following example.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static async Task Main()
   {
      var t = Task.Run( () => { DateTime dat = DateTime.Now;
                                if (dat == DateTime.MinValue)
                                   throw new ArgumentException("The clock is not working.");

                                if (dat.Hour > 17)
                                   return "evening";
                                else if (dat.Hour > 12)
                                   return "afternoon";
                                else
                                   return "morning"; });
      await t.ContinueWith( (antecedent) => { Console.WriteLine("Good {0}!",
                                                                antecedent.Result);
                                              Console.WriteLine("And how are you this fine {0}?",
                                                                antecedent.Result); },
                              TaskContinuationOptions.OnlyOnRanToCompletion);
   }
}
// The example displays output like the following:
//       Good afternoon!
//       And how are you this fine afternoon?
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim t = Task.Run( Function()
                           Dim dat As DateTime = DateTime.Now
                           If dat = DateTime.MinValue Then
                              Throw New ArgumentException("The clock is not working.")
                           End If

                           If dat.Hour > 17 Then
                              Return "evening"
                           Else If dat.Hour > 12 Then
                              Return "afternoon"
                           Else
                              Return "morning"
                           End If
                        End Function)
      Dim c = t.ContinueWith( Sub(antecedent)
                                 Console.WriteLine("Good {0}!",
                                                   antecedent.Result)
                                 Console.WriteLine("And how are you this fine {0}?",
                                                   antecedent.Result)
                              End Sub, TaskContinuationOptions.OnlyOnRanToCompletion)
      c.Wait()
   End Sub
End Module
' The example displays output like the following:
'       Good afternoon!
'       And how are you this fine afternoon?
```

If you want the continuation to run even if the antecedent did not run to successful completion, you must guard against the exception. One approach is to test the Task.Status property of the antecedent, and only attempt to access the Result property if the status is not Faulted or Canceled. You can also examine the Exception property of the antecedent. For more information, see Exception Handling. The following example modifies the previous example to access antecedent's Task<TResult>.Result property only if its status is TaskStatus.RanToCompletion.

```csharp
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var t = Task.Run( () => { DateTime dat = DateTime.Now;
                                if (dat == DateTime.MinValue)
                                   throw new ArgumentException("The clock is not working.");

                                if (dat.Hour > 17)
                                   return "evening";
                                else if (dat.Hour > 12)
                                   return "afternoon";
                                else
                                   return "morning"; });
      var c = t.ContinueWith( (antecedent) => { if (t.Status == TaskStatus.RanToCompletion) {
                                                   Console.WriteLine("Good {0}!",
                                                                     antecedent.Result);
                                                   Console.WriteLine("And how are you this fine {0}?",
                                                                  antecedent.Result);
                                                }
                                                else if (t.Status == TaskStatus.Faulted) {
                                                   Console.WriteLine(t.Exception.GetBaseException().Message);
                                                }} );
   }
}
// The example displays output like the following:
//       Good afternoon!
//       And how are you this fine afternoon?
```

```vb
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim t = Task.Run( Function()
                           Dim dat As DateTime = DateTime.Now
                           If dat = DateTime.MinValue Then
                              Throw New ArgumentException("The clock is not working.")
                           End If

                           If dat.Hour > 17 Then
                              Return "evening"
                           Else If dat.Hour > 12 Then
                              Return "afternoon"
                           Else
                              Return "morning"
                           End If
                        End Function)
      Dim c = t.ContinueWith( Sub(antecedent)
                                 If t.Status = TaskStatus.RanToCompletion Then
                                    Console.WriteLine("Good {0}!",
                                                      antecedent.Result)
                                    Console.WriteLine("And how are you this fine {0}?",
                                                      antecedent.Result)
                                 Else If t.Status = TaskStatus.Faulted Then
                                    Console.WriteLine(t.Exception.GetBaseException().Message)
                                 End If
                              End Sub)
   End Sub
End Module
' The example displays output like the following:
'       Good afternoon!
'       And how are you this fine afternoon?
```

Canceling a Continuation

The Task.Status property of a continuation is set to TaskStatus.Canceled in the following situations:

- It throws an OperationCanceledException exception in response to a cancellation request. As with any task, if the exception contains the same token that was passed to the continuation, it is treated as an acknowledgement of cooperative cancellation.
- The continuation is passed a System.Threading.CancellationToken whose IsCancellationRequested property is `true`. In this case, the continuation does not start, and it transitions to the TaskStatus.Canceled state.
- The continuation never runs because the condition set by its TaskContinuationOptions argument was not met. For example, if an antecedent goes into a TaskStatus.Faulted state, its continuation that was passed the TaskContinuationOptions.NotOnFaulted option will not run but will transition to the Canceled state.

If a task and its continuation represent two parts of the same logical operation, you can pass the same cancellation token to both tasks, as shown in the following example. It consists of an antecedent that generates a list of integers that are divisible by 33, which it passes to the continuation. The continuation in turn displays the list. Both the antecedent and the continuation pause regularly for random intervals. In addition, a System.Threading.Timer object is used to execute the `Elapsed` method after a five-second timeout interval. This example calls the CancellationTokenSource.Cancel method, which causes the currently executing task to call the CancellationToken.ThrowIfCancellationRequested method. Whether the CancellationTokenSource.Cancel method is called when the antecedent or its continuation is executing depends on the duration of the randomly generated pauses. If the antecedent is canceled, the continuation will not start. If the antecedent is not canceled, the token can still be used to cancel the continuation.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Random rnd = new Random();
      var cts = new CancellationTokenSource();
      CancellationToken token = cts.Token;
      Timer timer = new Timer(Elapsed, cts, 5000, Timeout.Infinite);

      var t = Task.Run( () => { List<int> product33 = new List<int>();
                                for (int ctr = 1; ctr < Int16.MaxValue; ctr++) {
                                   if (token.IsCancellationRequested) {
                                      Console.WriteLine("\nCancellation requested in antecedent...\n");
                                      token.ThrowIfCancellationRequested();
                                   }
                                   if (ctr % 2000 == 0) {
                                      int delay = rnd.Next(16,501);
                                      Thread.Sleep(delay);
                                   }

                                   if (ctr % 33 == 0)
                                      product33.Add(ctr);
                                }
                                return product33.ToArray();
                              }, token);

      Task continuation = t.ContinueWith(antecedent => { Console.WriteLine("Multiples of 33:\n");
                                                         var arr = antecedent.Result;
                                                         for (int ctr = 0; ctr < arr.Length; ctr++)
                                                         {
```

```
                                                                  if (token.IsCancellationRequested) {
                                                                     Console.WriteLine("\nCancellation requested in
continuation...\n");
                                                                     token.ThrowIfCancellationRequested();
                                                                  }

                                                                  if (ctr % 100 == 0) {
                                                                     int delay = rnd.Next(16,251);
                                                                     Thread.Sleep(delay);
                                                                  }
                                                                  Console.Write("{0:N0}{1}", arr[ctr],
                                                                                ctr != arr.Length - 1 ? ", " : "");
                                                                  if (Console.CursorLeft >= 74)
                                                                     Console.WriteLine();
                                                               }
                                                               Console.WriteLine();
                                                             } , token);

      try {
          continuation.Wait();
      }
      catch (AggregateException e) {
         foreach (Exception ie in e.InnerExceptions)
            Console.WriteLine("{0}: {1}", ie.GetType().Name,
                              ie.Message);
      }
      finally {
         cts.Dispose();
      }

      Console.WriteLine("\nAntecedent Status: {0}", t.Status);
      Console.WriteLine("Continuation Status: {0}", continuation.Status);
  }

   private static void Elapsed(object state)
   {
      CancellationTokenSource cts = state as CancellationTokenSource;
      if (cts == null) return;

      cts.Cancel();
      Console.WriteLine("\nCancellation request issued...\n");
   }
}
// The example displays the following output:
//    Multiples of 33:
//
//    33, 66, 99, 132, 165, 198, 231, 264, 297, 330, 363, 396, 429, 462, 495, 528,
//    561, 594, 627, 660, 693, 726, 759, 792, 825, 858, 891, 924, 957, 990, 1,023,
//    1,056, 1,089, 1,122, 1,155, 1,188, 1,221, 1,254, 1,287, 1,320, 1,353, 1,386,
//    1,419, 1,452, 1,485, 1,518, 1,551, 1,584, 1,617, 1,650, 1,683, 1,716, 1,749,
//    1,782, 1,815, 1,848, 1,881, 1,914, 1,947, 1,980, 2,013, 2,046, 2,079, 2,112,
//    2,145, 2,178, 2,211, 2,244, 2,277, 2,310, 2,343, 2,376, 2,409, 2,442, 2,475,
//    2,508, 2,541, 2,574, 2,607, 2,640, 2,673, 2,706, 2,739, 2,772, 2,805, 2,838,
//    2,871, 2,904, 2,937, 2,970, 3,003, 3,036, 3,069, 3,102, 3,135, 3,168, 3,201,
//    3,234, 3,267, 3,300, 3,333, 3,366, 3,399, 3,432, 3,465, 3,498, 3,531, 3,564,
//    3,597, 3,630, 3,663, 3,696, 3,729, 3,762, 3,795, 3,828, 3,861, 3,894, 3,927,
//    3,960, 3,993, 4,026, 4,059, 4,092, 4,125, 4,158, 4,191, 4,224, 4,257, 4,290,
//    4,323, 4,356, 4,389, 4,422, 4,455, 4,488, 4,521, 4,554, 4,587, 4,620, 4,653,
//    4,686, 4,719, 4,752, 4,785, 4,818, 4,851, 4,884, 4,917, 4,950, 4,983, 5,016,
//    5,049, 5,082, 5,115, 5,148, 5,181, 5,214, 5,247, 5,280, 5,313, 5,346, 5,379,
//    5,412, 5,445, 5,478, 5,511, 5,544, 5,577, 5,610, 5,643, 5,676, 5,709, 5,742,
//    5,775, 5,808, 5,841, 5,874, 5,907, 5,940, 5,973, 6,006, 6,039, 6,072, 6,105,
//    6,138, 6,171, 6,204, 6,237, 6,270, 6,303, 6,336, 6,369, 6,402, 6,435, 6,468,
//    6,501, 6,534, 6,567, 6,600, 6,633, 6,666, 6,699, 6,732, 6,765, 6,798, 6,831,
//    6,864, 6,897, 6,930, 6,963, 6,996, 7,029, 7,062, 7,095, 7,128, 7,161, 7,194,
//    7,227, 7,260, 7,293, 7,326, 7,359, 7,392, 7,425, 7,458, 7,491, 7,524, 7,557,
//    7,590, 7,623, 7,656, 7,689, 7,722, 7,755, 7,788, 7,821, 7,854, 7,887, 7,920,
//    7,953, 7,986, 8,019, 8,052, 8,085, 8,118, 8,151, 8,184, 8,217, 8,250, 8,283,
//    8,316, 8,349, 8,382, 8,415, 8,448, 8,481, 8,514, 8,547, 8,580, 8,613, 8,646,
//    8,679, 8,712, 8,745, 8,778, 8,811, 8,844, 8,877, 8,910, 8,943, 8,976, 9,009,
//    9,042, 9,075, 9,108, 9,141, 9,174, 9,207, 9,240, 9,273, 9,306, 9,339, 9,372,
//    9,405, 9,438, 9,471, 9,504, 9,537, 9,570, 9,603, 9,636, 9,669, 9,702, 9,735,
//    9,768, 9,801, 9,834, 9,867, 9,900, 9,933, 9,966, 9,999, 10,032, 10,065, 10,098,
//    10,131, 10,164, 10,197, 10,230, 10,263, 10,296, 10,329, 10,362, 10,395, 10,428,
//    10,461, 10,494, 10,527, 10,560, 10,593, 10,626, 10,659, 10,692, 10,725, 10,758,
//    10,791, 10,824, 10,857, 10,890, 10,923, 10,956, 10,989, 11,022, 11,055, 11,088,
//    11,121, 11,154, 11,187, 11,220, 11,253, 11,286, 11,319, 11,352, 11,385, 11,418,
//    11,451, 11,484, 11,517, 11,550, 11,583, 11,616, 11,649, 11,682, 11,715, 11,748,
//    11,781, 11,814, 11,847, 11,880, 11,913, 11,946, 11,979, 12,012, 12,045, 12,078,
//    12,111, 12,144, 12,177, 12,210, 12,243, 12,276, 12,309, 12,342, 12,375, 12,408,
//    12,441, 12,474, 12,507, 12,540, 12,573, 12,606, 12,639, 12,672, 12,705, 12,738,
//    12,771, 12,804, 12,837, 12,870, 12,903, 12,936, 12,969, 13,002, 13,035, 13,068,
//    13,101, 13,134, 13,167, 13,200, 13,233, 13,266,
//    Cancellation requested in continuation...
//
//
//    Cancellation request issued...
//
//    TaskCanceledException: A task was canceled.
//
//    Antecedent Status: RanToCompletion
//    Continuation Status: Canceled
```

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

Module Example
   Public Sub Main()
      Dim rnd As New Random()
      Dim lockObj As New Object()
      Dim cts As New CancellationTokenSource()
      Dim token As CancellationToken = cts.Token
      Dim timer As New Timer(AddressOf Elapsed, cts, 5000, Timeout.Infinite)

      Dim t = Task.Run( Function()
                           Dim product33 As New List(Of Integer)()
                           For ctr As Integer = 1 To Int16.MaxValue
                              ' Check for cancellation.
                              If token.IsCancellationRequested Then
                                 Console.WriteLine("\nCancellation requested in antecedent...\n")
                                 token.ThrowIfCancellationRequested()
                              End If
                              ' Introduce a delay.
                              If ctr Mod 2000 = 0 Then
                                 Dim delay As Integer
                                 SyncLock lockObj
                                    delay = rnd.Next(16,501)
                                 End SyncLock
                                 Thread.Sleep(delay)
                              End If

                              ' Determine if this is a multiple of 33.
                              If ctr Mod 33 = 0 Then product33.Add(ctr)
                           Next
                           Return product33.ToArray()
                        End Function, token)

      Dim continuation = t.ContinueWith(Sub(antecedent)
                                           Console.WriteLine("Multiples of 33:" + vbCrLf)
                                           Dim arr = antecedent.Result
                                           For ctr As Integer = 0 To arr.Length - 1
                                              If token.IsCancellationRequested Then
                                                 Console.WriteLine("{0}Cancellation requested in
continuation...{0}",
                                                                   vbCrLf)
```

```
                                                   token.ThrowIfCancellationRequested()
                                                End If

                                                If ctr Mod 100 = 0 Then
                                                   Dim delay As Integer
                                                   SyncLock lockObj
                                                      delay = rnd.Next(16,251)
                                                   End SyncLock
                                                   Thread.Sleep(delay)
                                                End If
                                                Console.Write("{0:N0}{1}", arr(ctr),
                                                              If(ctr <> arr.Length - 1, ", ", ""))
                                                If Console.CursorLeft >= 74 Then Console.WriteLine()
                                             Next
                                             Console.WriteLine()
                                          End Sub, token)

         Try
            continuation.Wait()
         Catch e As AggregateException
            For Each ie In e.InnerExceptions
               Console.WriteLine("{0}: {1}", ie.GetType().Name,
                                 ie.Message)
            Next
         Finally
            cts.Dispose()
         End Try

         Console.WriteLine(vbCrLf + "Antecedent Status: {0}", t.Status)
         Console.WriteLine("Continuation Status: {0}", continuation.Status)
     End Sub

      Private Sub Elapsed(state As Object)
         Dim cts As CancellationTokenSource = TryCast(state, CancellationTokenSource)
         If cts Is Nothing Then return

         cts.Cancel()
         Console.WriteLine("{0}Cancellation request issued...{0}", vbCrLf)
      End Sub
   End Module
   ' The example displays output like the following:
   '    Multiples of 33:
   '
   '    33, 66, 99, 132, 165, 198, 231, 264, 297, 330, 363, 396, 429, 462, 495, 528,
   '    561, 594, 627, 660, 693, 726, 759, 792, 825, 858, 891, 924, 957, 990, 1,023,
   '    1,056, 1,089, 1,122, 1,155, 1,188, 1,221, 1,254, 1,287, 1,320, 1,353, 1,386,
   '    1,419, 1,452, 1,485, 1,518, 1,551, 1,584, 1,617, 1,650, 1,683, 1,716, 1,749,
   '    1,782, 1,815, 1,848, 1,881, 1,914, 1,947, 1,980, 2,013, 2,046, 2,079, 2,112,
   '    2,145, 2,178, 2,211, 2,244, 2,277, 2,310, 2,343, 2,376, 2,409, 2,442, 2,475,
   '    2,508, 2,541, 2,574, 2,607, 2,640, 2,673, 2,706, 2,739, 2,772, 2,805, 2,838,
   '    2,871, 2,904, 2,937, 2,970, 3,003, 3,036, 3,069, 3,102, 3,135, 3,168, 3,201,
   '    3,234, 3,267, 3,300, 3,333, 3,366, 3,399, 3,432, 3,465, 3,498, 3,531, 3,564,
   '    3,597, 3,630, 3,663, 3,696, 3,729, 3,762, 3,795, 3,828, 3,861, 3,894, 3,927,
   '    3,960, 3,993, 4,026, 4,059, 4,092, 4,125, 4,158, 4,191, 4,224, 4,257, 4,290,
   '    4,323, 4,356, 4,389, 4,422, 4,455, 4,488, 4,521, 4,554, 4,587, 4,620, 4,653,
   '    4,686, 4,719, 4,752, 4,785, 4,818, 4,851, 4,884, 4,917, 4,950, 4,983, 5,016,
   '    5,049, 5,082, 5,115, 5,148, 5,181, 5,214, 5,247, 5,280, 5,313, 5,346, 5,379,
   '    5,412, 5,445, 5,478, 5,511, 5,544, 5,577, 5,610, 5,643, 5,676, 5,709, 5,742,
   '    5,775, 5,808, 5,841, 5,874, 5,907, 5,940, 5,973, 6,006, 6,039, 6,072, 6,105,
   '    6,138, 6,171, 6,204, 6,237, 6,270, 6,303, 6,336, 6,369, 6,402, 6,435, 6,468,
   '    6,501, 6,534, 6,567, 6,600, 6,633, 6,666, 6,699, 6,732, 6,765, 6,798, 6,831,
   '    6,864, 6,897, 6,930, 6,963, 6,996, 7,029, 7,062, 7,095, 7,128, 7,161, 7,194,
   '    7,227, 7,260, 7,293, 7,326, 7,359, 7,392, 7,425, 7,458, 7,491, 7,524, 7,557,
   '    7,590, 7,623, 7,656, 7,689, 7,722, 7,755, 7,788, 7,821, 7,854, 7,887, 7,920,
   '    7,953, 7,986, 8,019, 8,052, 8,085, 8,118, 8,151, 8,184, 8,217, 8,250, 8,283,
   '    8,316, 8,349, 8,382, 8,415, 8,448, 8,481, 8,514, 8,547, 8,580, 8,613, 8,646,
   '    8,679, 8,712, 8,745, 8,778, 8,811, 8,844, 8,877, 8,910, 8,943, 8,976, 9,009,
   '    9,042, 9,075, 9,108, 9,141, 9,174, 9,207, 9,240, 9,273, 9,306, 9,339, 9,372,
   '    9,405, 9,438, 9,471, 9,504, 9,537, 9,570, 9,603, 9,636, 9,669, 9,702, 9,735,
   '    9,768, 9,801, 9,834, 9,867, 9,900, 9,933, 9,966, 9,999, 10,032, 10,065, 10,098,
   '    10,131, 10,164, 10,197, 10,230, 10,263, 10,296, 10,329, 10,362, 10,395, 10,428,
   '    10,461, 10,494, 10,527, 10,560, 10,593, 10,626, 10,659, 10,692, 10,725, 10,758,
   '    10,791, 10,824, 10,857, 10,890, 10,923, 10,956, 10,989, 11,022, 11,055, 11,088,
   '    11,121, 11,154, 11,187, 11,220, 11,253, 11,286, 11,319, 11,352, 11,385, 11,418,
   '    11,451, 11,484, 11,517, 11,550, 11,583, 11,616, 11,649, 11,682, 11,715, 11,748,
   '    11,781, 11,814, 11,847, 11,880, 11,913, 11,946, 11,979, 12,012, 12,045, 12,078,
   '    12,111, 12,144, 12,177, 12,210, 12,243, 12,276, 12,309, 12,342, 12,375, 12,408,
   '    12,441, 12,474, 12,507, 12,540, 12,573, 12,606, 12,639, 12,672, 12,705, 12,738,
   '    12,771, 12,804, 12,837, 12,870, 12,903, 12,936, 12,969, 13,002, 13,035, 13,068,
   '    13,101, 13,134, 13,167, 13,200, 13,233, 13,266,
   '    Cancellation requested in continuation...
   '
   '
   '    Cancellation request issued...
   '
   '    TaskCanceledException: A task was canceled.
   '
   '    Antecedent Status: RanToCompletion
   '    Continuation Status: Canceled
```

You can also prevent a continuation from executing if its antecedent is canceled without supplying the continuation a cancellation token by specifying the TaskContinuationOptions.NotOnCanceled option when you create the continuation. The following is a simple example.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var cts = new CancellationTokenSource();
      CancellationToken token = cts.Token;
      cts.Cancel();

      var t = Task.FromCanceled(token);
      var continuation = t.ContinueWith( (antecedent) => {
                                          Console.WriteLine("The continuation is running.");
                                        } , TaskContinuationOptions.NotOnCanceled);
      try {
         t.Wait();
      }
      catch (AggregateException ae) {
         foreach (var ie in ae.InnerExceptions)
            Console.WriteLine("{0}: {1}", ie.GetType().Name, ie.Message);

         Console.WriteLine();
      }
      finally {
         cts.Dispose();
      }

      Console.WriteLine("Task {0}: {1:G}", t.Id, t.Status);
      Console.WriteLine("Task {0}: {1:G}", continuation.Id,
                        continuation.Status);
   }
}
// The example displays the following output:
//       TaskCanceledException: A task was canceled.
//
//       Task 1: Canceled
//       Task 2: Canceled
```

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

Module Example
   Public Sub Main()
      Dim cts As New CancellationTokenSource()
      Dim token As CancellationToken = cts.Token
      cts.Cancel()

      Dim t As  Task = Task.FromCanceled(token)
      Dim continuation As Task = t.ContinueWith(Sub(antecedent)
                                                   Console.WriteLine("The continuation is running.")
                                                End Sub, TaskContinuationOptions.NotOnCanceled)
      Try
         t.Wait()
      Catch e As AggregateException
         For Each ie In e.InnerExceptions
            Console.WriteLine("{0}: {1}", ie.GetType().Name, ie.Message)
         Next
         Console.WriteLine()
      Finally
         cts.Dispose()
      End Try

      Console.WriteLine("Task {0}: {1:G}", t.Id, t.Status)
      Console.WriteLine("Task {0}: {1:G}", continuation.Id,
                        continuation.Status)
   End Sub
End Module
' The example displays the following output:
'       TaskCanceledException: A task was canceled.
'
'       Task 1: Canceled
'       Task 2: Canceled
```

After a continuation goes into the Canceled state, it may affect continuations that follow, depending on the TaskContinuationOptions that were specified for those continuations.

Continuations that are disposed will not start.

Continuations and Child Tasks

A continuation does not run until the antecedent and all of its attached child tasks have completed. The continuation does not wait for detached child tasks to finish. The following two examples illustrate child tasks that are attached to and detached from an antecedent that creates a continuation. In the following example, the continuation runs only after all child tasks have completed, and running the example multiple times produces identical output each time. The example launches the antecedent by calling the TaskFactory.StartNew method, since by default the Task.Run method creates a parent task whose default task creation option is TaskCreationOptions.DenyChildAttach.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var t = Task.Factory.StartNew( () => { Console.WriteLine("Running antecedent task {0}...",
                                                  Task.CurrentId);
                                             Console.WriteLine("Launching attached child tasks...");
                                             for (int ctr = 1; ctr <= 5; ctr++)  {
                                                int index = ctr;
                                                Task.Factory.StartNew( (value) => {
                                                                       Console.WriteLine("   Attached child
task #{0} running",
                                                                                         value);
                                                                       Thread.Sleep(1000);
                                                                     }, index,
TaskCreationOptions.AttachedToParent);
                                             }
                                             Console.WriteLine("Finished launching attached child tasks...");
                                           });
      var continuation = t.ContinueWith( (antecedent) => { Console.WriteLine("Executing continuation of Task
{0}",
                                                                             antecedent.Id);
                                                         });
      continuation.Wait();
   }
}
// The example displays the following output:
//       Running antecedent task 1...
//       Launching attached child tasks...
//       Finished launching attached child tasks...
//          Attached child task #5 running
//          Attached child task #1 running
//          Attached child task #2 running
//          Attached child task #3 running
//          Attached child task #4 running
//       Executing continuation of Task 1
```

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

Public Module Example
   Public Sub Main()
      Dim t = Task.Factory.StartNew( Sub()
                                        Console.WriteLine("Running antecedent task {0}...",
                                                          Task.CurrentId)
                                        Console.WriteLine("Launching attached child tasks...")
                                        For ctr As Integer = 1 To 5
                                           Dim index As Integer = ctr
                                           Task.Factory.StartNew( Sub(value)
                                                                     Console.WriteLine("   Attached child task
#{0} running",
                                                                                       value)
                                                                     Thread.Sleep(1000)
                                                                  End Sub, index,
TaskCreationOptions.AttachedToParent)
                                        Next
                                        Console.WriteLine("Finished launching attached child tasks...")
                                     End Sub)
      Dim continuation = t.ContinueWith( Sub(antecedent)
                                            Console.WriteLine("Executing continuation of Task {0}",
                                                              antecedent.Id)
                                         End Sub)
      continuation.Wait()
   End Sub
End Module
' The example displays the following output:
'       Running antecedent task 1...
'       Launching attached child tasks...
'       Finished launching attached child tasks...
'          Attached child task #5 running
'          Attached child task #1 running
'          Attached child task #2 running
'          Attached child task #3 running
'          Attached child task #4 running
'       Executing continuation of Task 1
```

If child tasks are detached from the antecedent, however, the continuation runs as soon as the antecedent has terminated, regardless of the state of the child tasks. As a result, multiple runs of the following example can produce variable output that depends on how the task scheduler handled each child task.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var t = Task.Factory.StartNew( () => { Console.WriteLine("Running antecedent task {0}...",
                                                  Task.CurrentId);
                                             Console.WriteLine("Launching attached child tasks...");
                                             for (int ctr = 1; ctr <= 5; ctr++)  {
                                                int index = ctr;
                                                Task.Factory.StartNew( (value) => {
                                                                       Console.WriteLine("   Attached child
task #{0} running",
                                                                                         value);
                                                                       Thread.Sleep(1000);
                                                                     }, index);
                                             }
                                             Console.WriteLine("Finished launching detached child tasks...");
                                           }, TaskCreationOptions.DenyChildAttach);
      var continuation = t.ContinueWith( (antecedent) => { Console.WriteLine("Executing continuation of Task
{0}",
                                                                             antecedent.Id);
                                                         });
      continuation.Wait();
   }
}
// The example displays output like the following:
//       Running antecedent task 1...
//       Launching attached child tasks...
//       Finished launching detached child tasks...
//          Attached child task #1 running
//          Attached child task #2 running
//          Attached child task #5 running
//          Attached child task #3 running
//       Executing continuation of Task 1
//          Attached child task #4 running
```

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

Public Module Example
   Public Sub Main()
      Dim t = Task.Factory.StartNew( Sub()
                                        Console.WriteLine("Running antecedent task {0}...",
                                                          Task.CurrentId)
                                        Console.WriteLine("Launching attached child tasks...")
                                        For ctr As Integer = 1 To 5
                                           Dim index As Integer = ctr
                                           Task.Factory.StartNew( Sub(value)
                                                                     Console.WriteLine("   Attached child task
#{0} running",
                                                                                       value)
                                                                     Thread.Sleep(1000)
                                                                  End Sub, index)
                                        Next
                                        Console.WriteLine("Finished launching detached child tasks...")
                                     End Sub, TaskCreationOptions.DenyChildAttach)
      Dim continuation = t.ContinueWith( Sub(antecedent)
                                            Console.WriteLine("Executing continuation of Task {0}",
                                                              antecedent.Id)
                                         End Sub)
      continuation.Wait()
   End Sub
End Module
' The example displays output like the following:
'       Running antecedent task 1...
'       Launching attached child tasks...
'       Finished launching detached child tasks...
'          Attached child task #1 running
'          Attached child task #2 running
'          Attached child task #5 running
'          Attached child task #3 running
'       Executing continuation of Task 1
'          Attached child task #4 running
```

The final status of the antecedent task depends on the final status of any attached child tasks. The status of detached child tasks does not affect the parent. For more information, see Attached and Detached Child Tasks.

Associating State with Continuations

You can associate arbitrary state with a task continuation. The ContinueWith method provides overloaded versions that each take an Object value that represents the state of the continuation. You can later access this state object by using the Task.AsyncState property. This state object is `null` if you do not provide a value.

Continuation state is useful when you convert existing code that uses the Asynchronous Programming Model (APM) to use the TPL. In the APM, you typically provide object state in the **Begin***Method* method and later access that state by using the IAsyncResult.AsyncState property. By using the ContinueWith method, you can preserve this state when you convert code that uses the APM to use the TPL.

Continuation state can also be useful when you work with Task objects in the Visual Studio debugger. For example, in the **Parallel Tasks** window, the **Task** column displays the string representation of the state object for each task. For more information about the **Parallel Tasks** window, see Using the Tasks Window.

The following example shows how to use continuation state. It creates a chain of continuation tasks. Each task provides the current time, a DateTime object, for the `state` parameter of the ContinueWith method. Each DateTime object represents the time at which the continuation task is created. Each task produces as its result a second DateTime object that represents the time at which the task finishes. After all tasks finish, this example displays the creation time and the time at which each continuation task finishes.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

// Demonstrates how to associate state with task continuations.
class ContinuationState
{
   // Simluates a lengthy operation and returns the time at which
   // the operation completed.
   public static DateTime DoWork()
   {
      // Simulate work by suspending the current thread
      // for two seconds.
      Thread.Sleep(2000);

      // Return the current time.
      return DateTime.Now;
   }

   static void Main(string[] args)
   {
      // Start a root task that performs work.
      Task<DateTime> t = Task<DateTime>.Run(delegate { return DoWork(); });

      // Create a chain of continuation tasks, where each task is
      // followed by another task that performs work.
      List<Task<DateTime>> continuations = new List<Task<DateTime>>();
      for (int i = 0; i < 5; i++)
      {
         // Provide the current time as the state of the continuation.
         t = t.ContinueWith(delegate { return DoWork(); }, DateTime.Now);
         continuations.Add(t);
      }

      // Wait for the last task in the chain to complete.
      t.Wait();

      // Print the creation time of each continuation (the state object)
      // and the completion time (the result of that task) to the console.
      foreach (var continuation in continuations)
      {
         DateTime start = (DateTime)continuation.AsyncState;
         DateTime end = continuation.Result;

         Console.WriteLine("Task was created at {0} and finished at {1}.",
            start.TimeOfDay, end.TimeOfDay);
      }
   }
}

/* Sample output:
Task was created at 10:56:21.1561762 and finished at 10:56:25.1672062.
Task was created at 10:56:21.1610677 and finished at 10:56:27.1707646.
Task was created at 10:56:21.1610677 and finished at 10:56:29.1743230.
Task was created at 10:56:21.1610677 and finished at 10:56:31.1779883.
Task was created at 10:56:21.1610677 and finished at 10:56:33.1837083.
*/
```

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

' Demonstrates how to associate state with task continuations.
Public Module ContinuationState
   ' Simluates a lengthy operation and returns the time at which
   ' the operation completed.
   Public Function DoWork() As Date
      ' Simulate work by suspending the current thread
      ' for two seconds.
      Thread.Sleep(2000)

      ' Return the current time.
      Return Date.Now
   End Function

   Public Sub Main()
      ' Start a root task that performs work.
      Dim t As Task(Of Date) = Task(Of Date).Run(Function() DoWork())

      ' Create a chain of continuation tasks, where each task is
      ' followed by another task that performs work.
      Dim continuations As New List(Of Task(Of DateTime))()
      For i As Integer = 0 To 4
         ' Provide the current time as the state of the continuation.
         t = t.ContinueWith(Function(antecedent, state) DoWork(), DateTime.Now)
         continuations.Add(t)
      Next

      ' Wait for the last task in the chain to complete.
      t.Wait()

      ' Display the creation time of each continuation (the state object)
      ' and the completion time (the result of that task) to the console.
      For Each continuation In continuations
         Dim start As DateTime = CDate(continuation.AsyncState)
         Dim [end] As DateTime = continuation.Result

         Console.WriteLine("Task was created at {0} and finished at {1}.",
            start.TimeOfDay, [end].TimeOfDay)
      Next
   End Sub
End Module
' The example displays output like the following:
'       Task was created at 10:56:21.1561762 and finished at 10:56:25.1672062.
'       Task was created at 10:56:21.1610677 and finished at 10:56:27.1707646.
'       Task was created at 10:56:21.1610677 and finished at 10:56:29.1743230.
'       Task was created at 10:56:21.1610677 and finished at 10:56:31.1779883.
'       Task was created at 10:56:21.1610677 and finished at 10:56:33.1837083.
```

Handling Exceptions Thrown from Continuations

An antecedent-continuation relationship is not a parent-child relationship. Exceptions thrown by continuations are not propagated to the antecedent. Therefore, handle exceptions thrown by continuations as you would handle them in any other task, as follows:

- You can use the Wait, WaitAll, or WaitAny method, or its generic counterpart, to wait on the continuation. You can wait for an antecedent and its continuations in the same `try` statement, as shown in the following example.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task<int>.Run( () => { Console.WriteLine("Executing task {0}",
                                                           Task.CurrentId);
                                         return 54; });
      var continuation = task1.ContinueWith( (antecedent) =>
                                             { Console.WriteLine("Executing continuation task {0}",
                                                                 Task.CurrentId);
                                               Console.WriteLine("Value from antecedent: {0}",
                                                                 antecedent.Result);
                                               throw new InvalidOperationException();
                                            } );

      try {
         task1.Wait();
         continuation.Wait();
      }
      catch (AggregateException ae) {
          foreach (var ex in ae.InnerExceptions)
              Console.WriteLine(ex.Message);
      }
   }
}
// The example displays the following output:
//       Executing task 1
//       Executing continuation task 2
//       Value from antecedent: 54
//       Operation is not valid due to the current state of the object.
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task(Of Integer).Run(Function()
                                          Console.WriteLine("Executing task {0}",
                                                            Task.CurrentId)
                                          Return 54
                                       End Function)
      Dim continuation = task1.ContinueWith(Sub(antecedent)
                                               Console.WriteLine("Executing continuation task {0}",
                                                                 Task.CurrentId)
                                               Console.WriteLine("Value from antecedent: {0}",
                                                                 antecedent.Result)
                                               Throw New InvalidOperationException()
                                            End Sub)

      Try
           task1.Wait()
           continuation.Wait()
      Catch ae As AggregateException
          For Each ex In ae.InnerExceptions
              Console.WriteLine(ex.Message)
          Next
      End Try
   End Sub
End Module
' The example displays the following output:
'       Executing task 1
'       Executing continuation task 2
'       Value from antecedent: 54
'       Operation is not valid due to the current state of the object.
```

- You can use a second continuation to observe the Exception property of the first continuation. In the following example, a task attempts to read from a non-existent file. The continuation then displays information about the exception in the antecedent task.

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var t = Task.Run( () => { string s = File.ReadAllText(@"C:\NonexistentFile.txt");
                                return s;
                              });

      var c = t.ContinueWith( (antecedent) =>
                              { // Get the antecedent's exception information.
                                foreach (var ex in antecedent.Exception.InnerExceptions) {
                                   if (ex is FileNotFoundException)
                                      Console.WriteLine(ex.Message);
                                }
                              }, TaskContinuationOptions.OnlyOnFaulted);

      c.Wait();
   }
}
// The example displays the following output:
//        Could not find file 'C:\NonexistentFile.txt'.
```

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

Module Example
   Public Sub Main()
      Dim t = Task.Run( Function()
                           Dim s As String = File.ReadAllText("C:\NonexistentFile.txt")
                           Return s
                        End Function)

      Dim c = t.ContinueWith( Sub(antecedent)
                                 ' Get the antecedent's exception information.
                                 For Each ex In antecedent.Exception.InnerExceptions
                                    If TypeOf ex Is FileNotFoundException
                                       Console.WriteLine(ex.Message)
                                    End If
                                 Next
                              End Sub, TaskContinuationOptions.OnlyOnFaulted)

      c.Wait()
   End Sub
End Module
' The example displays the following output:
'       Could not find file 'C:\NonexistentFile.txt'.
```

Because it was run with the TaskContinuationOptions.OnlyOnFaulted option, the continuation executes only if an exception occurs in the antecedent, and therefore it can assume that the antecedent's Exception property is not `null`. If the continuation executes whether or not an exception is thrown in the antecedent, it would have to check whether the antecedent's Exception property is not `null` before attempting to handle the exception, as the following code fragment shows.

```
// Determine whether an exception occurred.
if (antecedent.Exception != null) {
   foreach (var ex in antecedent.Exception.InnerExceptions) {
      if (ex is FileNotFoundException)
         Console.WriteLine(ex.Message);
   }
}
```

```
' Determine whether an exception occurred.
 If antecedent.Exception IsNot Nothing Then
   ' Get the antecedent's exception information.
    For Each ex In antecedent.Exception.InnerExceptions
       If TypeOf ex Is FileNotFoundException
          Console.WriteLine(ex.Message)
       End If
    Next
 End If
```

For more information, see Exception Handling.

- If the continuation is an attached child task that was created by using the TaskContinuationOptions.AttachedToParent option, its exceptions will be propagated by the parent back to the calling thread, as is the case in any other attached child. For more information, see Attached and Detached Child Tasks.

See also

- Task Parallel Library (TPL)

Attached and Detached Child Tasks

1/23/2019 • 8 minutes to read • Edit Online

A *child task* (or *nested task*) is a System.Threading.Tasks.Task instance that is created in the user delegate of another task, which is known as the *parent task*. A child task can be either detached or attached. A *detached child task* is a task that executes independently of its parent. An *attached child task* is a nested task that is created with the TaskCreationOptions.AttachedToParent option whose parent does not explicitly or by default prohibit it from being attached. A task may create any number of attached and detached child tasks, limited only by system resources.

The following table lists the basic differences between the two kinds of child tasks.

CATEGORY	DETACHED CHILD TASKS	ATTACHED CHILD TASKS
Parent waits for child tasks to complete.	No	Yes
Parent propagates exceptions thrown by child tasks.	No	Yes
Status of parent depends on status of child.	No	Yes

In most scenarios, we recommend that you use detached child tasks, because their relationships with other tasks are less complex. That is why tasks created inside parent tasks are detached by default, and you must explicitly specify the TaskCreationOptions.AttachedToParent option to create an attached child task.

Detached child tasks

Although a child task is created by a parent task, by default it is independent of the parent task. In the following example, a parent task creates one simple child task. If you run the example code multiple times, you may notice that the output from the example differs from that shown, and also that the output may change each time you run the code. This occurs because the parent task and child tasks execute independently of each other; the child is a detached task. The example waits only for the parent task to complete, and the child task may not execute or complete before the console app terminates.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
         Console.WriteLine("Outer task executing.");

         var child = Task.Factory.StartNew(() => {
            Console.WriteLine("Nested task starting.");
            Thread.SpinWait(500000);
            Console.WriteLine("Nested task completing.");
         });
      });

      parent.Wait();
      Console.WriteLine("Outer has completed.");
   }
}
// The example produces output like the following:
//        Outer task executing.
//        Nested task starting.
//        Outer has completed.
//        Nested task completing.
```

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

Module Example
   Public Sub Main()
      Dim parent = Task.Factory.StartNew(Sub()
                                            Console.WriteLine("Outer task executing.")
                                            Dim child = Task.Factory.StartNew(Sub()
                                                                                 Console.WriteLine("Nested
task starting.")
                                                                                 Thread.SpinWait(500000)
                                                                                 Console.WriteLine("Nested
task completing.")
                                                                              End Sub)
                                         End Sub)
      parent.Wait()
      Console.WriteLine("Outer task has completed.")
    End Sub
End Module
' The example produces output like the following:
'   Outer task executing.
'   Nested task starting.
'   Outer task has completed.
'   Nested task completing.
```

If the child task is represented by a Task<TResult> object rather than a Task object, you can ensure that the parent task will wait for the child to complete by accessing the Task<TResult>.Result property of the child even if it is a detached child task. The Result property blocks until its task completes, as the following example shows.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      var outer = Task<int>.Factory.StartNew(() => {
            Console.WriteLine("Outer task executing.");

            var nested = Task<int>.Factory.StartNew(() => {
                  Console.WriteLine("Nested task starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Nested task completing.");
                  return 42;
            });

            // Parent will wait for this detached child.
            return nested.Result;
      });

      Console.WriteLine("Outer has returned {0}.", outer.Result);
   }
}
// The example displays the following output:
//       Outer task executing.
//       Nested task starting.
//       Nested task completing.
//       Outer has returned 42.
```

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

Module Example
   Public Sub Main()
      Dim parent = Task(Of Integer).Factory.StartNew(Function()
                                                         Console.WriteLine("Outer task executing.")
                                                         Dim child = Task(Of
Integer).Factory.StartNew(Function()

Console.WriteLine("Nested task starting.")

Thread.SpinWait(5000000)

Console.WriteLine("Nested task completing.")

Return 42
                                                                                                     End
Function)
                                                         Return child.Result


                                                     End Function)
      Console.WriteLine("Outer has returned {0}", parent.Result)
   End Sub
End Module
' The example displays the following output:
'       Outer task executing.
'       Nested task starting.
'       Detached task completing.
'       Outer has returned 42
```

Attached child tasks

Unlike detached child tasks, attached child tasks are closely synchronized with the parent. You can change the detached child task in the previous example to an attached child task by using the TaskCreationOptions.AttachedToParent option in the task creation statement, as shown in the following example. In this code, the attached child task completes before its parent. As a result, the output from the example is the same each time you run the code.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
            Console.WriteLine("Parent task executing.");
            var child = Task.Factory.StartNew(() => {
                  Console.WriteLine("Attached child starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Attached child completing.");
            }, TaskCreationOptions.AttachedToParent);
      });
      parent.Wait();
      Console.WriteLine("Parent has completed.");
   }
}
// The example displays the following output:
//       Parent task executing.
//       Attached child starting.
//       Attached child completing.
//       Parent has completed.
```

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

Module Example
   Public Sub Main()
      Dim parent = Task.Factory.StartNew(Sub()
                                             Console.WriteLine("Parent task executing")
                                             Dim child = Task.Factory.StartNew(Sub()
                                                                                    Console.WriteLine("Attached
child starting.")

                                                                                    Thread.SpinWait(5000000)
                                                                                    Console.WriteLine("Attached
child completing.")
                                                                                 End Sub,
TaskCreationOptions.AttachedToParent)
                                          End Sub)
      parent.Wait()
      Console.WriteLine("Parent has completed.")
   End Sub
End Module
' The example displays the following output:
'       Parent task executing.
'       Attached child starting.
'       Attached child completing.
'       Parent has completed.
```

You can use attached child tasks to create tightly synchronized graphs of asynchronous operations.

However, a child task can attach to its parent only if its parent does not prohibit attached child tasks. Parent tasks can explicitly prevent child tasks from attaching to them by specifying the TaskCreationOptions.DenyChildAttach option in the parent task's class constructor or the TaskFactory.StartNew method. Parent tasks implicitly prevent child tasks from attaching to them if they are created by calling the Task.Run method. The following example illustrates this. It is identical to the previous example, except that the parent task is created by calling the Task.Run(Action) method rather than the TaskFactory.StartNew(Action) method. Because the child task is not able to attach to its parent, the output from the example is unpredictable. Because the default task creation options for the Task.Run overloads include TaskCreationOptions.DenyChildAttach, this example is functionally equivalent to the first example in the "Detached child tasks" section.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Run(() => {
            Console.WriteLine("Parent task executing.");
            var child = Task.Factory.StartNew(() => {
                  Console.WriteLine("Child starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Child completing.");
            }, TaskCreationOptions.AttachedToParent);
      });
      parent.Wait();
      Console.WriteLine("Parent has completed.");
   }
}
// The example displays output like the following:
//       Parent task executing
//       Parent has completed.
//       Attached child starting.
```

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

Module Example
   Public Sub Main()
      Dim parent = Task.Run(Sub()
                              Console.WriteLine("Parent task executing")
                              Dim child = Task.Factory.StartNew(Sub()
                                             Console.WriteLine("Attached child starting.")
                                             Thread.SpinWait(5000000)
                                             Console.WriteLine("Attached child completing.")
                                          End Sub, TaskCreationOptions.AttachedToParent)
                           End Sub)
      parent.Wait()
      Console.WriteLine("Parent has completed.")
   End Sub
End Module
' The example displays output like the following:
'       Parent task executing
'       Parent has completed.
'       Attached child starting.
```

Exceptions in child tasks

If a detached child task throws an exception, that exception must be observed or handled directly in the parent task just as with any non-nested task. If an attached child task throws an exception, the exception is automatically propagated to the parent task and back to the thread that waits or tries to access the task's Task<TResult>.Result property. Therefore, by using attached child tasks, you can handle all exceptions at just one point in the call to Task.Wait on the calling thread. For more information, see Exception Handling.

Cancellation and child tasks

Task cancellation is cooperative. That is, to be cancelable, every attached or detached child task must monitor the status of the cancellation token. If you want to cancel a parent and all its children by using one cancellation request, you pass the same token as an argument to all tasks and provide in each task the logic to respond to the request in each task. For more information, see Task Cancellation and How to: Cancel a Task and Its Children.

When the parent cancels

If a parent cancels itself before its child task is started, the child never starts. If a parent cancels itself after its child task has already started, the child runs to completion unless it has its own cancellation logic. For more information, see Task Cancellation.

When a detached child task cancels

If a detached child task cancels itself by using the same token that was passed to the parent, and the parent does not wait for the child task, no exception is propagated, because the exception is treated as benign cooperation cancellation. This behavior is the same as that of any top-level task.

When an attached child task cancels

When an attached child task cancels itself by using the same token that was passed to its parent task, a TaskCanceledException is propagated to the joining thread inside an AggregateException. You must wait for the parent task so that you can handle all benign exceptions in addition to all faulting exceptions that are propagated up through a graph of attached child tasks.

For more information, see Exception Handling.

Preventing a child task from attaching to its parent

An unhandled exception that is thrown by a child task is propagated to the parent task. You can use this behavior to observe all child task exceptions from one root task instead of traversing a tree of tasks. However, exception propagation can be problematic when a parent task does not expect attachment from other code. For example, consider an app that calls a third-party library component from a Task object. If the third-party library component also creates a Task object and specifies TaskCreationOptions.AttachedToParent to attach it to the parent task, any unhandled exceptions that occur in the child task propagate to the parent. This could lead to unexpected behavior in the main app.

To prevent a child task from attaching to its parent task, specify the TaskCreationOptions.DenyChildAttach option when you create the parent Task or Task<TResult> object. When a task tries to attach to its parent and the parent specifies the TaskCreationOptions.DenyChildAttach option, the child task will not be able to attach to a parent and will execute just as if the TaskCreationOptions.AttachedToParent option was not specified.

You might also want to prevent a child task from attaching to its parent when the child task does not finish in a timely manner. Because a parent task does not finish until all child tasks finish, a long-running child task can cause the overall app to perform poorly. For an example that shows how to improve app performance by preventing a task from attaching to its parent task, see How to: Prevent a Child Task from Attaching to its Parent.

See also

- Parallel Programming
- Data Parallelism

Task Cancellation

4/28/2019 • 3 minutes to read • Edit Online

The System.Threading.Tasks.Task and System.Threading.Tasks.Task<TResult> classes support cancellation through the use of cancellation tokens in the .NET Framework. For more information, see Cancellation in Managed Threads. In the Task classes, cancellation involves cooperation between the user delegate, which represents a cancelable operation and the code that requested the cancellation. A successful cancellation involves the requesting code calling the CancellationTokenSource.Cancel method, and the user delegate terminating the operation in a timely manner. You can terminate the operation by using one of these options:

- By simply returning from the delegate. In many scenarios this is sufficient; however, a task instance that is canceled in this way transitions to the TaskStatus.RanToCompletion state, not to the TaskStatus.Canceled state.
- By throwing a OperationCanceledException and passing it the token on which cancellation was requested. The preferred way to do this is to use the ThrowIfCancellationRequested method. A task that is canceled in this way transitions to the Canceled state, which the calling code can use to verify that the task responded to its cancellation request.

The following example shows the basic pattern for task cancellation that throws the exception. Note that the token is passed to the user delegate and to the task instance itself.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        var tokenSource2 = new CancellationTokenSource();
        CancellationToken ct = tokenSource2.Token;

        var task = Task.Run(() =>
        {
            // Were we already canceled?
            ct.ThrowIfCancellationRequested();

            bool moreToDo = true;
            while (moreToDo)
            {
                // Poll on this property if you have to do
                // other cleanup before throwing.
                if (ct.IsCancellationRequested)
                {
                    // Clean up here, then...
                    ct.ThrowIfCancellationRequested();
                }

            }
        }, tokenSource2.Token); // Pass same token to Task.Run.

        tokenSource2.Cancel();

        // Just continue on this thread, or await with try-catch:
        try
        {
            await task;
        }
        catch (OperationCanceledException e)
        {
            Console.WriteLine($"{nameof(OperationCanceledException)} thrown with message: {e.Message}");
        }
        finally
        {
            tokenSource2.Dispose();
        }

        Console.ReadKey();
    }
}
```

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

Module Test
    Sub Main()
        Dim tokenSource2 As New CancellationTokenSource()
        Dim ct As CancellationToken = tokenSource2.Token

        Dim t2 = Task.Factory.StartNew(Sub()
                                           ' Were we already canceled?
                                           ct.ThrowIfCancellationRequested()

                                           Dim moreToDo As Boolean = True
                                           While moreToDo = True
                                               ' Poll on this property if you have to do
                                               ' other cleanup before throwing.
                                               If ct.IsCancellationRequested Then

                                                   ' Clean up here, then...
                                                   ct.ThrowIfCancellationRequested()
                                               End If

                                           End While
                                       End Sub _
        , tokenSource2.Token) ' Pass same token to StartNew.

        ' Cancel the task.
        tokenSource2.Cancel()

        ' Just continue on this thread, or Wait/WaitAll with try-catch:
        Try
            t2.Wait()

        Catch e As AggregateException

            For Each item In e.InnerExceptions
                Console.WriteLine(e.Message & " " & item.Message)
            Next
        Finally
           tokenSource2.Dispose()
        End Try

        Console.ReadKey()
    End Sub
End Module
```

For a more complete example, see How to: Cancel a Task and Its Children.

When a task instance observes an OperationCanceledException thrown by user code, it compares the exception's token to its associated token (the one that was passed to the API that created the Task). If they are the same and the token's IsCancellationRequested property returns true, the task interprets this as acknowledging cancellation and transitions to the Canceled state. If you do not use a Wait or WaitAll method to wait for the task, then the task just sets its status to Canceled.

If you are waiting on a Task that transitions to the Canceled state, a System.Threading.Tasks.TaskCanceledException exception (wrapped in an AggregateException exception) is thrown. Note that this exception indicates successful cancellation instead of a faulty situation. Therefore, the task's Exception property returns `null`.

If the token's IsCancellationRequested property returns false or if the exception's token does not match the Task's token, the OperationCanceledException is treated like a normal exception, causing the Task to transition to the Faulted state. Also note that the presence of other exceptions will also cause the Task to transition to the Faulted state. You can get the status of the completed task in the Status property.

It is possible that a task may continue to process some items after cancellation is requested.

See also

- Cancellation in Managed Threads
- How to: Cancel a Task and Its Children

Exception handling (Task Parallel Library)

5/15/2019 • 14 minutes to read • Edit Online

Unhandled exceptions that are thrown by user code that is running inside a task are propagated back to the calling thread, except in certain scenarios that are described later in this topic. Exceptions are propagated when you use one of the static or instance Task.Wait methods, and you handle them by enclosing the call in a `try`/ `catch` statement. If a task is the parent of attached child tasks, or if you are waiting on multiple tasks, multiple exceptions could be thrown.

To propagate all the exceptions back to the calling thread, the Task infrastructure wraps them in an AggregateException instance. The AggregateException exception has an InnerExceptions property that can be enumerated to examine all the original exceptions that were thrown, and handle (or not handle) each one individually. You can also handle the original exceptions by using the AggregateException.Handle method.

Even if only one exception is thrown, it is still wrapped in an AggregateException exception, as the following example shows.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      try
      {
          task1.Wait();
      }
      catch (AggregateException ae)
      {
          foreach (var e in ae.InnerExceptions) {
              // Handle the custom exception.
              if (e is CustomException) {
                  Console.WriteLine(e.Message);
              }
              // Rethrow any other exception.
              else {
                  throw;
              }
          }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```vb
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

      Try
         task1.Wait()
      Catch ae As AggregateException
         For Each ex In ae.InnerExceptions
            ' Handle the custom exception.
            If TypeOf ex Is CustomException Then
               Console.WriteLine(ex.Message)
            ' Rethrow any other exception.
            Else
               Throw
            End If
         Next
      End Try
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

You could avoid an unhandled exception by just catching the AggregateException and not observing any of the inner exceptions. However, we recommend that you do not do this because it is analogous to catching the base Exception type in non-parallel scenarios. To catch an exception without taking specific actions to recover from it can leave your program in an indeterminate state.

If you do not want to call the Task.Wait method to wait for a task's completion, you can also retrieve the AggregateException exception from the task's Exception property, as the following example shows. For more information, see the Observing exceptions by using the Task.Exception property section in this topic.

```csharp
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      while(! task1.IsCompleted) {}

      if (task1.Status == TaskStatus.Faulted) {
          foreach (var e in task1.Exception.InnerExceptions) {
              // Handle the custom exception.
              if (e is CustomException) {
                  Console.WriteLine(e.Message);
              }
              // Rethrow any other exception.
              else {
                  throw e;
              }
          }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```vb
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

      While Not task1.IsCompleted
      End While

      If task1.Status = TaskStatus.Faulted Then
         For Each ex In task1.Exception.InnerExceptions
            ' Handle the custom exception.
            If TypeOf ex Is CustomException Then
               Console.WriteLine(ex.Message)
            ' Rethrow any other exception.
            Else
               Throw ex
            End If
         Next
      End If
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

If you do not wait on a task that propagates an exception, or access its Exception property, the exception is escalated according to the .NET exception policy when the task is garbage-collected.

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

NOTE

When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue and see the exception-handling behavior that is demonstrated in these examples. To prevent Visual Studio from breaking on the first error, just uncheck the **Enable Just My Code** checkbox under **Tools, Options, Debugging, General**.

Attached child tasks and nested AggregateExceptions

If a task has an attached child task that throws an exception, that exception is wrapped in an AggregateException before it is propagated to the parent task, which wraps that exception in its own AggregateException before it propagates it back to the calling thread. In such cases, the InnerExceptions property of the AggregateException exception that is caught at the Task.Wait, WaitAny, or WaitAll method contains one or more AggregateException instances, not the original exceptions that caused the fault. To avoid having to iterate over nested AggregateException exceptions, you can use the Flatten method to remove all the nested AggregateException exceptions, so that the AggregateException.InnerExceptions property contains the original exceptions. In the following example, nested AggregateException instances are flattened and handled in just one loop.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Factory.StartNew(() => {
                     var child1 = Task.Factory.StartNew(() => {
                        var child2 = Task.Factory.StartNew(() => {
                            // This exception is nested inside three AggregateExceptions.
                            throw new CustomException("Attached child2 faulted.");
                        }, TaskCreationOptions.AttachedToParent);

                        // This exception is nested inside two AggregateExceptions.
                        throw new CustomException("Attached child1 faulted.");
                     }, TaskCreationOptions.AttachedToParent);
      });

      try {
         task1.Wait();
      }
      catch (AggregateException ae) {
         foreach (var e in ae.Flatten().InnerExceptions) {
            if (e is CustomException) {
               Console.WriteLine(e.Message);
            }
            else {
               throw;
            }
         }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//    Attached child1 faulted.
//    Attached child2 faulted.
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task.Factory.StartNew(Sub()
                                          Dim child1 = Task.Factory.StartNew(Sub()
                                                                                Dim child2 =
Task.Factory.StartNew(Sub()

Throw New CustomException("Attached child2 faulted.")

End Sub,

TaskCreationOptions.AttachedToParent)

Throw New CustomException("Attached child1 faulted.")
                                                                             End Sub,

TaskCreationOptions.AttachedToParent)
                                       End Sub)

      Try
         task1.Wait()
      Catch ae As AggregateException
         For Each ex In ae.Flatten().InnerExceptions
            If TypeOf ex Is CustomException Then
               Console.WriteLine(ex.Message)
            Else
               Throw
            End If
         Next
      End Try
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays the following output:
'       Attached child1 faulted.
'       Attached child2 faulted.
```

You can also use the AggregateException.Flatten method to rethrow the inner exceptions from multiple AggregateException instances thrown by multiple tasks in a single AggregateException instance, as the following example shows.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
public class Example
{
   public static void Main()
   {
        try {
            ExecuteTasks();
        }
        catch (AggregateException ae) {
            foreach (var e in ae.InnerExceptions) {
                Console.WriteLine("{0}:\n   {1}", e.GetType().Name, e.Message);
            }
        }
   }

   static void ExecuteTasks()
   {
        // Assume this is a user-entered String.
        String path = @"C:\";
        List<Task> tasks = new List<Task>();

        tasks.Add(Task.Run(() => {
                             // This should throw an UnauthorizedAccessException.
                              return Directory.GetFiles(path, "*.txt",
                                                         SearchOption.AllDirectories);
                           }));

        tasks.Add(Task.Run(() => {
                              if (path == @"C:\")
                                 throw new ArgumentException("The system root is not a valid path.");
                              return new String[] { ".txt", ".dll", ".exe", ".bin", ".dat" };
                           }));

        tasks.Add(Task.Run(() => {
                               throw new NotImplementedException("This operation has not been implemented.");
                           }));

        try {
            Task.WaitAll(tasks.ToArray());
        }
        catch (AggregateException ae) {
            throw ae.Flatten();
        }
    }
}
// The example displays the following output:
//       UnauthorizedAccessException:
//          Access to the path 'C:\Documents and Settings' is denied.
//       ArgumentException:
//          The system root is not a valid path.
//       NotImplementedException:
//          This operation has not been implemented.
```

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

Module Example
    Public Sub Main()
       Try
          ExecuteTasks()
       Catch ae As AggregateException
          For Each e In ae.InnerExceptions
             Console.WriteLine("{0}:{2}   {1}", e.GetType().Name, e.Message,
                               vbCrLf)
          Next
       End Try
    End Sub

    Sub ExecuteTasks()
        ' Assume this is a user-entered String.
        Dim path = "C:\"
        Dim tasks As New List(Of Task)

        tasks.Add(Task.Run(Function()
                             ' This should throw an UnauthorizedAccessException.
                              Return Directory.GetFiles(path, "*.txt",
                                                        SearchOption.AllDirectories)
                           End Function))

        tasks.Add(Task.Run(Function()
                              If path = "C:\" Then
                                 Throw New ArgumentException("The system root is not a valid path.")
                              End If
                              Return { ".txt", ".dll", ".exe", ".bin", ".dat" }
                           End Function))

        tasks.Add(Task.Run(Sub()
                              Throw New NotImplementedException("This operation has not been implemented.")
                           End Sub))

        Try
            Task.WaitAll(tasks.ToArray)
        Catch ae As AggregateException
            Throw ae.Flatten()
        End Try
    End Sub
End Module
' The example displays the following output:
'       UnauthorizedAccessException:
'          Access to the path 'C:\Documents and Settings' is denied.
'       ArgumentException:
'          The system root is not a valid path.
'       NotImplementedException:
'          This operation has not been implemented.
```

Exceptions from detached child tasks

By default, child tasks are created as detached. Exceptions thrown from detached tasks must be handled or rethrown in the immediate parent task; they are not propagated back to the calling thread in the same way as attached child tasks propagated back. The topmost parent can manually rethrow an exception from a detached child to cause it to be wrapped in an AggregateException and propagated back to the calling thread.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run(() => {
                       var nested1 = Task.Run(() => {
                                          throw new CustomException("Detached child task faulted.");
                                     });

          // Here the exception will be escalated back to the calling thread.
          // We could use try/catch here to prevent that.
          nested1.Wait();
      });

      try {
         task1.Wait();
      }
      catch (AggregateException ae) {
         foreach (var e in ae.Flatten().InnerExceptions) {
            if (e is CustomException) {
               Console.WriteLine(e.Message);
            }
         }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//    Detached child task faulted.
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task.Run(Sub()
                              Dim nestedTask1 = Task.Run(Sub()
                                                            Throw New CustomException("Detached child task
faulted.")
                                                         End Sub)
                              ' Here the exception will be escalated back to joining thread.
                              ' We could use try/catch here to prevent that.
                              nestedTask1.Wait()
                           End Sub)

      Try
          task1.Wait()
      Catch ae As AggregateException
          For Each ex In ae.Flatten().InnerExceptions
              If TypeOf ex Is CustomException Then
                  ' Recover from the exception. Here we just
                  ' print the message for demonstration purposes.
                  Console.WriteLine(ex.Message)
              End If
          Next
      End Try
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays the following output:
'       Detached child task faulted.
```

Even if you use a continuation to observe an exception in a child task, the exception still must be observed by the parent task.

Exceptions that indicate cooperative cancellation

When user code in a task responds to a cancellation request, the correct procedure is to throw an OperationCanceledException passing in the cancellation token on which the request was communicated. Before it attempts to propagate the exception, the task instance compares the token in the exception to the one that was passed to it when it was created. If they are the same, the task propagates a TaskCanceledException wrapped in the AggregateException, and it can be seen when the inner exceptions are examined. However, if the calling thread is not waiting on the task, this specific exception will not be propagated. For more information, see Task Cancellation.

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var task1 = Task.Factory.StartNew(() =>
{
    CancellationToken ct = token;
    while (someCondition)
    {
        // Do some work...
        Thread.SpinWait(50000);
        ct.ThrowIfCancellationRequested();
    }
},
token);

// No waiting required.
tokenSource.Dispose();
```

```
Dim someCondition As Boolean = True
Dim tokenSource = New CancellationTokenSource()
Dim token = tokenSource.Token

Dim task1 = Task.Factory.StartNew(Sub()
                                      Dim ct As CancellationToken = token
                                      While someCondition = True
                                          ' Do some work...
                                          Thread.SpinWait(500000)
                                          ct.ThrowIfCancellationRequested()
                                      End While
                                  End Sub,
                                  token)
```

Using the handle method to filter inner exceptions

You can use the AggregateException.Handle method to filter out exceptions that you can treat as "handled" without using any further logic. In the user delegate that is supplied to the AggregateException.Handle(Func<Exception,Boolean>) method, you can examine the exception type, its Message property, or any other information about it that will let you determine whether it is benign. Any exceptions for which the delegate returns `false` are rethrown in a new AggregateException instance immediately after the AggregateException.Handle method returns.

The following example is functionally equivalent to the first example in this topic, which examines each exception in the AggregateException.InnerExceptions collection. Instead, this exception handler calls the AggregateException.Handle method object for each exception, and only rethrows exceptions that are not `CustomException` instances.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      try {
          task1.Wait();
      }
      catch (AggregateException ae)
      {
         // Call the Handle method to handle the custom exception,
         // otherwise rethrow the exception.
         ae.Handle(ex => { if (ex is CustomException)
                             Console.WriteLine(ex.Message);
                          return ex is CustomException;
                        });
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

      Try
         task1.Wait()
      Catch ae As AggregateException
         ' Call the Handle method to handle the custom exception,
         ' otherwise rethrow the exception.
         ae.Handle(Function(e)
                      If TypeOf e Is CustomException Then
                         Console.WriteLine(e.Message)
                      End If
                      Return TypeOf e Is CustomException
                   End Function)
      End Try
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

The following is a more complete example that uses the AggregateException.Handle method to provide special handling for an UnauthorizedAccessException exception when enumerating files.

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        // This should throw an UnauthorizedAccessException.
       try {
           var files = GetAllFiles(@"C:\");
           if (files != null)
              foreach (var file in files)
                 Console.WriteLine(file);
        }
        catch (AggregateException ae) {
           foreach (var ex in ae.InnerExceptions)
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message);
        }
        Console.WriteLine();

        // This should throw an ArgumentException.
        try {
           foreach (var s in GetAllFiles(""))
              Console.WriteLine(s);
        }
        catch (AggregateException ae) {
           foreach (var ex in ae.InnerExceptions)
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message);
        }
    }

    static string[] GetAllFiles(string path)
    {
       var task1 = Task.Run( () => Directory.GetFiles(path, "*.txt",
                                                      SearchOption.AllDirectories));

       try {
          return task1.Result;
       }
       catch (AggregateException ae) {
          ae.Handle( x => { // Handle an UnauthorizedAccessException
                            if (x is UnauthorizedAccessException) {
                                Console.WriteLine("You do not have permission to access all folders in this
path.");
                                Console.WriteLine("See your network administrator or try another path.");
                            }
                            return x is UnauthorizedAccessException;
                          });
          return Array.Empty<String>();
       }
   }
}
// The example displays the following output:
//       You do not have permission to access all folders in this path.
//       See your network administrator or try another path.
//
//       ArgumentException: The path is not of a legal form.
```

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

Module Example
    Public Sub Main()
        ' This should throw an UnauthorizedAccessException.
       Try
           Dim files = GetAllFiles("C:\")
           If files IsNot Nothing Then
              For Each file In files
                 Console.WriteLine(file)
              Next
           End If
        Catch ae As AggregateException
           For Each ex In ae.InnerExceptions
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message)
           Next
        End Try
        Console.WriteLine()

       ' This should throw an ArgumentException.
        Try
           For Each s In GetAllFiles("")
              Console.WriteLine(s)
           Next
        Catch ae As AggregateException
           For Each ex In ae.InnerExceptions
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message)
           Next
        End Try
        Console.WriteLine()
    End Sub

    Function GetAllFiles(ByVal path As String) As String()
       Dim task1 = Task.Run( Function()
                                Return Directory.GetFiles(path, "*.txt",
                                                          SearchOption.AllDirectories)
                             End Function)
       Try
          Return task1.Result
       Catch ae As AggregateException
          ae.Handle( Function(x)
                        ' Handle an UnauthorizedAccessException
                        If TypeOf x Is UnauthorizedAccessException Then
                            Console.WriteLine("You do not have permission to access all folders in this
path.")
                            Console.WriteLine("See your network administrator or try another path.")
                        End If
                        Return TypeOf x Is UnauthorizedAccessException
                     End Function)
       End Try
       Return Array.Empty(Of String)()
    End Function
End Module
' The example displays the following output:
'       You do not have permission to access all folders in this path.
'       See your network administrator or try another path.
'
'       ArgumentException: The path is not of a legal form.
```

Observing exceptions by using the Task.Exception property

If a task completes in the TaskStatus.Faulted state, its Exception property can be examined to discover which specific exception caused the fault. A good way to observe the Exception property is to use a continuation that runs only if the antecedent task faults, as shown in the following example.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run(() =>
                           { throw new CustomException("task1 faulted.");
      }).ContinueWith( t => { Console.WriteLine("{0}: {1}",
                                                t.Exception.InnerException.GetType().Name,
                                                t.Exception.InnerException.Message);
                            }, TaskContinuationOptions.OnlyOnFaulted);
      Thread.Sleep(500);
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays output like the following:
//        CustomException: task1 faulted.
```

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

Module Example
   Public Sub Main()
      Dim task1 = Task.Factory.StartNew(Sub()
                                           Throw New CustomException("task1 faulted.")
                                        End Sub).
                  ContinueWith(Sub(t)
                                  Console.WriteLine("{0}: {1}",
                                                    t.Exception.InnerException.GetType().Name,
                                                    t.Exception.InnerException.Message)
                               End Sub, TaskContinuationOptions.OnlyOnFaulted)

      Thread.Sleep(500)
   End Sub
End Module

Class CustomException : Inherits Exception
   Public Sub New(s As String)
      MyBase.New(s)
   End Sub
End Class
' The example displays output like the following:
'       CustomException: task1 faulted.
```

In a real application, the continuation delegate could log detailed information about the exception and possibly spawn new tasks to recover from the exception.

UnobservedTaskException event

In some scenarios, such as when hosting untrusted plug-ins, benign exceptions might be common, and it might be too difficult to manually observe them all. In these cases, you can handle the TaskScheduler.UnobservedTaskException event. The System.Threading.Tasks.UnobservedTaskExceptionEventArgs instance that is passed to your handler can be used to prevent the unobserved exception from being propagated back to the joining thread.

See also

- Task Parallel Library (TPL)

How to: Use Parallel.Invoke to Execute Parallel Operations

9/19/2018 • 4 minutes to read • Edit Online

This example shows how to parallelize operations by using Invoke in the Task Parallel Library. Three operations are performed on a shared data source. Because none of the operations modifies the source, they can be executed in parallel in a straightforward manner.

> **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.

Example

```
namespace ParallelTasks
{
    using System;
    using System.IO;
    using System.Linq;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using System.Net;

    class ParallelInvoke
    {
        static void Main()
        {
            // Retrieve Goncharov's "Oblomov" from Gutenberg.org.
            string[] words = CreateWordArray(@"http://www.gutenberg.org/files/54700/54700-0.txt");

            #region ParallelTasks
            // Perform three tasks in parallel on the source array
            Parallel.Invoke(() =>
                             {
                                 Console.WriteLine("Begin first task...");
                                 GetLongestWord(words);
                             },  // close first Action

                             () =>
                             {
                                 Console.WriteLine("Begin second task...");
                                 GetMostCommonWords(words);
                             }, //close second Action

                             () =>
                             {
                                 Console.WriteLine("Begin third task...");
                                 GetCountForWord(words, "sleep");
                             } //close third Action
                         ); //close parallel.invoke

            Console.WriteLine("Returned from Parallel.Invoke");
            #endregion

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        #region HelperMethods
        private static void GetCountForWord(string[] words, string term)
        {
            var findWord = from word in words
                           where word.ToUpper().Contains(term.ToUpper())
                           select word;

            Console.WriteLine($@"Task 3 -- The word ""{term}"" occurs {findWord.Count()} times.");
        }

        private static void GetMostCommonWords(string[] words)
        {
            var frequencyOrder = from word in words
                                 where word.Length > 6
                                 group word by word into g
                                 orderby g.Count() descending
                                 select g.Key;

            var commonWords = frequencyOrder.Take(10);

            StringBuilder sb = new StringBuilder();
            sb.AppendLine("Task 2 -- The most common words are:");
            foreach (var v in commonWords)
            {
                sb.AppendLine("  " + v);
            }
            Console.WriteLine(sb.ToString());
        }

        private static string GetLongestWord(string[] words)
        {
            var longestWord = (from w in words
                               orderby w.Length descending
                               select w).First();

            Console.WriteLine($"Task 1 -- The longest word is {longestWord}.");
            return longestWord;
        }

        // An http request performed synchronously for simplicity.
        static string[] CreateWordArray(string uri)
        {
            Console.WriteLine($"Retrieving from {uri}");

            // Download a web page the easy way.
            string s = new WebClient().DownloadString(uri);

            // Separate string into an array of words, removing some common punctuation.
            return s.Split(
                new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '_', '/' },
                StringSplitOptions.RemoveEmptyEntries);
        }
        #endregion
    }
}
//        The example displays output like the following:
//              Retrieving from http://www.gutenberg.org/files/54700/54700-0.txt
//              Begin first task...
//              Begin second task...
//              Begin third task...
//              Task 2 -- The most common words are:
//              Oblomov
//              himself
//              Schtoltz
//              Gutenberg
//              Project
//              another
//              thought
//              Oblomov's
//              nothing
//              replied
//
//              Task 1 -- The longest word is incomprehensible.
//              Task 3 -- The word "sleep" occurs 57 times.
//              Returned from Parallel.Invoke
//              Press any key to exit
```

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

Module ParallelTasks
    Sub Main()
        ' Retrieve Goncharov's "Oblomov" from Gutenberg.org.
        Dim words As String() = CreateWordArray("http://www.gutenberg.org/files/54700/54700-0.txt")

        '#Region "ParallelTasks"
        ' Perform three tasks in parallel on the source array
        Parallel.Invoke(Sub()
                            Console.WriteLine("Begin first task...")
                            GetLongestWord(words)
                            ' close first Action
                        End Sub,
            Sub()
                Console.WriteLine("Begin second task...")
                GetMostCommonWords(words)
                'close second Action
            End Sub,
            Sub()
                Console.WriteLine("Begin third task...")
                GetCountForWord(words, "sleep")
                'close third Action
            End Sub)
        'close parallel.invoke
        Console.WriteLine("Returned from Parallel.Invoke")
        '#End Region

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

#Region "HelperMethods"
    Sub GetCountForWord(ByVal words As String(), ByVal term As String)
        Dim findWord = From word In words
            Where word.ToUpper().Contains(term.ToUpper())
            Select word

        Console.WriteLine($"Task 3 -- The word ""{term}"" occurs {findWord.Count()} times.")
    End Sub

    Sub GetMostCommonWords(ByVal words As String())
        Dim frequencyOrder = From word In words
            Where word.Length > 6
            Group By word
            Into wordGroup = Group, Count()
            Order By wordGroup.Count() Descending
            Select wordGroup

        Dim commonWords = From grp In frequencyOrder
                          Select grp
                          Take (10)

        Dim s As String
        s = "Task 2 -- The most common words are:" & vbCrLf
        For Each v In commonWords
            s = s & v(0) & vbCrLf
        Next
        Console.WriteLine(s)
    End Sub

    Function GetLongestWord(ByVal words As String()) As String
        Dim longestWord = (From w In words
            Order By w.Length Descending
            Select w).First()

        Console.WriteLine($"Task 1 -- The longest word is {longestWord}.")
        Return longestWord
    End Function


    ' An http request performed synchronously for simplicity.
    Function CreateWordArray(ByVal uri As String) As String()
        Console.WriteLine($"Retrieving from {uri}")

        ' Download a web page the easy way.
        Dim s As String = New WebClient().DownloadString(uri)

        ' Separate string into an array of words, removing some common punctuation.
        Return s.Split(New Char() {" "c, ControlChars.Lf, ","c, "."c, ";"c, ":"c,
        "-"c, "_"c, "/"c}, StringSplitOptions.RemoveEmptyEntries)
    End Function
#End Region
End Module
' The exmaple displays output like the following:
'       Retrieving from http://www.gutenberg.org/files/54700/54700-0.txt
'       Begin first task...
'       Begin second task...
'       Begin third task...
'       Task 2 -- The most common words are:
'       Oblomov
'       himself
'       Schtoltz
'       Gutenberg
'       Project
'       another
'       thought
'       Oblomov's
'       nothing
'       replied
'
'       Task 1 -- The longest word is incomprehensible.
'       Task 3 -- The word "sleep" occurs 57 times.
'       Returned from Parallel.Invoke
'       Press any key to exit
```

Note that with Invoke, you simply express which actions you want to run concurrently, and the runtime handles all thread scheduling details, including scaling automatically to the number of cores on the host computer.

This example parallelizes the operations, not the data. As an alternate approach, you can parallelize the LINQ queries by using PLINQ and run the queries sequentially. Alternatively, you could parallelize the data by using PLINQ. Another option is to parallelize both the queries and the tasks. Although the resulting overhead might degrade performance on host computers with relatively few processors, it would scale much better on computers with many processors.

Compile the Code

Copy and paste the entire example into a Microsoft Visual Studio project and press **F5**.

See also

- Parallel Programming
- How to: Cancel a Task and Its Children
- Parallel LINQ (PLINQ)

How to: Return a Value from a Task

1/23/2019 • 2 minutes to read • Edit Online

This example shows how to use the System.Threading.Tasks.Task<TResult> type to return a value from the Result property. It requires that the C:\Users\Public\Pictures\Sample Pictures\ directory exists, and that it contains files.

Example

```
using System;
using System.Linq;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        // Return a value type with a lambda expression
        Task<int> task1 = Task<int>.Factory.StartNew(() => 1);
        int i = task1.Result;

        // Return a named reference type with a multi-line statement lambda.
        Task<Test> task2 = Task<Test>.Factory.StartNew(() =>
        {
            string s = ".NET";
            double d = 4.0;
            return new Test { Name = s, Number = d };
        });
        Test test = task2.Result;

        // Return an array produced by a PLINQ query
        Task<string[]> task3 = Task<string[]>.Factory.StartNew(() =>
        {
            string path = @"C:\Users\Public\Pictures\Sample Pictures\";
            string[] files = System.IO.Directory.GetFiles(path);

            var result = (from file in files.AsParallel()
                          let info = new System.IO.FileInfo(file)
                          where info.Extension == ".jpg"
                          select file).ToArray();

            return result;
        });

        foreach (var name in task3.Result)
            Console.WriteLine(name);

    }
    class Test
    {
        public string Name { get; set; }
        public double Number { get; set; }

    }
}
```

```
Imports System.Threading.Tasks

Module Module1

    Sub Main()
        ReturnAValue()

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

    End Sub

    Sub ReturnAValue()

        ' Return a value type with a lambda expression
        Dim task1 = Task(Of Integer).Factory.StartNew(Function() 1)
        Dim i As Integer = task1.Result

        ' Return a named reference type with a multi-line statement lambda.
        Dim task2 As Task(Of Test) = Task.Factory.StartNew(Function()
                                                               Dim s As String = ".NET"
                                                               Dim d As Integer = 4
                                                               Return New Test With {.Name = s, .Number = d}
                                                           End Function)

        Dim myTest As Test = task2.Result
        Console.WriteLine(myTest.Name & ": " & myTest.Number)

        ' Return an array produced by a PLINQ query.
        Dim task3 As Task(Of String())= Task(Of String()).Factory.StartNew(Function()

                                                           Dim path = "C:\Users\Public\Pictures\Sample 
Pictures\"
                                                           Dim files = System.IO.Directory.GetFiles(path)

                                                           Dim result = (From file In files.AsParallel()
                                                                Let info = New System.IO.FileInfo(file)
                                                                Where info.Extension = ".jpg"
                                                                Select file).ToArray()
                                                           Return result
                                                       End Function)

        For Each name As String In task3.Result
            Console.WriteLine(name)
        Next
    End Sub

    Class Test
        Public Name As String
        Public Number As Double
    End Class
End Module
```

The Result property blocks the calling thread until the task finishes.

To see how to pass the result of one System.Threading.Tasks.Task<TResult> to a continuation task, see Chaining Tasks by Using Continuation Tasks.

See also

- Task-based Asynchronous Programming
- Lambda Expressions in PLINQ and TPL

How to: Cancel a Task and Its Children

4/9/2019 • 7 minutes to read • Edit Online

These examples show how to perform the following tasks:

1. Create and start a cancelable task.
2. Pass a cancellation token to your user delegate and optionally to the task instance.
3. Notice and respond to the cancellation request in your user delegate.
4. Optionally notice on the calling thread that the task was canceled.

The calling thread does not forcibly end the task; it only signals that cancellation is requested. If the task is already running, it is up to the user delegate to notice the request and respond appropriately. If cancellation is requested before the task runs, then the user delegate is never executed and the task object transitions into the Canceled state.

Example

This example shows how to terminate a Task and its children in response to a cancellation request. It also shows that when a user delegate terminates by throwing a TaskCanceledException, the calling thread can optionally use the Wait method or WaitAll method to wait for the tasks to finish. In this case, you must use a `try/catch` block to handle the exceptions on the calling thread.

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
    {
        var tokenSource = new CancellationTokenSource();
        var token = tokenSource.Token;

        // Store references to the tasks so that we can wait on them and
        // observe their status after cancellation.
        Task t;
        var tasks = new ConcurrentBag<Task>();

        Console.WriteLine("Press any key to begin tasks...");
        Console.ReadKey(true);
        Console.WriteLine("To terminate the example, press 'c' to cancel and exit...");
        Console.WriteLine();

        // Request cancellation of a single task when the token source is canceled.
        // Pass the token to the user delegate, and also to the task so it can
        // handle the exception correctly.
        t = Task.Run(() => DoSomeWork(1, token), token);
        Console.WriteLine("Task {0} executing", t.Id);
        tasks.Add(t);

        // Request cancellation of a task and its children. Note the token is passed
        // to (1) the user delegate and (2) as the second argument to Task.Run, so
        // that the task instance can correctly handle the OperationCanceledException.
        t = Task.Run(() =>
        {
            // Create some cancelable child tasks.
            Task tc;
            for (int i = 3; i <= 10; i++)
            {
                // For each child task, pass the same token
                // to each user delegate and to Task.Run.
                tc = Task.Run(() => DoSomeWork(i, token), token);
                Console.WriteLine("Task {0} executing", tc.Id);
                tasks.Add(tc);
                // Pass the same token again to do work on the parent task.
                // All will be signaled by the call to tokenSource.Cancel below.
                DoSomeWork(2, token);
            }
        }, token);

        Console.WriteLine("Task {0} executing", t.Id);
        tasks.Add(t);

        // Request cancellation from the UI thread.
        char ch = Console.ReadKey().KeyChar;
        if (ch == 'c' || ch == 'C')
        {
            tokenSource.Cancel();
            Console.WriteLine("\nTask cancellation requested.");

            // Optional: Observe the change in the Status property on the task.
            // It is not necessary to wait on tasks that have canceled. However,
            // if you do wait, you must enclose the call in a try-catch block to
            // catch the TaskCanceledExceptions that are thrown. If you do
            // not wait, no exception is thrown if the token that was passed to the
            // Task.Run method is the same token that requested the cancellation.
        }

        try
        {
            await Task.WhenAll(tasks.ToArray());
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine($"\n{nameof(OperationCanceledException)} thrown\n");
        }
        finally
        {
            tokenSource.Dispose();
        }

        // Display status of all tasks.
        foreach (var task in tasks)
            Console.WriteLine("Task {0} status is now {1}", task.Id, task.Status);
    }

    static void DoSomeWork(int taskNum, CancellationToken ct)
    {
        // Was cancellation already requested?
        if (ct.IsCancellationRequested)
        {
            Console.WriteLine("Task {0} was cancelled before it got started.",
                              taskNum);
            ct.ThrowIfCancellationRequested();
        }

        int maxIterations = 100;

        // NOTE!!! A "TaskCanceledException was unhandled
        // by user code" error will be raised here if "Just My Code"
        // is enabled on your computer. On Express editions JMC is
        // enabled and cannot be disabled. The exception is benign.
        // Just press F5 to continue executing your code.
        for (int i = 0; i <= maxIterations; i++)
        for (int i = 0; i < maxIterations; i++)
        {
            // Do a bit of work. Not too much.
            var sw = new SpinWait();
            for (int j = 0; j <= 100; j++)
                sw.SpinOnce();

            if (ct.IsCancellationRequested)
            {
                Console.WriteLine("Task {0} cancelled", taskNum);
                ct.ThrowIfCancellationRequested();
            }
        }
    }
}
// The example displays output like the following:
//       Press any key to begin tasks...
//    To terminate the example, press 'c' to cancel and exit...
//
//    Task 1 executing
//    Task 2 executing
//    Task 3 executing
//    Task 4 executing
//    Task 5 executing
//    Task 6 executing
//    Task 7 executing
//    Task 8 executing
//    c
//    Task cancellation requested.
//    Task 2 cancelled
//    Task 7 cancelled
//
//    OperationCanceledException thrown
//
//    Task 2 status is now Canceled
//    Task 1 status is now RanToCompletion
//    Task 8 status is now Canceled
//    Task 7 status is now Canceled
//    Task 6 status is now RanToCompletion
//    Task 5 status is now RanToCompletion
//    Task 4 status is now RanToCompletion
//    Task 3 status is now RanToCompletion
```

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

Module Example
    Sub Main()
        Dim tokenSource As New CancellationTokenSource()
        Dim token As CancellationToken = tokenSource.Token

        ' Store references to the tasks so that we can wait on them and
        ' observe their status after cancellation.
        Dim t As Task
        Dim tasks As New ConcurrentBag(Of Task)()

        Console.WriteLine("Press any key to begin tasks...")
        Console.ReadKey(True)
        Console.WriteLine("To terminate the example, press 'c' to cancel and exit...")
        Console.WriteLine()

        ' Request cancellation of a single task when the token source is canceled.
        ' Pass the token to the user delegate, and also to the task so it can
        ' handle the exception correctly.
        t = Task.Factory.StartNew(Sub() DoSomeWork(1, token), token)
        Console.WriteLine("Task {0} executing", t.Id)
        tasks.Add(t)
        ' Request cancellation of a task and its children. Note the token is passed
        ' to (1) the user delegate and (2) as the second argument to StartNew, so
        ' that the task instance can correctly handle the OperationCanceledException.
        t = Task.Factory.StartNew(Sub()
                                     ' Create some cancelable child tasks.
                                     Dim tc As Task
                                     For i As Integer = 3 To 10
                                         ' For each child task, pass the same token
                                         ' to each user delegate and to StartNew.
                                         tc = Task.Factory.StartNew(Sub(iteration) DoSomeWork(iteration,
token), i, token)
                                         Console.WriteLine("Task {0} executing", tc.Id)
                                         tasks.Add(tc)
                                         ' Pass the same token again to do work on the parent task.
                                         ' All will be signaled by the call to tokenSource.Cancel below.
                                         DoSomeWork(2, token)
                                     Next
                                 End Sub,
                                 token)

        Console.WriteLine("Task {0} executing", t.Id)
        tasks.Add(t)

        ' Request cancellation from the UI thread.
        Dim ch As Char = Console.ReadKey().KeyChar
        If ch = "c"c Or ch = "C"c Then
            tokenSource.Cancel()
            Console.WriteLine(vbCrLf + "Task cancellation requested.")

            ' Optional: Observe the change in the Status property on the task.
            ' It is not necessary to wait on tasks that have canceled. However,
            ' if you do wait, you must enclose the call in a try-catch block to
            ' catch the TaskCanceledExceptions that are thrown. If you do
            ' not wait, no exception is thrown if the token that was passed to the
            ' StartNew method is the same token that requested the cancellation.
        End If

        Try
            Task.WaitAll(tasks.ToArray())
        Catch e As AggregateException
           Console.WriteLine()
           Console.WriteLine("AggregateException thrown with the following inner exceptions:")
           ' Display information about each exception.
           For Each v In e.InnerExceptions
              If TypeOf v Is TaskCanceledException
                 Console.WriteLine("   TaskCanceledException: Task {0}",
                                   DirectCast(v, TaskCanceledException).Task.Id)
              Else
                 Console.WriteLine("   Exception: {0}", v.GetType().Name)
              End If
           Next
           Console.WriteLine()
        Finally
           tokenSource.Dispose()
        End Try

        ' Display status of all tasks.
        For Each t In tasks
            Console.WriteLine("Task {0} status is now {1}", t.Id, t.Status)
        Next
    End Sub

    Sub DoSomeWork(ByVal taskNum As Integer, ByVal ct As CancellationToken)
        ' Was cancellation already requested?
        If ct.IsCancellationRequested = True Then
            Console.WriteLine("Task {0} was cancelled before it got started.",
                              taskNum)
            ct.ThrowIfCancellationRequested()
            ct.ThrowIfCancellationRequested()
        End If

        Dim maxIterations As Integer = 100

        ' NOTE!!! A "TaskCanceledException was unhandled
        ' by user code" error will be raised here if "Just My Code"
        ' is enabled on your computer. On Express editions JMC is
        ' enabled and cannot be disabled. The exception is benign.
        ' Just press F5 to continue executing your code.
        For i As Integer = 0 To maxIterations
            ' Do a bit of work. Not too much.
            Dim sw As New SpinWait()
            For j As Integer = 0 To 100
                sw.SpinOnce()
            Next
            If ct.IsCancellationRequested Then
                Console.WriteLine("Task {0} cancelled", taskNum)
                ct.ThrowIfCancellationRequested()
            End If
        Next
    End Sub
End Module
' The example displays output like the following:
'    Press any key to begin tasks...
'    To terminate the example, press 'c' to cancel and exit...
'
'    Task 1 executing
'    Task 2 executing
'    Task 3 executing
'    Task 4 executing
'    Task 5 executing
'    Task 6 executing
'    Task 7 executing
'    Task 8 executing
'    c
'    Task cancellation requested.
'    Task 2 cancelled
'    Task 7 cancelled
'
'    AggregateException thrown with the following inner exceptions:
'       TaskCanceledException: Task 2
'       TaskCanceledException: Task 8
'       TaskCanceledException: Task 7
'
'    Task 2 status is now Canceled
'    Task 1 status is now RanToCompletion
'    Task 8 status is now Canceled
'    Task 7 status is now Canceled
'    Task 6 status is now RanToCompletion
'    Task 5 status is now RanToCompletion
'    Task 4 status is now RanToCompletion
'    Task 3 status is now RanToCompletion
```

The System.Threading.Tasks.Task class is fully integrated with the cancellation model that is based on the System.Threading.CancellationTokenSource and System.Threading.CancellationToken types. For more information, see Cancellation in Managed Threads and Task Cancellation.

See also

- System.Threading.CancellationTokenSource
- System.Threading.CancellationToken
- System.Threading.Tasks.Task
- System.Threading.Tasks.Task<TResult>
- Task-based Asynchronous Programming

- Attached and Detached Child Tasks
- Lambda Expressions in PLINQ and TPL

How to: Create Pre-Computed Tasks

5/14/2019 • 3 minutes to read • Edit Online

This document describes how to use the Task.FromResult method to retrieve the results of asynchronous download operations that are held in a cache. The FromResult method returns a finished Task<TResult> object that holds the provided value as its Result property. This method is useful when you perform an asynchronous operation that returns a Task<TResult> object, and the result of that Task<TResult> object is already computed.

Example

The following example downloads strings from the web. It defines the `DownloadStringAsync` method. This method downloads strings from the web asynchronously. This example also uses a ConcurrentDictionary<TKey,TValue> object to cache the results of previous operations. If the input address is held in this cache, `DownloadStringAsync` uses the FromResult method to produce a Task<TResult> object that holds the content at that address. Otherwise, `DownloadStringAsync` downloads the file from the web and adds the result to the cache.

```
using System;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Linq;
using System.Net;
using System.Threading.Tasks;

// Demonstrates how to use Task<TResult>.FromResult to create a task
// that holds a pre-computed result.
class CachedDownloads
{
   // Holds the results of download operations.
   static ConcurrentDictionary<string, string> cachedDownloads =
      new ConcurrentDictionary<string, string>();

   // Asynchronously downloads the requested resource as a string.
   public static Task<string> DownloadStringAsync(string address)
   {
      // First try to retrieve the content from cache.
      string content;
      if (cachedDownloads.TryGetValue(address, out content))
      {
         return Task.FromResult<string>(content);
      }

      // If the result was not in the cache, download the
      // string and add it to the cache.
      return Task.Run(async () =>
      {
         content = await new WebClient().DownloadStringTaskAsync(address);
         cachedDownloads.TryAdd(address, content);
         return content;
      });
   }

   static void Main(string[] args)
   {
      // The URLs to download.
      string[] urls = new string[]
      {
         "http://msdn.microsoft.com",
         "http://www.contoso.com",
         "http://www.microsoft.com"
      };

      // Used to time download operations.
      Stopwatch stopwatch = new Stopwatch();

      // Compute the time required to download the URLs.
      stopwatch.Start();
      var downloads = from url in urls
                      select DownloadStringAsync(url);
      Task.WhenAll(downloads).ContinueWith(results =>
      {
         stopwatch.Stop();

         // Print the number of characters download and the elapsed time.
         Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
            results.Result.Sum(result => result.Length),
            stopwatch.ElapsedMilliseconds);
      })
      .Wait();

      // Perform the same operation a second time. The time required
      // should be shorter because the results are held in the cache.
      stopwatch.Restart();
      downloads = from url in urls
                  select DownloadStringAsync(url);
      Task.WhenAll(downloads).ContinueWith(results =>
      {
         stopwatch.Stop();

         // Print the number of characters download and the elapsed time.
         Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
            results.Result.Sum(result => result.Length),
            stopwatch.ElapsedMilliseconds);
      })
      .Wait();
   }
}

/* Sample output:
Retrieved 27798 characters. Elapsed time was 1045 ms.
Retrieved 27798 characters. Elapsed time was 0 ms.
*/
```

```vb
Imports System.Collections.Concurrent
Imports System.Diagnostics
Imports System.Linq
Imports System.Net
Imports System.Threading.Tasks

' Demonstrates how to use Task<TResult>.FromResult to create a task
' that holds a pre-computed result.
Friend Class CachedDownloads
   ' Holds the results of download operations.
   Private Shared cachedDownloads As New ConcurrentDictionary(Of String, String)()

   ' Asynchronously downloads the requested resource as a string.
   Public Shared Function DownloadStringAsync(ByVal address As String) As Task(Of String)
      ' First try to retrieve the content from cache.
      Dim content As String
      If cachedDownloads.TryGetValue(address, content) Then
         Return Task.FromResult(Of String)(content)
      End If

      ' If the result was not in the cache, download the
      ' string and add it to the cache.
      Return Task.Run(async Function()
          content = await New WebClient().DownloadStringTaskAsync(address)
          cachedDownloads.TryAdd(address, content)
          Return content
      End Function)
   End Function

   Shared Sub Main(ByVal args() As String)
      ' The URLs to download.
      Dim urls() As String = { "http://msdn.microsoft.com", "http://www.contoso.com",
"http://www.microsoft.com" }

      ' Used to time download operations.
      Dim stopwatch As New Stopwatch()

      ' Compute the time required to download the URLs.
      stopwatch.Start()
      Dim downloads = From url In urls _
                      Select DownloadStringAsync(url)
      Task.WhenAll(downloads).ContinueWith(Sub(results)
         ' Print the number of characters download and the elapsed time.
          stopwatch.Stop()
          Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
results.Result.Sum(Function(result) result.Length), stopwatch.ElapsedMilliseconds)
      End Sub).Wait()

      ' Perform the same operation a second time. The time required
      ' should be shorter because the results are held in the cache.
      stopwatch.Restart()
      downloads = From url In urls _
                  Select DownloadStringAsync(url)
      Task.WhenAll(downloads).ContinueWith(Sub(results)
         ' Print the number of characters download and the elapsed time.
          stopwatch.Stop()
          Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
results.Result.Sum(Function(result) result.Length), stopwatch.ElapsedMilliseconds)
      End Sub).Wait()
   End Sub
End Class

' Sample output:
'Retrieved 27798 characters. Elapsed time was 1045 ms.
'Retrieved 27798 characters. Elapsed time was 0 ms.
'
```

This example computes the time that is required to download multiple strings two times. The second set of download operations should take less time than the first set because the results are held in the cache. The FromResult method enables the `DownloadStringAsync` method to create Task<TResult> objects that hold these pre-computed results.

See also

- Task-based Asynchronous Programming

How to: Traverse a Binary Tree with Parallel Tasks

9/6/2018 • 2 minutes to read • Edit Online

The following example shows two ways in which parallel tasks can be used to traverse a tree data structure. The creation of the tree itself is left as an exercise.

Example

```
public class TreeWalk
{
    static void Main()
    {
        Tree<MyClass> tree = new Tree<MyClass>();

        // ...populate tree (left as an exercise)

        // Define the Action to perform on each node.
        Action<MyClass> myAction = x => Console.WriteLine("{0} : {1}", x.Name, x.Number);

        // Traverse the tree with parallel tasks.
        DoTree(tree, myAction);
    }

    public class MyClass
    {
        public string Name { get; set; }
        public int Number { get; set; }
    }
    public class Tree<T>
    {
        public Tree<T> Left;
        public Tree<T> Right;
        public T Data;
    }



    // By using tasks explcitly.
    public static void DoTree<T>(Tree<T> tree, Action<T> action)
    {
        if (tree == null) return;
        var left = Task.Factory.StartNew(() => DoTree(tree.Left, action));
        var right = Task.Factory.StartNew(() => DoTree(tree.Right, action));
        action(tree.Data);

        try
        {
            Task.WaitAll(left, right);
        }
        catch (AggregateException )
        {
            //handle exceptions here
        }
    }

    // By using Parallel.Invoke
    public static void DoTree2<T>(Tree<T> tree, Action<T> action)
    {
        if (tree == null) return;
        Parallel.Invoke(
            () => DoTree2(tree.Left, action),
            () => DoTree2(tree.Right, action),
            () => action(tree.Data)
        );
    }

}
```

```
Imports System.Threading.Tasks

Public Class TreeWalk

    Shared Sub Main()

        Dim tree As Tree(Of Person) = New Tree(Of Person)()

        ' ...populate tree (left as an exercise)

        ' Define the Action to perform on each node.
        Dim myAction As Action(Of Person) = New Action(Of Person)(Sub(x)
                                                Console.WriteLine("{0}  : {1} ", x.Name, x.Number)
                                            End Sub)

        ' Traverse the tree with parallel tasks.
        DoTree(tree, myAction)
    End Sub

    Public Class Person
         Public Name As String
        Public Number As Integer
    End Class

    Public Class Tree(Of T)
        Public Left As Tree(Of T)
        Public Right As Tree(Of T)
        Public Data As T
    End Class

    ' By using tasks explicitly.
    Public Shared Sub DoTree(Of T)(ByVal myTree As Tree(Of T), ByVal a As Action(Of T))
        If myTree Is Nothing Then
            Return
        End If
        Dim left = Task.Factory.StartNew(Sub() DoTree(myTree.Left, a))
        Dim right = Task.Factory.StartNew(Sub() DoTree(myTree.Right, a))
        a(myTree.Data)

        Try
            Task.WaitAll(left, right)
        Catch ae As AggregateException
            'handle exceptions here
        End Try
    End Sub

    ' By using Parallel.Invoke
    Public Shared Sub DoTree2(Of T)(ByVal myTree As Tree(Of T), ByVal myAct As Action(Of T))
        If myTree Is Nothing Then
            Return
        End If
        Parallel.Invoke(
            Sub() DoTree2(myTree.Left, myAct),
            Sub() DoTree2(myTree.Right, myAct),
            Sub() myAct(myTree.Data)
        )
    End Sub
End Class
```

The two methods shown are functionally equivalent. By using the StartNew method to create and run the tasks, you get a handle back from the tasks which can be used to wait on the tasks and handle exceptions.

See also

- Task Parallel Library (TPL)

How to: Unwrap a Nested Task

1/23/2019 • 4 minutes to read • Edit Online

You can return a task from a method, and then wait on or continue from that task, as shown in the following example:

```
static Task<string> DoWorkAsync()
{
    return Task<String>.Factory.StartNew(() =>
    {
       //...
        return "Work completed.";
    });
}

static void StartTask()
{
    Task<String> t = DoWorkAsync();
    t.Wait();
    Console.WriteLine(t.Result);
}
```

```
 Shared Function DoWorkAsync() As Task(Of String)

   Return Task(Of String).Run(Function()
                                  '...
                                  Return "Work completed."
                               End Function)
End Function

Shared Sub StartTask()

   Dim t As Task(Of String) = DoWorkAsync()
   t.Wait()
   Console.WriteLine(t.Result)
End Sub
```

In the previous example, the Result property is of type `string` (`String` in Visual Basic).

However, in some scenarios, you might want to create a task within another task, and then return the nested task. In this case, the `TResult` of the enclosing task is itself a task. In the following example, the Result property is a `Task<Task<string>>` in C# or `Task(Of Task(Of String))` in Visual Basic.

```
// Note the type of t and t2.
Task<Task<string>> t = Task.Factory.StartNew(() => DoWorkAsync());
Task<Task<string>> t2 = DoWorkAsync().ContinueWith((s) => DoMoreWorkAsync());

// Outputs: System.Threading.Tasks.Task`1[System.String]
Console.WriteLine(t.Result);
```

```
' Note the type of t and t2.
Dim t As Task(Of Task(Of String)) = Task.Run(Function() DoWorkAsync())
Dim t2 As Task(Of Task(Of String)) = DoWorkAsync().ContinueWith(Function(s) DoMoreWorkAsync())

' Outputs: System.Threading.Tasks.Task`1[System.String]
Console.WriteLine(t.Result)
```

Although it is possible to write code to unwrap the outer task and retrieve the original task and its Result property, such code is not easy to write because you must handle exceptions and also cancellation requests. In this situation, we recommend that you use one of the Unwrap extension methods, as shown in the following example.

```
// Unwrap the inner task.
Task<string> t3 = DoWorkAsync().ContinueWith((s) => DoMoreWorkAsync()).Unwrap();

// Outputs "More work completed."
Console.WriteLine(t.Result);
```

```
' Unwrap the inner task.
Dim t3 As Task(Of String) = DoWorkAsync().ContinueWith(Function(s) DoMoreWorkAsync()).Unwrap()

' Outputs "More work completed."
Console.WriteLine(t.Result)
```

The Unwrap methods can be used to transform any `Task<Task>` or `Task<Task<TResult>>` (`Task(Of Task)` or `Task(Of Task(Of TResult))` in Visual Basic) to a `Task` or `Task<TResult>` (`Task(Of TResult)` in Visual Basic). The new task fully represents the inner nested task, and includes cancellation state and all exceptions.

Example

The following example demonstrates how to use the Unwrap extension methods.

```
namespace Unwrap
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    // A program whose only use is to demonstrate Unwrap.
    class Program
    {
        static void Main()
        {
            // An arbitrary threshold value.
            byte threshold = 0x40;

            // data is a Task<byte[]>
            var data = Task<byte[]>.Factory.StartNew(() =>
                {
                    return GetData();
                });

            // We want to return a task so that we can
            // continue from it later in the program.
            // Without Unwrap: stepTwo is a Task<Task<byte[]>>
            // With Unwrap: stepTwo is a Task<byte[]>
            var stepTwo = data.ContinueWith((antecedent) =>
                {
                    return Task<byte>.Factory.StartNew( () => Compute(antecedent.Result));
                })
                .Unwrap();

            // Without Unwrap: antecedent.Result = Task<byte>
            // and the following method will not compile.
            // With Unwrap: antecedent.Result = byte and
            // we can work directly with the result of the Compute method.
            var lastStep = stepTwo.ContinueWith( (antecedant) =>
                {
                    if (antecedant.Result >= threshold)
                    {
                      return Task.Factory.StartNew( () =>  Console.WriteLine("Program complete. Final =
0x{0:x} threshold = 0x{1:x}", stepTwo.Result, threshold));
                    }
                    else
                    {
                        return DoSomeOtherAsyncronousWork(stepTwo.Result, threshold);
                    }
                });

            lastStep.Wait();
            Console.WriteLine("Press any key");
            Console.ReadKey();
        }

        #region Dummy_Methods
        private static byte[] GetData()
        {
            Random rand = new Random();
            byte[] bytes = new byte[64];
            rand.NextBytes(bytes);
            return bytes;
        }

        static Task DoSomeOtherAsyncronousWork(int i, byte b2)
        {
            return Task.Factory.StartNew(() =>
                {
                    Thread.SpinWait(500000);
                    Console.WriteLine("Doing more work. Value was <= threshold");
                });
        }
        static byte Compute(byte[] data)
        {

            byte final = 0;
            foreach (byte item in data)
            {
                final ^= item;
                Console.WriteLine("{0:x}", final);
            }
            Console.WriteLine("Done computing");
            return final;
        }
        #endregion
    }
}
```

```
'How to: Unwrap a Task
Imports System.Threading
Imports System.Threading.Tasks

Module UnwrapATask2

    Sub Main()
```

```
        ' An arbitrary threshold value.
        Dim threshold As Byte = &H40

        ' myData is a Task(Of Byte())

        Dim myData As Task(Of Byte()) = Task.Factory.StartNew(Function()
                                                                  Return GetData()
                                                              End Function)
        ' We want to return a task so that we can
        ' continue from it later in the program.
        ' Without Unwrap: stepTwo is a Task(Of Task(Of Byte))
        ' With Unwrap: stepTwo is a Task(Of Byte)

        Dim stepTwo = myData.ContinueWith(Function(antecedent)
                                              Return Task.Factory.StartNew(Function()
                                                                               Return
Compute(antecedent.Result)
                                                                           End Function)
                                          End Function).Unwrap()

        Dim lastStep = stepTwo.ContinueWith(Function(antecedent)
                                                Console.WriteLine("Result = {0}", antecedent.Result)
                                                If antecedent.Result >= threshold Then
                                                    Return Task.Factory.StartNew(Sub()

Console.WriteLine("Program complete. Final = &H{1:x} threshold = &H{1:x}",

stepTwo.Result, threshold)
                                                                                 End Sub)
                                                Else
                                                    Return DoSomeOtherAsynchronousWork(stepTwo.Result,
threshold)
                                                End If
                                            End Function)
        Try
            lastStep.Wait()
        Catch ae As AggregateException
            For Each ex As Exception In ae.InnerExceptions
                Console.WriteLine(ex.Message & ex.StackTrace & ex.GetBaseException.ToString())
            Next
        End Try

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

#Region "Dummy_Methods"
    Function GetData() As Byte()
        Dim rand As Random = New Random()
        Dim bytes(64) As Byte
        rand.NextBytes(bytes)
        Return bytes
    End Function

    Function DoSomeOtherAsynchronousWork(ByVal i As Integer, ByVal b2 As Byte) As Task
        Return Task.Factory.StartNew(Sub()
                                         Thread.SpinWait(500000)
                                         Console.WriteLine("Doing more work. Value was <= threshold.")
                                     End Sub)
    End Function

    Function Compute(ByVal d As Byte()) As Byte
        Dim final As Byte = 0
        For Each item As Byte In d
            final = final Xor item
            Console.WriteLine("{0:x}", final)
        Next
        Console.WriteLine("Done computing")
        Return final
    End Function
#End Region
End Module
```

See also

- System.Threading.Tasks.TaskExtensions
- Task-based Asynchronous Programming

How to: Prevent a Child Task from Attaching to its Parent

5/14/2019 • 5 minutes to read • Edit Online

This document demonstrates how to prevent a child task from attaching to the parent task. Preventing a child task from attaching to its parent is useful when you call a component that is written by a third party and that also uses tasks. For example, a third-party component that uses the TaskCreationOptions.AttachedToParent option to create a Task or Task<TResult> object can cause problems in your code if it is long-running or throws an unhandled exception.

Example

The following example compares the effects of using the default options to the effects of preventing a child task from attaching to the parent. The example creates a Task object that calls into a third-party library that also uses a Task object. The third-party library uses the AttachedToParent option to create the Task object. The application uses the TaskCreationOptions.DenyChildAttach option to create the parent task. This option instructs the runtime to remove the AttachedToParent specification in child tasks.

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

// Defines functionality that is provided by a third-party.
// In a real-world scenario, this would likely be provided
// in a separate code file or assembly.
namespace Contoso
{
   public class Widget
   {
      public Task Run()
      {
         // Create a long-running task that is attached to the
         // parent in the task hierarchy.
         return Task.Factory.StartNew(() =>
         {
            // Simulate a lengthy operation.
            Thread.Sleep(5000);

         }, TaskCreationOptions.AttachedToParent);
      }
   }
}

// Demonstrates how to prevent a child task from attaching to the parent.
class DenyChildAttach
{
   static void RunWidget(Contoso.Widget widget,
      TaskCreationOptions parentTaskOptions)
   {
      // Record the time required to run the parent
      // and child tasks.
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      Console.WriteLine("Starting widget as a background task...");
      // Run the widget task in the background.
      Task<Task> runWidget = Task.Factory.StartNew(() =>
         {
            Task widgetTask = widget.Run();

            // Perform other work while the task runs...
            Thread.Sleep(1000);

            return widgetTask;
         }, parentTaskOptions);

      // Wait for the parent task to finish.
      Console.WriteLine("Waiting for parent task to finish...");
      runWidget.Wait();
      Console.WriteLine("Parent task has finished. Elapsed time is {0} ms.",
         stopwatch.ElapsedMilliseconds);

      // Perform more work...
      Console.WriteLine("Performing more work on the main thread...");
      Thread.Sleep(2000);
      Console.WriteLine("Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds);

      // Wait for the child task to finish.
      Console.WriteLine("Waiting for child task to finish...");
      runWidget.Result.Wait();
      Console.WriteLine("Child task has finished. Elapsed time is {0} ms.",
        stopwatch.ElapsedMilliseconds);
   }

   static void Main(string[] args)
   {
      Contoso.Widget w = new Contoso.Widget();

      // Perform the same operation two times. The first time, the operation
      // is performed by using the default task creation options. The second
      // time, the operation is performed by using the DenyChildAttach option
      // in the parent task.

      Console.WriteLine("Demonstrating parent/child tasks with default options...");
      RunWidget(w, TaskCreationOptions.None);

      Console.WriteLine();

      Console.WriteLine("Demonstrating parent/child tasks with the DenyChildAttach option...");
      RunWidget(w, TaskCreationOptions.DenyChildAttach);
   }
}

/* Sample output:
Demonstrating parent/child tasks with default options...
Starting widget as a background task...
Waiting for parent task to finish...
Parent task has finished. Elapsed time is 5014 ms.
Performing more work on the main thread...
Elapsed time is 7019 ms.
Waiting for child task to finish...
Child task has finished. Elapsed time is 7019 ms.

Demonstrating parent/child tasks with the DenyChildAttach option...
Starting widget as a background task...
Waiting for parent task to finish...
Parent task has finished. Elapsed time is 1007 ms.
Performing more work on the main thread...
Elapsed time is 3015 ms.
Waiting for child task to finish...
Child task has finished. Elapsed time is 5015 ms.
*/
```

```vb
Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks

' Defines functionality that is provided by a third-party.
' In a real-world scenario, this would likely be provided
' in a separate code file or assembly.
Namespace Contoso
   Public Class Widget
      Public Function Run() As Task
         ' Create a long-running task that is attached to the
         ' parent in the task hierarchy.
         Return Task.Factory.StartNew(Sub() Thread.Sleep(5000), TaskCreationOptions.AttachedToParent)
            ' Simulate a lengthy operation.
      End Function
   End Class
End Namespace

' Demonstrates how to prevent a child task from attaching to the parent.
Friend Class DenyChildAttach
   Private Shared Sub RunWidget(ByVal widget As Contoso.Widget, ByVal parentTaskOptions As
TaskCreationOptions)
      ' Record the time required to run the parent
      ' and child tasks.
      Dim stopwatch As New Stopwatch()
      stopwatch.Start()

      Console.WriteLine("Starting widget as a background task...")

      ' Run the widget task in the background.
      Dim runWidget As Task(Of Task) = Task.Factory.StartNew(Function()
            ' Perform other work while the task runs...
          Dim widgetTask As Task = widget.Run()
          Thread.Sleep(1000)
          Return widgetTask
      End Function, parentTaskOptions)

      ' Wait for the parent task to finish.
      Console.WriteLine("Waiting for parent task to finish...")
      runWidget.Wait()
      Console.WriteLine("Parent task has finished. Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds)

      ' Perform more work...
      Console.WriteLine("Performing more work on the main thread...")
      Thread.Sleep(2000)
      Console.WriteLine("Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds)

      ' Wait for the child task to finish.
      Console.WriteLine("Waiting for child task to finish...")
      runWidget.Result.Wait()
      Console.WriteLine("Child task has finished. Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds)
   End Sub

   Shared Sub Main(ByVal args() As String)
      Dim w As New Contoso.Widget()

      ' Perform the same operation two times. The first time, the operation
      ' is performed by using the default task creation options. The second
      ' time, the operation is performed by using the DenyChildAttach option
      ' in the parent task.

      Console.WriteLine("Demonstrating parent/child tasks with default options...")
      RunWidget(w, TaskCreationOptions.None)

      Console.WriteLine()

      Console.WriteLine("Demonstrating parent/child tasks with the DenyChildAttach option...")
      RunWidget(w, TaskCreationOptions.DenyChildAttach)
```

```
    End Sub
End Class

' Sample output:
'Demonstrating parent/child tasks with default options...
'Starting widget as a background task...
'Waiting for parent task to finish...
'Parent task has finished. Elapsed time is 5014 ms.
'Performing more work on the main thread...
'Elapsed time is 7019 ms.
'Waiting for child task to finish...
'Child task has finished. Elapsed time is 7019 ms.
'
'Demonstrating parent/child tasks with the DenyChildAttach option...
'Starting widget as a background task...
'Waiting for parent task to finish...
'Parent task has finished. Elapsed time is 1007 ms.
'Performing more work on the main thread...
'Elapsed time is 3015 ms.
'Waiting for child task to finish...
'Child task has finished. Elapsed time is 5015 ms.
'
```

Because a parent task does not finish until all child tasks finish, a long-running child task can cause the overall application to perform poorly. In this example, when the application uses the default options to create the parent task, the child task must finish before the parent task finishes. When the application uses the TaskCreationOptions.DenyChildAttach option, the child is not attached to the parent. Therefore, the application can perform additional work after the parent task finishes and before it must wait for the child task to finish.

See also

- Task-based Asynchronous Programming

Dataflow (Task Parallel Library)

8/22/2019 • 36 minutes to read • Edit Online

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.

> **NOTE**
> The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

This document contains the following sections:

- Programming Model
- Predefined Dataflow Block Types
- Configuring Dataflow Block Behavior
- Custom Dataflow Blocks

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<TOutput> interface to represent sources, System.Threading.Tasks.Dataflow.ITargetBlock<TInput> to represent targets, and System.Threading.Tasks.Dataflow.IPropagatorBlock<TInput,TOutput> to represent propagators. IPropagatorBlock<TInput,TOutput> inherits from both ISourceBlock<TOutput>, and ITargetBlock<TInput>.

The TPL Dataflow Library provides several predefined dataflow block types that implement the ISourceBlock<TOutput>, ITargetBlock<TInput>, and IPropagatorBlock<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<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<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<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 and DataflowBlock.SendAsync methods to send messages to target dataflow blocks post (Post acts synchronously; SendAsync acts asynchronously) and the Receive, ReceiveAsync, and TryReceive 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<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 call the ISourceBlock<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<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<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<TOutput> and ITargetBlock<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<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.

```
// Create an ActionBlock<int> object that prints its input
// and throws ArgumentOutOfRangeException if the input
// is less than zero.
var throwIfNegative = new ActionBlock<int>(n =>
{
   Console.WriteLine("n = {0}", n);
   if (n < 0)
   {
      throw new ArgumentOutOfRangeException();
   }
});

// 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 (AggregateException ae)
{
   // If an unhandled exception occurs during dataflow processing, all
   // exceptions are propagated through an AggregateException object.
   ae.Handle(e =>
   {
      Console.WriteLine("Encountered {0}: {1}",
         e.GetType().Name, e.Message);
      return true;
   });
}

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

```
         ' 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 section.

The second way to determine the completion status of a dataflow block is to use a continuation 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 continuation task that prints the status of the overall dataflow operation.

```csharp
// Create an ActionBlock<int> object that prints its input
// and throws ArgumentOutOfRangeException if the input
// is less than zero.
var throwIfNegative = new ActionBlock<int>(n =>
{
   Console.WriteLine("n = {0}", n);
   if (n < 0)
   {
      throw new ArgumentOutOfRangeException();
   }
});

// Create a continuation task that prints the overall
// task status to the console when the block finishes.
throwIfNegative.Completion.ContinueWith(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 (AggregateException ae)
{
   // If an unhandled exception occurs during dataflow processing, all
   // exceptions are propagated through an AggregateException object.
   ae.Handle(e =>
   {
      Console.WriteLine("Encountered {0}: {1}",
         e.GetType().Name, e.Message);
      return true;
   });
}

/* 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.
*/
```

```
         ' 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, and Exception Handling.

[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<T>, System.Threading.Tasks.Dataflow.BroadcastBlock<T>, and System.Threading.Tasks.Dataflow.WriteOnceBlock<T>.

BufferBlock(T)

The BufferBlock<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<T> object, that message is removed from the message queue. Therefore, although a BufferBlock<T> object can have multiple targets, only one target will receive each message. The BufferBlock<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<T> object and then reads those values back from that object.

```
// Create a BufferBlock<int> object.
var bufferBlock = new BufferBlock<int>();

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   bufferBlock.Post(i);
}

// Receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(bufferBlock.Receive());
}

/* Output:
   0
   1
   2
 */
```

```
         ' 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<T> object, see How to: Write Messages to and Read Messages from a Dataflow Block.

BroadcastBlock(T)

The BroadcastBlock<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<T> object and then reads that value back from that object several times. Because values are not removed from BroadcastBlock<T> objects after they are read, the same value is available every time.

```
// Create a BroadcastBlock<double> object.
var broadcastBlock = new BroadcastBlock<double>(null);

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

// Receive the messages back from the block several times.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(broadcastBlock.Receive());
}

/* Output:
   3.14159265358979
   3.14159265358979
   3.14159265358979
 */
```

```
         ' 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<T> to broadcast a message to multiple target blocks, see How to: Specify a Task Scheduler in a Dataflow Block.

WriteOnceBlock(T)

The WriteOnceBlock<T> class resembles the BroadcastBlock<T> class, except that a WriteOnceBlock<T> object can be written to one time only. You can think of WriteOnceBlock<T> as being similar to the C# readonly (ReadOnly in Visual Basic) keyword, except that a WriteOnceBlock<T> object becomes immutable after it receives a value instead of at construction. Like the BroadcastBlock<T> class, when a target receives a message from a WriteOnceBlock<T> object, that message is not removed from that object. Therefore, multiple targets receive a copy of the message. The WriteOnceBlock<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<T> object and then reads the value back from that object. Because a WriteOnceBlock<T> object can be written to one time only, after a WriteOnceBlock<T> object receives a message, it discards subsequent messages.

```
// Create a WriteOnceBlock<string> object.
var writeOnceBlock = new WriteOnceBlock<string>(null);

// 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(
   () => writeOnceBlock.Post("Message 1"),
   () => writeOnceBlock.Post("Message 2"),
   () => writeOnceBlock.Post("Message 3"));

// Receive the message from the block.
Console.WriteLine(writeOnceBlock.Receive());

/* Sample output:
   Message 2
 */
```

```
         ' 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<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<TInput>, System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput>, and System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput>.

ActionBlock(T)

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

The following basic example posts multiple Int32 values to an ActionBlock<TInput> object. The ActionBlock<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.

```
// Create an ActionBlock<int> object that prints values
// to the console.
var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   actionBlock.Post(i * 10);
}

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

/* Output:
   0
   10
   20
 */
```

```
         ' 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<TInput> class, see How to: Perform Action When a Dataflow Block Receives Data.

TransformBlock(TInput, TOutput)

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

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

```
// Create a TransformBlock<int, double> object that
// computes the square root of its input.
var transformBlock = new TransformBlock<int, double>(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 (int i = 0; i < 3; i++)
{
   Console.WriteLine(transformBlock.Receive());
}

/* Output:
   3.16227766016838
   4.47213595499958
   5.47722557505166
 */
```

```
         ' 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<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<TInput,TOutput> class resembles the TransformBlock<TInput,TOutput> class, except that TransformManyBlock<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<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<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<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<TInput,TOutput> object that splits strings into their individual character sequences. The TransformManyBlock<TInput,TOutput> object takes String values as input and produces Char values as output.

```csharp
// Create a TransformManyBlock<string, char> object that splits
// a string into its individual characters.
var transformManyBlock = new TransformManyBlock<string, char>(
   s => s.ToCharArray());

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

// Receive all output values from the block.
for (int i = 0; i < ("Hello" + "World").Length; i++)
{
   Console.WriteLine(transformManyBlock.Receive());
}

/* Output:
   H
   e
   l
   l
   o
   W
   o
   r
   l
   d
 */
```

```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<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<TInput>, TransformBlock<TInput,TOutput>, and TransformManyBlock<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<TInput>, TransformBlock<TInput,TOutput> and TransformManyBlock<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<TInput>, TransformBlock<TInput,TOutput>, and TransformManyBlock<TInput,TOutput> objects. This table also specifies whether the delegate type operates synchronously or asynchronously.

TYPE	SYNCHRONOUS DELEGATE TYPE	ASYNCHRONOUS DELEGATE TYPE
ActionBlock<TInput>	`System.Action`	`System.Func<TInput, Task>`
TransformBlock<TInput,TOutput>	`System.Func<TInput, TOutput>`	`System.Func<TInput, Task<TOutput>>`
TransformManyBlock<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<T>, JoinBlock<T1,T2>, and BatchedJoinBlock<T1,T2>.

BatchBlock(T)

The BatchBlock<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<T> object. When the BatchBlock<T> object receives the specified count of input elements, it asynchronously propagates out an array that contains those elements. If a BatchBlock<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<T> class operates in either *greedy* or *non-greedy* mode. In greedy mode, which is the default, a BatchBlock<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<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<T> constructor.

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

```
// Create a BatchBlock<int> object that holds ten
// elements per batch.
var batchBlock = new BatchBlock<int>(10);

// Post several values to the block.
for (int i = 0; i < 13; i++)
{
   batchBlock.Post(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.
 */
```

```
         ' 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<T> to improve the efficiency of database insert operations, see Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency.

JoinBlock(T1, T2, ...)

The JoinBlock<T1,T2> and JoinBlock<T1,T2,T3> classes collect input elements and propagate out System.Tuple<T1,T2> or System.Tuple<T1,T2,T3> objects that contain those elements. The JoinBlock<T1,T2> and JoinBlock<T1,T2,T3> classes do not inherit from ITargetBlock<TInput>. Instead, they provide properties, Target1, Target2, and Target3, that implement ITargetBlock<TInput>.

Like BatchBlock<T>, JoinBlock<T1,T2> and JoinBlock<T1,T2,T3> operate in either greedy or non-greedy mode. In greedy mode, which is the default, a JoinBlock<T1,T2> or JoinBlock<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<T1,T2> or JoinBlock<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<T1,T2,T3> object requires multiple data to compute a value. This example creates a JoinBlock<T1,T2,T3> object that requires two Int32 values and a Char value to perform an arithmetic operation.

```
// Create a JoinBlock<int, int, char> object that requires
// two numbers and an operator.
var joinBlock = new JoinBlock<int, int, 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('+');
joinBlock.Target3.Post('-');

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

for (int i = 0; i < 2; i++)
{
   var data = joinBlock.Receive();
   switch (data.Item3)
   {
      case '+':
         Console.WriteLine("{0} + {1} = {2}",
            data.Item1, data.Item2, data.Item1 + data.Item2);
         break;
      case '-':
         Console.WriteLine("{0} - {1} = {2}",
            data.Item1, data.Item2, data.Item1 - data.Item2);
         break;
      default:
         Console.WriteLine("Unknown operator '{0}'.", data.Item3);
         break;
   }
}

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

```
        ' 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<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<T1,T2> and BatchedJoinBlock<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<T1,T2> as a combination of BatchBlock<T> and JoinBlock<T1,T2>. Specify the size of each batch when you create a BatchedJoinBlock<T1,T2> object. BatchedJoinBlock<T1,T2> also provides properties, Target1 and Target2, that implement ITargetBlock<TInput>. When the specified count of input elements are received from across all targets, the BatchedJoinBlock<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<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<T1,T2> object. Because the count of successful and failed operations is unknown in advance, the IList<T> objects enable each target to receive zero or more values.

```csharp
// For demonstration, create a Func<int, int> that
// returns its argument, or throws ArgumentOutOfRangeException
// if the argument is less than zero.
Func<int, int> DoWork = n =>
{
   if (n < 0)
      throw new ArgumentOutOfRangeException();
   return n;
};

// Create a BatchedJoinBlock<int, Exception> object that holds
// seven elements per batch.
var batchedJoinBlock = new BatchedJoinBlock<int, Exception>(7);

// Post several items to the block.
foreach (int i in new int[] { 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 (ArgumentOutOfRangeException e)
   {
      // If an error occurred, post the Exception to the
      // second target of the block.
      batchedJoinBlock.Target2.Post(e);
   }
}

// Read the results from the block.
var results = batchedJoinBlock.Receive();

// Print the results to the console.

// Print the results.
foreach (int n in results.Item1)
{
   Console.WriteLine(n);
}
// Print failures.
foreach (Exception e in results.Item2)
{
   Console.WriteLine(e.Message);
}

/* 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 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<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<T>	DataflowBlockOptions
BroadcastBlock<T>	DataflowBlockOptions
WriteOnceBlock<T>	DataflowBlockOptions
ActionBlock<TInput>	ExecutionDataflowBlockOptions
TransformBlock<TInput,TOutput>	ExecutionDataflowBlockOptions
TransformManyBlock<TInput,TOutput>	ExecutionDataflowBlockOptions
BatchBlock<T>	GroupingDataflowBlockOptions
JoinBlock<T1,T2>	GroupingDataflowBlockOptions
BatchedJoinBlock<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<TInput>, TransformBlock<TInput,TOutput>, and TransformManyBlock<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 is, however, the same one in which they are received.

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<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<T> class, greedy and non-greedy behavior is similar, except that under non-greedy mode, a BatchBlock<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<TOutput> or ITargetBlock<TInput> interfaces directly or use the Encapsulate 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<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<TInput>, TransformBlock<TInput,TOutput>, and TransformManyBlock<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<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.

TITLE	DESCRIPTION
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<T> class to improve the efficiency of database insert operations, and how to use the BatchedJoinBlock<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.

How to: Write Messages to and Read Messages from a Dataflow Block

8/30/2019 • 6 minutes to read • Edit Online

This document describes how to use the TPL Dataflow Library to write messages to and read messages from a dataflow block. The TPL Dataflow Library provides both synchronous and asynchronous methods for writing messages to and reading messages from a dataflow block. This document uses the System.Threading.Tasks.Dataflow.BufferBlock<T> class. The BufferBlock<T> class buffers messages and behaves as both a message source and as a message target.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Writing to and Reading from a Dataflow Block Synchronously

The following example uses the Post method to write to a BufferBlock<T> dataflow block and the Receive method to read from the same object.

```
// Create a BufferBlock<int> object.
var bufferBlock = new BufferBlock<int>();

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   bufferBlock.Post(i);
}

// Receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(bufferBlock.Receive());
}

/* Output:
   0
   1
   2
 */
```

```
      ' 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
'
```

You can also use the TryReceive method to read from a dataflow block, as shown in the following example. The TryReceive method does not block the current thread and is useful when you occasionally poll for data.

```
// Post more messages to the block.
for (int i = 0; i < 3; i++)
{
   bufferBlock.Post(i);
}

// Receive the messages back from the block.
int value;
while (bufferBlock.TryReceive(out value))
{
   Console.WriteLine(value);
}

/* Output:
   0
   1
   2
 */
```

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

      ' Receive the messages back from the block.
      Dim value As Integer
      Do While bufferBlock.TryReceive(value)
         Console.WriteLine(value)
      Loop

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

Because the Post method acts synchronously, the BufferBlock<T> object in the previous examples receives all data before the second loop reads data. The following example extends the first example by using Invoke to read from and write to the message block concurrently. Because Invoke performs actions concurrently, the values are not written to the BufferBlock<T> object in any specific order.

```
// Write to and read from the message block concurrently.
var post01 = Task.Run(() =>
   {
      bufferBlock.Post(0);
      bufferBlock.Post(1);
   });
var receive = Task.Run(() =>
   {
      for (int i = 0; i < 3; i++)
      {
         Console.WriteLine(bufferBlock.Receive());
      }
   });
var post2 = Task.Run(() =>
   {
      bufferBlock.Post(2);
   });
Task.WaitAll(post01, receive, post2);

/* Sample output:
   2
   0
   1
 */
```

```
      ' Write to and read from the message block concurrently.
      Dim post01 = Task.Run(Sub()
          bufferBlock.Post(0)
          bufferBlock.Post(1)
      End Sub)
      Dim receive = Task.Run(Sub()
            For i As Integer = 0 To 2
               Console.WriteLine(bufferBlock.Receive())
            Next i
      End Sub)
      Dim post2 = Task.Run(Sub() bufferBlock.Post(2))
      Task.WaitAll(post01, receive, post2)

'       Sample output:
'         2
'         0
'         1
'
```

Writing to and Reading from a Dataflow Block Asynchronously

The following example uses the SendAsync method to asynchronously write to a BufferBlock<T> object and the ReceiveAsync method to asynchronously read from the same object. This example uses the async and await operators (Async and Await in Visual Basic) to asynchronously send data to and read data from the target block. The SendAsync method is useful when you must enable a dataflow block to postpone messages. The ReceiveAsync method is useful when you want to act on data when that data becomes available. For more information about how messages propagate among message blocks, see the section Message Passing in Dataflow.

```
// Post more messages to the block asynchronously.
for (int i = 0; i < 3; i++)
{
   await bufferBlock.SendAsync(i);
}

// Asynchronously receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(await bufferBlock.ReceiveAsync());
}

/* Output:
   0
   1
   2
 */
```

```
      ' Post more messages to the block asynchronously.
      For i As Integer = 0 To 2
         await bufferBlock.SendAsync(i)
      Next i

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

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

A Complete Example

The following example shows the complete code for this document.

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates a how to write to and read from a dataflow block.
class DataflowReadWrite
{
   // Demonstrates asynchronous dataflow operations.
   static async Task AsyncSendReceive(BufferBlock<int> bufferBlock)
   {
      // Post more messages to the block asynchronously.
      for (int i = 0; i < 3; i++)
      {
         await bufferBlock.SendAsync(i);
      }

      // Asynchronously receive the messages back from the block.
      for (int i = 0; i < 3; i++)
      {
         Console.WriteLine(await bufferBlock.ReceiveAsync());
      }

      /* Output:
         0
         1
         2
        */
   }

   static void Main(string[] args)
   {
      // Create a BufferBlock<int> object.
      var bufferBlock = new BufferBlock<int>();

      // Post several messages to the block.
      for (int i = 0; i < 3; i++)
      {
         bufferBlock.Post(i);
      }

      // Receive the messages back from the block.
      for (int i = 0; i < 3; i++)
      {
         Console.WriteLine(bufferBlock.Receive());
      }

      /* Output:
         0
         1
         2
       */

      // Post more messages to the block.
      for (int i = 0; i < 3; i++)
      {
         bufferBlock.Post(i);
      }

      // Receive the messages back from the block.
      int value;
      while (bufferBlock.TryReceive(out value))
      {
         Console.WriteLine(value);
      }

      /* Output:
         0
         1
         2
       */

      // Write to and read from the message block concurrently.
      var post01 = Task.Run(() =>
         {
            bufferBlock.Post(0);
            bufferBlock.Post(1);
         });
      var receive = Task.Run(() =>
         {
            for (int i = 0; i < 3; i++)
            {
               Console.WriteLine(bufferBlock.Receive());
            }
         });
      var post2 = Task.Run(() =>
         {
            bufferBlock.Post(2);
         });
      Task.WaitAll(post01, receive, post2);

      /* Sample output:
         2
         0
         1
       */

      // Demonstrate asynchronous dataflow operations.
      AsyncSendReceive(bufferBlock).Wait();
   }
}
```

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

' Demonstrates a how to write to and read from a dataflow block.
Friend Class DataflowReadWrite
   ' Demonstrates asynchronous dataflow operations.
   Private Shared async Function AsyncSendReceive(ByVal bufferBlock As BufferBlock(Of Integer)) As Task
      ' Post more messages to the block asynchronously.
      For i As Integer = 0 To 2
         await bufferBlock.SendAsync(i)
      Next i

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

'       Output:
'         0
'         1
'         2
'
   End Function

   Shared Sub Main(ByVal args() As String)
      ' 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
'

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

      ' Receive the messages back from the block.
      Dim value As Integer
      Do While bufferBlock.TryReceive(value)
         Console.WriteLine(value)
      Loop

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

```
        ' Write to and read from the message block concurrently.
        Dim post01 = Task.Run(Sub()
            bufferBlock.Post(0)
            bufferBlock.Post(1)
        End Sub)
        Dim receive = Task.Run(Sub()
              For i As Integer = 0 To 2
                 Console.WriteLine(bufferBlock.Receive())
              Next i
        End Sub)
        Dim post2 = Task.Run(Sub() bufferBlock.Post(2))
        Task.WaitAll(post01, receive, post2)

'          Sample output:
'            2
'            0
'            1
'


        ' Demonstrate asynchronous dataflow operations.
        AsyncSendReceive(bufferBlock).Wait()
    End Sub

End Class
```

Next Steps

This example shows how to read from and write to a message block directly. You can also connect dataflow blocks to form *pipelines*, which are linear sequences of dataflow blocks, or *networks*, which are graphs of dataflow blocks. In a pipeline or network, sources asynchronously propagate data to targets as that data becomes available. For an example that creates a basic dataflow pipeline, see Walkthrough: Creating a Dataflow Pipeline. For an example that creates a more complex dataflow network, see Walkthrough: Using Dataflow in a Windows Forms Application.

See also

- Dataflow

How to: Implement a Producer-Consumer Dataflow Pattern

8/30/2019 • 5 minutes to read • Edit Online

This document describes how to use the TPL Dataflow Library to implement a producer-consumer pattern. In this pattern, the *producer* sends messages to a message block, and the *consumer* reads messages from that block.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Example

The following example demonstrates a basic producer- consumer model that uses dataflow. The `Produce` method writes arrays that contain random bytes of data to a System.Threading.Tasks.Dataflow.ITargetBlock<TInput> object and the `Consume` method reads bytes from a System.Threading.Tasks.Dataflow.ISourceBlock<TOutput> object. By acting on the ISourceBlock<TOutput> and ITargetBlock<TInput> interfaces, instead of their derived types, you can write reusable code that can act on a variety of dataflow block types. This example uses the BufferBlock<T> class. Because the BufferBlock<T> class acts as both a source block and as a target block, the producer and the consumer can use a shared object to transfer data.

The `Produce` method calls the Post method in a loop to synchronously write data to the target block. After the `Produce` method writes all data to the target block, it calls the Complete method to indicate that the block will never have additional data available. The `Consume` method uses the async and await operators (Async and Await in Visual Basic) to asynchronously compute the total number of bytes that are received from the ISourceBlock<TOutput> object. To act asynchronously, the `Consume` method calls the OutputAvailableAsync method to receive a notification when the source block has data available and when the source block will never have additional data available.

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates a basic producer and consumer pattern that uses dataflow.
class DataflowProducerConsumer
{
   // Demonstrates the production end of the producer and consumer pattern.
   static void Produce(ITargetBlock<byte[]> target)
   {
      // Create a Random object to generate random data.
      Random rand = new Random();

      // In a loop, fill a buffer with random data and
      // post the buffer to the target block.
      for (int i = 0; i < 100; i++)
      {
         // Create an array to hold random byte data.
         byte[] buffer = new byte[1024];

         // Fill the buffer with random bytes.
         rand.NextBytes(buffer);

         // Post the result to the message block.
         target.Post(buffer);
      }

      // Set the target to the completed state to signal to the consumer
      // that no more data will be available.
      target.Complete();
   }

   // Demonstrates the consumption end of the producer and consumer pattern.
   static async Task<int> ConsumeAsync(ISourceBlock<byte[]> source)
   {
      // Initialize a counter to track the number of bytes that are processed.
      int bytesProcessed = 0;

      // Read from the source buffer until the source buffer has no
      // available output data.
      while (await source.OutputAvailableAsync())
      {
         byte[] data = source.Receive();

         // Increment the count of bytes received.
         bytesProcessed += data.Length;
      }

      return bytesProcessed;
   }

   static void Main(string[] args)
   {
      // Create a BufferBlock<byte[]> object. This object serves as the
      // target block for the producer and the source block for the consumer.
      var buffer = new BufferBlock<byte[]>();

      // Start the consumer. The Consume method runs asynchronously.
      var consumer = ConsumeAsync(buffer);

      // Post source data to the dataflow block.
      Produce(buffer);

      // Wait for the consumer to process all data.
      consumer.Wait();

      // Print the count of bytes processed to the console.
      Console.WriteLine("Processed {0} bytes.", consumer.Result);
   }
}

/* Output:
Processed 102400 bytes.
*/
```

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

' Demonstrates a basic producer and consumer pattern that uses dataflow.
Friend Class DataflowProducerConsumer
   ' Demonstrates the production end of the producer and consumer pattern.
   Private Shared Sub Produce(ByVal target As ITargetBlock(Of Byte()))
      ' Create a Random object to generate random data.
      Dim rand As New Random()

      ' In a loop, fill a buffer with random data and
      ' post the buffer to the target block.
      For i As Integer = 0 To 99
```

```
        ' Create an array to hold random byte data.
        Dim buffer(1023) As Byte

        ' Fill the buffer with random bytes.
        rand.NextBytes(buffer)

        ' Post the result to the message block.
        target.Post(buffer)
      Next i

      ' Set the target to the completed state to signal to the consumer
      ' that no more data will be available.
      target.Complete()
    End Sub

    ' Demonstrates the consumption end of the producer and consumer pattern.
    Private Shared async Function ConsumeAsync(ByVal source As ISourceBlock(Of Byte())) As Task(Of Integer)
      ' Initialize a counter to track the number of bytes that are processed.
      Dim bytesProcessed As Integer = 0

      ' Read from the source buffer until the source buffer has no
      ' available output data.
      Do While await source.OutputAvailableAsync()
        Dim data() As Byte = source.Receive()

        ' Increment the count of bytes received.
        bytesProcessed += data.Length
      Loop

      Return bytesProcessed
    End Function

    Shared Sub Main(ByVal args() As String)
      ' Create a BufferBlock<byte[]> object. This object serves as the
      ' target block for the producer and the source block for the consumer.
      Dim buffer = New BufferBlock(Of Byte())()

      ' Start the consumer. The Consume method runs asynchronously.
      Dim consumer = ConsumeAsync(buffer)

      ' Post source data to the dataflow block.
      Produce(buffer)

      ' Wait for the consumer to process all data.
      consumer.Wait()

      ' Print the count of bytes processed to the console.
      Console.WriteLine("Processed {0} bytes.", consumer.Result)
    End Sub
  End Class

  ' Output:
  'Processed 102400 bytes.
  '
```

Robust Programming

The preceding example uses just one consumer to process the source data. If you have multiple consumers in your application, use the TryReceive method to read data from the source block, as shown in the following example.

```csharp
// Demonstrates the consumption end of the producer and consumer pattern.
static async Task<int> ConsumeAsync(IReceivableSourceBlock<byte[]> source)
{
   // Initialize a counter to track the number of bytes that are processed.
   int bytesProcessed = 0;

   // Read from the source buffer until the source buffer has no
   // available output data.
   while (await source.OutputAvailableAsync())
   {
      byte[] data;
      while (source.TryReceive(out data))
      {
         // Increment the count of bytes received.
         bytesProcessed += data.Length;
      }
   }

   return bytesProcessed;
}
```

```vb
' Demonstrates the consumption end of the producer and consumer pattern.
Private Shared async Function ConsumeAsync(ByVal source As IReceivableSourceBlock(Of Byte())) As Task(Of
Integer)
   ' Initialize a counter to track the number of bytes that are processed.
   Dim bytesProcessed As Integer = 0

   ' Read from the source buffer until the source buffer has no
   ' available output data.
   Do While await source.OutputAvailableAsync()
      Dim data() As Byte
      Do While source.TryReceive(data)
         ' Increment the count of bytes received.
         bytesProcessed += data.Length
      Loop
   Loop

   Return bytesProcessed
End Function
```

The TryReceive method returns `False` when no data is available. When multiple consumers must access the source block concurrently, this mechanism guarantees that data is still available after the call to OutputAvailableAsync.

See also

- Dataflow

How to: Perform Action When a Dataflow Block Receives Data

8/30/2019 • 6 minutes to read • Edit Online

Execution dataflow block types call a user-provided delegate when they receive data. The System.Threading.Tasks.Dataflow.ActionBlock<TInput>, System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput>, and System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput> classes are execution dataflow block types. You can use the `delegate` keyword (`Sub` in Visual Basic), Action<T>, Func<T,TResult>, or a lambda expression when you provide a work function to an execution dataflow block. This document describes how to use Func<T,TResult> and lambda expressions to perform action in execution blocks.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Example

The following example uses dataflow to read a file from disk and computes the number of bytes in that file that are equal to zero. It uses TransformBlock<TInput,TOutput> to read the file and compute the number of zero bytes, and ActionBlock<TInput> to print the number of zero bytes to the console. The TransformBlock<TInput,TOutput> object specifies a Func<T,TResult> object to perform work when the blocks receive data. The ActionBlock<TInput> object uses a lambda expression to print to the console the number of zero bytes that are read.

```
using System;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to provide delegates to exectution dataflow blocks.
class DataflowExecutionBlocks
{
   // Computes the number of zero bytes that the provided file
   // contains.
   static int CountBytes(string path)
   {
      byte[] buffer = new byte[1024];
      int totalZeroBytesRead = 0;
      using (var fileStream = File.OpenRead(path))
      {
         int bytesRead = 0;
         do
         {
            bytesRead = fileStream.Read(buffer, 0, buffer.Length);
            totalZeroBytesRead += buffer.Count(b => b == 0);
         } while (bytesRead > 0);
      }

      return totalZeroBytesRead;
      return totalZeroBytesRead;
   }

   static void Main(string[] args)
   {
      // Create a temporary file on disk.
      string tempFile = Path.GetTempFileName();

      // Write random data to the temporary file.
      using (var fileStream = File.OpenWrite(tempFile))
      {
         Random rand = new Random();
         byte[] buffer = new byte[1024];
         for (int i = 0; i < 512; i++)
         {
            rand.NextBytes(buffer);
            fileStream.Write(buffer, 0, buffer.Length);
         }
      }

      // Create an ActionBlock<int> object that prints to the console
      // the number of bytes read.
      var printResult = new ActionBlock<int>(zeroBytesRead =>
      {
         Console.WriteLine("{0} contains {1} zero bytes.",
            Path.GetFileName(tempFile), zeroBytesRead);
      });

      // Create a TransformBlock<string, int> object that calls the
      // CountBytes function and returns its result.
      var countBytes = new TransformBlock<string, int>(
         new Func<string, int>(CountBytes));

      // Link the TransformBlock<string, int> object to the
      // ActionBlock<int> object.
      countBytes.LinkTo(printResult);

      // Create a continuation task that completes the ActionBlock<int>
      // object when the TransformBlock<string, int> finishes.
      countBytes.Completion.ContinueWith(delegate { printResult.Complete(); });

      // Post the path to the temporary file to the
      // TransformBlock<string, int> object.
      countBytes.Post(tempFile);

      // Requests completion of the TransformBlock<string, int> object.
      countBytes.Complete();

      // Wait for the ActionBlock<int> object to print the message.
      printResult.Completion.Wait();

      // Delete the temporary file.
      File.Delete(tempFile);
   }
}

/* Sample output:
tmp4FBE.tmp contains 2081 zero bytes.
*/
```

```
Imports System.IO
Imports System.Linq
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to provide delegates to exectution dataflow blocks.
Friend Class DataflowExecutionBlocks
   ' Computes the number of zero bytes that the provided file
   ' contains.
   Private Shared Function CountBytes(ByVal path As String) As Integer
      Dim buffer(1023) As Byte
      Dim totalZeroBytesRead As Integer = 0
      Using fileStream = File.OpenRead(path)
         Dim bytesRead As Integer = 0
         Do
            bytesRead = fileStream.Read(buffer, 0, buffer.Length)
            totalZeroBytesRead += buffer.Count(Function(b) b = 0)
         Loop While bytesRead > 0
      End Using

      Return totalZeroBytesRead
   End Function

   Shared Sub Main(ByVal args() As String)
      ' Create a temporary file on disk.
      Dim tempFile As String = Path.GetTempFileName()

      ' Write random data to the temporary file.
      Using fileStream = File.OpenWrite(tempFile)
         Dim rand As New Random()
         Dim buffer(1023) As Byte
         For i As Integer = 0 To 511
            rand.NextBytes(buffer)
            fileStream.Write(buffer, 0, buffer.Length)
         Next i
      End Using

      ' Create an ActionBlock<int> object that prints to the console
      ' the number of bytes read.
      Dim printResult = New ActionBlock(Of Integer)(Sub(zeroBytesRead) Console.WriteLine("{0} contains {1}
zero bytes.", Path.GetFileName(tempFile), zeroBytesRead))

      ' Create a TransformBlock<string, int> object that calls the
      ' CountBytes function and returns its result.
      Dim countBytes = New TransformBlock(Of String, Integer)(New Func(Of String, Integer)(AddressOf
DataflowExecutionBlocks.CountBytes))

      ' Link the TransformBlock<string, int> object to the
      ' ActionBlock<int> object.
      countBytes.LinkTo(printResult)

      ' Create a continuation task that completes the ActionBlock<int>
      ' object when the TransformBlock<string, int> finishes.
      countBytes.Completion.ContinueWith(Sub() printResult.Complete())

      ' Post the path to the temporary file to the
      ' TransformBlock<string, int> object.
      countBytes.Post(tempFile)

      ' Requests completion of the TransformBlock<string, int> object.
      countBytes.Complete()

      ' Wait for the ActionBlock<int> object to print the message.
      printResult.Completion.Wait()

      ' Delete the temporary file.
      File.Delete(tempFile)
   End Sub
End Class

' Sample output:
'tmp4FBE.tmp contains 2081 zero bytes.
'
```

Although you can provide a lambda expression to a TransformBlock<TInput,TOutput> object, this example uses

Func<T,TResult> to enable other code to use the `CountBytes` method. The ActionBlock<TInput> object uses a lambda expression because the work to be performed is specific to this task and is not likely to be useful from other code. For more information about how lambda expressions work in the Task Parallel Library, see Lambda Expressions in PLINQ and TPL.

The section Summary of Delegate Types in the Dataflow document summarizes the delegate types that you can provide to ActionBlock<TInput>, TransformBlock<TInput,TOutput>, and TransformManyBlock<TInput,TOutput> objects. The table also specifies whether the delegate type operates synchronously or asynchronously.

Robust Programming

This example provides a delegate of type Func<T,TResult> to the TransformBlock<TInput,TOutput> object to perform the task of the dataflow block synchronously. To enable the dataflow block to behave asynchronously, provide a delegate of type Func<TResult> to the dataflow block. When a dataflow block behaves asynchronously, the task of the dataflow block is complete only when the returned Task<TResult> object finishes. The following example modifies the `CountBytes` method and uses the async and await operators (Async and Await in Visual Basic) to asynchronously compute the total number of bytes that are zero in the provided file. The ReadAsync method performs file read operations asynchronously.

```
// Asynchronously computes the number of zero bytes that the provided file
// contains.
static async Task<int> CountBytesAsync(string path)
{
   byte[] buffer = new byte[1024];
   int totalZeroBytesRead = 0;
   using (var fileStream = new FileStream(
      path, FileMode.Open, FileAccess.Read, FileShare.Read, 0x1000, true))
   {
      int bytesRead = 0;
      do
      {
         // Asynchronously read from the file stream.
         bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length);
         totalZeroBytesRead += buffer.Count(b => b == 0);
      } while (bytesRead > 0);
   }

   return totalZeroBytesRead;
}
```

```
' Asynchronously computes the number of zero bytes that the provided file
' contains.
Private Shared async Function CountBytesAsync(ByVal path As String) As Task(Of Integer)
   Dim buffer(1023) As Byte
   Dim totalZeroBytesRead As Integer = 0
   Using fileStream = New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, &H1000, True)
      Dim bytesRead As Integer = 0
      Do
         ' Asynchronously read from the file stream.
         bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)
         totalZeroBytesRead += buffer.Count(Function(b) b = 0)
      Loop While bytesRead > 0
   End Using

   Return totalZeroBytesRead
End Function
```

You can also use asynchronous lambda expressions to perform action in an execution dataflow block. The following example modifies the TransformBlock<TInput,TOutput> object that is used in the previous example so that it uses a lambda expression to perform the work asynchronously.

```
// Create a TransformBlock<string, int> object that calls the
// CountBytes function and returns its result.
var countBytesAsync = new TransformBlock<string, int>(async path =>
{
   byte[] buffer = new byte[1024];
   int totalZeroBytesRead = 0;
   using (var fileStream = new FileStream(
      path, FileMode.Open, FileAccess.Read, FileShare.Read, 0x1000, true))
   {
      int bytesRead = 0;
      do
      {
         // Asynchronously read from the file stream.
         bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length);
         totalZeroBytesRead += buffer.Count(b => b == 0);
      } while (bytesRead > 0);
   }

   return totalZeroBytesRead;
});
```

```
' Create a TransformBlock<string, int> object that calls the
' CountBytes function and returns its result.
Dim countBytesAsync = New TransformBlock(Of String, Integer)(async Function(path)
         ' Asynchronously read from the file stream.
    Dim buffer(1023) As Byte
    Dim totalZeroBytesRead As Integer = 0
    Using fileStream = New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, &H1000, True)
        Dim bytesRead As Integer = 0
        Do
            bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)
            totalZeroBytesRead += buffer.Count(Function(b) b = 0)
        Loop While bytesRead > 0
    End Using
    Return totalZeroBytesRead
End Function)
```

See also

- Dataflow

Walkthrough: Creating a Dataflow Pipeline

4/9/2019 • 12 minutes to read • Edit Online

Although you can use the DataflowBlock.Receive, DataflowBlock.ReceiveAsync, and DataflowBlock.TryReceive methods to receive messages from source blocks, you can also connect message blocks to form a *dataflow pipeline*. A dataflow pipeline is a series of components, or *dataflow blocks*, each of which performs a specific task that contributes to a larger goal. Every dataflow block in a dataflow pipeline performs work when it receives a message from another dataflow block. An analogy to this is an assembly line for automobile manufacturing. As each vehicle passes through the assembly line, one station assembles the frame, the next one installs the engine, and so on. Because an assembly line enables multiple vehicles to be assembled at the same time, it provides better throughput than assembling complete vehicles one at a time.

This document demonstrates a dataflow pipeline that downloads the book *The Iliad of Homer* from a website and searches the text to match individual words with words that reverse the first word's characters. The formation of the dataflow pipeline in this document consists of the following steps:

1. Create the dataflow blocks that participate in the pipeline.
2. Connect each dataflow block to the next block in the pipeline. Each block receives as input the output of the previous block in the pipeline.
3. For each dataflow block, create a continuation task that sets the next block to the completed state after the previous block finishes.
4. Post data to the head of the pipeline.
5. Mark the head of the pipeline as completed.
6. Wait for the pipeline to complete all work.

Prerequisites

Read Dataflow before you start this walkthrough.

Creating a Console Application

In Visual Studio, create a Visual C# or Visual Basic Console Application project. Install the System.Threading.Tasks.Dataflow NuGet package.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Add the following code to your project to create the basic application.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a basic dataflow pipeline.
// This program downloads the book "The Iliad of Homer" by Homer from the Web
// and finds all reversed words that appear in that book.
static class Program
{
   static void Main()
   {
   }
}
```

```
Imports System.Net.Http
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a basic dataflow pipeline.
' This program downloads the book "The Iliad of Homer" by Homer from the Web
' and finds all reversed words that appear in that book.
Module DataflowReversedWords

   Sub Main()
   End Sub

End Module
```

Creating the Dataflow Blocks

Add the following code to the `Main` method to create the dataflow blocks that participate in the pipeline. The table that follows summarizes the role of each member of the pipeline.

```csharp
//
// Create the members of the pipeline.
//

// Downloads the requested resource as a string.
var downloadString = new TransformBlock<string, string>(async uri =>
{
   Console.WriteLine("Downloading '{0}'...", uri);

   return await new HttpClient().GetStringAsync(uri);
});

// Separates the specified text into an array of words.
var createWordList = new TransformBlock<string, string[]>(text =>
{
   Console.WriteLine("Creating word list...");

   // Remove common punctuation by replacing all non-letter characters
   // with a space character.
   char[] tokens = text.Select(c => char.IsLetter(c) ? c : ' ').ToArray();
   text = new string(tokens);

   // Separate the text into an array of words.
   return text.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
});

// Removes short words and duplicates.
var filterWordList = new TransformBlock<string[], string[]>(words =>
{
   Console.WriteLine("Filtering word list...");

   return words
      .Where(word => word.Length > 3)
      .Distinct()
      .ToArray();
});

// Finds all words in the specified collection whose reverse also
// exists in the collection.
var findReversedWords = new TransformManyBlock<string[], string>(words =>
{
   Console.WriteLine("Finding reversed words...");

   var wordsSet = new HashSet<string>(words);

   return from word in words.AsParallel()
          let reverse = new string(word.Reverse().ToArray())
          where word != reverse && wordsSet.Contains(reverse)
          select word;
});

// Prints the provided reversed words to the console.
var printReversedWords = new ActionBlock<string>(reversedWord =>
{
   Console.WriteLine("Found reversed words {0}/{1}",
      reversedWord, new string(reversedWord.Reverse().ToArray()));
});
```

```vb
'
' Create the members of the pipeline.
'

' Downloads the requested resource as a string.
Dim downloadString = New TransformBlock(Of String, String)(
    Async Function(uri)
       Console.WriteLine("Downloading '{0}'...", uri)

       Return Await New HttpClient().GetStringAsync(uri)
    End Function)

' Separates the specified text into an array of words.
Dim createWordList = New TransformBlock(Of String, String())(
   Function(text)
      Console.WriteLine("Creating word list...")

      ' Remove common punctuation by replacing all non-letter characters
      ' with a space character.
      Dim tokens() As Char = text.Select(Function(c) If(Char.IsLetter(c), c, " "c)).ToArray()
      text = New String(tokens)

      ' Separate the text into an array of words.
      Return text.Split(New Char() {" "c}, StringSplitOptions.RemoveEmptyEntries)
   End Function)

' Removes short words and duplicates.
Dim filterWordList = New TransformBlock(Of String(), String())(
   Function(words)
      Console.WriteLine("Filtering word list...")

      Return words.Where(Function(word) word.Length > 3).Distinct().ToArray()
   End Function)

' Finds all words in the specified collection whose reverse also
' exists in the collection.
Dim findReversedWords = New TransformManyBlock(Of String(), String)(
   Function(words)

      Dim wordsSet = New HashSet(Of String)(words)

      Return From word In words.AsParallel()
             Let reverse = New String(word.Reverse().ToArray())
             Where word <> reverse AndAlso wordsSet.Contains(reverse)
             Select word
   End Function)

' Prints the provided reversed words to the console.
Dim printReversedWords = New ActionBlock(Of String)(
   Sub(reversedWord)
      Console.WriteLine("Found reversed words {0}/{1}", reversedWord, New
String(reversedWord.Reverse().ToArray()))
   End Sub)
```

MEMBER	TYPE	DESCRIPTION
`downloadString`	TransformBlock<TInput,TOutput>	Downloads the book text from the Web.
`createWordList`	TransformBlock<TInput,TOutput>	Separates the book text into an array of words.

MEMBER	TYPE	DESCRIPTION
`filterWordList`	TransformBlock<TInput,TOutput>	Removes short words and duplicates from the word array.
`findReversedWords`	TransformManyBlock<TInput,TOutput>	Finds all words in the filtered word array collection whose reverse also occurs in the word array.
`printReversedWords`	ActionBlock<TInput>	Displays words and the corresponding reverse words to the console.

Although you could combine multiple steps in the dataflow pipeline in this example into one step, the example illustrates the concept of composing multiple independent dataflow tasks to perform a larger task. The example uses TransformBlock<TInput,TOutput> to enable each member of the pipeline to perform an operation on its input data and send the results to the next step in the pipeline. The `findReversedWords` member of the pipeline is a TransformManyBlock<TInput,TOutput> object because it produces multiple independent outputs for each input. The tail of the pipeline, `printReversedWords`, is an ActionBlock<TInput> object because it performs an action on its input, and does not produce a result.

Forming the Pipeline

Add the following code to connect each block to the next block in the pipeline.

When you call the LinkTo method to connect a source dataflow block to a target dataflow block, the source dataflow block propagates data to the target block as data becomes available. If you also provide DataflowLinkOptions with PropagateCompletion set to true, successful or unsuccessful completion of one block in the pipeline will cause completion of the next block in the pipeline.

```
//
// Connect the dataflow blocks to form a pipeline.
//

var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };

downloadString.LinkTo(createWordList, linkOptions);
createWordList.LinkTo(filterWordList, linkOptions);
filterWordList.LinkTo(findReversedWords, linkOptions);
findReversedWords.LinkTo(printReversedWords, linkOptions);
```

```
'
' Connect the dataflow blocks to form a pipeline.
'

Dim linkOptions = New DataflowLinkOptions With { .PropagateCompletion = True }

downloadString.LinkTo(createWordList, linkOptions)
createWordList.LinkTo(filterWordList, linkOptions)
filterWordList.LinkTo(findReversedWords, linkOptions)
findReversedWords.LinkTo(printReversedWords, linkOptions)
```

Posting Data to the Pipeline

Add the following code to post the URL of the book *The Iliad of Homer* to the head of the dataflow pipeline.

```
// Process "The Iliad of Homer" by Homer.
downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt");
```

```
' Process "The Iliad of Homer" by Homer.
downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt")
```

This example uses DataflowBlock.Post to synchronously send data to the head of the pipeline. Use the DataflowBlock.SendAsync method when you must asynchronously send data to a dataflow node.

Completing Pipeline Activity

Add the following code to mark the head of the pipeline as completed. The head of the pipeline propagates its completion after it processes all buffered messages.

```
// Mark the head of the pipeline as complete.
downloadString.Complete();
```

```
' Mark the head of the pipeline as complete.
downloadString.Complete()
```

This example sends one URL through the dataflow pipeline to be processed. If you send more than one input through a pipeline, call the IDataflowBlock.Complete method after you submit all the input. You can omit this step if your application has no well-defined point at which data is no longer available or the application does not have to wait for the pipeline to finish.

Waiting for the Pipeline to Finish

Add the following code to wait for the pipeline to finish. The overall operation is finished when the tail of the pipeline finishes.

```
// Wait for the last block in the pipeline to process all messages.
printReversedWords.Completion.Wait();
```

```
' Wait for the last block in the pipeline to process all messages.
printReversedWords.Completion.Wait()
```

You can wait for dataflow completion from any thread or from multiple threads at the same time.

The Complete Example

The following example shows the complete code for this walkthrough.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a basic dataflow pipeline.
// This program downloads the book "The Iliad of Homer" by Homer from the Web
// and finds all reversed words that appear in that book.
static class DataflowReversedWords
```

```
{
   static void Main()
   {
      //
      // Create the members of the pipeline.
      //

      // Downloads the requested resource as a string.
      var downloadString = new TransformBlock<string, string>(async uri =>
      {
         Console.WriteLine("Downloading '{0}'...", uri);

         return await new HttpClient().GetStringAsync(uri);
      });

      // Separates the specified text into an array of words.
      var createWordList = new TransformBlock<string, string[]>(text =>
      {
         Console.WriteLine("Creating word list...");

         // Remove common punctuation by replacing all non-letter characters
         // with a space character.
         char[] tokens = text.Select(c => char.IsLetter(c) ? c : ' ').ToArray();
         text = new string(tokens);

         // Separate the text into an array of words.
         return text.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
      });

      // Removes short words and duplicates.
      var filterWordList = new TransformBlock<string[], string[]>(words =>
      {
         Console.WriteLine("Filtering word list...");

         return words
            .Where(word => word.Length > 3)
            .Distinct()
            .ToArray();
      });

      // Finds all words in the specified collection whose reverse also
      // exists in the collection.
      var findReversedWords = new TransformManyBlock<string[], string>(words =>
      {
         Console.WriteLine("Finding reversed words...");

         var wordsSet = new HashSet<string>(words);

         return from word in words.AsParallel()
                let reverse = new string(word.Reverse().ToArray())
                where word != reverse && wordsSet.Contains(reverse)
                select word;
      });

      // Prints the provided reversed words to the console.
      var printReversedWords = new ActionBlock<string>(reversedWord =>
      {
         Console.WriteLine("Found reversed words {0}/{1}",
            reversedWord, new string(reversedWord.Reverse().ToArray()));
      });

      //
      // Connect the dataflow blocks to form a pipeline.
      //

      var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };

      downloadString.LinkTo(createWordList, linkOptions);
      createWordList.LinkTo(filterWordList, linkOptions);
```

```
        filterWordList.LinkTo(findReversedWords, linkOptions);
        findReversedWords.LinkTo(printReversedWords, linkOptions);

        // Process "The Iliad of Homer" by Homer.
        downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt");

        // Mark the head of the pipeline as complete.
        downloadString.Complete();

        // Wait for the last block in the pipeline to process all messages.
        printReversedWords.Completion.Wait();
    }
}
/* Sample output:
   Downloading 'http://www.gutenberg.org/cache/epub/16452/pg16452.txt'...
   Creating word list...
   Filtering word list...
   Finding reversed words...
   Found reversed words doom/mood
   Found reversed words draw/ward
   Found reversed words aera/area
   Found reversed words seat/taes
   Found reversed words live/evil
   Found reversed words port/trop
   Found reversed words sleek/keels
   Found reversed words area/aera
   Found reversed words tops/spot
   Found reversed words evil/live
   Found reversed words mood/doom
   Found reversed words speed/deeps
   Found reversed words moor/room
   Found reversed words trop/port
   Found reversed words spot/tops
   Found reversed words spots/stops
   Found reversed words stops/spots
   Found reversed words reed/deer
   Found reversed words keels/sleek
   Found reversed words deeps/speed
   Found reversed words deer/reed
   Found reversed words taes/seat
   Found reversed words room/moor
   Found reversed words ward/draw
*/
```

```
Imports System.Net.Http
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a basic dataflow pipeline.
' This program downloads the book "The Iliad of Homer" by Homer from the Web
' and finds all reversed words that appear in that book.
Module DataflowReversedWords

   Sub Main()
      '
      ' Create the members of the pipeline.
      '

      ' Downloads the requested resource as a string.
      Dim downloadString = New TransformBlock(Of String, String)(
          Async Function(uri)
             Console.WriteLine("Downloading '{0}'...", uri)

             Return Await New HttpClient().GetStringAsync(uri)
          End Function)

      ' Separates the specified text into an array of words.
      Dim createWordList = New TransformBlock(Of String, String())(
         Function(text)
            Console.WriteLine("Creating word list...")

            ' Remove common punctuation by replacing all non-letter characters
            ' with a space character.
            Dim tokens() As Char = text.Select(Function(c) If(Char.IsLetter(c), c, " "c)).ToArray()
            text = New String(tokens)

            ' Separate the text into an array of words.
            Return text.Split(New Char() {" "c}, StringSplitOptions.RemoveEmptyEntries)
         End Function)

      ' Removes short words and duplicates.
      Dim filterWordList = New TransformBlock(Of String(), String())(
         Function(words)
            Console.WriteLine("Filtering word list...")

            Return words.Where(Function(word) word.Length > 3).Distinct().ToArray()
         End Function)

      ' Finds all words in the specified collection whose reverse also
      ' exists in the collection.
      Dim findReversedWords = New TransformManyBlock(Of String(), String)(
         Function(words)

            Dim wordsSet = New HashSet(Of String)(words)

            Return From word In words.AsParallel()
                   Let reverse = New String(word.Reverse().ToArray())
                   Where word <> reverse AndAlso wordsSet.Contains(reverse)
                   Select word
         End Function)

      ' Prints the provided reversed words to the console.
      Dim printReversedWords = New ActionBlock(Of String)(
         Sub(reversedWord)
            Console.WriteLine("Found reversed words {0}/{1}", reversedWord, New
String(reversedWord.Reverse().ToArray()))
         End Sub)

      '
      ' Connect the dataflow blocks to form a pipeline.
      '

      Dim linkOptions = New DataflowLinkOptions With { .PropagateCompletion = True }

      downloadString.LinkTo(createWordList, linkOptions)
      createWordList.LinkTo(filterWordList, linkOptions)
      filterWordList.LinkTo(findReversedWords, linkOptions)
      findReversedWords.LinkTo(printReversedWords, linkOptions)

      ' Process "The Iliad of Homer" by Homer.
      downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt")

      ' Mark the head of the pipeline as complete.
      downloadString.Complete()

      ' Wait for the last block in the pipeline to process all messages.
      printReversedWords.Completion.Wait()
   End Sub

End Module

' Sample output:
'Downloading 'http://www.gutenberg.org/cache/epub/16452/pg16452.txt'...
'Creating word list...
'Filtering word list...
'Finding reversed words...
'Found reversed words aera/area
```

```
'Found reversed words doom/mood
'Found reversed words draw/ward
'Found reversed words live/evil
'Found reversed words seat/taes
'Found reversed words area/aera
'Found reversed words port/trop
'Found reversed words sleek/keels
'Found reversed words tops/spot
'Found reversed words evil/live
'Found reversed words speed/deeps
'Found reversed words mood/doom
'Found reversed words moor/room
'Found reversed words spot/tops
'Found reversed words spots/stops
'Found reversed words trop/port
'Found reversed words stops/spots
'Found reversed words reed/deer
'Found reversed words deeps/speed
'Found reversed words deer/reed
'Found reversed words taes/seat
'Found reversed words keels/sleek
'Found reversed words room/moor
'Found reversed words ward/draw
```

Next Steps

This example sends one URL to process through the dataflow pipeline. If you send more than one input value through a pipeline, you can introduce a form of parallelism into your application that resembles how parts might move through an automobile factory. When the first member of the pipeline sends its result to the second member, it can process another item in parallel as the second member processes the first result.

The parallelism that is achieved by using dataflow pipelines is known as *coarse-grained parallelism* because it typically consists of fewer, larger tasks. You can also use a more *fine-grained parallelism* of smaller, short-running tasks in a dataflow pipeline. In this example, the `findReversedWords` member of the pipeline uses PLINQ to process multiple items in the work list in parallel. The use of fine-grained parallelism in a coarse-grained pipeline can improve overall throughput.

You can also connect a source dataflow block to multiple target blocks to create a *dataflow network*. The overloaded version of the LinkTo method takes a Predicate<T> object that defines whether the target block accepts each message based on its value. Most dataflow block types that act as sources offer messages to all connected target blocks, in the order in which they were connected, until one of the blocks accepts that message. By using this filtering mechanism, you can create systems of connected dataflow blocks that direct certain data through one path and other data through another path. For an example that uses filtering to create a dataflow network, see Walkthrough: Using Dataflow in a Windows Forms Application.

See also

- Dataflow

How to: Unlink Dataflow Blocks

5/14/2019 • 4 minutes to read • Edit Online

This document describes how to unlink a target dataflow block from its source.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Example

The following example creates three TransformBlock<TInput,TOutput> objects, each of which calls the `TrySolution` method to compute a value. This example requires only the result from the first call to `TrySolution` to finish.

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to unlink dataflow blocks.
class DataflowReceiveAny
{
   // Receives the value from the first provided source that has
   // a message.
   public static T ReceiveFromAny<T>(params ISourceBlock<T>[] sources)
   {
      // Create a WriteOnceBlock<T> object and link it to each source block.
      var writeOnceBlock = new WriteOnceBlock<T>(e => e);
      foreach (var source in sources)
      {
         // Setting MaxMessages to one instructs
         // the source block to unlink from the WriteOnceBlock<T> object
         // after offering the WriteOnceBlock<T> object one message.
         source.LinkTo(writeOnceBlock, new DataflowLinkOptions { MaxMessages = 1 });
      }
      // Return the first value that is offered to the WriteOnceBlock object.
      return writeOnceBlock.Receive();
   }

   // Demonstrates a function that takes several seconds to produce a result.
   static int TrySolution(int n, CancellationToken ct)
   {
      // Simulate a lengthy operation that completes within three seconds
      // or when the provided CancellationToken object is cancelled.
      SpinWait.SpinUntil(() => ct.IsCancellationRequested,
         new Random().Next(3000));

      // Return a value.
      return n + 42;
   }

   static void Main(string[] args)
   {
      // Create a shared CancellationTokenSource object to enable the
      // TrySolution method to be cancelled.
```

```
        var cts = new CancellationTokenSource();

        // Create three TransformBlock<int, int> objects.
        // Each TransformBlock<int, int> object calls the TrySolution method.
        Func<int, int> action = n => TrySolution(n, cts.Token);
        var trySolution1 = new TransformBlock<int, int>(action);
        var trySolution2 = new TransformBlock<int, int>(action);
        var trySolution3 = new TransformBlock<int, int>(action);

        // Post data to each TransformBlock<int, int> object.
        trySolution1.Post(11);
        trySolution2.Post(21);
        trySolution3.Post(31);

        // Call the ReceiveFromAny<T> method to receive the result from the
        // first TransformBlock<int, int> object to finish.
        int result = ReceiveFromAny(trySolution1, trySolution2, trySolution3);

        // Cancel all calls to TrySolution that are still active.
        cts.Cancel();

        // Print the result to the console.
        Console.WriteLine("The solution is {0}.", result);

        cts.Dispose();
    }
}

/* Sample output:
The solution is 53.
*/
```

```vb
Imports System.Threading
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to unlink dataflow blocks.
Friend Class DataflowReceiveAny
   ' Receives the value from the first provided source that has
   ' a message.
   Public Shared Function ReceiveFromAny(Of T)(ParamArray ByVal sources() As ISourceBlock(Of T)) As T
      ' Create a WriteOnceBlock<T> object and link it to each source block.
      Dim writeOnceBlock = New WriteOnceBlock(Of T)(Function(e) e)
      For Each source In sources
            ' Setting MaxMessages to one instructs
            ' the source block to unlink from the WriteOnceBlock<T> object
            ' after offering the WriteOnceBlock<T> object one message.
            source.LinkTo(writeOnceBlock, New DataflowLinkOptions With {.MaxMessages = 1})
      Next source
      ' Return the first value that is offered to the WriteOnceBlock object.
      Return writeOnceBlock.Receive()
   End Function

   ' Demonstrates a function that takes several seconds to produce a result.
   Private Shared Function TrySolution(ByVal n As Integer, ByVal ct As CancellationToken) As Integer
      ' Simulate a lengthy operation that completes within three seconds
      ' or when the provided CancellationToken object is cancelled.
      SpinWait.SpinUntil(Function() ct.IsCancellationRequested, New Random().Next(3000))

      ' Return a value.
      Return n + 42
   End Function

   Shared Sub Main(ByVal args() As String)
      ' Create a shared CancellationTokenSource object to enable the
      ' TrySolution method to be cancelled.
      Dim cts = New CancellationTokenSource()

      ' Create three TransformBlock<int, int> objects.
      ' Each TransformBlock<int, int> object calls the TrySolution method.
      Dim action As Func(Of Integer, Integer) = Function(n) TrySolution(n, cts.Token)
      Dim trySolution1 = New TransformBlock(Of Integer, Integer)(action)
      Dim trySolution2 = New TransformBlock(Of Integer, Integer)(action)
      Dim trySolution3 = New TransformBlock(Of Integer, Integer)(action)

      ' Post data to each TransformBlock<int, int> object.
      trySolution1.Post(11)
      trySolution2.Post(21)
      trySolution3.Post(31)

      ' Call the ReceiveFromAny<T> method to receive the result from the
      ' first TransformBlock<int, int> object to finish.
      Dim result As Integer = ReceiveFromAny(trySolution1, trySolution2, trySolution3)

      ' Cancel all calls to TrySolution that are still active.
      cts.Cancel()

      ' Print the result to the console.
      Console.WriteLine("The solution is {0}.", result)

      cts.Dispose()
   End Sub
End Class

' Sample output:
'The solution is 53.
'
```

To receive the value from the first TransformBlock<TInput,TOutput> object that finishes, this example defines the

`ReceiveFromAny(T)` method. The `ReceiveFromAny(T)` method accepts an array of ISourceBlock<TOutput> objects and links each of these objects to a WriteOnceBlock<T> object. When you use the LinkTo method to link a source dataflow block to a target block, the source propagates messages to the target as data becomes available. Because the WriteOnceBlock<T> class accepts only the first message that it is offered, the `ReceiveFromAny(T)` method produces its result by calling the Receive method. This produces the first message that is offered to the WriteOnceBlock<T> object. The LinkTo method has an overloaded version that takes an DataflowLinkOptions object with a MaxMessages property that, when it is set to `1`, instructs the source block to unlink from the target after the target receives one message from the source. It is important for the WriteOnceBlock<T> object to unlink from its sources because the relationship between the array of sources and the WriteOnceBlock<T> object is no longer required after the WriteOnceBlock<T> object receives a message.

To enable the remaining calls to `TrySolution` to end after one of them computes a value, the `TrySolution` method takes a CancellationToken object that is canceled after the call to `ReceiveFromAny(T)` returns. The SpinUntil method returns when this CancellationToken object is canceled.

See also

- Dataflow

Walkthrough: Using Dataflow in a Windows Forms Application

8/22/2019 • 16 minutes to read • Edit Online

This document demonstrates how to create a network of dataflow blocks that perform image processing in a Windows Forms application.

This example loads image files from the specified folder, creates a composite image, and displays the result. The example uses the dataflow model to route images through the network. In the dataflow model, independent components of a program communicate with one another by sending messages. When a component receives a message, it performs some action and then passes the result to another component. Compare this with the control flow model, in which an application uses control structures, for example, conditional statements, loops, and so on, to control the order of operations in a program.

Prerequisites

Read Dataflow before you start this walkthrough.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Sections

This walkthrough contains the following sections:

- Creating the Windows Forms Application
- Creating the Dataflow Network
- Connecting the Dataflow Network to the User Interface
- The Complete Example

Creating the Windows Forms Application

This section describes how to create the basic Windows Forms application and add controls to the main form.

To Create the Windows Forms Application

1. In Visual Studio, create a Visual C# or Visual Basic **Windows Forms Application** project. In this document, the project is named `CompositeImages`.

2. On the form designer for the main form, Form1.cs (Form1.vb for Visual Basic), add a ToolStrip control.

3. Add a ToolStripButton control to the ToolStrip control. Set the DisplayStyle property to Text and the Text property to **Choose Folder**.

4. Add a second ToolStripButton control to the ToolStrip control. Set the DisplayStyle property to Text, the

Text property to **Cancel**, and the Enabled property to `False`.

5. Add a PictureBox object to the main form. Set the Dock property to Fill.

Creating the Dataflow Network

This section describes how to create the dataflow network that performs image processing.

To Create the Dataflow Network

1. Add a reference to System.Threading.Tasks.Dataflow.dll to your project.

2. Ensure that Form1.cs (Form1.vb for Visual Basic) contains the following `using` (`Using` in Visual Basic) statements:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

3. Add the following data members to the `Form1` class:

```
// The head of the dataflow network.
ITargetBlock<string> headBlock = null;

// Enables the user interface to signal cancellation to the network.
CancellationTokenSource cancellationTokenSource;
```

4. Add the following method, `CreateImageProcessingNetwork`, to the `Form1` class. This method creates the image processing network.

```
// Creates the image processing dataflow network and returns the
// head node of the network.
ITargetBlock<string> CreateImageProcessingNetwork()
{
   //
   // Create the dataflow blocks that form the network.
   //

   // Create a dataflow block that takes a folder path as input
   // and returns a collection of Bitmap objects.
   var loadBitmaps = new TransformBlock<string, IEnumerable<Bitmap>>(path =>
      {
         try
         {
            return LoadBitmaps(path);
         }
         catch (OperationCanceledException)
         {
            // Handle cancellation by passing the empty collection
            // to the next stage of the network.
            return Enumerable.Empty<Bitmap>();
         }
      });

   // Create a dataflow block that takes a collection of Bitmap objects
   // and returns a single composite bitmap.
   // and returns a single composite bitmap.
   var createCompositeBitmap = new TransformBlock<IEnumerable<Bitmap>, Bitmap>(bitmaps =>
      {
         try
         {
            return CreateCompositeBitmap(bitmaps);
         }
         catch (OperationCanceledException)
         {
            // Handle cancellation by passing null to the next stage
            // of the network.
            return null;
         }
      });

   // Create a dataflow block that displays the provided bitmap on the form.
   var displayCompositeBitmap = new ActionBlock<Bitmap>(bitmap =>
      {
         // Display the bitmap.
         pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
         pictureBox1.Image = bitmap;

         // Enable the user to select another folder.
         toolStripButton1.Enabled = true;
         toolStripButton2.Enabled = false;
         Cursor = DefaultCursor;
      },
      // Specify a task scheduler from the current synchronization context
      // so that the action runs on the UI thread.
      new ExecutionDataflowBlockOptions
      {
          TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
      });

   // Create a dataflow block that responds to a cancellation request by
   // displaying an image to indicate that the operation is cancelled and
   // enables the user to select another folder.
   var operationCancelled = new ActionBlock<object>(delegate
      {
         // Display the error image to indicate that the operation
         // was cancelled.
         pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
         pictureBox1.Image = pictureBox1.ErrorImage;

         // Enable the user to select another folder.
         toolStripButton1.Enabled = true;
         toolStripButton2.Enabled = false;
         Cursor = DefaultCursor;
      },
      // Specify a task scheduler from the current synchronization context
      // so that the action runs on the UI thread.
      new ExecutionDataflowBlockOptions
      {
         TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
      });

   //
   // Connect the network.
   //

   // Link loadBitmaps to createCompositeBitmap.
   // The provided predicate ensures that createCompositeBitmap accepts the
   // collection of bitmaps only if that collection has at least one member.
   loadBitmaps.LinkTo(createCompositeBitmap, bitmaps => bitmaps.Count() > 0);

   // Also link loadBitmaps to operationCancelled.
   // When createCompositeBitmap rejects the message, loadBitmaps
   // offers the message to operationCancelled.
   // operationCancelled accepts all messages because we do not provide a
   // predicate.
```

```
   // predicate.
   loadBitmaps.LinkTo(operationCancelled);

   // Link createCompositeBitmap to displayCompositeBitmap.
   // The provided predicate ensures that displayCompositeBitmap accepts the
   // bitmap only if it is non-null.
   createCompositeBitmap.LinkTo(displayCompositeBitmap, bitmap => bitmap != null);

   // Also link createCompositeBitmap to operationCancelled.
   // When displayCompositeBitmap rejects the message, createCompositeBitmap
   // offers the message to operationCancelled.
   // operationCancelled accepts all messages because we do not provide a
   // predicate.
   createCompositeBitmap.LinkTo(operationCancelled);

   // Return the head of the network.
   return loadBitmaps;
}
```

5. Implement the `LoadBitmaps` method.

```
// Loads all bitmap files that exist at the provided path.
IEnumerable<Bitmap> LoadBitmaps(string path)
{
   List<Bitmap> bitmaps = new List<Bitmap>();

   // Load a variety of image types.
   foreach (string bitmapType in
      new string[] { "*.bmp", "*.gif", "*.jpg", "*.png", "*.tif" })
   {
      // Load each bitmap for the current extension.
      foreach (string fileName in Directory.GetFiles(path, bitmapType))
      {
         // Throw OperationCanceledException if cancellation is requested.
         cancellationTokenSource.Token.ThrowIfCancellationRequested();

         try
         {
            // Add the Bitmap object to the collection.
            bitmaps.Add(new Bitmap(fileName));
         }
         catch (Exception)
         {
            // TODO: A complete application might handle the error.
         }
      }
   }
   return bitmaps;
}
```

6. Implement the `CreateCompositeBitmap` method.

```
// Creates a composite bitmap from the provided collection of Bitmap objects.
// This method computes the average color of each pixel among all bitmaps
// to create the composite image.
Bitmap CreateCompositeBitmap(IEnumerable<Bitmap> bitmaps)
{
   Bitmap[] bitmapArray = bitmaps.ToArray();

   // Compute the maximum width and height components of all
   // bitmaps in the collection.
   Rectangle largest = new Rectangle();
   foreach (var bitmap in bitmapArray)
   {
      if (bitmap.Width > largest.Width)
         largest.Width = bitmap.Width;
         largest.Width = bitmap.Width;
      if (bitmap.Height > largest.Height)
         largest.Height = bitmap.Height;
   }

   // Create a 32-bit Bitmap object with the greatest dimensions.
   Bitmap result = new Bitmap(largest.Width, largest.Height,
      PixelFormat.Format32bppArgb);

   // Lock the result Bitmap.
   var resultBitmapData = result.LockBits(
      new Rectangle(new Point(), result.Size), ImageLockMode.WriteOnly,
      result.PixelFormat);

   // Lock each source bitmap to create a parallel list of BitmapData objects.
   var bitmapDataList = (from bitmap in bitmapArray
                         select bitmap.LockBits(
                           new Rectangle(new Point(), bitmap.Size),
                           ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb))
                        .ToList();

   // Compute each column in parallel.
   Parallel.For(0, largest.Width, new ParallelOptions
   {
      CancellationToken = cancellationTokenSource.Token
   },
   i =>
   {
      // Compute each row.
      for (int j = 0; j < largest.Height; j++)
      {
         // Counts the number of bitmaps whose dimensions
         // contain the current location.
         int count = 0;

         // The sum of all alpha, red, green, and blue components.
         int a = 0, r = 0, g = 0, b = 0;

         // For each bitmap, compute the sum of all color components.
         foreach (var bitmapData in bitmapDataList)
         {
            // Ensure that we stay within the bounds of the image.
            if (bitmapData.Width > i && bitmapData.Height > j)
            {
               unsafe
               {
                  byte* row = (byte*)(bitmapData.Scan0 + (j * bitmapData.Stride));
                  byte* pix = (byte*)(row + (4 * i));
                  a += *pix; pix++;
                  r += *pix; pix++;
                  g += *pix; pix++;
                  b += *pix;
               }
               count++;
            }
         }

         //prevent divide by zero in bottom right pixelless corner
         if (count == 0)
             break;

         unsafe
         {
            // Compute the average of each color component.
            a /= count;
            r /= count;
            g /= count;
            b /= count;

            // Set the result pixel.
```

```
            // Set the result pixel.
            byte* row = (byte*)(resultBitmapData.Scan0 + (j * resultBitmapData.Stride));
            byte* pix = (byte*)(row + (4 * i));
            *pix = (byte)a; pix++;
            *pix = (byte)r; pix++;
            *pix = (byte)g; pix++;
            *pix = (byte)b;
         }
      }
   });

   // Unlock the source bitmaps.
   for (int i = 0; i < bitmapArray.Length; i++)
   {
      bitmapArray[i].UnlockBits(bitmapDataList[i]);
   }

   // Unlock the result bitmap.
   result.UnlockBits(resultBitmapData);

   // Return the result.
   return result;
}
```

NOTE

The C# version of the `CreateCompositeBitmap` method uses pointers to enable efficient processing of the System.Drawing.Bitmap objects. Therefore, you must enable the **Allow unsafe code** option in your project in order to use the unsafe keyword. For more information about how to enable unsafe code in a Visual C# project, see Build Page, Project Designer (C#).

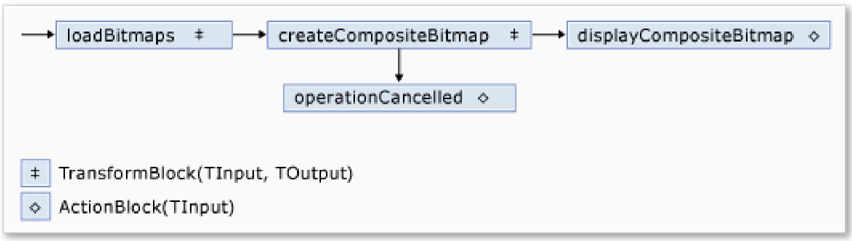
The following table describes the members of the network.

MEMBER	TYPE	DESCRIPTION
`loadBitmaps`	TransformBlock<TInput,TOutput>	Takes a folder path as input and produces a collection of Bitmap objects as output.
`createCompositeBitmap`	TransformBlock<TInput,TOutput>	Takes a collection of Bitmap objects as input and produces a composite bitmap as output.
`displayCompositeBitmap`	ActionBlock<TInput>	Displays the composite bitmap on the form.
`operationCancelled`	ActionBlock<TInput>	Displays an image to indicate that the operation is canceled and enables the user to select another folder.

To connect the dataflow blocks to form a network, this example uses the LinkTo method. The LinkTo method contains an overloaded version that takes a Predicate<T> object that determines whether the target block accepts or rejects a message. This filtering mechanism enables message blocks to receive only certain values. In this example, the network can branch in one of two ways. The main branch loads the images from disk, creates the composite image, and displays that image on the form. The alternate branch cancels the current operation. The Predicate<T> objects enable the dataflow blocks along the main branch to switch to the alternative branch by rejecting certain messages. For example, if the user cancels the operation, the dataflow block `createCompositeBitmap` produces `null` (`Nothing` in Visual Basic) as its output. The dataflow block `displayCompositeBitmap` rejects `null` input values, and therefore, the message is offered to `operationCancelled`.

The dataflow block `operationCancelled` accepts all messages and therefore, displays an image to indicate that the operation is canceled.

The following illustration shows the image processing network:



Because the `displayCompositeBitmap` and `operationCancelled` dataflow blocks act on the user interface, it is important that these actions occur on the user-interface thread. To accomplish this, during construction, these objects each provide a ExecutionDataflowBlockOptions object that has the TaskScheduler property set to TaskScheduler.FromCurrentSynchronizationContext. The TaskScheduler.FromCurrentSynchronizationContext method creates a TaskScheduler object that performs work on the current synchronization context. Because the `CreateImageProcessingNetwork` method is called from the handler of the **Choose Folder** button, which runs on the user-interface thread, the actions for the `displayCompositeBitmap` and `operationCancelled` dataflow blocks also run on the user-interface thread.

This example uses a shared cancellation token instead of setting the CancellationToken property because the CancellationToken property permanently cancels dataflow block execution. A cancellation token enables this example to reuse the same dataflow network multiple times, even when the user cancels one or more operations. For an example that uses CancellationToken to permanently cancel the execution of a dataflow block, see How to: Cancel a Dataflow Block.

Connecting the Dataflow Network to the User Interface

This section describes how to connect the dataflow network to the user interface. The creation of the composite image and cancellation of the operation are initiated from the **Choose Folder** and **Cancel** buttons. When the user chooses either of these buttons, the appropriate action is initiated in an asynchronous manner.

To Connect the Dataflow Network to the User Interface

1. On the form designer for the main form, create an event handler for the Click event for the **Choose Folder** button.

2. Implement the Click event for the **Choose Folder** button.

```
// Event handler for the Choose Folder button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // Create a FolderBrowserDialog object to enable the user to
   // select a folder.
   FolderBrowserDialog dlg = new FolderBrowserDialog
   {
      ShowNewFolderButton = false
   };

   // Set the selected path to the common Sample Pictures folder
   // if it exists.
   string initialDirectory = Path.Combine(
      Environment.GetFolderPath(Environment.SpecialFolder.CommonPictures),
      "Sample Pictures");
   if (Directory.Exists(initialDirectory))
   {
      dlg.SelectedPath = initialDirectory;
   }

   // Show the dialog and process the dataflow network.
   if (dlg.ShowDialog() == DialogResult.OK)
   {
      // Create a new CancellationTokenSource object to enable
      // cancellation.
      cancellationTokenSource = new CancellationTokenSource();

      // Create the image processing network if needed.
      headBlock ??= CreateImageProcessingNetwork();

      // Post the selected path to the network.
      headBlock.Post(dlg.SelectedPath);

      // Enable the Cancel button and disable the Choose Folder button.
      toolStripButton1.Enabled = false;
      toolStripButton2.Enabled = true;

      // Show a wait cursor.
      Cursor = Cursors.WaitCursor;
   }
}
```

3. On the form designer for the main form, create an event handler for the Click event for the **Cancel** button.

4. Implement the Click event for the **Cancel** button.

```
// Event handler for the Cancel button.
private void toolStripButton2_Click(object sender, EventArgs e)
{
   // Signal the request for cancellation. The current component of
   // the dataflow network will respond to the cancellation request.
   cancellationTokenSource.Cancel();
}
```

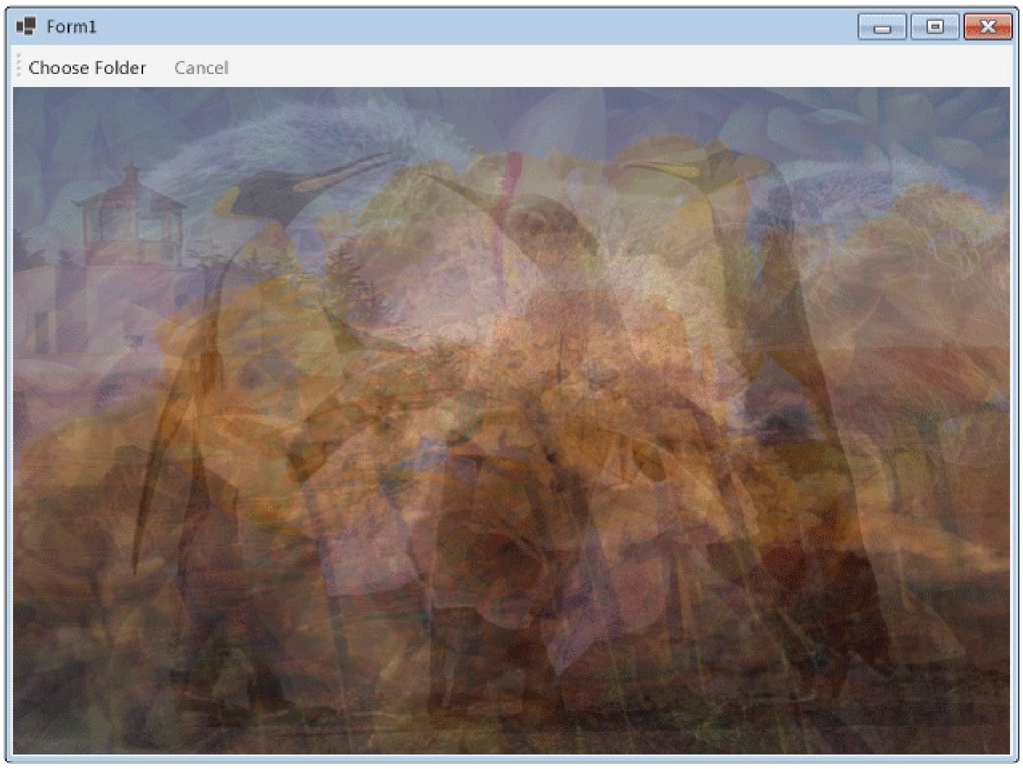
The Complete Example

The following example shows the complete code for this walkthrough.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace CompositeImages
{
   public partial class Form1 : Form
   {
      // The head of the dataflow network.
      ITargetBlock<string> headBlock = null;

      // Enables the user interface to signal cancellation to the network.
      CancellationTokenSource cancellationTokenSource;

      public Form1()
      {
         InitializeComponent();
      }

      // Creates the image processing dataflow network and returns the
      // head node of the network.
      ITargetBlock<string> CreateImageProcessingNetwork()
      {
         //
         // Create the dataflow blocks that form the network.
         //

         // Create a dataflow block that takes a folder path as input
         // and returns a collection of Bitmap objects.
         var loadBitmaps = new TransformBlock<string, IEnumerable<Bitmap>>(path =>
            {
               try
               {
                  return LoadBitmaps(path);
               }
               catch (OperationCanceledException)
               {
                  // Handle cancellation by passing the empty collection
                  // to the next stage of the network.
                  return Enumerable.Empty<Bitmap>();
               }
            });

         // Create a dataflow block that takes a collection of Bitmap objects
         // and returns a single composite bitmap.
         var createCompositeBitmap = new TransformBlock<IEnumerable<Bitmap>, Bitmap>(bitmaps =>
            {
               try
               {
                  return CreateCompositeBitmap(bitmaps);
               }
               catch (OperationCanceledException)
               {
                  // Handle cancellation by passing null to the next stage
                  // of the network.
                  return null;
               }
            });

         // Create a dataflow block that displays the provided bitmap on the form.
         var displayCompositeBitmap = new ActionBlock<Bitmap>(bitmap =>
            {
               // Display the bitmap.
               pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
               pictureBox1.Image = bitmap;

               // Enable the user to select another folder.
               toolStripButton1.Enabled = true;
               toolStripButton2.Enabled = false;
               Cursor = DefaultCursor;
            },
            // Specify a task scheduler from the current synchronization context
            // so that the action runs on the UI thread.
            new ExecutionDataflowBlockOptions
            {
                TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
            });

         // Create a dataflow block that responds to a cancellation request by
         // displaying an image to indicate that the operation is cancelled and
         // enables the user to select another folder.
         var operationCancelled = new ActionBlock<object>(delegate
            {
               // Display the error image to indicate that the operation
               // was cancelled.
               pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
               pictureBox1.Image = pictureBox1.ErrorImage;

               // Enable the user to select another folder.
               toolStripButton1.Enabled = true;
               toolStripButton2.Enabled = false;
               Cursor = DefaultCursor;
            },
            // Specify a task scheduler from the current synchronization context
            // so that the action runs on the UI thread.
            new ExecutionDataflowBlockOptions
            {
               TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
            });

         //
         // Connect the network.
         //

         // Link loadBitmaps to createCompositeBitmap.
         // The provided predicate ensures that createCompositeBitmap accepts the
         // collection of bitmaps only if that collection has at least one member.
         loadBitmaps.LinkTo(createCompositeBitmap, bitmaps => bitmaps.Count() > 0);

         // Also link loadBitmaps to operationCancelled.
         // When createCompositeBitmap rejects the message, loadBitmaps
         // offers the message to operationCancelled.
         // operationCancelled accepts all messages because we do not provide a
         // predicate.
         loadBitmaps.LinkTo(operationCancelled);

         // Link createCompositeBitmap to displayCompositeBitmap.
         // The provided predicate ensures that displayCompositeBitmap accepts the
         // bitmap only if it is non-null.
         createCompositeBitmap.LinkTo(displayCompositeBitmap, bitmap => bitmap != null);

         // Also link createCompositeBitmap to operationCancelled.
         // When displayCompositeBitmap rejects the message, createCompositeBitmap
         // offers the message to operationCancelled.
         // operationCancelled accepts all messages because we do not provide a
         // predicate.
         createCompositeBitmap.LinkTo(operationCancelled);

         // Return the head of the network.
         return loadBitmaps;
      }

      // Loads all bitmap files that exist at the provided path.
      IEnumerable<Bitmap> LoadBitmaps(string path)
      {
         List<Bitmap> bitmaps = new List<Bitmap>();
         // Load a variety of image types.
         foreach (string bitmapType in
            new string[] { "*.bmp", "*.gif", "*.jpg", "*.png", "*.tif" })
         {
            // Load each bitmap for the current extension.
            foreach (string fileName in Directory.GetFiles(path, bitmapType))
            {
               // Throw OperationCanceledException if cancellation is requested.
               cancellationTokenSource.Token.ThrowIfCancellationRequested();

               try
               {
                  // Add the Bitmap object to the collection.
                  bitmaps.Add(new Bitmap(fileName));
               }
               catch (Exception)
               {
                  // TODO: A complete application might handle the error.
               }
            }
         }
         return bitmaps;
      }

      // Creates a composite bitmap from the provided collection of Bitmap objects.
      // This method computes the average color of each pixel among all bitmaps
      // to create the composite image.
      Bitmap CreateCompositeBitmap(IEnumerable<Bitmap> bitmaps)
      {
         Bitmap[] bitmapArray = bitmaps.ToArray();

         // Compute the maximum width and height components of all
         // bitmaps in the collection.
         Rectangle largest = new Rectangle();
         foreach (var bitmap in bitmapArray)
         {
            if (bitmap.Width > largest.Width)
               largest.Width = bitmap.Width;
            if (bitmap.Height > largest.Height)
               largest.Height = bitmap.Height;
         }

         // Create a 32-bit Bitmap object with the greatest dimensions.
         Bitmap result = new Bitmap(largest.Width, largest.Height,
            PixelFormat.Format32bppArgb);

         // Lock the result Bitmap.
         var resultBitmapData = result.LockBits(
            new Rectangle(new Point(), result.Size), ImageLockMode.WriteOnly,
            result.PixelFormat);

         // Lock each source bitmap to create a parallel list of BitmapData objects.
         var bitmapDataList = (from bitmap in bitmapArray
                               select bitmap.LockBits(
                                 new Rectangle(new Point(), bitmap.Size),
                                 ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb))
                              .ToList();

         // Compute each column in parallel.
         Parallel.For(0, largest.Width, new ParallelOptions
         {
            CancellationToken = cancellationTokenSource.Token
         },
         i =>
         {
            // Compute each row.
            for (int j = 0; j < largest.Height; j++)
            {
```

```
      {
         // Counts the number of bitmaps whose dimensions
         // contain the current location.
         int count = 0;

         // The sum of all alpha, red, green, and blue components.
         int a = 0, r = 0, g = 0, b = 0;

         // For each bitmap, compute the sum of all color components.
         foreach (var bitmapData in bitmapDataList)
         {
            // Ensure that we stay within the bounds of the image.
            if (bitmapData.Width > i && bitmapData.Height > j)
            {
               unsafe
               {
                  byte* row = (byte*)(bitmapData.Scan0 + (j * bitmapData.Stride));
                  byte* pix = (byte*)(row + (4 * i));
                  a += *pix; pix++;
                  r += *pix; pix++;
                  g += *pix; pix++;
                  b += *pix;
               }
               count++;
            }
         }

         //prevent divide by zero in bottom right pixelless corner
         if (count == 0)
             break;

         unsafe
         {
            // Compute the average of each color component.
            a /= count;
            r /= count;
            g /= count;
            b /= count;

            // Set the result pixel.
            byte* row = (byte*)(resultBitmapData.Scan0 + (j * resultBitmapData.Stride));
            byte* pix = (byte*)(row + (4 * i));
            *pix = (byte)a; pix++;
            *pix = (byte)r; pix++;
            *pix = (byte)g; pix++;
            *pix = (byte)b;
         }
      }
   });

   // Unlock the source bitmaps.
   for (int i = 0; i < bitmapArray.Length; i++)
   {
      bitmapArray[i].UnlockBits(bitmapDataList[i]);
   }

   // Unlock the result bitmap.
   result.UnlockBits(resultBitmapData);

   // Return the result.
   return result;
}

// Event handler for the Choose Folder button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // Create a FolderBrowserDialog object to enable the user to
   // select a folder.
   FolderBrowserDialog dlg = new FolderBrowserDialog
   {
         {
            ShowNewFolderButton = false
         };

         // Set the selected path to the common Sample Pictures folder
         // if it exists.
         string initialDirectory = Path.Combine(
            Environment.GetFolderPath(Environment.SpecialFolder.CommonPictures),
            "Sample Pictures");
         if (Directory.Exists(initialDirectory))
         {
            dlg.SelectedPath = initialDirectory;
         }

         // Show the dialog and process the dataflow network.
         if (dlg.ShowDialog() == DialogResult.OK)
         {
            // Create a new CancellationTokenSource object to enable
            // cancellation.
            cancellationTokenSource = new CancellationTokenSource();

            // Create the image processing network if needed.
            headBlock ??= CreateImageProcessingNetwork();

            // Post the selected path to the network.
            headBlock.Post(dlg.SelectedPath);

            // Enable the Cancel button and disable the Choose Folder button.
            toolStripButton1.Enabled = false;
            toolStripButton2.Enabled = true;

            // Show a wait cursor.
            Cursor = Cursors.WaitCursor;
         }
      }

      // Event handler for the Cancel button.
      private void toolStripButton2_Click(object sender, EventArgs e)
      {
         // Signal the request for cancellation. The current component of
         // the dataflow network will respond to the cancellation request.
         cancellationTokenSource.Cancel();
      }

      ~Form1()
      {
         cancellationTokenSource.Dispose();
      }
   }
}
```

The following illustration shows typical output for the common \Sample Pictures\ folder.



See also

- Dataflow

How to: Cancel a Dataflow Block

4/9/2019 • 15 minutes to read • Edit Online

This document demonstrates how to enable cancellation in your application. This example uses Windows Forms to show where work items are active in a dataflow pipeline and also the effects of cancellation.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

To Create the Windows Forms Application

1. Create a C# or Visual Basic **Windows Forms Application** project. In the following steps, the project is named `CancellationWinForms`.

2. On the form designer for the main form, Form1.cs (Form1.vb for Visual Basic), add a ToolStrip control.

3. Add a ToolStripButton control to the ToolStrip control. Set the DisplayStyle property to Text and the Text property to **Add Work Items**.

4. Add a second ToolStripButton control to the ToolStrip control. Set the DisplayStyle property to Text, the Text property to **Cancel**, and the Enabled property to `False`.

5. Add four ToolStripProgressBar objects to the ToolStrip control.

Creating the Dataflow Pipeline

This section describes how to create the dataflow pipeline that processes work items and updates the progress bars.

To Create the Dataflow Pipeline

1. In your project, add a reference to System.Threading.Tasks.Dataflow.dll.

2. Ensure that Form1.cs (Form1.vb for Visual Basic) contains the following `using` statements (`Imports` in Visual Basic).

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

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

3. Add the `WorkItem` class as an inner type of the `Form1` class.

```
// A placeholder type that performs work.
class WorkItem
{
   // Performs work for the provided number of milliseconds.
   public void DoWork(int milliseconds)
   {
      // For demonstration, suspend the current thread.
      Thread.Sleep(milliseconds);
   }
}
```

```
' A placeholder type that performs work.
Private Class WorkItem
    ' Performs work for the provided number of milliseconds.
    Public Sub DoWork(ByVal milliseconds As Integer)
        ' For demonstration, suspend the current thread.
        Thread.Sleep(milliseconds)
    End Sub
End Class
```

4. Add the following data members to the `Form1` class.

```
// Enables the user interface to signal cancellation.
CancellationTokenSource cancellationSource;

// The first node in the dataflow pipeline.
TransformBlock<WorkItem, WorkItem> startWork;

// The second, and final, node in the dataflow pipeline.
ActionBlock<WorkItem> completeWork;

// Increments the value of the provided progress bar.
ActionBlock<ToolStripProgressBar> incrementProgress;

// Decrements the value of the provided progress bar.
ActionBlock<ToolStripProgressBar> decrementProgress;

// Enables progress bar actions to run on the UI thread.
TaskScheduler uiTaskScheduler;
```

```
' Enables the user interface to signal cancellation.
Private cancellationSource As CancellationTokenSource

' The first node in the dataflow pipeline.
Private startWork As TransformBlock(Of WorkItem, WorkItem)

' The second, and final, node in the dataflow pipeline.
Private completeWork As ActionBlock(Of WorkItem)

' Increments the value of the provided progress bar.
Private incrementProgress As ActionBlock(Of ToolStripProgressBar)

' Decrements the value of the provided progress bar.
Private decrementProgress As ActionBlock(Of ToolStripProgressBar)

' Enables progress bar actions to run on the UI thread.
Private uiTaskScheduler As TaskScheduler
```

5. Add the following method, `CreatePipeline`, to the `Form1` class.

```
// Creates the blocks that participate in the dataflow pipeline.
// Creates the blocks that participate in the dataflow pipeline.
private void CreatePipeline()
{
   // Create the cancellation source.
   cancellationSource = new CancellationTokenSource();

   // Create the first node in the pipeline.
   startWork = new TransformBlock<WorkItem, WorkItem>(workItem =>
   {
      // Perform some work.
      workItem.DoWork(250);

      // Decrement the progress bar that tracks the count of
      // active work items in this stage of the pipeline.
      decrementProgress.Post(toolStripProgressBar1);

      // Increment the progress bar that tracks the count of
      // active work items in the next stage of the pipeline.
      incrementProgress.Post(toolStripProgressBar2);

      // Send the work item to the next stage of the pipeline.
      return workItem;
   },
   new ExecutionDataflowBlockOptions
   {
      CancellationToken = cancellationSource.Token
   });

   // Create the second, and final, node in the pipeline.
   completeWork = new ActionBlock<WorkItem>(workItem =>
   {
      // Perform some work.
      workItem.DoWork(1000);

      // Decrement the progress bar that tracks the count of
      // active work items in this stage of the pipeline.
      decrementProgress.Post(toolStripProgressBar2);

      // Increment the progress bar that tracks the overall
      // count of completed work items.
      incrementProgress.Post(toolStripProgressBar3);
   },
   new ExecutionDataflowBlockOptions
   {
      CancellationToken = cancellationSource.Token,
      MaxDegreeOfParallelism = 2
   });

   // Connect the two nodes of the pipeline. When the first node completes,
   // set the second node also to the completed state.
   startWork.LinkTo(
      completeWork, new DataflowLinkOptions { PropagateCompletion = true });

   // Create the dataflow action blocks that increment and decrement
   // progress bars.
   // These blocks use the task scheduler that is associated with
   // the UI thread.

   incrementProgress = new ActionBlock<ToolStripProgressBar>(
      progressBar => progressBar.Value++,
      new ExecutionDataflowBlockOptions
      {
         CancellationToken = cancellationSource.Token,
         TaskScheduler = uiTaskScheduler
      });

   decrementProgress = new ActionBlock<ToolStripProgressBar>(
      progressBar => progressBar.Value--,
      new ExecutionDataflowBlockOptions
      {
```

```
      {
         CancellationToken = cancellationSource.Token,
         TaskScheduler = uiTaskScheduler
      });
}
```

```
' Creates the blocks that participate in the dataflow pipeline.
Private Sub CreatePipeline()
    ' Create the cancellation source.
    cancellationSource = New CancellationTokenSource()

    ' Create the first node in the pipeline.
    startWork = New TransformBlock(Of WorkItem, WorkItem)(Function(workItem)
        ' Perform some work.
        ' Decrement the progress bar that tracks the count of
        ' active work items in this stage of the pipeline.
        ' Increment the progress bar that tracks the count of
        ' active work items in the next stage of the pipeline.
        ' Send the work item to the next stage of the pipeline.
        workItem.DoWork(250)
        decrementProgress.Post(toolStripProgressBar1)
        incrementProgress.Post(toolStripProgressBar2)
        Return workItem
    End Function,
    New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token})

    ' Create the second, and final, node in the pipeline.
    completeWork = New ActionBlock(Of WorkItem)(Sub(workItem)
       ' Perform some work.
       ' Decrement the progress bar that tracks the count of
       ' active work items in this stage of the pipeline.
       ' Increment the progress bar that tracks the overall
       ' count of completed work items.
       workItem.DoWork(1000)
       decrementProgress.Post(toolStripProgressBar2)
       incrementProgress.Post(toolStripProgressBar3)
    End Sub,
    New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                            .MaxDegreeOfParallelism = 2 })

    ' Connect the two nodes of the pipeline. When the first node completes,
    ' set the second node also to the completed state.
    startWork.LinkTo(
       completeWork, New DataflowLinkOptions With {.PropagateCompletion = true})

    ' Create the dataflow action blocks that increment and decrement
    ' progress bars.
    ' These blocks use the task scheduler that is associated with
    ' the UI thread.

    incrementProgress = New ActionBlock(Of ToolStripProgressBar)(
       Sub(progressBar) progressBar.Value += 1,
       New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                               .TaskScheduler = uiTaskScheduler})

    decrementProgress = New ActionBlock(Of ToolStripProgressBar)(
       Sub(progressBar) progressBar.Value -= 1,
       New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                               .TaskScheduler = uiTaskScheduler})

End Sub
```

Because the `incrementProgress` and `decrementProgress` dataflow blocks act on the user interface, it is important that these actions occur on the user-interface thread. To accomplish this, during construction these objects each provide a ExecutionDataflowBlockOptions object that has the TaskScheduler property set to

TaskScheduler.FromCurrentSynchronizationContext. The TaskScheduler.FromCurrentSynchronizationContext method creates a TaskScheduler object that performs work on the current synchronization context. Because the `Form1` constructor is called from the user-interface thread, the actions for the `incrementProgress` and `decrementProgress` dataflow blocks also run on the user-interface thread.

This example sets the CancellationToken property when it constructs the members of the pipeline. Because the CancellationToken property permanently cancels dataflow block execution, the whole pipeline must be recreated after the user cancels the operation and then wants to add more work items to the pipeline. For an example that demonstrates an alternative way to cancel a dataflow block so that other work can be performed after an operation is canceled, see Walkthrough: Using Dataflow in a Windows Forms Application.

Connecting the Dataflow Pipeline to the User Interface

This section describes how to connect the dataflow pipeline to the user interface. Both creating the pipeline and adding work items to the pipeline are controlled by the event handler for the **Add Work Items** button. Cancellation is initiated by the **Cancel** button. When the user clicks either of these buttons, the appropriate action is initiated in an asynchronous manner.

To Connect the Dataflow Pipeline to the User Interface

1. On the form designer for the main form, create an event handler for the Click event for the **Add Work Items** button.

2. Implement the Click event for the **Add Work Items** button.

```
// Event handler for the Add Work Items button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // The Cancel button is disabled when the pipeline is not active.
   // Therefore, create the pipeline and enable the Cancel button
   // if the Cancel button is disabled.
   if (!toolStripButton2.Enabled)
   {
      CreatePipeline();

      // Enable the Cancel button.
      toolStripButton2.Enabled = true;
   }

   // Post several work items to the head of the pipeline.
   for (int i = 0; i < 5; i++)
   {
      toolStripProgressBar1.Value++;
      startWork.Post(new WorkItem());
   }
}
```

```
' Event handler for the Add Work Items button.
Private Sub toolStripButton1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton1.Click
    ' The Cancel button is disabled when the pipeline is not active.
    ' Therefore, create the pipeline and enable the Cancel button
    ' if the Cancel button is disabled.
    If Not toolStripButton2.Enabled Then
        CreatePipeline()

        ' Enable the Cancel button.
        toolStripButton2.Enabled = True
    End If

    ' Post several work items to the head of the pipeline.
    For i As Integer = 0 To 4
        toolStripProgressBar1.Value += 1
        startWork.Post(New WorkItem())
    Next i
End Sub
```

3. On the form designer for the main form, create an event handler for the Click event handler for the **Cancel** button.

4. Implement the Click event handler for the **Cancel** button.

```
// Event handler for the Cancel button.
private async void toolStripButton2_Click(object sender, EventArgs e)
{
   // Disable both buttons.
   toolStripButton1.Enabled = false;
   toolStripButton2.Enabled = false;

   // Trigger cancellation.
   cancellationSource.Cancel();

   try
   {
      // Asynchronously wait for the pipeline to complete processing and for
      // the progress bars to update.
      await Task.WhenAll(
         completeWork.Completion,
         incrementProgress.Completion,
         decrementProgress.Completion);
   }
   catch (OperationCanceledException)
   {
   }

   // Increment the progress bar that tracks the number of cancelled
   // work items by the number of active work items.
   toolStripProgressBar4.Value += toolStripProgressBar1.Value;
   toolStripProgressBar4.Value += toolStripProgressBar2.Value;

   // Reset the progress bars that track the number of active work items.
   toolStripProgressBar1.Value = 0;
   toolStripProgressBar2.Value = 0;

   // Enable the Add Work Items button.
   toolStripButton1.Enabled = true;
}
```

```vb
' Event handler for the Cancel button.
Private Async Sub toolStripButton2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton2.Click
    ' Disable both buttons.
    toolStripButton1.Enabled = False
    toolStripButton2.Enabled = False

    ' Trigger cancellation.
    cancellationSource.Cancel()

    Try
        ' Asynchronously wait for the pipeline to complete processing and for
        ' the progress bars to update.
        Await Task.WhenAll(completeWork.Completion, incrementProgress.Completion,
decrementProgress.Completion)
    Catch e1 As OperationCanceledException
    End Try

    ' Increment the progress bar that tracks the number of cancelled
    ' work items by the number of active work items.
    toolStripProgressBar4.Value += toolStripProgressBar1.Value
    toolStripProgressBar4.Value += toolStripProgressBar2.Value

    ' Reset the progress bars that track the number of active work items.
    toolStripProgressBar1.Value = 0
    toolStripProgressBar2.Value = 0

    ' Enable the Add Work Items button.
    toolStripButton1.Enabled = True
End Sub
```

Example

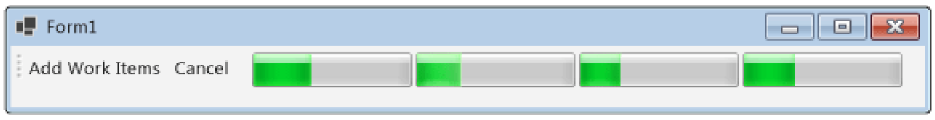
The following example shows the complete code for Form1.cs (Form1.vb for Visual Basic).

```csharp
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace CancellationWinForms
{
   public partial class Form1 : Form
   {
      // A placeholder type that performs work.
      class WorkItem
      {
         // Performs work for the provided number of milliseconds.
         public void DoWork(int milliseconds)
         {
            // For demonstration, suspend the current thread.
            Thread.Sleep(milliseconds);
         }
      }

      // Enables the user interface to signal cancellation.
      CancellationTokenSource cancellationSource;

      // The first node in the dataflow pipeline.
      TransformBlock<WorkItem, WorkItem> startWork;

      // The second, and final, node in the dataflow pipeline.
      ActionBlock<WorkItem> completeWork;
      // Increments the value of the provided progress bar.
      ActionBlock<ToolStripProgressBar> incrementProgress;

      // Decrements the value of the provided progress bar.
      ActionBlock<ToolStripProgressBar> decrementProgress;

      // Enables progress bar actions to run on the UI thread.
      TaskScheduler uiTaskScheduler;

      public Form1()
      {
         InitializeComponent();

         // Create the UI task scheduler from the current sychronization
         // context.
         uiTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
      }

      // Creates the blocks that participate in the dataflow pipeline.
      private void CreatePipeline()
      {
         // Create the cancellation source.
         cancellationSource = new CancellationTokenSource();

         // Create the first node in the pipeline.
         startWork = new TransformBlock<WorkItem, WorkItem>(workItem =>
         {
            // Perform some work.
            workItem.DoWork(250);

            // Decrement the progress bar that tracks the count of
            // active work items in this stage of the pipeline.
            decrementProgress.Post(toolStripProgressBar1);

            // Increment the progress bar that tracks the count of
            // active work items in the next stage of the pipeline.
            incrementProgress.Post(toolStripProgressBar2);

            // Send the work item to the next stage of the pipeline.
            return workItem;
         },
         new ExecutionDataflowBlockOptions
         {
            CancellationToken = cancellationSource.Token
         });

         // Create the second, and final, node in the pipeline.
         completeWork = new ActionBlock<WorkItem>(workItem =>
         {
            // Perform some work.
            workItem.DoWork(1000);

            // Decrement the progress bar that tracks the count of
            // active work items in this stage of the pipeline.
            decrementProgress.Post(toolStripProgressBar2);

            // Increment the progress bar that tracks the overall
            // count of completed work items.
            incrementProgress.Post(toolStripProgressBar3);
         },
         new ExecutionDataflowBlockOptions
         {
            CancellationToken = cancellationSource.Token,
            MaxDegreeOfParallelism = 2
         });

         // Connect the two nodes of the pipeline. When the first node completes,
         // set the second node also to the completed state.
         startWork.LinkTo(
            completeWork, new DataflowLinkOptions { PropagateCompletion = true });

         // Create the dataflow action blocks that increment and decrement
         // progress bars.
         // These blocks use the task scheduler that is associated with
         // the UI thread.

         incrementProgress = new ActionBlock<ToolStripProgressBar>(
            progressBar => progressBar.Value++,
            new ExecutionDataflowBlockOptions
            {
               CancellationToken = cancellationSource.Token,
               TaskScheduler = uiTaskScheduler
            });

         decrementProgress = new ActionBlock<ToolStripProgressBar>(
            progressBar => progressBar.Value--,
            new ExecutionDataflowBlockOptions
            {
               CancellationToken = cancellationSource.Token,
               TaskScheduler = uiTaskScheduler
            });
      }

      // Event handler for the Add Work Items button.
      private void toolStripButton1_Click(object sender, EventArgs e)
      {
         // The Cancel button is disabled when the pipeline is not active.
         // Therefore, create the pipeline and enable the Cancel button
         // if the Cancel button is disabled.
         if (!toolStripButton2.Enabled)
         {
            CreatePipeline();

            // Enable the Cancel button.
            toolStripButton2.Enabled = true;
         }

         // Post several work items to the head of the pipeline.
         for (int i = 0; i < 5; i++)
         {
            toolStripProgressBar1.Value++;
            startWork.Post(new WorkItem());
         }
      }

      // Event handler for the Cancel button.
      private async void toolStripButton2_Click(object sender, EventArgs e)
      {
         // Disable both buttons.
         toolStripButton1.Enabled = false;
         toolStripButton2.Enabled = false;

         // Trigger cancellation.
         cancellationSource.Cancel();

         try
         {
            // Asynchronously wait for the pipeline to complete processing and for
            // the progress bars to update.
            await Task.WhenAll(
               completeWork.Completion,
               incrementProgress.Completion,
               decrementProgress.Completion);
         }
         catch (OperationCanceledException)
         {
         }
```

```
         // Increment the progress bar that tracks the number of cancelled
         // work items by the number of active work items.
         toolStripProgressBar4.Value += toolStripProgressBar1.Value;
         toolStripProgressBar4.Value += toolStripProgressBar2.Value;

         // Reset the progress bars that track the number of active work items.
         toolStripProgressBar1.Value = 0;
         toolStripProgressBar2.Value = 0;

         // Enable the Add Work Items button.
         toolStripButton1.Enabled = true;
      }

      ~Form1()
      {
         cancellationSource.Dispose();
      }
   }
}
```

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


Namespace CancellationWinForms
    Partial Public Class Form1
        Inherits Form
        ' A placeholder type that performs work.
        Private Class WorkItem
            ' Performs work for the provided number of milliseconds.
            Public Sub DoWork(ByVal milliseconds As Integer)
                ' For demonstration, suspend the current thread.
                Thread.Sleep(milliseconds)
            End Sub
        End Class

        ' Enables the user interface to signal cancellation.
        Private cancellationSource As CancellationTokenSource

        ' The first node in the dataflow pipeline.
        Private startWork As TransformBlock(Of WorkItem, WorkItem)

        ' The second, and final, node in the dataflow pipeline.
        Private completeWork As ActionBlock(Of WorkItem)

        ' Increments the value of the provided progress bar.
        Private incrementProgress As ActionBlock(Of ToolStripProgressBar)

        ' Decrements the value of the provided progress bar.
        Private decrementProgress As ActionBlock(Of ToolStripProgressBar)

        ' Enables progress bar actions to run on the UI thread.
        Private uiTaskScheduler As TaskScheduler

        Public Sub New()
            InitializeComponent()

            ' Create the UI task scheduler from the current sychronization
            ' context.
            uiTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
        End Sub

        ' Creates the blocks that participate in the dataflow pipeline.
        Private Sub CreatePipeline()
            ' Create the cancellation source.
            cancellationSource = New CancellationTokenSource()
            ' Create the first node in the pipeline.
            startWork = New TransformBlock(Of WorkItem, WorkItem)(Function(workItem)
                ' Perform some work.
                ' Decrement the progress bar that tracks the count of
                ' active work items in this stage of the pipeline.
                ' Increment the progress bar that tracks the count of
                ' active work items in the next stage of the pipeline.
                ' Send the work item to the next stage of the pipeline.
                workItem.DoWork(250)
                decrementProgress.Post(toolStripProgressBar1)
                incrementProgress.Post(toolStripProgressBar2)
                Return workItem
            End Function,
            New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token})

            ' Create the second, and final, node in the pipeline.
            completeWork = New ActionBlock(Of WorkItem)(Sub(workItem)
               ' Perform some work.
               ' Decrement the progress bar that tracks the count of
               ' active work items in this stage of the pipeline.
               ' Increment the progress bar that tracks the overall
               ' count of completed work items.
               workItem.DoWork(1000)
               decrementProgress.Post(toolStripProgressBar2)
               incrementProgress.Post(toolStripProgressBar3)
            End Sub,
            New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                    .MaxDegreeOfParallelism = 2 })

            ' Connect the two nodes of the pipeline. When the first node completes,
            ' set the second node also to the completed state.
            startWork.LinkTo(
               completeWork, New DataflowLinkOptions With {.PropagateCompletion = true})

            ' Create the dataflow action blocks that increment and decrement
            ' progress bars.
            ' These blocks use the task scheduler that is associated with
            ' the UI thread.

            incrementProgress = New ActionBlock(Of ToolStripProgressBar)(
               Sub(progressBar) progressBar.Value += 1,
               New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                       .TaskScheduler = uiTaskScheduler})

            decrementProgress = New ActionBlock(Of ToolStripProgressBar)(
               Sub(progressBar) progressBar.Value -= 1,
               New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                       .TaskScheduler = uiTaskScheduler})

        End Sub

        ' Event handler for the Add Work Items button.
        Private Sub toolStripButton1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton1.Click
            ' The Cancel button is disabled when the pipeline is not active.
            ' Therefore, create the pipeline and enable the Cancel button
            ' if the Cancel button is disabled.
            If Not toolStripButton2.Enabled Then
                CreatePipeline()

                ' Enable the Cancel button.
                toolStripButton2.Enabled = True
            End If

            ' Post several work items to the head of the pipeline.
            For i As Integer = 0 To 4
                toolStripProgressBar1.Value += 1
                startWork.Post(New WorkItem())
                startWork.Post(New WorkItem())
            Next i
        End Sub

        ' Event handler for the Cancel button.
        Private Async Sub toolStripButton2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton2.Click
            ' Disable both buttons.
            toolStripButton1.Enabled = False
            toolStripButton2.Enabled = False

            ' Trigger cancellation.
            cancellationSource.Cancel()

            Try
                ' Asynchronously wait for the pipeline to complete processing and for
                ' the progress bars to update.
                Await Task.WhenAll(completeWork.Completion, incrementProgress.Completion,
decrementProgress.Completion)
            Catch e1 As OperationCanceledException
            End Try

            ' Increment the progress bar that tracks the number of cancelled
            ' work items by the number of active work items.
            toolStripProgressBar4.Value += toolStripProgressBar1.Value
            toolStripProgressBar4.Value += toolStripProgressBar2.Value

            ' Reset the progress bars that track the number of active work items.
            toolStripProgressBar1.Value = 0
            toolStripProgressBar2.Value = 0

            ' Enable the Add Work Items button.
            toolStripButton1.Enabled = True
        End Sub

        Protected Overrides Sub Finalize()
           cancellationSource.Dispose()
           MyBase.Finalize()
        End Sub
    End Class
End Namespace
```

The following illustration shows the running application.



See also

- Dataflow

Walkthrough: Creating a Custom Dataflow Block Type

5/14/2019 • 20 minutes to read • Edit Online

Although the TPL Dataflow Library provides several dataflow block types that enable a variety of functionality, you can also create custom block types. This document describes how to create a dataflow block type that implements custom behavior.

Prerequisites

Read Dataflow before you read this document.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Defining the Sliding Window Dataflow Block

Consider a dataflow application that requires that input values be buffered and then output in a sliding window manner. For example, for the input values {0, 1, 2, 3, 4, 5} and a window size of three, a sliding window dataflow block produces the output arrays {0, 1, 2}, {1, 2, 3}, {2, 3, 4}, and {3, 4, 5}. The following sections describe two ways to create a dataflow block type that implements this custom behavior. The first technique uses the Encapsulate method to combine the functionality of an ISourceBlock<TOutput> object and an ITargetBlock<TInput> object into one propagator block. The second technique defines a class that derives from IPropagatorBlock<TInput,TOutput> and combines existing functionality to perform custom behavior.

Using the Encapsulate Method to Define the Sliding Window Dataflow Block

The following example uses the Encapsulate method to create a propagator block from a target and a source. A propagator block enables a source block and a target block to act as a receiver and sender of data.

This technique is useful when you require custom dataflow functionality, but you do not require a type that provides additional methods, properties, or fields.

```
// Creates a IPropagatorBlock<T, T[]> object propagates data in a
// sliding window fashion.
public static IPropagatorBlock<T, T[]> CreateSlidingWindow<T>(int windowSize)
{
   // Create a queue to hold messages.
   var queue = new Queue<T>();

   // The source part of the propagator holds arrays of size windowSize
   // and propagates data out to any connected targets.
   var source = new BufferBlock<T[]>();

   // The target part receives data and adds them to the queue.
   var target = new ActionBlock<T>(item =>
   {
      // Add the item to the queue.
      queue.Enqueue(item);
      // Remove the oldest item when the queue size exceeds the window size.
      if (queue.Count > windowSize)
         queue.Dequeue();
      // Post the data in the queue to the source block when the queue size
      // equals the window size.
      if (queue.Count == windowSize)
         source.Post(queue.ToArray());
   });

   // When the target is set to the completed state, propagate out any
   // remaining data and set the source to the completed state.
   target.Completion.ContinueWith(delegate
   {
      if (queue.Count > 0 && queue.Count < windowSize)
         source.Post(queue.ToArray());
      source.Complete();
   });

   // Return a IPropagatorBlock<T, T[]> object that encapsulates the
   // target and source blocks.
   return DataflowBlock.Encapsulate(target, source);
}
```

```
' Creates a IPropagatorBlock<T, T[]> object propagates data in a
' sliding window fashion.
Public Shared Function CreateSlidingWindow(Of T)(ByVal windowSize As Integer) As IPropagatorBlock(Of T, T())
    ' Create a queue to hold messages.
    Dim queue = New Queue(Of T)()

    ' The source part of the propagator holds arrays of size windowSize
    ' and propagates data out to any connected targets.
    Dim source = New BufferBlock(Of T())()

    ' The target part receives data and adds them to the queue.
    Dim target = New ActionBlock(Of T)(Sub(item)
        ' Add the item to the queue.
        ' Remove the oldest item when the queue size exceeds the window size.
        ' Post the data in the queue to the source block when the queue size
        ' equals the window size.
        queue.Enqueue(item)
        If queue.Count > windowSize Then
            queue.Dequeue()
        End If
        If queue.Count = windowSize Then
            source.Post(queue.ToArray())
        End If
    End Sub)

    ' When the target is set to the completed state, propagate out any
    ' remaining data and set the source to the completed state.
    target.Completion.ContinueWith(Sub()
        If queue.Count > 0 AndAlso queue.Count < windowSize Then
            source.Post(queue.ToArray())
        End If
        source.Complete()
    End Sub)

    ' Return a IPropagatorBlock<T, T[]> object that encapsulates the
    ' target and source blocks.
    Return DataflowBlock.Encapsulate(target, source)
End Function
```

Deriving from IPropagatorBlock to Define the Sliding Window Dataflow Block

The following example shows the `SlidingWindowBlock` class. This class derives from IPropagatorBlock<TInput,TOutput> so that it can act as both a source and a target of data. As in the previous example, the `SlidingWindowBlock` class is built on existing dataflow block types. However, the `SlidingWindowBlock` class also implements the methods that are required by the ISourceBlock<TOutput>, ITargetBlock<TInput>, and IDataflowBlock interfaces. These methods all forward work to the predefined dataflow block type members. For example, the `Post` method defers work to the `m_target` data member, which is also an ITargetBlock<TInput> object.

This technique is useful when you require custom dataflow functionality, and also require a type that provides additional methods, properties, or fields. For example, the `SlidingWindowBlock` class also derives from IReceivableSourceBlock<TOutput> so that it can provide the TryReceive and TryReceiveAll methods. The `SlidingWindowBlock` class also demonstrates extensibility by providing the `WindowSize` property, which retrieves the number of elements in the sliding window.

```
// Propagates data in a sliding window fashion.
public class SlidingWindowBlock<T> : IPropagatorBlock<T, T[]>,
                                     IReceivableSourceBlock<T[]>
{
   // The size of the window.
   private readonly int m_windowSize;
   // The target part of the block.
   private readonly ITargetBlock<T> m_target;
   // The source part of the block.
   private readonly IReceivableSourceBlock<T[]> m_source;

   // Constructs a SlidingWindowBlock object.
   public SlidingWindowBlock(int windowSize)
   {
      // Create a queue to hold messages.
      var queue = new Queue<T>();

      // The source part of the propagator holds arrays of size windowSize
      // and propagates data out to any connected targets.
      var source = new BufferBlock<T[]>();

      // The target part receives data and adds them to the queue.
      var target = new ActionBlock<T>(item =>
      {
         // Add the item to the queue.
         queue.Enqueue(item);
         // Remove the oldest item when the queue size exceeds the window size.
         if (queue.Count > windowSize)
            queue.Dequeue();
         // Post the data in the queue to the source block when the queue size
         // equals the window size.
         if (queue.Count == windowSize)
            source.Post(queue.ToArray());
      });

      // When the target is set to the completed state, propagate out any
      // remaining data and set the source to the completed state.
      target.Completion.ContinueWith(delegate
      {
         if (queue.Count > 0 && queue.Count < windowSize)
            source.Post(queue.ToArray());
         source.Complete();
      });

      m_windowSize = windowSize;
      m_target = target;
      m_source = source;
   }

   // Retrieves the size of the window.
   public int WindowSize { get { return m_windowSize; } }

   #region IReceivableSourceBlock<TOutput> members

   // Attempts to synchronously receive an item from the source.
   public bool TryReceive(Predicate<T[]> filter, out T[] item)
   {
      return m_source.TryReceive(filter, out item);
   }

   // Attempts to remove all available elements from the source into a new
   // array that is returned.
   public bool TryReceiveAll(out IList<T[]> items)
   {
      return m_source.TryReceiveAll(out items);
   }

   #endregion

   #region ISourceBlock<TOutput> members

   // Links this dataflow block to the provided target.
   public IDisposable LinkTo(ITargetBlock<T[]> target, DataflowLinkOptions linkOptions)
   {
      return m_source.LinkTo(target, linkOptions);
   }

   // Called by a target to reserve a message previously offered by a source
   // but not yet consumed by this target.
   bool ISourceBlock<T[]>.ReserveMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target)
   {
      return m_source.ReserveMessage(messageHeader, target);
   }

   // Called by a target to consume a previously offered message from a source.
   T[] ISourceBlock<T[]>.ConsumeMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target, out bool messageConsumed)
   {
      return m_source.ConsumeMessage(messageHeader,
         target, out messageConsumed);
   }

   // Called by a target to release a previously reserved message from a source.
   void ISourceBlock<T[]>.ReleaseReservation(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target)
   {
      m_source.ReleaseReservation(messageHeader, target);
   }

   #endregion

   #region ITargetBlock<TInput> members

   // Asynchronously passes a message to the target block, giving the target the
   // opportunity to consume the message.
   DataflowMessageStatus ITargetBlock<T>.OfferMessage(DataflowMessageHeader messageHeader,
      T messageValue, ISourceBlock<T> source, bool consumeToAccept)
   {
      return m_target.OfferMessage(messageHeader,
         messageValue, source, consumeToAccept);
   }

   #endregion

   #region IDataflowBlock members

   // Gets a Task that represents the completion of this dataflow block.
   public Task Completion { get { return m_source.Completion; } }

   // Signals to this target block that it should not accept any more messages,
   // nor consume postponed messages.
   public void Complete()
   {
      m_target.Complete();
   }

   public void Fault(Exception error)
   {
      m_target.Fault(error);
   }

   #endregion
}
```

```
' Propagates data in a sliding window fashion.
Public Class SlidingWindowBlock(Of T)
    Implements IPropagatorBlock(Of T, T()), IReceivableSourceBlock(Of T())
    ' The size of the window.
    Private ReadOnly m_windowSize As Integer
    ' The target part of the block.
```

```
        ' The target part of the block.
        Private ReadOnly m_target As ITargetBlock(Of T)
        ' The source part of the block.
        Private ReadOnly m_source As IReceivableSourceBlock(Of T())

        ' Constructs a SlidingWindowBlock object.
        Public Sub New(ByVal windowSize As Integer)
            ' Create a queue to hold messages.
            Dim queue = New Queue(Of T)()

            ' The source part of the propagator holds arrays of size windowSize
            ' and propagates data out to any connected targets.
            Dim source = New BufferBlock(Of T())()

            ' The target part receives data and adds them to the queue.
            Dim target = New ActionBlock(Of T)(Sub(item)
                ' Add the item to the queue.
                ' Remove the oldest item when the queue size exceeds the window size.
                ' Post the data in the queue to the source block when the queue size
                ' equals the window size.
                queue.Enqueue(item)
                If queue.Count > windowSize Then
                    queue.Dequeue()
                End If
                If queue.Count = windowSize Then
                    source.Post(queue.ToArray())
                End If
            End Sub)

            ' When the target is set to the completed state, propagate out any
            ' remaining data and set the source to the completed state.
            target.Completion.ContinueWith(Sub()
                If queue.Count > 0 AndAlso queue.Count < windowSize Then
                    source.Post(queue.ToArray())
                End If
                source.Complete()
            End Sub)

            m_windowSize = windowSize
            m_target = target
            m_source = source
        End Sub

        ' Retrieves the size of the window.
        Public ReadOnly Property WindowSize() As Integer
            Get
                Return m_windowSize
            End Get
        End Property

        '#Region "IReceivableSourceBlock<TOutput> members"

        ' Attempts to synchronously receive an item from the source.
        Public Function TryReceive(ByVal filter As Predicate(Of T()), <System.Runtime.InteropServices.Out()>
ByRef item() As T) As Boolean Implements IReceivableSourceBlock(Of T()).TryReceive
            Return m_source.TryReceive(filter, item)
        End Function

        ' Attempts to remove all available elements from the source into a new
        ' array that is returned.
        Public Function TryReceiveAll(<System.Runtime.InteropServices.Out()> ByRef items As IList(Of T())) As
Boolean Implements IReceivableSourceBlock(Of T()).TryReceiveAll
            Return m_source.TryReceiveAll(items)
        End Function

        '#End Region

#Region "ISourceBlock<TOutput> members"

        ' Links this dataflow block to the provided target.
        ' Links this dataflow block to the provided target.
        Public Function LinkTo(ByVal target As ITargetBlock(Of T()), ByVal linkOptions As DataflowLinkOptions)
As IDisposable Implements ISourceBlock(Of T()).LinkTo
            Return m_source.LinkTo(target, linkOptions)
        End Function

        ' Called by a target to reserve a message previously offered by a source
        ' but not yet consumed by this target.
        Private Function ReserveMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) As Boolean Implements ISourceBlock(Of T()).ReserveMessage
            Return m_source.ReserveMessage(messageHeader, target)
        End Function

        ' Called by a target to consume a previously offered message from a source.
        Private Function ConsumeMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T()), ByRef messageConsumed As Boolean) As T() Implements ISourceBlock(Of T()).ConsumeMessage
            Return m_source.ConsumeMessage(messageHeader, target, messageConsumed)
        End Function

        ' Called by a target to release a previously reserved message from a source.
        Private Sub ReleaseReservation(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) Implements ISourceBlock(Of T()).ReleaseReservation
            m_source.ReleaseReservation(messageHeader, target)
        End Sub

#End Region

#Region "ITargetBlock<TInput> members"

        ' Asynchronously passes a message to the target block, giving the target the
        ' opportunity to consume the message.
        Private Function OfferMessage(ByVal messageHeader As DataflowMessageHeader, ByVal messageValue As T,
ByVal source As ISourceBlock(Of T), ByVal consumeToAccept As Boolean) As DataflowMessageStatus Implements
ITargetBlock(Of T).OfferMessage
            Return m_target.OfferMessage(messageHeader, messageValue, source, consumeToAccept)
        End Function

#End Region

#Region "IDataflowBlock members"

        ' Gets a Task that represents the completion of this dataflow block.
        Public ReadOnly Property Completion() As Task Implements IDataflowBlock.Completion
            Get
                Return m_source.Completion
            End Get
        End Property

        ' Signals to this target block that it should not accept any more messages,
        ' nor consume postponed messages.
        Public Sub Complete() Implements IDataflowBlock.Complete
            m_target.Complete()
        End Sub

        Public Sub Fault(ByVal [error] As Exception) Implements IDataflowBlock.Fault
            m_target.Fault([error])
        End Sub

#End Region
    End Class
```

The Complete Example

The following example shows the complete code for this walkthrough. It also demonstrates how to use the both sliding window blocks in a method that writes to the block, reads from it, and prints the results to the console.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a custom dataflow block type.
class Program
{
   // Creates a IPropagatorBlock<T, T[]> object propagates data in a
   // sliding window fashion.
   public static IPropagatorBlock<T, T[]> CreateSlidingWindow<T>(int windowSize)
   {
      // Create a queue to hold messages.
      var queue = new Queue<T>();

      // The source part of the propagator holds arrays of size windowSize
      // and propagates data out to any connected targets.
      var source = new BufferBlock<T[]>();

      // The target part receives data and adds them to the queue.
      var target = new ActionBlock<T>(item =>
      {
         // Add the item to the queue.
         queue.Enqueue(item);
         // Remove the oldest item when the queue size exceeds the window size.
         if (queue.Count > windowSize)
            queue.Dequeue();
         // Post the data in the queue to the source block when the queue size
         // equals the window size.
         if (queue.Count == windowSize)
            source.Post(queue.ToArray());
      });

      // When the target is set to the completed state, propagate out any
      // remaining data and set the source to the completed state.
      target.Completion.ContinueWith(delegate
      {
         if (queue.Count > 0 && queue.Count < windowSize)
            source.Post(queue.ToArray());
         source.Complete();
      });

      // Return a IPropagatorBlock<T, T[]> object that encapsulates the
      // target and source blocks.
      return DataflowBlock.Encapsulate(target, source);
   }

   // Propagates data in a sliding window fashion.
   public class SlidingWindowBlock<T> : IPropagatorBlock<T, T[]>,
                                        IReceivableSourceBlock<T[]>
   {
      // The size of the window.
      private readonly int m_windowSize;
      // The target part of the block.
      private readonly ITargetBlock<T> m_target;
      // The source part of the block.
      private readonly IReceivableSourceBlock<T[]> m_source;

      // Constructs a SlidingWindowBlock object.
      public SlidingWindowBlock(int windowSize)
      {
         // Create a queue to hold messages.
         var queue = new Queue<T>();

         // The source part of the propagator holds arrays of size windowSize
         // and propagates data out to any connected targets.
         var source = new BufferBlock<T[]>();
         // The target part receives data and adds them to the queue.
         var target = new ActionBlock<T>(item =>
         {
            // Add the item to the queue.
            queue.Enqueue(item);
            // Remove the oldest item when the queue size exceeds the window size.
            if (queue.Count > windowSize)
               queue.Dequeue();
            // Post the data in the queue to the source block when the queue size
            // equals the window size.
            if (queue.Count == windowSize)
               source.Post(queue.ToArray());
         });

         // When the target is set to the completed state, propagate out any
         // remaining data and set the source to the completed state.
         target.Completion.ContinueWith(delegate
         {
            if (queue.Count > 0 && queue.Count < windowSize)
               source.Post(queue.ToArray());
            source.Complete();
         });

         m_windowSize = windowSize;
         m_target = target;
         m_source = source;
      }

      // Retrieves the size of the window.
      public int WindowSize { get { return m_windowSize; } }

      #region IReceivableSourceBlock<TOutput> members

      // Attempts to synchronously receive an item from the source.
      public bool TryReceive(Predicate<T[]> filter, out T[] item)
      {
         return m_source.TryReceive(filter, out item);
      }

      // Attempts to remove all available elements from the source into a new
      // array that is returned.
      public bool TryReceiveAll(out IList<T[]> items)
      {
         return m_source.TryReceiveAll(out items);
      }

      #endregion

      #region ISourceBlock<TOutput> members

      // Links this dataflow block to the provided target.
      public IDisposable LinkTo(ITargetBlock<T[]> target, DataflowLinkOptions linkOptions)
      {
         return m_source.LinkTo(target, linkOptions);
      }

      // Called by a target to reserve a message previously offered by a source
      // but not yet consumed by this target.
      bool ISourceBlock<T[]>.ReserveMessage(DataflowMessageHeader messageHeader,
         ITargetBlock<T[]> target)
      {
         return m_source.ReserveMessage(messageHeader, target);
      }

      // Called by a target to consume a previously offered message from a source.
      T[] ISourceBlock<T[]>.ConsumeMessage(DataflowMessageHeader messageHeader,
         ITargetBlock<T[]> target, out bool messageConsumed)
      {
         return m_source.ConsumeMessage(messageHeader,
            target, out messageConsumed);
      }

      // Called by a target to release a previously reserved message from a source.
      void ISourceBlock<T[]>.ReleaseReservation(DataflowMessageHeader messageHeader,
         ITargetBlock<T[]> target)
      {
         m_source.ReleaseReservation(messageHeader, target);
      }

      #endregion

      #region ITargetBlock<TInput> members

      // Asynchronously passes a message to the target block, giving the target the
      // opportunity to consume the message.
      DataflowMessageStatus ITargetBlock<T>.OfferMessage(DataflowMessageHeader messageHeader,
         T messageValue, ISourceBlock<T> source, bool consumeToAccept)
      {
         return m_target.OfferMessage(messageHeader,
            messageValue, source, consumeToAccept);
      }

      #endregion

      #region IDataflowBlock members

      // Gets a Task that represents the completion of this dataflow block.
      public Task Completion { get { return m_source.Completion; } }

      // Signals to this target block that it should not accept any more messages,
      // nor consume postponed messages.
      public void Complete()
      {
         m_target.Complete();
      }

      public void Fault(Exception error)
      {
         m_target.Fault(error);
      }

      #endregion
   }

   // Demonstrates usage of the sliding window block by sending the provided
   // values to the provided propagator block and printing the output of
   // that block to the console.
   static void DemonstrateSlidingWindow<T>(IPropagatorBlock<T, T[]> slidingWindow,
      IEnumerable<T> values)
   {
      // Create an action block that prints arrays of data to the console.
      string windowComma = string.Empty;
      var printWindow = new ActionBlock<T[]>(window =>
      {
         Console.Write(windowComma);
         Console.Write("{");

         string comma = string.Empty;
         foreach (T item in window)
         {
            Console.Write(comma);
            Console.Write(item);
            comma = ",";
         }
         Console.Write("}");

         windowComma = ", ";
      });
      // Link the printer block to the sliding window block.
      slidingWindow.LinkTo(printWindow);

      // Set the printer block to the completed state when the sliding window
      // block completes.
      slidingWindow.Completion.ContinueWith(delegate { printWindow.Complete(); });

      // Print an additional newline to the console when the printer block completes.
      var completion = printWindow.Completion.ContinueWith(delegate { Console.WriteLine(); });

      // Post the provided values to the sliding window block and then wait
      // for the sliding window block to complete.
      foreach (T value in values)
      {
         slidingWindow.Post(value);
      }
      slidingWindow.Complete();

      // Wait for the printer to complete and perform its final action.
      completion.Wait();
   }

   static void Main(string[] args)
   {

      Console.Write("Using the DataflowBlockExtensions.Encapsulate method ");
      Console.WriteLine("(T=int, windowSize=3):");
      DemonstrateSlidingWindow(CreateSlidingWindow<int>(3), Enumerable.Range(0, 10));

      Console.WriteLine();

      var slidingWindow = new SlidingWindowBlock<char>(4);

      Console.Write("Using SlidingWindowBlock<T> ");
      Console.WriteLine("(T=char, windowSize={0}):", slidingWindow.WindowSize);
      DemonstrateSlidingWindow(slidingWindow, from n in Enumerable.Range(65, 10)
                                              select (char)n);
   }
}

/* Output:
Using the DataflowBlockExtensions.Encapsulate method (T=int, windowSize=3):
{0,1,2}, {1,2,3}, {2,3,4}, {3,4,5}, {4,5,6}, {5,6,7}, {6,7,8}, {7,8,9}

Using SlidingWindowBlock<T> (T=char, windowSize=4):
{A,B,C,D}, {B,C,D,E}, {C,D,E,F}, {D,E,F,G}, {E,F,G,H}, {F,G,H,I}, {G,H,I,J}
 */
```

```
Imports System.Collections.Generic
Imports System.Linq
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a custom dataflow block type.
Friend Class Program
    ' Creates a IPropagatorBlock<T, T[]> object propagates data in a
    ' sliding window fashion.
    Public Shared Function CreateSlidingWindow(Of T)(ByVal windowSize As Integer) As IPropagatorBlock(Of T,
T())
        ' Create a queue to hold messages.
        Dim queue = New Queue(Of T)()

        ' The source part of the propagator holds arrays of size windowSize
        ' and propagates data out to any connected targets.
        Dim source = New BufferBlock(Of T())()
        ' The target part receives data and adds them to the queue.
        Dim target = New ActionBlock(Of T)(Sub(item)
            ' Add the item to the queue.
            ' Remove the oldest item when the queue size exceeds the window size.
            ' Post the data in the queue to the source block when the queue size
            ' equals the window size.
            queue.Enqueue(item)
            If queue.Count > windowSize Then
                queue.Dequeue()
            End If
            If queue.Count = windowSize Then
                source.Post(queue.ToArray())
            End If
        End Sub)

        ' When the target is set to the completed state, propagate out any
        ' remaining data and set the source to the completed state.
        target.Completion.ContinueWith(Sub()
            If queue.Count > 0 AndAlso queue.Count < windowSize Then
                source.Post(queue.ToArray())
            End If
            source.Complete()
        End Sub)

        ' Return a IPropagatorBlock<T, T[]> object that encapsulates the
        ' target and source blocks.
        Return DataflowBlock.Encapsulate(target, source)
    End Function

    ' Propagates data in a sliding window fashion.
    Public Class SlidingWindowBlock(Of T)
        Implements IPropagatorBlock(Of T, T()), IReceivableSourceBlock(Of T())
        ' The size of the window.
        Private ReadOnly m_windowSize As Integer
        ' The target part of the block.
        Private ReadOnly m_target As ITargetBlock(Of T)
        ' The source part of the block.
        Private ReadOnly m_source As IReceivableSourceBlock(Of T())

        ' Constructs a SlidingWindowBlock object.
        Public Sub New(ByVal windowSize As Integer)
            ' Create a queue to hold messages.
            Dim queue = New Queue(Of T)()

            ' The source part of the propagator holds arrays of size windowSize
            ' and propagates data out to any connected targets.
            Dim source = New BufferBlock(Of T())()

            ' The target part receives data and adds them to the queue.
            Dim target = New ActionBlock(Of T)(Sub(item)
                ' Add the item to the queue.
                ' Remove the oldest item when the queue size exceeds the window size.
                ' Post the data in the queue to the source block when the queue size
                ' equals the window size.
                queue.Enqueue(item)
                If queue.Count > windowSize Then
                    queue.Dequeue()
                End If
                If queue.Count = windowSize Then
                    source.Post(queue.ToArray())
                End If
            End Sub)

            ' When the target is set to the completed state, propagate out any
            ' remaining data and set the source to the completed state.
            target.Completion.ContinueWith(Sub()
                If queue.Count > 0 AndAlso queue.Count < windowSize Then
                    source.Post(queue.ToArray())
                End If
                source.Complete()
            End Sub)

            m_windowSize = windowSize
            m_target = target
            m_source = source
        End Sub

        ' Retrieves the size of the window.
        Public ReadOnly Property WindowSize() As Integer
            Get
                Return m_windowSize
            End Get
        End Property

        '#Region "IReceivableSourceBlock<TOutput> members"

        ' Attempts to synchronously receive an item from the source.
        Public Function TryReceive(ByVal filter As Predicate(Of T()), <System.Runtime.InteropServices.Out()>
ByRef item() As T) As Boolean Implements IReceivableSourceBlock(Of T()).TryReceive
            Return m_source.TryReceive(filter, item)
        End Function

        ' Attempts to remove all available elements from the source into a new
        ' array that is returned.
        Public Function TryReceiveAll(<System.Runtime.InteropServices.Out()> ByRef items As IList(Of T())) As
Boolean Implements IReceivableSourceBlock(Of T()).TryReceiveAll
            Return m_source.TryReceiveAll(items)
        End Function

        '#End Region

#Region "ISourceBlock<TOutput> members"

        ' Links this dataflow block to the provided target.
        Public Function LinkTo(ByVal target As ITargetBlock(Of T()), ByVal linkOptions As DataflowLinkOptions)
As IDisposable Implements ISourceBlock(Of T()).LinkTo
            Return m_source.LinkTo(target, linkOptions)
        End Function

        ' Called by a target to reserve a message previously offered by a source
        ' but not yet consumed by this target.
        Private Function ReserveMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) As Boolean Implements ISourceBlock(Of T()).ReserveMessage
            Return m_source.ReserveMessage(messageHeader, target)
        End Function

        ' Called by a target to consume a previously offered message from a source.
        Private Function ConsumeMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T()), ByRef messageConsumed As Boolean) As T() Implements ISourceBlock(Of T()).ConsumeMessage
            Return m_source.ConsumeMessage(messageHeader, target, messageConsumed)
        End Function

        ' Called by a target to release a previously reserved message from a source.
        Private Sub ReleaseReservation(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) Implements ISourceBlock(Of T()).ReleaseReservation
            m_source.ReleaseReservation(messageHeader, target)
        End Sub

#End Region

#Region "ITargetBlock<TInput> members"

        ' Asynchronously passes a message to the target block, giving the target the
        ' opportunity to consume the message.
        Private Function OfferMessage(ByVal messageHeader As DataflowMessageHeader, ByVal messageValue As T,
ByVal source As ISourceBlock(Of T), ByVal consumeToAccept As Boolean) As DataflowMessageStatus Implements
ITargetBlock(Of T).OfferMessage
            Return m_target.OfferMessage(messageHeader, messageValue, source, consumeToAccept)
            Return m_target.OfferMessage(messageHeader, messageValue, source, consumeToAccept)
        End Function

#End Region

#Region "IDataflowBlock members"

        ' Gets a Task that represents the completion of this dataflow block.
        Public ReadOnly Property Completion() As Task Implements IDataflowBlock.Completion
            Get
                Return m_source.Completion
            End Get
        End Property

        ' Signals to this target block that it should not accept any more messages,
        ' nor consume postponed messages.
        Public Sub Complete() Implements IDataflowBlock.Complete
            m_target.Complete()
        End Sub

        Public Sub Fault(ByVal [error] As Exception) Implements IDataflowBlock.Fault
            m_target.Fault([error])
        End Sub

#End Region
    End Class

    ' Demonstrates usage of the sliding window block by sending the provided
    ' values to the provided propagator block and printing the output of
    ' that block to the console.
    Private Shared Sub DemonstrateSlidingWindow(Of T)(ByVal slidingWindow As IPropagatorBlock(Of T, T()),
ByVal values As IEnumerable(Of T))
        ' Create an action block that prints arrays of data to the console.
        Dim windowComma As String = String.Empty
        Dim printWindow = New ActionBlock(Of T())(Sub(window)
            Console.Write(windowComma)
            Console.Write("{")
            Dim comma As String = String.Empty
            For Each item As T In window
                Console.Write(comma)
                Console.Write(item)
                comma = ","
            Next item
            Console.Write("}")
            windowComma = ", "
        End Sub)

        ' Link the printer block to the sliding window block.
        slidingWindow.LinkTo(printWindow)

        ' Set the printer block to the completed state when the sliding window
        ' block completes.
        slidingWindow.Completion.ContinueWith(Sub() printWindow.Complete())

        ' Print an additional newline to the console when the printer block completes.
        Dim completion = printWindow.Completion.ContinueWith(Sub() Console.WriteLine())

        ' Post the provided values to the sliding window block and then wait
        ' for the sliding window block to complete.
        For Each value As T In values
            slidingWindow.Post(value)
        Next value
        slidingWindow.Complete()

        ' Wait for the printer to complete and perform its final action.
        completion.Wait()
    End Sub

    Shared Sub Main(ByVal args() As String)
```

```
        Console.Write("Using the DataflowBlockExtensions.Encapsulate method ")
        Console.WriteLine("(T=int, windowSize=3):")
        DemonstrateSlidingWindow(CreateSlidingWindow(Of Integer)(3), Enumerable.Range(0, 10))

        Console.WriteLine()

        Dim slidingWindow = New SlidingWindowBlock(Of Char)(4)

        Console.Write("Using SlidingWindowBlock<T> ")
        Console.WriteLine("(T=char, windowSize={0}):", slidingWindow.WindowSize)
        DemonstrateSlidingWindow(slidingWindow, _
            From n In Enumerable.Range(65, 10) _
            Select ChrW(n))
    End Sub
End Class

' Output:
'Using the DataflowBlockExtensions.Encapsulate method (T=int, windowSize=3):
'{0,1,2}, {1,2,3}, {2,3,4}, {3,4,5}, {4,5,6}, {5,6,7}, {6,7,8}, {7,8,9}
'
'Using SlidingWindowBlock<T> (T=char, windowSize=4):
'{A,B,C,D}, {B,C,D,E}, {C,D,E,F}, {D,E,F,G}, {E,F,G,H}, {F,G,H,I}, {G,H,I,J}
'
```

See also

- Dataflow

How to: Use JoinBlock to Read Data From Multiple Sources

5/14/2019 • 7 minutes to read • Edit Online

This document explains how to use the JoinBlock<T1,T2> class to perform an operation when data is available from multiple sources. It also demonstrates how to use non-greedy mode to enable multiple join blocks to share a data source more efficiently.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Example

The following example defines three resource types, `NetworkResource`, `FileResource`, and `MemoryResource`, and performs operations when resources become available. This example requires a `NetworkResource` and `MemoryResource` pair in order to perform the first operation and a `FileResource` and `MemoryResource` pair in order to perform the second operation. To enable these operations to occur when all required resources are available, this example uses the JoinBlock<T1,T2> class. When a JoinBlock<T1,T2> object receives data from all sources, it propagates that data to its target, which in this example is an ActionBlock<TInput> object. Both JoinBlock<T1,T2> objects read from a shared pool of `MemoryResource` objects.

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to use non-greedy join blocks to distribute
// resources among a dataflow network.
class Program
{
   // Represents a resource. A derived class might represent
   // a limited resource such as a memory, network, or I/O
   // device.
   abstract class Resource
   {
   }

   // Represents a memory resource. For brevity, the details of
   // this class are omitted.
   class MemoryResource : Resource
   {
   }

   // Represents a network resource. For brevity, the details of
   // this class are omitted.
   class NetworkResource : Resource
   {
   }

   // Represents a file resource. For brevity, the details of
   // this class are omitted.
   class FileResource : Resource
   {
   }

   static void Main(string[] args)
   {
      // Create three BufferBlock<T> objects. Each object holds a different
      // type of resource.
      var networkResources = new BufferBlock<NetworkResource>();
      var fileResources = new BufferBlock<FileResource>();
      var memoryResources = new BufferBlock<MemoryResource>();

      // Create two non-greedy JoinBlock<T1, T2> objects.
      // The first join works with network and memory resources;
      // the second pool works with file and memory resources.

      var joinNetworkAndMemoryResources =
         new JoinBlock<NetworkResource, MemoryResource>(
            new GroupingDataflowBlockOptions
            {
               Greedy = false
            });

      var joinFileAndMemoryResources =
         new JoinBlock<FileResource, MemoryResource>(
            new GroupingDataflowBlockOptions
            {
               Greedy = false
            });

      // Create two ActionBlock<T> objects.
      // The first block acts on a network resource and a memory resource.
      // The second block acts on a file resource and a memory resource.

      var networkMemoryAction =
         new ActionBlock<Tuple<NetworkResource, MemoryResource>>(
            data =>
            {
               // Perform some action on the resources.

               // Print a message.
               Console.WriteLine("Network worker: using resources...");

               // Simulate a lengthy operation that uses the resources.
               Thread.Sleep(new Random().Next(500, 2000));

               // Print a message.
               Console.WriteLine("Network worker: finished using resources...");

               // Release the resources back to their respective pools.
               networkResources.Post(data.Item1);
               memoryResources.Post(data.Item2);
            });

      var fileMemoryAction =
         new ActionBlock<Tuple<FileResource, MemoryResource>>(
            data =>
            {
               // Perform some action on the resources.

               // Print a message.
               Console.WriteLine("File worker: using resources...");

               // Simulate a lengthy operation that uses the resources.
               Thread.Sleep(new Random().Next(500, 2000));

               // Print a message.
               Console.WriteLine("File worker: finished using resources...");
               // Release the resources back to their respective pools.
               fileResources.Post(data.Item1);
               memoryResources.Post(data.Item2);
            });

      // Link the resource pools to the JoinBlock<T1, T2> objects.
      // Because these join blocks operate in non-greedy mode, they do not
      // take the resource from a pool until all resources are available from
      // all pools.

      networkResources.LinkTo(joinNetworkAndMemoryResources.Target1);
      memoryResources.LinkTo(joinNetworkAndMemoryResources.Target2);

      fileResources.LinkTo(joinFileAndMemoryResources.Target1);
      memoryResources.LinkTo(joinFileAndMemoryResources.Target2);

      // Link the JoinBlock<T1, T2> objects to the ActionBlock<T> objects.

      joinNetworkAndMemoryResources.LinkTo(networkMemoryAction);
      joinFileAndMemoryResources.LinkTo(fileMemoryAction);

      // Populate the resource pools. In this example, network and
      // file resources are more abundant than memory resources.

      networkResources.Post(new NetworkResource());
      networkResources.Post(new NetworkResource());
      networkResources.Post(new NetworkResource());

      memoryResources.Post(new MemoryResource());

      fileResources.Post(new FileResource());
      fileResources.Post(new FileResource());
      fileResources.Post(new FileResource());

      // Allow data to flow through the network for several seconds.
      Thread.Sleep(10000);
   }
}

/* Sample output:
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
*/
```

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

' Demonstrates how to use non-greedy join blocks to distribute
' resources among a dataflow network.
Friend Class Program
   ' Represents a resource. A derived class might represent
   ' a limited resource such as a memory, network, or I/O
```

```
    ' device.
    Private MustInherit Class Resource
    End Class

    ' Represents a memory resource. For brevity, the details of
    ' this class are omitted.
    Private Class MemoryResource
        Inherits Resource
    End Class

    ' Represents a network resource. For brevity, the details of
    ' this class are omitted.
    Private Class NetworkResource
        Inherits Resource
    End Class

    ' Represents a file resource. For brevity, the details of
    ' this class are omitted.
    Private Class FileResource
        Inherits Resource
    End Class

    Shared Sub Main(ByVal args() As String)
        ' Create three BufferBlock<T> objects. Each object holds a different
        ' type of resource.
        Dim networkResources = New BufferBlock(Of NetworkResource)()
        Dim fileResources = New BufferBlock(Of FileResource)()
        Dim memoryResources = New BufferBlock(Of MemoryResource)()

        ' Create two non-greedy JoinBlock<T1, T2> objects.
        ' The first join works with network and memory resources;
        ' the second pool works with file and memory resources.

        Dim joinNetworkAndMemoryResources = New JoinBlock(Of NetworkResource, MemoryResource)(New
GroupingDataflowBlockOptions With {.Greedy = False})

        Dim joinFileAndMemoryResources = New JoinBlock(Of FileResource, MemoryResource)(New
GroupingDataflowBlockOptions With {.Greedy = False})

        ' Create two ActionBlock<T> objects.
        ' The first block acts on a network resource and a memory resource.
        ' The second block acts on a file resource and a memory resource.

        Dim networkMemoryAction = New ActionBlock(Of Tuple(Of NetworkResource, MemoryResource))(Sub(data)
                 ' Perform some action on the resources.
                 ' Print a message.
                 ' Simulate a lengthy operation that uses the resources.
                 ' Print a message.
                 ' Release the resources back to their respective pools.
            Console.WriteLine("Network worker: using resources...")
            Thread.Sleep(New Random().Next(500, 2000))
            Console.WriteLine("Network worker: finished using resources...")
            networkResources.Post(data.Item1)
            memoryResources.Post(data.Item2)
        End Sub)

        Dim fileMemoryAction = New ActionBlock(Of Tuple(Of FileResource, MemoryResource))(Sub(data)
                 ' Perform some action on the resources.
                 ' Print a message.
                 ' Simulate a lengthy operation that uses the resources.
                 ' Print a message.
                 ' Release the resources back to their respective pools.
            Console.WriteLine("File worker: using resources...")
            Thread.Sleep(New Random().Next(500, 2000))
            Console.WriteLine("File worker: finished using resources...")
            fileResources.Post(data.Item1)
            memoryResources.Post(data.Item2)
        End Sub)
        ' Link the resource pools to the JoinBlock<T1, T2> objects.
        ' Because these join blocks operate in non-greedy mode, they do not
        ' take the resource from a pool until all resources are available from
        ' all pools.

        networkResources.LinkTo(joinNetworkAndMemoryResources.Target1)
        memoryResources.LinkTo(joinNetworkAndMemoryResources.Target2)

        fileResources.LinkTo(joinFileAndMemoryResources.Target1)
        memoryResources.LinkTo(joinFileAndMemoryResources.Target2)

        ' Link the JoinBlock<T1, T2> objects to the ActionBlock<T> objects.

        joinNetworkAndMemoryResources.LinkTo(networkMemoryAction)
        joinFileAndMemoryResources.LinkTo(fileMemoryAction)

        ' Populate the resource pools. In this example, network and
        ' file resources are more abundant than memory resources.

        networkResources.Post(New NetworkResource())
        networkResources.Post(New NetworkResource())
        networkResources.Post(New NetworkResource())

        memoryResources.Post(New MemoryResource())

        fileResources.Post(New FileResource())
        fileResources.Post(New FileResource())
        fileResources.Post(New FileResource())

        ' Allow data to flow through the network for several seconds.
        Thread.Sleep(10000)

    End Sub

End Class

' Sample output:
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
'
```

To enable efficient use of the shared pool of `MemoryResource` objects, this example specifies a GroupingDataflowBlockOptions object that has the Greedy property set to `False` to create JoinBlock<T1,T2> objects that act in non-greedy mode. A non-greedy join block postpones all incoming messages until one is available from each source. If any of the postponed messages were accepted by another block, the join block restarts the process. Non-greedy mode enables join blocks that share one or more source blocks to make forward progress as the other blocks wait for data. In this example, if a `MemoryResource` object is added to the `memoryResources` pool, the first join block to receive its second data source can make forward progress. If this example were to use greedy mode, which is the default, one join block might take the `MemoryResource` object and wait for the second resource to become available. However, if the other join block has its second data source available, it cannot make forward progress because the `MemoryResource` object has been taken by the other join block.

Robust Programming

The use of non-greedy joins can also help you prevent deadlock in your application. In a software application, *deadlock* occurs when two or more processes each hold a resource and mutually wait for another process to release some other resource. Consider an application that defines two JoinBlock<T1,T2> objects. Both objects each read data from two shared source blocks. In greedy mode, if one join block reads from the first source and the second join block reads from the second source, the application might deadlock because both join blocks mutually wait for the other to release its resource. In non-greedy mode, each join block reads from its sources only when all data is available, and therefore, the risk of deadlock is eliminated.

See also

- Dataflow

How to: Specify the Degree of Parallelism in a Dataflow Block

5/14/2019 • 5 minutes to read • Edit Online

This document describes how to set the ExecutionDataflowBlockOptions.MaxDegreeOfParallelism property to enable an execution dataflow block to process more than one message at a time. Doing this is useful when you have a dataflow block that performs a long-running computation and can benefit from processing messages in parallel. This example uses the System.Threading.Tasks.Dataflow.ActionBlock<TInput> class to perform multiple dataflow operations concurrently; however, you can specify the maximum degree of parallelism in any of the predefined execution block types that the TPL Dataflow Library provides, ActionBlock<TInput>, System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput>, and System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput>.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Example

The following example performs two dataflow computations and prints the elapsed time that is required for each computation. The first computation specifies a maximum degree of parallelism of 1, which is the default. A maximum degree of parallelism of 1 causes the dataflow block to process messages serially. The second computation resembles the first, except that it specifies a maximum degree of parallelism that is equal to the number of available processors. This enables the dataflow block to perform multiple operations in parallel.

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to specify the maximum degree of parallelism
// when using dataflow.
class Program
{
   // Performs several computations by using dataflow and returns the elapsed
   // time required to perform the computations.
   static TimeSpan TimeDataflowComputations(int maxDegreeOfParallelism,
      int messageCount)
   {
      // Create an ActionBlock<int> that performs some work.
      var workerBlock = new ActionBlock<int>(
         // Simulate work by suspending the current thread.
         millisecondsTimeout => Thread.Sleep(millisecondsTimeout),
         // Specify a maximum degree of parallelism.
         new ExecutionDataflowBlockOptions
         {
            MaxDegreeOfParallelism = maxDegreeOfParallelism
         });

      // Compute the time that it takes for several messages to
      // flow through the dataflow block.
      // flow through the dataflow block.

      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      for (int i = 0; i < messageCount; i++)
      {
         workerBlock.Post(1000);
      }
      workerBlock.Complete();

      // Wait for all messages to propagate through the network.
      workerBlock.Completion.Wait();

      // Stop the timer and return the elapsed number of milliseconds.
      stopwatch.Stop();
      return stopwatch.Elapsed;
   }
   static void Main(string[] args)
   {
      int processorCount = Environment.ProcessorCount;
      int messageCount = processorCount;

      // Print the number of processors on this computer.
      Console.WriteLine("Processor count = {0}.", processorCount);

      TimeSpan elapsed;

      // Perform two dataflow computations and print the elapsed
      // time required for each.

      // This call specifies a maximum degree of parallelism of 1.
      // This causes the dataflow block to process messages serially.
      elapsed = TimeDataflowComputations(1, messageCount);
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " +
         "elapsed time = {2}ms.", 1, messageCount, (int)elapsed.TotalMilliseconds);

      // Perform the computations again. This time, specify the number of
      // processors as the maximum degree of parallelism. This causes
      // multiple messages to be processed in parallel.
      elapsed = TimeDataflowComputations(processorCount, messageCount);
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " +
         "elapsed time = {2}ms.", processorCount, messageCount, (int)elapsed.TotalMilliseconds);
   }
}

/* Sample output:
Processor count = 4.
Degree of parallelism = 1; message count = 4; elapsed time = 4032ms.
Degree of parallelism = 4; message count = 4; elapsed time = 1001ms.
*/
```

```
Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to specify the maximum degree of parallelism
' when using dataflow.
Friend Class Program
   ' Performs several computations by using dataflow and returns the elapsed
   ' time required to perform the computations.
   Private Shared Function TimeDataflowComputations(ByVal maxDegreeOfParallelism As Integer, ByVal
messageCount As Integer) As TimeSpan
      ' Create an ActionBlock<int> that performs some work.
      Dim workerBlock = New ActionBlock(Of Integer)(Function(millisecondsTimeout) Pause(millisecondsTimeout),
New ExecutionDataflowBlockOptions() With { .MaxDegreeOfParallelism = maxDegreeOfParallelism})
         ' Simulate work by suspending the current thread.
         ' Specify a maximum degree of parallelism.
      ' Compute the time that it takes for several messages to
      ' flow through the dataflow block.

      Dim stopwatch As New Stopwatch()
      stopwatch.Start()

      For i As Integer = 0 To messageCount - 1
         workerBlock.Post(1000)
      Next i
      workerBlock.Complete()

      ' Wait for all messages to propagate through the network.
      workerBlock.Completion.Wait()

      ' Stop the timer and return the elapsed number of milliseconds.
      stopwatch.Stop()
      Return stopwatch.Elapsed
   End Function

   Private Shared Function Pause(ByVal obj As Object)
      Thread.Sleep(obj)
      Return Nothing
   End Function
   Shared Sub Main(ByVal args() As String)
      Dim processorCount As Integer = Environment.ProcessorCount
      Dim messageCount As Integer = processorCount

      ' Print the number of processors on this computer.
      Console.WriteLine("Processor count = {0}.", processorCount)

      Dim elapsed As TimeSpan

      ' Perform two dataflow computations and print the elapsed
      ' time required for each.

      ' This call specifies a maximum degree of parallelism of 1.
      ' This causes the dataflow block to process messages serially.
      elapsed = TimeDataflowComputations(1, messageCount)
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " & "elapsed time = {2}ms.", 1,
messageCount, CInt(Fix(elapsed.TotalMilliseconds)))

      ' Perform the computations again. This time, specify the number of
      ' processors as the maximum degree of parallelism. This causes
      ' multiple messages to be processed in parallel.
      elapsed = TimeDataflowComputations(processorCount, messageCount)
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " & "elapsed time = {2}ms.",
processorCount, messageCount, CInt(Fix(elapsed.TotalMilliseconds)))
   End Sub
End Class

' Sample output:
'Processor count = 4.
'Degree of parallelism = 1; message count = 4; elapsed time = 4032ms.
'Degree of parallelism = 4; message count = 4; elapsed time = 1001ms.
'
```

Robust Programming

By default, each predefined dataflow block propagates out messages in the order in which the messages are received. Although multiple messages are processed simultaneously when you specify a maximum degree of parallelism that is greater than 1, they are still propagated out in the order in which they are received.

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 can use a lesser degree of parallelism to meet its functional requirements or to account for a lack of available system resources. A dataflow block never chooses a greater degree of parallelism than you specify.

See also

- Dataflow

How to: Specify a Task Scheduler in a Dataflow Block

4/9/2019 • 11 minutes to read • Edit Online

This document demonstrates how to associate a specific task scheduler when you use dataflow in your application. The example uses the System.Threading.Tasks.ConcurrentExclusiveSchedulerPair class in a Windows Forms application to show when reader tasks are active and when a writer task is active. It also uses the TaskScheduler.FromCurrentSynchronizationContext method to enable a dataflow block to run on the user-interface thread.

NOTE

The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

To Create the Windows Forms Application

1. Create a Visual C# or Visual Basic **Windows Forms Application** project. In the following steps, the project is named `WriterReadersWinForms`.

2. On the form designer for the main form, Form1.cs (Form1.vb for Visual Basic), add four CheckBox controls. Set the Text property to **Reader 1** for `checkBox1`, **Reader 2** for `checkBox2`, **Reader 3** for `checkBox3`, and **Writer** for `checkBox4`. Set the Enabled property for each control to `False`.

3. Add a Timer control to the form. Set the Interval property to `2500`.

Adding Dataflow Functionality

This section describes how to create the dataflow blocks that participate in the application and how to associate each one with a task scheduler.

To Add Dataflow Functionality to the Application

1. In your project, add a reference to System.Threading.Tasks.Dataflow.dll.

2. Ensure that Form1.cs (Form1.vb for Visual Basic) contains the following `using` statements (`Imports` in Visual Basic).

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

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

3. Add a BroadcastBlock<T> data member to the `Form1` class.

```
// Broadcasts values to an ActionBlock<int> object that is associated
// with each check box.
BroadcastBlock<int> broadcaster = new BroadcastBlock<int>(null);
```

```
' Broadcasts values to an ActionBlock<int> object that is associated
' with each check box.
Private broadcaster As New BroadcastBlock(Of Integer)(Nothing)
```

4. In the `Form1` constructor, after the call to `InitializeComponent`, create an ActionBlock<TInput> object that toggles the state of CheckBox objects.

```
// Create an ActionBlock<CheckBox> object that toggles the state
// of CheckBox objects.
// Specifying the current synchronization context enables the
// action to run on the user-interface thread.
var toggleCheckBox = new ActionBlock<CheckBox>(checkBox =>
{
   checkBox.Checked = !checkBox.Checked;
},
new ExecutionDataflowBlockOptions
{
   TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
});
```

```
   ' Create an ActionBlock<CheckBox> object that toggles the state
   ' of CheckBox objects.
   ' Specifying the current synchronization context enables the
   ' action to run on the user-interface thread.
Dim toggleCheckBox = New ActionBlock(Of CheckBox)(Sub(checkBox) checkBox.Checked = Not
checkBox.Checked, New ExecutionDataflowBlockOptions With {.TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext()})
```

5. In the `Form1` constructor, create a ConcurrentExclusiveSchedulerPair object and four ActionBlock<TInput> objects, one ActionBlock<TInput> object for each CheckBox object. For each ActionBlock<TInput> object, specify a ExecutionDataflowBlockOptions object that has the TaskScheduler property set to the ConcurrentScheduler property for the readers, and the ExclusiveScheduler property for the writer.

```
// Create a ConcurrentExclusiveSchedulerPair object.
// Readers will run on the concurrent part of the scheduler pair.
// The writer will run on the exclusive part of the scheduler pair.
var taskSchedulerPair = new ConcurrentExclusiveSchedulerPair();

// Create an ActionBlock<int> object for each reader CheckBox object.
// Each ActionBlock<int> object represents an action that can read
// from a resource in parallel to other readers.
// Specifying the concurrent part of the scheduler pair enables the
// reader to run in parallel to other actions that are managed by
// that scheduler.
var readerActions =
   from checkBox in new CheckBox[] {checkBox1, checkBox2, checkBox3}
   select new ActionBlock<int>(milliseconds =>
   {
      // Toggle the check box to the checked state.
      toggleCheckBox.Post(checkBox);

      // Perform the read action. For demonstration, suspend the current
      // thread to simulate a lengthy read operation.
      Thread.Sleep(milliseconds);

      // Toggle the check box to the unchecked state.
      toggleCheckBox.Post(checkBox);
   },
   new ExecutionDataflowBlockOptions
   {
      TaskScheduler = taskSchedulerPair.ConcurrentScheduler
   });

// Create an ActionBlock<int> object for the writer CheckBox object.
// This ActionBlock<int> object represents an action that writes to
// a resource, but cannot run in parallel to readers.
// Specifying the exclusive part of the scheduler pair enables the
// writer to run in exclusively with respect to other actions that are
// managed by the scheduler pair.
var writerAction = new ActionBlock<int>(milliseconds =>
{
   // Toggle the check box to the checked state.
   toggleCheckBox.Post(checkBox4);

   // Perform the write action. For demonstration, suspend the current
   // thread to simulate a lengthy write operation.
   Thread.Sleep(milliseconds);

   // Toggle the check box to the unchecked state.
   toggleCheckBox.Post(checkBox4);
},
new ExecutionDataflowBlockOptions
{
   TaskScheduler = taskSchedulerPair.ExclusiveScheduler
});

// Link the broadcaster to each reader and writer block.
// The BroadcastBlock<T> class propagates values that it
// receives to all connected targets.
foreach (var readerAction in readerActions)
{
   broadcaster.LinkTo(readerAction);
}
broadcaster.LinkTo(writerAction);
```

```
' Create a ConcurrentExclusiveSchedulerPair object.
' Readers will run on the concurrent part of the scheduler pair.
' The writer will run on the exclusive part of the scheduler pair.
Dim taskSchedulerPair = New ConcurrentExclusiveSchedulerPair()

' Create an ActionBlock<int> object for each reader CheckBox object.
' Each ActionBlock<int> object represents an action that can read
' from a resource in parallel to other readers.
' Specifying the concurrent part of the scheduler pair enables the
' reader to run in parallel to other actions that are managed by
' that scheduler.
Dim readerActions = From checkBox In New CheckBox() {checkBox1, checkBox2, checkBox3} _
    Select New ActionBlock(Of Integer)(Sub(milliseconds)
        ' Toggle the check box to the checked state.
        ' Perform the read action. For demonstration, suspend the current
        ' thread to simulate a lengthy read operation.
        ' Toggle the check box to the unchecked state.
        toggleCheckBox.Post(checkBox)
        Thread.Sleep(milliseconds)
        toggleCheckBox.Post(checkBox)
    End Sub, New ExecutionDataflowBlockOptions With {.TaskScheduler =
taskSchedulerPair.ConcurrentScheduler})

' Create an ActionBlock<int> object for the writer CheckBox object.
' This ActionBlock<int> object represents an action that writes to
' a resource, but cannot run in parallel to readers.
' Specifying the exclusive part of the scheduler pair enables the
' writer to run in exclusively with respect to other actions that are
' managed by the scheduler pair.
Dim writerAction = New ActionBlock(Of Integer)(Sub(milliseconds)
    ' Toggle the check box to the checked state.
    ' Perform the write action. For demonstration, suspend the current
    ' thread to simulate a lengthy write operation.
    ' Toggle the check box to the unchecked state.
    toggleCheckBox.Post(checkBox4)
    Thread.Sleep(milliseconds)
    toggleCheckBox.Post(checkBox4)
End Sub, New ExecutionDataflowBlockOptions With {.TaskScheduler =
taskSchedulerPair.ExclusiveScheduler})

' Link the broadcaster to each reader and writer block.
' The BroadcastBlock<T> class propagates values that it
' receives to all connected targets.
For Each readerAction In readerActions
    broadcaster.LinkTo(readerAction)
Next readerAction
broadcaster.LinkTo(writerAction)
```

6. In the `Form1` constructor, start the Timer object.

```
// Start the timer.
timer1.Start();
```

```
' Start the timer.
timer1.Start()
```

7. On the form designer for the main form, create an event handler for the Tick event for the timer.

8. Implement the Tick event for the timer.

```
// Event handler for the timer.
private void timer1_Tick(object sender, EventArgs e)
{
   // Post a value to the broadcaster. The broadcaster
   // sends this message to each target.
   broadcaster.Post(1000);
}
```

```
' Event handler for the timer.
Private Sub timer1_Tick(ByVal sender As Object, ByVal e As EventArgs) Handles timer1.Tick
    ' Post a value to the broadcaster. The broadcaster
    ' sends this message to each target.
    broadcaster.Post(1000)
End Sub
```

Because the `toggleCheckBox` dataflow block acts on the user interface, it is important that this action occur on the user-interface thread. To accomplish this, during construction this object provides a ExecutionDataflowBlockOptions object that has the TaskScheduler property set to TaskScheduler.FromCurrentSynchronizationContext. The FromCurrentSynchronizationContext method creates a TaskScheduler object that performs work on the current synchronization context. Because the `Form1` constructor is called from the user-interface thread, the action for the `toggleCheckBox` dataflow block also runs on the user-interface thread.

This example also uses the ConcurrentExclusiveSchedulerPair class to enable some dataflow blocks to act concurrently, and another dataflow block to act exclusive with respect to all other dataflow blocks that run on the same ConcurrentExclusiveSchedulerPair object. This technique is useful when multiple dataflow blocks share a resource and some require exclusive access to that resource, because it eliminates the requirement to manually synchronize access to that resource. The elimination of manual synchronization can make code more efficient.

Example

The following example shows the complete code for Form1.cs (Form1.vb for Visual Basic).

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace WriterReadersWinForms
{
   public partial class Form1 : Form
   {
      // Broadcasts values to an ActionBlock<int> object that is associated
      // with each check box.
      BroadcastBlock<int> broadcaster = new BroadcastBlock<int>(null);

      public Form1()
      {
         InitializeComponent();

         // Create an ActionBlock<CheckBox> object that toggles the state
         // of CheckBox objects.
         // Specifying the current synchronization context enables the
         // action to run on the user-interface thread.
         var toggleCheckBox = new ActionBlock<CheckBox>(checkBox =>
         {
            checkBox.Checked = !checkBox.Checked;
         },
         new ExecutionDataflowBlockOptions
         {
            TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
         });

         // Create a ConcurrentExclusiveSchedulerPair object.
         // Readers will run on the concurrent part of the scheduler pair.
         // The writer will run on the exclusive part of the scheduler pair.
         var taskSchedulerPair = new ConcurrentExclusiveSchedulerPair();

         // Create an ActionBlock<int> object for each reader CheckBox object.
         // Each ActionBlock<int> object represents an action that can read
         // from a resource in parallel to other readers.
         // Specifying the concurrent part of the scheduler pair enables the
         // reader to run in parallel to other actions that are managed by
         // that scheduler.
         var readerActions =
            from checkBox in new CheckBox[] {checkBox1, checkBox2, checkBox3}
            select new ActionBlock<int>(milliseconds =>
            {
               // Toggle the check box to the checked state.
               toggleCheckBox.Post(checkBox);

               // Perform the read action. For demonstration, suspend the current
               // thread to simulate a lengthy read operation.
               Thread.Sleep(milliseconds);

               // Toggle the check box to the unchecked state.
               toggleCheckBox.Post(checkBox);
            },
            new ExecutionDataflowBlockOptions
            {
               TaskScheduler = taskSchedulerPair.ConcurrentScheduler
            });

         // Create an ActionBlock<int> object for the writer CheckBox object.
         // This ActionBlock<int> object represents an action that writes to
         // a resource, but cannot run in parallel to readers.
         // Specifying the exclusive part of the scheduler pair enables the
         // writer to run in exclusively with respect to other actions that are
         // managed by the scheduler pair.
         var writerAction = new ActionBlock<int>(milliseconds =>
         {
            // Toggle the check box to the checked state.
            toggleCheckBox.Post(checkBox4);

            // Perform the write action. For demonstration, suspend the current
            // thread to simulate a lengthy write operation.
            Thread.Sleep(milliseconds);

            // Toggle the check box to the unchecked state.
            toggleCheckBox.Post(checkBox4);
         },
         new ExecutionDataflowBlockOptions
         {
            TaskScheduler = taskSchedulerPair.ExclusiveScheduler
         });

         // Link the broadcaster to each reader and writer block.
         // The BroadcastBlock<T> class propagates values that it
         // receives to all connected targets.
         foreach (var readerAction in readerActions)
         {
            broadcaster.LinkTo(readerAction);
         }
         broadcaster.LinkTo(writerAction);

         // Start the timer.
         timer1.Start();
         timer1.Start();
      }

      // Event handler for the timer.
      private void timer1_Tick(object sender, EventArgs e)
      {
         // Post a value to the broadcaster. The broadcaster
         // sends this message to each target.
         broadcaster.Post(1000);
      }
   }
}
```

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


Namespace WriterReadersWinForms
    Partial Public Class Form1
        Inherits Form
        ' Broadcasts values to an ActionBlock<int> object that is associated
        ' with each check box.
        Private broadcaster As New BroadcastBlock(Of Integer)(Nothing)

        Public Sub New()
            InitializeComponent()

            ' Create an ActionBlock<CheckBox> object that toggles the state
            ' of CheckBox objects.
            ' Specifying the current synchronization context enables the
            ' action to run on the user-interface thread.
         Dim toggleCheckBox = New ActionBlock(Of CheckBox)(Sub(checkBox) checkBox.Checked = Not
checkBox.Checked, New ExecutionDataflowBlockOptions With {.TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext()})

            ' Create a ConcurrentExclusiveSchedulerPair object.
            ' Readers will run on the concurrent part of the scheduler pair.
            ' The writer will run on the exclusive part of the scheduler pair.
            Dim taskSchedulerPair = New ConcurrentExclusiveSchedulerPair()

            ' Create an ActionBlock<int> object for each reader CheckBox object.
            ' Each ActionBlock<int> object represents an action that can read
            ' from a resource in parallel to other readers.
            ' Specifying the concurrent part of the scheduler pair enables the
            ' reader to run in parallel to other actions that are managed by
            ' that scheduler.
            Dim readerActions = From checkBox In New CheckBox() {checkBox1, checkBox2, checkBox3} _
                Select New ActionBlock(Of Integer)(Sub(milliseconds)
                    ' Toggle the check box to the checked state.
                    ' Perform the read action. For demonstration, suspend the current
                    ' thread to simulate a lengthy read operation.
                    ' Toggle the check box to the unchecked state.
                    toggleCheckBox.Post(checkBox)
                    Thread.Sleep(milliseconds)
                    toggleCheckBox.Post(checkBox)
                End Sub, New ExecutionDataflowBlockOptions With {.TaskScheduler =
taskSchedulerPair.ConcurrentScheduler})

            ' Create an ActionBlock<int> object for the writer CheckBox object.
            ' This ActionBlock<int> object represents an action that writes to
            ' a resource, but cannot run in parallel to readers.
            ' Specifying the exclusive part of the scheduler pair enables the
            ' writer to run in exclusively with respect to other actions that are
            ' managed by the scheduler pair.
            Dim writerAction = New ActionBlock(Of Integer)(Sub(milliseconds)
                ' Toggle the check box to the checked state.
                ' Perform the write action. For demonstration, suspend the current
                ' Perform the write action. For demonstration, suspend the current
                ' thread to simulate a lengthy write operation.
                ' Toggle the check box to the unchecked state.
                toggleCheckBox.Post(checkBox4)
                Thread.Sleep(milliseconds)
                toggleCheckBox.Post(checkBox4)
            End Sub, New ExecutionDataflowBlockOptions With {.TaskScheduler =
taskSchedulerPair.ExclusiveScheduler})

            ' Link the broadcaster to each reader and writer block.
            ' The BroadcastBlock<T> class propagates values that it
            ' receives to all connected targets.
            For Each readerAction In readerActions
                broadcaster.LinkTo(readerAction)
            Next readerAction
            broadcaster.LinkTo(writerAction)

            ' Start the timer.
            timer1.Start()
        End Sub

        ' Event handler for the timer.
        Private Sub timer1_Tick(ByVal sender As Object, ByVal e As EventArgs) Handles timer1.Tick
            ' Post a value to the broadcaster. The broadcaster
            ' sends this message to each target.
            broadcaster.Post(1000)
        End Sub
    End Class
End Namespace
```

See also

- Dataflow

Walkthrough: Using BatchBlock and BatchedJoinBlock to Improve Efficiency

9/12/2019 • 29 minutes to read • Edit Online

The TPL Dataflow Library provides the System.Threading.Tasks.Dataflow.BatchBlock<T> and System.Threading.Tasks.Dataflow.BatchedJoinBlock<T1,T2> classes so that you can receive and buffer data from one or more sources and then propagate out that buffered data as one collection. This batching mechanism is useful when you collect data from one or more sources and then process multiple data elements as a batch. For example, consider an application that uses dataflow to insert records into a database. This operation can be more efficient if multiple items are inserted at the same time instead of one at a time sequentially. This document describes how to use the BatchBlock<T> class to improve the efficiency of such database insert operations. It also describes how to use the BatchedJoinBlock<T1,T2> class to capture both the results and any exceptions that occur when the program reads from a database.

> **NOTE**
> The TPL Dataflow Library (the System.Threading.Tasks.Dataflow namespace) is not distributed with .NET. To install the System.Threading.Tasks.Dataflow namespace in Visual Studio, open your project, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `System.Threading.Tasks.Dataflow` package. Alternatively, to install it using the .NET Core CLI, run `dotnet add package System.Threading.Tasks.Dataflow`.

Prerequisites

1. Read the Join Blocks section in the Dataflow document before you start this walkthrough.

2. Ensure that you have a copy of the Northwind database, Northwind.sdf, available on your computer. This file is typically located in the folder %Program Files%\Microsoft SQL Server Compact Edition\v3.5\Samples\.

   > **IMPORTANT**
   > In some versions of Windows, you cannot connect to Northwind.sdf if Visual Studio is running in a non-administrator mode. To connect to Northwind.sdf, start Visual Studio or a Developer Command Prompt for Visual Studio in the **Run as administrator** mode.

This walkthrough contains the following sections:

- Creating the Console Application
- Defining the Employee Class
- Defining Employee Database Operations
- Adding Employee Data to the Database without Using Buffering
- Using Buffering to Add Employee Data to the Database
- Using Buffered Join to Read Employee Data from the Database
- The Complete Example

Creating the Console Application

1. In Visual Studio, create a Visual C# or Visual Basic **Console Application** project. In this document, the project is named `DataflowBatchDatabase`.

2. In your project, add a reference to System.Data.SqlServerCe.dll and a reference to System.Threading.Tasks.Dataflow.dll.

3. Ensure that Form1.cs (Form1.vb for Visual Basic) contains the following `using` (`Imports` in Visual Basic) statements.

```
using System;
using System.Collections.Generic;
using System.Data.SqlServerCe;
using System.Diagnostics;
using System.IO;
using System.Threading.Tasks.Dataflow;
```

```
Imports System.Collections.Generic
Imports System.Data.SqlServerCe
Imports System.Diagnostics
Imports System.IO
Imports System.Threading.Tasks.Dataflow
```

4. Add the following data members to the `Program` class.

```
// The number of employees to add to the database.
// TODO: Change this value to experiment with different numbers of
// employees to insert into the database.
static readonly int insertCount = 256;

// The size of a single batch of employees to add to the database.
// TODO: Change this value to experiment with different batch sizes.
static readonly int insertBatchSize = 96;

// The source database file.
// TODO: Change this value if Northwind.sdf is at a different location
// on your computer.
static readonly string sourceDatabase =
   @"C:\Program Files\Microsoft SQL Server Compact Edition\v3.5\Samples\Northwind.sdf";

// TODO: Change this value if you require a different temporary location.
static readonly string scratchDatabase =
   @"C:\Temp\Northwind.sdf";
```

```
' The number of employees to add to the database.
' TODO: Change this value to experiment with different numbers of
' employees to insert into the database.
Private Shared ReadOnly insertCount As Integer = 256

' The size of a single batch of employees to add to the database.
' TODO: Change this value to experiment with different batch sizes.
Private Shared ReadOnly insertBatchSize As Integer = 96

' The source database file.
' TODO: Change this value if Northwind.sdf is at a different location
' on your computer.
Private Shared ReadOnly sourceDatabase As String = "C:\Program Files\Microsoft SQL Server Compact
Edition\v3.5\Samples\Northwind.sdf"

' TODO: Change this value if you require a different temporary location.
Private Shared ReadOnly scratchDatabase As String = "C:\Temp\Northwind.sdf"
```

Defining the Employee Class

Add to the `Program` class the `Employee` class.

```
// Describes an employee. Each property maps to a
// column in the Employees table in the Northwind database.
// For brevity, the Employee class does not contain
// all columns from the Employees table.
class Employee
{
   public int EmployeeID { get; set; }
   public string LastName { get; set; }
   public string FirstName { get; set; }

   // A random number generator that helps tp generate
   // Employee property values.
   static Random rand = new Random(42);

   // Possible random first names.
   static readonly string[] firstNames = { "Tom", "Mike", "Ruth", "Bob", "John" };
   // Possible random last names.
   static readonly string[] lastNames = { "Jones", "Smith", "Johnson", "Walker" };

   // Creates an Employee object that contains random
   // property values.
   public static Employee Random()
   {
      return new Employee
      {
         EmployeeID = -1,
         LastName = lastNames[rand.Next() % lastNames.Length],
         FirstName = firstNames[rand.Next() % firstNames.Length]
      };
   }
}
```

```
' Describes an employee. Each property maps to a
' column in the Employees table in the Northwind database.
' For brevity, the Employee class does not contain
' all columns from the Employees table.
Private Class Employee
   Public Property EmployeeID() As Integer
   Public Property LastName() As String
   Public Property FirstName() As String

   ' A random number generator that helps tp generate
   ' Employee property values.
   Private Shared rand As New Random(42)

   ' Possible random first names.
   Private Shared ReadOnly firstNames() As String = { "Tom", "Mike", "Ruth", "Bob", "John" }
   ' Possible random last names.
   Private Shared ReadOnly lastNames() As String = { "Jones", "Smith", "Johnson", "Walker" }

   ' Creates an Employee object that contains random
   ' property values.
   Public Shared Function Random() As Employee
      Return New Employee With {.EmployeeID = -1, .LastName = lastNames(rand.Next() Mod lastNames.Length),
.FirstName = firstNames(rand.Next() Mod firstNames.Length)}
   End Function
End Class
```

The `Employee` class contains three properties, `EmployeeID`, `LastName`, and `FirstName`. These properties correspond to the `Employee ID`, `Last Name`, and `First Name` columns in the `Employees` table in the Northwind database. For this demonstration, the `Employee` class also defines the `Random` method, which creates an `Employee` object that has random values for its properties.

Defining Employee Database Operations

Add to the `Program` class the `InsertEmployees`, `GetEmployeeCount`, and `GetEmployeeID` methods.

```
// Adds new employee records to the database.
static void InsertEmployees(Employee[] employees, string connectionString)
{
   using (SqlCeConnection connection =
      new SqlCeConnection(connectionString))
   {
      try
      {
         // Create the SQL command.
         SqlCeCommand command = new SqlCeCommand(
            "INSERT INTO Employees ([Last Name], [First Name])" +
            "VALUES (@lastName, @firstName)",
            connection);

         connection.Open();
         for (int i = 0; i < employees.Length; i++)
         {
            // Set parameters.
            command.Parameters.Clear();
            command.Parameters.Add("@lastName", employees[i].LastName);
            command.Parameters.Add("@firstName", employees[i].FirstName);

            // Execute the command.
            command.ExecuteNonQuery();
         }
      }
      finally
      {
         connection.Close();
      }
   }
}

// Retrieves the number of entries in the Employees table in
// the Northwind database.
static int GetEmployeeCount(string connectionString)
{
   int result = 0;
   using (SqlCeConnection sqlConnection =
      new SqlCeConnection(connectionString))
   {
      SqlCeCommand sqlCommand = new SqlCeCommand(
         "SELECT COUNT(*) FROM Employees", sqlConnection);

      sqlConnection.Open();
      try
      {
         result = (int)sqlCommand.ExecuteScalar();
      }
      finally
      {
         sqlConnection.Close();
      }
   }
   return result;
}

// Retrieves the ID of the first employee that has the provided name.
static int GetEmployeeID(string lastName, string firstName,
   string connectionString)
{
   using (SqlCeConnection connection =
      new SqlCeConnection(connectionString))
   {
      SqlCeCommand command = new SqlCeCommand(
         string.Format(
            "SELECT [Employee ID] FROM Employees " +
            "WHERE [Last Name] = '{0}' AND [First Name] = '{1}'",
            lastName, firstName),
         connection);

      connection.Open();
      try
      {
         return (int)command.ExecuteScalar();
      }
      finally
      {
         connection.Close();
      }
   }
}
```

```
' Adds new employee records to the database.
Private Shared Sub InsertEmployees(ByVal employees() As Employee, ByVal connectionString As String)
   Using connection As New SqlCeConnection(connectionString)
      Try
         ' Create the SQL command.
         Dim command As New SqlCeCommand("INSERT INTO Employees ([Last Name], [First Name])" & "VALUES
(@lastName, @firstName)", connection)

         connection.Open()
         For i As Integer = 0 To employees.Length - 1
            ' Set parameters.
            command.Parameters.Clear()
            command.Parameters.Add("@lastName", employees(i).LastName)
            command.Parameters.Add("@firstName", employees(i).FirstName)

            ' Execute the command.
            command.ExecuteNonQuery()
         Next i
      Finally
         connection.Close()
      End Try
   End Using
End Sub

' Retrieves the number of entries in the Employees table in
' the Northwind database.
Private Shared Function GetEmployeeCount(ByVal connectionString As String) As Integer
   Dim result As Integer = 0
   Using sqlConnection As New SqlCeConnection(connectionString)
      Dim sqlCommand As New SqlCeCommand("SELECT COUNT(*) FROM Employees", sqlConnection)

      sqlConnection.Open()
      Try
         result = CInt(Fix(sqlCommand.ExecuteScalar()))
      Finally
         sqlConnection.Close()
      End Try
   End Using
   Return result
End Function

' Retrieves the ID of the first employee that has the provided name.
Private Shared Function GetEmployeeID(ByVal lastName As String, ByVal firstName As String, ByVal
connectionString As String) As Integer
   Using connection As New SqlCeConnection(connectionString)
      Dim command As New SqlCeCommand(String.Format("SELECT [Employee ID] FROM Employees " & "WHERE [Last
Name] = '{0}' AND [First Name] = '{1}'", lastName, firstName), connection)

      connection.Open()
      Try
         Return CInt(Fix(command.ExecuteScalar()))
      Finally
         connection.Close()
      End Try
   End Using
End Function
```

The `InsertEmployees` method adds new employee records to the database. The `GetEmployeeCount` method retrieves the number of entries in the `Employees` table. The `GetEmployeeID` method retrieves the identifier of the first employee that has the provided name. Each of these methods takes a connection string to the Northwind database and uses functionality in the `System.Data.SqlServerCe` namespace to communicate with the database.

Adding Employee Data to the Database Without Using Buffering

Add to the `Program` class the `AddEmployees` and `PostRandomEmployees` methods.

```
// Posts random Employee data to the provided target block.
static void PostRandomEmployees(ITargetBlock<Employee> target, int count)
{
   Console.WriteLine("Adding {0} entries to Employee table...", count);

   for (int i = 0; i < count; i++)
   {
      target.Post(Employee.Random());
   }
}

// Adds random employee data to the database by using dataflow.
static void AddEmployees(string connectionString, int count)
{
   // Create an ActionBlock<Employee> object that adds a single
   // employee entry to the database.
   var insertEmployee = new ActionBlock<Employee>(e =>
      InsertEmployees(new Employee[] { e }, connectionString));

   // Post several random Employee objects to the dataflow block.
   PostRandomEmployees(insertEmployee, count);

   // Set the dataflow block to the completed state and wait for
   // all insert operations to complete.
   insertEmployee.Complete();
   insertEmployee.Completion.Wait();
}
```

```
' Posts random Employee data to the provided target block.
Private Shared Sub PostRandomEmployees(ByVal target As ITargetBlock(Of Employee), ByVal count As Integer)
   Console.WriteLine("Adding {0} entries to Employee table...", count)

   For i As Integer = 0 To count - 1
      target.Post(Employee.Random())
   Next i
End Sub

' Adds random employee data to the database by using dataflow.
Private Shared Sub AddEmployees(ByVal connectionString As String, ByVal count As Integer)
   ' Create an ActionBlock<Employee> object that adds a single
   ' employee entry to the database.
   Dim insertEmployee = New ActionBlock(Of Employee)(Sub(e) InsertEmployees(New Employee() { e },
connectionString))

   ' Post several random Employee objects to the dataflow block.
   PostRandomEmployees(insertEmployee, count)

   ' Set the dataflow block to the completed state and wait for
   ' all insert operations to complete.
   insertEmployee.Complete()
   insertEmployee.Completion.Wait()
End Sub
```

The `AddEmployees` method adds random employee data to the database by using dataflow. It creates an ActionBlock<TInput> object that calls the `InsertEmployees` method to add an employee entry to the database. The `AddEmployees` method then calls the `PostRandomEmployees` method to post multiple `Employee` objects to the ActionBlock<TInput> object. The `AddEmployees` method then waits for all insert operations to finish.

Using Buffering to Add Employee Data to the Database

Add to the `Program` class the `AddEmployeesBatched` method.

```csharp
// Adds random employee data to the database by using dataflow.
// This method is similar to AddEmployees except that it uses batching
// to add multiple employees to the database at a time.
static void AddEmployeesBatched(string connectionString, int batchSize,
   int count)
{
   // Create a BatchBlock<Employee> that holds several Employee objects and
   // then propagates them out as an array.
   var batchEmployees = new BatchBlock<Employee>(batchSize);

   // Create an ActionBlock<Employee[]> object that adds multiple
   // employee entries to the database.
   var insertEmployees = new ActionBlock<Employee[]>(a =>
      InsertEmployees(a, connectionString));

   // Link the batch block to the action block.
   batchEmployees.LinkTo(insertEmployees);

   // When the batch block completes, set the action block also to complete.
   batchEmployees.Completion.ContinueWith(delegate { insertEmployees.Complete(); });

   // Post several random Employee objects to the batch block.
   PostRandomEmployees(batchEmployees, count);

   // Set the batch block to the completed state and wait for
   // all insert operations to complete.
   batchEmployees.Complete();
   insertEmployees.Completion.Wait();
}
```

```vb
' Adds random employee data to the database by using dataflow.
' This method is similar to AddEmployees except that it uses batching
' to add multiple employees to the database at a time.
Private Shared Sub AddEmployeesBatched(ByVal connectionString As String, ByVal batchSize As Integer, ByVal
count As Integer)
   ' Create a BatchBlock<Employee> that holds several Employee objects and
   ' then propagates them out as an array.
   Dim batchEmployees = New BatchBlock(Of Employee)(batchSize)

   ' Create an ActionBlock<Employee[]> object that adds multiple
   ' employee entries to the database.
   Dim insertEmployees = New ActionBlock(Of Employee())(Sub(a) Program.InsertEmployees(a, connectionString))

   ' Link the batch block to the action block.
   batchEmployees.LinkTo(insertEmployees)

   ' When the batch block completes, set the action block also to complete.
   batchEmployees.Completion.ContinueWith(Sub() insertEmployees.Complete())

   ' Post several random Employee objects to the batch block.
   PostRandomEmployees(batchEmployees, count)

   ' Set the batch block to the completed state and wait for
   ' all insert operations to complete.
   batchEmployees.Complete()
   insertEmployees.Completion.Wait()
End Sub
```

This method resembles `AddEmployees`, except that it also uses the BatchBlock<T> class to buffer multiple `Employee` objects before it sends those objects to the ActionBlock<TInput> object. Because the BatchBlock<T> class propagates out multiple elements as a collection, the ActionBlock<TInput> object is modified to act on an array of `Employee` objects. As in the `AddEmployees` method, `AddEmployeesBatched` calls the `PostRandomEmployees` method to post multiple `Employee` objects; however, `AddEmployeesBatched` posts these objects to the BatchBlock<T> object. The `AddEmployeesBatched` method also waits for all insert operations to finish.

Using Buffered Join to Read Employee Data from the Database

Add to the `Program` class the `GetRandomEmployees` method.

```
// Displays information about several random employees to the console.
static void GetRandomEmployees(string connectionString, int batchSize,
   int count)
{
   // Create a BatchedJoinBlock<Employee, Exception> object that holds
   // both employee and exception data.
   var selectEmployees = new BatchedJoinBlock<Employee, Exception>(batchSize);

   // Holds the total number of exceptions that occurred.
   int totalErrors = 0;

   // Create an action block that prints employee and error information
   // to the console.
   var printEmployees =
      new ActionBlock<Tuple<IList<Employee>, IList<Exception>>>(data =>
      {
         // Print information about the employees in this batch.
         Console.WriteLine("Received a batch...");
         foreach (Employee e in data.Item1)
         {
            Console.WriteLine("Last={0} First={1} ID={2}",
               e.LastName, e.FirstName, e.EmployeeID);
         }

         // Print the error count for this batch.
         Console.WriteLine("There were {0} errors in this batch...",
            data.Item2.Count);

         // Update total error count.
         totalErrors += data.Item2.Count;
      });

   // Link the batched join block to the action block.
   selectEmployees.LinkTo(printEmployees);

   // When the batched join block completes, set the action block also to complete.
   selectEmployees.Completion.ContinueWith(delegate { printEmployees.Complete(); });

   // Try to retrieve the ID for several random employees.
   Console.WriteLine("Selecting random entries from Employees table...");
   for (int i = 0; i < count; i++)
   {
      try
      {
         // Create a random employee.
         Employee e = Employee.Random();

         // Try to retrieve the ID for the employee from the database.
         e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString);

         // Post the Employee object to the Employee target of
         // the batched join block.
         selectEmployees.Target1.Post(e);
      }
      catch (NullReferenceException e)
      {
         // GetEmployeeID throws NullReferenceException when there is
         // no such employee with the given name. When this happens,
         // post the Exception object to the Exception target of
         // the batched join block.
         selectEmployees.Target2.Post(e);
      }
   }

   // Set the batched join block to the completed state and wait for
   // all retrieval operations to complete.
   selectEmployees.Complete();
   printEmployees.Completion.Wait();

   // Print the total error count.
   Console.WriteLine("Finished. There were {0} total errors.", totalErrors);
}
```

```vb
' Displays information about several random employees to the console.
Private Shared Sub GetRandomEmployees(ByVal connectionString As String, ByVal batchSize As Integer, ByVal
count As Integer)
   ' Create a BatchedJoinBlock<Employee, Exception> object that holds
   ' both employee and exception data.
   Dim selectEmployees = New BatchedJoinBlock(Of Employee, Exception)(batchSize)

   ' Holds the total number of exceptions that occurred.
   Dim totalErrors As Integer = 0

   ' Create an action block that prints employee and error information
   ' to the console.
   Dim printEmployees = New ActionBlock(Of Tuple(Of IList(Of Employee), IList(Of Exception)))(Sub(data)
         ' Print information about the employees in this batch.
         ' Print the error count for this batch.
         ' Update total error count.
       Console.WriteLine("Received a batch...")
       For Each e As Employee In data.Item1
          Console.WriteLine("Last={0} First={1} ID={2}", e.LastName, e.FirstName, e.EmployeeID)
       Next e
       Console.WriteLine("There were {0} errors in this batch...", data.Item2.Count)
       totalErrors += data.Item2.Count
   End Sub)

   ' Link the batched join block to the action block.
   selectEmployees.LinkTo(printEmployees)

   ' When the batched join block completes, set the action block also to complete.
   selectEmployees.Completion.ContinueWith(Sub() printEmployees.Complete())

   ' Try to retrieve the ID for several random employees.
   Console.WriteLine("Selecting random entries from Employees table...")
   For i As Integer = 0 To count - 1
      Try
         ' Create a random employee.
         Dim e As Employee = Employee.Random()

         ' Try to retrieve the ID for the employee from the database.
         e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString)

         ' Post the Employee object to the Employee target of
         ' the batched join block.
         selectEmployees.Target1.Post(e)
      Catch e As NullReferenceException
         ' GetEmployeeID throws NullReferenceException when there is
         ' no such employee with the given name. When this happens,
         ' post the Exception object to the Exception target of
         ' the batched join block.
         selectEmployees.Target2.Post(e)
      End Try
   Next i

   ' Set the batched join block to the completed state and wait for
   ' all retrieval operations to complete.
   selectEmployees.Complete()
   printEmployees.Completion.Wait()

   ' Print the total error count.
   Console.WriteLine("Finished. There were {0} total errors.", totalErrors)
End Sub
```

This method prints information about random employees to the console. It creates several random `Employee` objects and calls the `GetEmployeeID` method to retrieve the unique identifier for each object. Because the `GetEmployeeID` method throws an exception if there is no matching employee with the given first and last names, the `GetRandomEmployees` method uses the BatchedJoinBlock<T1,T2> class to store `Employee` objects for successful calls to `GetEmployeeID` and System.Exception objects for calls that fail. The ActionBlock<TInput> object in this example acts on a Tuple<T1,T2> object that holds a list of `Employee` objects and a list of Exception objects. The BatchedJoinBlock<T1,T2> object propagates out this data when the sum of the received `Employee` and Exception object counts equals the batch size.

The Complete Example

The following example shows the complete code. The `Main` method compares the time that is required to perform batched database insertions versus the time to perform non-batched database insertions. It also demonstrates the use of buffered join to read employee data from the database and also report errors.

```
using System;
using System.Collections.Generic;
using System.Data.SqlServerCe;
using System.Diagnostics;
using System.IO;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to use batched dataflow blocks to improve
// the performance of database operations.
namespace DataflowBatchDatabase
{
   class Program
   {
      // The number of employees to add to the database.
      // TODO: Change this value to experiment with different numbers of
      // employees to insert into the database.
      static readonly int insertCount = 256;

      // The size of a single batch of employees to add to the database.
      // TODO: Change this value to experiment with different batch sizes.
      static readonly int insertBatchSize = 96;

      // The source database file.
      // TODO: Change this value if Northwind.sdf is at a different location
      // on your computer.
      static readonly string sourceDatabase =
         @"C:\Program Files\Microsoft SQL Server Compact Edition\v3.5\Samples\Northwind.sdf";

      // TODO: Change this value if you require a different temporary location.
      static readonly string scratchDatabase =
         @"C:\Temp\Northwind.sdf";

      // Describes an employee. Each property maps to a
      // column in the Employees table in the Northwind database.
      // For brevity, the Employee class does not contain
      // all columns from the Employees table.
      class Employee
      {
         public int EmployeeID { get; set; }
         public string LastName { get; set; }
         public string FirstName { get; set; }

         // A random number generator that helps tp generate
         // Employee property values.
         static Random rand = new Random(42);

         // Possible random first names.
         static readonly string[] firstNames = { "Tom", "Mike", "Ruth", "Bob", "John" };
         // Possible random last names.
         static readonly string[] lastNames = { "Jones", "Smith", "Johnson", "Walker" };

         // Creates an Employee object that contains random
         // property values.
         public static Employee Random()
```

```
        {
            return new Employee
            {
                EmployeeID = -1,
                LastName = lastNames[rand.Next() % lastNames.Length],
                FirstName = firstNames[rand.Next() % firstNames.Length]
            };
        }
    }

    // Adds new employee records to the database.
    static void InsertEmployees(Employee[] employees, string connectionString)
    {
        using (SqlCeConnection connection =
            new SqlCeConnection(connectionString))
        {
            try
            {
                // Create the SQL command.
                SqlCeCommand command = new SqlCeCommand(
                    "INSERT INTO Employees ([Last Name], [First Name])" +
                    "VALUES (@lastName, @firstName)",
                    connection);

                connection.Open();
                for (int i = 0; i < employees.Length; i++)
                {
                    // Set parameters.
                    command.Parameters.Clear();
                    command.Parameters.Add("@lastName", employees[i].LastName);
                    command.Parameters.Add("@firstName", employees[i].FirstName);

                    // Execute the command.
                    command.ExecuteNonQuery();
                }
            }
            finally
            {
                connection.Close();
            }
        }
    }

    // Retrieves the number of entries in the Employees table in
    // the Northwind database.
    static int GetEmployeeCount(string connectionString)
    {
        int result = 0;
        using (SqlCeConnection sqlConnection =
            new SqlCeConnection(connectionString))
        {
            SqlCeCommand sqlCommand = new SqlCeCommand(
                "SELECT COUNT(*) FROM Employees", sqlConnection);

            sqlConnection.Open();
            try
            {
                result = (int)sqlCommand.ExecuteScalar();
            }
            finally
            {
                sqlConnection.Close();
            }
        }
        return result;
    }

    // Retrieves the ID of the first employee that has the provided name.
    static int GetEmployeeID(string lastName, string firstName,
      string connectionString)
   {
      using (SqlCeConnection connection =
         new SqlCeConnection(connectionString))
      {
         SqlCeCommand command = new SqlCeCommand(
            string.Format(
               "SELECT [Employee ID] FROM Employees " +
               "WHERE [Last Name] = '{0}' AND [First Name] = '{1}'",
               lastName, firstName),
            connection);

         connection.Open();
         try
         {
            return (int)command.ExecuteScalar();
         }
         finally
         {
            connection.Close();
         }
      }
   }

   // Posts random Employee data to the provided target block.
   static void PostRandomEmployees(ITargetBlock<Employee> target, int count)
   {
      Console.WriteLine("Adding {0} entries to Employee table...", count);

      for (int i = 0; i < count; i++)
      {
         target.Post(Employee.Random());
      }
   }

   // Adds random employee data to the database by using dataflow.
   static void AddEmployees(string connectionString, int count)
   {
      // Create an ActionBlock<Employee> object that adds a single
      // employee entry to the database.
      var insertEmployee = new ActionBlock<Employee>(e =>
         InsertEmployees(new Employee[] { e }, connectionString));

      // Post several random Employee objects to the dataflow block.
      PostRandomEmployees(insertEmployee, count);

      // Set the dataflow block to the completed state and wait for
      // all insert operations to complete.
      insertEmployee.Complete();
      insertEmployee.Completion.Wait();
   }

   // Adds random employee data to the database by using dataflow.
   // This method is similar to AddEmployees except that it uses batching
   // to add multiple employees to the database at a time.
   static void AddEmployeesBatched(string connectionString, int batchSize,
      int count)
   {
      // Create a BatchBlock<Employee> that holds several Employee objects and
      // then propagates them out as an array.
      var batchEmployees = new BatchBlock<Employee>(batchSize);

      // Create an ActionBlock<Employee[]> object that adds multiple
      // employee entries to the database.
      var insertEmployees = new ActionBlock<Employee[]>(a =>
         InsertEmployees(a, connectionString));

      // Link the batch block to the action block.
      batchEmployees.LinkTo(insertEmployees);
```

```
         batchEmployees.LinkTo(insertEmployees);

         // When the batch block completes, set the action block also to complete.
         batchEmployees.Completion.ContinueWith(delegate { insertEmployees.Complete(); });

         // Post several random Employee objects to the batch block.
         PostRandomEmployees(batchEmployees, count);

         // Set the batch block to the completed state and wait for
         // all insert operations to complete.
         batchEmployees.Complete();
         insertEmployees.Completion.Wait();
      }

      // Displays information about several random employees to the console.
      static void GetRandomEmployees(string connectionString, int batchSize,
         int count)
      {
         // Create a BatchedJoinBlock<Employee, Exception> object that holds
         // both employee and exception data.
         var selectEmployees = new BatchedJoinBlock<Employee, Exception>(batchSize);

         // Holds the total number of exceptions that occurred.
         int totalErrors = 0;

         // Create an action block that prints employee and error information
         // to the console.
         var printEmployees =
            new ActionBlock<Tuple<IList<Employee>, IList<Exception>>>(data =>
            {
               // Print information about the employees in this batch.
               Console.WriteLine("Received a batch...");
               foreach (Employee e in data.Item1)
               {
                  Console.WriteLine("Last={0} First={1} ID={2}",
                     e.LastName, e.FirstName, e.EmployeeID);
               }

               // Print the error count for this batch.
               Console.WriteLine("There were {0} errors in this batch...",
                  data.Item2.Count);

               // Update total error count.
               totalErrors += data.Item2.Count;
            });

         // Link the batched join block to the action block.
         selectEmployees.LinkTo(printEmployees);

         // When the batched join block completes, set the action block also to complete.
         selectEmployees.Completion.ContinueWith(delegate { printEmployees.Complete(); });

         // Try to retrieve the ID for several random employees.
         Console.WriteLine("Selecting random entries from Employees table...");
         for (int i = 0; i < count; i++)
         {
            try
            {
               // Create a random employee.
               Employee e = Employee.Random();

               // Try to retrieve the ID for the employee from the database.
               e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString);

               // Post the Employee object to the Employee target of
               // the batched join block.
               selectEmployees.Target1.Post(e);
            }
            catch (NullReferenceException e)
            {
```

```
            {
               // GetEmployeeID throws NullReferenceException when there is
               // no such employee with the given name. When this happens,
               // post the Exception object to the Exception target of
               // the batched join block.
               selectEmployees.Target2.Post(e);
            }
         }

         // Set the batched join block to the completed state and wait for
         // all retrieval operations to complete.
         selectEmployees.Complete();
         printEmployees.Completion.Wait();

         // Print the total error count.
         Console.WriteLine("Finished. There were {0} total errors.", totalErrors);
      }

      static void Main(string[] args)
      {
         // Create a connection string for accessing the database.
         // The connection string refers to the temporary database location.
         string connectionString = string.Format(@"Data Source={0}",
            scratchDatabase);

         // Create a Stopwatch object to time database insert operations.
         Stopwatch stopwatch = new Stopwatch();

         // Start with a clean database file by copying the source database to
         // the temporary location.
         File.Copy(sourceDatabase, scratchDatabase, true);

         // Demonstrate multiple insert operations without batching.
         Console.WriteLine("Demonstrating non-batched database insert operations...");
         Console.WriteLine("Original size of Employee table: {0}.",
            GetEmployeeCount(connectionString));
         stopwatch.Start();
         AddEmployees(connectionString, insertCount);
         stopwatch.Stop();
         Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
            GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds);

         Console.WriteLine();

         // Start again with a clean database file.
         File.Copy(sourceDatabase, scratchDatabase, true);

         // Demonstrate multiple insert operations, this time with batching.
         Console.WriteLine("Demonstrating batched database insert operations...");
         Console.WriteLine("Original size of Employee table: {0}.",
            GetEmployeeCount(connectionString));
         stopwatch.Restart();
         AddEmployeesBatched(connectionString, insertBatchSize, insertCount);
         stopwatch.Stop();
         Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
            GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds);

         Console.WriteLine();

         // Start again with a clean database file.
         File.Copy(sourceDatabase, scratchDatabase, true);

         // Demonstrate multiple retrieval operations with error reporting.
         Console.WriteLine("Demonstrating batched join database select operations...");
         // Add a small number of employees to the database.
         AddEmployeesBatched(connectionString, insertBatchSize, 16);
         // Query for random employees.
         GetRandomEmployees(connectionString, insertBatchSize, 10);
      }
   }
   }
}
/* Sample output:
Demonstrating non-batched database insert operations...
Original size of Employee table: 15.
Adding 256 entries to Employee table...
New size of Employee table: 271; elapsed insert time: 11035 ms.

Demonstrating batched database insert operations...
Original size of Employee table: 15.
Adding 256 entries to Employee table...
New size of Employee table: 271; elapsed insert time: 197 ms.

Demonstrating batched join database insert operations...
Adding 16 entries to Employee table...
Selecting items from Employee table...
Received a batch...
Last=Jones First=Tom ID=21
Last=Jones First=John ID=24
Last=Smith First=Tom ID=26
Last=Jones First=Tom ID=21
There were 4 errors in this batch...
Received a batch...
Last=Smith First=Tom ID=26
Last=Jones First=Mike ID=28
There were 0 errors in this batch...
Finished. There were 4 total errors.
*/
```

```
Imports System.Collections.Generic
Imports System.Data.SqlServerCe
Imports System.Diagnostics
Imports System.IO
Imports System.Threading.Tasks.Dataflow


' Demonstrates how to use batched dataflow blocks to improve
' the performance of database operations.
Namespace DataflowBatchDatabase
   Friend Class Program
      ' The number of employees to add to the database.
      ' TODO: Change this value to experiment with different numbers of
      ' employees to insert into the database.
      Private Shared ReadOnly insertCount As Integer = 256

      ' The size of a single batch of employees to add to the database.
      ' TODO: Change this value to experiment with different batch sizes.
      Private Shared ReadOnly insertBatchSize As Integer = 96

      ' The source database file.
      ' TODO: Change this value if Northwind.sdf is at a different location
      ' on your computer.
      Private Shared ReadOnly sourceDatabase As String = "C:\Program Files\Microsoft SQL Server Compact
Edition\v3.5\Samples\Northwind.sdf"

      ' TODO: Change this value if you require a different temporary location.
      Private Shared ReadOnly scratchDatabase As String = "C:\Temp\Northwind.sdf"

      ' Describes an employee. Each property maps to a
      ' column in the Employees table in the Northwind database.
      ' For brevity, the Employee class does not contain
      ' all columns from the Employees table.
      Private Class Employee
         Public Property EmployeeID() As Integer
         Public Property LastName() As String
         Public Property FirstName() As String

         ' A random number generator that helps to generate
         ' A random number generator that helps tp generate
         ' Employee property values.
         Private Shared rand As New Random(42)

         ' Possible random first names.
         Private Shared ReadOnly firstNames() As String = { "Tom", "Mike", "Ruth", "Bob", "John" }
         ' Possible random last names.
         Private Shared ReadOnly lastNames() As String = { "Jones", "Smith", "Johnson", "Walker" }

         ' Creates an Employee object that contains random
         ' property values.
         Public Shared Function Random() As Employee
            Return New Employee With {.EmployeeID = -1, .LastName = lastNames(rand.Next() Mod 
lastNames.Length), .FirstName = firstNames(rand.Next() Mod firstNames.Length)}
         End Function
      End Class

      ' Adds new employee records to the database.
      Private Shared Sub InsertEmployees(ByVal employees() As Employee, ByVal connectionString As String)
         Using connection As New SqlCeConnection(connectionString)
            Try
               ' Create the SQL command.
               Dim command As New SqlCeCommand("INSERT INTO Employees ([Last Name], [First Name])" & "VALUES 
(@lastName, @firstName)", connection)

               connection.Open()
               For i As Integer = 0 To employees.Length - 1
                  ' Set parameters.
                  command.Parameters.Clear()
                  command.Parameters.Add("@lastName", employees(i).LastName)
                  command.Parameters.Add("@firstName", employees(i).FirstName)

                  ' Execute the command.
                  command.ExecuteNonQuery()
               Next i
            Finally
               connection.Close()
            End Try
         End Using
      End Sub

      ' Retrieves the number of entries in the Employees table in
      ' the Northwind database.
      Private Shared Function GetEmployeeCount(ByVal connectionString As String) As Integer
         Dim result As Integer = 0
         Using sqlConnection As New SqlCeConnection(connectionString)
            Dim sqlCommand As New SqlCeCommand("SELECT COUNT(*) FROM Employees", sqlConnection)

            sqlConnection.Open()
            Try
               result = CInt(Fix(sqlCommand.ExecuteScalar()))
            Finally
               sqlConnection.Close()
            End Try
         End Using
         Return result
      End Function

      ' Retrieves the ID of the first employee that has the provided name.
      Private Shared Function GetEmployeeID(ByVal lastName As String, ByVal firstName As String, ByVal 
connectionString As String) As Integer
         Using connection As New SqlCeConnection(connectionString)
            Dim command As New SqlCeCommand(String.Format("SELECT [Employee ID] FROM Employees " & "WHERE 
[Last Name] = '{0}' AND [First Name] = '{1}'", lastName, firstName), connection)

            connection.Open()
            Try
               Return CInt(Fix(command.ExecuteScalar()))
            Finally
               connection.Close()
```

```
                connection.Close()
            End Try
        End Using
    End Function

    ' Posts random Employee data to the provided target block.
    Private Shared Sub PostRandomEmployees(ByVal target As ITargetBlock(Of Employee), ByVal count As 
Integer)
        Console.WriteLine("Adding {0} entries to Employee table...", count)

        For i As Integer = 0 To count - 1
            target.Post(Employee.Random())
        Next i
    End Sub

    ' Adds random employee data to the database by using dataflow.
    Private Shared Sub AddEmployees(ByVal connectionString As String, ByVal count As Integer)
        ' Create an ActionBlock<Employee> object that adds a single
        ' employee entry to the database.
        Dim insertEmployee = New ActionBlock(Of Employee)(Sub(e) InsertEmployees(New Employee() { e }, 
connectionString))

        ' Post several random Employee objects to the dataflow block.
        PostRandomEmployees(insertEmployee, count)

        ' Set the dataflow block to the completed state and wait for
        ' all insert operations to complete.
        insertEmployee.Complete()
        insertEmployee.Completion.Wait()
    End Sub

    ' Adds random employee data to the database by using dataflow.
    ' This method is similar to AddEmployees except that it uses batching
    ' to add multiple employees to the database at a time.
    Private Shared Sub AddEmployeesBatched(ByVal connectionString As String, ByVal batchSize As Integer, 
ByVal count As Integer)
        ' Create a BatchBlock<Employee> that holds several Employee objects and
        ' then propagates them out as an array.
        Dim batchEmployees = New BatchBlock(Of Employee)(batchSize)

        ' Create an ActionBlock<Employee[]> object that adds multiple
        ' employee entries to the database.
        Dim insertEmployees = New ActionBlock(Of Employee())(Sub(a) Program.InsertEmployees(a, 
connectionString))

        ' Link the batch block to the action block.
        batchEmployees.LinkTo(insertEmployees)

        ' When the batch block completes, set the action block also to complete.
        batchEmployees.Completion.ContinueWith(Sub() insertEmployees.Complete())

        ' Post several random Employee objects to the batch block.
        PostRandomEmployees(batchEmployees, count)

        ' Set the batch block to the completed state and wait for
        ' all insert operations to complete.
        batchEmployees.Complete()
        insertEmployees.Completion.Wait()
    End Sub

    ' Displays information about several random employees to the console.
    Private Shared Sub GetRandomEmployees(ByVal connectionString As String, ByVal batchSize As Integer, 
ByVal count As Integer)
        ' Create a BatchedJoinBlock<Employee, Exception> object that holds
        ' both employee and exception data.
        Dim selectEmployees = New BatchedJoinBlock(Of Employee, Exception)(batchSize)

        ' Holds the total number of exceptions that occurred.
        Dim totalErrors As Integer = 0
```

```
      ' Create an action block that prints employee and error information
      ' to the console.
      Dim printEmployees = New ActionBlock(Of Tuple(Of IList(Of Employee), IList(Of Exception)))(Sub(data)
            ' Print information about the employees in this batch.
            ' Print the error count for this batch.
            ' Update total error count.
          Console.WriteLine("Received a batch...")
          For Each e As Employee In data.Item1
              Console.WriteLine("Last={0} First={1} ID={2}", e.LastName, e.FirstName, e.EmployeeID)
          Next e
          Console.WriteLine("There were {0} errors in this batch...", data.Item2.Count)
          totalErrors += data.Item2.Count
      End Sub)

      ' Link the batched join block to the action block.
      selectEmployees.LinkTo(printEmployees)

      ' When the batched join block completes, set the action block also to complete.
      selectEmployees.Completion.ContinueWith(Sub() printEmployees.Complete())

      ' Try to retrieve the ID for several random employees.
      Console.WriteLine("Selecting random entries from Employees table...")
      For i As Integer = 0 To count - 1
         Try
            ' Create a random employee.
            Dim e As Employee = Employee.Random()

            ' Try to retrieve the ID for the employee from the database.
            e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString)

            ' Post the Employee object to the Employee target of
            ' the batched join block.
            selectEmployees.Target1.Post(e)
         Catch e As NullReferenceException
            ' GetEmployeeID throws NullReferenceException when there is
            ' no such employee with the given name. When this happens,
            ' post the Exception object to the Exception target of
            ' the batched join block.
            selectEmployees.Target2.Post(e)
         End Try
      Next i

      ' Set the batched join block to the completed state and wait for
      ' all retrieval operations to complete.
      selectEmployees.Complete()
      printEmployees.Completion.Wait()

      ' Print the total error count.
      Console.WriteLine("Finished. There were {0} total errors.", totalErrors)
   End Sub

   Shared Sub Main(ByVal args() As String)
      ' Create a connection string for accessing the database.
      ' The connection string refers to the temporary database location.
      Dim connectionString As String = String.Format("Data Source={0}", scratchDatabase)

      ' Create a Stopwatch object to time database insert operations.
      Dim stopwatch As New Stopwatch()

      ' Start with a clean database file by copying the source database to
      ' the temporary location.
      File.Copy(sourceDatabase, scratchDatabase, True)

      ' Demonstrate multiple insert operations without batching.
      Console.WriteLine("Demonstrating non-batched database insert operations...")
      Console.WriteLine("Original size of Employee table: {0}.", GetEmployeeCount(connectionString))
      stopwatch.Start()
      AddEmployees(connectionString, insertCount)
```

```
         stopwatch.Stop()
         Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds)

         Console.WriteLine()

         ' Start again with a clean database file.
         File.Copy(sourceDatabase, scratchDatabase, True)

         ' Demonstrate multiple insert operations, this time with batching.
         Console.WriteLine("Demonstrating batched database insert operations...")
         Console.WriteLine("Original size of Employee table: {0}.", GetEmployeeCount(connectionString))
         stopwatch.Restart()
         AddEmployeesBatched(connectionString, insertBatchSize, insertCount)
         stopwatch.Stop()
         Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds)

         Console.WriteLine()

         ' Start again with a clean database file.
         File.Copy(sourceDatabase, scratchDatabase, True)

         ' Demonstrate multiple retrieval operations with error reporting.
         Console.WriteLine("Demonstrating batched join database select operations...")
         ' Add a small number of employees to the database.
         AddEmployeesBatched(connectionString, insertBatchSize, 16)
         ' Query for random employees.
         GetRandomEmployees(connectionString, insertBatchSize, 10)
      End Sub
   End Class
End Namespace
' Sample output:
'Demonstrating non-batched database insert operations...
'Original size of Employee table: 15.
'Adding 256 entries to Employee table...
'New size of Employee table: 271; elapsed insert time: 11035 ms.
'
'Demonstrating batched database insert operations...
'Original size of Employee table: 15.
'Adding 256 entries to Employee table...
'New size of Employee table: 271; elapsed insert time: 197 ms.
'
'Demonstrating batched join database insert operations...
'Adding 16 entries to Employee table...
'Selecting items from Employee table...
'Received a batch...
'Last=Jones First=Tom ID=21
'Last=Jones First=John ID=24
'Last=Smith First=Tom ID=26
'Last=Jones First=Tom ID=21
'There were 4 errors in this batch...
'Received a batch...
'Last=Smith First=Tom ID=26
'Last=Jones First=Mike ID=28
'There were 0 errors in this batch...
'Finished. There were 4 total errors.
'
```

See also

- Dataflow

Using TPL with Other Asynchronous Patterns

9/6/2018 • 2 minutes to read • Edit Online

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

4/28/2019 • 16 minutes to read • Edit Online

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<TResult>.

Wrapping APM Operations in a Task

Both the System.Threading.Tasks.TaskFactory and System.Threading.Tasks.TaskFactory<TResult> classes provide several overloads of the TaskFactory.FromAsync and TaskFactory<TResult>.FromAsync methods that let you encapsulate an APM Begin/End method pair in one Task or Task<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<TResult> that create a Task<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.

```
public Task<TResult> FromAsync<TArg1, TArg2, TArg3>(
    Func<TArg1, TArg2, TArg3, AsyncCallback, object, IAsyncResult> beginMethod, //BeginRead
     Func<IAsyncResult, TResult> endMethod, //EndRead
     TArg1 arg1, // the byte[] buffer
     TArg2 arg2, // the offset in arg1 at which to start writing data
     TArg3 arg3, // the maximum number of bytes to read
     object state // optional state information
    )
```

```
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<T1,T2,T3,T4,T5,TResult> delegate that matches the signature of the FileStream.BeginRead method. The second parameter is a Func<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.

```
const int MAX_FILE_SIZE = 14000000;
public static Task<string> GetFileStringAsync(string path)
{
    FileInfo fi = new FileInfo(path);
    byte[] data = null;
    data = new byte[fi.Length];

    FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, data.Length, true);

    //Task<int> returns the number of bytes read
    Task<int> task = Task<int>.Factory.FromAsync(
            fs.BeginRead, fs.EndRead, data, 0, data.Length, null);

    // 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 task.ContinueWith((antecedent) =>
    {
        fs.Close();

        // Result = "number of bytes read" (if we need it.)
        if (antecedent.Result < 100)
        {
            return "Data is too small to bother with.";
        }
        else
        {
            // If we did not receive the entire file, the end of the
            // data buffer will contain garbage.
            if (antecedent.Result < data.Length)
                Array.Resize(ref data, antecedent.Result);

            // 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);
        }
    });
}
```

```
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.

```
 Task<string> t = GetFileStringAsync(path);

 // Do some other work:
 // ...

 try
 {
      Console.WriteLine(t.Result.Substring(0, 500));
 }
 catch (AggregateException ae)
 {
     Console.WriteLine(ae.InnerException.Message);
 }
```

```
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.

```
public Task<string> GetFileStringAsync2(string path)
{
    FileInfo fi = new FileInfo(path);
    byte[] data = new byte[fi.Length];
    MyCustomState state = GetCustomState();
    FileStream fs = 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.
    Task<int> task = Task<int>.Factory.FromAsync(
            fs.BeginRead, fs.EndRead, data, 0, data.Length, null);

    return task.ContinueWith((antecedent) =>
    {
        // It is safe to close the filestream now.
        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"))
        {
            return "Start spreading the news!";
        }
        else
        {
            // If we did not receive the entire file, the end of the
            // data buffer will contain garbage.
            if (antecedent.Result < data.Length)
                Array.Resize(ref data, antecedent.Result);

            // 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);
        }
    });

}
```

```
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.

```
public Task<string> GetMultiFileData(string[] filesToRead)
{
    FileStream fs;
    Task<string>[] tasks = new Task<string>[filesToRead.Length];
    byte[] fileData = null;
    for (int i = 0; i < filesToRead.Length; i++)
    {
        fileData = new byte[0x1000];
        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<int>.Factory.FromAsync(
                 fs.BeginRead, fs.EndRead, fileData, 0, fileData.Length, null)
                 .ContinueWith((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)
                             Array.Resize(ref fileData, antecedent.Result);

                         // 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);
                     });
    }

    // Wait for all tasks to complete.
    return Task<string>.Factory.ContinueWhenAll(tasks, (data) =>
    {
        // Propagate all exceptions and mark all faulted tasks as observed.
        Task.WaitAll(data);

        // Combine the results from all tasks.
        StringBuilder sb = new StringBuilder();
        foreach (var t in data)
        {
            sb.Append(t.Result);
        }
        // Final result to be returned eventually on the calling thread.
        return sb.ToString();
    });

}
```

```
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<TResult>.FromAsync(IAsyncResult, Func<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.

```
static Task<String> ReturnTaskFromAsyncResult()
{
    IAsyncResult ar = DoSomethingAsynchronously();
    Task<String> t = Task<string>.Factory.FromAsync(ar, _ =>
        {
            return (string)ar.AsyncState;
        });

    return t;
}
```

```
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<TResult> class, which can be used to represent any arbitrary set of operations as a Task<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<TResult> to expose a set of asynchronous WebClient operations to client code as a basic Task<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.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading;
using System.Threading.Tasks;

public class SimpleWebExample
{
  public Task<string[]> GetWordCountsSimplified(string[] urls, string name,
                                              CancellationToken token)
  {
      TaskCompletionSource<string[]> tcs = new TaskCompletionSource<string[]>();
      WebClient[] webClients = new WebClient[urls.Length];
      object m_lock = new object();
      int count = 0;
      List<string> results = new List<string>();
```

```
        // If the user cancels the CancellationToken, then we can use the
        // WebClient's ability to cancel its own async operations.
        token.Register(() =>
        {
            foreach (var wc in webClients)
            {
                if (wc != null)
                    wc.CancelAsync();
            }
        });


        for (int i = 0; i < urls.Length; i++)
        {
            webClients[i] = new WebClient();

            #region callback
            // Specify the callback for the DownloadStringCompleted
            // event that will be raised by this WebClient instance.
            webClients[i].DownloadStringCompleted += (obj, args) =>
            {

                // Argument validation and exception handling omitted for brevity.

                // Split the string into an array of words,
                // then count the number of elements that match
                // the search term.
                string[] words = args.Result.Split(' ');
                string NAME = name.ToUpper();
                int 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.
                lock (m_lock)
                {
                   results.Add(String.Format("{0} has {1} instances of {2}", args.UserState, nameCount, name));

                   // If this is the last async operation to complete,
                   // then set the Result property on the underlying Task.
                   count++;
                   if (count == urls.Length)
                   {
                      tcs.TrySetResult(results.ToArray());
                   }
                }
            };
            #endregion

            // Call DownloadStringAsync for each URL.
            Uri address = null;
            address = new Uri(urls[i]);
            webClients[i].DownloadStringAsync(address, address);

        } // end for

        // Return the underlying Task. The client code
        // waits on the Result property, and handles exceptions
        // in the try-catch block there.
        return tcs.Task;
    }
}
```

```
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<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.

```
class Calculator
{
    public IAsyncResult BeginCalculate(int decimalPlaces, AsyncCallback ac, object state)
    {
        Console.WriteLine("Calling BeginCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId);
        Task<string> f = Task<string>.Factory.StartNew(_ => Compute(decimalPlaces), state);
        if (ac != null) f.ContinueWith((res) => ac(f));
        return f;
    }

    public string Compute(int numPlaces)
    {
        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";
    }

    public string EndCalculate(IAsyncResult ar)
    {
        Console.WriteLine("Calling EndCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId);
        return ((Task<string>)ar).Result;
    }
}

public class CalculatorClient
{
    static int decimalPlaces = 12;
    public static void Main()
    {
        Calculator calc = new Calculator();
        int places = 35;

        AsyncCallback callBack = new AsyncCallback(PrintResult);
        IAsyncResult ar = calc.BeginCalculate(places, callBack, calc);

        // Do some work on this thread while the calulator is busy.
        Console.WriteLine("Working...");
        Thread.SpinWait(500000);
        Console.ReadLine();
    }

    public static void PrintResult(IAsyncResult result)
    {
        Calculator c = (Calculator)result.AsyncState;
        string piString = c.EndCalculate(result);
        Console.WriteLine("Calling PrintResult on thread {0}; result = {1}",
                    Thread.CurrentThread.ManagedThreadId, piString);
    }
}
```

```
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, 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

9/6/2018 • 5 minutes to read • Edit Online

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

Example

```
class WebDataDownloader
{

    static void Main()
    {
        WebDataDownloader downloader = new WebDataDownloader();
        string[] 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" };
        CancellationTokenSource cts = new CancellationTokenSource();

        // Create a UI thread from which to cancel the entire operation
        Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Press c to cancel");
            if (Console.ReadKey().KeyChar == 'c')
                cts.Cancel();
        });

        // Using a neutral search term that is sure to get some hits.
        Task<string[]> webTask = downloader.GetWordCounts(addresses, "the", cts.Token);

        // Do some other work here unless the method has already completed.
        if (!webTask.IsCompleted)
        {
            // Simulate some work.
            Thread.SpinWait(5000000);
        }

        string[] results = null;
        try
        {
            results = webTask.Result;
        }
        catch (AggregateException e)
        {
            foreach (var ex in e.InnerExceptions)
            {
                OperationCanceledException oce = ex as OperationCanceledException;
                if (oce != null)
                {
                    if (oce.CancellationToken == cts.Token)
                    {
                        Console.WriteLine("Operation canceled by user.");
                    }
                }
                else
                    Console.WriteLine(ex.Message);
            }
        }
        finally
        {
            cts.Dispose();
        }
        if (results != null)
        {
            foreach (var item in results)
                Console.WriteLine(item);
        }
        Console.ReadKey();
    }

    Task<string[]> GetWordCounts(string[] urls, string name, CancellationToken token)
    {
        TaskCompletionSource<string[]> tcs = new TaskCompletionSource<string[]>();
        WebClient[] webClients = new WebClient[urls.Length];

        // If the user cancels the CancellationToken, then we can use the
        // WebClient's ability to cancel its own async operations.
        token.Register(() =>
            {
                foreach (var wc in webClients)
                {
                    if (wc != null)
                        wc.CancelAsync();
                }
            });

        object m_lock = new object();
        int count = 0;
        List<string> results = new List<string>();
        for (int i = 0; i < urls.Length; i++)
        {
            webClients[i] = new WebClient();

            #region callback
            // Specify the callback for the DownloadStringCompleted
            // event that will be raised by this WebClient instance.
            webClients[i].DownloadStringCompleted += (obj, args) =>
             {
                 if (args.Cancelled == true)
                 {
                     tcs.TrySetCanceled();
                     return;
                 }
                 else if (args.Error != null)
                 {
                     // Pass through to the underlying Task
                     // any exceptions thrown by the WebClient
                     // during the asynchronous operation.
                     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.
                     string[] words = null;
                     words = args.Result.Split(' ');
                     string NAME = name.ToUpper();
                     int 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,
name));
```

```
                 }

                 // If this is the last async operation to complete,
                 // then set the Result property on the underlying Task.
                 lock (m_lock)
                 {
                     count++;
                     if (count == urls.Length)
                     {
                         tcs.TrySetResult(results.ToArray());
                     }
                 }
             };
            #endregion

            // Call DownloadStringAsync for each URL.
            Uri address = null;
            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 (UriFormatException ex)
            {
                // Abandon the entire operation if one url is malformed.
                // Other actions are possible here.
                tcs.TrySetException(ex);
                return tcs.Task;
            }
        }

        // Return the underlying Task. The client code
        // waits on the Result property, and handles exceptions
        // in the try-catch block there.
        return tcs.Task;
    }
```

```
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
```

```
          ' 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
                [illegible]
                ' 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

8/22/2019 • 7 minutes to read • Edit Online

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<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 Partition-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.

```
FileStream fs = File.OpenWrite(path);
byte[] bytes = new Byte[10000000];
// ...
Parallel.For(0, bytes.Length, (i) => fs.WriteByte(bytes[i]));
```

```
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 Specifying a synchronization context.

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 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:

```
ManualResetEventSlim mre = new ManualResetEventSlim();
Enumerable.Range(0, Environment.ProcessorCount * 100)
    .AsParallel()
    .ForAll((j) =>
        {
            if (j == Environment.ProcessorCount)
            {
                Console.WriteLine("Set on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Set();
            }
            else
            {
                Console.WriteLine("Waiting on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Wait();
            }
        }); //deadlocks
```

```
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.

```
private void button1_Click(object sender, EventArgs e)
{
    Parallel.For(0, N, i =>
    {
        // do work for i
        button1.Invoke((Action)delegate { DisplayProgress(i); });
    });
}
```

```
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.

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Factory.StartNew(() =>
        Parallel.For(0, N, i =>
        {
            // do work for i
            button1.Invoke((Action)delegate { DisplayProgress(i); });
        })
         );
}
```

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

Parallel LINQ (PLINQ)

2/8/2019 • 2 minutes to read • Edit Online

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

Introduction to PLINQ

Understanding Speedup in PLINQ

Order Preservation in PLINQ

Merge Options in PLINQ

How to: Create and Execute a Simple PLINQ Query

How to: Control Ordering in a PLINQ Query

How to: Combine Parallel and Sequential LINQ Queries

How to: Handle Exceptions in a PLINQ Query

How to: Cancel a PLINQ Query

How to: Write a Custom PLINQ Aggregate Function

How to: Specify the Execution Mode in PLINQ

How to: Specify Merge Options in PLINQ

How to: Iterate File Directories with PLINQ

How to: Measure PLINQ Query Performance

PLINQ Data Sample

See also

- ParallelEnumerable
- Parallel Programming
- Language-Integrated Query (LINQ) - C#
- Language-Integrated Query (LINQ) - Visual Basic

Introduction to PLINQ

6/4/2019 • 10 minutes to read • Edit Online

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<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<T> and arrays. This article assumes that you have a basic understanding of LINQ. For more information, see Language-Integrated Query (LINQ) - C# or Language-Integrated Query (LINQ) - Visual Basic.

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<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 AsParallel query 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 (C#) and Introduction to LINQ (Visual Basic).

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	The entry point for PLINQ. Specifies that the rest of the query should be parallelized, if it is possible.

PARALLELENUMERABLE OPERATOR	DESCRIPTION
AsSequential	Specifies that the rest of the query should be run sequentially, as a non-parallel LINQ query.
AsOrdered	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 Visual Basic) clause.
AsUnordered	Specifies that PLINQ for the rest of the query is not required to preserve the ordering of the source sequence.
WithCancellation	Specifies that PLINQ should periodically monitor the state of the provided cancellation token and cancel execution if it is requested.
WithDegreeOfParallelism	Specifies the maximum number of processors that PLINQ should use to parallelize the query.
WithMergeOptions	Provides a hint about how PLINQ should, if it is possible, merge parallel results back into just one sequence on the consuming thread.
WithExecutionMode	Specifies whether PLINQ should parallelize the query even when the default behavior would be to run it sequentially.
ForAll	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 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 extension method on the data source, as shown in the following example.

```
var source = Enumerable.Range(1, 10000);

// Opt in to PLINQ with AsParallel.
var evenNums = from num in source.AsParallel()
               where num % 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
```

```
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 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 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. You can instruct PLINQ to use no more than a specified number of processors by using the WithDegreeOfParallelism 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.

```
var query = from item in source.AsParallel().WithDegreeOfParallelism(2)
            where Compute(item) > 42
            select item;
```

```
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 operator for this purpose. AsOrdered is distinct from AsSequential. An AsOrdered sequence is still processed in parallel, but its results are buffered and sorted. Because order preservation typically involves extra work, an AsOrdered sequence might be processed more slowly than the default AsUnordered 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.

```
var evenNums = from num in numbers.AsParallel().AsOrdered()
              where num % 2 == 0
              select num;
```

```
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 method. When you use AsSequential, all subsequent operators in the query are executed sequentially until AsParallel 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 method, and the ParallelMergeOptions enumeration. For more information, see Merge Options in PLINQ.

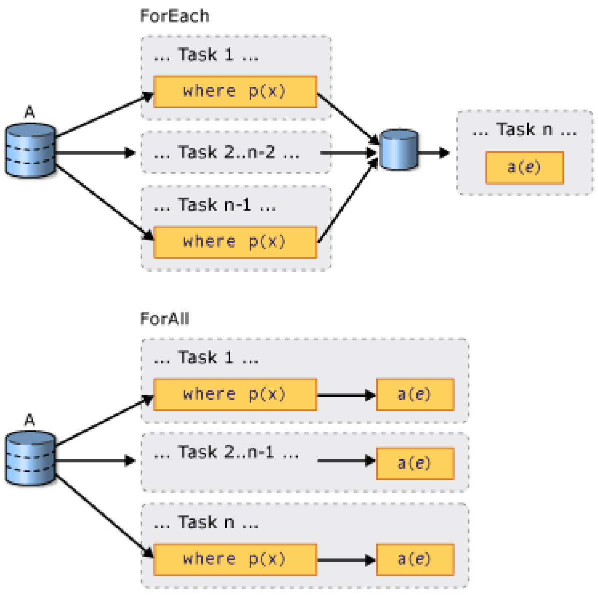
The ForAll Operator

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 , ToArray , or ToDictionary. 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 method to execute a PLINQ query. ForAll does not perform this final merge step. The following code example shows how to use the ForAll method. System.Collections.Concurrent.ConcurrentBag<T> is used here because it is optimized for multiple threads adding concurrently without attempting to remove any items.

```
var nums = Enumerable.Range(10, 10000);
var query = from num in nums.AsParallel()
            where num % 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(e => concurrentBag.Add(Compute(e)));
```

```
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 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 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.

```
int[] arr = new int[9999];
Partitioner<int> partitioner = new MyArrayPartitioner<int>(arr);
var query = partitioner.AsParallel().Select(x => SomeFunction(x));
```

```
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

Understanding Speedup in PLINQ

5/21/2019 • 6 minutes to read • Edit Online

The primary purpose of PLINQ is to speed up the execution of LINQ to Objects queries by executing the query delegates in parallel on multi-core computers. PLINQ performs best when the processing of each element in a source collection is independent, with no shared state involved among the individual delegates. Such operations are common in LINQ to Objects and PLINQ, and are often called "*delightfully parallel*" because they lend themselves easily to scheduling on multiple threads. However, not all queries consist entirely of delightfully parallel operations; in most cases, a query involves some operators that either cannot be parallelized, or that slow down parallel execution. And even with queries that are entirely delightfully parallel, PLINQ must still partition the data source and schedule the work on the threads, and usually merge the results when the query completes. All these operations add to the computational cost of parallelization; these costs of adding parallelization are called *overhead*. To achieve optimum performance in a PLINQ query, the goal is to maximize the parts that are delightfully parallel and minimize the parts that require overhead. This article provides information that will help you write PLINQ queries that are as efficient as possible while still yielding correct results.

Factors that Impact PLINQ Query Performance

The following sections lists some of the most important factors that impact parallel query performance. These are general statements that by themselves are not sufficient to predict query performance in all cases. As always, it is important to measure actual performance of specific queries on computers with a range of representative configurations and loads.

1. Computational cost of the overall work.

   To achieve speedup, a PLINQ query must have enough delightfully parallel work to offset the overhead. The work can be expressed as the computational cost of each delegate multiplied by the number of elements in the source collection. Assuming that an operation can be parallelized, the more computationally expensive it is, the greater the opportunity for speedup. For example, if a function takes one millisecond to execute, a sequential query over 1000 elements will take one second to perform that operation, whereas a parallel query on a computer with four cores might take only 250 milliseconds. This yields a speedup of 750 milliseconds. If the function required one second to execute for each element, then the speedup would be 750 seconds. If the delegate is very expensive, then PLINQ might offer significant speedup with only a few items in the source collection. Conversely, small source collections with trivial delegates are generally not good candidates for PLINQ.

   In the following example, queryA is probably a good candidate for PLINQ, assuming that its Select function involves a lot of work. queryB is probably not a good candidate because there is not enough work in the Select statement, and the overhead of parallelization will offset most or all of the speedup.

   ```
   Dim queryA = From num In numberList.AsParallel()
                Select ExpensiveFunction(num); 'good for PLINQ

   Dim queryB = From num In numberList.AsParallel()
                Where num Mod 2 > 0
                Select num; 'not as good for PLINQ
   ```

```
var queryA = from num in numberList.AsParallel()
             select ExpensiveFunction(num); //good for PLINQ

var queryB = from num in numberList.AsParallel()
             where num % 2 > 0
             select num; //not as good for PLINQ
```

2. The number of logical cores on the system (degree of parallelism).

   This point is an obvious corollary to the previous section, queries that are delightfully parallel run faster on machines with more cores because the work can be divided among more concurrent threads. The overall amount of speedup depends on what percentage of the overall work of the query is parallelizable. However, do not assume that all queries will run twice as fast on an eight core computer as a four core computer. When tuning queries for optimal performance, it is important to measure actual results on computers with various numbers of cores. This point is related to point #1: larger datasets are required to take advantage of greater computing resources.

3. The number and kind of operations.

   PLINQ provides the AsOrdered operator for situations in which it is necessary to maintain the order of elements in the source sequence. There is a cost associated with ordering, but this cost is usually modest. GroupBy and Join operations likewise incur overhead. PLINQ performs best when it is allowed to process elements in the source collection in any order, and pass them to the next operator as soon as they are ready. For more information, see Order Preservation in PLINQ.

4. The form of query execution.

   If you are storing the results of a query by calling ToArray or ToList, then the results from all parallel threads must be merged into the single data structure. This involves an unavoidable computational cost. Likewise, if you iterate the results by using a foreach (For Each in Visual Basic) loop, the results from the worker threads need to be serialized onto the enumerator thread. But if you just want to perform some action based on the result from each thread, you can use the ForAll method to perform this work on multiple threads.

5. The type of merge options.

   PLINQ can be configured to either buffer its output, and produce it in chunks or all at once after the entire result set is produced, or else to stream individual results as they are produced. The former results in decreased overall execution time and the latter results in decreased latency between yielded elements. While the merge options do not always have a major impact on overall query performance, they can impact perceived performance because they control how long a user must wait to see results. For more information, see Merge Options in PLINQ.

6. The kind of partitioning.

   In some cases, a PLINQ query over an indexable source collection may result in an unbalanced work load. When this occurs, you might be able to increase the query performance by creating a custom partitioner. For more information, see Custom Partitioners for PLINQ and TPL.

When PLINQ Chooses Sequential Mode

PLINQ will always attempt to execute a query at least as fast as the query would run sequentially. Although PLINQ does not look at how computationally expensive the user delegates are, or how big the input source is, it does look for certain query "shapes." Specifically, it looks for query operators or combinations of operators that typically cause a query to execute more slowly in parallel mode. When it finds such shapes, PLINQ by default falls back to sequential mode.

However, after measuring a specific query's performance, you may determine that it actually runs faster in parallel mode. In such cases you can use the ParallelExecutionMode.ForceParallelism flag via the WithExecutionMode method to instruct PLINQ to parallelize the query. For more information, see How to: Specify the Execution Mode in PLINQ.

The following list describes the query shapes that PLINQ by default will execute in sequential mode:

- Queries that contain a Select, indexed Where, indexed SelectMany, or ElementAt clause after an ordering or filtering operator that has removed or rearranged original indices.
- Queries that contain a Take, TakeWhile, Skip, SkipWhile operator and where indices in the source sequence are not in the original order.
- Queries that contain Zip or SequenceEquals, unless one of the data sources has an originally ordered index and the other data source is indexable (i.e. an array or IList(T)).
- Queries that contain Concat, unless it is applied to indexable data sources.
- Queries that contain Reverse, unless applied to an indexable data source.

See also

- Parallel LINQ (PLINQ)

Order Preservation in PLINQ

4/28/2019 • 5 minutes to read • Edit Online

In PLINQ, the goal is to maximize performance while maintaining correctness. A query should run as fast as possible but still produce the correct results. In some cases, correctness requires the order of the source sequence to be preserved; however, ordering can be computationally expensive. Therefore, by default, PLINQ does not preserve the order of the source sequence. In this regard, PLINQ resembles LINQ to SQL, but is unlike LINQ to Objects, which does preserve ordering.

To override the default behavior, you can turn on order-preservation by using the AsOrdered operator on the source sequence. You can then turn off order preservation later in the query by using the AsUnordered method. With both methods, the query is processed based on the heuristics that determine whether to execute the query as parallel or as sequential. For more information, see Understanding Speedup in PLINQ.

The following example shows an unordered parallel query that filters for all the elements that match a condition, without trying to order the results in any way.

```
var cityQuery = (from city in cities.AsParallel()
                 where city.Population > 10000
                 select city)
                   .Take(1000);
```

```
Dim cityQuery = From city In cities.AsParallel()
               Where City.Population > 10000
               Take (1000)
```

This query does not necessarily produce the first 1000 cities in the source sequence that meet the condition, but rather some set of 1000 cities that meet the condition. PLINQ query operators partition the source sequence into multiple subsequences that are processed as concurrent tasks. If order preservation is not specified, the results from each partition are handed off to the next stage of the query in an arbitrary order. Also, a partition may yield a subset of its results before it continues to process the remaining elements. The resulting order may be different every time. Your application cannot control this because it depends on how the operating system schedules the threads.

The following example overrides the default behavior by using the AsOrdered operator on the source sequence. This ensures that the Take method returns the first 1000 cities in the source sequence that meet the condition.

```
var orderedCities = (from city in cities.AsParallel().AsOrdered()
                     where city.Population > 10000
                     select city)
                    .Take(1000);
```

```
Dim orderedCities = From city In cities.AsParallel().AsOrdered()
                    Where City.Population > 10000
                    Take (1000)
```

However, this query probably does not run as fast as the unordered version because it must keep track of the original ordering throughout the partitions and at merge time ensure that the ordering is consistent. Therefore, we recommend that you use AsOrdered only when it is required, and only for those parts of the query that require it.

When order preservation is no longer required, use AsUnordered to turn it off. The following example achieves this by composing two queries.

```
var orderedCities2 = (from city in cities.AsParallel().AsOrdered()
                      where city.Population > 10000
                      select city)
                        .Take(1000);


var finalResult = from city in orderedCities2.AsUnordered()
                  join p in people.AsParallel() on city.Name equals p.CityName into details
                  from c in details
                  select new { Name = city.Name, Pop = city.Population, Mayor = c.Mayor };

foreach (var city in finalResult) { /*...*/ }
```

```
Dim orderedCities2 = From city In cities.AsParallel().AsOrdered()
                     Where city.Population > 10000
                     Select city
                     Take (1000)

Dim finalResult = From city In orderedCities2.AsUnordered()
                    Join p In people.AsParallel() On city.Name Equals p.CityName
                    Select New With {.Name = city.Name, .Pop = city.Population, .Mayor = city.Mayor}

For Each city In finalResult
    Console.WriteLine(city.Name & ":" & city.Pop & ":" & city.Mayor)
Next
```

Note that PLINQ preserves the ordering of a sequence produced by order-imposing operators for the rest of the query. In other words, operators such as OrderBy and ThenBy are treated as if they were followed by a call to AsOrdered.

Query Operators and Ordering

The following query operators introduce order preservation into all subsequent operations in a query, or until AsUnordered is called:

- OrderBy
- OrderByDescending
- ThenBy
- ThenByDescending

The following PLINQ query operators may in some cases require ordered source sequences to produce correct results:

- Reverse
- SequenceEqual
- TakeWhile
- SkipWhile
- Zip

Some PLINQ query operators behave differently, depending on whether their source sequence is ordered or unordered. The following table lists these operators.

OPERATOR	RESULT WHEN THE SOURCE SEQUENCE IS ORDERED	RESULT WHEN THE SOURCE SEQUENCE IS UNORDERED
Aggregate	Nondeterministic output for nonassociative or noncommutative operations	Nondeterministic output for nonassociative or noncommutative operations
All	Not applicable	Not applicable
Any	Not applicable	Not applicable
AsEnumerable	Not applicable	Not applicable
Average	Nondeterministic output for nonassociative or noncommutative operations	Nondeterministic output for nonassociative or noncommutative operations
Cast	Ordered results	Unordered results
Concat	Ordered results	Unordered results
Count	Not applicable	Not applicable
DefaultIfEmpty	Not applicable	Not applicable
Distinct	Ordered results	Unordered results
ElementAt	Return specified element	Arbitrary element
ElementAtOrDefault	Return specified element	Arbitrary element
Except	Unordered results	Unordered results
First	Return specified element	Arbitrary element
FirstOrDefault	Return specified element	Arbitrary element
ForAll	Executes nondeterministically in parallel	Executes nondeterministically in parallel
GroupBy	Ordered results	Unordered results
GroupJoin	Ordered results	Unordered results
Intersect	Ordered results	Unordered results
Join	Ordered results	Unordered results
Last	Return specified element	Arbitrary element
LastOrDefault	Return specified element	Arbitrary element
LongCount	Not applicable	Not applicable

OPERATOR	RESULT WHEN THE SOURCE SEQUENCE IS ORDERED	RESULT WHEN THE SOURCE SEQUENCE IS UNORDERED
Min	Not applicable	Not applicable
OrderBy	Reorders the sequence	Starts new ordered section
OrderByDescending	Reorders the sequence	Starts new ordered section
Range	Not applicable (same default as AsParallel)	Not applicable
Repeat	Not applicable (same default as AsParallel)	Not applicable
Reverse	Reverses	Does nothing
Select	Ordered results	Unordered results
Select (indexed)	Ordered results	Unordered results.
SelectMany	Ordered results.	Unordered results
SelectMany (indexed)	Ordered results.	Unordered results.
SequenceEqual	Ordered comparison	Unordered comparison
Single	Not applicable	Not applicable
SingleOrDefault	Not applicable	Not applicable
Skip	Skips first *n* elements	Skips any *n* elements
SkipWhile	Ordered results.	Nondeterministic. Performs SkipWhile on the current arbitrary order
Sum	Nondeterministic output for nonassociative or noncommutative operations	Nondeterministic output for nonassociative or noncommutative operations
Take	Takes first `n` elements	Takes any `n` elements
TakeWhile	Ordered results	Nondeterministic. Performs TakeWhile on the current arbitrary order
ThenBy	Supplements `OrderBy`	Supplements `OrderBy`
ThenByDescending	Supplements `OrderBy`	Supplements `OrderBy`
ToArray	Ordered results	Unordered results
ToDictionary	Not applicable	Not applicable

OPERATOR	RESULT WHEN THE SOURCE SEQUENCE IS ORDERED	RESULT WHEN THE SOURCE SEQUENCE IS UNORDERED
ToList	Ordered results	Unordered results
ToLookup	Ordered results	Unordered results
Union	Ordered results	Unordered results
Where	Ordered results	Unordered results
Where (indexed)	Ordered results	Unordered results
Zip	Ordered results	Unordered results

Unordered results are not actively shuffled; they simply do not have any special ordering logic applied to them. In some cases, an unordered query may retain the ordering of the source sequence. For queries that use the indexed Select operator, PLINQ guarantees that the output elements will come out in the order of increasing indices, but makes no guarantees about which indices will be assigned to which elements.

See also

- Parallel LINQ (PLINQ)
- Parallel Programming

Merge Options in PLINQ

4/28/2019 • 3 minutes to read • Edit Online

When a query is executing as parallel, PLINQ partitions the source sequence so that multiple threads can work on different parts concurrently, typically on separate threads. If the results are to be consumed on one thread, for example, in a `foreach` (`For Each` in Visual Basic) loop, then the results from every thread must be merged back into one sequence. The kind of merge that PLINQ performs depends on the operators that are present in the query. For example, operators that impose a new order on the results must buffer all elements from all threads. From the perspective of the consuming thread (which is also that of the application user) a fully buffered query might run for a noticeable period of time before it produces its first result. Other operators, by default, are partially buffered; they yield their results in batches. One operator, ForAll is not buffered by default. It yields all elements from all threads immediately.

By using the WithMergeOptions method, as shown in the following example, you can provide a hint to PLINQ that indicates what kind of merging to perform.

```
var scanLines = from n in nums.AsParallel()
                    .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                where n % 2 == 0
                select ExpensiveFunc(n);
```

```
Dim scanlines = From n In nums.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
                Where n Mod 2 = 0
                Select ExpensiveFunc(n)
```

For the complete example, see How to: Specify Merge Options in PLINQ.

If the particular query cannot support the requested option, then the option will just be ignored. In most cases, you do not have to specify a merge option for a PLINQ query. However, in some cases you may find by testing and measurement that a query executes best in a non-default mode. A common use of this option is to force a chunk-merging operator to stream its results in order to provide a more responsive user interface.

ParallelMergeOptions

The ParallelMergeOptions enumeration includes the following options that specify, for supported query shapes, how the final output of the query is yielded when the results are consumed on one thread:

- `Not Buffered`

  The NotBuffered option causes each processed element to be returned from each thread as soon as it is produced. This behavior is analogous to "streaming" the output. If the AsOrdered operator is present in the query, `NotBuffered` preserves the order of the source elements. Although `NotBuffered` starts yielding results as soon as they're available,, the total time to produce all the results might still be longer than using one of the other merge options.

- `Auto Buffered`

  The AutoBuffered option causes the query to collect elements into a buffer and then periodically yield the buffer contents all at once to the consuming thread. This is analogous to yielding the source data in "chunks" instead of using the "streaming" behavior of `NotBuffered`. `AutoBuffered` may take longer than

`NotBuffered` to make the first element available on the consuming thread. The size of the buffer and the exact yielding behavior are not configurable and may vary, depending on various factors that relate to the query.

- `FullyBuffered`

  The FullyBuffered option causes the output of the whole query to be buffered before any of the elements are yielded. When you use this option, it can take longer before the first element is available on the consuming thread, but the complete results might still be produced faster than by using the other options.

Query Operators that Support Merge Options

The following table lists the operators that support all merge option modes, subject to the specified restrictions.

OPERATOR	RESTRICTIONS
AsEnumerable	None
Cast	None
Concat	Non-ordered queries that have an Array or List source only.
DefaultIfEmpty	None
OfType	None
Reverse	Non-ordered queries that have an Array or List source only.
Select	None
SelectMany	None
Skip	None
Take	None
Where	None

All other PLINQ query operators might ignore user-provided merge options. Some query operators, for example, Reverse and OrderBy, cannot yield any elements until all have been produced and reordered. Therefore, when ParallelMergeOptions is used in a query that also contains an operator such as Reverse, the merge behavior will not be applied in the query until after that operator has produced its results.

The ability of some operators to handle merge options depends on the type of the source sequence, and whether the AsOrdered operator was used earlier in the query. ForAll is always NotBuffered ; it yields its elements immediately. OrderBy is always FullyBuffered; it must sort the whole list before it yields.

See also

- Parallel LINQ (PLINQ)
- How to: Specify Merge Options in PLINQ

Potential Pitfalls with PLINQ

3/8/2019 • 5 minutes to read • Edit Online

In many cases, PLINQ can provide significant performance improvements over sequential LINQ to Objects queries. However, the work of parallelizing the query execution 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 PLINQ queries.

Do Not Assume That Parallel Is Always Faster

Parallelization sometimes causes a PLINQ query to run slower than its LINQ to Objects equivalent. The basic rule of thumb is that queries that have few source elements and fast user delegates are unlikely to speedup much. However, because many factors are involved in performance, we recommend that you measure actual results before you decide whether to use PLINQ. For more information, see Understanding Speedup in PLINQ.

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 PLINQ query as much as possible.

Avoid Over-Parallelization

By using the `AsParallel` operator, 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 query.

The most common scenario in which over-parallelization can occur is in nested queries, as shown in the following snippet.

```
var q = from cust in customers.AsParallel()
        from order in cust.Orders.AsParallel()
        where order.OrderDate > date
        select new { cust, order };
```

```
Dim q = From cust In customers.AsParallel()
                From order In cust.Orders.AsParallel()
                Where order.OrderDate > aDate
                Select New With {cust, order}
```

In this case, it is best to parallelize only the outer data source (customers) unless one or more of the following conditions apply:

- The inner data source (cust.Orders) 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. For more information, see How to: Measure PLINQ Query Performance.

Avoid Calls to Non-Thread-Safe Methods

Writing to non-thread-safe instance methods from a PLINQ query 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.Write` method simultaneously, which is not supported by the class.

```
Dim fs As FileStream = File.OpenWrite(…)
a.AsParallel().Where(...).OrderBy(...).Select(...).ForAll(Sub(x) fs.Write(x))
```

```
FileStream fs = File.OpenWrite(...);
a.AsParallel().Where(...).OrderBy(...).Select(...).ForAll(x => fs.Write(x));
```

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 PLINQ queries.

Avoid Unnecessary Ordering Operations

When PLINQ executes a query in parallel, it divides the source sequence into partitions that can be operated on concurrently on multiple threads. By default, the order in which the partitions are processed and the results are delivered is not predictable (except for operators such as `OrderBy`). You can instruct PLINQ to preserve the ordering of any source sequence, but this has a negative impact on performance. The best practice, whenever possible, is to structure queries so that they do not rely on order preservation. For more information, see Order Preservation in PLINQ.

Prefer ForAll to ForEach When It Is Possible

Although PLINQ executes a query on multiple threads, if you consume the results in a `foreach` loop (`For Each` in Visual Basic), then the query results must be merged back into one thread and accessed serially by the enumerator. In some cases, this is unavoidable; however, whenever possible, use the `ForAll` method to enable each thread to output its own results, for example, by writing to a thread-safe collection such as System.Collections.Concurrent.ConcurrentBag<T>.

The same issue applies to Parallel.ForEach In other words, `source.AsParallel().Where().ForAll(...)` should be strongly preferred to

`Parallel.ForEach(source.AsParallel().Where(), ...)`.

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. If you try to access the shared state of a Windows Forms control in a PLINQ query, an exception is raised if you are running in the debugger. (This setting can be turned off.) However, if your query is consumed on the UI thread, then you can access the control from the `foreach` loop that enumerates the query results because that code executes on just one thread.

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 Parallel.For, Parallel.ForEach or ForAll 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:

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

```
ManualResetEventSlim mre = new ManualResetEventSlim();
Enumerable.Range(0, Environment.ProcessorCount * 100).AsParallel().ForAll((j) =>
{
    if (j == Environment.ProcessorCount)
    {
        Console.WriteLine("Set on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j);
        mre.Set();
    }
    else
    {
        Console.WriteLine("Waiting on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j);
        mre.Wait();
    }
}); //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.

See also

- Parallel LINQ (PLINQ)

How to: Create and Execute a Simple PLINQ Query

8/22/2019 • 2 minutes to read • Edit Online

The following example shows how to create a simple Parallel LINQ query by using the AsParallel extension method on the source sequence, and executing the query by using the ForAll method.

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.

Example

```
using System;
using System.Linq;

public class Example
{
   public static void Main()
   {
      var source = Enumerable.Range(100, 20000);

      // Result sequence might be out of order.
      var parallelQuery = from num in source.AsParallel()
                          where num % 10 == 0
                          select num;

      // Process result sequence in parallel
      parallelQuery.ForAll((e) => DoSomething(e));

      // Or use foreach to merge results first.
      foreach (var n in parallelQuery) {
         Console.WriteLine(n);
      }

      // You can also use ToArray, ToList, etc as with LINQ to Objects.
      var parallelQuery2 = (from num in source.AsParallel()
                            where num % 10 == 0
                            select num).ToArray();

      // Method syntax is also supported
      var parallelQuery3 = source.AsParallel().Where(n => n % 10 == 0).Select(n => n);

      Console.WriteLine("\nPress any key to exit...");
      Console.ReadLine();
   }

   static void DoSomething(int i) { }
}
```

```
Imports System.Linq

Module Example
   Public Sub Main()
      Dim source = Enumerable.Range(100, 20000)

      ' Result sequence might be out of order.
      Dim parallelQuery = From num In source.AsParallel()
                          Where num Mod 10 = 0
                          Select num

      ' Process result sequence in parallel
      parallelQuery.ForAll(Sub(e)
                              DoSomething(e)
                           End Sub)

      ' Or use For Each to merge results first
      ' as in this example, Where results must
      ' be serialized sequentially through static Console method.
      For Each n In parallelQuery
         Console.Write("{0} ", n)
      Next

      ' You can also use ToArray, ToList, etc, as with LINQ to Objects.
      Dim parallelQuery2 = (From num In source.AsParallel()
                            Where num Mod 10 = 0
                            Select num).ToArray()

      ' Method syntax is also supported
      Dim parallelQuery3 = source.AsParallel().Where(Function(n)
                                          Return (n Mod 10) = 0
                                          End Function).Select(Function(n)
                                          Return n
                           End Function)

      For Each i As Integer In parallelQuery3
         Console.Write("{0} ", i)
      Next


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

   ' A toy function to demonstrate syntax. Typically you need a more
   ' computationally expensive method to see speedup over sequential queries.
   Sub DoSomething(ByVal i As Integer)
      Console.Write("{0:###.## }", Math.Sqrt(i))
   End Sub
End Module
```

This example demonstrates the basic pattern for creating and executing any Parallel LINQ query when the ordering of the result sequence is not important; unordered queries are generally faster than ordered queries. The query partitions the source into tasks that are executed asynchronously on multiple threads. The order in which each task completes depends not only on the amount of work involved to process the elements in the partition, but also on external factors such as how the operating system schedules each thread. This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ. For more information about how to preserve the ordering of elements in a query, see How to: Control Ordering in a PLINQ Query.

See also

- Parallel LINQ (PLINQ)

How to: Control Ordering in a PLINQ Query

8/23/2019 • 3 minutes to read • Edit Online

These examples show how to control the ordering in a PLINQ query by using the AsOrdered extension method.

WARNING

These examples are primarily intended to demonstrate usage, and may or may not run faster than the equivalent sequential LINQ to Objects queries.

Example

The following example preserves the ordering of the source sequence. This is sometimes necessary; for example some query operators require an ordered source sequence to produce correct results.

```
var source = Enumerable.Range(9, 10000);

// Source is ordered; let's preserve it.
var parallelQuery = from num in source.AsParallel().AsOrdered()
                    where num % 3 == 0
                    select num;

// Use foreach to preserve order at execution time.
foreach (var v in parallelQuery)
    Console.Write("{0} ", v);

// Some operators expect an ordered source sequence.
var lowValues = parallelQuery.Take(10);
```

```
Sub OrderedQuery()

    Dim source = Enumerable.Range(9, 10000)

    ' Source is ordered let's preserve it.
    Dim parallelQuery = From num In source.AsParallel().AsOrdered()
                        Where num Mod 3 = 0
                        Select num

    ' Use For Each to preserve order at execution time.
    For Each item In parallelQuery
        Console.Write("{0} ", item)
    Next

    ' Some operators expect an ordered source sequence.
    Dim lowValues = parallelQuery.Take(10)

End Sub
```

Example

The following example shows some query operators whose source sequence is probably expected to be ordered. These operators will work on unordered sequences, but they might produce unexpected results.

```
// Paste into PLINQDataSample class.
static void SimpleOrdering()
{

    var customers = GetCustomers();

    // Take the first 20, preserving the original order
    var firstTwentyCustomers = customers
                                .AsParallel()
                                .AsOrdered()
                                .Take(20);

    foreach (var c in firstTwentyCustomers)
        Console.Write("{0} ", c.CustomerID);

    // All elements in reverse order.
    var reverseOrder = customers
                        .AsParallel()
                        .AsOrdered()
                        .Reverse();

    foreach (var v in reverseOrder)
        Console.Write("{0} ", v.CustomerID);

    // Get the element at a specified index.
    var cust = customers.AsParallel()
                        .AsOrdered()
                        .ElementAt(48);

    Console.WriteLine("Element #48 is: {0}", cust.CustomerID);

}
```

```
' Paste into PLINQDataSample class
Shared Sub SimpleOrdering()
    Dim customers As List(Of Customer) = GetCustomers().ToList()

    ' Take the first 20, preserving the original order

    Dim firstTwentyCustomers = customers _
                            .AsParallel() _
                            .AsOrdered() _
                            .Take(20)

    Console.WriteLine("Take the first 20 in original order")
    For Each c As Customer In firstTwentyCustomers
        Console.Write(c.CustomerID & " ")
    Next

    ' All elements in reverse order.
    Dim reverseOrder = customers _
                    .AsParallel() _
                    .AsOrdered() _
                    .Reverse()

    Console.WriteLine(vbCrLf & "Take all elements in reverse order")
    For Each c As Customer In reverseOrder
        Console.Write("{0} ", c.CustomerID)
    Next
    ' Get the element at a specified index.
    Dim cust = customers.AsParallel() _
                    .AsOrdered() _
                    .ElementAt(48)

    Console.WriteLine("Element #48 is: " & cust.CustomerID)

End Sub
```

To run this method, paste it into the PLINQDataSample class in the PLINQ Data Sample project and press F5.

Example

The following example shows how to preserve ordering for the first part of a query, then remove the ordering to increase the performance of a join clause, and then reapply ordering to the final result sequence.

```csharp
// Paste into PLINQDataSample class.
static void OrderedThenUnordered()
{

    var orders = GetOrders();
    var orderDetails = GetOrderDetails();

    var q2 = orders.AsParallel()
       .Where(o => o.OrderDate < DateTime.Parse("07/04/1997"))
       .Select(o => o)
       .OrderBy(o => o.CustomerID) // Preserve original ordering for Take operation.
       .Take(20)
       .AsUnordered()  // Remove ordering constraint to make join faster.
       .Join(
              orderDetails.AsParallel(),
              ord => ord.OrderID,
              od => od.OrderID,
              (ord, od) =>
              new
              {
                  ID = ord.OrderID,
                  Customer = ord.CustomerID,
                  Product = od.ProductID
              }
             )
       .OrderBy(i => i.Product); // Apply new ordering to final result sequence.

    foreach (var v in q2)
        Console.WriteLine("{0} {1} {2}", v.ID, v.Customer, v.Product);

}
```

```vb
' Paste into PLINQDataSample class
Sub OrderedThenUnordered()
    Dim Orders As IEnumerable(Of Order) = GetOrders()
    Dim orderDetails As IEnumerable(Of OrderDetail) = GetOrderDetails()

    ' Sometimes it's easier to create a query
    ' by composing two subqueries
    Dim query1 = From ord In Orders.AsParallel()
             Where ord.OrderDate < DateTime.Parse("07/04/1997")
             Select ord
             Order By ord.CustomerID
             Take 20

    Dim query2 = From ord In query1.AsUnordered()
             Join od In orderDetails.AsParallel() On ord.OrderID Equals od.OrderID
            Order By od.ProductID
            Select New With {ord.OrderID, ord.CustomerID, od.ProductID}


    For Each item In query2
        Console.WriteLine("{0} {1} {2}", item.OrderID, item.CustomerID, item.ProductID)
    Next
End Sub
```

To run this method, paste it into the PLINQDataSample class in the PLINQ Data Sample project and press F5.

See also

- ParallelEnumerable
- Parallel LINQ (PLINQ)

How to: Combine Parallel and Sequential LINQ Queries

8/23/2019 • 2 minutes to read • Edit Online

This example shows how to use the AsSequential method to instruct PLINQ to process all subsequent operators in the query sequentially. Although sequential processing is generally slower than parallel, sometimes it is necessary to produce correct results.

WARNING

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

The following example shows one scenario in which AsSequential is required, namely to preserve the ordering that was established in a previous clause of the query.

```
// Paste into PLINQDataSample class.
static void SequentialDemo()
{
    var orders = GetOrders();
    var query = (from ord in orders.AsParallel()
                 orderby ord.CustomerID
                 select new
                 {
                     Details = ord.OrderID,
                     Date = ord.OrderDate,
                     Shipped = ord.ShippedDate
                 }).
                    AsSequential().Take(5);
}
```

```
' Paste into PLINQDataSample class
Shared Sub SequentialDemo()

    Dim orders = GetOrders()
    Dim query = From ord In orders.AsParallel()
                 Order By ord.CustomerID
                 Select New With
                 {
                     ord.OrderID,
                     ord.OrderDate,
                     ord.ShippedDate
                 }

    Dim query2 = query.AsSequential().Take(5)

    For Each item In query2
        Console.WriteLine("{0}, {1}, {2}", item.OrderDate, item.OrderID, item.ShippedDate)
    Next
End Sub
```

Compiling the Code

To compile and run this code, paste it into the PLINQ Data Sample project, add a line to call the method from `Main`, and press F5.

See also

- Parallel LINQ (PLINQ)

How to: Handle Exceptions in a PLINQ Query

8/27/2019 • 4 minutes to read • Edit Online

The first example in this topic shows how to handle the System.AggregateException that can be thrown from a PLINQ query when it executes. The second example shows how to put try-catch blocks within delegates, as close as possible to where the exception will be thrown. In this way, you can catch them as soon as they occur and possibly continue query execution. 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.

In some cases when PLINQ falls back to sequential execution, and an exception occurs, the exception may be propagated directly, and not wrapped in an AggregateException. Also, ThreadAbortExceptions are always propagated directly.

NOTE

When "Just My Code" is enabled, Visual Studio will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the examples below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

This example shows how to put the try-catch blocks around the code that executes the query to catch any System.AggregateExceptions that are thrown.

```csharp
// Paste into PLINQDataSample class.
static void PLINQExceptions_1()
{
    // Using the raw string array here. See PLINQ Data Sample.
    string[] customers = GetCustomersAsStrings().ToArray();


    // First, we must simulate some currupt input.
    customers[54] = "###";

    var parallelQuery = from cust in customers.AsParallel()
                        let fields = cust.Split(',')
                        where fields[3].StartsWith("C") //throw indexoutofrange
                        select new { city = fields[3], thread = Thread.CurrentThread.ManagedThreadId };
    try
    {
        // We use ForAll although it doesn't really improve performance
        // since all output is serialized through the Console.
        parallelQuery.ForAll(e => Console.WriteLine("City: {0}, Thread:{1}", e.city, e.thread));
    }

    // In this design, we stop query processing when the exception occurs.
    catch (AggregateException e)
    {
        foreach (var ex in e.InnerExceptions)
        {
            Console.WriteLine(ex.Message);
            if (ex is IndexOutOfRangeException)
                Console.WriteLine("The data source is corrupt. Query stopped.");
        }
    }
}
```

```vb
' Paste into PLINQDataSample class
Shared Sub PLINQExceptions_1()

    ' Using the raw string array here. See PLINQ Data Sample.
    Dim customers As String() = GetCustomersAsStrings().ToArray()

    ' First, we must simulate some currupt input.
    customers(20) = "###"

    'throws indexoutofrange
    Dim query = From cust In customers.AsParallel() _
        Let fields = cust.Split(","c) _
        Where fields(3).StartsWith("C") _
        Select fields
    Try
        ' We use ForAll although it doesn't really improve performance
        ' since all output is serialized through the Console.
        query.ForAll(Sub(e)
                         Console.WriteLine("City: {0}, Thread:{1}")
                     End Sub)
    Catch e As AggregateException

        ' In this design, we stop query processing when the exception occurs.
        For Each ex In e.InnerExceptions
            Console.WriteLine(ex.Message)
            If TypeOf ex Is IndexOutOfRangeException Then
                Console.WriteLine("The data source is corrupt. Query stopped.")
            End If
        Next
    End Try
End Sub
```

In this example, the query cannot continue after the exception is thrown. By the time your application code catches the exception, PLINQ has already stopped the query on all threads.

Example

The following example shows how to put a try-catch block in a delegate to make it possible to catch an exception and continue with the query execution.

```
// Paste into PLINQDataSample class.
static void PLINQExceptions_2()
{

    var customers = GetCustomersAsStrings().ToArray();
    // Using the raw string array here.
    // First, we must simulate some currupt input
    customers[54] = "###";

    // Create a delegate with a lambda expression.
    // Assume that in this app, we expect malformed data
    // occasionally and by design we just report it and continue.
    Func<string[], string, bool> isTrue = (f, c) =>
    {
        try
        {
            string s = f[3];
            return s.StartsWith(c);
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine("Malformed cust: {0}", f);
            return false;
        }
    };

    // Using the raw string array here
    var parallelQuery = from cust in customers.AsParallel()
                        let fields = cust.Split(',')
                        where isTrue(fields, "C") //use a named delegate with a try-catch
                        select new { city = fields[3] };
    try
    {
        // We use ForAll although it doesn't really improve performance
        // since all output must be serialized through the Console.
        parallelQuery.ForAll(e => Console.WriteLine(e.city));
    }

    // IndexOutOfRangeException will not bubble up
    // because we handle it where it is thrown.
    catch (AggregateException e)
    {
        foreach (var ex in e.InnerExceptions)
            Console.WriteLine(ex.Message);
    }
}
```

```
' Paste into PLINQDataSample class
Shared Sub PLINQExceptions_2()

    Dim customers() = GetCustomersAsStrings().ToArray()
    ' Using the raw string array here.
    ' First, we must simulate some currupt input
    customers(20) = "###"

    ' Create a delegate with a lambda expression.
    ' Assume that in this app, we expect malformed data
    ' occasionally and by design we just report it and continue.
    Dim isTrue As Func(Of String(), String, Boolean) = Function(f, c)

                                                           Try

                                                               Dim s As String = f(3)
                                                               Return s.StartsWith(c)

                                                           Catch e As IndexOutOfRangeException

                                                               Console.WriteLine("Malformed cust: {0}", f)
                                                               Return False
                                                           End Try
                                                       End Function

    ' Using the raw string array here
    Dim query = From cust In customers.AsParallel()
                    Let fields = cust.Split(","c)
                    Where isTrue(fields, "C")
                   Select New With {.City = fields(3)}
    Try
        ' We use ForAll although it doesn't really improve performance
        ' since all output must be serialized through the Console.
        query.ForAll(Sub(e) Console.WriteLine(e.city))


        ' IndexOutOfRangeException will not bubble up
        ' because we handle it where it is thrown.
    Catch e As AggregateException
        For Each ex In e.InnerExceptions
            Console.WriteLine(ex.Message)
        Next
    End Try
End Sub
```

Compiling the Code

- To compile and run these examples, copy them into the PLINQ Data Sample example and call the method from Main.

Robust Programming

Do not catch an exception unless you know how to handle it so that you do not corrupt the state of your program.

See also

- ParallelEnumerable
- Parallel LINQ (PLINQ)

How to: Cancel a PLINQ Query

8/27/2019 • 6 minutes to read • Edit Online

The following examples show two ways to cancel a PLINQ query. The first example shows how to cancel a query that consists mostly of data traversal. The second example shows how to cancel a query that contains a user function that is computationally expensive.

NOTE

When "Just My Code" is enabled, Visual Studio will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the examples below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

```
namespace PLINQCancellation_1
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
    using static System.Console;

    class Program
    {
        static void Main(string[] args)
        {
            int[] source = Enumerable.Range(1, 10000000).ToArray();
            var cts = new CancellationTokenSource();

            // Start a new asynchronous task that will cancel the
            // operation from another thread. Typically you would call
            // Cancel() in response to a button click or some other
            // user interface event.
            Task.Factory.StartNew(() =>
            {
                UserClicksTheCancelButton(cts);
            });

            int[] results = null;
            try
            {
                results = (from num in source.AsParallel().WithCancellation(cts.Token)
                           where num % 3 == 0
                           orderby num descending
                           select num).ToArray();
            }
            catch (OperationCanceledException e)
            {
                WriteLine(e.Message);
            }
            catch (AggregateException ae)
            {
                if (ae.InnerExceptions != null)
                {
                    foreach (Exception e in ae.InnerExceptions)
                        WriteLine(e.Message);
                }
            }
            finally
            {
               cts.Dispose();
            }

            if (results != null)
            {
                foreach (var v in results)
                    WriteLine(v);
            }
            WriteLine();
            ReadKey();
        }

        static void UserClicksTheCancelButton(CancellationTokenSource cts)
        {
            // Wait between 150 and 500 ms, then cancel.
            // Adjust these values if necessary to make
            // cancellation fire while query is still executing.
            Random rand = new Random();
            Thread.Sleep(rand.Next(150, 500));
            cts.Cancel();
        }
    }
}
```

```
Class Program
    Private Shared Sub Main(ByVal args As String())
        Dim source As Integer() = Enumerable.Range(1, 10000000).ToArray()
        Dim cs As New CancellationTokenSource()

        ' Start a new asynchronous task that will cancel the
        ' operation from another thread. Typically you would call
        ' Cancel() in response to a button click or some other
        ' user interface event.
        Task.Factory.StartNew(Sub()
                                  UserClicksTheCancelButton(cs)
                              End Sub)

        Dim results As Integer() = Nothing
        Try

            results = (From num In source.AsParallel().WithCancellation(cs.Token) _
                Where num Mod 3 = 0 _
                Order By num Descending _
                Select num).ToArray()
        Catch e As OperationCanceledException

            Console.WriteLine(e.Message)
        Catch ae As AggregateException

            If ae.InnerExceptions IsNot Nothing Then
                For Each e As Exception In ae.InnerExceptions
                    Console.WriteLine(e.Message)
                Next
            End If
        Finally
            cs.Dispose()
        End Try

        If results IsNot Nothing Then
            For Each item In results
                Console.WriteLine(item)
            Next
        End If
        Console.WriteLine()

        Console.ReadKey()
    End Sub

    Private Shared Sub UserClicksTheCancelButton(ByVal cs As CancellationTokenSource)
        ' Wait between 150 and 500 ms, then cancel.
        ' Adjust these values if necessary to make
        ' cancellation fire while query is still executing.
        Dim rand As New Random()
        Thread.Sleep(rand.[Next](150, 350))
        cs.Cancel()
    End Sub
End Class
```

The PLINQ framework does not roll a single OperationCanceledException into an System.AggregateException; the OperationCanceledException must be handled in a separate catch block. If one or more user delegates throws an OperationCanceledException(externalCT) (by using an external System.Threading.CancellationToken) but no other exception, and the query was defined as `AsParallel().WithCancellation(externalCT)`, then PLINQ will issue a single OperationCanceledException (externalCT) rather than an System.AggregateException. However, if one user delegate throws an OperationCanceledException, and another delegate throws another exception type, then both exceptions will be rolled into an AggregateException.

The general guidance on cancellation is as follows:

1. If you perform user-delegate cancellation you should inform PLINQ about the external CancellationToken and throw an OperationCanceledException(externalCT).
2. If cancellation occurs and no other exceptions are thrown, then you should handle an OperationCanceledException rather than an AggregateException.

Example

The following example shows how to handle cancellation when you have a computationally expensive function in user code.

```
namespace PLINQCancellation_2
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
    using static System.Console;

    class Program
    {
        static void Main(string[] args)
        {
            int[] source = Enumerable.Range(1, 10000000).ToArray();
            var cts = new CancellationTokenSource();

            // Start a new asynchronous task that will cancel the
            // operation from another thread. Typically you would call
            // Cancel() in response to a button click or some other
            // user interface event.
            Task.Factory.StartNew(() =>
            {
                UserClicksTheCancelButton(cts);
            });

            double[] results = null;
            try
            {
                results = (from num in source.AsParallel().WithCancellation(cts.Token)
                           where num % 3 == 0
                           select Function(num, cts.Token)).ToArray();
            }
            catch (OperationCanceledException e)
            {
                WriteLine(e.Message);
            }
            catch (AggregateException ae)
            {
                if (ae.InnerExceptions != null)
                {
                    foreach (Exception e in ae.InnerExceptions)
                        WriteLine(e.Message);
                }
            }
            finally
            {
                cts.Dispose();
            }

            if (results != null)
            {
                foreach (var v in results)
                    WriteLine(v);
            }
            WriteLine();
            ReadKey();
        }

        // A toy method to simulate work.
        static double Function(int n, CancellationToken ct)
        {
            // If work is expected to take longer than 1 ms
            // then try to check cancellation status more
            // often within that work.
            for (int i = 0; i < 5; i++)
            {
                // Work hard for approx 1 millisecond.
                Thread.SpinWait(50000);

                // Check for cancellation request.
                ct.ThrowIfCancellationRequested();
            }
            // Anything will do for our purposes.
            return Math.Sqrt(n);
        }

        static void UserClicksTheCancelButton(CancellationTokenSource cts)
        {
            // Wait between 150 and 500 ms, then cancel.
            // Adjust these values if necessary to make
            // cancellation fire while query is still executing.
            Random rand = new Random();
            Thread.Sleep(rand.Next(150, 500));
            WriteLine("Press 'c' to cancel");
            if (ReadKey().KeyChar == 'c')
                cts.Cancel();
        }
    }
}
```

```
Class Program2
    Private Shared Sub Main(ByVal args As String())


        Dim source As Integer() = Enumerable.Range(1, 10000000).ToArray()
        Dim cs As New CancellationTokenSource()

        ' Start a new asynchronous task that will cancel the
        ' operation from another thread. Typically you would call
        ' Cancel() in response to a button click or some other
        ' user interface event.
        Task.Factory.StartNew(Sub()

                                  UserClicksTheCancelButton(cs)
                              End Sub)

        Dim results As Double() = Nothing
        Try

            results = (From num In source.AsParallel().WithCancellation(cs.Token) _
                Where num Mod 3 = 0 _
                Select [Function](num, cs.Token)).ToArray()
        Catch e As OperationCanceledException


            Console.WriteLine(e.Message)
        Catch ae As AggregateException
            If ae.InnerExceptions IsNot Nothing Then
                For Each e As Exception In ae.InnerExceptions
                    Console.WriteLine(e.Message)
                Next
            End If
        Finally
```

```
            cs.Dispose()
        End Try

        If results IsNot Nothing Then
            For Each item In results
                Console.WriteLine(item)
            Next
        End If
        Console.WriteLine()

        Console.ReadKey()
    End Sub

    ' A toy method to simulate work.
    Private Shared Function [Function](ByVal n As Integer, ByVal ct As CancellationToken) As Double
        ' If work is expected to take longer than 1 ms
        ' then try to check cancellation status more
        ' often within that work.
        For i As Integer = 0 To 4
            ' Work hard for approx 1 millisecond.
            Thread.SpinWait(50000)

            ' Check for cancellation request.
            If ct.IsCancellationRequested Then
                Throw New OperationCanceledException(ct)
            End If
        Next
        ' Anything will do for our purposes.
        Return Math.Sqrt(n)
    End Function

    Private Shared Sub UserClicksTheCancelButton(ByVal cs As CancellationTokenSource)
        ' Wait between 150 and 500 ms, then cancel.
        ' Adjust these values if necessary to make
        ' cancellation fire while query is still executing.
        Dim rand As New Random()
        Thread.Sleep(rand.[Next](150, 350))
        Console.WriteLine("Press 'c' to cancel")
        If Console.ReadKey().KeyChar = "c"c Then
            cs.Cancel()

        End If
    End Sub
End Class
```

When you handle the cancellation in user code, you do not have to use WithCancellation in the query definition. However, we recommended that you do this because WithCancellation has no effect on query performance and it enables the cancellation to be handled by query operators and your user code.

In order to ensure system responsiveness, we recommend that you check for cancellation around once per millisecond; however, any period up to 10 milliseconds is considered acceptable. This frequency should not have a negative impact on your code's performance.

When an enumerator is disposed, for example when code breaks out of a foreach (For Each in Visual Basic) loop that is iterating over query results, then the query is cancelled, but no exception is thrown.

See also

- ParallelEnumerable
- Parallel LINQ (PLINQ)
- Cancellation in Managed Threads

How to: Write a Custom PLINQ Aggregate Function

8/23/2019 • 2 minutes to read • Edit Online

This example shows how to use the Aggregate method to apply a custom aggregation function to a source sequence.

WARNING

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

The following example calculates the standard deviation of a sequence of integers.

```
namespace PLINQAggregation
{
    using System;
    using System.Linq;

    class aggregation
    {
        static void Main(string[] args)
        {

            // Create a data source for demonstration purposes.
            int[] source = new int[100000];
            Random rand = new Random();
            for (int x = 0; x < source.Length; x++)
            {
                // Should result in a mean of approximately 15.0.
                source[x] = rand.Next(10, 20);
            }

            // Standard deviation calculation requires that we first
            // calculate the mean average. Average is a predefined
            // aggregation operator, along with Max, Min and Count.
            double mean = source.AsParallel().Average();


            // We use the overload that is unique to ParallelEnumerable. The
            // third Func parameter combines the results from each thread.
            double standardDev = source.AsParallel().Aggregate(
                // initialize subtotal. Use decimal point to tell
                // the compiler this is a type double. Can also use: 0d.
                0.0,

                // do this on each thread
                 (subtotal, item) => subtotal + Math.Pow((item - mean), 2),

                 // aggregate results after all threads are done.
                 (total, thisThread) => total + thisThread,

                // perform standard deviation calc on the aggregated result.
                (finalSum) => Math.Sqrt((finalSum / (source.Length - 1)))
            );
            Console.WriteLine("Mean value is = {0}", mean);
            Console.WriteLine("Standard deviation is {0}", standardDev);
            Console.ReadLine();

        }
    }
}
```

```
Class aggregation
    Private Shared Sub Main(ByVal args As String())

        ' Create a data source for demonstration purposes.
        Dim source As Integer() = New Integer(99999) {}
        Dim rand As New Random()
        For x As Integer = 0 To source.Length - 1
            ' Should result in a mean of approximately 15.0.
            source(x) = rand.[Next](10, 20)
        Next

        ' Standard deviation calculation requires that we first
        ' calculate the mean average. Average is a predefined
        ' aggregation operator, along with Max, Min and Count.
        Dim mean As Double = source.AsParallel().Average()


        ' We use the overload that is unique to ParallelEnumerable. The
        ' third Func parameter combines the results from each thread.
        ' initialize subtotal. Use decimal point to tell
        ' the compiler this is a type double. Can also use: 0d.

        ' do this on each thread

        ' aggregate results after all threads are done.

        ' perform standard deviation calc on the aggregated result.
        Dim standardDev As Double = source.AsParallel().Aggregate(0.0R, Function(subtotal, item) subtotal +
Math.Pow((item - mean), 2), Function(total, thisThread) total + thisThread, Function(finalSum)
Math.Sqrt((finalSum / (source.Length - 1))))
        Console.WriteLine("Mean value is = {0}", mean)
        Console.WriteLine("Standard deviation is {0}", standardDev)

        Console.ReadLine()
    End Sub
End Class
```

This example uses an overload of the Aggregate standard query operator that is unique to PLINQ. This overload takes an extra System.Func<T1,T2,TResult> as the third input parameter. This delegate combines the results from all threads before it performs the final calculation on the aggregated results. In this example we add together the sums from all the threads.

Note that when a lambda expression body consists of a single expression, the return value of the System.Func<T,TResult> delegate is the value of the expression.

See also

- ParallelEnumerable
- Parallel LINQ (PLINQ)

How to: Specify the Execution Mode in PLINQ

8/23/2019 • 2 minutes to read • Edit Online

This example shows how to force PLINQ to bypass its default heuristics and parallelize a query regardless of the query's shape.

WARNING

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

```
// Paste into PLINQDataSample class.
static void ForceParallel()
{
    var customers = GetCustomers();
    var parallelQuery = (from cust in customers.AsParallel()
                            .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                         where cust.City == "Berlin"
                         select cust.CustomerName)
                        .ToList();
}
```

```
Private Shared Sub ForceParallel()
    Dim customers = GetCustomers()
    Dim parallelQuery = (From cust In
customers.AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism) _
        Where cust.City = "Berlin" _
        Select cust.CustomerName).ToList()
End Sub
```

PLINQ is designed to exploit opportunities for parallelization. However, not all queries benefit from parallel execution. For example, when a query contains a single user delegate that does very little work, the query will usually run faster sequentially. This is because the overhead involved in enabling parallelizing execution is more expensive than the speedup that is obtained. Therefore, PLINQ does not automatically parallelize every query. It first examines the shape of the query and the various operators that comprise it. Based on this analysis, PLINQ in the default execution mode may decide to execute some or all of the query sequentially. However, in some cases you may know more about your query than PLINQ is able to determine from its analysis. For example, you may know that a delegate is very expensive, and that the query will definitely benefit from parallelization. In such cases, you can use the WithExecutionMode method and specify the ForceParallelism value to instruct PLINQ to always run the query as parallel.

Compiling the Code

Cut and paste this code into the PLINQ Data Sample and call the method from `Main`.

See also

- AsSequential

- Parallel LINQ (PLINQ)

How to: Specify Merge Options in PLINQ

8/23/2019 • 2 minutes to read • Edit Online

This example shows how to specify the merge options that will apply to all subsequent operators in a PLINQ query. You do not have to set merge options explicitly, but doing so may improve performance. For more information about merge options, see Merge Options in PLINQ.

WARNING

This example is intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

The following example demonstrates the behavior of merge options in a basic scenario that has an unordered source and applies an expensive function to every element.

```
namespace MergeOptions
{
    using System;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading;

    class Program
    {
        static void Main(string[] args)
        {

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

            // Replace NotBuffered with AutoBuffered
            // or FullyBuffered to compare behavior.
            var scanLines = from n in nums.AsParallel()
                                .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                            where n % 2 == 0
                            select ExpensiveFunc(n);

            Stopwatch sw = Stopwatch.StartNew();
            foreach (var line in scanLines)
            {
                Console.WriteLine(line);
            }

            Console.WriteLine("Elapsed time: {0} ms. Press any key to exit.",
                            sw.ElapsedMilliseconds);
            Console.ReadKey();
        }

        // A function that demonstrates what a fly
        // sees when it watches television :-)
        static string ExpensiveFunc(int i)
        {
            Thread.SpinWait(2000000);
            return String.Format("{0} *****************************************", i);
        }
    }
}
```

```
Class MergeOptions2


    Sub DoMergeOptions()

        Dim nums = Enumerable.Range(1, 10000)

        ' Replace NotBuffered with AutoBuffered
        ' or FullyBuffered to compare behavior.
        Dim scanLines = From n In nums.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
              Where n Mod 2 = 0
              Select ExpensiveFunc(n)

        Dim sw = System.Diagnostics.Stopwatch.StartNew()
        For Each line In scanLines
            Console.WriteLine(line)
        Next

        Console.WriteLine("Elapsed time: {0} ms. Press any key to exit.")
        Console.ReadKey()

    End Sub
    ' A function that demonstrates what a fly
    ' sees when it watches television :-)
    Function ExpensiveFunc(ByVal i As Integer) As String
        System.Threading.Thread.SpinWait(2000000)
        Return String.Format("{0} *****************************************", i)
    End Function
End Class
```

In cases where the AutoBuffered option incurs an undesirable latency before the first element is yielded, try the NotBuffered option to yield result elements faster and more smoothly.

See also

- ParallelMergeOptions
- Parallel LINQ (PLINQ)

How to: Iterate File Directories with PLINQ

9/12/2019 • 3 minutes to read • Edit Online

This example shows two simple ways to parallelize operations on file directories. The first query uses the GetFiles method to populate an array of file names in a directory and all subdirectories. This method does not return until the entire array is populated, and therefore it can introduce latency at the beginning of the operation. However, after the array is populated, PLINQ can process it in parallel very quickly.

The second query uses the static EnumerateDirectories and EnumerateFiles methods which begin returning results immediately. This approach can be faster when you are iterating over large directory trees, although the processing time compared to the first example can depend on many factors.

WARNING

These examples are intended to demonstrate usage, and might not run faster than the equivalent sequential LINQ to Objects query. For more information about speedup, see Understanding Speedup in PLINQ.

Example

The following example shows how to iterate over file directories in simple scenarios when you have access to all directories in the tree, the file sizes are not very large, and the access times are not significant. This approach involves a period of latency at the beginning while the array of file names is being constructed.

```
struct FileResult
{
    public string Text;
    public string FileName;
}
// Use Directory.GetFiles to get the source sequence of file names.
public static void FileIteration_1(string path)
{
    var sw = Stopwatch.StartNew();
    int count = 0;
    string[] files = null;
    try
    {
        files = Directory.GetFiles(path, "*.*", SearchOption.AllDirectories);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine("You do not have permission to access one or more folders in this directory tree.");
        return;
    }

    catch (FileNotFoundException)
    {
        Console.WriteLine("The specified directory {0} was not found.", path);
    }

    var fileContents = from file in files.AsParallel()
            let extension = Path.GetExtension(file)
            where extension == ".txt" || extension == ".htm"
            let text = File.ReadAllText(file)
            select new FileResult { Text = text , FileName = file }; //Or ReadAllBytes, ReadAllLines, etc.

    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine(Path.GetFileName(item.FileName) + ":" + item.Text.Length);
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle((ex) =>
            {
                if (ex is UnauthorizedAccessException)
                {
                   Console.WriteLine(ex.Message);
                   return true;
                }
                return false;
            });
    }

    Console.WriteLine("FileIteration_1 processed {0} files in {1} milliseconds", count,
sw.ElapsedMilliseconds);
    }
```

Example

The following example shows how to iterate over file directories in simple scenarios when you have access to all directories in the tree, the file sizes are not very large, and the access times are not significant. This approach begins producing results faster than the previous example.

```
struct FileResult
{
    public string Text;
    public string FileName;
}

// Use Directory.EnumerateDirectories and EnumerateFiles to get the source sequence of file names.
public static void FileIteration_2(string path) //225512 ms
{
    var count = 0;
    var sw = Stopwatch.StartNew();
    var fileNames = from dir in Directory.EnumerateFiles(path, "*.*", SearchOption.AllDirectories)
                    select dir;


    var fileContents = from file in fileNames.AsParallel() // Use AsOrdered to preserve source ordering
                       let extension = Path.GetExtension(file)
                       where extension == ".txt" || extension == ".htm"
                       let Text = File.ReadAllText(file)
                       select new { Text, FileName = file }; //Or ReadAllBytes, ReadAllLines, etc.
    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine(Path.GetFileName(item.FileName) + ":" + item.Text.Length);
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle((ex) =>
            {
                if (ex is UnauthorizedAccessException)
                {
                   Console.WriteLine(ex.Message);
                   return true;
                }
                return false;
            });
    }

    Console.WriteLine("FileIteration_2 processed {0} files in {1} milliseconds", count,
sw.ElapsedMilliseconds);
}
```

When using GetFiles, be sure that you have sufficient permissions on all directories in the tree. Otherwise an exception will be thrown and no results will be returned. When using the EnumerateDirectories in a PLINQ query, it is problematic to handle I/O exceptions in a graceful way that enables you to continue iterating. If your code must handle I/O or unauthorized access exceptions, then you should consider the approach described in How to: Iterate File Directories with the Parallel Class.

If I/O latency is an issue, for example with file I/O over a network, consider using one of the asynchronous I/O techniques described in TPL and Traditional .NET Framework Asynchronous Programming and in this blog post.

See also

- Parallel LINQ (PLINQ)

How to: Measure PLINQ Query Performance

9/6/2018 • 2 minutes to read • Edit Online

This example shows how use the Stopwatch class to measure the time it takes for a PLINQ query to execute.

Example

This example uses an empty `foreach` loop (`For Each` in Visual Basic) to measure the time it takes for the query to execute. In real-world code, the loop typically contains additional processing steps that add to the total query execution time. Notice that the stopwatch is not started until just before the loop, because that is when the query execution begins. If you require more fine-grained measurement, you can use the `ElapsedTicks` property instead of `ElapsedMilliseconds`.

```
using System;
using System.Diagnostics;
using System.Linq;

class Example
{
   static void Main()
   {
       var source = Enumerable.Range(0, 3000000);

       var queryToMeasure = from num in source.AsParallel()
                            where num % 3 == 0
                            select Math.Sqrt(num);

       Console.WriteLine("Measuring...");

       // The query does not run until it is enumerated.
       // Therefore, start the timer here.
       Stopwatch sw = Stopwatch.StartNew();

       // For pure query cost, enumerate and do nothing else.
       foreach (var n in queryToMeasure) { }

       sw.Stop();
       long elapsed = sw.ElapsedMilliseconds; // or sw.ElapsedTicks
       Console.WriteLine("Total query time: {0} ms", elapsed);

       Console.WriteLine("Press any key to exit.");
       Console.ReadKey();
   }
}
```

```
Module Example
    Sub Main()
        Dim source = Enumerable.Range(0, 3000000)
        ' Define parallel and non-parallel queries.
        Dim queryToMeasure = From num In source.AsParallel()
                             Where num Mod 3 = 0
                             Select Math.Sqrt(num)

        Console.WriteLine("Measuring...")

        ' The query does not run until it is enumerated.
        ' Therefore, start the timer here.
        Dim sw = System.Diagnostics.Stopwatch.StartNew()

        ' For pure query cost, enumerate and do nothing else.
        For Each n As Double In queryToMeasure
        Next

        sw.Stop()
        Dim elapsed As Long = sw.ElapsedMilliseconds  ' or sw.ElapsedTicks
        Console.WriteLine("Total query time: {0} ms.", elapsed)

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

The total execution time is a useful metric when you are experimenting with query implementations, but it does not always tell the whole story. To get a deeper and richer view of the interaction of the query threads with one another and with other running processes, use the Concurrency Visualizer. For more information, see Concurrency Visualizer.

See also

- Parallel LINQ (PLINQ)

PLINQ Data Sample

9/27/2019 • 19 minutes to read • Edit Online

This sample contains example data in .csv format, together with methods that transform it into in-memory collections of Customers, Products, Orders, and Order Details. To further experiment with PLINQ, you can paste code examples from certain other topics into the code in this topic and invoke it from the `Main` method. You can also use this data with your own PLINQ queries.

The data represents a subset of the Northwind database. Fifty (50) customer records are included, but not all fields. A subset of the rows from the Orders and corresponding Order_Detail data for every Customer is included. All Products are included.

NOTE

The data set is not large enough to demonstrate that PLINQ is faster than LINQ to Objects for queries that contain just basic `where` and `select` clauses. To observe speed increases for small data sets such as this, use queries that contain computationally expensive operations on every element in the data set.

To set up this sample

1. Create a Visual Basic or Visual C# console application project.
2. Replace the contents of Module1.vb or Program.cs by using the code that follows these steps.
3. On the **Project** menu, click **Add New Item**. Select **Text File** and then click **OK**. Copy the data in this topic and then paste it in the new text file. On the **File** menu, click **Save**, name the file Plinqdata.csv, and then save it in the folder that contains your source code files.
4. Press F5 to verify that the project builds and runs correctly. The following output should be displayed in the console window.

```
Customer count: 50
Product count: 77
Order count: 190
Order Details count: 483
Press any key to exit.
```

```
// This class contains a subset of data from the Northwind database
// in the form of string arrays. The methods such as GetCustomers, GetOrders, and so on
// transform the strings into object arrays that you can query in an object-oriented way.
// Many of the code examples in the PLINQ How-to topics are designed to be pasted into
// the PLINQDataSample class and invoked from the Main method.
partial class PLINQDataSample
{

    public static void Main()
    {
        ////Call methods here.
        TestDataSource();


        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void TestDataSource()
```

```
        static void TestDataSource()
        {
            Console.WriteLine("Customer count: {0}", GetCustomers().Count());
            Console.WriteLine("Product count: {0}", GetProducts().Count());
            Console.WriteLine("Order count: {0}", GetOrders().Count());
            Console.WriteLine("Order Details count: {0}", GetOrderDetails().Count());
        }

        #region DataClasses
        public class Order
        {
            private Lazy<OrderDetail[]> _orderDetails;
            public Order()
            {
                _orderDetails = new Lazy<OrderDetail[]>(() => GetOrderDetailsForOrder(OrderID));
            }
            public int OrderID { get; set; }
            public string CustomerID { get; set; }
            public DateTime OrderDate { get; set; }
            public DateTime ShippedDate { get; set; }
            public OrderDetail[] OrderDetails { get { return _orderDetails.Value; } }
        }

        public class Customer
        {
            private Lazy<Order[]> _orders;
            public Customer()
            {
                _orders = new Lazy<Order[]>(() => GetOrdersForCustomer(CustomerID));
            }
            public string CustomerID { get; set; }
            public string CustomerName { get; set; }
            public string Address { get; set; }
            public string City { get; set; }
            public string PostalCode { get; set; }
            public Order[] Orders
            {
                get
                {
                    return _orders.Value;
                }
            }
        }

        public class Product
        {
            public string ProductName { get; set; }
            public int ProductID { get; set; }
            public double UnitPrice { get; set; }
        }

        public class OrderDetail
        {
            public int OrderID { get; set; }
            public int ProductID { get; set; }
            public double UnitPrice { get; set; }
            public double Quantity { get; set; }
            public double Discount { get; set; }
        }
        #endregion

        public static IEnumerable<string> GetCustomersAsStrings()
        {
            return System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                            .SkipWhile((line) => line.StartsWith("CUSTOMERS") == false)
                                             .Skip(1)
                                            .TakeWhile((line) => line.StartsWith("END CUSTOMERS") == false);
        }
```

```
    public static IEnumerable<Customer> GetCustomers()
    {
        var customers = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("CUSTOMERS") == false)
                                         .Skip(1)
                                         .TakeWhile((line) => line.StartsWith("END CUSTOMERS") == false);
        return (from line in customers
                let fields = line.Split(',')
                let custID = fields[0].Trim()
                select new Customer()
                {
                    CustomerID = custID,
                    CustomerName = fields[1].Trim(),
                    Address = fields[2].Trim(),
                    City = fields[3].Trim(),
                    PostalCode = fields[4].Trim()
                });
    }

    public static Order[] GetOrdersForCustomer(string id)
    {
        // Assumes we copied the file correctly!
        var orders = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("ORDERS") == false)
                                          .Skip(1)
                                        .TakeWhile((line) => line.StartsWith("END ORDERS") == false);
        var orderStrings = from line in orders
                           let fields = line.Split(',')
                           where fields[1].CompareTo(id) == 0
                           select new Order()
                           {
                               OrderID = Convert.ToInt32(fields[0]),
                               CustomerID = fields[1].Trim(),
                               OrderDate = DateTime.Parse(fields[2]),
                               ShippedDate = DateTime.Parse(fields[3])
                           };
        return orderStrings.ToArray();
    }

    //  "10248, VINET, 7/4/1996 12:00:00 AM, 7/16/1996 12:00:00 AM
    public static IEnumerable<Order> GetOrders()
    {
        // Assumes we copied the file correctly!
        var orders = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                        .SkipWhile((line) => line.StartsWith("ORDERS") == false)
                                         .Skip(1)
                                        .TakeWhile((line) => line.StartsWith("END ORDERS") == false);
        return from line in orders
               let fields = line.Split(',')

               select new Order()
               {
                   OrderID = Convert.ToInt32(fields[0]),
                   CustomerID = fields[1].Trim(),
                   OrderDate = DateTime.Parse(fields[2]),
                   ShippedDate = DateTime.Parse(fields[3])
               };
    }

    public static IEnumerable<Product> GetProducts()
    {
        // Assumes we copied the file correctly!
        var products = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                        .SkipWhile((line) => line.StartsWith("PRODUCTS") == false)
                                         .Skip(1)
                                        .TakeWhile((line) => line.StartsWith("END PRODUCTS") == false);
        return from line in products
               let fields = line.Split(',')
               select new Product()
```

```
                {
                    ProductID = Convert.ToInt32(fields[0]),
                    ProductName = fields[1].Trim(),
                    UnitPrice = Convert.ToDouble(fields[2])

                };
    }

    public static IEnumerable<OrderDetail> GetOrderDetails()
    {
        // Assumes we copied the file correctly!
        var orderDetails = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("ORDER DETAILS") == false)
                                          .Skip(1)
                                         .TakeWhile((line) => line.StartsWith("END ORDER DETAILS") == false);

        return from line in orderDetails
               let fields = line.Split(',')
               select new OrderDetail()
               {
                   OrderID = Convert.ToInt32(fields[0]),
                   ProductID = Convert.ToInt32(fields[1]),
                   UnitPrice = Convert.ToDouble(fields[2]),
                   Quantity = Convert.ToDouble(fields[3]),
                   Discount = Convert.ToDouble(fields[4])
               };
    }

    public static OrderDetail[] GetOrderDetailsForOrder(int id)
    {
        // Assumes we copied the file correctly!
        var orderDetails = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("ORDER DETAILS") == false)
                                          .Skip(1)
                                         .TakeWhile((line) => line.StartsWith("END ORDER DETAILS") == false);

        var orderDetailStrings = from line in orderDetails
                                 let fields = line.Split(',')
                                 let ordID = Convert.ToInt32(fields[0])
                                 where ordID == id
                                 select new OrderDetail()
                                 {
                                     OrderID = ordID,
                                     ProductID = Convert.ToInt32(fields[1]),
                                     UnitPrice = Convert.ToDouble(fields[2]),
                                     Quantity = Convert.ToDouble(fields[3]),
                                     Discount = Convert.ToDouble(fields[4])
                                 };

        return orderDetailStrings.ToArray();
    }
}
```

```
    ' This class contains a subset of data from the Northwind database
    ' in the form of string arrays. The methods such as GetCustomers, GetOrders, and so on
    ' transform the strings into object arrays that you can query in an object-oriented way.
    ' Many of the code examples in the PLINQ How-to topics are designed to be pasted into
    ' the PLINQDataSample class and invoked from the Main method.
    ' IMPORTANT: This data set is not large enough for meaningful comparisons of PLINQ vs. LINQ performance.
    Class PLINQDataSample
        Shared Sub Main(ByVal args As String())

            'Call methods here.
            TestDataSource()

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

        Shared Sub TestDataSource()
            Console.WriteLine("Customer count: {0}", GetCustomers().Count())
            Console.WriteLine("Product count: {0}", GetProducts().Count())
            Console.WriteLine("Order count: {0}", GetOrders().Count())
            Console.WriteLine("Order Details count: {0}", GetOrderDetails().Count())
        End Sub

#Region "DataClasses"
        Class Order
            Public Sub New()
                _OrderDetails = New Lazy(Of OrderDetail())(Function() GetOrderDetailsForOrder(OrderID))
            End Sub
            Private _OrderID As Integer
            Public Property OrderID() As Integer
                Get
                    Return _OrderID
                End Get
                Set(ByVal value As Integer)
                    _OrderID = value
                End Set
            End Property
            Private _CustomerID As String
            Public Property CustomerID() As String
                Get
                    Return _CustomerID
                End Get
                Set(ByVal value As String)
                    _CustomerID = value
                End Set
            End Property
            Private _OrderDate As DateTime
            Public Property OrderDate() As DateTime
                Get
                    Return _OrderDate
                End Get
                Set(ByVal value As DateTime)
                    _OrderDate = value
                End Set
            End Property
            Private _ShippedDate As DateTime
            Public Property ShippedDate() As DateTime
                Get
                    Return _ShippedDate
                End Get
                Set(ByVal value As DateTime)
                    _ShippedDate = value
                End Set
            End Property
            Private _OrderDetails As Lazy(Of OrderDetail())
            Public ReadOnly Property OrderDetails As OrderDetail()
                Get
                    Return _OrderDetails.Value
                End Get
            End Property
        End Class

        Class Customer

            Private _Orders As Lazy(Of Order())
            Public Sub New()
                _Orders = New Lazy(Of Order())(Function() GetOrdersForCustomer(_CustomerID))
            End Sub

            Private _CustomerID As String
            Public Property CustomerID() As String
                Get
                    Return _CustomerID
                End Get
                Set(ByVal value As String)
                    _CustomerID = value
                End Set
            End Property
            Private _CustomerName As String
            Public Property CustomerName() As String
                Get
                    Return _CustomerName
                End Get
                Set(ByVal value As String)
                    _CustomerName = value
                End Set
            End Property
            Private _Address As String
            Public Property Address() As String
                Get
                    Return _Address
                End Get
                Set(ByVal value As String)
                    _Address = value
                End Set
            End Property
            Private _City As String
            Public Property City() As String
                Get
                    Return _City
                End Get
                Set(ByVal value As String)
                    _City = value
                End Set
            End Property
            Private _PostalCode As String
            Public Property PostalCode() As String
                Get
                    Return _PostalCode
                End Get
                Set(ByVal value As String)
                    _PostalCode = value
                End Set
            End Property
            Public ReadOnly Property Orders() As Order()
                Get
                    Return _Orders.Value
                End Get
            End Property
        End Class

        Class Product
            Private _ProductName As String
            Public Property ProductName() As String
                Get
                    Return _ProductName
                End Get
                Set(ByVal value As String)
                    _ProductName = value
                End Set
            End Property
            Private _ProductID As Integer
            Public Property ProductID() As Integer
                Get
                    Return _ProductID
                End Get
                Set(ByVal value As Integer)
                    _ProductID = value
                End Set
            End Property
            Private _UnitPrice As Double
            Public Property UnitPrice() As Double
```

```
                Get
                    Return _UnitPrice
                End Get
                Set(ByVal value As Double)
                    _UnitPrice = value
                End Set
            End Property
        End Class

        Class OrderDetail
            Private _OrderID As Integer
            Public Property OrderID() As Integer
                Get
                    Return _OrderID
                End Get
                Set(ByVal value As Integer)
                    _OrderID = value
                End Set
            End Property
            Private _ProductID As Integer
            Public Property ProductID() As Integer
                Get
                    Return _ProductID
                End Get
                Set(ByVal value As Integer)
                    _ProductID = value
                End Set
            End Property
            Private _UnitPrice As Double
            Public Property UnitPrice() As Double
                Get
                    Return _UnitPrice
                End Get
                Set(ByVal value As Double)
                    _UnitPrice = value
                End Set
            End Property
            Private _Quantity As Double
            Public Property Quantity() As Double
                Get
                    Return _Quantity
                End Get
                Set(ByVal value As Double)
                    _Quantity = value
                End Set
            End Property
            Private _Discount As Double
            Public Property Discount() As Double
                Get
                    Return _Discount
                End Get
                Set(ByVal value As Double)
                    _Discount = value
                End Set
            End Property


        End Class
#End Region

        Shared Function GetCustomersAsStrings() As IEnumerable(Of String)

            Return System.IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("CUSTOMERS") = False).
                Skip(1).
                TakeWhile(Function(line) line.StartsWith("END CUSTOMERS") = False)
        End Function
        Shared Function GetCustomers() As IEnumerable(Of Customer)
```

```
    Dim customers = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
        SkipWhile(Function(line) line.StartsWith("CUSTOMERS") = False).
        Skip(1).
        TakeWhile(Function(line) line.StartsWith("END CUSTOMERS") = False)

    Return From line In customers
           Let fields = line.Split(","c)
           Select New Customer With {
            .CustomerID = fields(0).Trim(),
            .CustomerName = fields(1).Trim(),
            .Address = fields(2).Trim(),
            .City = fields(3).Trim(),
            .PostalCode = fields(4).Trim()}
End Function

Shared Function GetOrders() As IEnumerable(Of Order)

    Dim orders = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
                    SkipWhile(Function(line) line.StartsWith("ORDERS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END ORDERS") = False)

    Return From line In orders
           Let fields = line.Split(","c)
           Select New Order With {
            .OrderID = CType(fields(0).Trim(), Integer),
            .CustomerID = fields(1).Trim(),
            .OrderDate = DateTime.Parse(fields(2)),
            .ShippedDate = DateTime.Parse(fields(3))}

End Function

Shared Function GetOrdersForCustomer(ByVal id As String) As Order()

    Dim orders = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
                    SkipWhile(Function(line) line.StartsWith("ORDERS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END ORDERS") = False)

    Dim orderStrings = From line In orders
           Let fields = line.Split(","c)
           Let custID = fields(1).Trim()
           Where custID = id
           Select New Order With {
            .OrderID = CType(fields(0).Trim(), Integer),
            .CustomerID = custID,
            .OrderDate = DateTime.Parse(fields(2)),
            .ShippedDate = DateTime.Parse(fields(3))}

    Return orderStrings.ToArray()

End Function
Shared Function GetProducts() As IEnumerable(Of Product)

    Dim products = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
        SkipWhile(Function(line) line.StartsWith("PRODUCTS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END PRODUCTS") = False)

    Return From line In products
           Let fields = line.Split(","c)
           Select New Product With {
               .ProductID = CType(fields(0), Integer),
                .ProductName = fields(1).Trim(),
                .UnitPrice = CType(fields(2), Double)}
End Function

Shared Function GetOrderDetailsForOrder(ByVal orderID As Integer) As OrderDetail()
    Dim orderDetails = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
```

```
            Dim orderDetails = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("ORDER DETAILS") = False).
                            Skip(1).
                            TakeWhile(Function(line) line.StartsWith("END ORDER DETAILS") = False)

            Dim ordDetailStrings = From line In orderDetails
                   Let fields = line.Split(","c)
                   Let ordID = Convert.ToInt32(fields(0))
                   Where ordID = orderID
                       Select New OrderDetail With {
                        .OrderID = ordID,
                        .ProductID = CType(fields(1), Integer),
                        .UnitPrice = CType(fields(2), Double),
                        .Quantity = CType(fields(3), Double),
                        .Discount = CType(fields(4), Double)}
            Return ordDetailStrings.ToArray()

        End Function

        Shared Function GetOrderDetails() As IEnumerable(Of OrderDetail)
            Dim orderDetails = System.IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("ORDER DETAILS") = False).
                            Skip(1).
                            TakeWhile(Function(line) line.StartsWith("END ORDER DETAILS") = False)

            Return From line In orderDetails
                   Let fields = line.Split(","c)
                       Select New OrderDetail With {
                        .OrderID = CType(fields(0), Integer),
                        .ProductID = CType(fields(1), Integer),
                        .UnitPrice = CType(fields(2), Double),
                        .Quantity = CType(fields(3), Double),
                        .Discount = CType(fields(4), Double)}
        End Function

    End Class
```

Data

```
CUSTOMERS
ALFKI,Alfreds Futterkiste,Obere Str. 57,Berlin,12209
ANATR,Ana Trujillo Emparedados y helados,Avda. de la Constitución 2222,México D.F.,05021
ANTON,Antonio Moreno Taquería,Mataderos  2312,México D.F.,05023
AROUT,Around the Horn,120 Hanover Sq.,London,WA1 1DP
BERGS,Berglunds snabbköp,Berguvsvägen  8,Luleå,S-958 22
BLAUS,Blauer See Delikatessen,Forsterstr. 57,Mannheim,68306
BLONP,Blondesddsl père et fils,24, place Kléber,Strasbourg,67000
BOLID,Bólido Comidas preparadas,C/ Araquil, 67,Madrid,28023
BONAP,Bon app',12, rue des Bouchers,Marseille,13008
BOTTM,Bottom-Dollar Markets,23 Tsawassen Blvd.,Tsawassen,T2F 8M4
BSBEV,B's Beverages,Fauntleroy Circus,London,EC2 5NT
CACTU,Cactus Comidas para llevar,Cerrito 333,Buenos Aires,1010
CENTC,Centro comercial Moctezuma,Sierras de Granada 9993,México D.F.,05022
CHOPS,Chop-suey Chinese,Hauptstr. 29,Bern,3012
COMMI,Comércio Mineiro,Av. dos Lusíadas, 23,Sao Paulo,05432-043
CONSH,Consolidated Holdings,Berkeley Gardens 12  Brewery,London,WX1 6LT
DRACD,Drachenblut Delikatessen,Walserweg 21,Aachen,52066
DUMON,Du monde entier,67, rue des Cinquante Otages,Nantes,44000
EASTC,Eastern Connection,35 King George,London,WX3 6FW
ERNSH,Ernst Handel,Kirchgasse 6,Graz,8010
FAMIA,Familia Arquibaldo,Rua Orós, 92,Sao Paulo,05442-030
FISSA,FISSA Fabrica Inter. Salchichas S.A.,C/ Moralzarzal, 86,Madrid,28034
FOLIG,Folies gourmandes,184, chaussée de Tournai,Lille,59000
FOLKO,Folk och fä HB,Åkergatan 24,Bräcke,S-844 67
FRANK,Frankenversand,Berliner Platz 43,München,80805
FRANR,France restauration,54, rue Royale,Nantes,44000
```

```
FRANK,France restauration,54, rue Royale,Nantes,44000
FRANS,Franchi S.p.A.,Via Monte Bianco 34,Torino,10100
FURIB,Furia Bacalhau e Frutos do Mar,Jardim das rosas n. 32,Lisboa,1675
GALED,Galería del gastrónomo,Rambla de Cataluña, 23,Barcelona,08022
GODOS,Godos Cocina Típica,C/ Romero, 33,Sevilla,41101
GOURL,Gourmet Lanchonetes,Av. Brasil, 442,Campinas,04876-786
GREAL,Great Lakes Food Market,2732 Baker Blvd.,Eugene,97403
GROSR,GROSELLA-Restaurante,5ª Ave. Los Palos Grandes,Caracas,1081
HANAR,Hanari Carnes,Rua do Paço, 67,Rio de Janeiro,05454-876
HILAA,HILARION-Abastos,Carrera 22 con Ave. Carlos Soublette #8-35,San Cristóbal,5022
HUNGC,Hungry Coyote Import Store,City Center Plaza 516 Main St.,Elgin,97827
HUNGO,Hungry Owl All-Night Grocers,8 Johnstown Road,Cork,
ISLAT,Island Trading,Garden House Crowther Way,Cowes,PO31 7PJ
KOENE,Königlich Essen,Maubelstr. 90,Brandenburg,14776
LACOR,La corne d'abondance,67, avenue de l'Europe,Versailles,78000
LAMAI,La maison d'Asie,1 rue Alsace-Lorraine,Toulouse,31000
LAUGB,Laughing Bacchus Wine Cellars,1900 Oak St.,Vancouver,V3F 2K1
LAZYK,Lazy K Kountry Store,12 Orchestra Terrace,Walla Walla,99362
LEHMS,Lehmanns Marktstand,Magazinweg 7,Frankfurt a.M.,60528
LETSS,Let's Stop N Shop,87 Polk St. Suite 5,San Francisco,94117
LILAS,LILA-Supermercado,Carrera 52 con Ave. Bolívar #65-98 Llano Largo,Barquisimeto,3508
LINOD,LINO-Delicateses,Ave. 5 de Mayo Porlamar,I. de Margarita,4980
LONEP,Lonesome Pine Restaurant,89 Chiaroscuro Rd.,Portland,97219
MAGAA,Magazzini Alimentari Riuniti,Via Ludovico il Moro 22,Bergamo,24100
MAISD,Maison Dewey,Rue Joseph-Bens 532,Bruxelles,B-1180
END CUSTOMERS

ORDERS
10250,HANAR,7/8/1996 12:00:00 AM,7/12/1996 12:00:00 AM
10253,HANAR,7/10/1996 12:00:00 AM,7/16/1996 12:00:00 AM
10254,CHOPS,7/11/1996 12:00:00 AM,7/23/1996 12:00:00 AM
10257,HILAA,7/16/1996 12:00:00 AM,7/22/1996 12:00:00 AM
10258,ERNSH,7/17/1996 12:00:00 AM,7/23/1996 12:00:00 AM
10263,ERNSH,7/23/1996 12:00:00 AM,7/31/1996 12:00:00 AM
10264,FOLKO,7/24/1996 12:00:00 AM,8/23/1996 12:00:00 AM
10265,BLONP,7/25/1996 12:00:00 AM,8/12/1996 12:00:00 AM
10267,FRANK,7/29/1996 12:00:00 AM,8/6/1996 12:00:00 AM
10275,MAGAA,8/7/1996 12:00:00 AM,8/9/1996 12:00:00 AM
10278,BERGS,8/12/1996 12:00:00 AM,8/16/1996 12:00:00 AM
10279,LEHMS,8/13/1996 12:00:00 AM,8/16/1996 12:00:00 AM
10280,BERGS,8/14/1996 12:00:00 AM,9/12/1996 12:00:00 AM
10283,LILAS,8/16/1996 12:00:00 AM,8/23/1996 12:00:00 AM
10284,LEHMS,8/19/1996 12:00:00 AM,8/27/1996 12:00:00 AM
10289,BSBEV,8/26/1996 12:00:00 AM,8/28/1996 12:00:00 AM
10290,COMMI,8/27/1996 12:00:00 AM,9/3/1996 12:00:00 AM
10296,LILAS,9/3/1996 12:00:00 AM,9/11/1996 12:00:00 AM
10297,BLONP,9/4/1996 12:00:00 AM,9/10/1996 12:00:00 AM
10298,HUNGO,9/5/1996 12:00:00 AM,9/11/1996 12:00:00 AM
10300,MAGAA,9/9/1996 12:00:00 AM,9/18/1996 12:00:00 AM
10303,GODOS,9/11/1996 12:00:00 AM,9/18/1996 12:00:00 AM
10307,LONEP,9/17/1996 12:00:00 AM,9/25/1996 12:00:00 AM
10309,HUNGO,9/19/1996 12:00:00 AM,10/23/1996 12:00:00 AM
10315,ISLAT,9/26/1996 12:00:00 AM,10/3/1996 12:00:00 AM
10317,LONEP,9/30/1996 12:00:00 AM,10/10/1996 12:00:00 AM
10318,ISLAT,10/1/1996 12:00:00 AM,10/4/1996 12:00:00 AM
10321,ISLAT,10/3/1996 12:00:00 AM,10/11/1996 12:00:00 AM
10323,KOENE,10/7/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10325,KOENE,10/9/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10327,FOLKO,10/11/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10328,FURIB,10/14/1996 12:00:00 AM,10/17/1996 12:00:00 AM
10330,LILAS,10/16/1996 12:00:00 AM,10/28/1996 12:00:00 AM
10331,BONAP,10/16/1996 12:00:00 AM,10/21/1996 12:00:00 AM
10335,HUNGO,10/22/1996 12:00:00 AM,10/24/1996 12:00:00 AM
10337,FRANK,10/24/1996 12:00:00 AM,10/29/1996 12:00:00 AM
10340,BONAP,10/29/1996 12:00:00 AM,11/8/1996 12:00:00 AM
10342,FRANK,10/30/1996 12:00:00 AM,11/4/1996 12:00:00 AM
10343,LEHMS,10/31/1996 12:00:00 AM,11/6/1996 12:00:00 AM
10347,FAMIA,11/6/1996 12:00:00 AM,11/8/1996 12:00:00 AM
10350,LAMAI,11/11/1996 12:00:00 AM,12/3/1996 12:00:00 AM
10351,ERNSH,11/11/1996 12:00:00 AM,11/20/1996 12:00:00 AM
```

```
10351,ERNSH,11/11/1996 12:00:00 AM,11/20/1996 12:00:00 AM
10352,FURIB,11/12/1996 12:00:00 AM,11/18/1996 12:00:00 AM
10355,AROUT,11/15/1996 12:00:00 AM,11/20/1996 12:00:00 AM
10357,LILAS,11/19/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10358,LAMAI,11/20/1996 12:00:00 AM,11/27/1996 12:00:00 AM
10360,BLONP,11/22/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10362,BONAP,11/25/1996 12:00:00 AM,11/28/1996 12:00:00 AM
10363,DRACD,11/26/1996 12:00:00 AM,12/4/1996 12:00:00 AM
10364,EASTC,11/26/1996 12:00:00 AM,12/4/1996 12:00:00 AM
10365,ANTON,11/27/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10366,GALED,11/28/1996 12:00:00 AM,12/30/1996 12:00:00 AM
10368,ERNSH,11/29/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10370,CHOPS,12/3/1996 12:00:00 AM,12/27/1996 12:00:00 AM
10371,LAMAI,12/3/1996 12:00:00 AM,12/24/1996 12:00:00 AM
10373,HUNGO,12/5/1996 12:00:00 AM,12/11/1996 12:00:00 AM
10375,HUNGC,12/6/1996 12:00:00 AM,12/9/1996 12:00:00 AM
10378,FOLKO,12/10/1996 12:00:00 AM,12/19/1996 12:00:00 AM
10380,HUNGO,12/12/1996 12:00:00 AM,1/16/1997 12:00:00 AM
10381,LILAS,12/12/1996 12:00:00 AM,12/13/1996 12:00:00 AM
10382,ERNSH,12/13/1996 12:00:00 AM,12/16/1996 12:00:00 AM
10383,AROUT,12/16/1996 12:00:00 AM,12/18/1996 12:00:00 AM
10384,BERGS,12/16/1996 12:00:00 AM,12/20/1996 12:00:00 AM
10386,FAMIA,12/18/1996 12:00:00 AM,12/25/1996 12:00:00 AM
10389,BOTTM,12/20/1996 12:00:00 AM,12/24/1996 12:00:00 AM
10391,DRACD,12/23/1996 12:00:00 AM,12/31/1996 12:00:00 AM
10394,HUNGC,12/25/1996 12:00:00 AM,1/3/1997 12:00:00 AM
10395,HILAA,12/26/1996 12:00:00 AM,1/3/1997 12:00:00 AM
10396,FRANK,12/27/1996 12:00:00 AM,1/6/1997 12:00:00 AM
10400,EASTC,1/1/1997 12:00:00 AM,1/16/1997 12:00:00 AM
10404,MAGAA,1/3/1997 12:00:00 AM,1/8/1997 12:00:00 AM
10405,LINOD,1/6/1997 12:00:00 AM,1/22/1997 12:00:00 AM
10408,FOLIG,1/8/1997 12:00:00 AM,1/14/1997 12:00:00 AM
10410,BOTTM,1/10/1997 12:00:00 AM,1/15/1997 12:00:00 AM
10411,BOTTM,1/10/1997 12:00:00 AM,1/21/1997 12:00:00 AM
10413,LAMAI,1/14/1997 12:00:00 AM,1/16/1997 12:00:00 AM
10414,FAMIA,1/14/1997 12:00:00 AM,1/17/1997 12:00:00 AM
10415,HUNGC,1/15/1997 12:00:00 AM,1/24/1997 12:00:00 AM
10422,FRANS,1/22/1997 12:00:00 AM,1/31/1997 12:00:00 AM
10423,GOURL,1/23/1997 12:00:00 AM,2/24/1997 12:00:00 AM
10425,LAMAI,1/24/1997 12:00:00 AM,2/14/1997 12:00:00 AM
10426,GALED,1/27/1997 12:00:00 AM,2/6/1997 12:00:00 AM
10431,BOTTM,1/30/1997 12:00:00 AM,2/7/1997 12:00:00 AM
10434,FOLKO,2/3/1997 12:00:00 AM,2/13/1997 12:00:00 AM
10436,BLONP,2/5/1997 12:00:00 AM,2/11/1997 12:00:00 AM
10444,BERGS,2/12/1997 12:00:00 AM,2/21/1997 12:00:00 AM
10445,BERGS,2/13/1997 12:00:00 AM,2/20/1997 12:00:00 AM
10449,BLONP,2/18/1997 12:00:00 AM,2/27/1997 12:00:00 AM
10453,AROUT,2/21/1997 12:00:00 AM,2/26/1997 12:00:00 AM
10456,KOENE,2/25/1997 12:00:00 AM,2/28/1997 12:00:00 AM
10457,KOENE,2/25/1997 12:00:00 AM,3/3/1997 12:00:00 AM
10460,FOLKO,2/28/1997 12:00:00 AM,3/3/1997 12:00:00 AM
10464,FURIB,3/4/1997 12:00:00 AM,3/14/1997 12:00:00 AM
10466,COMMI,3/6/1997 12:00:00 AM,3/13/1997 12:00:00 AM
10467,MAGAA,3/6/1997 12:00:00 AM,3/11/1997 12:00:00 AM
10468,KOENE,3/7/1997 12:00:00 AM,3/12/1997 12:00:00 AM
10470,BONAP,3/11/1997 12:00:00 AM,3/14/1997 12:00:00 AM
10471,BSBEV,3/11/1997 12:00:00 AM,3/18/1997 12:00:00 AM
10473,ISLAT,3/13/1997 12:00:00 AM,3/21/1997 12:00:00 AM
10476,HILAA,3/17/1997 12:00:00 AM,3/24/1997 12:00:00 AM
10480,FOLIG,3/20/1997 12:00:00 AM,3/24/1997 12:00:00 AM
10484,BSBEV,3/24/1997 12:00:00 AM,4/1/1997 12:00:00 AM
10485,LINOD,3/25/1997 12:00:00 AM,3/31/1997 12:00:00 AM
10486,HILAA,3/26/1997 12:00:00 AM,4/2/1997 12:00:00 AM
10488,FRANK,3/27/1997 12:00:00 AM,4/2/1997 12:00:00 AM
10490,HILAA,3/31/1997 12:00:00 AM,4/3/1997 12:00:00 AM
10491,FURIB,3/31/1997 12:00:00 AM,4/8/1997 12:00:00 AM
10492,BOTTM,4/1/1997 12:00:00 AM,4/11/1997 12:00:00 AM
10494,COMMI,4/2/1997 12:00:00 AM,4/9/1997 12:00:00 AM
10497,LEHMS,4/4/1997 12:00:00 AM,4/7/1997 12:00:00 AM
```

```
10501,BLAUS,4/9/1997 12:00:00 AM,4/16/1997 12:00:00 AM
10507,ANTON,4/15/1997 12:00:00 AM,4/22/1997 12:00:00 AM
10509,BLAUS,4/17/1997 12:00:00 AM,4/29/1997 12:00:00 AM
10511,BONAP,4/18/1997 12:00:00 AM,4/21/1997 12:00:00 AM
10512,FAMIA,4/21/1997 12:00:00 AM,4/24/1997 12:00:00 AM
10519,CHOPS,4/28/1997 12:00:00 AM,5/1/1997 12:00:00 AM
10521,CACTU,4/29/1997 12:00:00 AM,5/2/1997 12:00:00 AM
10522,LEHMS,4/30/1997 12:00:00 AM,5/6/1997 12:00:00 AM
10528,GREAL,5/6/1997 12:00:00 AM,5/9/1997 12:00:00 AM
10529,MAISD,5/7/1997 12:00:00 AM,5/9/1997 12:00:00 AM
10532,EASTC,5/9/1997 12:00:00 AM,5/12/1997 12:00:00 AM
10535,ANTON,5/13/1997 12:00:00 AM,5/21/1997 12:00:00 AM
10538,BSBEV,5/15/1997 12:00:00 AM,5/16/1997 12:00:00 AM
10539,BSBEV,5/16/1997 12:00:00 AM,5/23/1997 12:00:00 AM
10541,HANAR,5/19/1997 12:00:00 AM,5/29/1997 12:00:00 AM
10544,LONEP,5/21/1997 12:00:00 AM,5/30/1997 12:00:00 AM
10550,GODOS,5/28/1997 12:00:00 AM,6/6/1997 12:00:00 AM
10551,FURIB,5/28/1997 12:00:00 AM,6/6/1997 12:00:00 AM
10558,AROUT,6/4/1997 12:00:00 AM,6/10/1997 12:00:00 AM
10568,GALED,6/13/1997 12:00:00 AM,7/9/1997 12:00:00 AM
10573,ANTON,6/19/1997 12:00:00 AM,6/20/1997 12:00:00 AM
10581,FAMIA,6/26/1997 12:00:00 AM,7/2/1997 12:00:00 AM
10582,BLAUS,6/27/1997 12:00:00 AM,7/14/1997 12:00:00 AM
10589,GREAL,7/4/1997 12:00:00 AM,7/14/1997 12:00:00 AM
10600,HUNGC,7/16/1997 12:00:00 AM,7/21/1997 12:00:00 AM
10614,BLAUS,7/29/1997 12:00:00 AM,8/1/1997 12:00:00 AM
10616,GREAL,7/31/1997 12:00:00 AM,8/5/1997 12:00:00 AM
10617,GREAL,7/31/1997 12:00:00 AM,8/4/1997 12:00:00 AM
10621,ISLAT,8/5/1997 12:00:00 AM,8/11/1997 12:00:00 AM
10629,GODOS,8/12/1997 12:00:00 AM,8/20/1997 12:00:00 AM
10634,FOLIG,8/15/1997 12:00:00 AM,8/21/1997 12:00:00 AM
10635,MAGAA,8/18/1997 12:00:00 AM,8/21/1997 12:00:00 AM
10638,LINOD,8/20/1997 12:00:00 AM,9/1/1997 12:00:00 AM
10643,ALFKI,8/25/1997 12:00:00 AM,9/2/1997 12:00:00 AM
10645,HANAR,8/26/1997 12:00:00 AM,9/2/1997 12:00:00 AM
10649,MAISD,8/28/1997 12:00:00 AM,8/29/1997 12:00:00 AM
10652,GOURL,9/1/1997 12:00:00 AM,9/8/1997 12:00:00 AM
10656,GREAL,9/4/1997 12:00:00 AM,9/10/1997 12:00:00 AM
10660,HUNGC,9/8/1997 12:00:00 AM,10/15/1997 12:00:00 AM
10662,LONEP,9/9/1997 12:00:00 AM,9/18/1997 12:00:00 AM
10665,LONEP,9/11/1997 12:00:00 AM,9/17/1997 12:00:00 AM
10677,ANTON,9/22/1997 12:00:00 AM,9/26/1997 12:00:00 AM
10685,GOURL,9/29/1997 12:00:00 AM,10/3/1997 12:00:00 AM
10690,HANAR,10/2/1997 12:00:00 AM,10/3/1997 12:00:00 AM
10692,ALFKI,10/3/1997 12:00:00 AM,10/13/1997 12:00:00 AM
10697,LINOD,10/8/1997 12:00:00 AM,10/14/1997 12:00:00 AM
10702,ALFKI,10/13/1997 12:00:00 AM,10/21/1997 12:00:00 AM
10707,AROUT,10/16/1997 12:00:00 AM,10/23/1997 12:00:00 AM
10709,GOURL,10/17/1997 12:00:00 AM,11/20/1997 12:00:00 AM
10710,FRANS,10/20/1997 12:00:00 AM,10/23/1997 12:00:00 AM
10726,EASTC,11/3/1997 12:00:00 AM,12/5/1997 12:00:00 AM
10729,LINOD,11/4/1997 12:00:00 AM,11/14/1997 12:00:00 AM
10731,CHOPS,11/6/1997 12:00:00 AM,11/14/1997 12:00:00 AM
10734,GOURL,11/7/1997 12:00:00 AM,11/12/1997 12:00:00 AM
10746,CHOPS,11/19/1997 12:00:00 AM,11/21/1997 12:00:00 AM
10753,FRANS,11/25/1997 12:00:00 AM,11/27/1997 12:00:00 AM
10760,MAISD,12/1/1997 12:00:00 AM,12/10/1997 12:00:00 AM
10763,FOLIG,12/3/1997 12:00:00 AM,12/8/1997 12:00:00 AM
10782,CACTU,12/17/1997 12:00:00 AM,12/22/1997 12:00:00 AM
10789,FOLIG,12/22/1997 12:00:00 AM,12/31/1997 12:00:00 AM
10797,DRACD,12/25/1997 12:00:00 AM,1/5/1998 12:00:00 AM
10807,FRANS,12/31/1997 12:00:00 AM,1/30/1998 12:00:00 AM
10819,CACTU,1/7/1998 12:00:00 AM,1/16/1998 12:00:00 AM
10825,DRACD,1/9/1998 12:00:00 AM,1/14/1998 12:00:00 AM
10835,ALFKI,1/15/1998 12:00:00 AM,1/21/1998 12:00:00 AM
10853,BLAUS,1/27/1998 12:00:00 AM,2/3/1998 12:00:00 AM
10872,GODOS,2/5/1998 12:00:00 AM,2/9/1998 12:00:00 AM
10874,GODOS,2/6/1998 12:00:00 AM,2/11/1998 12:00:00 AM
10881,CACTU,2/11/1998 12:00:00 AM,2/18/1998 12:00:00 AM
```

```
10887,GALED,2/13/1998 12:00:00 AM,2/16/1998 12:00:00 AM
10892,MAISD,2/17/1998 12:00:00 AM,2/19/1998 12:00:00 AM
10896,MAISD,2/19/1998 12:00:00 AM,2/27/1998 12:00:00 AM
10928,GALED,3/5/1998 12:00:00 AM,3/18/1998 12:00:00 AM
10937,CACTU,3/10/1998 12:00:00 AM,3/13/1998 12:00:00 AM
10952,ALFKI,3/16/1998 12:00:00 AM,3/24/1998 12:00:00 AM
10969,COMMI,3/23/1998 12:00:00 AM,3/30/1998 12:00:00 AM
10987,EASTC,3/31/1998 12:00:00 AM,4/6/1998 12:00:00 AM
11026,FRANS,4/15/1998 12:00:00 AM,4/28/1998 12:00:00 AM
11036,DRACD,4/20/1998 12:00:00 AM,4/22/1998 12:00:00 AM
11042,COMMI,4/22/1998 12:00:00 AM,5/1/1998 12:00:00 AM
END ORDERS

ORDER DETAILS
10250,41,7.7000,10,0
10250,51,42.4000,35,0.15
10250,65,16.8000,15,0.15
10253,31,10.0000,20,0
10253,39,14.4000,42,0
10253,49,16.0000,40,0
10254,24,3.6000,15,0.15
10254,55,19.2000,21,0.15
10254,74,8.0000,21,0
10257,27,35.1000,25,0
10257,39,14.4000,6,0
10257,77,10.4000,15,0
10258,2,15.2000,50,0.2
10258,5,17.0000,65,0.2
10258,32,25.6000,6,0.2
10263,16,13.9000,60,0.25
10263,24,3.6000,28,0
10263,30,20.7000,60,0.25
10263,74,8.0000,36,0.25
10264,2,15.2000,35,0
10264,41,7.7000,25,0.15
10265,17,31.2000,30,0
10265,70,12.0000,20,0
10267,40,14.7000,50,0
10267,59,44.0000,70,0.15
10267,76,14.4000,15,0.15
10275,24,3.6000,12,0.05
10275,59,44.0000,6,0.05
10278,44,15.5000,16,0
10278,59,44.0000,15,0
10278,63,35.1000,8,0
10278,73,12.0000,25,0
10279,17,31.2000,15,0.25
10280,24,3.6000,12,0
10280,55,19.2000,20,0
10280,75,6.2000,30,0
10283,15,12.4000,20,0
10283,19,7.3000,18,0
10283,60,27.2000,35,0
10283,72,27.8000,3,0
10284,27,35.1000,15,0.25
10284,44,15.5000,21,0
10284,60,27.2000,20,0.25
10284,67,11.2000,5,0.25
10289,3,8.0000,30,0
10289,64,26.6000,9,0
10290,5,17.0000,20,0
10290,29,99.0000,15,0
10290,49,16.0000,15,0
10290,77,10.4000,10,0
10296,11,16.8000,12,0
10296,16,13.9000,30,0
10296,69,28.8000,15,0
10297,39,14.4000,60,0
10297,72,27.8000,20,0
```

```
10298,2,15.2000,40,0
10298,36,15.2000,40,0.25
10298,59,44.0000,30,0.25
10298,62,39.4000,15,0
10300,66,13.6000,30,0
10300,68,10.0000,20,0
10303,40,14.7000,40,0.1
10303,65,16.8000,30,0.1
10303,68,10.0000,15,0.1
10307,62,39.4000,10,0
10307,68,10.0000,3,0
10309,4,17.6000,20,0
10309,6,20.0000,30,0
10309,42,11.2000,2,0
10309,43,36.8000,20,0
10309,71,17.2000,3,0
10315,34,11.2000,14,0
10315,70,12.0000,30,0
10317,1,14.4000,20,0
10318,41,7.7000,20,0
10318,76,14.4000,6,0
10321,35,14.4000,10,0
10323,15,12.4000,5,0
10323,25,11.2000,4,0
10323,39,14.4000,4,0
10325,6,20.0000,6,0
10325,13,4.8000,12,0
10325,14,18.6000,9,0
10325,31,10.0000,4,0
10325,72,27.8000,40,0
10327,2,15.2000,25,0.2
10327,11,16.8000,50,0.2
10327,30,20.7000,35,0.2
10327,58,10.6000,30,0.2
10328,59,44.0000,9,0
10328,65,16.8000,40,0
10328,68,10.0000,10,0
10330,26,24.9000,50,0.15
10330,72,27.8000,25,0.15
10331,54,5.9000,15,0
10335,2,15.2000,7,0.2
10335,31,10.0000,25,0.2
10335,32,25.6000,6,0.2
10335,51,42.4000,48,0.2
10337,23,7.2000,40,0
10337,26,24.9000,24,0
10337,36,15.2000,20,0
10337,37,20.8000,28,0
10337,72,27.8000,25,0
10340,18,50.0000,20,0.05
10340,41,7.7000,12,0.05
10340,43,36.8000,40,0.05
10342,2,15.2000,24,0.2
10342,31,10.0000,56,0.2
10342,36,15.2000,40,0.2
10342,55,19.2000,40,0.2
10343,64,26.6000,50,0
10343,68,10.0000,4,0.05
10343,76,14.4000,15,0
10347,25,11.2000,10,0
10347,39,14.4000,50,0.15
10347,40,14.7000,4,0
10347,75,6.2000,6,0.15
10350,50,13.0000,15,0.1
10350,69,28.8000,18,0.1
10351,38,210.8000,20,0.05
10351,41,7.7000,13,0
10351,44,15.5000,77,0.05
10351,65,16.8000,10,0.05
```

```
10352,24,3.6000,10,0
10352,54,5.9000,20,0.15
10355,24,3.6000,25,0
10355,57,15.6000,25,0
10357,10,24.8000,30,0.2
10357,26,24.9000,16,0
10357,60,27.2000,8,0.2
10358,24,3.6000,10,0.05
10358,34,11.2000,10,0.05
10358,36,15.2000,20,0.05
10360,28,36.4000,30,0
10360,29,99.0000,35,0
10360,38,210.8000,10,0
10360,49,16.0000,35,0
10360,54,5.9000,28,0
10362,25,11.2000,50,0
10362,51,42.4000,20,0
10362,54,5.9000,24,0
10363,31,10.0000,20,0
10363,75,6.2000,12,0
10363,76,14.4000,12,0
10364,69,28.8000,30,0
10364,71,17.2000,5,0
10365,11,16.8000,24,0
10366,65,16.8000,5,0
10366,77,10.4000,5,0
10368,21,8.0000,5,0.1
10368,28,36.4000,13,0.1
10368,57,15.6000,25,0
10368,64,26.6000,35,0.1
10370,1,14.4000,15,0.15
10370,64,26.6000,30,0
10370,74,8.0000,20,0.15
10371,36,15.2000,6,0.2
10373,58,10.6000,80,0.2
10373,71,17.2000,50,0.2
10375,14,18.6000,15,0
10375,54,5.9000,10,0
10378,71,17.2000,6,0
10380,30,20.7000,18,0.1
10380,53,26.2000,20,0.1
10380,60,27.2000,6,0.1
10380,70,12.0000,30,0
10381,74,8.0000,14,0
10382,5,17.0000,32,0
10382,18,50.0000,9,0
10382,29,99.0000,14,0
10382,33,2.0000,60,0
10382,74,8.0000,50,0
10383,13,4.8000,20,0
10383,50,13.0000,15,0
10383,56,30.4000,20,0
10384,20,64.8000,28,0
10384,60,27.2000,15,0
10386,24,3.6000,15,0
10386,34,11.2000,10,0
10389,10,24.8000,16,0
10389,55,19.2000,15,0
10389,62,39.4000,20,0
10389,70,12.0000,30,0
10391,13,4.8000,18,0
10394,13,4.8000,10,0
10394,62,39.4000,10,0
10395,46,9.6000,28,0.1
10395,53,26.2000,70,0.1
10395,69,28.8000,8,0
10396,23,7.2000,40,0
10396,71,17.2000,60,0
10396,72,27.8000,21,0
```

```
10400,29,99.0000,21,0
10400,35,14.4000,35,0
10400,49,16.0000,30,0
10404,26,24.9000,30,0.05
10404,42,11.2000,40,0.05
10404,49,16.0000,30,0.05
10405,3,8.0000,50,0
10408,37,20.8000,10,0
10408,54,5.9000,6,0
10408,62,39.4000,35,0
10410,33,2.0000,49,0
10410,59,44.0000,16,0
10411,41,7.7000,25,0.2
10411,44,15.5000,40,0.2
10411,59,44.0000,9,0.2
10413,1,14.4000,24,0
10413,62,39.4000,40,0
10413,76,14.4000,14,0
10414,19,7.3000,18,0.05
10414,33,2.0000,50,0
10415,17,31.2000,2,0
10415,33,2.0000,20,0
10422,26,24.9000,2,0
10423,31,10.0000,14,0
10423,59,44.0000,20,0
10425,55,19.2000,10,0.25
10425,76,14.4000,20,0.25
10426,56,30.4000,5,0
10426,64,26.6000,7,0
10431,17,31.2000,50,0.25
10431,40,14.7000,50,0.25
10431,47,7.6000,30,0.25
10434,11,16.8000,6,0
10434,76,14.4000,18,0.15
10436,46,9.6000,5,0
10436,56,30.4000,40,0.1
10436,64,26.6000,30,0.1
10436,75,6.2000,24,0.1
10444,17,31.2000,10,0
10444,26,24.9000,15,0
10444,35,14.4000,8,0
10444,41,7.7000,30,0
10445,39,14.4000,6,0
10445,54,5.9000,15,0
10449,10,24.8000,14,0
10449,52,5.6000,20,0
10449,62,39.4000,35,0
10453,48,10.2000,15,0.1
10453,70,12.0000,25,0.1
10456,21,8.0000,40,0.15
10456,49,16.0000,21,0.15
10457,59,44.0000,36,0
10460,68,10.0000,21,0.25
10460,75,6.2000,4,0.25
10464,4,17.6000,16,0.2
10464,43,36.8000,3,0
10464,56,30.4000,30,0.2
10464,60,27.2000,20,0
10466,11,16.8000,10,0
10466,46,9.6000,5,0
10467,24,3.6000,28,0
10467,25,11.2000,12,0
10468,30,20.7000,8,0
10468,43,36.8000,15,0
10470,18,50.0000,30,0
10470,23,7.2000,15,0
10470,64,26.6000,8,0
10471,7,24.0000,30,0
```

```
10471,56,30.4000,20,0
10473,33,2.0000,12,0
10473,71,17.2000,12,0
10476,55,19.2000,2,0.05
10476,70,12.0000,12,0
10480,47,7.6000,30,0
10480,59,44.0000,12,0
10484,21,8.0000,14,0
10484,40,14.7000,10,0
10484,51,42.4000,3,0
10485,2,15.2000,20,0.1
10485,3,8.0000,20,0.1
10485,55,19.2000,30,0.1
10485,70,12.0000,60,0.1
10486,11,16.8000,5,0
10486,51,42.4000,25,0
10486,74,8.0000,16,0
10488,59,44.0000,30,0
10488,73,12.0000,20,0.2
10490,59,44.0000,60,0
10490,68,10.0000,30,0
10490,75,6.2000,36,0
10491,44,15.5000,15,0.15
10491,77,10.4000,7,0.15
10492,25,11.2000,60,0.05
10492,42,11.2000,20,0.05
10494,56,30.4000,30,0
10497,56,30.4000,14,0
10497,72,27.8000,25,0
10497,77,10.4000,25,0
10501,54,7.4500,20,0
10507,43,46.0000,15,0.15
10507,48,12.7500,15,0.15
10509,28,45.6000,3,0
10511,4,22.0000,50,0.15
10511,7,30.0000,50,0.15
10511,8,40.0000,10,0.15
10512,24,4.5000,10,0.15
10512,46,12.0000,9,0.15
10512,47,9.5000,6,0.15
10512,60,34.0000,12,0.15
10519,10,31.0000,16,0.05
10519,56,38.0000,40,0
10519,60,34.0000,10,0.05
10521,35,18.0000,3,0
10521,41,9.6500,10,0
10521,68,12.5000,6,0
10522,1,18.0000,40,0.2
10522,8,40.0000,24,0
10522,30,25.8900,20,0.2
10522,40,18.4000,25,0.2
10528,11,21.0000,3,0
10528,33,2.5000,8,0.2
10528,72,34.8000,9,0
10529,55,24.0000,14,0
10529,68,12.5000,20,0
10529,69,36.0000,10,0
10532,30,25.8900,15,0
10532,66,17.0000,24,0
10535,11,21.0000,50,0.1
10535,40,18.4000,10,0.1
10535,57,19.5000,5,0.1
10535,59,55.0000,15,0.1
10538,70,15.0000,7,0
10538,72,34.8000,1,0
10539,13,6.0000,8,0
10539,21,10.0000,15,0
10539,33,2.5000,15,0
10539,49,20.0000,6,0
```

```
10541,24,4.5000,35,0.1
10541,38,263.5000,4,0.1
10541,65,21.0500,36,0.1
10541,71,21.5000,9,0.1
10544,28,45.6000,7,0
10544,67,14.0000,7,0
10550,17,39.0000,8,0.1
10550,19,9.2000,10,0
10550,21,10.0000,6,0.1
10550,61,28.5000,10,0.1
10551,16,17.4500,40,0.15
10551,35,18.0000,20,0.15
10551,44,19.4500,40,0
10558,47,9.5000,25,0
10558,51,53.0000,20,0
10558,52,7.0000,30,0
10558,53,32.8000,18,0
10558,73,15.0000,3,0
10568,10,31.0000,5,0
10573,17,39.0000,18,0
10573,34,14.0000,40,0
10573,53,32.8000,25,0
10581,75,7.7500,50,0.2
10582,57,19.5000,4,0
10582,76,18.0000,14,0
10589,35,18.0000,4,0
10600,54,7.4500,4,0
10600,73,15.0000,30,0
10614,11,21.0000,14,0
10614,21,10.0000,8,0
10614,39,18.0000,5,0
10616,38,263.5000,15,0.05
10616,56,38.0000,14,0
10616,70,15.0000,15,0.05
10616,71,21.5000,15,0.05
10617,59,55.0000,30,0.15
10621,19,9.2000,5,0
10621,23,9.0000,10,0
10621,70,15.0000,20,0
10621,71,21.5000,15,0
10629,29,123.7900,20,0
10629,64,33.2500,9,0
10634,7,30.0000,35,0
10634,18,62.5000,50,0
10634,51,53.0000,15,0
10634,75,7.7500,2,0
10635,4,22.0000,10,0.1
10635,5,21.3500,15,0.1
10635,22,21.0000,40,0
10638,45,9.5000,20,0
10638,65,21.0500,21,0
10638,72,34.8000,60,0
10643,28,45.6000,15,0.25
10643,39,18.0000,21,0.25
10643,46,12.0000,2,0.25
10645,18,62.5000,20,0
10645,36,19.0000,15,0
10649,28,45.6000,20,0
10649,72,34.8000,15,0
10652,30,25.8900,2,0.25
10652,42,14.0000,20,0
10656,14,23.2500,3,0.1
10656,44,19.4500,28,0.1
10656,47,9.5000,6,0.1
10660,20,81.0000,21,0
10662,68,12.5000,10,0
10665,51,53.0000,20,0
10665,59,55.0000,1,0
10665,76,18.0000,10,0
```

```
10677,26,31.2300,30,0.15
10677,33,2.5000,8,0.15
10685,10,31.0000,20,0
10685,41,9.6500,4,0
10685,47,9.5000,15,0
10690,56,38.0000,20,0.25
10690,77,13.0000,30,0.25
10692,63,43.9000,20,0
10697,19,9.2000,7,0.25
10697,35,18.0000,9,0.25
10697,58,13.2500,30,0.25
10697,70,15.0000,30,0.25
10702,3,10.0000,6,0
10702,76,18.0000,15,0
10707,55,24.0000,21,0
10707,57,19.5000,40,0
10707,70,15.0000,28,0.15
10709,8,40.0000,40,0
10709,51,53.0000,28,0
10709,60,34.0000,10,0
10710,19,9.2000,5,0
10710,47,9.5000,5,0
10726,4,22.0000,25,0
10726,11,21.0000,5,0
10729,1,18.0000,50,0
10729,21,10.0000,30,0
10729,50,16.2500,40,0
10731,21,10.0000,40,0.05
10731,51,53.0000,30,0.05
10734,6,25.0000,30,0
10734,30,25.8900,15,0
10734,76,18.0000,20,0
10746,13,6.0000,6,0
10746,42,14.0000,28,0
10746,62,49.3000,9,0
10746,69,36.0000,40,0
10753,45,9.5000,4,0
10753,74,10.0000,5,0
10760,25,14.0000,12,0.25
10760,27,43.9000,40,0
10760,43,46.0000,30,0.25
10763,21,10.0000,40,0
10763,22,21.0000,6,0
10763,24,4.5000,20,0
10782,31,12.5000,1,0
10789,18,62.5000,30,0
10789,35,18.0000,15,0
10789,63,43.9000,30,0
10789,68,12.5000,18,0
10797,11,21.0000,20,0
10807,40,18.4000,1,0
10819,43,46.0000,7,0
10819,75,7.7500,20,0
10825,26,31.2300,12,0
10825,53,32.8000,20,0
10835,59,55.0000,15,0
10835,77,13.0000,2,0.2
10853,18,62.5000,10,0
10872,55,24.0000,10,0.05
10872,62,49.3000,20,0.05
10872,64,33.2500,15,0.05
10872,65,21.0500,21,0.05
10874,10,31.0000,10,0
10881,73,15.0000,10,0
10887,25,14.0000,5,0
10892,59,55.0000,40,0.05
10896,45,9.5000,15,0
10896,56,38.0000,16,0
10928,47,9.5000,5,0
```

```
10928,76,18.0000,5,0
10937,28,45.6000,8,0
10937,34,14.0000,20,0
10952,6,25.0000,16,0.05
10952,28,45.6000,2,0
10969,46,12.0000,9,0
10987,7,30.0000,60,0
10987,43,46.0000,6,0
10987,72,34.8000,20,0
11026,18,62.5000,8,0
11026,51,53.0000,10,0
11036,13,6.0000,7,0
11036,59,55.0000,30,0
11042,44,19.4500,15,0
11042,61,28.5000,4,0
END ORDER DETAILS

PRODUCTS
1,Chai,18.0000
2,Chang,19.0000
3,Aniseed Syrup,10.0000
4,Chef Anton's Cajun Seasoning,22.0000
5,Chef Anton's Gumbo Mix,21.3500
6,Grandma's Boysenberry Spread,25.0000
7,Uncle Bob's Organic Dried Pears,30.0000
8,Northwoods Cranberry Sauce,40.0000
9,Mishi Kobe Niku,97.0000
10,Ikura,31.0000
11,Queso Cabrales,21.0000
12,Queso Manchego La Pastora,38.0000
13,Konbu,6.0000
14,Tofu,23.2500
15,Genen Shouyu,15.5000
16,Pavlova,17.4500
17,Alice Mutton,39.0000
18,Carnarvon Tigers,62.5000
19,Teatime Chocolate Biscuits,9.2000
20,Sir Rodney's Marmalade,81.0000
21,Sir Rodney's Scones,10.0000
22,Gustaf's Knäckebröd,21.0000
23,Tunnbröd,9.0000
24,Guaraná Fantástica,4.5000
25,NuNuCa Nuß-Nougat-Creme,14.0000
26,Gumbär Gummibärchen,31.2300
27,Schoggi Schokolade,43.9000
28,Rössle Sauerkraut,45.6000
29,Thüringer Rostbratwurst,123.7900
30,Nord-Ost Matjeshering,25.8900
31,Gorgonzola Telino,12.5000
32,Mascarpone Fabioli,32.0000
33,Geitost,2.5000
34,Sasquatch Ale,14.0000
35,Steeleye Stout,18.0000
36,Inlagd Sill,19.0000
37,Gravad lax,26.0000
38,Côte de Blaye,263.5000
39,Chartreuse verte,18.0000
40,Boston Crab Meat,18.4000
41,Jack's New England Clam Chowder,9.6500
42,Singaporean Hokkien Fried Mee,14.0000
43,Ipoh Coffee,46.0000
44,Gula Malacca,19.4500
45,Rogede sild,9.5000
46,Spegesild,12.0000
47,Zaanse koeken,9.5000
48,Chocolade,12.7500
49,Maxilaku,20.0000
50,Valkoinen suklaa,16.2500
51,Manjimup Dried Apples,53.0000
```

```
52,Filo Mix,7.0000
53,Perth Pasties,32.8000
54,Tourtière,7.4500
55,Pâté chinois,24.0000
56,Gnocchi di nonna Alice,38.0000
57,Ravioli Angelo,19.5000
58,Escargots de Bourgogne,13.2500
59,Raclette Courdavault,55.0000
60,Camembert Pierrot,34.0000
61,Sirop d'érable,28.5000
62,Tarte au sucre,49.3000
63,Vegie-spread,43.9000
64,Wimmers gute Semmelknödel,33.2500
65,Louisiana Fiery Hot Pepper Sauce,21.0500
66,Louisiana Hot Spiced Okra,17.0000
67,Laughing Lumberjack Lager,14.0000
68,Scottish Longbreads,12.5000
69,Gudbrandsdalsost,36.0000
70,Outback Lager,15.0000
71,Flotemysost,21.5000
72,Mozzarella di Giovanni,34.8000
73,Röd Kaviar,15.0000
74,Longlife Tofu,10.0000
75,Rhönbräu Klosterbier,7.7500
76,Lakkalikööri,18.0000
77,Original Frankfurter grüne Soße,13.0000
END PRODUCTS
```

See also

- Parallel LINQ (PLINQ)

Data Structures for Parallel Programming

4/28/2019 • 3 minutes to read • Edit Online

The .NET Framework version 4 introduces several new types that are useful in parallel programming, including a set of concurrent collection classes, lightweight synchronization primitives, and types for lazy initialization. You can use these types with any multithreaded application code, including the Task Parallel Library and PLINQ.

Concurrent Collection Classes

The collection classes in the System.Collections.Concurrent namespace provide thread-safe add and remove operations that avoid locks wherever possible and use fine-grained locking where locks are necessary. Unlike collections that were introduced in the .NET Framework versions 1.0 and 2.0, a concurrent collection class does not require user code to take any locks when it accesses items. The concurrent collection classes can significantly improve performance over types such as System.Collections.ArrayList and System.Collections.Generic.List<T> (with user-implemented locking) in scenarios where multiple threads add and remove items from a collection.

The following table lists the new concurrent collection classes:

TYPE	DESCRIPTION
System.Collections.Concurrent.BlockingCollection<T>	Provides blocking and bounding capabilities for thread-safe collections that implement System.Collections.Concurrent.IProducerConsumerCollection<T>. Producer threads block if no slots are available or if the collection is full. Consumer threads block if the collection is empty. This type also supports non-blocking access by consumers and producers. BlockingCollection<T> can be used as a base class or backing store to provide blocking and bounding for any collection class that supports IEnumerable<T>.
System.Collections.Concurrent.ConcurrentBag<T>	A thread-safe bag implementation that provides scalable add and get operations.
System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>	A concurrent and scalable dictionary type.
System.Collections.Concurrent.ConcurrentQueue<T>	A concurrent and scalable FIFO queue.
System.Collections.Concurrent.ConcurrentStack<T>	A concurrent and scalable LIFO stack.

For more information, see Thread-Safe Collections.

Synchronization Primitives

The new synchronization primitives in the System.Threading namespace enable fine-grained concurrency and faster performance by avoiding expensive locking mechanisms found in legacy multithreading code. Some of the new types, such as System.Threading.Barrier and System.Threading.CountdownEvent have no counterparts in earlier releases of the .NET Framework.

The following table lists the new synchronization types:

TYPE	DESCRIPTION
System.Threading.Barrier	Enables multiple threads to work on an algorithm in parallel by providing a point at which each task can signal its arrival and then block until some or all tasks have arrived. For more information, see Barrier.
System.Threading.CountdownEvent	Simplifies fork and join scenarios by providing an easy rendezvous mechanism. For more information, see CountdownEvent.
System.Threading.ManualResetEventSlim	A synchronization primitive similar to System.Threading.ManualResetEvent. ManualResetEventSlim is lighter-weight but can only be used for intra-process communication.
System.Threading.SemaphoreSlim	A synchronization primitive that limits the number of threads that can concurrently access a resource or a pool of resources. For more information, see Semaphore and SemaphoreSlim.
System.Threading.SpinLock	A mutual exclusion lock primitive that causes the thread that is trying to acquire the lock to wait in a loop, or *spin*, for a period of time before yielding its quantum. In scenarios where the wait for the lock is expected to be short, SpinLock offers better performance than other forms of locking. For more information, see SpinLock.
System.Threading.SpinWait	A small, lightweight type that will spin for a specified time and eventually put the thread into a wait state if the spin count is exceeded. For more information, see SpinWait.

For more information, see:

- How to: Use SpinLock for Low-Level Synchronization
- How to: Synchronize Concurrent Operations with a Barrier.

Lazy Initialization Classes

With lazy initialization, the memory for an object is not allocated until it is needed. Lazy initialization can improve performance by spreading object allocations evenly across the lifetime of a program. You can enable lazy initialization for any custom type by wrapping the type Lazy<T>.

The following table lists the lazy initialization types:

TYPE	DESCRIPTION
System.Lazy<T>	Provides lightweight, thread-safe lazy-initialization.
System.Threading.ThreadLocal<T>	Provides a lazily-initialized value on a per-thread basis, with each thread lazily-invoking the initialization function.
System.Threading.LazyInitializer	Provides static methods that avoid the need to allocate a dedicated, lazy-initialization instance. Instead, they use references to ensure targets have been initialized as they are accessed.

For more information, see Lazy Initialization.

Aggregate Exceptions

The System.AggregateException type can be used to capture multiple exceptions that are thrown concurrently on separate threads, and return them to the joining thread as a single exception. The System.Threading.Tasks.Task and System.Threading.Tasks.Parallel types and PLINQ use AggregateException extensively for this purpose. For more information, see Exception Handling and How to: Handle Exceptions in a PLINQ Query.

See also

- System.Collections.Concurrent
- System.Threading
- Parallel Programming

Parallel Diagnostic Tools

4/28/2019 • 2 minutes to read • Edit Online

Visual Studio provides extensive support for debugging and profiling multi-threaded applications.

Debugging

The Visual Studio debugger adds new windows for debugging parallel applications. For more information, see the following topics:

- Using the Parallel Stacks Window
- Using the Tasks Window
- Walkthrough: Debugging a Parallel Application.

Profiling

The Concurrency Visualizer report views enable you to visualize how the threads in a parallel program interact with each other and with threads from other processes on the system. For more information, see Concurrency Visualizer.

See also

- Parallel Programming

Custom Partitioners for PLINQ and TPL

7/9/2019 • 10 minutes to read • Edit Online

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.

```
// Static partitioning requires indexable source. Load balancing
// can use any IEnumerable.
var nums = Enumerable.Range(0, 100000000).ToArray();

// Create a load-balancing partitioner. Or specify false for static partitioning.
Partitioner<int> customPartitioner = Partitioner.Create(nums, true);

// The partitioner is the query's data source.
var q = from x in customPartitioner.AsParallel()
        select x * Math.PI;

q.ForAll((x) =>
{
    ProcessData(x);
});
```

```
' 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<TSource>(IEnumerable<TSource>)	Always
Create<TSource>(TSource[], Boolean)	When the Boolean argument is specified as true
Create<TSource>(IList<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<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.

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {

        // Source must be array or IList.
        var source = Enumerable.Range(0, 100000).ToArray();

        // Partition the entire source array.
        var rangePartitioner = Partitioner.Create(0, source.Length);

        double[] results = new double[source.Length];

        // Loop over the partitions in parallel.
        Parallel.ForEach(rangePartitioner, (range, loopState) =>
        {
            // Loop over each range element without a delegate invocation.
            for (int i = range.Item1; i < range.Item2; i++)
            {
                results[i] = source[i] * Math.PI;
            }
        });

        Console.WriteLine("Operation complete. Print results? y/n");
        char input = Console.ReadKey().KeyChar;
        if (input == 'y' || input == 'Y')
        {
            foreach(double d in results)
            {
                Console.Write("{0} ", d);
            }
        }
    }
}
```

```
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<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<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<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<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<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<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<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<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
- How to: Implement Dynamic Partitions
- How to: Implement a Partitioner for Static Partitioning

How to: Implement Dynamic Partitions

5/31/2019 • 3 minutes to read • Edit Online

The following example shows how to implement a custom System.Collections.Concurrent.OrderablePartitioner<TSource> that implements dynamic partitioning and can be used from certain overloads ForEach and from PLINQ.

Example

Each time a partition calls MoveNext on the enumerator, the enumerator provides the partition with one list element. In the case of PLINQ and ForEach, the partition is a Task instance. Because requests are happening concurrently on multiple threads, access to the current index is synchronized.

```
//
// An orderable dynamic partitioner for lists
//
using System;
using System.Collections;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Xml.Linq;
using System.Numerics;

class OrderableListPartitioner<TSource> : OrderablePartitioner<TSource>
{
    private readonly IList<TSource> m_input;

    // Must override to return true.
    public override bool SupportsDynamicPartitions => true;

    public OrderableListPartitioner(IList<TSource> input) : base(true, false, true) =>
        m_input = input;

    public override IList<IEnumerator<KeyValuePair<long, TSource>>> GetOrderablePartitions(int partitionCount)
    {
        var dynamicPartitions = GetOrderableDynamicPartitions();
        var partitions =
            new IEnumerator<KeyValuePair<long, TSource>>[partitionCount];

        for (int i = 0; i < partitionCount; i++)
        {
            partitions[i] = dynamicPartitions.GetEnumerator();
        }
        return partitions;
    }

    public override IEnumerable<KeyValuePair<long, TSource>> GetOrderableDynamicPartitions() =>
        new ListDynamicPartitions(m_input);

    private class ListDynamicPartitions : IEnumerable<KeyValuePair<long, TSource>>
    {
        private IList<TSource> m_input;
        private int m_pos = 0;

        internal ListDynamicPartitions(IList<TSource> input) =>
```

```
            m_input = input;

        public IEnumerator<KeyValuePair<long, TSource>> GetEnumerator()
        {
            while (true)
            {
                // Each task gets the next item in the list. The index is
                // incremented in a thread-safe manner to avoid races.
                int elemIndex = Interlocked.Increment(ref m_pos) - 1;

                if (elemIndex >= m_input.Count)
                {
                    yield break;
                }

                yield return new KeyValuePair<long, TSource>(
                    elemIndex, m_input[elemIndex]);
            }
        }

        IEnumerator IEnumerable.GetEnumerator() =>
            ((IEnumerable<KeyValuePair<long, TSource>>)this).GetEnumerator();
    }
}

class ConsumerClass
{
    static void Main()
    {
        var nums = Enumerable.Range(0, 10000).ToArray();
        OrderableListPartitioner<int> partitioner = new OrderableListPartitioner<int>(nums);

        // Use with Parallel.ForEach
        Parallel.ForEach(partitioner, (i) => Console.WriteLine(i));


        // Use with PLINQ
        var query = from num in partitioner.AsParallel()
                    where num % 2 == 0
                    select num;

        foreach (var v in query)
            Console.WriteLine(v);
    }
}
```

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

Module Module1
    Public Class OrderableListPartitioner(Of TSource)
        Inherits OrderablePartitioner(Of TSource)


        Private ReadOnly m_input As IList(Of TSource)

        Public Sub New(ByVal input As IList(Of TSource))
            MyBase.New(True, False, True)
            m_input = input
        End Sub

        ' Must override to return true.
        Public Overrides ReadOnly Property SupportsDynamicPartitions As Boolean
            Get
                Return True
            End Get
        End Property

        Public Overrides Function GetOrderablePartitions(ByVal partitionCount As Integer) As IList(Of
IEnumerator(Of KeyValuePair(Of Long, TSource)))
            Dim dynamicPartitions = GetOrderableDynamicPartitions()
            Dim partitions(partitionCount - 1) As IEnumerator(Of KeyValuePair(Of Long, TSource))

            For i = 0 To partitionCount - 1
                partitions(i) = dynamicPartitions.GetEnumerator()
            Next

            Return partitions
        End Function

        Public Overrides Function GetOrderableDynamicPartitions() As IEnumerable(Of KeyValuePair(Of Long,
TSource))
            Return New ListDynamicPartitions(m_input)
        End Function

        Private Class ListDynamicPartitions
            Implements IEnumerable(Of KeyValuePair(Of Long, TSource))

            Private m_input As IList(Of TSource)

            Friend Sub New(ByVal input As IList(Of TSource))
                m_input = input
            End Sub

            Public Function GetEnumerator() As IEnumerator(Of KeyValuePair(Of Long, TSource)) Implements
IEnumerable(Of KeyValuePair(Of Long, TSource)).GetEnumerator
                Return New ListDynamicPartitionsEnumerator(m_input)
            End Function

            Public Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
                Return CType(Me, IEnumerable).GetEnumerator()
            End Function
        End Class

        Private Class ListDynamicPartitionsEnumerator
            Implements IEnumerator(Of KeyValuePair(Of Long, TSource))

            Private m_input As IList(Of TSource)
            Shared m_pos As Integer = 0
            Private m_current As KeyValuePair(Of Long, TSource)

            Public Sub New(ByVal input As IList(Of TSource))
                m_input = input
                m_pos = 0
                Me.disposedValue = False
            End Sub

            Public ReadOnly Property Current As KeyValuePair(Of Long, TSource) Implements IEnumerator(Of
KeyValuePair(Of Long, TSource)).Current
                Get
                    Return m_current
                End Get
            End Property

            Public ReadOnly Property Current1 As Object Implements IEnumerator.Current
                Get
                    Return Me.Current
                End Get
            End Property

            Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
                Dim elemIndex = Interlocked.Increment(m_pos) - 1
                If elemIndex >= m_input.Count Then
                    Return False
                End If
                m_current = New KeyValuePair(Of Long, TSource)(elemIndex, m_input(elemIndex))
                Return True
            End Function

            Public Sub Reset() Implements IEnumerator.Reset
                m_pos = 0
            End Sub

            Private disposedValue As Boolean ' To detect redundant calls

            Protected Overridable Sub Dispose(ByVal disposing As Boolean)
                If Not Me.disposedValue Then
                    m_input = Nothing
                    m_current = Nothing
                End If
                Me.disposedValue = True
            End Sub

            Public Sub Dispose() Implements IDisposable.Dispose
                Dispose(True)
                GC.SuppressFinalize(Me)
            End Sub

        End Class

    End Class

    Class ConsumerClass

        Shared Sub Main()

            Console.BufferHeight = 20000
            Dim nums = Enumerable.Range(0, 2000).ToArray()

            Dim partitioner = New OrderableListPartitioner(Of Integer)(nums)

            ' Use with Parallel.ForEach
            Parallel.ForEach(partitioner, Sub(i) Console.Write("{0}:{1}  ", i,
Thread.CurrentThread.ManagedThreadId))

            Console.WriteLine("PLINQ -----------------------------------")


            ' create a new partitioner, since Enumerators are not reusable.
            Dim partitioner2 = New OrderableListPartitioner(Of Integer)(nums)
            ' Use with PLINQ
            Dim query = From num In partitioner2.AsParallel()
                        Where num Mod 8 = 0
                        Select num

            For Each v In query
                Console.Write("{0}  ", v)
            Next

            Console.WriteLine("press any key")
            Console.ReadKey()
        End Sub
    End Class

End Module
```

This is an example of chunk partitioning, with each chunk consisting of one element. By providing more elements at a time, you could reduce the contention over the lock and theoretically achieve faster performance. However, at some point, larger chunks might require additional load-balancing logic in order to keep all threads busy until all the work is done.

See also

- Custom Partitioners for PLINQ and TPL
- How to: Implement a Partitioner for Static Partitioning

How to: Implement a Partitioner for Static Partitioning

9/6/2018 • 3 minutes to read • Edit Online

The following example shows one way to implement a simple custom partitioner for PLINQ that performs static partitioning. Because the partitioner does not support dynamic partitions, it is not consumable from Parallel.ForEach. This particular partitioner might provide speedup over the default range partitioner for data sources for which each element requires an increasing amount of processing time.

Example

```
// A static range partitioner for sources that require
// a linear increase in processing time for each succeeding element.
// The range sizes are calculated based on the rate of increase
// with the first partition getting the most elements and the
// last partition getting the least.
class MyPartitioner : Partitioner<int>
{
    int[] source;
    double rateOfIncrease = 0;

    public MyPartitioner(int[] source, double rate)
    {
        this.source = source;
        rateOfIncrease = rate;
    }

    public override IEnumerable<int> GetDynamicPartitions()
    {
        throw new NotImplementedException();
    }

    // Not consumable from Parallel.ForEach.
    public override bool SupportsDynamicPartitions
    {
        get
        {
            return false;
        }
    }

    public override IList<IEnumerator<int>> GetPartitions(int partitionCount)
    {
        List<IEnumerator<int>> _list = new List<IEnumerator<int>>();
        int end = 0;
        int start = 0;
        int[] nums = CalculatePartitions(partitionCount, source.Length);

        for (int i = 0; i < nums.Length; i++)
        {
            start = nums[i];
            if (i < nums.Length - 1)
                end = nums[i + 1];
            else
                end = source.Length;

            _list.Add(GetItemsForPartition(start, end));

            // For demonstratation.
            Console.WriteLine("start = {0} b (end) = {1}", start, end);
        }
        return (IList<IEnumerator<int>>)_list;
    }
    /*
     *
     *
     *                                                               B
      // Model increasing workloads as a right triangle           /  |
         divided into equal areas along vertical lines.         / |  |
         Each partition  is taller and skinnier               /   |  |
         than the last.                                     / |   |  |
                                                          /   |   |  |
                                                        /     |   |  |
                                                      /  |    |   |  |
                                                    /    |    |   |  |
                                            A     /______|____|___|__| C
     */
    private int[] CalculatePartitions(int partitionCount, int sourceLength)
    {
        // Corresponds to the opposite side of angle A, which corresponds
        // to an index into the source array.
        int[] partitionLimits = new int[partitionCount];
        partitionLimits[0] = 0;

        // Represent total work as rectangle of source length times "most expensive element"
        // Note: RateOfIncrease can be factored out of equation.
        double totalWork = sourceLength * (sourceLength * rateOfIncrease);
        // Divide by two to get the triangle whose slope goes from zero on the left to "most"
        // on the right. Then divide by number of partitions to get area of each partition.
        totalWork /= 2;
        double partitionArea = totalWork / partitionCount;

        // Draw the next partitionLimit on the vertical coordinate that gives
        // an area of partitionArea * currentPartition.
        for (int i = 1; i < partitionLimits.Length; i++)
        {
            double area = partitionArea * i;

           // Solve for base given the area and the slope of the hypotenuse.
            partitionLimits[i] = (int)Math.Floor(Math.Sqrt((2 * area) / rateOfIncrease));
        }
        return partitionLimits;
    }


    IEnumerator<int> GetItemsForPartition(int start, int end)
    {
        // For demonstration purpsoes. Each thread receives its own enumerator.
        Console.WriteLine("called on thread {0}", Thread.CurrentThread.ManagedThreadId);
        for (int i = start; i < end; i++)
            yield return source[i];
    }
}

class Consumer
{
    public static void Main2()
    {
        var source = Enumerable.Range(0, 10000).ToArray();

        Stopwatch sw = Stopwatch.StartNew();
        MyPartitioner partitioner = new MyPartitioner(source, .5);

        var query = from n in partitioner.AsParallel()
                    select ProcessData(n);

        foreach (var v in query) { }
        Console.WriteLine("Processing time with custom partitioner {0}", sw.ElapsedMilliseconds);
        var source2 = Enumerable.Range(0, 10000).ToArray();

        sw = Stopwatch.StartNew();


        var query2 = from n in source2.AsParallel()
                     select ProcessData(n);

        foreach (var v in query2) { }
        Console.WriteLine("Processing time with default partitioner {0}", sw.ElapsedMilliseconds);
    }

    // Consistent processing time for measurement purposes.
    static int ProcessData(int i)
    {
        Thread.SpinWait(i * 1000);
        return i;
    }


}
```

The partitions in this example are based on the assumption of a linear increase in processing time for each element. In the real world, it might be difficult to predict processing times in this way. If you are using a static partitioner with a specific data source, you can optimize the partitioning formula for the source, add load-balancing logic, or use a chunk partitioning approach as demonstrated in How to: Implement Dynamic Partitions.

See also

- Custom Partitioners for PLINQ and TPL

Lambda Expressions in PLINQ and TPL

9/10/2019 • 3 minutes to read • Edit Online

The Task Parallel Library (TPL) contains many methods that take one of the System.Func<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 and Delegates. For more information about lambda expressions in C# and Visual Basic, see Lambda Expressions and Lambda Expressions.

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<T,TResult>; in fact this type takes only one input parameter. The Framework Class Library defines 17 versions of `Func`: System.Func<TResult>, System.Func<T,TResult>, System.Func<T1,T2,TResult>, and so on up through System.Func<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<TSource,TLocal>(IEnumerable<TSource>, Func<TLocal>, Func<TSource,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) method shows how to express both Func and Action delegates by using lambda expressions.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

class ForEachWithThreadLocal
{
    // 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/library/dd990270(VS.100).aspx
    static void Main()
    {
        // The sum of these elements is 40.
        int[] input = { 4, 1, 6, 2, 9, 5, 10, 3 };
        int sum = 0;

        try
        {
            Parallel.ForEach(
                    input,             // source collection
                    () => 0,             // thread local initializer
                    (n, loopState, localSum) =>  // body
                    {
                        localSum += n;
                        Console.WriteLine("Thread={0}, n={1}, localSum={2}",
Thread.CurrentThread.ManagedThreadId, n, localSum);
                        return localSum;
                    },
                    (localSum) => Interlocked.Add(ref sum, localSum)     // thread local aggregator
                );

            Console.WriteLine("\nSum={0}", sum);
        }
        // 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.
        catch (AggregateException e)
        {
            Console.WriteLine("Parallel.ForEach has thrown an exception. THIS WAS NOT EXPECTED.\n{0}", e);
        }
    }

}
```

```
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/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

For Further Reading (Parallel Programming)

9/12/2019 • 2 minutes to read • Edit Online

The following resources contain additional information about parallel programming in .NET:

- The Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 document describes common parallel patterns and best practices for developing parallel components utilizing those patterns.
- The Design Patterns for Decomposition and Coordination on Multicore Architectures book describes patterns for parallel programming that use the parallel programming support introduced in the .NET Framework 4.
- The Parallel Programming with .NET blog contains many in-depth articles about parallel programming in .NET.
- The Samples for Parallel Programming with the .NET Framework page contains many samples that demonstrate intermediate and advanced parallel programming techniques.

See also

- Parallel Computing Developer Center
- Parallel Programming in Visual C++