* 1. 

Hands-On Lab

Introduction to the Parallel Extensions Library

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281933625)

[Exercise 1: Parallelize an Existing Algorithm using the Static Parallel Helper Class 5](#_Toc281933626)

[Task 1 – Parallelizing a Long Running Service 5](#_Toc281933627)

[Exercise 2: Create and Run Parallelized Tasks 23](#_Toc281933628)

[Task 1 – Natively Running Parallelized Tasks 23](#_Toc281933629)

[Task 2 – Using the Wait() and WaitAll() Methods 25](#_Toc281933630)

[Task 3 – Using the IsCompleted Property 28](#_Toc281933631)

[Task 4 – Using the ContinueWith() Method 31](#_Toc281933632)

[Exercise 3: Use the Generic Task Class to Create and Run Tasks that Return a Value 33](#_Toc281933633)

[Task 1 – Capturing a Task’s Return Value 33](#_Toc281933634)

[Exercise 4: Parallelize LINQ Queries using PLINQ 35](#_Toc281933635)

[Task 1 – Using the ParallelEnumerable Class’ Static Methods to Parallelize LINQ 35](#_Toc281933636)

[Task 2 – Using the ParallelEnumerable Class’ Extension Methods to Parallelize LINQ 40](#_Toc281933637)

[Task 3 – Using AsParallel() With Query Comprehension Syntax 43](#_Toc281933638)

[Summary 45](#_Toc281933639)

Overview

Modern computers have seen explosive growth in the number of processors and cores available for systems running on them. System developers can take advantage of this power in a number of ways in their software, particularly when working on complex algorithms or large sets of data.

Microsoft’s Parallel Computing Platform (PCP) is providing tools enabling developers to leverage this power in an efficient, maintainable, and scalable manner. Parallel Extensions brings to the .NET Framework several important concepts into this toolset: imperative and task parallelism via the Task Parallel Library (TPL), and Parallel LINQ (PLINQ), which gives developers a declarative way to deal with data parallelism.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Parallelize an existing algorithm by using the static Parallel helper class and have the expression of concurrency handled automatically.
  + Create and run Tasks that enable abilities like cancellation of in-process tasks.
  + Use the Task<T> class to create and run Tasks that return a value.
  + Use Parallel LINQ (PLINQ) to optimize LINQ queries to exectue in a parallel environment.

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + NET Framework 4

# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

# Exercises

* 1. This Hands-On Lab is comprised by the following exercises:
  + Parallelize an Existing Algorithm by using the Static Parallel Class.
  + Create and Run Parallelized Tasks.
  + Use the Task<T> Class to Create and Run a Task that Returns a Value.
  + Parallelize LINQ Queries using PLINQ.
  1. Estimated time to complete this lab: **60 minutes**.
  2. **Note:** Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

**Note:** Each exercise contains a Visual Basic and a C# version; Inside the **End/Begin** solution folder you will find two folders: **VB**, containing the Visual Basic version of the exercise, and **C#**, containing the C# version of it.

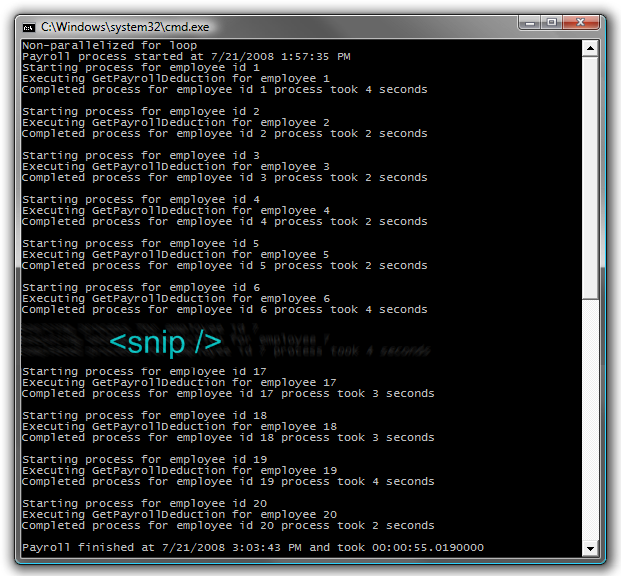
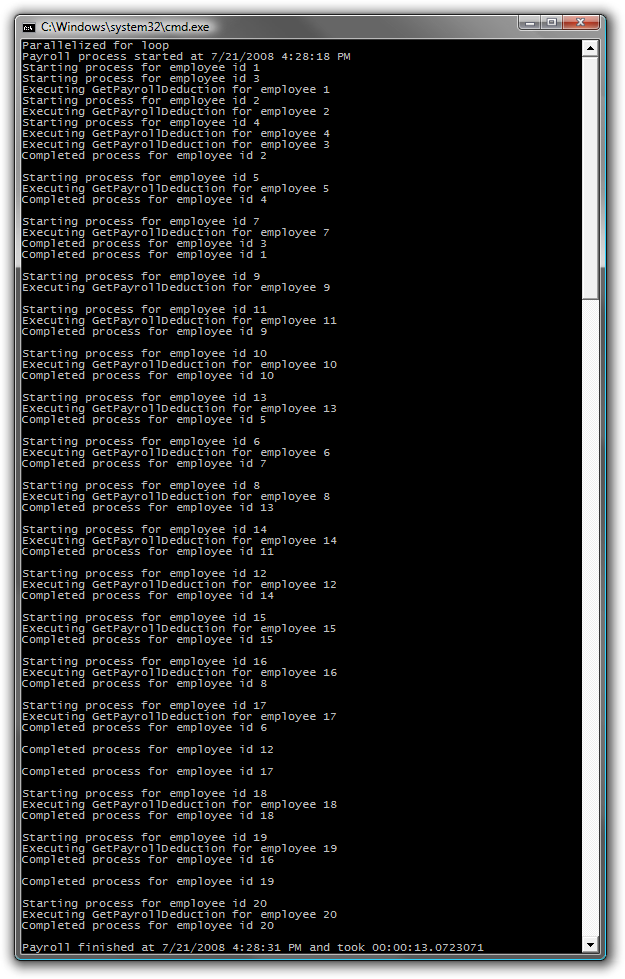
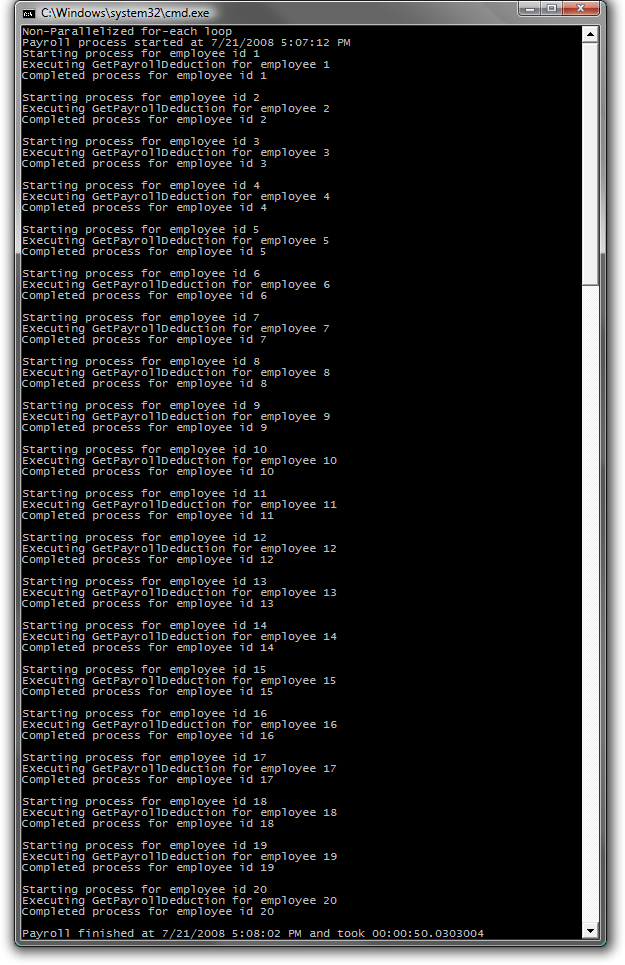
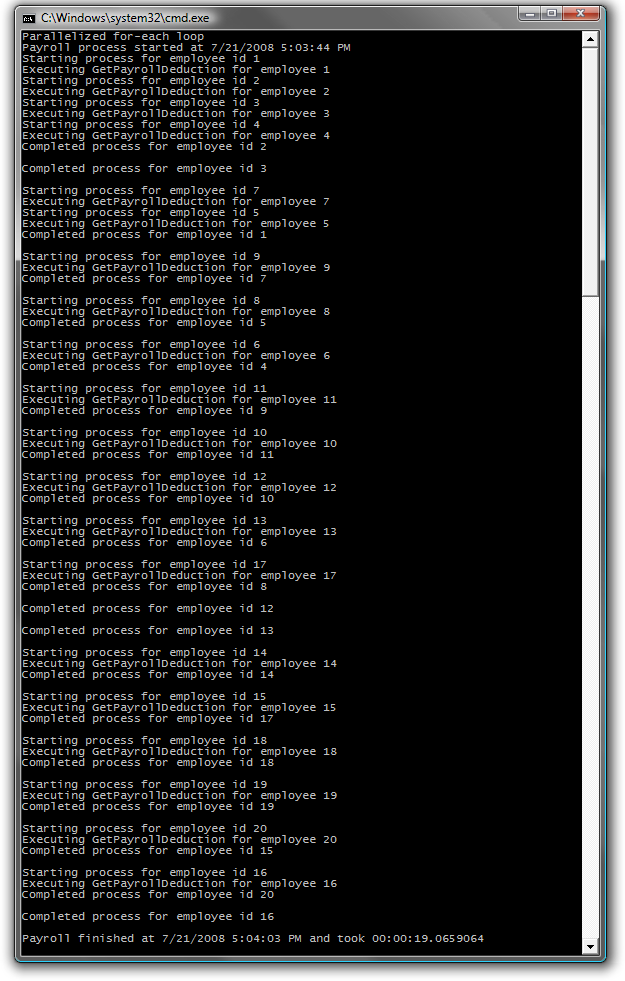
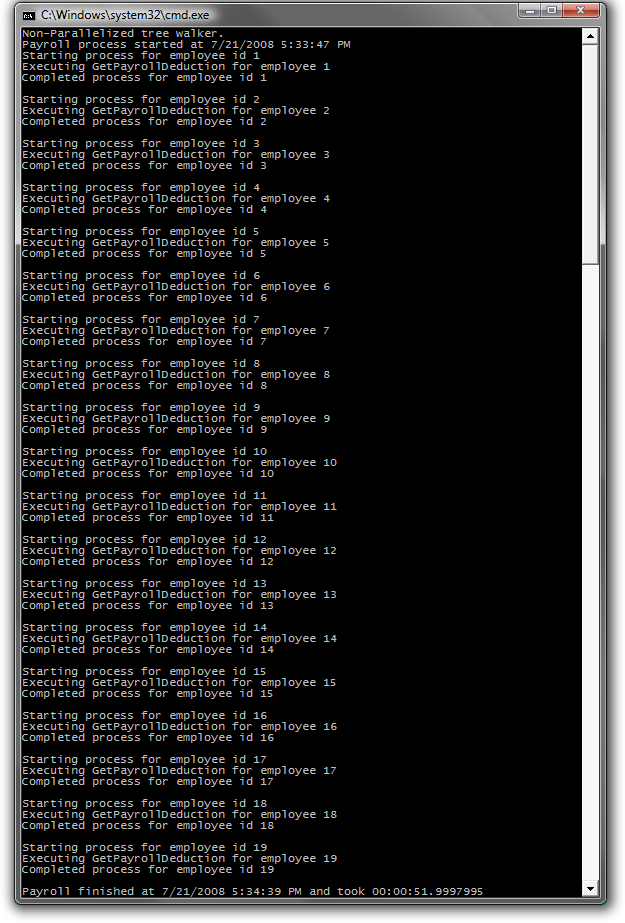
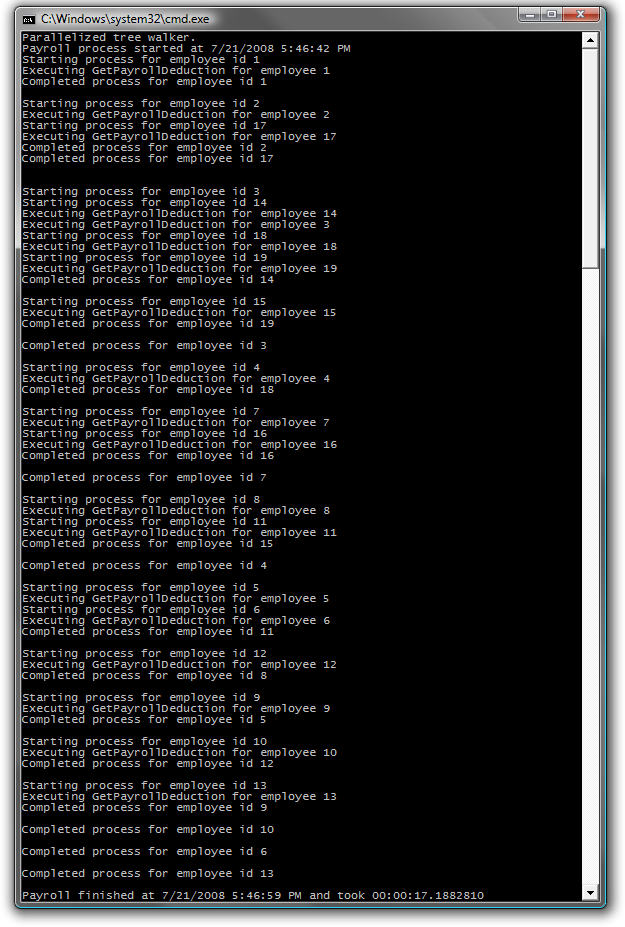
# Next Step:

* 1. Exercise 1: Parallelize an Existing Algorithm using the Static Parallel Helper Class

Exercise 1: Parallelize an Existing Algorithm using the Static Parallel Helper Class

* 1. In this exercise, you will learn how to parallelize and existing algorithm by using the static **Parallel** helper class. This allows you to do things like replacing **for()** with **Parallel.For()**.
  2. **Note:** To verify that each step is correctly performed, it is recommended to build the solution at the end of each task.

Task 1 – Parallelizing a Long Running Service

* 1. In this task, you will write some simple sample code that will simulate a long running service call.
  2. You are going to use the **PayrollServices.GetPayrollDeduction()** method which is provided with the begin solution of this exercise. This is the type of long running code that you would ultimately like to run in parallel.
  3. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  4. Open the solution file **ParallelExtLab.sln** located under ***Source\Ex01-UsingStaticParallelHelper\begin***. **(Choosing the folder that matches the language of your preference.)**
     1. **Note:** This solution contains a starting point for your work, and includes a helper class **EmployeeList** which holds the data you’ll be working with.
  5. In Visual Studio, open the **Program.cs (C#)** or **Module1.vb (Visual Basic)** file and navigate to its **Main()** method. First, you will need to create a list of employees to work on, so add a class variable and initialize it in the **Main()** method:
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Create employee list CSharp*)
     2. C#
     3. class Program
     4. {
     5. **private static EmployeeList employeeData;**
     7. **static void Main(string[] args)**
     8. **{**
     9. **employeeData = new EmployeeList();**
     10. **Console.WriteLine("Payroll process started at {0}", DateTime.Now);**
     11. **var sw = Stopwatch.StartNew();**
     12. **// Methods to call**
     13. **Console.WriteLine("Payroll finished at {0} and took {1}",**
     14. **DateTime.Now, sw.Elapsed.TotalSeconds);**
     15. **Console.WriteLine();**
     16. **Console.ReadLine();**
     17. **}**
     18. }
     19. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Create employee list VB*)
     20. Visual Basic
     21. Module Module1
     22. **Private employeeData As EmployeeList**
     23. **Sub Main(ByVal args() As String)**
     24. **employeeData = New EmployeeList()**
     25. **Console.WriteLine("Payroll process started at {0}", DateTime.Now)**
     26. **Dim sw = Stopwatch.StartNew()**
     27. **' Methods to call**
     28. **Console.WriteLine("Payroll finished at {0} and took {1}", DateTime.Now, sw.Elapsed.TotalSeconds)**
     29. **Console.WriteLine()**
     30. **Console.ReadLine()**
     31. **End Sub**
     32. End Module
  6. Now add the following method to **Program.cs** (C#) or **Module1.vb** (Visual Basic). This method will use a standard **for** loop to iterate through a list of **Employees**, as provided by the pre-built code and call the long-running **PayrollServices.GetPayrollDeduction()** method. The code should look like:
     1. (Code Snippet – *Intro to Parallel Extensions Lab –Ex1 Ex1Task1\_ParallelizeLongRunningService CSharp*)
     2. C#
     3. **private static void Ex1Task1\_ParallelizeLongRunningService()**
     4. **{**
     5. **Console.WriteLine("Non-parallelized for loop");**
     6. **for (int i = 0; i < employeeData.Count; i++)**
     7. **{**
     8. **Console.WriteLine("Starting process for employee id {0}",**
     9. **employeeData[i].EmployeeID);**
     10. **decimal span =**
     11. **PayrollServices.GetPayrollDeduction(employeeData[i]);**
     12. **Console.WriteLine("Completed process for employee id {0}" +**
     13. **"process took {1} seconds",**
     14. **employeeData[i].EmployeeID, span);**
     15. **Console.WriteLine();**
     16. **}**
     17. **}**
     18. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_ParallelizeLongRunningService Visual Basic*)
     19. Visual Basic
     20. **Private Sub Ex1Task1\_ParallelizeLongRunningService()**
     21. **Console.WriteLine("Non-parallelized for loop")**
     22. **For i = 0 To employeeData.Count - 1**
     23. **Console.WriteLine("Starting process for employee id {0}", employeeData(i).EmployeeID)**
     24. **Dim span As Decimal = PayrollServices.GetPayrollDeduction(employeeData(i))**
     25. **Console.WriteLine("Completed process for employee id {0}" & "process took {1} seconds", employeeData(i).EmployeeID, span)**
     26. **Console.WriteLine()**
     27. **Next i**
     28. **End Sub**
  7. Call the method **Ex1Task1\_ParallelizeLongRunningService** from **Main()**.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex1Task1\_ParallelizeLongRunningService();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex1Task1\_ParallelizeLongRunningService()**
     14. ...
     15. End Sub
  8. Build and run the application.
  9. You should see that the employees are all processed in order of their IDs, similar to the following (the exact time to complete will vary):
     1. 
     2. Figure 1
     3. Output from non-parallel calls to a long running service
  10. To work with the parallelization features, add the following method to **Program.cs** (C#) or **Module1.vb** (Visual Basic). This code uses the **For()** method from the static **Parallel** object:
      1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_UseParallelForMethod CSharp*)
      2. C#
      3. **private static void Ex1Task1\_UseParallelForMethod()**
      4. **{**
      5. **Parallel.For(0, employeeData.Count, i =>**
      6. **{**
      7. **Console.WriteLine("Starting process for employee id {0}",**
      8. **employeeData[i].EmployeeID);**
      9. **decimal span =**
      10. **PayrollServices.GetPayrollDeduction(employeeData[i]);**
      11. **Console.WriteLine("Completed process for employee id {0}",**
      12. **employeeData[i].EmployeeID);**
      13. **Console.WriteLine();**
      14. **});**
      15. **}**
      16. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_UseParallelForMethod VB*)
      17. Visual Basic
      18. **Private Sub Ex1Task1\_UseParallelForMethod()**
      19. **Parallel.For(0, employeeData.Count,**
      20. **Sub(i)**
      21. **Console.WriteLine("Starting process for employee id {0}", employeeData(i).EmployeeID)**
      22. **Dim span As Decimal = PayrollServices.GetPayrollDeduction(employeeData(i))**
      23. **Console.WriteLine("Completed process for employee id {0}", employeeData(i).EmployeeID)**
      24. **Console.WriteLine()**
      25. **End Sub)**
      26. **End Sub**
  11. Replace the current method calls from **Main()** with a call to **Ex1Task1\_UseParallelForMethod()** method.
      1. C#
      2. static void Main(string[] args)
      3. {
      4. ...
      5. // Methods to call
      6. **Ex1Task1\_UseParallelForMethod();**
      7. ...
      8. }
      9. Visual Basic
      10. Sub Main(ByVal args() As String)
      11. ...
      12. ' Methods to call
      13. **Ex1Task1\_UseParallelForMethod()**
      14. ...
      15. End Sub
  12. Build and run the application.
  13. You should observe that the employees are not necessarily processed in the order of their IDs. You’ll also notice that multiple calls to the **GetPayrollDeduction()** method are made before the first call returns. And finally, you should observe that by running the calls in parallel, the entire job completed much faster than when run in serial.
      1. 
      2. Figure 2
      3. Output from parallel calls to a long running service
      4. **Note:** Because the loop is run in parallel, each iteration is scheduled and run individually on whatever core is available. This means that the list is not necessarily processed in order, which can drastically affect your code. You should design your code in a way that each iteration of the loop is completely independent from the others. Any single iteration should not rely on another in order to complete correctly.
  14. The Parallel Extensions library also provides a parallel version of the **foreach** structure. The following code demonstrates the non-parallel way to implement this structure. Add the following method to **Program.cs** (C#) or **Module1.vb** (Visual Basic).
      1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_StandardForEach CSharp*)
      2. C#
      3. **private static void Ex1Task1\_StandardForEach()**
      4. **{**
      5. **foreach (Employee employee in employeeData)**
      6. **{**
      7. **Console.WriteLine("Starting process for employee id {0}",**
      8. **employee.EmployeeID);**
      9. **decimal span =**
      10. **PayrollServices.GetPayrollDeduction(employee);**
      11. **Console.WriteLine("Completed process for employee id {0}",**
      12. **employee.EmployeeID);**
      13. **Console.WriteLine();**
      14. **}**
      15. **}**
      16. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_StandardForEach VB*)
      17. Visual Basic
      18. **Private Sub Ex1Task1\_StandardForEach()**
      19. **For Each employee As Employee In employeeData**
      20. **Console.WriteLine("Starting process for employee id {0}", employee.EmployeeID)**
      21. **Dim span As Decimal = PayrollServices.GetPayrollDeduction(employee)**
      22. **Console.WriteLine("Completed process for employee id {0}", employee.EmployeeID)**
      23. **Console.WriteLine()**
      24. **Next employee**
      25. **End Sub**
  15. In the **Main()** method, replace the **Parallel.For(…)** loop with the following code:
      1. C#
      2. static void Main(string[] args)
      3. {
      5. // Methods to call
      6. **Ex1Task1\_StandardForEach();**
      8. }
      9. Visual Basic
      10. Sub Main(ByVal args() As String)
      12. ' Methods to call
      13. **Ex1Task1\_StandardForEach()**
      15. End Sub
  16. Build and run the application.
      1. **Note:** You should observe that the employees are once again processed in the order of the IDs. Also take note of the total amount of time required to complete this job (the exact time required will vary)
      2. 
      3. Figure 3
      4. Output from non-parallel for…each implementation
  17. To utilize the Parallel Extensions implementation of the **for…each** structure you’ll need to change the code to use the **ForEach()** method. In **Program.cs** (C#) or **Module1.vb** (Visual Basic)add the following method:
      1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_ParallelForEach CSharp*)
      2. C#
      3. **private static void Ex1Task1\_ParallelForEach()**
      4. **{**
      5. **Parallel.ForEach(employeeData, ed =>**
      6. **{**
      7. **Console.WriteLine("Starting process for employee id {0}",**
      8. **ed.EmployeeID);**
      9. **decimal span = PayrollServices.GetPayrollDeduction(ed);**
      10. **Console.WriteLine("Completed process for employee id {0}",**
      11. **ed.EmployeeID);**
      12. **Console.WriteLine();**
      13. **});**
      14. **}**
      15. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_ParallelForEach VB*)
      16. Visual Basic
      17. **Private Sub Ex1Task1\_ParallelForEach()**
      18. **Parallel.ForEach(employeeData,**
      19. **Sub(ed)**
      20. **Console.WriteLine("Starting process for employee id {0}", ed.EmployeeID)**
      21. **Dim span As Decimal = PayrollServices.GetPayrollDeduction(ed)**
      22. **Console.WriteLine("Completed process for employee id {0}", ed.EmployeeID)**
      23. **Console.WriteLine()**
      24. **End Sub)**
      25. **End Sub**
  18. Replace the current method calls from **Main()**, with a call to **Ex1Task1\_ParallelForEach** method.
      1. C#
      2. static void Main(string[] args)
      3. {
      4. ...
      5. // Methods to call
      6. **Ex1Task1\_ParallelForEach();**
      7. ...
      8. }
      9. Visual Basic
      10. Sub Main(ByVal args() As String)
      11. ...
      12. ' Methods to call
      13. **Ex1Task1\_ParallelForEach()**
      14. ...
      15. End Sub
  19. Build and run the application.
  20. You will again observe that the employees are not necessarily processed in order of their ID and because each loop is run in parallel, each iteration of the loop is run individually on whatever core is available. Also, since the application is utilizing all available cores the job is able to complete faster than when run in a serial manner.
      1. 
      2. Figure 4
      3. Output from parallel for…each implementation
      4. **Note:** The Parallel Extensions Library also provides a useful **Invoke()** method, that allows parallel execution of anonymous methods or lambda expressions. To help illustrate how to use the Invoke method you will examine a common tree-walking algorithm and then see how it can be parallelized to reduce the total time needed to walk the entire tree.
      5. In this example you will walk an employee hierarchy and call the **GetPayrollDeduction()** method for each employee you encounter.
  21. Replace the current method calls from **Main()**, with a call to **Ex1Task1\_WalkTree()** method. This code instantiate the employee hierarchy and call the tree walker method.
      1. C#
      2. static void Main(string[] args)
      3. {
      4. ...
      5. // Methods to call
      6. **Ex1Task1\_WalkTree();**
      7. ...
      8. }
      9. Visual Basic
      10. Sub Main(ByVal args() As String)
      11. ...
      12. ' Methods to call
      13. **Ex1Task1\_WalkTree()**
      14. ...
      15. End Sub
      16. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_WalkTree CSharp*)
      17. C#
      18. **private static void Ex1Task1\_WalkTree()**
      19. **{**
      20. **EmployeeHierarchy employeeHierarchy = new EmployeeHierarchy();**
      21. **WalkTree(employeeHierarchy);**
      22. **}**
      23. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1\_WalkTree VB*)
      24. Visual Basic
      25. **Private Sub Ex1Task1\_WalkTree()**
      26. **Dim employeeHierarchy As New EmployeeHierarchy()**
      27. **WalkTree(employeeHierarchy)**
      28. **End Sub**
  22. Add the following method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
      1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 WalkTree CSharp*)
      2. C#
      3. **private static void WalkTree(Tree<Employee> node)**
      4. **{**
      5. **if (node == null)**
      6. **return;**
      7. **if (node.Data != null)**
      8. **{**
      9. **Employee emp = node.Data;**
      10. **Console.WriteLine("Starting process for employee id {0}",**
      11. **emp.EmployeeID);**
      12. **decimal span = PayrollServices.GetPayrollDeduction(emp);**
      13. **Console.WriteLine("Completed process for employee id {0}",**
      14. **emp.EmployeeID);**
      15. **Console.WriteLine();**
      16. **}**
      17. **WalkTree(node.Left);**
      18. **WalkTree(node.Right);**
      19. **}**
      20. (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 WalkTree VB*)
      21. Visual Basic
      22. **Private Sub WalkTree(ByVal node As Tree(Of Employee))**
      23. **If node Is Nothing Then**
      24. **Return**
      25. **End If**
      26. **If node.Data IsNot Nothing Then**
      27. **Dim emp As Employee = node.Data**
      28. **Console.WriteLine("Starting process for employee id {0}", emp.EmployeeID)**
      29. **Dim span As Decimal = PayrollServices.GetPayrollDeduction(emp)**
      30. **Console.WriteLine("Completed process for employee id {0}", emp.EmployeeID)**
      31. **Console.WriteLine()**
      32. **End If**
      33. **WalkTree(node.Left)**
      34. **WalkTree(node.Right)**
      35. **End Sub**
  23. Build and run the application.
  24. You should observe the employees are processed in order of their IDs. Also note the total amount of time required to walk the tree (the exact time required will vary):
      1. 
      2. Figure 5
      3. Output from a non-parallel tree walker
      4. **Note:** The tree has been structured so that the data will be written out in ID order when the tree is walked using the non-parallel algorithm provided above.
  25. To walk the tree in a parallel manner remove the two calls to **WalkTree()** at the end of the **WalkTree()** method and replace them with a call to the **Invoke()** Method of the static Parallel class:
      1. C#
      2. private static void WalkTree(Tree<Employee> node)
      3. {
      4. if (node == null)
      5. return;
      6. if (node.Data != null)
      7. {
      8. Employee emp = node.Data;
      9. Console.WriteLine("Starting process for employee id {0}",
      10. emp.EmployeeID);
      11. decimal span = PayrollServices.GetPayrollDeduction(emp);
      12. Console.WriteLine("Completed process for employee id {0}",
      13. emp.EmployeeID);
      14. Console.WriteLine();
      15. }
      16. **Parallel.Invoke(delegate { WalkTree(node.Left); }, delegate { WalkTree(node.Right); });**
      17. }
      18. Visual Basic
      19. Private Sub WalkTree(ByVal node As Tree(Of Employee))
      20. If node Is Nothing Then
      21. Return
      22. End If
      23. If node.Data IsNot Nothing Then
      24. Dim emp As Employee = node.Data
      25. Console.WriteLine("Starting process for employee id {0}", emp.EmployeeID)
      26. Dim span As Decimal = PayrollServices.GetPayrollDeduction(emp)
      27. Console.WriteLine("Completed process for employee id {0}", emp.EmployeeID)
      28. Console.WriteLine()
      29. End If
      30. **Parallel.Invoke(Sub() WalkTree(node.Left), Sub() WalkTree(node.Right))**
      31. End Sub
  26. Build and run the application.
  27. You should observe that the employees in the tree are no longer processed in the same order and that several nodes start processing before others have completed. Also note that it took less time to walk the entire tree.
      1. 
      2. Figure 6
      3. Output from a parallel tree walker
      4. **Note:** The **Invoke()** method schedules each call to **WalkTree()** individually, based on core availability. This means that the tree will not necessarily be walked in a predictable manner. Again, keep this in mind as you design your code.

# Next Step:

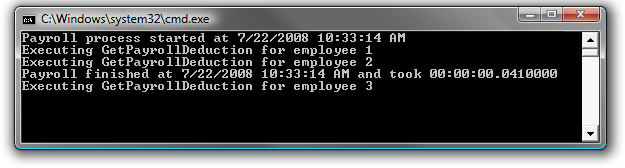
* 1. Exercise 2: Create and Run Parallelized Tasks

Exercise 2: Create and Run Parallelized Tasks

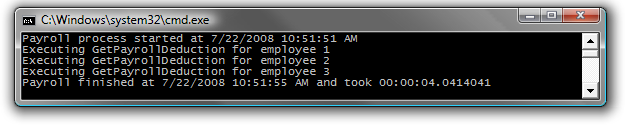
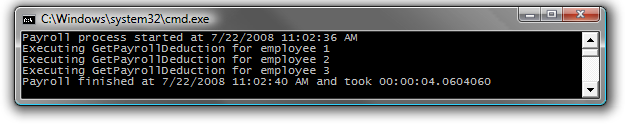
* 1. The Parallel Extensions library provides a **Task** class that can be used to execute work items in parallel, across multiple cores. Basically, you can think of a **Task** object as a lightweight unit of work that might be scheduled to run in parallel to other units, if the **TaskManager** decides it is necessary.
  2. As **Task** objects are created you need to supply them with a delegate or lambda statement containing the logic to execute. Then the **TaskManager**, which is the real heart of the Parallel Extensions library, will schedule the **Task** to execute, possibly on a different thread running on a different core.

1. **Note:** To verify that each step is correctly performed, it is recommended to build the solution at the end of each task.

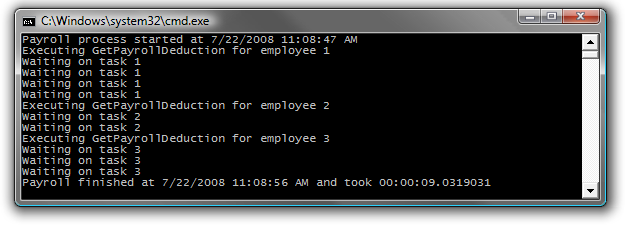
Task 1 – Natively Running Parallelized Tasks

* 1. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  2. Open the solution file **ParallelExtLab.sln** located under ***Source\Ex02-CreateAndRunParallelizedTasks\begin* (choosing** **the folder that matches the language of your preference)**. Optionally, you can continue working with the solution you created in the previous exercise.
  3. Replace the current method calls from **Main()**, with a call to **Ex2Task1\_NativeParallelTasks()** method.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex2Task1\_NativeParallelTasks();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex2Task1\_NativeParallelTasks()**
     14. ...
     15. End Sub
  4. Add the Ex2Task1\_NativeParallelTasks method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task1\_NativeParallelTasks CSharp*)
     2. C#
     3. **private static void Ex2Task1\_NativeParallelTasks()**
     4. **{**
     5. **Task task1 = Task.Factory.StartNew(delegate**
     6. **{ PayrollServices.GetPayrollDeduction(employeeData[0]); });**
     7. **Task task2 = Task.Factory.StartNew(delegate**
     8. **{ PayrollServices.GetPayrollDeduction(employeeData[1]); });**
     9. **Task task3 = Task.Factory.StartNew(delegate**
     10. **{ PayrollServices.GetPayrollDeduction(employeeData[2]); });**
     11. **}**
     12. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task1\_NativeParallelTasks VB*)
     13. Visual Basic
     14. **Private Sub Ex2Task1\_NativeParallelTasks()**
     15. **Dim task1 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(0)))**
     16. **Dim task2 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(1)))**
     17. **Dim task3 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(2)))**
     18. **End Sub**
  5. Build and run the application.
  6. You should observe that when run in parallel, some of the tasks might not complete execution until after the **Ex2Task1\_NativeParallelTasks** method has exited and control has returned to **Main**. Because of this, the output time also does not reflect the total processing time as it is likely the tasks have not completed before returning to **Main**.
     1. 
     2. Figure 7
     3. Output from running several Tasks in parallel

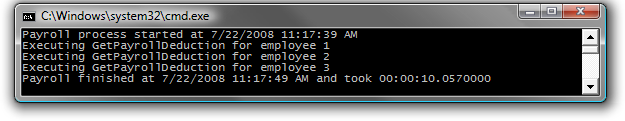
Task 2 – Using the Wait() and WaitAll() Methods

* 1. The benefit of executing tasks in parallel is faster execution and the ability to leverage multi-core processors. However, you should also notice that the current implementation introduces the possibility the main application could exit before the thread processing the task finishes.
  2. You can handle this possible situation by invoking the **Wait()** method on the individual **Task** objects. This causes the main thread to wait until the indicated tasks are complete before continuing on to the next instruction.
  3. Replace the current method calls from **Main()**, with a call to **Ex2Task2\_WaitHandling()**. This code will add wait handling to your example.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex2Task2\_WaitHandling();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex2Task2\_WaitHandling()**
     14. ...
     15. End Sub
  4. Add the Ex2Task2\_WaitHandling() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2\_WaitHandling CSharp*)
     2. C#
     3. **private static void Ex2Task2\_WaitHandling()**
     4. **{**
     5. **Task task1 = Task.Factory.StartNew(delegate**
     6. **{ PayrollServices.GetPayrollDeduction(employeeData[0]); });**
     7. **Task task2 = Task.Factory.StartNew(delegate**
     8. **{ PayrollServices.GetPayrollDeduction(employeeData[1]); });**
     9. **Task task3 = Task.Factory.StartNew(delegate**
     10. **{ PayrollServices.GetPayrollDeduction(employeeData[2]); });**
     11. **task1.Wait();**
     12. **task2.Wait();**
     13. **task3.Wait();**
     14. **}**
     15. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2\_WaitHandling VB*)
     16. C#
     17. **Private Sub Ex2Task2\_WaitHandling()**
     18. **Dim task1 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(0)))**
     19. **Dim task2 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(1)))**
     20. **Dim task3 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(2)))**
     21. **task1.Wait()**
     22. **task2.Wait()**
     23. **task3.Wait()**
     24. **End Sub**
  5. Build and run the application.
  6. You should observe that this time all three Tasks completed before the final time was reported, indicating the end of the main thread.
     1. 
     2. Figure 8
     3. Output from running tasks in parallel with individual Wait() conditions
     4. **Note:** The main thread waited for the created **Task** objects to complete before continuing operation. This approach is much simpler and cleaner than using **ThreadPool.QueueUserWorkItem**, which involves the creation and management of manual reset events, possible with the addition of Interlocked operations as well.
  7. In addition to the **Wait()** method on the individual **Task** objects, the static **Task** class also offers a **WaitAll()** method allowing you to wait on a specified list of tasks with one call. To see this method in action, remove the individual calls to **Wait()** for task1, task2, and task3 and replace them with the following:
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex2Task2\_WaitHandlingWaitAll();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex2Task2\_WaitHandlingWaitAll()**
     14. ...
     15. End Sub
  8. Add the Ex2Tas2\_WaitHandlingWaitAll() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2\_WaitHandlingWaitAll CSharp*)
     2. C#
     3. **private static void Ex2Task2\_WaitHandlingWaitAll()**
     4. **{**
     5. **Task task1 = Task.Factory.StartNew(delegate**
     6. **{ PayrollServices.GetPayrollDeduction(employeeData[0]); });**
     7. **Task task2 = Task.Factory.StartNew (delegate**
     8. **{ PayrollServices.GetPayrollDeduction(employeeData[1]); });**
     9. **Task task3 = Task.Factory.StartNew (delegate**
     10. **{ PayrollServices.GetPayrollDeduction(employeeData[2]); });**
     11. **Task.WaitAll(task1, task2, task3);**
     12. **}**
     13. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2\_WaitHandlingWaitAll VB*)
     14. Visual Basic
     15. **Private Sub Ex2Task2\_WaitHandlingWaitAll()**
     16. **Dim task1 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(0)))**
     17. **Dim task2 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(1)))**
     18. **Dim task3 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(2)))**
     19. **Task.WaitAll(task1, task2, task3)**
     20. **End Sub**
  9. Build and run the application.
  10. You should observe the main application waits until all individual Tasks are completed before continuing.
      1. 
      2. Figure 9
      3. Output from running tasks in parallel with the WaitAll() method

Task 3 – Using the IsCompleted Property

* 1. There will be times when you want to check on the completion status of a **Task** before doing other work (for instance, you may have another task to run that is dependent on the first task completing first), but you may not want to utilize the **Wait()** method because **Wait()** blocks execution on the thread you’ve launched your **Task** from. For these situations, the **Task** class exposes an **IsCompleted** property. This enables you to check whether **Task** objects have completed their work before you continue with other processing.
  2. Replace the current method calls from **Main()**, with a call to **Ex2Task3\_TaskIsCompleted()**.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex2Task3\_TaskIsCompleted();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex2Task3\_TaskIsCompleted()**
     14. ...
     15. End Sub
  3. Add the Ex2Task3\_TaskIsCompleted() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task3\_TaskIsCompleted CSharp*)
     2. C#
     3. **private static void Ex2Task3\_TaskIsCompleted()**
     4. **{**
     5. **Task task1 = Task.Factory.StartNew(delegate**
     6. **{ PayrollServices.GetPayrollDeduction(employeeData[0]); });**
     7. **while (!task1.IsCompleted)**
     8. **{**
     9. **Thread.Sleep(1000);**
     10. **Console.WriteLine("Waiting on task 1");**
     11. **}**
     12. **Task task2 = Task.Factory.StartNew(delegate**
     13. **{ PayrollServices.GetPayrollDeduction(employeeData[1]); });**
     14. **while (!task2.IsCompleted)**
     15. **{**
     16. **Thread.Sleep(1000);**
     17. **Console.WriteLine("Waiting on task 2");**
     18. **}**
     19. **Task task3 = Task.Factory.StartNew(delegate**
     20. **{ PayrollServices.GetPayrollDeduction(employeeData[2]); });**
     21. **while (!task3.IsCompleted)**
     22. **{**
     23. **Thread.Sleep(1000);**
     24. **Console.WriteLine("Waiting on task 3");**
     25. **}**
     26. **}**
     27. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task3\_TaskIsCompleted VB*)
     28. Visual Basic
     29. **Private Sub Ex2Task3\_TaskIsCompleted()**
     30. **Dim task1 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(0)))**
     31. **Do While Not task1.IsCompleted**
     32. **Thread.Sleep(1000)**
     33. **Console.WriteLine("Waiting on task 1")**
     34. **Loop**
     35. **Dim task2 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(1)))**
     36. **Do While Not task2.IsCompleted**
     37. **Thread.Sleep(1000)**
     38. **Console.WriteLine("Waiting on task 2")**
     39. **Loop**
     40. **Dim task3 As Task = Task.Factory.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(2)))**
     41. **Do While Not task3.IsCompleted**
     42. **Thread.Sleep(1000)**
     43. **Console.WriteLine("Waiting on task 3")**
     44. **Loop**
     45. **End Sub**
  4. Build and run the application.
  5. You should observe that Tasks two and three do not start until the previous Task’s **IsCompleted** property is true.
     1. 
     2. Figure 10
     3. Output from running tasks in parallel and utilizing the IsCompleted property

Task 4 – Using the ContinueWith() Method

* 1. While the **IsCompleted** property is useful for polling a **Task** to see if it is finished in order to be able to fire off more work, the **Task** class offers an even more convenient option. Using the **ContinueWith()** method makes it easy to string tasks together to run in a specific order.
  2. The functionality passed in as arguments to the **ContinueWith()** method will be executed once the **Task** object’s logic continues.
  3. Replace the current method calls from **Main()**, with a call to **Ex2Task4\_ContinueWith()**.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex2Task4\_ContinueWith();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex2Task4\_ContinueWith()**
     14. ...
     15. End Sub
  4. Add the Ex2Task4\_ContinueWith() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task4\_ContinueWith CSharp*)
     2. C#
     3. **private static void Ex2Task4\_ContinueWith()**
     4. **{**
     5. **Task task3 = Task.Factory.StartNew(delegate**
     6. **{ PayrollServices.GetPayrollDeduction(employeeData[0]); })**
     7. **.ContinueWith(delegate**
     8. **{ PayrollServices.GetPayrollDeduction(employeeData[1]); })**
     9. **.ContinueWith(delegate**
     10. **{ PayrollServices.GetPayrollDeduction(employeeData[2]); });**
     11. **task3.Wait();**
     12. **}**
     13. (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task4\_ContinueWith VB*)
     14. Visual Basic
     15. **Private Sub Ex2Task4\_ContinueWith()**
     16. **Dim task3 As Task = Task.Factory \_**
     17. **.StartNew(Sub() PayrollServices.GetPayrollDeduction(employeeData(0))) \_**
     18. **.ContinueWith(Sub() PayrollServices.GetPayrollDeduction(employeeData(1))) \_**
     19. **.ContinueWith(Sub() PayrollServices.GetPayrollDeduction(employeeData(2)))**
     20. **task3.Wait()**
     21. **End Sub**
     22. **Note:** Here you created the first **Task** as normal, but you used the **ContinueWith()** method to have the runtime execute the subsequent calls in order.
  5. Build and run the application.
  6. You should observe that the tasks execute in order – employee 1 first, followed by employee 2, and finally employee 3.
     1. 
     2. Figure 11
     3. Output from running tasks in parallel and using ContinueWith to ensure their order and wait conditions

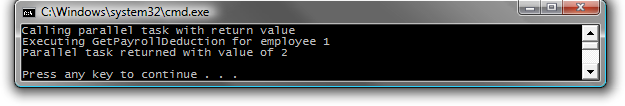
# Next Step:

* 1. Exercise 3: Use the Generic Task Class to Create and Run Tasks that Return a Value

Exercise 3: Use the Generic Task Class to Create and Run Tasks that Return a Value

* 1. As you can see, **Tasks** are useful for launching a unit of functionality in a parallel environment; they also provide a mechanism for returning values as a result of executing the unit.
  2. To demonstrate this, you are going to create a new instance of a **Task<decimal>** and then use the static **Task.Factory**.**StartNew()** method to execute the **GetPayrollDeduction()** method in a manner that allows you to capture its return value.

Task 1 – Capturing a Task’s Return Value

* 1. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  2. Open the solution file **ParallelExtLab.sln** located under **Source\Ex03-UseTaskResult\begin***.* **(choosing the folder that matches the language of your preference)** Optionally, you can continue working with the solution you created in the previous exercise.
  3. Replace the current method calls from **Main()**, with a call to **Ex3Task1\_TaskReturnValue()**.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex3Task1\_TaskReturnValue();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex3Task1\_TaskReturnValue()**
     14. ...
     15. End Sub
  4. Add the Ex3Task1\_TaskReturnValue method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex3 Ex3Task1\_TaskReturnValue CSharp*)
     2. C#
     3. **private static void Ex3Task1\_TaskReturnValue()**
     4. **{**
     5. **Console.WriteLine("Calling parallel task with return value");**
     6. **var data = Task.Factory.StartNew(() =>**
     7. **PayrollServices.GetPayrollDeduction(employeeData[0]));**
     8. **Console.WriteLine("Parallel task returned with value of {0}",**
     9. **data.Result);**
     10. **}**
     11. (Code Snippet – *Intro to Parallel Extensions Lab - Ex3 Ex3Task1\_TaskReturnValue VB*)
     12. Visual Basic
     13. **Private Sub Ex3Task1\_TaskReturnValue()**
     14. **Console.WriteLine("Calling parallel task with return value")**
     15. **Dim data As Task(Of Decimal) = Task.Factory.StartNew(Function() PayrollServices.GetPayrollDeduction(employeeData(0)))**
     16. **Console.WriteLine("Parallel task returned with value of {0}", data.Result)**
     17. **End Sub**
     18. **Note:** The value is captured by inspecting the **data.Result** property. If the task has completed when the **Result** property is invoked then it will return the captured value immediately, otherwise it will block the executing code until the task has completed and the value can be retrieved. In the above example, you are accessing the **Result** property right away, which is not the ideal situation. Where **Task<T>** becomes very useful is when you are firing off units of works where you will not be retrieving the returned values until a later time.
  5. Build and run the application.
  6. You should observe that the task completes and a return value is provided.
     1. 
     2. Figure 12
     3. Output from running a Task to capture a return value

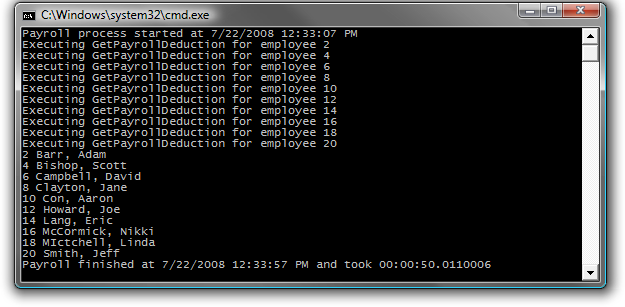
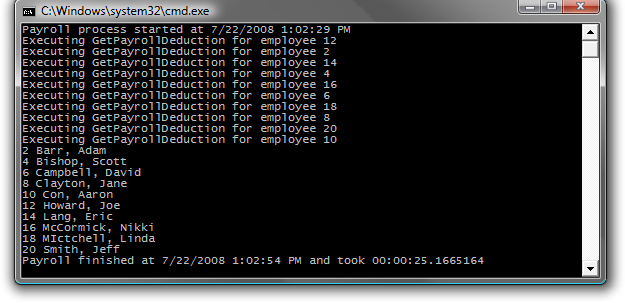
# Next Step:

* 1. Exercise 4: Parallelize LINQ Queries using PLINQ

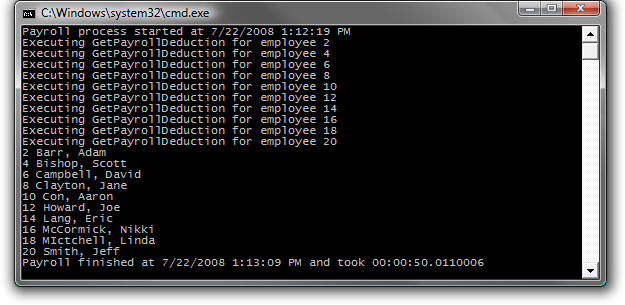
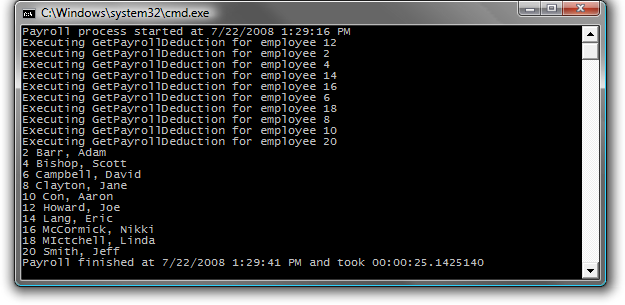
Exercise 4: Parallelize LINQ Queries using PLINQ

* 1. Developers can optimize their LINQ queries to execute in a parallel environment by utilizing Parallelized LINQ (PLINQ).
  2. The Parallel Extensions library offers many different ways to implement parallelism in LINQ queries. PLINQ provides you with the **System.Linq.ParallelEnumerable** class which offers functionality similar to the **System.Linq.Enumerable** class.

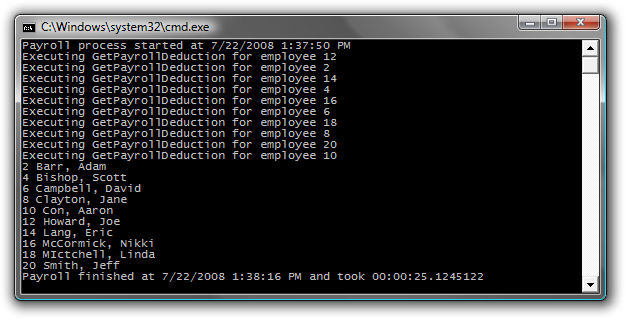
Task 1 – Using the ParallelEnumerable Class’ Static Methods to Parallelize LINQ

* 1. In this task, you will continue to use the same solution as the previous exercises.
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. Open the solution file **ParallelExtLab.sln** located under **Source\Ex04-PLINQ\begin****(Choosing the folder that matches the language of your preference.)**Optionally, you can continue working the solution you created in the previous exercise.
  4. Replace the current method calls from **Main()**, with a call to **Ex4Task1\_PLINQ()**.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex4Task1\_PLINQ();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex4Task1\_PLINQ()**
     14. ...
     15. End Sub
  5. Add the Ex4Task1\_PLINQ() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1\_PLINQ CSharp*)
     2. C#
     3. **private static void Ex4Task1\_PLINQ()**
     4. **{**
     5. **var q = Enumerable.Select(**
     6. **Enumerable.OrderBy(**
     7. **Enumerable.Where(employeeData,**
     8. **x => x.EmployeeID % 2 == 0),**
     9. **x => x.EmployeeID),**
     10. **x => PayrollServices.GetEmployeeInfo(x))**
     11. **.ToList();**
     12. **foreach (var e in q)**
     13. **{**
     14. **Console.WriteLine(e);**
     15. **}**
     16. **}**
     17. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1\_PLINQ VB*)
     18. Visual Basic
     19. **Private Sub Ex4Task1\_PLINQ()**
     20. **Dim q = Enumerable.Select(**
     21. **Enumerable.OrderBy(**
     22. **Enumerable.Where(**
     23. **employeeData,**
     24. **Function(x) x.EmployeeID Mod 2 = 0),**
     25. **Function(x) x.EmployeeID),**
     26. **Function(x) PayrollServices.GetEmployeeInfo(x)) \_**
     27. **.ToList()**
     28. **For Each e In q**
     29. **Console.WriteLine(e)**
     30. **Next e**
     31. **End Sub**
     32. **Note:** The **Select()**, **OrderBy()**, and **Where()** methods are extension methods off of the **IEnumerable** generic class, however you are accessing them in a static manner here. You will try a more succinct usage later.
     33. The **ToList()** call is for illustrative purposes and is not always necessary in production code. It is used here because you want to immediately fire the LINQ query to collect all of the **Employee Info** strings, and then write them out to screen later.
     34. If you were to leave the **ToList()** off, the query will still fire in order of the **Employee ID** but each call to **GetEmployeeInfo()** wouldn’t fire until the **IEnumerable generic** is iterated through during the **foreach** loop. This is known as Delayed Execution.
     35. See Scott Wisniewski’s article at <http://msdn.microsoft.com/en-us/magazine/cc163378.aspx> for more information.
  6. Build and run the application.
  7. You should observe that the LINQ query performs the operations in order of the **Employee ID**. Also observe the total amount of time required to complete the work (the exact time required will vary):
     1. 
     2. Figure 13
     3. Output from a non-parallelized LINQ query
  8. It is easy to parallelize this query by making use of the static **ParallelEnumerable** class’s version of the same LINQ methods. Additionally you’ll need to add an **AsParallel()** call to the query’s data source. Modify the **Main()** method just to call the PLINQAsParallel method.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex4Task1\_PLINQAsParallel();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex4Task1\_PLINQAsParallel()**
     14. ...
     15. End Sub
  9. Add the Ex4Task1\_PLINQAsParallel() to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1\_PLINQAsParallel CSharp*)
     2. C#
     3. **private static void Ex4Task1\_PLINQAsParallel()**
     4. **{**
     5. **var q = ParallelEnumerable.Select(**
     6. **ParallelEnumerable.OrderBy(**
     7. **ParallelEnumerable.Where(employeeData.AsParallel(),**
     8. **x => x.EmployeeID % 2 == 0),**
     9. **x => x.EmployeeID),**
     10. **x => PayrollServices.GetEmployeeInfo(x))**
     11. **.ToList();**
     12. **foreach (var e in q)**
     13. **{**
     14. **Console.WriteLine(e);**
     15. **}**
     16. **}**
     17. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1\_PLINQAsParallel VB*)
     18. Visual Basic
     19. **Private Sub Ex4Task1\_PLINQAsParallel()**
     20. **Dim q = ParallelEnumerable.Select(**
     21. **ParallelEnumerable.OrderBy(**
     22. **ParallelEnumerable.Where(**
     23. **employeeData.AsParallel(),**
     24. **Function(x) x.EmployeeID Mod 2 = 0),**
     25. **Function(x) x.EmployeeID),**
     26. **Function(x) PayrollServices.GetEmployeeInfo(x)) \_**
     27. **.ToList()**
     28. **For Each e In q**
     29. **Console.WriteLine(e)**
     30. **Next e**
     31. **End Sub**
     32. **Note:** The calls to the **Select()**, **OrderBy()**, and **Where()** methods, which were previously being made on **Enumerable**, are now being made on **ParallelEnumerable**. Also notice that a call to **AsParallel()** has been added to the data source.
  10. Build and run the application.
  11. You should observe the LINQ query no longer performs the operations in a particular order. Also note that in this example the parallelized version completes in less time than the non-parallelized version (in this case, it completed in roughly half the time due to it being run on a dual-core machine; your results will vary based on the hardware you run this example on).
      1. 
      2. Figure 14
      3. Output from a parallelized LINQ query
      4. **Note:** The operations are executed in parallel with as many operations occurring concurrently as the number of physical cores will allow.

Task 2 – Using the ParallelEnumerable Class’ Extension Methods to Parallelize LINQ

* 1. As mentioned earlier, a more succinct way to take advantage of the **Enumerable** and **ParallelEnumerable** classes’ static LINQ methods is to use them as Extension methods.
  2. Converting a non-parallelized LINQ query implemented using extension methods to a PLINQ query is straight forward. Replace the PLINQ query in the **Main()** method to match the following LINQ query:
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex4Task2\_Extensions();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex4Task2\_Extensions()**
     14. ...
     15. End Sub
  3. Add the Ex4Task2\_Extensions() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2\_Extensions CSharp*)
     2. C#
     3. **private static void Ex4Task2\_Extensions()**
     4. **{**
     5. **var q = employeeData.**
     6. **Where(x => x.EmployeeID % 2 == 0).OrderBy(x => x.EmployeeID)**
     7. **.Select(x => PayrollServices.GetEmployeeInfo(x))**
     8. **.ToList();**
     9. **foreach (var e in q)**
     10. **{**
     11. **Console.WriteLine(e);**
     12. **}**
     13. **}**
     14. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2\_Extensions VB*)
     15. Visual Basic
     16. **Private Sub Ex4Task2\_Extensions()**
     17. **Dim q = employeeData \_**
     18. **.Where(Function(x) x.EmployeeID Mod 2 = 0) \_**
     19. **.OrderBy(Function(x) x.EmployeeID) \_**
     20. **.Select(Function(x) PayrollServices.GetEmployeeInfo(x)) \_**
     21. **.ToList()**
     22. **For Each e In q**
     23. **Console.WriteLine(e)**
     24. **Next e**
     25. **End Sub**
     26. **Note:** Again, the **ToList()** used to execute the LINQ query immediately rather than waiting for it to execute when the **IEnumerable<T>** returned by **Select()** is iterated over during the **foreach** that comes later. You are avoiding Delayed Execution.
  4. Build and run the application.
  5. You should observe that the LINQ query performs the operations in order of the Employee ID. Also observe the total amount of time required to complete the work (the exact time required will vary):
     1. 
     2. Figure 15
     3. Output from non-parallelized LINQ query with extension methods
  6. To parallelize this LINQ query just replace the current method call from **Main()**, with the following one.
     1. C#
     2. static void Main(string[] args)
     3. {
     4. ...
     5. // Methods to call
     6. **Ex4Task2\_ConvertToParallelExtensions();**
     7. ...
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex4Task2\_ConvertToParallelExtensions()**
     14. ...
     15. End Sub
  7. Add the Ex4Task2\_ConvertToParallelExtensions() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2\_ConvertToParallelExtensions CSharp*)
     2. C#
     3. **private static void Ex4Task2\_ConvertToParallelExtensions()**
     4. **{**
     5. **var q = employeeData.AsParallel()**
     6. **.Where(x => x.EmployeeID % 2 == 0).OrderBy(x => x.EmployeeID)**
     7. **.Select(x => PayrollServices.GetEmployeeInfo(x))**
     8. **.ToList();**
     9. **foreach (var e in q)**
     10. **{**
     11. **Console.WriteLine(e);**
     12. **}**
     13. **}**
     14. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2\_ConvertToParallelExtensions VB*)
     15. Visual Basic
     16. **Private Sub Ex4Task2\_ConvertToParallelExtensions()**
     17. **Dim q = employeeData.AsParallel() \_**
     18. **.Where(Function(x) x.EmployeeID Mod 2 = 0) \_**
     19. **.OrderBy(Function(x) x.EmployeeID) \_**
     20. **.Select(Function(x) PayrollServices.GetEmployeeInfo(x)) \_**
     21. **.ToList()**
     22. **For Each e In q**
     23. **Console.WriteLine(e)**
     24. **Next e**
     25. **End Sub**
  8. Build and run the application.
  9. You should observe that like the LINQ query based on the **ParallelEnumerable** static class, the PLINQ query implemented as extension methods no longer performs operations in order by **EmployeeID**. Instead the operations are executed in parallel with as many operations occurring concurrently as the number of physical cores will allow. Also note that as in the previous parallelize LINQ example, the parallelized version completes in roughly half the time as the non-parallelized version.
     1. 
     2. Figure 16
     3. Output from parallelized LINQ query with extension methods

Task 3 – Using AsParallel() With Query Comprehension Syntax

* 1. In this task you will use the Parallel Extensions library and the **AsParallel()** method to create parallelized LINQ queries using the query comprehension syntax.
  2. Replace the LINQ query in the **Main()** method with the following query comprehension syntax:
     1. C#
     2. static void Main(string[] args)
     3. {
     5. // Methods to call
     6. **Ex4Task3\_PLINQComprehensionSyntax();**
     8. }
     9. Visual Basic
     10. Sub Main(ByVal args() As String)
     11. ...
     12. ' Methods to call
     13. **Ex4Task3\_PLINQComprehensionSyntax()**
     14. ...
     15. End Sub
  3. Add the Ex4Task3\_PLINQComprehensionSyntax() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):
     1. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task3\_PLINQComprehensionSyntax CSharp*)
     2. C#
     3. **private static void Ex4Task3\_PLINQComprehensionSyntax()**
     4. **{**
     5. **var q = from e in employeeData.AsParallel()**
     6. **where e.EmployeeID % 2 == 0**
     7. **orderby e.EmployeeID**
     8. **select PayrollServices.GetEmployeeInfo(e);**
     9. **foreach (var e in q)**
     10. **{**
     11. **Console.WriteLine(e);**
     12. **}**
     13. **}**
     14. (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task3\_PLINQComprehensionSyntax VB*)
     15. Visual Basic
     16. **Private Sub Ex4Task3\_PLINQComprehensionSyntax()**
     17. **Dim q = From e In employeeData.AsParallel()**
     18. **Where e.EmployeeID Mod 2 = 0**
     19. **Order By e.EmployeeID**
     20. **Select PayrollServices.GetEmployeeInfo(e)**
     21. **For Each e In q**
     22. **Console.WriteLine(e)**
     23. **Next e**
     24. **End Sub**
  4. Build and run the application.
  5. You should observe that although the LINQ syntax was changed, the data was processed in the same parallel manner as it was with the **ParallelEnumerable** extension methods.
     1. 
     2. Figure 17
     3. Output from parallelized LINQ query using the query comprehension syntax

# Next Step:

* 1. Summary

Summary

In this lab, you have worked with the Parallel Extensions library to understand its features to help you work with parallel tasks in a simple, controllable fashion. You’ve learned how to use Parallel Extensions classes like Parallel and Task to manage units of work. You’ve dealt with Parallel Extensions features like Wait(), WaitAll(), IsComplete(), and ContinueWith() to control the flow of your execution. You’ve also worked through examples of using PLINQ to deal with parallelizing queries.

This lab has given you a solid introduction to the Parallel Extensions library and its power and benefits. For more education we recommend you visit these locations:

* + The Parallel Extensions blog on MSDN: <http://blogs.msdn.com/pfxteam/>
  + The Parallel Computing Forms on MSDN: <http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=551&SiteID=1>
  1. The Parallel Computing Developer Center: <http://msdn.microsoft.com/en-us/concurrency/default.aspx>.