Microsoft

Parallel Computing Platform

Hands-On Lab

Introduction to the Parallel Extensions Library

Lab version: 1.0.0

Last updated: 3/13/2010

developer & platform **evangelism**

CONTENTS

OVERVIEW **3**

EXERCISE 1: PARALLELIZE AN EXISTING ALGORITHM USING THE STATIC PARALLEL HELPER CLASS **5**

Task 1 – Parallelizing a Long Running Service 5

EXERCISE 2: CREATE AND RUN PARALLELIZED TASKS **23**

Task 1 – Natively Running Parallelized Tasks 23

Task 2 – Using the Wait() and WaitAll() Methods 25

Task 3 – Using the IsCompleted Property 28

Task 4 – Using the ContinueWith() Method 31

EXERCISE 3: USE THE GENERIC TASK CLASS TO CREATE AND RUN TASKS THAT RETURN A VALUE **33**

Task 1 – Capturing a Task's Return Value 33

EXERCISE 4: PARALLELIZE LINQ QUERIES USING PLINQ **35**

Task 1 – Using the ParallelEnumerable Class' Static Methods to Parallelize LINQ 35

Task 2 – Using the ParallelEnumerable Class' Extension Methods to Parallelize LINQ 40

Task 3 – Using AsParallel() With Query Comprehension Syntax 43

SUMMARY **45**

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

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

You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- Net Framework 4

Setup

All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.

Note: To perform the setup steps you need to run the scripts in a command window with administrator privileges.

1. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, run the **CheckDependencies.cmd** script located under the **Setup** folder. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.

   Note: For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **CheckDependencies.cmd** file launches the Visual Studio installer file that installs the code snippets.

Exercises

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.

Estimated time to complete this lab: **60 minutes**.

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:

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

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

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()**.

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

In this task you will write some simple sample code that will simulate a long running service call.

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.

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\Ex01-UsingStaticParallelHelper\begin***. **(Choosing the folder that matches the language of your preference)**

   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.

3. 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:

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Create employee list CSharp*)

   C#

   ```
   class Program
   {
       private static EmployeeList employeeData;
   ```

```
    static void Main(string[] args)
    {
       employeeData = new EmployeeList();

       Console.WriteLine("Payroll process started at {0}", DateTime.Now);
       var sw = Stopwatch.StartNew();

       // Methods to call

       Console.WriteLine("Payroll finished at {0} and took {1}",
                             DateTime.Now, sw.Elapsed.TotalSeconds);
       Console.WriteLine();
       Console.ReadLine();
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Create employee list VB*)

Visual Basic

```
Module Module1
    Private employeeData As EmployeeList

    Sub Main(ByVal args() As String)
        employeeData = New EmployeeList()

        Console.WriteLine("Payroll process started at {0}", DateTime.Now)
        Dim sw = Stopwatch.StartNew()

        ' Methods to call

        Console.WriteLine("Payroll finished at {0} and took {1}",
DateTime.Now, sw.Elapsed.TotalSeconds)
        Console.WriteLine()
        Console.ReadLine()
    End Sub
End Module
```

4. 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:

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_ParallelizeLongRunningService CSharp*)

C#

```
private static void Ex1Task1_ParallelizeLongRunningService()
{
    Console.WriteLine("Non-parallelized for loop");

    for (int i = 0; i < employeeData.Count; i++)
    {
        Console.WriteLine("Starting process for employee id {0}",
            employeeData[i].EmployeeID);
        decimal span =
            PayrollServices.GetPayrollDeduction(employeeData[i]);
        Console.WriteLine("Completed process for employee id {0}" +
            "process took {1} seconds",
            employeeData[i].EmployeeID, span);
        Console.WriteLine();
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_ParallelizeLongRunningService Visual Basic*)

Visual Basic

```
Private Sub Ex1Task1_ParallelizeLongRunningService()
    Console.WriteLine("Non-parallelized for loop")

    For i = 0 To employeeData.Count - 1
        Console.WriteLine("Starting process for employee id {0}",
employeeData(i).EmployeeID)
        Dim span As Decimal =
PayrollServices.GetPayrollDeduction(employeeData(i))
        Console.WriteLine("Completed process for employee id {0}" & "process
took {1} seconds", employeeData(i).EmployeeID, span)
        Console.WriteLine()
    Next i
End Sub
```

5. Call the method **Ex1Task1_ParallelizeLongRunningService** from **Main()**.

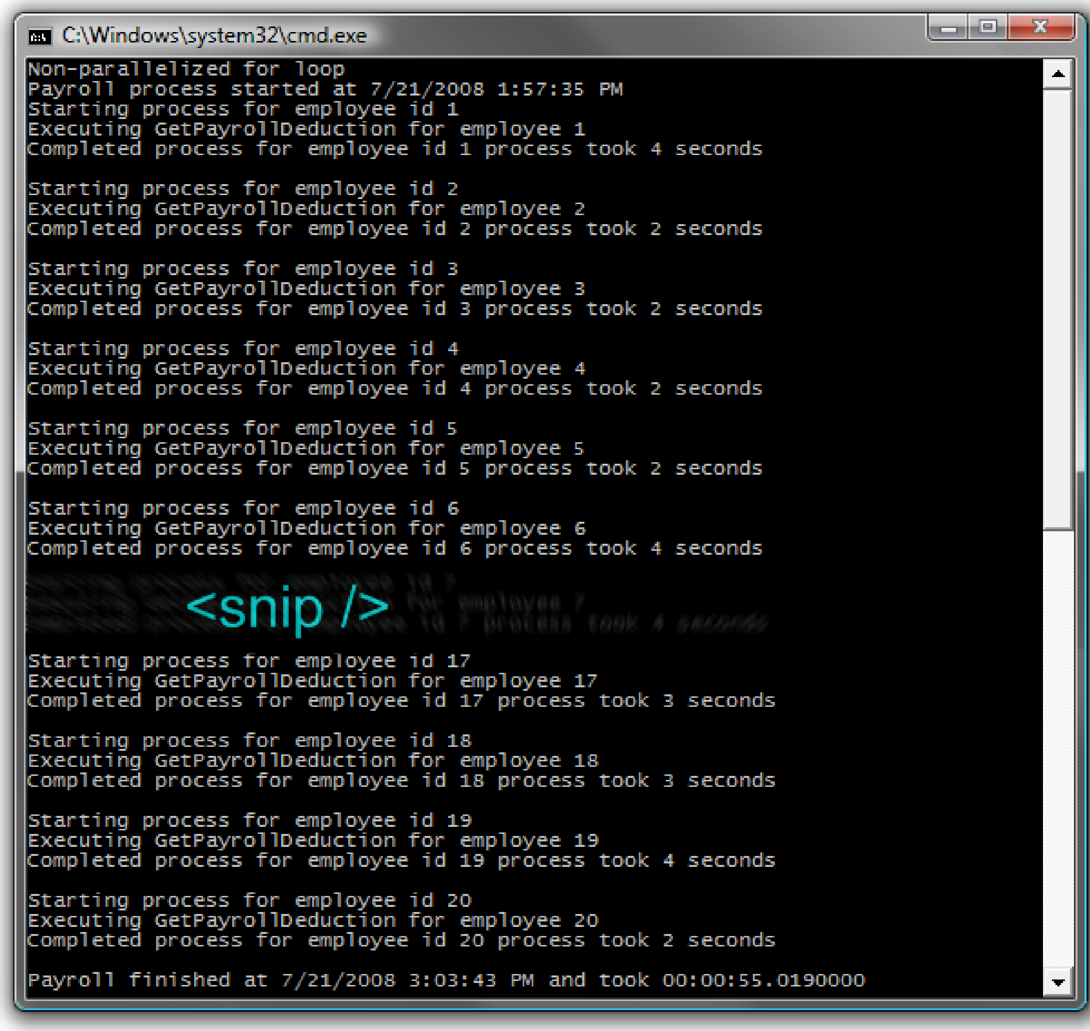
C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex1Task1_ParallelizeLongRunningService();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex1Task1_ParallelizeLongRunningService()
    ...
End Sub
```

6. Build and run the application.
7. 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):



```
Non-parallelized for loop
Payroll process started at 7/21/2008 1:57:35 PM
Starting process for employee id 1
Executing GetPayrollDeduction for employee 1
Completed process for employee id 1 process took 4 seconds

Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Completed process for employee id 2 process took 2 seconds

Starting process for employee id 3
Executing GetPayrollDeduction for employee 3
Completed process for employee id 3 process took 2 seconds

Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Completed process for employee id 4 process took 2 seconds

Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Completed process for employee id 5 process took 2 seconds

Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 6 process took 4 seconds

<snip />

Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 17 process took 3 seconds

Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Completed process for employee id 18 process took 3 seconds

Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 19 process took 4 seconds

Starting process for employee id 20
Executing GetPayrollDeduction for employee 20
Completed process for employee id 20 process took 2 seconds

Payroll finished at 7/21/2008 3:03:43 PM and took 00:00:55.0190000
```

Figure 1
Output from non-parallel calls to a long running service

8. 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:

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_UseParallelForMethod CSharp*)

C#

```
private static void Ex1Task1_UseParallelForMethod()
{
    Parallel.For(0, employeeData.Count, i =>
    {
        Console.WriteLine("Starting process for employee id {0}",
                                   employeeData[i].EmployeeID);
        decimal span =
            PayrollServices.GetPayrollDeduction(employeeData[i]);
        Console.WriteLine("Completed process for employee id {0}",
                                  employeeData[i].EmployeeID);
        Console.WriteLine();
    });
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_UseParallelForMethod VB*)

Visual Basic

```
Private Sub Ex1Task1_UseParallelForMethod()
    Parallel.For(0, employeeData.Count,
                 Sub(i)
                     Console.WriteLine("Starting process for employee id {0}",
employeeData(i).EmployeeID)
                     Dim span As Decimal =
PayrollServices.GetPayrollDeduction(employeeData(i))
                     Console.WriteLine("Completed process for employee id
{0}", employeeData(i).EmployeeID)
                     Console.WriteLine()
                 End Sub)
End Sub
```

9. Replace the current method calls from **Main()** with a call to **Ex1Task1_UseParallelForMethod()** method.

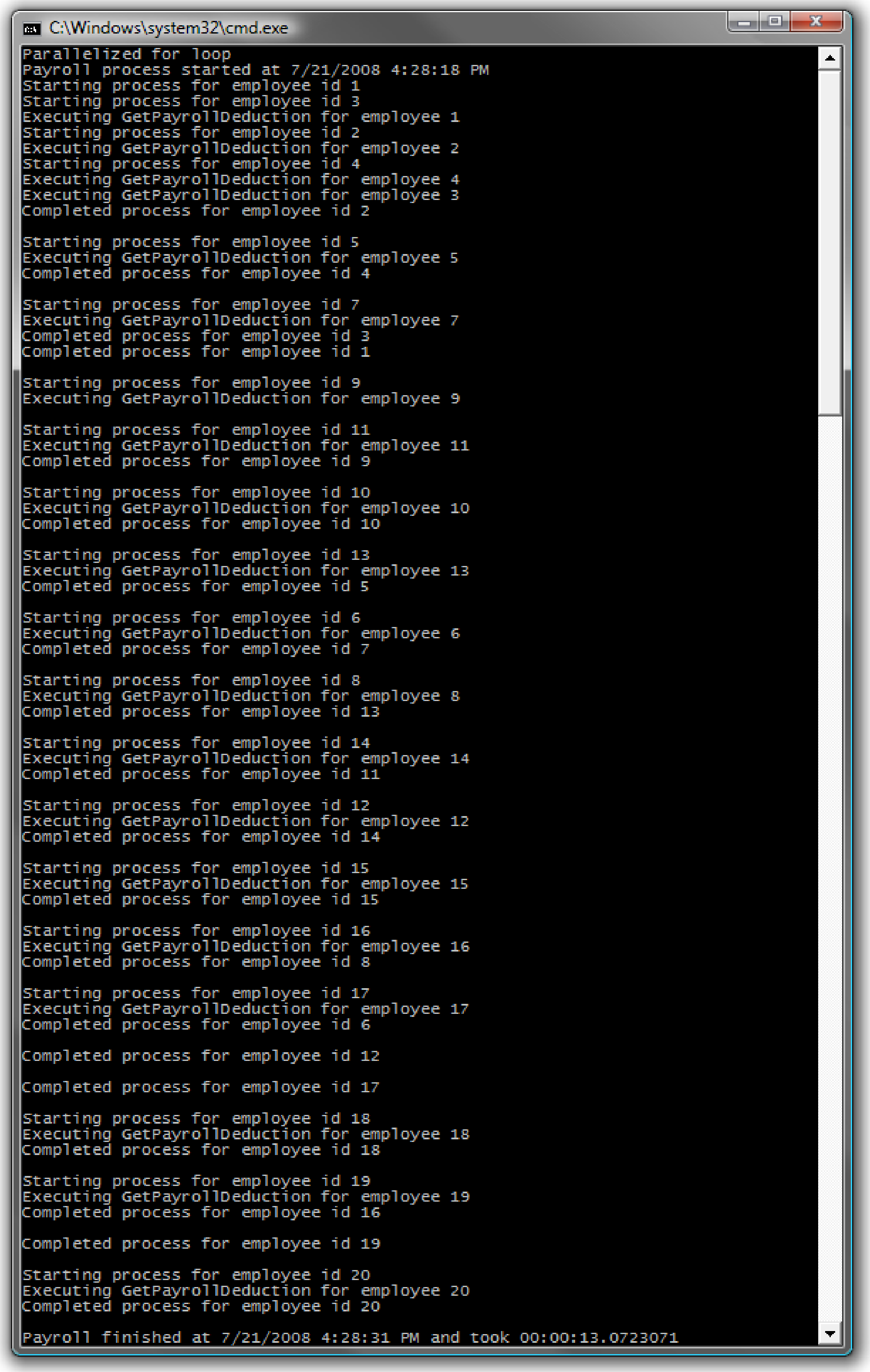
C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex1Task1_UseParallelForMethod();
    ...
}
```

Visual Basic

```vb
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex1Task1_UseParallelForMethod()
    ...
End Sub
```

10. Build and run the application.
11. 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:



```
Parallelized for loop
Payroll process started at 7/21/2008 4:28:18 PM
Starting process for employee id 1
Starting process for employee id 3
Executing GetPayrollDeduction for employee 1
Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 3
Completed process for employee id 2

Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Completed process for employee id 4

Starting process for employee id 7
Executing GetPayrollDeduction for employee 7
Completed process for employee id 3
Completed process for employee id 1

Starting process for employee id 9
Executing GetPayrollDeduction for employee 9

Starting process for employee id 11
Executing GetPayrollDeduction for employee 11
Completed process for employee id 9

Starting process for employee id 10
Executing GetPayrollDeduction for employee 10
Completed process for employee id 10

Starting process for employee id 13
Executing GetPayrollDeduction for employee 13
Completed process for employee id 5

Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 7

Starting process for employee id 8
Executing GetPayrollDeduction for employee 8
Completed process for employee id 13

Starting process for employee id 14
Executing GetPayrollDeduction for employee 14
Completed process for employee id 11

Starting process for employee id 12
Executing GetPayrollDeduction for employee 12
Completed process for employee id 14

Starting process for employee id 15
Executing GetPayrollDeduction for employee 15
Completed process for employee id 15

Starting process for employee id 16
Executing GetPayrollDeduction for employee 16
Completed process for employee id 8

Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 6

Completed process for employee id 12

Completed process for employee id 17

Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Completed process for employee id 18

Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 16

Completed process for employee id 19

Starting process for employee id 20
Executing GetPayrollDeduction for employee 20
Completed process for employee id 20

Payroll finished at 7/21/2008 4:28:31 PM and took 00:00:13.0723071
```

Figure 2

Output from parallel calls to a long running service

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.

12. 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).

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_StandardForEach CSharp*)

C#

```
private static void Ex1Task1_StandardForEach()
{
    foreach (Employee employee in employeeData)
    {
        Console.WriteLine("Starting process for employee id {0}",
            employee.EmployeeID);
        decimal span =
            PayrollServices.GetPayrollDeduction(employee);
        Console.WriteLine("Completed process for employee id {0}",
            employee.EmployeeID);
        Console.WriteLine();
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_StandardForEach VB*)

Visual Basic

```
Private Sub Ex1Task1_StandardForEach()
    For Each employee As Employee In employeeData
        Console.WriteLine("Starting process for employee id {0}",
employee.EmployeeID)
        Dim span As Decimal = PayrollServices.GetPayrollDeduction(employee)
        Console.WriteLine("Completed process for employee id {0}",
employee.EmployeeID)
        Console.WriteLine()
    Next employee
End Sub
```

13. In the **Main()** method, replace the **Parallel.For(…)** loop with the following code:

C#

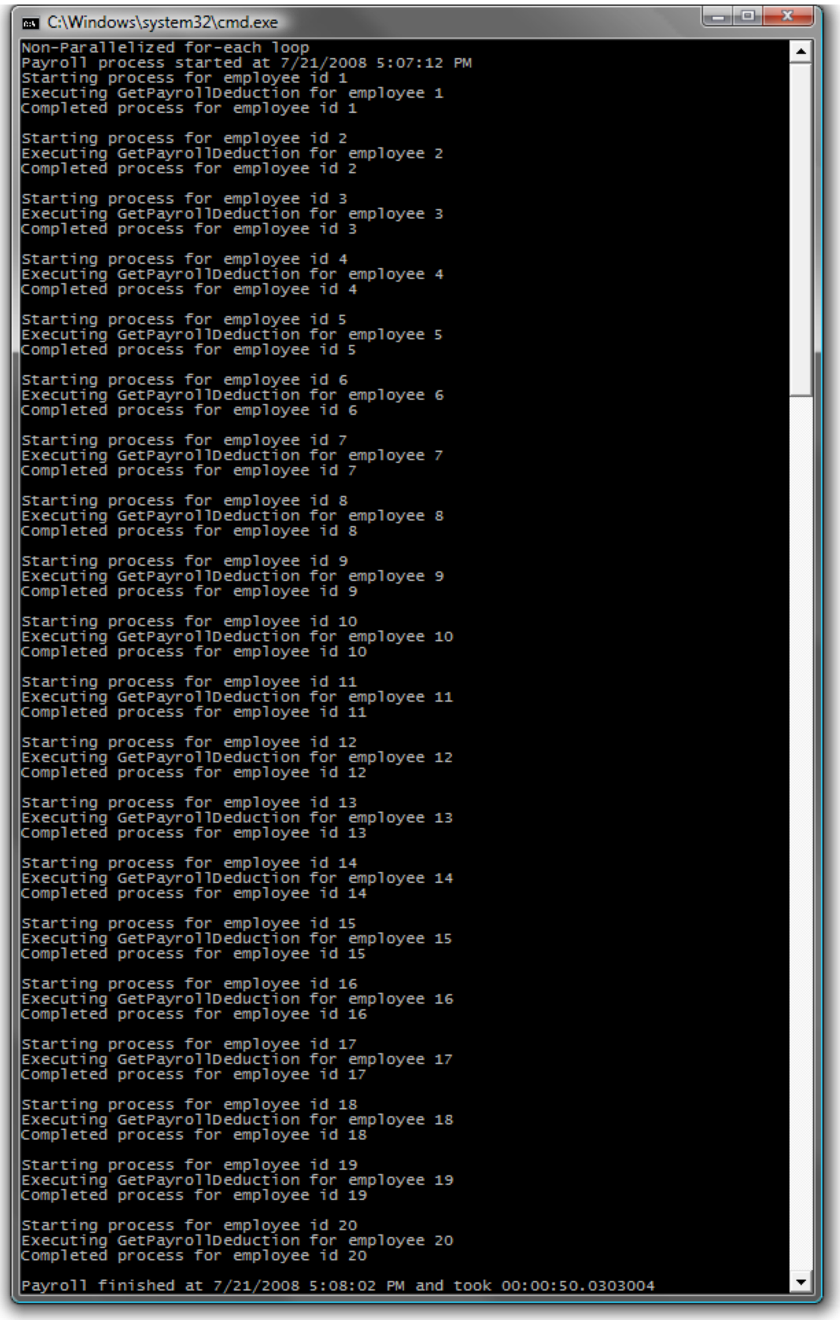
```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex1Task1_StandardForEach();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex1Task1_StandardForEach()
    ...
End Sub
```

14. Build and run the application.

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)



```
Non-Parallelized for-each loop
Payroll process started at 7/21/2008 5:07:12 PM
Starting process for employee id 1
Executing GetPayrollDeduction for employee 1
Completed process for employee id 1

Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Completed process for employee id 2

Starting process for employee id 3
Executing GetPayrollDeduction for employee 3
Completed process for employee id 3

Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Completed process for employee id 4

Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Completed process for employee id 5

Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 6

Starting process for employee id 7
Executing GetPayrollDeduction for employee 7
Completed process for employee id 7

Starting process for employee id 8
Executing GetPayrollDeduction for employee 8
Completed process for employee id 8

Starting process for employee id 9
Executing GetPayrollDeduction for employee 9
Completed process for employee id 9

Starting process for employee id 10
Executing GetPayrollDeduction for employee 10
Completed process for employee id 10

Starting process for employee id 11
Executing GetPayrollDeduction for employee 11
Completed process for employee id 11

Starting process for employee id 12
Executing GetPayrollDeduction for employee 12
Completed process for employee id 12

Starting process for employee id 13
Executing GetPayrollDeduction for employee 13
Completed process for employee id 13

Starting process for employee id 14
Executing GetPayrollDeduction for employee 14
Completed process for employee id 14

Starting process for employee id 15
Executing GetPayrollDeduction for employee 15
Completed process for employee id 15

Starting process for employee id 16
Executing GetPayrollDeduction for employee 16
Completed process for employee id 16

Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 17

Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Completed process for employee id 18

Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 19

Starting process for employee id 20
Executing GetPayrollDeduction for employee 20
Completed process for employee id 20

Payroll finished at 7/21/2008 5:08:02 PM and took 00:00:50.0303004
```

Figure 3
Output from non-parallel for…each implementation

15. 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:

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_ParallelForEach CSharp*)

C#

```
private static void Ex1Task1_ParallelForEach()
{
    Parallel.ForEach(employeeData, ed =>
    {
        Console.WriteLine("Starting process for employee id {0}",
            ed.EmployeeID);
        decimal span = PayrollServices.GetPayrollDeduction(ed);
        Console.WriteLine("Completed process for employee id {0}",
            ed.EmployeeID);
        Console.WriteLine();
    });
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_ParallelForEach VB*)

Visual Basic

```
Private Sub Ex1Task1_ParallelForEach()
    Parallel.ForEach(employeeData,
                     Sub(ed)
                         Console.WriteLine("Starting process for employee id
{0}", ed.EmployeeID)
                         Dim span As Decimal =
PayrollServices.GetPayrollDeduction(ed)
                         Console.WriteLine("Completed process for employee id
{0}", ed.EmployeeID)
                         Console.WriteLine()
                     End Sub)
End Sub
```

16. Replace the current method calls from **Main()**, with a call to **Ex1Task1_ParallelForEach** method.

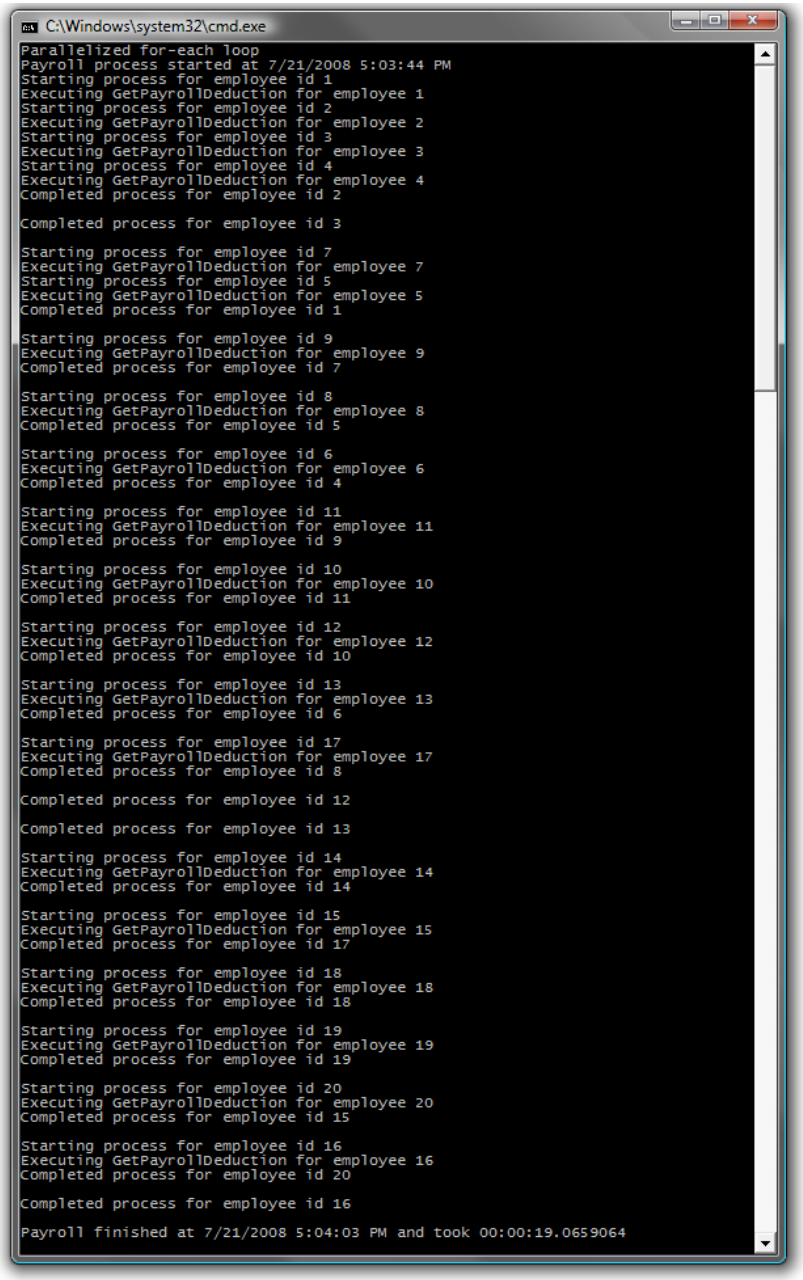
C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex1Task1_ParallelForEach();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex1Task1_ParallelForEach()
    ...
End Sub
```

17. Build and run the application.
18. 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.



```
Parallelized for-each loop
Payroll process started at 7/21/2008 5:03:44 PM
Starting process for employee id 1
Executing GetPayrollDeduction for employee 1
Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Starting process for employee id 3
Executing GetPayrollDeduction for employee 3
Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Completed process for employee id 2

Completed process for employee id 3

Starting process for employee id 7
Executing GetPayrollDeduction for employee 7
Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Completed process for employee id 1

Starting process for employee id 9
Executing GetPayrollDeduction for employee 9
Completed process for employee id 7

Starting process for employee id 8
Executing GetPayrollDeduction for employee 8
Completed process for employee id 5

Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 4

Starting process for employee id 11
Executing GetPayrollDeduction for employee 11
Completed process for employee id 9

Starting process for employee id 10
Executing GetPayrollDeduction for employee 10
Completed process for employee id 11

Starting process for employee id 12
Executing GetPayrollDeduction for employee 12
Completed process for employee id 10

Starting process for employee id 13
Executing GetPayrollDeduction for employee 13
Completed process for employee id 6

Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 8

Completed process for employee id 12

Completed process for employee id 13

Starting process for employee id 14
Executing GetPayrollDeduction for employee 14
Completed process for employee id 14

Starting process for employee id 15
Executing GetPayrollDeduction for employee 15
Completed process for employee id 17

Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Completed process for employee id 18

Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 19

Starting process for employee id 20
Executing GetPayrollDeduction for employee 20
Completed process for employee id 15

Starting process for employee id 16
Executing GetPayrollDeduction for employee 16
Completed process for employee id 20

Completed process for employee id 16

Payroll finished at 7/21/2008 5:04:03 PM and took 00:00:19.0659064
```

Figure 4
Output from parallel for…each implementation

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.

In this example you will walk an employee hierarchy and call the **GetPayrollDeduction()** method for each employee you encounter.

19. 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.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex1Task1_WalkTree();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex1Task1_WalkTree()
    ...
End Sub
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_WalkTree CSharp*)

C#

```
private static void Ex1Task1_WalkTree()
{
    EmployeeHierarchy employeeHierarchy = new EmployeeHierarchy();
    WalkTree(employeeHierarchy);
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 Ex1Task1_WalkTree VB*)

Visual Basic

```
Private Sub Ex1Task1_WalkTree()
    Dim employeeHierarchy As New EmployeeHierarchy()
    WalkTree(employeeHierarchy)
End Sub
```

20. Add the following method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 WalkTree CSharp*)

C#

```
private static void WalkTree(Tree<Employee> node)
{
    if (node == null)
        return;

    if (node.Data != null)
    {
        Employee emp = node.Data;
        Console.WriteLine("Starting process for employee id {0}",
            emp.EmployeeID);
        decimal span = PayrollServices.GetPayrollDeduction(emp);
        Console.WriteLine("Completed process for employee id {0}",
            emp.EmployeeID);
        Console.WriteLine();
    }

    WalkTree(node.Left);
    WalkTree(node.Right);
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex1 WalkTree VB*)

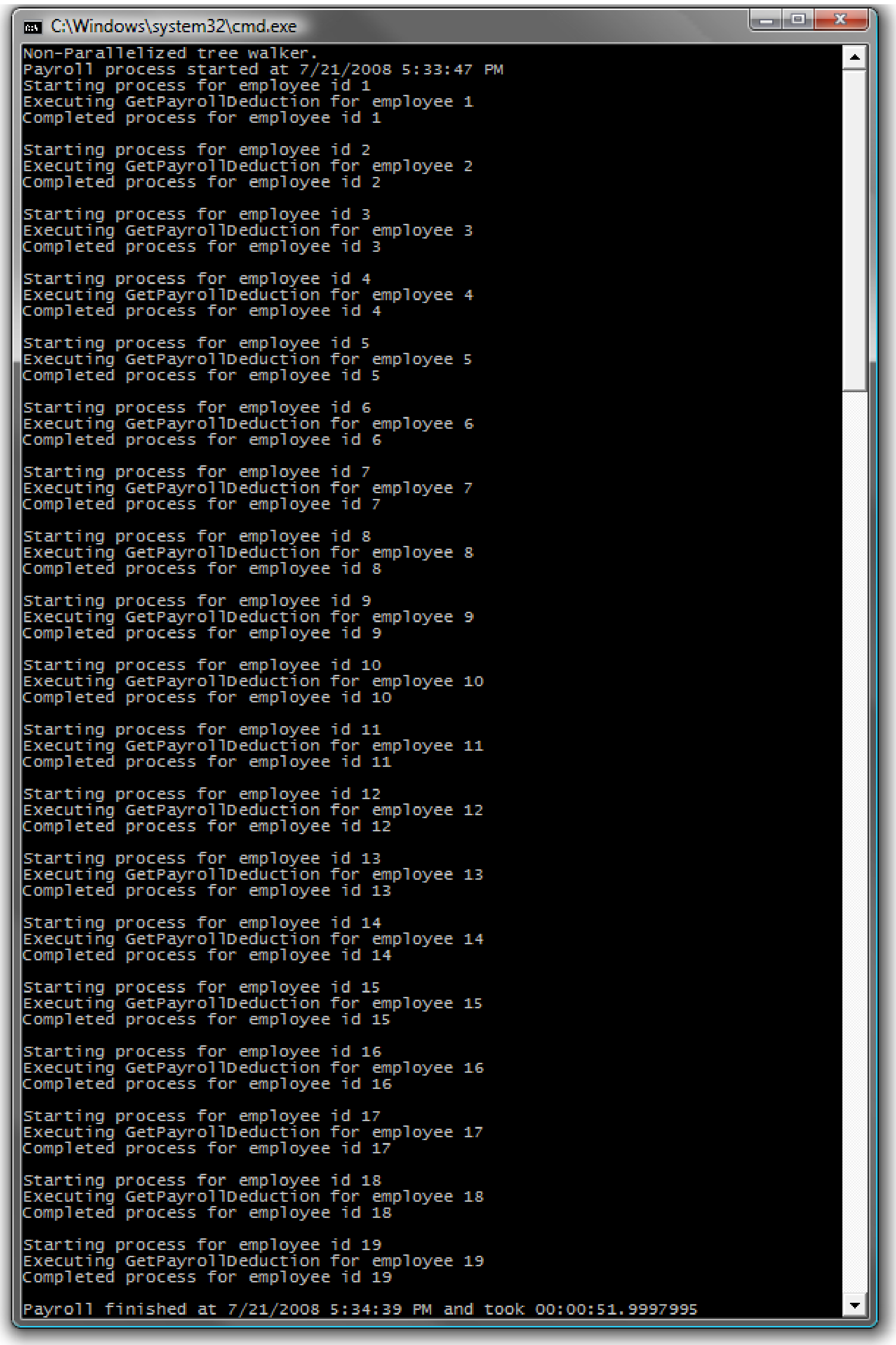
Visual Basic

```
Private Sub WalkTree(ByVal node As Tree(Of Employee))
    If node Is Nothing Then
        Return
    End If

    If node.Data IsNot Nothing Then
        Dim emp As Employee = node.Data
        Console.WriteLine("Starting process for employee id {0}",
emp.EmployeeID)
        Dim span As Decimal = PayrollServices.GetPayrollDeduction(emp)
        Console.WriteLine("Completed process for employee id {0}",
emp.EmployeeID)
        Console.WriteLine()
    End If

    WalkTree(node.Left)
    WalkTree(node.Right)
End Sub
```

21. Build and run the application.
22. 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):


C:\Windows\system32\cmd.exe

```
Non-Parallelized tree walker.
Payroll process started at 7/21/2008 5:33:47 PM
Starting process for employee id 1
Executing GetPayrollDeduction for employee 1
Completed process for employee id 1

Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Completed process for employee id 2

Starting process for employee id 3
Executing GetPayrollDeduction for employee 3
Completed process for employee id 3

Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Completed process for employee id 4

Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Completed process for employee id 5

Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 6

Starting process for employee id 7
Executing GetPayrollDeduction for employee 7
Completed process for employee id 7

Starting process for employee id 8
Executing GetPayrollDeduction for employee 8
Completed process for employee id 8

Starting process for employee id 9
Executing GetPayrollDeduction for employee 9
Completed process for employee id 9

Starting process for employee id 10
Executing GetPayrollDeduction for employee 10
Completed process for employee id 10

Starting process for employee id 11
Executing GetPayrollDeduction for employee 11
Completed process for employee id 11

Starting process for employee id 12
Executing GetPayrollDeduction for employee 12
Completed process for employee id 12

Starting process for employee id 13
Executing GetPayrollDeduction for employee 13
Completed process for employee id 13

Starting process for employee id 14
Executing GetPayrollDeduction for employee 14
Completed process for employee id 14

Starting process for employee id 15
Executing GetPayrollDeduction for employee 15
Completed process for employee id 15

Starting process for employee id 16
Executing GetPayrollDeduction for employee 16
Completed process for employee id 16

Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 17

Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Completed process for employee id 18

Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 19

Payroll finished at 7/21/2008 5:34:39 PM and took 00:00:51.9997995
```

Figure 5

Output from a non-parallel tree walker

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.

23. 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:

C#

```
private static void WalkTree(Tree<Employee> node)
{
    if (node == null)
        return;

    if (node.Data != null)
    {
        Employee emp = node.Data;
        Console.WriteLine("Starting process for employee id {0}",
            emp.EmployeeID);
        decimal span = PayrollServices.GetPayrollDeduction(emp);
        Console.WriteLine("Completed process for employee id {0}",
            emp.EmployeeID);
        Console.WriteLine();
    }

    Parallel.Invoke(delegate { WalkTree(node.Left); }, delegate {
WalkTree(node.Right); });
}
```

Visual Basic

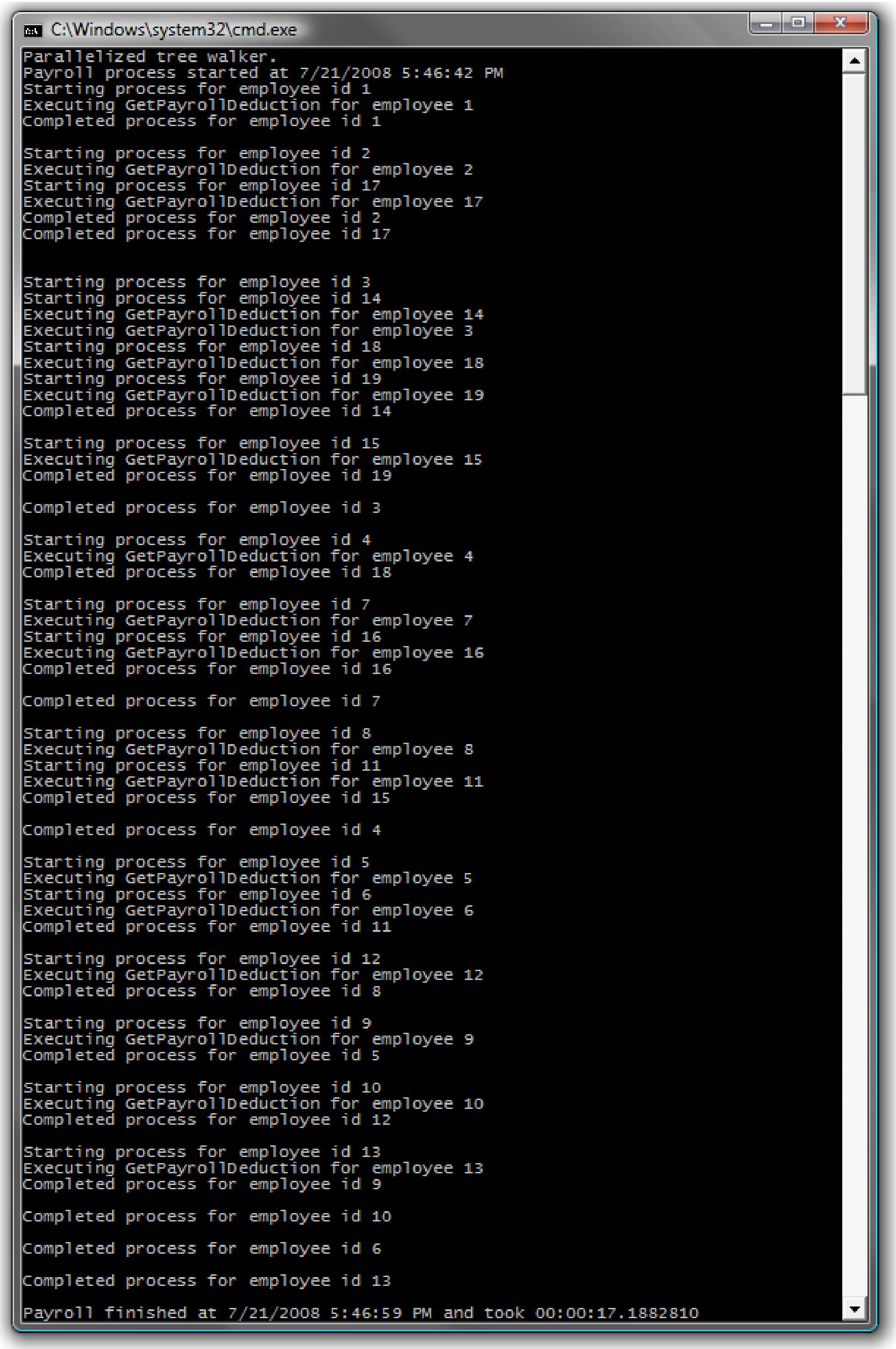
```
Private Sub WalkTree(ByVal node As Tree(Of Employee))
    If node Is Nothing Then
        Return
    End If

    If node.Data IsNot Nothing Then
        Dim emp As Employee = node.Data
        Console.WriteLine("Starting process for employee id {0}",
emp.EmployeeID)
        Dim span As Decimal = PayrollServices.GetPayrollDeduction(emp)
        Console.WriteLine("Completed process for employee id {0}",
emp.EmployeeID)
        Console.WriteLine()
    End If

    Parallel.Invoke(Sub() WalkTree(node.Left), Sub() WalkTree(node.Right))
End Sub
```

24. Build and run the application.

25. 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.



C:\Windows\system32\cmd.exe

```
Parallelized tree walker.
Payroll process started at 7/21/2008 5:46:42 PM
Starting process for employee id 1
Executing GetPayrollDeduction for employee 1
Completed process for employee id 1

Starting process for employee id 2
Executing GetPayrollDeduction for employee 2
Starting process for employee id 17
Executing GetPayrollDeduction for employee 17
Completed process for employee id 2
Completed process for employee id 17


Starting process for employee id 3
Starting process for employee id 14
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 3
Starting process for employee id 18
Executing GetPayrollDeduction for employee 18
Starting process for employee id 19
Executing GetPayrollDeduction for employee 19
Completed process for employee id 14

Starting process for employee id 15
Executing GetPayrollDeduction for employee 15
Completed process for employee id 19

Completed process for employee id 3

Starting process for employee id 4
Executing GetPayrollDeduction for employee 4
Completed process for employee id 18

Starting process for employee id 7
Executing GetPayrollDeduction for employee 7
Starting process for employee id 16
Executing GetPayrollDeduction for employee 16
Completed process for employee id 16

Completed process for employee id 7

Starting process for employee id 8
Executing GetPayrollDeduction for employee 8
Starting process for employee id 11
Executing GetPayrollDeduction for employee 11
Completed process for employee id 15

Completed process for employee id 4

Starting process for employee id 5
Executing GetPayrollDeduction for employee 5
Starting process for employee id 6
Executing GetPayrollDeduction for employee 6
Completed process for employee id 11

Starting process for employee id 12
Executing GetPayrollDeduction for employee 12
Completed process for employee id 8

Starting process for employee id 9
Executing GetPayrollDeduction for employee 9
Completed process for employee id 5

Starting process for employee id 10
Executing GetPayrollDeduction for employee 10
Completed process for employee id 12

Starting process for employee id 13
Executing GetPayrollDeduction for employee 13
Completed process for employee id 9

Completed process for employee id 10

Completed process for employee id 6

Completed process for employee id 13

Payroll finished at 7/21/2008 5:46:59 PM and took 00:00:17.1882810
```

Figure 6

Output from a parallel tree walker

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:

Exercise 2: Create and Run Parallelized Tasks

Exercise 2: Create and Run Parallelized Tasks

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.

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.

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.

   C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex2Task1_NativeParallelTasks();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex2Task1_NativeParallelTasks()
    ...
End Sub
```

4. Add the Ex2Task1_NativeParallelTasks method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task1_NativeParallelTasks CSharp*)

C#

```
private static void Ex2Task1_NativeParallelTasks()
{
    Task task1 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[0]); });
    Task task2 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[1]); });
    Task task3 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[2]); });
}
```

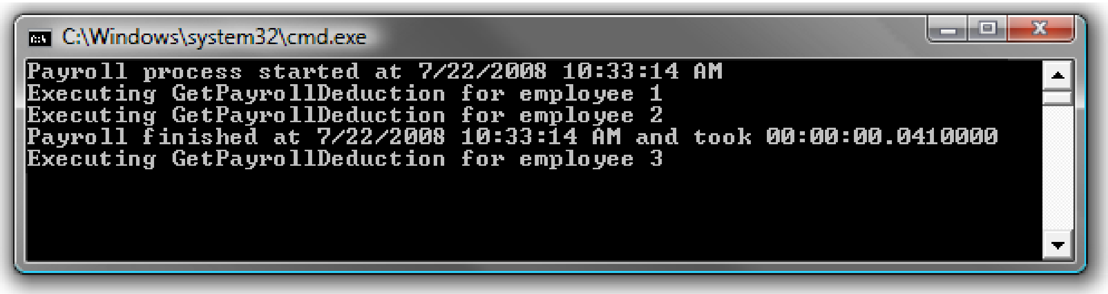
   (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task1_NativeParallelTasks VB*)

Visual Basic

```
Private Sub Ex2Task1_NativeParallelTasks()
    Dim task1 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(0)))
    Dim task2 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(1)))
    Dim task3 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(2)))
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**.



```
C:\Windows\system32\cmd.exe
Payroll process started at 7/22/2008 10:33:14 AM
Executing GetPayrollDeduction for employee 1
Executing GetPayrollDeduction for employee 2
Payroll finished at 7/22/2008 10:33:14 AM and took 00:00:00.0410000
Executing GetPayrollDeduction for employee 3
```

Figure 7

Output from running several Tasks in parallel

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

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.

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.

1. Replace the current method calls from **Main()**, with a call to **Ex2Task2_WaitHandling()**. This code will add wait handling to your example.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex2Task2_WaitHandling();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex2Task2_WaitHandling()
    ...
```

```
End Sub
```

2. Add the Ex2Task2_WaitHandling() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2_WaitHandling CSharp*)

   C#

```
private static void Ex2Task2_WaitHandling()
{
    Task task1 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[0]); });
    Task task2 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[1]); });
    Task task3 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[2]); });

    task1.Wait();
    task2.Wait();
    task3.Wait();
}
```

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2_WaitHandling VB*)

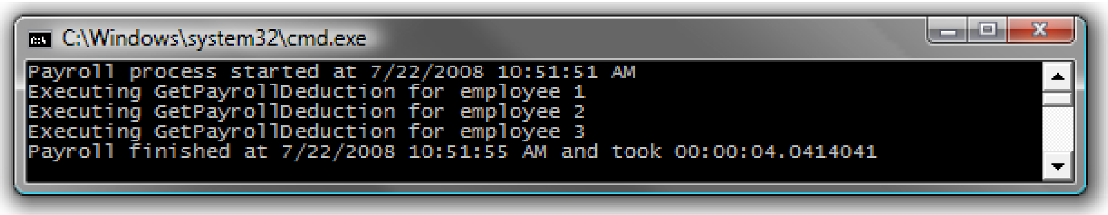
   C#

```
Private Sub Ex2Task2_WaitHandling()
    Dim task1 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(0)))
    Dim task2 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(1)))
    Dim task3 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(2)))

    task1.Wait()
    task2.Wait()
    task3.Wait()
End Sub
```

3. Build and run the application.

4. You should observe that this time all three Tasks completed before the final time was reported, indicating the end of the main thread.



```
Payroll process started at 7/22/2008 10:51:51 AM
Executing GetPayrollDeduction for employee 1
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 3
Payroll finished at 7/22/2008 10:51:55 AM and took 00:00:04.0414041
```

Figure 8

Output from running tasks in parallel with individual Wait() conditions

Note: The main thread waited for of 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.

5. 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:

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex2Task2_WaitHandlingWaitAll();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex2Task1_WaitHandlingWaitAll()
    ...
End Sub
```

6. Add the Ex2Tas2_WaitHandlingWaitAll() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2_WaitHandlingWaitAll CSharp*)

C#

```
private static void Ex2Task2_WaitHandlingWaitAll()
{
    Task task1 = Task.Factory.StartNew(delegate
        { PayrollServices.GetPayrollDeduction(employeeData[0]); });
    Task task2 = Task.Factory.StartNew (delegate
        { PayrollServices.GetPayrollDeduction(employeeData[1]); });
    Task task3 = Task.Factory.StartNew (delegate
        { PayrollServices.GetPayrollDeduction(employeeData[2]); });

    Task.WaitAll(task1, task2, task3);
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task2_WaitHandlingWaitAll VB*)

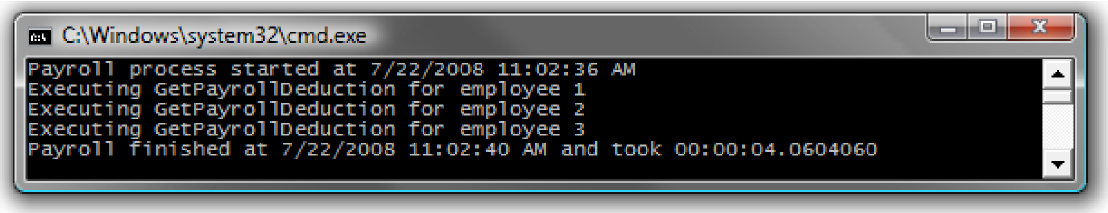
Visual Basic

```
Private Sub Ex2Task2_WaitHandlingWaitAll()
    Dim task1 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(0)))
    Dim task2 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(1)))
    Dim task3 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(2)))

    Task.WaitAll(task1, task2, task3)
End Sub
```

7. Build and run the application.

8. You should observe the main application waits until all individual Tasks are completed before continuing.



```
Payroll process started at 7/22/2008 11:02:36 AM
Executing GetPayrollDeduction for employee 1
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 3
Payroll finished at 7/22/2008 11:02:40 AM and took 00:00:04.0604060
```

Figure 9

Output from running tasks in parallel with the WaitAll() method

Task 3 – Using the IsCompleted Property

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.

1. Replace the current method calls from **Main()**, with a call to **Ex2Task3_TaskIsCompleted()**.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex2Task3_TaskIsCompleted();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex2Task3_TaskIsCompleted()
    ...
End Sub
```

2. Add the Ex2Task3_TaskIsCompleted() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task3_TaskIsCompleted CSharp*)

C#

```
private static void Ex2Task3_TaskIsCompleted()
{
    Task task1 = Task.Factory.StartNew(delegate
    { PayrollServices.GetPayrollDeduction(employeeData[0]); });

    while (!task1.IsCompleted)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Waiting on task 1");
    }

    Task task2 = Task.Factory.StartNew(delegate
    { PayrollServices.GetPayrollDeduction(employeeData[1]); });
    while (!task2.IsCompleted)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Waiting on task 2");
    }
    Task task3 = Task.Factory.StartNew(delegate
    { PayrollServices.GetPayrollDeduction(employeeData[2]); });
    while (!task3.IsCompleted)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Waiting on task 3");
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task3_TaskIsCompleted VB*)

Visual Basic

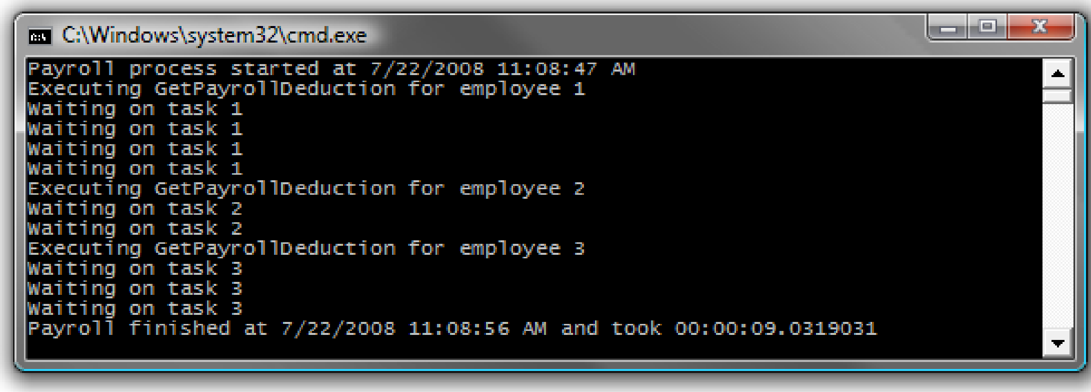
```
Private Sub Ex2Task3_TaskIsCompleted()
    Dim task1 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(0)))

    Do While Not task1.IsCompleted
        Thread.Sleep(1000)
        Console.WriteLine("Waiting on task 1")
    Loop

    Dim task2 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(1)))
    Do While Not task2.IsCompleted
        Thread.Sleep(1000)
        Console.WriteLine("Waiting on task 2")
    Loop

    Dim task3 As Task = Task.Factory.StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(2)))
    Do While Not task3.IsCompleted
        Thread.Sleep(1000)
        Console.WriteLine("Waiting on task 3")
    Loop
End Sub
```

3. Build and run the application.
4. You should observe that Tasks two and three do not start until the previous Task's **IsCompleted** property is true.



```
Payroll process started at 7/22/2008 11:08:47 AM
Executing GetPayrollDeduction for employee 1
Waiting on task 1
Waiting on task 1
Waiting on task 1
Waiting on task 1
Executing GetPayrollDeduction for employee 2
Waiting on task 2
Waiting on task 2
Executing GetPayrollDeduction for employee 3
Waiting on task 3
Waiting on task 3
Waiting on task 3
Payroll finished at 7/22/2008 11:08:56 AM and took 00:00:09.0319031
```

Figure 10
Output from running tasks in parallel and utilizing the IsCompleted property

Task 4 – Using the ContinueWith() Method

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.

The functionality passed in as arguments to the **ContinueWith()** method will be executed once the **Task** object's logic continues.

1. Replace the current method calls from **Main()**, with a call to **Ex2Task4_ContinueWith()**.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex2Task4_ContinueWith();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex2Task4_ContinueWith()
    ...
End Sub
```

2. Add the Ex2Task4_ContinueWith() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task4_ContinueWith CSharp*)

C#

```
private static void Ex2Task4_ContinueWith()
{
    Task task3 = Task.Factory.StartNew(delegate
            { PayrollServices.GetPayrollDeduction(employeeData[0]); })
        .ContinueWith(delegate
            { PayrollServices.GetPayrollDeduction(employeeData[1]); })
        .ContinueWith(delegate
            { PayrollServices.GetPayrollDeduction(employeeData[2]); });

    task3.Wait();
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex2 Ex2Task4_ContinueWith VB*)

Visual Basic

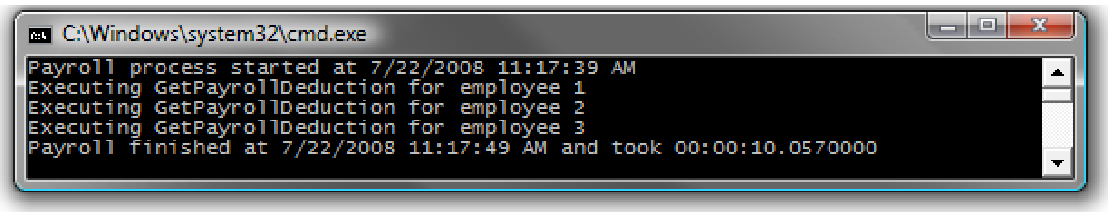
```
Private Sub Ex2Task4_ContinueWith()
    Dim task3 As Task = Task.Factory _
                        .StartNew(Sub()
PayrollServices.GetPayrollDeduction(employeeData(0))) _
                        .ContinueWith(Sub()
PayrollServices.GetPayrollDeduction(employeeData(1))) _
                        .ContinueWith(Sub()
PayrollServices.GetPayrollDeduction(employeeData(2)))

    task3.Wait()
End Sub
```

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.

3. Build and run the application.

4. You should observe that the tasks execute in order – employee 1 first, followed by employee 2, and finally employee 3.



```
Payroll process started at 7/22/2008 11:17:39 AM
Executing GetPayrollDeduction for employee 1
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 3
Payroll finished at 7/22/2008 11:17:49 AM and took 00:00:10.0570000
```

Figure 11

Output from running tasks in parallel and using ContinueWith to ensure their order and wait conditions.

Next Step:

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

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.

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()**.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex3Task1_TaskReturnValue();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
```

```
    ...
    ' Methods to call
    Ex3Task1_TaskReturnValue()
    ...
End Sub
```

4. Add the Ex3Task1_TaskReturnValue method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex3 Ex3Task1_TaskReturnValue CSharp*)

C#

```
private static void Ex3Task1_TaskReturnValue()
{
    Console.WriteLine("Calling parallel task with return value");
    var data = Task.Factory.StartNew(() =>
      PayrollServices.GetPayrollDeduction(employeeData[0]));
    Console.WriteLine("Parallel task returned with value of {0}",
        data.Result);
}
```

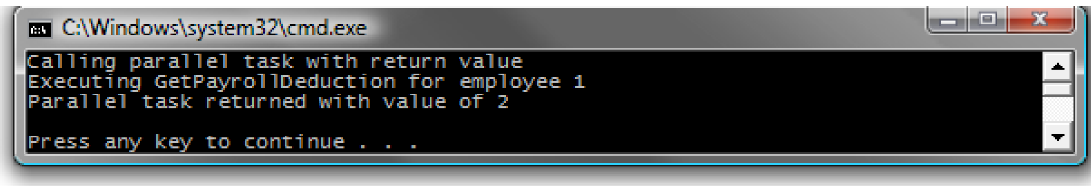
(Code Snippet – *Intro to Parallel Extensions Lab - Ex3 Ex3Task1_TaskReturnValue VB*)

Visual Basic

```
Private Sub Ex3Task1_TaskReturnValue()
    Console.WriteLine("Calling parallel task with return value")
    Dim data As Task(Of Decimal) = Task.Factory.StartNew(Function()
PayrollServices.GetPayrollDeduction(employeeData(0)))
    Console.WriteLine("Parallel task returned with value of {0}", data.Result)
End Sub
```

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.



```
Calling parallel task with return value
Executing GetPayrollDeduction for employee 1
Parallel task returned with value of 2

Press any key to continue . . .
```

Figure 12

Output from running a Task to capture a return value

Next Step:

Exercise 4: Parallelize LINQ Queries using PLINQ

Exercise 4: Parallelize LINQ Queries using PLINQ

Developers can optimize their LINQ queries to execute in a parallel environment by utilizing Parallelized LINQ (PLINQ).

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

In this task you will continue to use the same solution as the previous exercises.

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\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.

3. Replace the current method calls from **Main()**, with a call to **Ex4Task1_PLINQ()**.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex4Task1_PLINQ();
    ...
}
```

Visual Basic

```vb
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex4Task1_PLINQ()
    ...
End Sub
```

4. Add the Ex4Task1_PLINQ() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1_PLINQ CSharp*)

C#

```csharp
private static void Ex4Task1_PLINQ()
{
    var q = Enumerable.Select(
              Enumerable.OrderBy(
                Enumerable.Where(employeeData,
                x => x.EmployeeID % 2 == 0),
                x => x.EmployeeID),
              x => PayrollServices.GetEmployeeInfo(x))
              .ToList();

    foreach (var e in q)
    {
        Console.WriteLine(e);
    }
}
```

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1_PLINQ VB*)

Visual Basic

```vb
Private Sub Ex4Task1_PLINQ()
    Dim q = Enumerable.Select(
                Enumerable.OrderBy(
                    Enumerable.Where(
                        employeeData,
                        Function(x) x.EmployeeID Mod 2 = 0),
                        Function(x) x.EmployeeID),
                        Function(x) PayrollServices.GetEmployeeInfo(x)) _
            .ToList()

    For Each e In q
        Console.WriteLine(e)
    Next e
End Sub
```

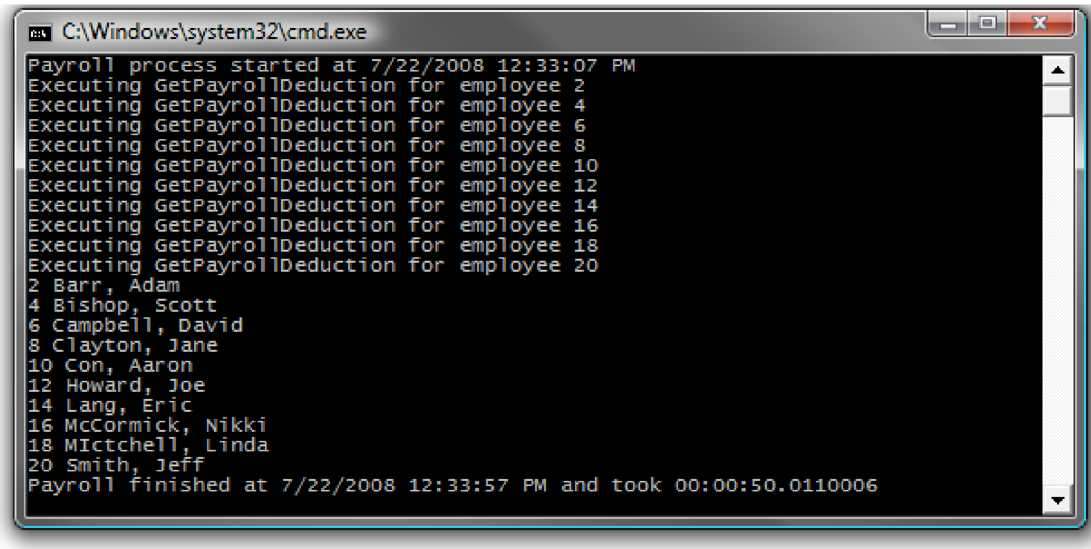
Note: The **Select()**, **OrderBy()**, and **Where()** methods are extension methods off of the **IEnumerable** generic class, however you are accessing them in a static manner here. You will try a more succinct usage later.

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.

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.

See Scott Wisniewski's article at http://msdn.microsoft.com/en-us/magazine/cc163378.aspx for more information.

5. Build and run the application.
6. 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):



```
Payroll process started at 7/22/2008 12:33:07 PM
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 6
Executing GetPayrollDeduction for employee 8
Executing GetPayrollDeduction for employee 10
Executing GetPayrollDeduction for employee 12
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 16
Executing GetPayrollDeduction for employee 18
Executing GetPayrollDeduction for employee 20
2 Barr, Adam
4 Bishop, Scott
6 Campbell, David
8 Clayton, Jane
10 Con, Aaron
12 Howard, Joe
14 Lang, Eric
16 McCormick, Nikki
18 MIctchell, Linda
20 Smith, Jeff
Payroll finished at 7/22/2008 12:33:57 PM and took 00:00:50.0110006
```

Figure 13
Output from a non-parallelized LINQ query

7. 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.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex4Task1_PLINQAsParallel();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex4Task1_PLINQAsParallel()
    ...
End Sub
```

8. Add the Ex4Task1_PLINQAsParallel() to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1_PLINQAsParallel CSharp*)

C#

```
private static void Ex4Task1_PLINQAsParallel()
{
    var q = ParallelEnumerable.Select(
            ParallelEnumerable.OrderBy(
              ParallelEnumerable.Where(employeeData.AsParallel(),
                x => x.EmployeeID % 2 == 0),
              x => x.EmployeeID),
            x => PayrollServices.GetEmployeeInfo(x))
          .ToList();

    foreach (var e in q)
    {
        Console.WriteLine(e);
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task1_PLINQAsParallel VB*)

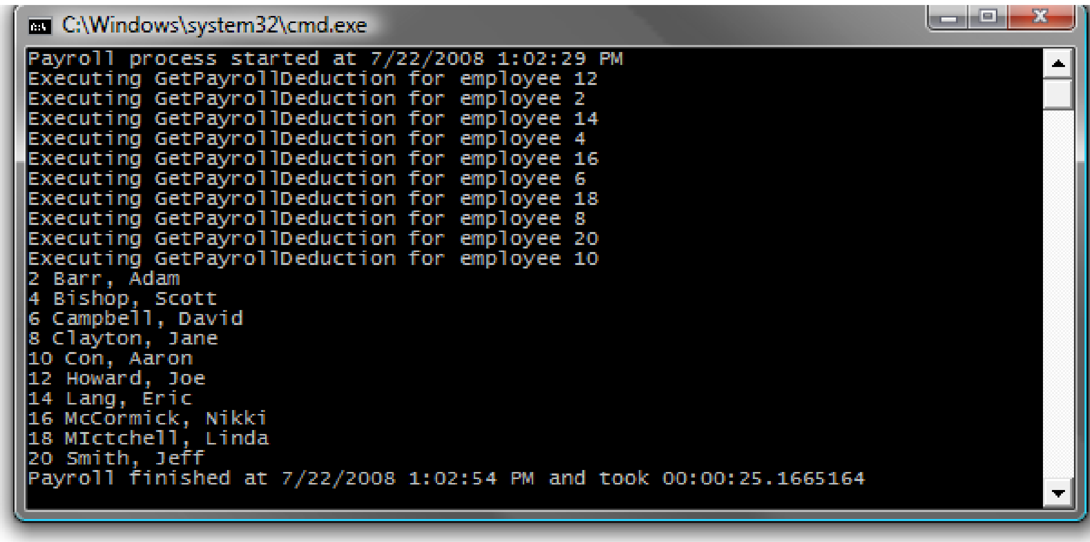
Visual Basic

```
Private Sub Ex4Task1_PLINQAsParallel()
    Dim q = ParallelEnumerable.Select(
                ParallelEnumerable.OrderBy(
                    ParallelEnumerable.Where(
                        employeeData.AsParallel(),
                        Function(x) x.EmployeeID Mod 2 = 0),
                        Function(x) x.EmployeeID),
                        Function(x) PayrollServices.GetEmployeeInfo(x)) _
            .ToList()

    For Each e In q
        Console.WriteLine(e)
    Next e
End Sub
```

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.

9. Build and run the application.
10. 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).



```
Payroll process started at 7/22/2008 1:02:29 PM
Executing GetPayrollDeduction for employee 12
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 16
Executing GetPayrollDeduction for employee 6
Executing GetPayrollDeduction for employee 18
Executing GetPayrollDeduction for employee 8
Executing GetPayrollDeduction for employee 20
Executing GetPayrollDeduction for employee 10
2 Barr, Adam
4 Bishop, Scott
6 Campbell, David
8 Clayton, Jane
10 Con, Aaron
12 Howard, Joe
14 Lang, Eric
16 McCormick, Nikki
18 MIctchell, Linda
20 Smith, Jeff
Payroll finished at 7/22/2008 1:02:54 PM and took 00:00:25.1665164
```

Figure 14

Output from a parallelized LINQ query

Note: Instead 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

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.

1. 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:

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex4Task2_Extensions();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex4Task2_Extensions()
    ...
End Sub
```

2. Add the Ex4Task2_Extensions() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

(Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2_Extensions CSharp*)

C#

```
private static void Ex4Task2_Extensions()
{
    var q = employeeData.
        Where(x => x.EmployeeID % 2 == 0).OrderBy(x => x.EmployeeID)
        .Select(x => PayrollServices.GetEmployeeInfo(x))
        .ToList();

    foreach (var e in q)
    {
        Console.WriteLine(e);
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2_Extensions VB*)

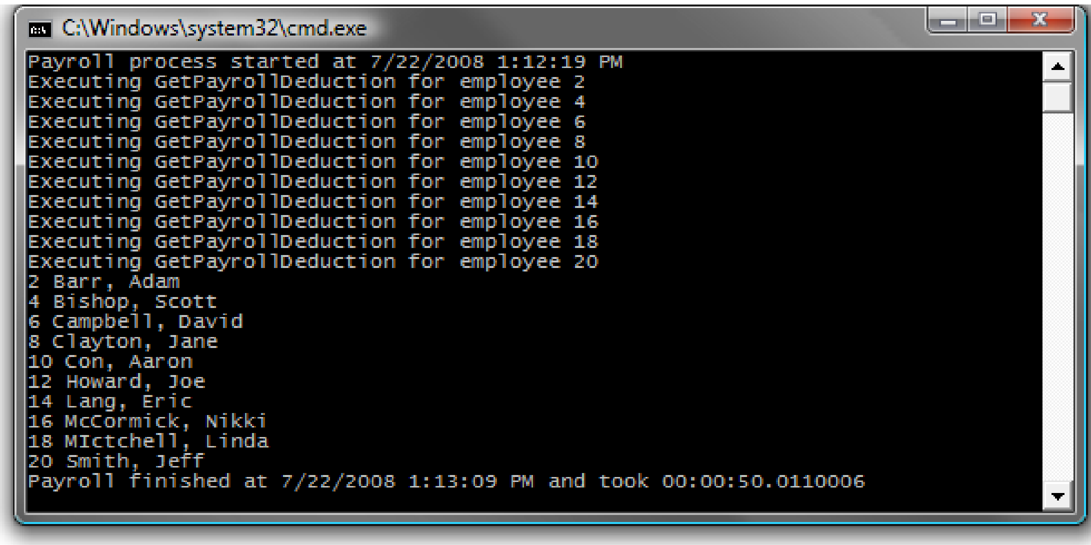
Visual Basic

```
Private Sub Ex4Task2_Extensions()
    Dim q = employeeData _
            .Where(Function(x) x.EmployeeID Mod 2 = 0) _
            .OrderBy(Function(x) x.EmployeeID) _
            .Select(Function(x) PayrollServices.GetEmployeeInfo(x)) _
            .ToList()

    For Each e In q
        Console.WriteLine(e)
    Next e
End Sub
```

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.

3. Build and run the application.
4. 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):



```
Payroll process started at 7/22/2008 1:12:19 PM
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 6
Executing GetPayrollDeduction for employee 8
Executing GetPayrollDeduction for employee 10
Executing GetPayrollDeduction for employee 12
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 16
Executing GetPayrollDeduction for employee 18
Executing GetPayrollDeduction for employee 20
2 Barr, Adam
4 Bishop, Scott
6 Campbell, David
8 Clayton, Jane
10 Con, Aaron
12 Howard, Joe
14 Lang, Eric
16 McCormick, Nikki
18 MIctchell, Linda
20 Smith, Jeff
Payroll finished at 7/22/2008 1:13:09 PM and took 00:00:50.0110006
```

Figure 15

Output from non-parallelized LINQ query with extension methods

5. To parallelize this LINQ query just replace the current method call from **Main()**, with the **AsParallel()** method.

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex4Task2_ConvertToParallelExtensions();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex4Task2_ConvertToParallelExtensions()
    ...
End Sub
```

6. Add the Ex4Task2_ConvertToParallelExtensions() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2_ConvertToParallelExtensions CSharp*)

C#

```
private static void Ex4Task2_ConvertToParallelExtensions()
{
    var q = employeeData.AsParallel()
        .Where(x => x.EmployeeID % 2 == 0).OrderBy(x => x.EmployeeID)
        .Select(x => PayrollServices.GetEmployeeInfo(x))
        .ToList();

    foreach (var e in q)
    {
        Console.WriteLine(e);
    }
}
```

(Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task2_ConvertToParallelExtensions VB*)

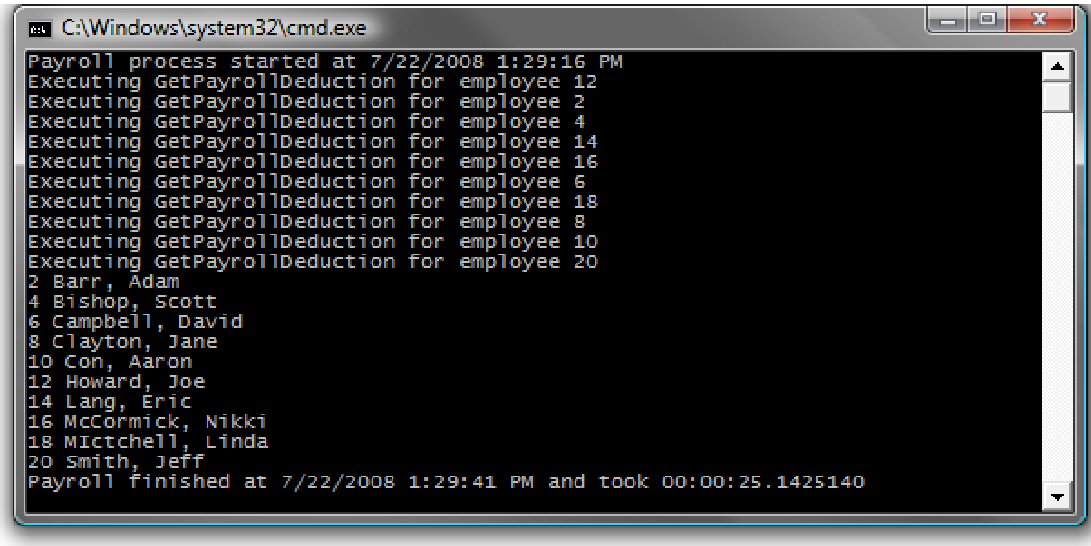
Visual Basic

```
Private Sub Ex4Task2_ConvertToParallelExtensions()
    Dim q = employeeData.AsParallel() _
                .Where(Function(x) x.EmployeeID Mod 2 = 0) _
                .OrderBy(Function(x) x.EmployeeID) _
                .Select(Function(x) PayrollServices.GetEmployeeInfo(x)) _
                .ToList()

    For Each e In q
        Console.WriteLine(e)
    Next e
End Sub
```

7. Build and run the application.

8. 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.



```
Payroll process started at 7/22/2008 1:29:16 PM
Executing GetPayrollDeduction for employee 12
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 16
Executing GetPayrollDeduction for employee 6
Executing GetPayrollDeduction for employee 18
Executing GetPayrollDeduction for employee 8
Executing GetPayrollDeduction for employee 10
Executing GetPayrollDeduction for employee 20
2 Barr, Adam
4 Bishop, Scott
6 Campbell, David
8 Clayton, Jane
10 Con, Aaron
12 Howard, Joe
14 Lang, Eric
16 McCormick, Nikki
18 MIctchell, Linda
20 Smith, Jeff
Payroll finished at 7/22/2008 1:29:41 PM and took 00:00:25.1425140
```

Figure 16

Output from parallelized LINQ query with extension methods

Task 3 – Using AsParallel() With Query Comprehension Syntax

In this task you will use the Parallel Extensions library and the **AsParallel()** method to create parallelized LINQ queries using the query comprehension syntax.

1. Replace the LINQ query in the **Main()** method with the following query comprehension syntax:

C#

```
static void Main(string[] args)
{
    ...
    // Methods to call
    Ex4Task3_PLINQComprehensionSyntax();
    ...
}
```

Visual Basic

```
Sub Main(ByVal args() As String)
    ...
    ' Methods to call
    Ex4Task3_PLINQComprehensionSyntax()
    ...
End Sub
```

2. Add the Ex4Task3_PLINQComprehensionSyntax() method to **Program.cs** (C#) or **Module1.vb** (Visual Basic):

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task3_PLINQComprehensionSyntax CSharp*)

C#

```
private static void Ex4Task3_PLINQComprehensionSyntax()
{
    var q = from e in employeeData.AsParallel()
            where e.EmployeeID % 2 == 0
            orderby e.EmployeeID
            select PayrollServices.GetEmployeeInfo(e);

    foreach (var e in q)
    {
        Console.WriteLine(e);
    }
}
```

   (Code Snippet – *Intro to Parallel Extensions Lab - Ex4 Ex4Task3_PLINQComprehensionSyntax VB*)

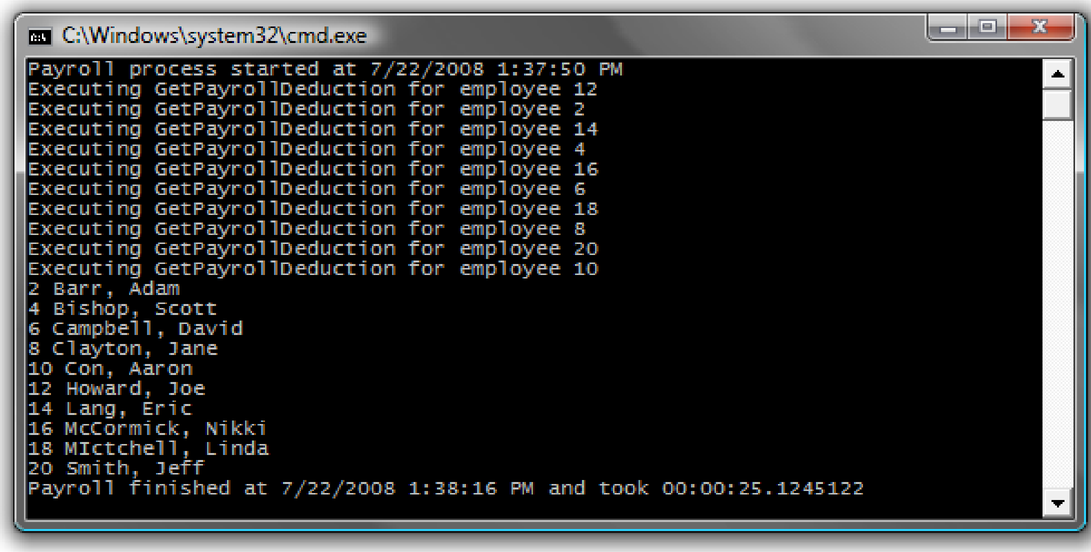
Visual Basic

```
Private Sub Ex4Task3_PLINQComprehensionSyntax()
    Dim q = From e In employeeData.AsParallel()
            Where e.EmployeeID Mod 2 = 0
            Order By e.EmployeeID
            Select PayrollServices.GetEmployeeInfo(e)

    For Each e In q
        Console.WriteLine(e)
    Next e
End Sub
```

3. Build and run the application.

4. 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.



```
Payroll process started at 7/22/2008 1:37:50 PM
Executing GetPayrollDeduction for employee 12
Executing GetPayrollDeduction for employee 2
Executing GetPayrollDeduction for employee 14
Executing GetPayrollDeduction for employee 4
Executing GetPayrollDeduction for employee 16
Executing GetPayrollDeduction for employee 6
Executing GetPayrollDeduction for employee 18
Executing GetPayrollDeduction for employee 8
Executing GetPayrollDeduction for employee 20
Executing GetPayrollDeduction for employee 10
2 Barr, Adam
4 Bishop, Scott
6 Campbell, David
8 Clayton, Jane
10 Con, Aaron
12 Howard, Joe
14 Lang, Eric
16 McCormick, Nikki
18 MIctchell, Linda
20 Smith, Jeff
Payroll finished at 7/22/2008 1:38:16 PM and took 00:00:25.1245122
```

Figure 17

Output from parallelized LINQ query using the query comprehension syntax

Next Step:

Summary

Summary

In this lab you've 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

The Parallel Computing Developer Center: http://msdn.microsoft.com/en-us/concurrency/default.aspx