

Lecture 5b

Querying In-Memory Data by Using PLINQ and TPL



OBJECTIVES

In this lecture you will learn:

- Define Language Integrated Query (LINQ) queries to examine the contents of enumerable collections.
- Use LINQ extension methods and query operators with Lambda expressions.
- Use TPL and PLINQ extension methods and query operators using parallel programming.

References:

Chapter 20, John Sharp, “*Microsoft Visual C# 2012 Step By Step*”, Microsoft Press, 2012 ISBN-13: 978-0-7356-6801-0



- 1 Introduction
- 2 Language Integrated Query (LINQ)
- 3 Using LINQ in a C# Application
- 4 Selecting Data
- 5 Filtering Data
- 6 Ordering, Grouping, and Aggregating Data
- 7 Task Parallel Library (TPL)
- 8 Parallel LINQ (PLINQ)

Problems to Solve

1 Introduction

One important aspect of the C# language is the support that C# provides for querying data.

You have seen that you can define structures and classes for modeling data and that you can use collections and arrays for temporarily storing data in memory.

However, how do you **perform common tasks** such as searching for items in a collection that match a specific set of criteria?

1 Introduction

For **example**, if you have a collection of *Student* objects, *how do you find all students that are born in Sofia* , or *how can you* find out which town has the most customers for your services?

You can **write your own code** to iterate through a collection and *examine the fields* in each object, but these types of *tasks occur so often* that the designers of C# decided to include **features to minimize the amount of code you need to write**. In this lecture, you will **learn how to use these advanced C# language features** to query and manipulate data.



2 Language Integrated Query (LINQ)

All but the most trivial of **applications need to process data**.

Historically, **most applications provided their own logic** for performing these operations.

However, this strategy can **lead to the code in an application becoming very tightly coupled** to the structure of the data that it processes.

If the data structures change, you might need to make a significant number of changes to the code that handles the data.

2 Language Integrated Query (LINQ)

The designers of the Microsoft .NET Framework thought long and hard about these issues and decided to make the life of an application developer easier by providing features that *abstract the mechanism that an application uses to query* data from application code itself.

These features are called **Language Integrated Query**, or **LINQ**

2 Language Integrated Query (LINQ)

Consider the way in which **relational database management systems**, such as Microsoft SQL Server, **separate the language used to query a database** from the **internal format of the data in the database**.

Developers accessing a SQL Server database issue Structured Query Language (SQL) statements to the database management system. **SQL provides a high-level description of the data** that the developer wants to retrieve but **does not indicate exactly** how the database management system should retrieve this data.



2 Language Integrated Query (LINQ)

The **details** about how a SQL query executes **are controlled by the database management system** itself.

Consequently, an application that invokes SQL statements **does not care how the database management system physically stores** or retrieves data.

The format used by the database management system can change (for example, if a new version is released) **without** the application developer **needing to modify** the SQL statements used by the application..

2 Language Integrated Query (LINQ)

LINQ provides syntax and semantics very reminiscent of SQL, and with many of the same advantages. You can change the underlying structure of the data being queried without needing to change the code that actually performs the queries. You should be aware that although **LINQ looks similar to SQL**, it is **far more flexible and can handle a wider variety of logical data structures**. For example, LINQ can handle data organized hierarchically, such as that found in an XML document. However, this lecture concentrates on using LINQ in a relational manner.

3 Using LINQ in a C# Application

Perhaps the easiest way to explain how to use the C# features that support LINQ is to work through some **simple examples** based on the following sets of **customer** and **address** information→.

3 Using LINQ in a C# Application

Customer Information

<i>CustomerID</i>	FirstName	LastName	CompanyName
1	Orlando	Gee	A Bike Store
2	Keith	Harris	Bike World
3	Donna	Carreras	A Bike Store
4	Janet	Gates	Fitness Hotel
5	Lucy	Harrington	Grand Industries
6	David	Liu	Bike World
7	Donald	Blanton	Grand Industries
8	Jackie	Blackwell	Fitness Hotel
9	Elsa	Leavitt	Grand Industries
10	Eric	Lang	Distant Inn

3 Using LINQ in a C# Application

Address Information

<i>CompanyName</i>	<i>City</i>	<i>Country</i>
A Bike Store	New York	United States
Bike World	Chicago	United States
Fitness Hotel	Ottawa	Canada
Grand Industries	London	United Kingdom
Distant Inn	Tetbury	United Kingdom

3 Using LINQ in a C# Application

LINQ requires the data to be stored in a data structure that is ***IEnumerable***. (an array, a *Hashtable*, a *List*, a *Queue* as we have studied in the last lecture are *IEnumerable*)

However, to keep things straightforward, the examples in this lecture assume that the ***customer*** and ***address*** information is held in the ***customers*** and ***addresses arrays*** with code examples shown on the following slide



3 Using LINQ in a C# Application

```
var customers = new[] {  
    new { CustomerID = 1, FirstName = "Orlando", LastName = "Gee",    CompanyName = "A Bike Store" },  
    new { CustomerID = 2, FirstName = "Keith",    LastName = "Harris", CompanyName = "Bike World" },  
    new { CustomerID = 3, FirstName = "Donna",    LastName = "Carreras", CompanyName = "A Bike Store" },  
    new { CustomerID = 4, FirstName = "Janet",    LastName = "Gates",    CompanyName = "Fitness Hotel" },  
    new { CustomerID = 5, FirstName = "Lucy", LastName = "Harrington", CompanyName = "Grand Industries" },  
    new { CustomerID = 6, FirstName = "David",    LastName = "Liu",      CompanyName = "Bike World" },  
    new { CustomerID = 7, FirstName = "Donald", LastName = "Blanton",    CompanyName = "Grand Industries" },  
    new { CustomerID = 8, FirstName = "Jackie",    LastName = "Blackwell", CompanyName = "Fitness Hotel" },  
    new { CustomerID = 9, FirstName = "Elsa",      LastName = "Leavitt",    CompanyName = "Grand Industries" },  
    new { CustomerID = 10, FirstName = "Eric",    LastName = "Lang",      CompanyName = "Distant Inn" }  
}
```

3 Using LINQ in a C# Application

```
var addresses = new[] {  
    new {CompanyName = "A Bike Store", City = "New York", Country = "United States"},  
    new {CompanyName = "Bike World", City = "Chicago", Country = "United States"},  
    new {CompanyName = "Fitness Hotel", City = "Ottawa", Country = "Canada"},  
    new {CompanyName = "Grand Industries", City = "London", Country = "UK"},  
    new {CompanyName = "Distant Inn", City = "Tetbury", Country = "UK"}  
};
```


4 Selecting Data

Suppose you want to **display a list comprising the *first name* of each customer** in the *customers* array.

You can achieve this task with the following code:

```
IEnumerable<string> customerFirstNames =  
    customers.Select(cust => cust.FirstName);  
  
foreach (string name in customerFirstNames)  
{  
    Console.WriteLine(name);  
}
```



4 Selecting Data

Although this block of **code** is quite short, **it does a lot, and it requires a degree of explanation**, starting with the use of the ***Select** method of the customers array*.

4 Selecting Data

The **Select** method enables you to retrieve specific data from the array- in this case, **just the value** in the **FirstName** field of each item in the array.

How does it work?

The parameter to the **Select** method is actually another **method** (**anonymous**) that **takes a row from the customers array and returns the selected data from that row**. You could define your own custom method to perform this task, but the simplest mechanism is to **use a lambda expression** to define an **anonymous method**, as shown in the preceding example.



4 Selecting Data

There are **three important things** that you need to understand at this point:

- ❑ The **type** of ***cust*** is the **type** of the parameter passed in to the method. You can think of ***cust*** as ***an alias for the type of each row in the customers array***. The compiler deduces this from the fact that you are calling the ***Select*** method on the ***customers*** array. You can use any legal C# identifier in place of ***cust***.

4 Selecting Data

- ❑ The **Select** method *does not actually retrieve the data at this time; it simply returns an enumerable object* that will **fetch** the data identified by the **Select** method *when* you iterate over it **later**
- ❑ The **Select** method *is not actually a method of the Array type. It is an extension method* of the **Enumerable** class. The **Enumerable** class is located in the **System.Linq** namespace and provides a substantial set of **static** methods for querying objects that implement the generic **IEnumerable<T>** interface

4 Selecting Data

The preceding example uses the **Select** method of the **customers** array to **generate** an **IEnumerable<string>** object named **customerFirstNames**.

*It is of type **IEnumerable<string>** because the **Select** method returns an enumerable collection of **customer** first names, which are **strings**.*

4 Selecting Data

The ***foreach*** statement *iterates* through this collection of strings, printing out the **first name** of each ***customer*** in the following sequence:

Orlando

Keith

Donna

Janet

Lucy

David

Donald

Jackie

Elsa

Eric

4 Selecting Data

Another way to achieve the same result is:

```
var customerFirstNames =  
    from cust in customers  
    select cust.FirstName;  
  
foreach (string name in customerFirstNames)  
{  
    Console.WriteLine(name);  
}
```


4 Selecting Data

Here the LINQ query uses a **from** clause, which specifies a **range variable** (**cust**) and the data source to query **customers** .

How do you fetch the **first and last name** of each customer?

```
var customerFirstNames =  
    from cust in customers  
    select new{cust.FirstName,cust.LastName};  
foreach (var name in customerFirstNames)  
{  
    Console.WriteLine(name);  
}
```

4 Selecting Data

The important point to understand from the preceding slide is that the **Select** method returns an enumerable collection based on a single type. If you want the **enumerator to return multiple items** of data, such as the first and last name of each customer, you have at least two options:

- ❑ You can **concatenate** the first and last names together into a single string in the **Select** method, like above or as:

```
IEnumerable<string> customerFullName =  
    customers.Select(cust => cust.FirstName + " " +  
                        cust.LastName) ;
```

- ❑ You can define a **new anonymous type** that wraps the first and last names and use the **Select** method to **construct instances** of this type

4 Selecting Data

The second option may be written also like this:

```
var customerName =  
customers.Select(cust => new { FirstName =  
    cust.FirstName, LastName = cust.LastName } );
```

Notice the use of the **var** keyword here to *define the type of the enumerable collection*.

*The type of objects in the collection is **anonymous**, so you cannot specify a specific type for the objects in the collection.*

4 Selecting Data

After C# 7 the new Tuples syntax

```
IEnumerable<(string, string)> customerNames =  
customers.Select(  
    cust =>(cust.FirstName, cust.LastName) );  
foreach((string, string) name in customerNames)  
{  
    Console.WriteLine("{0}, {1}",  
        name.Item1, name.Item2);  
}
```

4 Selecting Data

Notice: Use of the **var** keyword here to *define the type of the enumerable collection for shortness*

Simplify as follows:

```
var customerNames =  
    customers.Select(  
        cust => (cust.FirstName, cust.LastName) );  
foreach (var name in customerNames)  
{  
    Console.WriteLine("{0}, {1}",  
        name.Item1, name.Item2);  
}
```

4 Selecting Data

Notice: Use **var** to enable names to your elements (so they are not "**Item1**", "**Item2**" etc)

```
var customerNames =  
    customers.Select(  
        cust => (Fname: cust.FirstName,  
                Lname: cust.LastName) );  
foreach (var name in customerNames)  
{  
    Console.WriteLine("{0}, {1}",  
                      name.Fname, name.Lname) ;  
}
```

4 Selecting Data

Notice: Enable names to your elements in the **foreach** command

```
var customerNames =  
    customers.Select(  
        cust => (Fname: cust.FirstName,  
                 Lname: cust.LastName) );  
  
foreach (  
    (string Fname, string Lname) name in customerNames)  
{  
    Console.WriteLine("{0}, {1}",  
                      name.Fname, name.Lname) ;  
}
```

5 Filtering Data

The ***Select*** method enables you to specify, or project, the fields that you want to include in the enumerable collection. However, you might also want to ***restrict the rows*** that the enumerable collection contains.

For example, suppose you want to list the names of all companies in the ***addresses*** array that are located in the ***United States*** only.

5 Filtering Data

To do this, you can use the *Where method*, as follows:

```
IEnumerable<string> usCompanies =  
    addresses.Where(addr =>  
        String.Equals(addr.Country, "United States"))  
    .Select(usComp => usComp.CompanyName) ;  
  
foreach (string name in usCompanies)  
{  
    Console.WriteLine(name) ;  
}
```

5 Filtering Data

OR , the same, as follows:

```
var usCompanies =  
    from addr in addresses  
    where String.Equals(addr.Country, "United States")  
    select addr.CompanyName;  
  
foreach (string name in usCompanies)  
{  
    Console.WriteLine(name) ;  
}
```

5 Filtering Data

Syntactically, the **Where** method is similar to **Select**. It expects a parameter that defines a method that filters the data according to whatever criteria you specify.

This example makes use of another **lambda expression**. The type **addr** is an alias for a row in the **addresses** array, and the **lambda expression** returns all rows where the **Country** field matches the **string** “**United States**”.

The **Where** method returns an enumerable collection of rows containing every field from the original collection. The **Select** method is then applied to these rows to project only the **CompanyName** field from this enumerable collection to return another enumerable collection of **string** objects.

5 Filtering Data

The type **usComp** is *an alias for the type of each row in the enumerable collection returned by the **Where** method.*) The type of the result of this complete expression is therefore **IEnumerable<string>**.

*It is important to understand this sequence of operations- the **Where** method is **applied first** to filter the rows, followed by the **Select** method to specify the fields. The **foreach** statement that iterates through this collection displays the following companies:*

A Bike Store

Bike World

6 Ordering, Grouping, and Aggregating Data

If you are **familiar with SQL**, you are aware that **SQL enables you to perform a wide variety of relational operations** besides simple projection and filtering.

For example, you can specify that you **want data to be returned** in a **specific order**, you can **group the rows** returned according to one or more key fields, and you can **calculate summary values** based on the rows in each group.

LINQ provides the same functionality

6 Ordering, Grouping, and Aggregating Data

To retrieve data in a particular order, you can use the **OrderBy** method. Like the **Select** and **Where** methods, **OrderBy** expects a **method** as its argument.

This method **identifies the expressions** that you want to use to sort the data. For example, you can display the names of each company in the *addresses* array in ascending order, like this:

```
var companyNames =  
addresses.OrderBy(addr => addr.CompanyName).Select(comp  
=> comp.CompanyName) ;  
foreach (string name in companyNames)  
{  
    Console.WriteLine(name) ;  
}
```

6 Ordering, Grouping, and Aggregating Data

or:

```
var companyNames =  
    from addr in addresses  
    orderby addr.CompanyName  
    select addr.CompanyName) ;  
foreach (string name in companyNames)  
{  
    Console.WriteLine(name) ;  
}
```

6 Ordering, Grouping, and Aggregating Data

This block of code displays the companies in the addresses table in alphabetical order:

A Bike Store

Bike World

Distant Inn

Fitness Hotel

Grand Industries

6 Ordering, Grouping, and Aggregating Data

If you want to enumerate the data **in descending order**, you can use the *OrderByDescending* method instead. If you want to order by more than one key value, you can use the *ThenBy* or *ThenByDescending* method after *OrderBy* or *OrderByDescending*
OR

```
var companyNames =  
    from addr in addresses  
    orderby addr.CompanyName descending  
    select addr.CompanyName;  
  
// ascending is by default set for orderby  
foreach (string name in companyNames)  
{  
    Console.WriteLine(name);  
}
```



6 Ordering, Grouping, and Aggregating Data

To **group data according** to common values in one or more fields, you can use the *GroupBy* method. The next example shows how to group the *companies* in the *addresses* array by *country*:

```
var companiesGroupedByCountry =  
addresses.GroupBy(addr => addr.Country);
```

```
// or
```

OR

```
var companiesGroupedByCountry =  
    from addr in addresses  
    orderby addr.Country, addr.CompanyName descending  
    group addr by addr.Country into countryGroup  
    select countryGroup;  
// ascending is by default set for orderby
```



6 Ordering, Grouping, and Aggregating Data

```
foreach (var companiesPerCountry in
            companiesGroupedByCountry)
{
    Console.WriteLine("Country: {0}\t{1} companies",
                      companiesPerCountry.Key,
                      companiesPerCountry.Count());
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine("\t{0}", companies.CompanyName);
    }
}
```

6 Ordering, Grouping, and Aggregating Data

The output generated by the example code looks like this:

Country: United States 2 companies

A Bike Store

Bike World

Country: Canada 1 companies

Fitness Hotel

Country: United Kingdom 2 companies

Grand Industries

Distant Inn

6 Ordering, Grouping, and Aggregating Data

By now you should **recognize the pattern**. The *GroupBy* method expects a method that specifies the fields to group the data by. There are some subtle **differences** between the *GroupBy* method and the other methods that you have seen so far, though. The main point of interest is that you **don't need to use** the *Select* method to project the fields to the result.

The **enumerable set** returned by *GroupBy* contains all the fields in the original source collection, but the rows are ordered into a set of enumerable collections based on the field identified by the method specified by *GroupBy*.

In other words, the result of the *GroupBy* method is an **enumerable** set of groups, each of which is an **enumerable** set of rows

6 Ordering, Grouping, and Aggregating Data

In the **example** just shown, the enumerable set **companiesGroupedByCountry** is a set of countries. The items in this set are themselves enumerable collections containing the companies for each **country** in turn. The code that displays the companies in each country uses a **foreach** *loop to iterate* through the **companiesGroupedByCountry** *set to yield and display each country in turn and then uses a nested foreach loop to iterate through the set of companies in each country.*

Notice in the outer **foreach** *loop that you can access the value that you are grouping by using the Key field of each item, and you can also calculate summary data for each group by using methods such as Count, Max, Min, and many others*

6 Ordering, Grouping, and Aggregating Data

Group by **CountryName** and sort by the **length** of the **CountryName** and **CountryNames** with **identical** length are **sorted in ascending order**

```

1  // Using query expression syntax.
2  var companiesGroupedByCountryLength1 = from company in addresses
3      group company.CompanyName.ToUpper() by company.Country into grp
4      orderby grp.Key.Length, grp.Key
5      select new { Country = grp.Key, Companies = grp };
6
7  // Using method-based query syntax.
8  var companiesGroupedByCountryLength2 = addresses.
9      GroupBy(company => company.Country, company => company.CompanyName.ToUpper()).
10     Select(group => new { Country = group.Key, Companies = group }).
11     OrderBy(o => o.Country.Length).ThenBy(o => o.Country) ;
12
13 Console.WriteLine("Companies by Country length");
14 foreach (var obj in companiesGroupedByCountryLength1)
15 {
16     Console.WriteLine("Country {0} with Total Companies {1}:",
17                       obj.Country, obj.Companies.Count());
18     foreach (string word in obj.Companies)
19         Console.WriteLine("Company {0}:", word);
20 }

```

6 Ordering, Grouping, and Aggregating Data

Group by **CountryName** and sort by the **length** of the **CountryName** and **CountryNames** with **identical** length are **sorted in ascending order**

```
var addresses = new[] {
    new {CompanyName = "A Bike Store",    City = "New York", Country = "United States"},
    new {CompanyName = "Bike World",      City = "Chicago",  Country = "United States"} ,
    new {CompanyName = "Fitness Hotel",   City = "Ottawa",   Country = "Canada"},
    new {CompanyName = "Grand Industries", City = "London",   Country = "UK"},
    new {CompanyName = "Distant Inn",      City = "Tetbury",  Country = "UK"},
    new {CompanyName = "Distant Bike",     City = "Sofia",    Country = "BG"}
};
```

```
Companies by Country length
Country BG with Total Companies 1:
Company DISTANT BIKE:
Country UK with Total Companies 2:
Company GRAND INDUSTRIES:
Company DISTANT INN:
Country Canada with Total Companies 1:
Company FITNESS HOTEL:
Country United States with Total Companies 2:
Company A BIKE STORE:
Company BIKE WORLD:
Press any key to continue . . .
```


6 Ordering, Grouping, and Aggregating Data

You can use many of the summary methods such as **Count**, **Max**, and **Min** directly over the results of the **Select** method. If you want to know how many companies there are in the **addresses** array, you can use a block of code such as this:

```
int numberOfCompanies =  
addresses.Select(addr =>  
    addr.CompanyName).Count();  
Console.WriteLine("Number of companies: {0}",  
    numberOfCompanies);
```



6 Ordering, Grouping, and Aggregating Data

Similarly:

```
var usCompanies =  
    from addr in addresses  
    select addr.CompanyName;  
  
int numberOfCompanies = usCompanies.Count();  
Console.WriteLine("Number of companies: {0}",  
    numberOfCompanies);
```

6 Ordering, Grouping, and Aggregating Data

Notice that the **result of these methods is a single scalar value** rather than an enumerable collection. The output from this block of code looks like this:

```
Number of companies: 5
```

6 Ordering, Grouping, and Aggregating Data

These summary methods do not distinguish between rows in the underlying set that contain duplicate values in the fields you are projecting.

What this means is that, strictly speaking, the preceding example shows you **only how many rows in the *addresses* array contain a value in the *CompanyName* field.**

6 Ordering, Grouping, and Aggregating Data

In fact, **there are only three different countries** in the *addresses* array; *it just so happens that* **United States** and **United Kingdom** both occur twice. You can **eliminate duplicates** from the calculation by using the **Distinct** method, like this:

```
int numberOfCountries =  
    addresses.Select(addr =>  
        addr.Country).Distinct().Count();
```

//or

```
int numberOfCompanies =  
usCompanies.Distinct().Count();
```

6 Ordering, Grouping, and Aggregating Data

The `Console.WriteLine` *statement* will *now output* the expected result:

```
Number of countries: 3
```

6 Generate and processing a sequence

`Enumerable.Range()` generates a sequence of integral numbers within a specified range

```
public static IEnumerable<int> Range(  
    int start,  
    int count  
)
```

```
1    // Generate a sequence of three integers starting at 4,  
2    // and then select their squares.  
3    IEnumerable<int> squares = Enumerable.Range(4, 3).Select(x => x * x);  
4    foreach (int num in squares)  
5    {  
6        Console.WriteLine(num);  
7    }  
    /*  
    This code produces the following output:  
  
16  
25  
36  
  
    */
```

7 Task Parallel Library (TPL)

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. Data parallelism with imperative syntax is **supported by several overloads** of the `For` and `ForEach` methods in the `System.Threading.Tasks.Parallel` class.

In data parallel operations, **the source collection is partitioned so that multiple threads can operate on different segments concurrently.** [TPL](#) supports data parallelism through the `System.Threading.Tasks.Parallel` class. This class provides **method-based parallel implementations** of **for** and **foreach** loops

You **write the loop logic** for a `Parallel.For` or `Parallel.ForEach` loop much as you would write a sequential loop. You do not have to create threads or queue work item

7 Task Parallel Library (TPL)

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

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

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

Note: The operations executed in the lambda expression or lambda statement of a **Parallel.ForEach** or **Parallel.For** loop must be **independent** on each **sourceCollection item**, otherwise synchronization is required. **Shared data** must be **lock**-ed

```
List<int> numbers = new List<int>();
Parallel.ForEach(numbers, n =>
{
    // Do Processing of the current element n of numbers here
    // n is the current item
});
```



7 Task Parallel Library (TPL)

```
public static ParallelLoopResult For(  
    int fromInclusive,  
    int toExclusive,  
    Action<int> body  
)
```

Parameters

fromInclusive - The start index, inclusive (Type: **Int32**)

toExclusive - The end index, exclusive. (Type: **Int32**)

body - The Lambda Expression (delegate) that is invoked once per iteration. (Type: **Action<Int32>**)

Return Value

A structure that contains information about which portion of the loop completed. (Type: **ParallelLoopResult**)

7 Task Parallel Library (TPL)

```
public class Example
{
    public static void Main()
    {
        ParallelLoopResult result = Parallel.For(0, 100, ctr => { Random rnd = new Random(ctr * 100000);
                                                                    Byte[] bytes = new Byte[100];
                                                                    rnd.NextBytes(bytes);
                                                                    int sum = 0;
                                                                    foreach(var byt in bytes)
                                                                        sum += byt;
                                                                    Console.WriteLine("Iteration {0,2}: {1:N0}", ctr, sum);
                                                                    });
        Console.WriteLine("Result: {0}", result.IsCompleted ? "Completed Normally" :
                                                                    String.Format("Completed to {0}", result.LowestBreakIteration));
    }
}
```

// The following is a portion of the output displayed by the example:

```
//      Iteration  0: 12,509
//      Iteration 50: 12,823
//      Iteration 51: 11,275
//      Iteration 52: 12,531
//      Iteration  1: 13,007
//      Iteration 53: 13,799
//      Iteration  4: 12,945
//      Iteration  2: 13,246
//      Iteration 54: 13,008
//      ...
//      ...
//      Iteration 92: 12,275
//      Iteration 93: 13,282
//      Iteration 94: 12,745
//      Iteration 95: 11,957
//      Iteration 96: 12,455
//      Result: Completed Normally
```



7 Task Parallel Library (TPL)

You can use the most basic overload of the **For** method when **you do not need to cancel or break** out of the iterations or **maintain any thread-local state**.

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



7 Task Parallel Library (TPL)

```
1  static void MultiplyMatricesParallel(double[,] matA, double[,] matB, double[,] result)
2      {
3          int matACols = matA.GetLength(1);
4          int matBCols = matB.GetLength(1);
5          int matARows = matA.GetLength(0);
6
7          // A basic matrix multiplication.
8          // Parallelize the outer loop to partition the source array by rows.
9          Parallel.For(0, matARows, i =>
10             { // i is the current iteration
11                 for (int j = 0; j < matBCols; j++)
12                     {
13                         // Use a temporary to improve parallel performance.
14                         double temp = 0;
15                         for (int k = 0; k < matACols; k++)
16                             {
17                                 temp += matA[i, k] * matB[k, j];
18                             }
19                         result[i, j] = temp;
20                     }
21             }); // Parallel.For
22 }
```

7 Task Parallel Library (TPL)

The **third parameter** of this overload of **For** is a **Lambda expression**. It may take **zero, one or sixteen type parameters**, **always returns void**.

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

Synchronous calls to shared resources, like the **Console** or the **File System**, will **significantly degrade the performance** of a **parallel loop**.

When measuring performance, **try to avoid calls** such as **Console.WriteLine** within the loop

7 Task Parallel Library (TPL)

The **Parallel.ForEach** method includes several overloaded versions that **allow the item indexes to be identified within the iterations**. All you need to do is use an **Action delegate** (initialized by a **lambda expression**) that includes **three parameters**. The **first** will be set to **the value to be processed** during each iteration. The **second** is the **ParallelLoopState object** that allows you to **terminate the loop** if desired with methods **Stop()** and **Break()**. The **third parameter** is the one we are interested in. It is set to a 64-bit integer, or *long*, containing the **index of the item** being processed.

To see how this works, consider the sample code below. Here the **Lambda expression** (**Action delegate**) **for** the loop has **three parameters** named, "**value**", "**pls**" and "**index**".

- ✓ **value** will be set to the **value for processing**
- ✓ **loopState** is the **loop state object** and
- ✓ **index** will be **set to the index of the item for each iteration**.

The output from the method shows that the indexes are correctly matched to the results



7 Task Parallel Library (TPL)

```
var values = Enumerable.Range(0, 16).Select(v => (int)Math.Pow(2, v));
```

```
Parallel.ForEach(values, (value, loopState, index) =>
{
    Console.WriteLine("{0}:\t{1}", index, value);
});
```

/* OUTPUT

Note: `Parallel.ForEach()` cannot guarantee that its iterations complete in any given order.

```
0:      1
3:      8
1:      2
2:      4
4:     16
9:    512
10:   1024
11:   2048
12:  4096
13:  8192
14: 16384
6:     64
7:    128
8:    256
5:     32
15: 32768
```

*/



7 Task Parallel Library (TPL)

```
int n = ...
Parallel.For(0, n, (i, loopState) =>
{
    // ...
    if (/* stopping condition is true */)
    {
        loopState.Break();
        return;
    }
});
```

In a `Parallel.For` or `Parallel.ForEach` loop, you cannot use the same `break` statement that is used in a sequential loop because those language constructs are valid for loops, and a **parallel "loop" is actually a method**, not a loop. Instead, you use either the **Stop** or **Break methods**. In this context, "**Break**" means **complete all iterations on all threads that are prior to the current iteration** on the current thread, and then **exit the loop**. "**Stop**" means to **stop all iterations as soon as convenient**.

7 Task Parallel Library (TPL)

```
var n = ...
var loopResult = Parallel.For(0, n, (i, loopState) =>
{
    if (/* stopping condition is true */)
    {
        loopState.Stop();
        return;
    }
    result[i] = DoWork(i);
});

if (!loopResult.IsCompleted &&
    !loopResult.LowestBreakIteration.HasValue)
{
    Console.WriteLine("Loop was stopped");
}
```

7 Task Parallel Library (TPL)

```
ParallelLoopResult result = Parallel.  
    For(0, 1000, (int i, ParallelLoopState loopState) =>  
{  
    if (i == 500)  
    {  
        Console.WriteLine("Breaking loop");  
        loopState.Break();  
  
    }  
    return;  
});
```

When breaking the parallel loop, the result variable has an *IsCompleted* value of *false* and a *LowestBreakIteration* of 500. When you use the *Stop* method, the *LowestBreakIteration* is *null*.

7 Task Parallel Library (TPL)

Note: While **order-preserving data-parallel operations** can be implemented using `Parallel.ForEach`, it usually **requires a considerable amount of work** with the exception of a few trivial cases **as the following**.

```
public static double [] PairwiseMultiply( double[] v1, double[] v2)
{
    var length = Math.Min(v1.Length, v2.Length);
    double[] result = new double[length];
    Parallel.ForEach(v1, (element, loopstate, elementIndex) =>
        result[elementIndex] = element * v2[elementIndex]);

    return result;
}
```

Note that in this sample, the **input and output were fixed-size arrays**, which made **writing the results to the output collection** rather simple. However, the **disadvantages to this approach** become immediately obvious, if the original collection were an `IEnumerable` rather than an array. **Use the Parallel Loop pattern** when you need to perform the same **independent** operation for each element of a collection or for a **fixed number** of iterations. The **steps of a loop are independent** if they **don't write to memory locations** or **files that are read by other steps**.



7 Task Parallel Library (TPL)

The following example shows how to write a **ForEach** method that **uses thread-local variables**. When a **ForEach** loop executes, it **divides its source collection into multiple partitions**. Each partition will get its own copy of the "thread-local" variable.

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

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

        Console.WriteLine("The total from Parallel.ForEach is {0:N0}", total);
    }
}

// The example displays the following output:
//      The total from Parallel.ForEach is 499,999,500,000
```



7 Task Parallel Library (TPL)

The same result with `Parallel.For`.

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

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

        Console.WriteLine("The total from Parallel.For is {0:N0}", total);
    }
}

// The example displays the following output:
// The total from Parallel.ForEach is 499,999,500,000
```



7 Task Parallel Library (TPL)

```
// The number of parallel iterations to perform.
const int N = 1000000;
static void Main()
{
    // The result of all thread-local computations.
    double result = 0.0;
    object lockObject = new object();
    Parallel.For<double>(0, N, new ParallelOptions { MaxDegreeOfParallelism = 2 },
        // Initialize the local states
        () => 0.0,
        // Accumulate the thread-local computations in the loop body
        (i, loop, localState) =>
        {
            return localState + Compute(i);
        },
        // Combine all local states
        (localPartialSum) =>
        {
            // Use lock to enforce serial access to shared result
            lock (lockObject)
            {
                result += localPartialSum;
            }
        }
    );

    // Print the actual and expected results.
    Console.WriteLine("Actual result: {0}. Expected 1000000.", result);
}
// Simulates a lengthy operation.
private static double Compute(int n)
{
    for (int i = 0; i < 1000; i++) ;
    return 1;
}
```



7 Task Parallel Library (TPL)

```

class Program
{
    static void Main(string[] args)
    {
        double[] sequence = Enumerable.Range(0, 1000000).Select<int, double>(x => (double)x / 2.0).ToArray<double>();
        Console.WriteLine("Number of elements to process: {0}, ", sequence.Length);
        object lockObject = new object();
        double sum = 0.0d;
        Parallel.ForEach(
            // The input intervals
            sequence,
            // The local initial partial result
            () => 0.0,
            // The loop body for each interval
            (range, loopState, localPartialSum) =>
            {
                return localPartialSum + range * 2.0;
            },
            // The final step of each local context
            (localPartialSum) =>
            {
                // Use lock to enforce serial access to shared result
                lock (lockObject)
                {
                    sum += localPartialSum;
                }
            });
        Console.WriteLine("{0}, Expected{1}", sum, ((1000000 / 2.0) * 999999));
    }
}

```


8 Parallel LINQ (PLINQ)

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



8 Parallel LINQ (PLINQ)

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

8 Parallel LINQ (PLINQ)

One of the best features of PLINQ is that it's **easy to convert LINQ queries to PLINQ**. You can simply add the **AsParallel** clause. Here is a simple LINQ query that returns a selection of books. **The results are generated sequentially:**

```
from book in books
  where book.Publisher == "Lucerne Publishing"
  orderby book.Title
  select book;
```

8 Parallel LINQ (PLINQ)

Now, here's the same query updated for PLINQ. Note that the only **addition to the code is the call** to the **AsParallel** method of the books collection. This minor change, however, completely alters how the query is performed. When you iterate over the results, **the query is performed with parallel tasks**.

```
from book in books.AsParallel()  
  where book.Publisher=="Lucerne Publishing"  
  orderby book.Title  
  select book;
```



8 Parallel LINQ (PLINQ)

Assuming that an operation can be parallelized, the more computationally expensive it is, the greater the opportunity for speedup. For example, if a function takes one millisecond to execute, a sequential query over 1000 elements will take one second to perform that operation, whereas a parallel query on a computer with four cores might take only 250 milliseconds. This yields a speedup of 750 milliseconds. If the function required one second to execute for each element, then the speedup would be 750 seconds. If the delegate is very expensive, then PLINQ might offer significant speedup with only a few items in the source collection. Conversely, **small source collections with trivial delegates** are generally **not good candidates for PLINQ**

```
var fibonacciNumbers = numberList.AsParallel().Select(n => ComputeFibonacci(n));
```

```
var queryA = from num in numberList.AsParallel()  
             select ExpensiveFunction(num); //good for PLINQ
```

```
var queryB = from num in numberList.AsParallel()  
             where num % 2 > 0  
             select num; //not as good for PLINQ
```



8 Parallel LINQ (PLINQ) Example

`Enumerable.Range()` generates a sequence of integral numbers within a specified range

```
public static IEnumerable<int> Range(  
    int start,  
    int count  
)
```

```
1 var source = Enumerable.Range(1, 10000);  
2  
3 // Opt in to PLINQ with AsParallel.  
4 var evenNums = from num in source.AsParallel()  
5                 where num % 2 == 0  
6                 select num;  
7 Console.WriteLine("{0} even numbers out of {1} total",  
8                 evenNums.Count(), source.Count());  
9 // The example displays the following output:  
10 //      5000 even numbers out of 10000 total
```

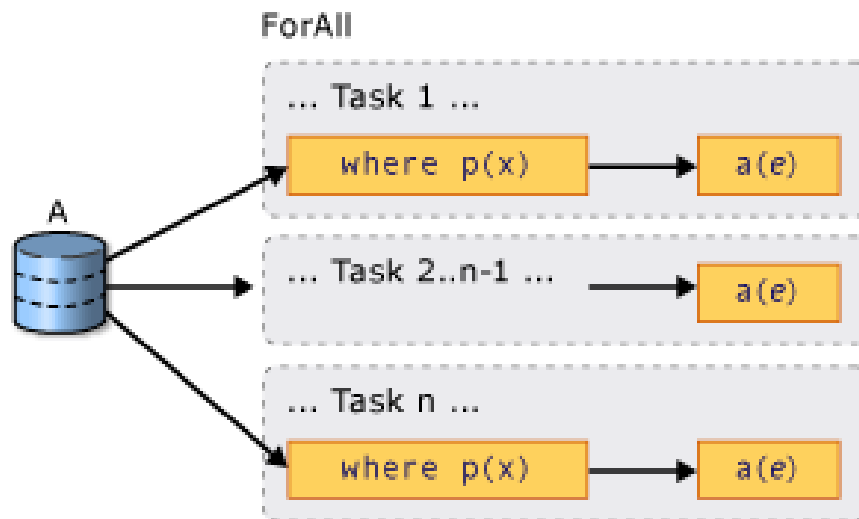
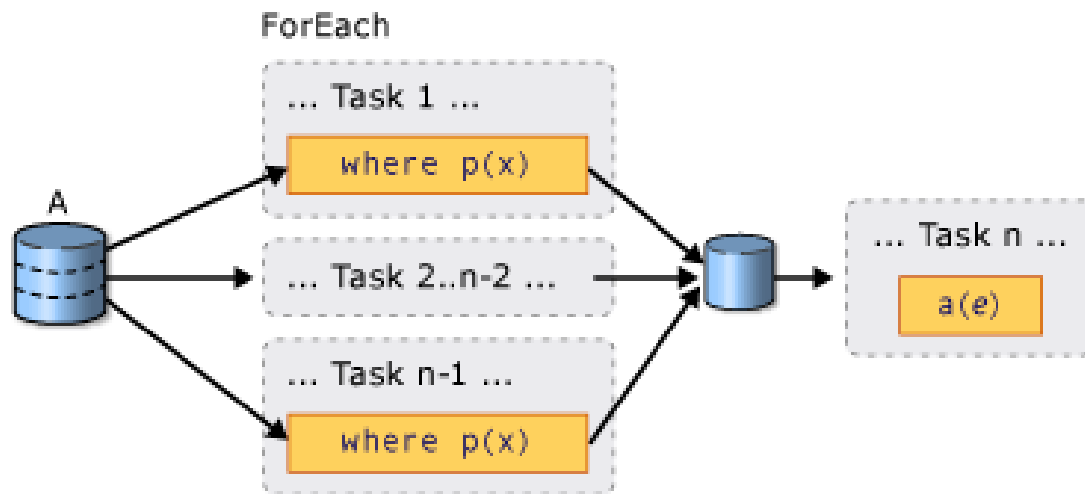
8 Parallel LINQ (PLINQ)

In sequential LINQ queries, execution is deferred until the query is enumerated either in a `foreach` loop or by invoking a method such as `ToList<TSource>` , `ToArray<TSource>` , or `ToDictionary`. In PLINQ, you can also use `foreach` to execute the query and iterate through the results.

However, `foreach` itself does not run in parallel, and therefore, it requires that the output from all parallel tasks be merged back into the thread on which the loop is running. In PLINQ, you can **use** `foreach` when you must preserve the final ordering of the query results, and also whenever you are processing the results in a serial manner, for example when you are calling `Console.WriteLine` for each element.

For **faster query execution when order preservation is not required** and when the processing of the results can itself be parallelized, use the **ForAll<TSource>** method to execute a PLINQ query.

8 Parallel LINQ (PLINQ)



8 Parallel LINQ (PLINQ)

```

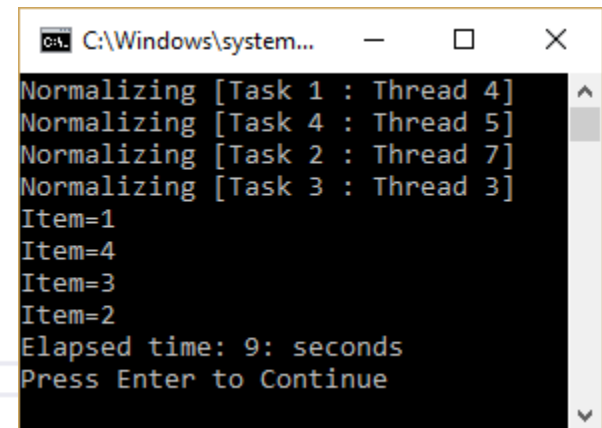
class Program
{
    static bool Normalize()
    {
        Console.WriteLine(
            "Normalizing [Task {0} : Thread {1}]",
            Task.CurrentId,
            Thread.CurrentThread.ManagedThreadId
        );
        Thread.SpinWait(int.MaxValue);
        return true;
    }

    static void Main(string[] args)
    {
        Stopwatch sw = new Stopwatch();
        var intArray = new[] { 1, 2, 3, 4 };
        //var result = intArray.Where((index) =>
        //    Normalize());
        var result =
            intArray.AsParallel().Where((index) => Normalize());

        sw.Start();
        foreach (int item in result)
        {
            Console.WriteLine("Item={0}", item);
        }
        sw.Stop();

        Console.WriteLine("Elapsed time: {0}: seconds", sw.ElapsedMilliseconds / 1000);
        Console.WriteLine("Press Enter to Continue");
        Console.ReadLine();
    }
}

```



```

C:\Windows\system...
Normalizing [Task 1 : Thread 4]
Normalizing [Task 4 : Thread 5]
Normalizing [Task 2 : Thread 7]
Normalizing [Task 3 : Thread 3]
Item=1
Item=4
Item=3
Item=2
Elapsed time: 9: seconds
Press Enter to Continue

```

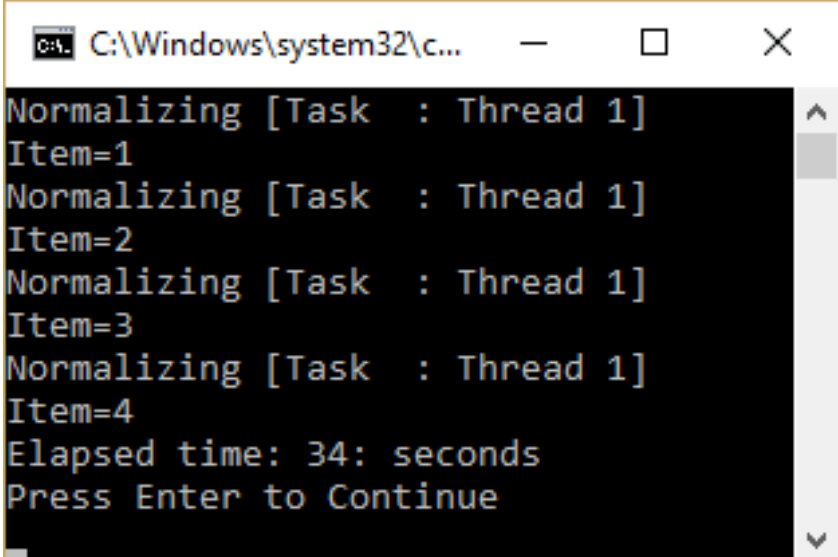
8 Parallel LINQ (PLINQ)

```
class Program
{
    static bool Normalize()
    {
        Console.WriteLine(
            "Normalizing [Task {0} : Thread {1}]",
            Task.CurrentId,
            Thread.CurrentThread.ManagedThreadId
        );
        Thread.SpinWait(int.MaxValue);
        return true;
    }

    static void Main(string[] args)
    {
        Stopwatch sw = new Stopwatch();
        var intArray = new[] { 1, 2, 3, 4 };
        var result = intArray.Where((index) =>
            Normalize());
        //var result =
        //intArray.AsParallel().Where((index) => Normalize());

        sw.Start();
        foreach (int item in result)
        {
            Console.WriteLine("Item={0}", item);
        }
        sw.Stop();

        Console.WriteLine("Elapsed time: {0}: seconds", sw.ElapsedMilliseconds / 1000);
        Console.WriteLine("Press Enter to Continue");
        Console.ReadLine();
    }
}
```



```
C:\Windows\system32\c...
Normalizing [Task : Thread 1]
Item=1
Normalizing [Task : Thread 1]
Item=2
Normalizing [Task : Thread 1]
Item=3
Normalizing [Task : Thread 1]
Item=4
Elapsed time: 34: seconds
Press Enter to Continue
```

8 Parallel LINQ (PLINQ)

Often, we need to perform some **filtering** or **grouping**, then **perform an action using the results** of our filter. Using a **standard foreach** statement to perform our action would look as follows:

```
var collection = Enumerable.Range(1, 10000);
var filteredItems = collection.AsParallel().Where( i => i.SomePredicate() );
// Now perform an action
foreach (var item in filteredItems)
{
    // These will now run serially
    item.DoSomething();
}
```

We could easily use a **Parallel.ForEach** instead, which **adds parallelism to the actions**. **Note: Parallel.ForEach** is not a **PLINQ** command rather belongs to the [Task Parallel Library](#)

```
// Now perform an action once the filter completes
Parallel.ForEach(filteredItems, item =>
{
    // These will now run in parallel
    item.DoSomething();
});
```



8 Parallel LINQ (PLINQ)

This is a noticeable improvement, since **both our filtering and our actions run parallelized**. However, there is still **a large bottleneck** in place here. Here, we're **parallelizing the filter**, then **collecting all of the results, blocking until the filter completes**. Once the filtering of every element is completed, we then **repartition the results of the filter**, reschedule into multiple threads, and **perform the action on each element**. By **moving this into two separate statements**, we potentially **double our parallelization overhead**, since we're forcing the work to be partitioned and scheduled twice **as many times**

8 Parallel LINQ (PLINQ)

LINQ's **Parallel.ForEach** method is **useful for parallelizing the same operation over a collection of values**. It would appear natural to adhere to the same model to process the results of a **PLINQ** query.

PLINQ returns a **ParallelQuery<TSource>** type, which **represents multiple streams of data**. However, **Parallel.ForEach** expects **a single stream of data**, which is then parsed into multiple streams.

For this reason, the **Parallel.ForEach** method **must recognize and convert multistream input to a single stream**. There is a **performance cost** for this conversion.

The **solution** is the **ParallelQuery<TSource>.ForAll** method.

The **ForAll** method directly accepts multiple streams, so it avoids the overhead of the **Parallel.ForEach** method

8 Parallel LINQ (PLINQ)

We use a **AsParallel()** **ForAll()** instead, which **adds parallelism to the actions**. The **ForAll** extension method should only be used to **process the results of a parallel query, as returned by a PLINQ expression**

```
var collection = Enumerable.Range(1, 10000);  
var filteredItems = collection  
    .AsParallel()  
    .Where( i => i.SomePredicate() )  
    .ForAll( i => i.DoSomething() );
```



8 Parallel LINQ (PLINQ)

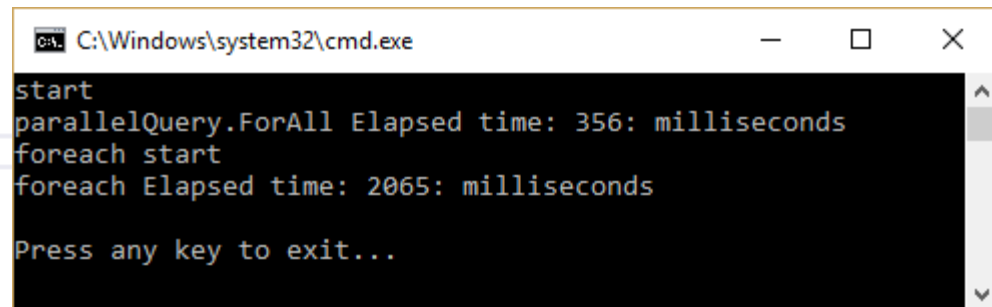
```
public static void Main()
{
    var source = Enumerable.Range(100, 100000);
    Stopwatch sw = new Stopwatch();
    // Result sequence might be out of order.
    var parallelQuery = from num in source.AsParallel()
                        where Compute(num)
                        select num;

    // Process result sequence in parallel
    Console.WriteLine("start");
    sw.Start();
    parallelQuery.ForAll((e) => DoSomething(e));
    sw.Stop();
    Console.WriteLine("parallelQuery.ForAll Elapsed time: {0}: milliseconds", sw.ElapsedMilliseconds);

    // Use foreach to merge results first.
    Console.WriteLine("foreach start ");
    sw.Reset();
    sw.Start();
    foreach (var n in parallelQuery)
    {
        DoSomething(n);
    }
    sw.Stop();
    Console.WriteLine("foreach Elapsed time: {0}: milliseconds", sw.ElapsedMilliseconds);
    Console.WriteLine("\nPress any key to exit...");
    Console.ReadLine();
}

static void DoSomething(int i) { //extensive computing task
    for (int ji = 0; ji < 1000000; ji++) ; }

static bool Compute(int i) { //extensive computing task
    for (int ji = 0; ji < 10000; ji++) ;
    return i % 10 == 0?true: false; }
```



```
C:\Windows\system32\cmd.exe
start
parallelQuery.ForAll Elapsed time: 356: milliseconds
foreach start
foreach Elapsed time: 2065: milliseconds

Press any key to exit...
```

8 Parallel LINQ (PLINQ)

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

namespace PLINQexample
{
    class Program
    {
        public static void Main()
        {
            var source = Enumerable.Range(100, 100);

            // Result sequence might be out of order.
            var parallelQuery = from num in source.AsParallel()
                               where num % 10 == 0
                               select num;

            // Process result sequence in parallel
            Console.WriteLine("start");
            parallelQuery.ForAll((e) => DoSomething(e));

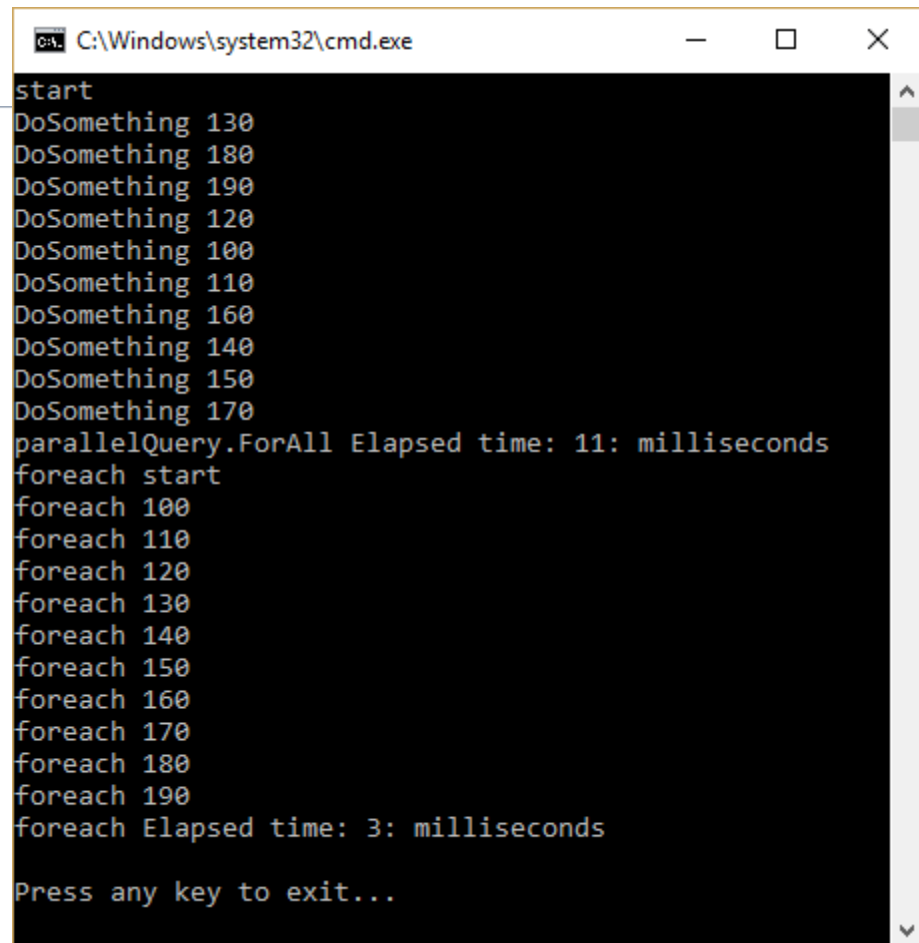
            // Or use foreach to merge results first.
            foreach (var n in parallelQuery)
            {
                Console.WriteLine("foreach {0}", n);
            }

            // You can also use ToArray, ToList, etc as with LINQ to Objects.
            var parallelQuery2 = (from num in source.AsParallel()
                                where num % 10 == 0
                                select num).ToArray();

            // Method syntax is also supported
            var parallelQuery3 = source.AsParallel().Where(n => n % 10 == 0).Select(n => n);

            Console.WriteLine("\nPress any key to exit...");
            Console.ReadLine();
        }

        static void DoSomething(int i) { Console.WriteLine("DoSomething {0}", i); }
    }
}
```



```
C:\Windows\system32\cmd.exe

start
DoSomething 130
DoSomething 180
DoSomething 190
DoSomething 120
DoSomething 100
DoSomething 110
DoSomething 160
DoSomething 140
DoSomething 150
DoSomething 170
parallelQuery.ForAll Elapsed time: 11: milliseconds
foreach start
foreach 100
foreach 110
foreach 120
foreach 130
foreach 140
foreach 150
foreach 160
foreach 170
foreach 180
foreach 190
foreach Elapsed time: 3: milliseconds

Press any key to exit...
```


Problems to solve

1. Complete the tasks in

Lab5.pdf

2. Visit

<http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx>

and run the examples