Using .NET4 Parallel Programming Model to Achieve Data Parallelism in Multi-tier Applications

Akash Verenkar

MSIT, Microsoft Corporation

Introduction

One reoccurring pattern that most application developers encounter is that of applying a set of business rules to large amounts of data. When developing such applications, developers face the arduous task of ensuring that voluminous data are processed with satisfactory performance. Since applying business logic to data is CPU intensive in nature, developers' often leverage concurrency to achieve the required performance. However developing concurrent program is extremely hard and drifts away the focus from the main business problem. The .NET Framework 4 introduces a new programming model for writing concurrent code that greatly simplifies the work of developers.

In this article I will explain how the new programming model in .NET Framework 4 can be used to achieve data parallelism. I will demonstrate how you can build a robust and scalable application that processes the data faster, yet reduce the load on the database. Further, all this is achieved without the hassle of having to create threads or directly dealing with thread pool.

First, we will create a simple hypothetical example to explain the business problem and the solution approach. Then I'll show you how to apply Parallel Programming model in .NET Framework 4 to attain data parallelism. As part of building this example, I will touch upon Parallel Language Integrated Query (PLINQ) and show how it can be used to process data faster than LINQ by adding '.AsParallel()' to LINQ constructs. Also, we will look at some new constructs like 'Parallel.For' and thread-safe data structures.

Prerequisite knowledge

For sake of brevity, the article assumes that the readers have good understanding of the following technologies:

- .NET Framework
- C#
- LINQ
- XML data types
- Lambda Expressions

Target Audience

Primary Audience: IT professionals and software application developers who want use the new features in .NET Framework 4 to achieve data/task parallelism and improve scalability across cores.

Secondary Audience: Any software developers who want to get a broad understanding of the new parallel constructs in .NET Framework 4

Business Problem

In a typical enterprise, data is stored in databases like SQL Server 2008 and different applications consume this data for various purposes. These applications may use a data access layer, built using .NET, to access the data. This data access layer can use dynamic SQL or stored procedures to access the data.

Consider a scenario where we have 'n' number of records(X_1, X_2, ... X_n) and for each record X, there are 't' number of rows (T_1, T_2, ... T_t) in the database where 't' and 'n' are approximately 1000 each, and the number of records 't' varies for different Xs. In all, there are about 1000*1000 number of rows in database for the entire input record set(X_1, X_2, ... X_n). Figure 1 gives a pictorial representation of data. Note that each colored group can be processed independently.

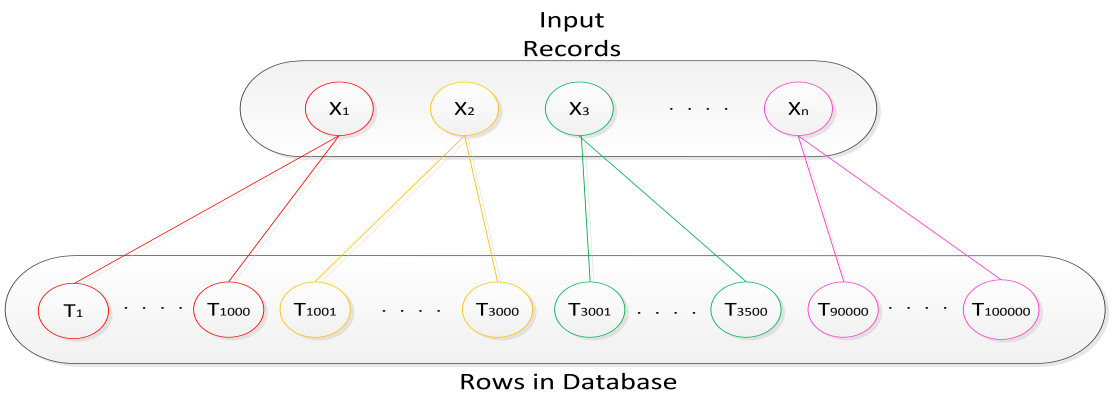


Figure 1

We need to fetch the data from the database, apply business rules and create a processed result set. The logic to apply business rules for processing the data is CPU intensive. There are different approaches to solve this problem and I will explain how I arrived at our solution using the new data parallelism features in .NET Framework 4.

Solution Approaches

As mentioned before, there are various approaches to solving the business problem of retrieving voluminous data from a data source and processing it. In this section, I will delineate the three approaches that I have used to solve the problem. For simplicity let's assume our application to be a 3-tier application:

- Front End: This is the customer facing layer that sends the Input data to the Middle Tier.
- Middle Tier: This is a Windows Communication Framework (WCF) web service which accepts the data from the front end and uses the inbuilt data access layer to access the data in Backend system.
- Backend: This is a SQL Server 2008 database instance where holds all the data the application needs to process.

Approach 1: Sequential processing

A traditional approach is to package the input data (X_1, X_2, ... X_n) into an XML and send it to a stored procedure which is deployed on the SQL Server 2008 database. In the backend, for each input record, the corresponding table entities in the database are retrieved and business rules are applied to obtain the output data. Finally, the output data is packaged into XML (or put in a table variable and do a select on it) and sent back to the middle tier service. The middle tier simply transfers the XML back to the front end. Figure 2 explains the approach:

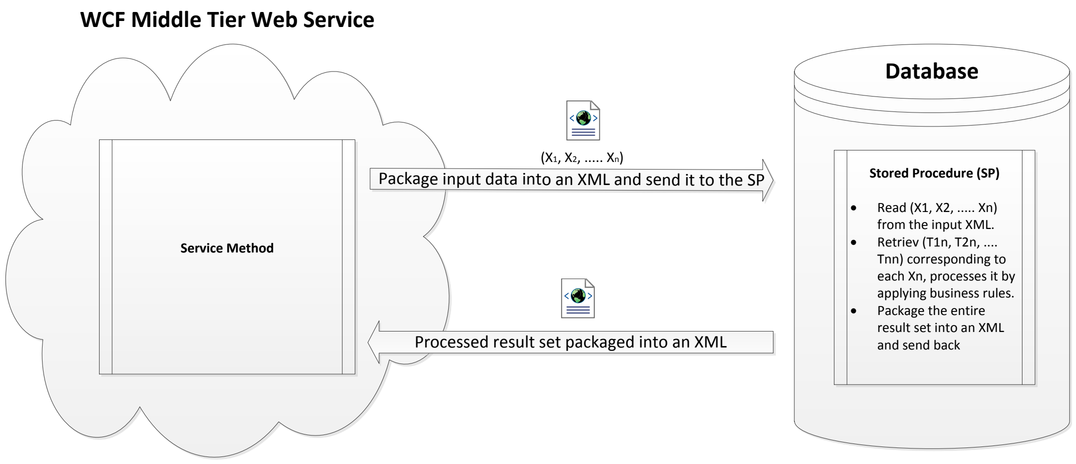

WCF Middle Tier Web Service
Service Method
(X₁, X₂, ... Xₙ)
Package input data into an XML and send it to the SP
Processed result set packaged into an XML
Database
Stored Procedure (SP)
Read (X1, X2, ... Xn) from the input XML.
Retriev (T1n, T2n, ... Tnn) corresponding to each Xn, processes it by applying business rules.
Package the entire result set into an XML and send back

Figure 2

This approach has some limitations. The major ones are as follows:

- It will bottleneck the performance of other applications sharing the SQL Database
  Since the work to apply the business rules is delegated to the backend SQL server, the performance of the application is heavily dependent on the performance of database. Further, if the database is shared across multiple applications the database instance will get overloaded with access requests and thereby degrading the performance of other applications that are using the database.
- Data is being processed sequentially
  Even though the input data (X_1, X_2, ... X_n) can be processed independently; stored procedure inherently processes the data sequentially. It is very difficult to achieve data parallelism using stored procedures.
- Unevenly distributed load between system components– Bad Architecture
  From architectural perspective, the system has unbalanced load, with Front End and Middle Tiers sitting idle while all work is done by the backend.

Of all the limitations in the above approach, the most severe one is the inability to process data in parallel. So I devised an approach that would parallelize the database access.

Approach 2: Parallelizing Database Access

Since SQL Server database can handle multiple calls simultaneously and implements concurrent access, I decided to partition the input data and invoke the stored procedure simultaneously for each partition. Figure 3 explains the approach:

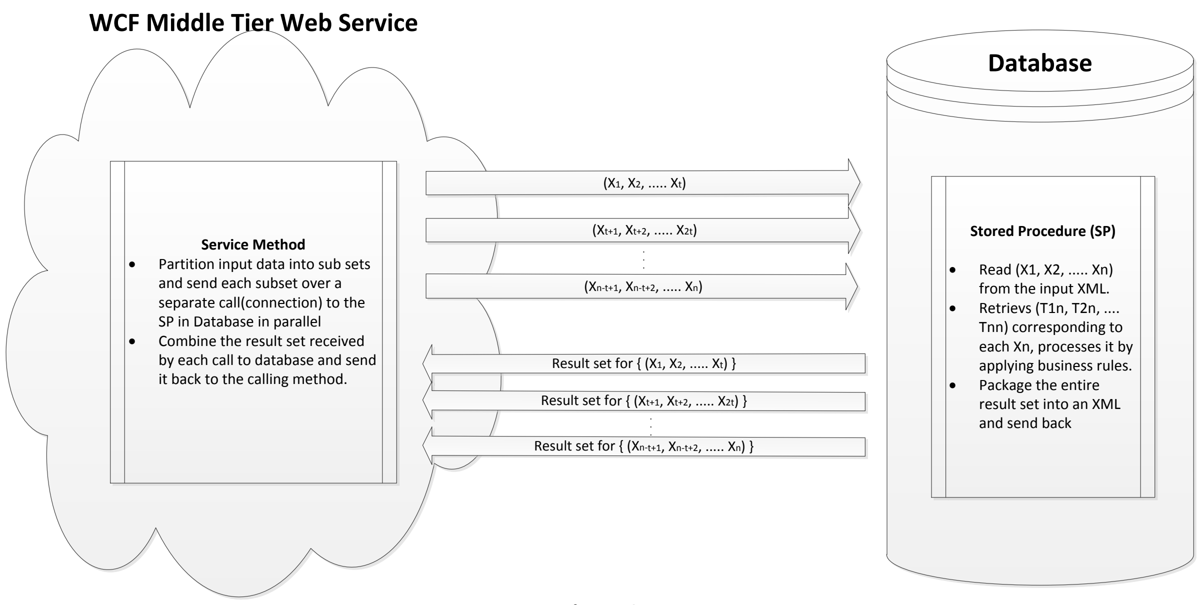

WCF Middle Tier Web Service
Service Method
Partition input data into sub sets and send each subset over a separate call(connection) to the SP in Database in parallel
Combine the result set received by each call to database and send it back to the calling method.
$(X_1, X_2, ... X_t)$
$(X_{t+1}, X_{t+2}, ... X_{2t})$
$(X_{n-t+1}, X_{n-t+2}, ... X_n)$
Result set for { $(X_1, X_2, ... X_t)$ }
Result set for { $(X_{t+1}, X_{t+2}, ... X_{2t})$ }
Result set for { $(X_{n-t+1}, X_{n-t+2}, ... X_n)$ }
Database
Stored Procedure (SP)
Read (X1, X2, ... Xn) from the input XML.
Retrievs (T1n, T2n, ... Tnn) corresponding to each Xn, processes it by applying business rules.
Package the entire result set into an XML and send back

Figure 3

This way I was able to reduce the overall processing time but limitation 1 (performance bottleneck at SQL) and limitation 3 (uneven load across the tiers) of the first approach still existed. Also I was evenly partitioning the input data without considering how much time it will take to process individual records. It might so happen that some partitions take more time to process than others as the corresponding data in DB for each input record was not taken into consideration at all while partitioning. One way to overcome this is to invoke the stored procedure for each input record. However, since the maximum number of SQL connections available is far lesser than the number of input records, this approach (parallelizing database access) was infeasible.

Approach 3: Using Data Parallelism in .NET Framework 4

Since parallelizing the database access was not of much help, it helped me to devise the third approach which catered to all the 3 limitations in our sequential approach. I created a new stored procedure that accepts an input XML with all the input data ($X_1, X_2, ... X_n$) and returns all the t rows ($T_1, T_2, ... T_t$) associated with the each X in input data. This way all the relevant data is transferred to the middle tier and can be processed in parallel (in the middle tier service method) using parallel programming constructs available in .NET Framework 4. Once the data is processed, the middle tier sends the processed data back the front end. Figure 4 explains the approach:

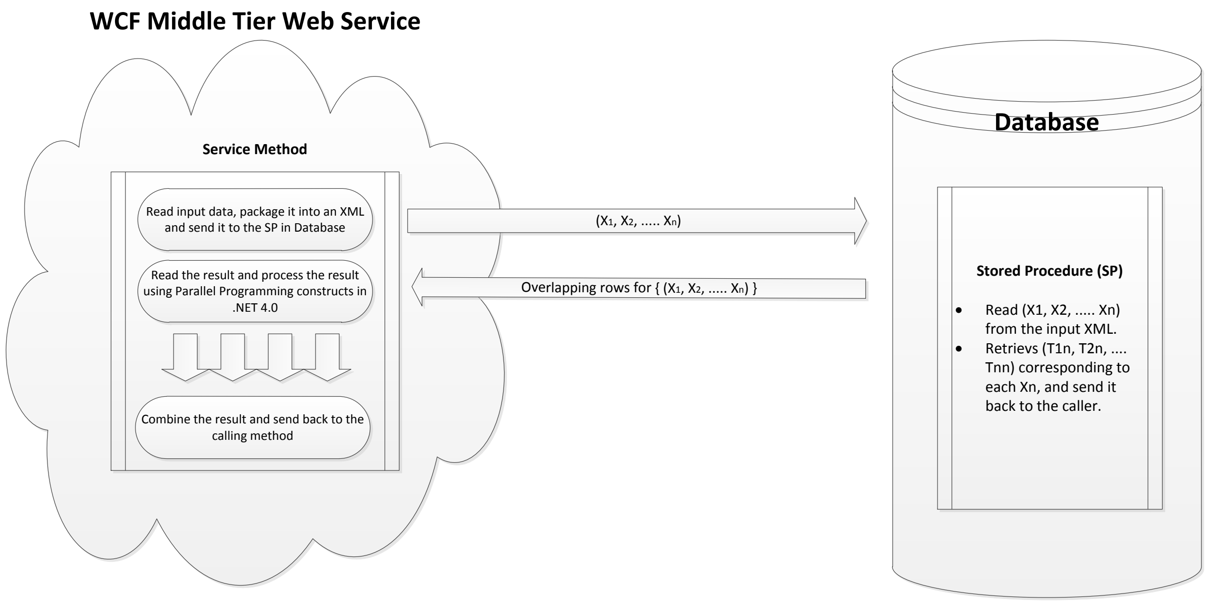

WCF Middle Tier Web Service
Service Method
Read input data, package it into an XML and send it to the SP in Database
Read the result and process the result using Parallel Programming constructs in .NET 4.0
Combine the result and send back to the calling method
(X1, X2, ... Xn)
Overlapping rows for { (X1, X2, ... Xn) }
Database
Stored Procedure (SP)
Read (X1, X2, ... Xn) from the input XML.
Retrievs (T1n, T2n, ... Tnn) corresponding to each Xn, and send it back to the caller.

Figure 5

I know by now there must be many questions/thoughts coming to your mind - "how will I process these 'N' number of rows in parallel?", "How many threads will I create?", "How will I join the data that is processed in parallel?", "It is too much of an overhead to create optimal number of threads", "How will I make my application scalable" etc. My response to all these worries is "Do not worry!!", the parallel programming model in .NET Framework 4 will help us do all this without having to deal with threads or thread pool directly.

In the following example, I will show how to apply the different features of .NET parallel programming model to attain data parallelism. The following example should also offer a glimpse of the performance gains attained by processing data in parallel and all this can be done without having to deal with the intricacies of threading.

Code Example

Background information

This section provides the background information for a software application used by a multinational software distributor. The distributor uses a database to store information pertaining to the software it sells. This is a properly designed database and we will just consider a part of the database for our example. Each software product type that is distributed has a unique identifier say ItemTypeID of type 'int' and each product have a unique identifier say ItemID of type 'int'. There are two tables that maintain this data: ItemType and ItemsRepository with ItemTypeID and ItemID as the primary key. ItemID and ItemtypeID together uniquely identify an item. Further, there is another table SoldItems, that records all the items sold to retailers in batches. The fields on that table are

- **ID**: is unique Identifier for rows in this table and is the primary key
- **ItemTypeID** maps to the ItemTypeID in the ItemType table.
- **ItemIDStart** and **ItemIDEnd** define a range of items that are sold to a retailer in one transaction.
- **RetailerID:** ID that identifies the retailer to whom the items were sold.
- **Date**: The date on which the transaction was made.

- **Comments**: Any comments specific to the transaction.

Listed below is an example of the table entries with row one showing that Items with ItemID 10 to 100 of ItemType with ItemTypeID 32 were sold off to Retailer with RetailerID 1 on 9th December 2009. Note that the table has entries in the order of 100000, with ItemIDStart >=0 and ItemIDEnd <=999999.

ID	ItemTypeID	ItemIDStart	ItemIDEnd	RetailerID	Date	Comments
1	32	10	100	1	09-12-2009	Online Transaction
2	32	250	5000	1	10-10-2009	Cash Payment
3	43	32	89	9	15-11-2009	Payment Pending
…	…	…	…	…	…	…

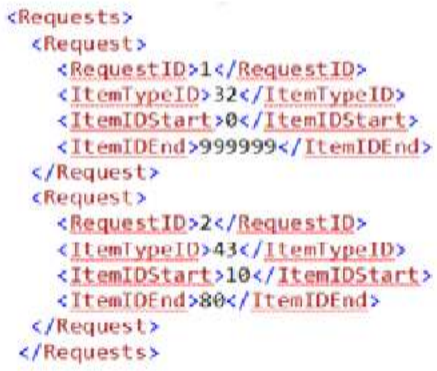
Table 1: Sold Items

Given ItemTypeID, ItemIDStart and ItemIDEnd, a customer would want to know which items are sold off and which are still on the shelf. For example, for the given input data ItemID = 32, ItemIDStart = 0 and ItemIDEnd = 999999, the output should be

ItemTypeID	ItemIDStart	ItemIDEnd	Result
32	0	9	OnShelf
32	10	100	SoldOff
32	101	249	OnShelf
32	250	5000	SoldOff
32	5001	999999	SoldOff

Table 2: Tabular representation of output

The front end can send multiple (maximum of 1000) input queries in an XML as shown in Figure5. Each of this request maps to the input data (X_1, X_2, ... X_n) listed in the business problem section.



```
<Requests>
  <Request>
    <RequestID>1</RequestID>
    <ItemTypeID>32</ItemTypeID>
    <ItemIDStart>0</ItemIDStart>
    <ItemIDEnd>999999</ItemIDEnd>
  </Request>
  <Request>
    <RequestID>2</RequestID>
    <ItemTypeID>43</ItemTypeID>
    <ItemIDStart>10</ItemIDStart>
    <ItemIDEnd>80</ItemIDEnd>
  </Request>
</Requests>
```

Figure 5

Implementation Details

The middle tier service method will

1) Receive (X_1, X_2, ... X_n) as an input XML from the front end
2) Send (X_1, X_2, ... X_n) to the SQL Database and receive (T_1, T_2, ... T_n), where T_n => Set of records in database corresponding to X_n.
3) Evaluates the SoldItems and the OnShelfItems for each X_n, say R_n.

4) Return back all (R_1,R_2,.....R_n) with a proper mapping between (X_1, X_2, ... X_n) $\leftarrow\rightarrow$(R_1,R_2,.....R_n). RequestID will be used to map R_n to corresponding X_n

To show the different constructs of Parallel Task Library and PLINQ, I have implemented the middle tier functionality in three different ways. To keep our example short, let us assume that:

1) The input XML string is sent to a stored procedure in the database. It is loaded into an XDocument for evaluation in Middle Tier.
2) The stored procedure queries the SoldItems table returns back a DataTable with four columns:
    a. ID – Identity column
    b. RequestID – Mapped to the RequestID of input XML
    c. ItemIDStart – ItemIDStart of a record existing in database
    d. ItemIDEnd– ItemIDEnd of a record existing in database
3) To store a single input request (X) I have created a request holder class:

```
public class Request
    {
        int itemTypeID, requestId, itemIdStart, itemIdEnd;
        public int ItemTypeID
        {
            get { return ItemTypeID; }
            set { ItemTypeID = value; }
        }
        public int RequestId
        {
            get { return requestId; }
            set { requestId = value; }
        }
        public int ItemIdStart
        {
            get { return itemIdStart; }
            set { itemIdStart = value; }
        }
        public int ItemIdEnd
        {
            get { return itemIdEnd; }
            set { itemIdEnd = value; }
        }
    }
```

4) To hold a single output record (R) I have created a result holder class:

```
public class ResultHolder
    {
        int requestId, itemIdStart, itemIdEnd;
        string comment;
        public ResultHolder(int requestId, int itemIdStart, int itemIdEnd, string comment)
        {
            this.requestId = requestId;
            this.itemIdStart = itemIdStart;
            this.itemIdEnd = itemIdEnd;
            this.comment = comment;
        }
        public int ItemIdEnd
        {
            get { return itemIdEnd; }
            set { itemIdEnd = value; }
        }

        public int ItemIdStart
        {
            get { return itemIdStart; }
            set { itemIdStart = value; }
        }

        public int RequestId
        {
            get { return requestId; }
            set { requestId = value; }
        }
    }
```

Now I'll show you the three different ways in which I have evaluated the same problem. The Evaluation method accepts the DataTable and XDocument as input and returns a an IEnumerable<ResultHolder>

Approach 1: Using PLINQ and ConcurrentBag

```
1.  private IEnumerable<ResultHolder> ProcessData_PLINQ_Concurrentbag(DataTable dataTable, XDocument inputDoc)
2.  {
3.    //Concurrent Data Structure to hold the output data
4.    ConcurrentBag<ResultHolder> resultBag = new ConcurrentBag<ResultHolder>();
5.    string SoldItem = "SoldOff";
6.    string OnShelfItem = "OnShelf";

7.  //For every Request Node in inputDoc evaluate the SoldItems and OnShelfItems from the
8.    //DataTable returned by the Store Procedure in Parallel
9.    (from x in inputDoc.Root.Elements("Request").AsParallel()
10.    select new Request
11.    {
12.        RequestId = Int32.Parse(x.Element("RequestID").Value),
13.         ItemTypeID = Int32.Parse(x.Element("ItemTypeID").Value),
14.        ItemIdStart = Int32.Parse(x.Element("ItemIDStart").Value),
15.        ItemIdEnd = Int32.Parse(x.Element("ItemIDEnd").Value)
16.    }).ForAll(req =>
17.               {
18.                   //Get a List of all DataRows that Map to this Request.
19.                   var tempResultList = (from dr in dataTable.AsEnumerable()
20.                                         where ((int)dr["RequestID"] == req.RequestId)
21.                                         select dr).OrderBy(
22.                                              (dr) => { return (int)dr["ItemIDStart"]; }
23.                                              ).ToList<DataRow>();
24.                   //Main Logic to evaluate the SoldItems and OnShelfItems
25.                   //Populate the ResultBag datastructure with Objects of type ResultHolder with the corresponding
26.                   //string in the Comment Property - SoldItem or OnShelfItem (Public const strings)
27.                    int pointer = req.ItemIdStart;

28.                   for (int j = 0; j < tempResultList.Count; j++)
29.                   {
30.                       int ss = (int)tempResultList[j]["ItemIDStart"];
31.                       int se = (int)tempResultList[j]["ItemIDEnd"];
32.                       if (ss <= pointer)
33.                       {
34.                           resultBag.Add(new ResultHolder(req.RequestId, pointer,
                                                              req.ItemIdEnd < se ? req.ItemIdEnd : se,
                                                              SoldItem));
35.                       }
36.                       else
37.                       {
38.                           resultBag.Add(new ResultHolder(req.RequestId, pointer, ss - 1, OnShelfItem));
39.                           resultBag.Add(new ResultHolder(req.RequestId, ss, req.ItemIdEnd < se ? req.ItemIdEnd : se,
                                                              SoldItem));
40.                       }

41.                       pointer = se + 1;
42.                   }
43.                   if (pointer < req.ItemIdEnd)
44.                   {
45.                       resultBag.Add(new ResultHolder(req.RequestId, pointer, req.ItemIdEnd, OnShelfItem));
46.                   }
47.
48.               });
49.    return resultBag;
50. }
```

Important Lines

Line Number	Description
Lines 4	The **System.Collections.Concurrent** namespace in .NET 4 provides several thread-safe collection classes that should be used in place of the corresponding types in the System.Collections and System.Collections.Generic namespaces whenever multiple threads are accessing the collection concurrently. ConcurrentBag(T) Class represents a thread-safe, unordered collection of objects. We use it to merge our results from different threads in a thread safe manner.
Line 9	This is a normal Linq query with just an addition of ".AsParallel()". This enables parallelization of a query.
Line 16	The ForAll API processes each element in the input in Parallel. You have to be careful not to use shared non-thread safe variables or objects in a ForAll as this will

	give erroneous results. Hence we used ConcurrentBag.
Line 19	Here we have a normal Linq query to filter the records returned from the database which map to the request node being processed. Here we could have used .AsParallel(). But this will result in nested parallelism. Nested Parallelism can be beneficial at times as it breaks down the problem further (and increases the performance). But at times it can also be an overhead (and lower the performance). In my case, it was becoming an overhead so the query was kept sequential.

Approach 2: Using the Parallel.Foreach API along with ConcurrentBag

```
 1. private IEnumerable<ResultHolder> ProcessData_ParallelFor(DataTable dataTable, XDocument inputDoc)
 2. {
        //Concurrent Data Structure to hold the output data
 3.     ConcurrentBag<ResultHolder> resultBag = new ConcurrentBag<ResultHolder>();
 4.     string SoldItem = "SoldOff";
 5.     string OnShelfItem = "OnShelf";
 6.     Parallel.ForEach<Request>(inputDoc.Root.Elements("Request").Select(
 7.                                req => new Request
 8.                                {
 9.                                    RequestId = Int32.Parse(req.Element("RequestID").Value),
10.                                    ItemTypeID = Int32.Parse(req.Element("ItemTypeID").Value),
11.                                    ItemIdStart = Int32.Parse(req.Element("ItemIDStart").Value),
12.                                    ItemIdEnd = Int32.Parse(req.Element("ItemIDEnd").Value)
13.                                }), (req) =>
14.                                {
15.                                    //Get a List of all DataRows that Map to this Request.
16.                                    var tempResultList = (from dr in dataTable.AsEnumerable()
17.                                                          where ((int)dr["RequestID"] == req.RequestId)
18.                                                          select dr).OrderBy(
19.                                                               (dr) => { return (int)dr["ItemIDStart"]; }
20.                                                               ).ToList<DataRow>();
21.                                    //Main Logic to evaluate the SoldItems and OnShelfItems
22.                                    //Populate the ResultBag datastructure with Objects of type ResultHolder with the
23.                                    //corresponding
24.                                    //string in the Comment Property - SoldItem or OnShelfItem (Public const strings)
25.                                    int pointer = req.ItemIdStart;
26.                                    for (int j = 0; j < tempResultList.Count; j++)
27.                                    {
28.                                        int ss = (int)tempResultList[j]["ItemIDStart"];
29.                                        int se = (int)tempResultList[j]["ItemIDEnd"];
30.                                        if (ss <= pointer)
31.                                        {
32.                                            resultBag.Add(new ResultHolder(req.RequestId, pointer, req.ItemIdEnd < se ?
                                                                  req.ItemIdEnd : se,
                                                                  SoldItem));
33.                                        }
34.                                        else
35.                                        {
36.                                            resultBag.Add(new ResultHolder(req.RequestId, pointer, ss - 1, OnShelfItem));
37.                                            resultBag.Add(new ResultHolder(req.RequestId, ss, req.ItemIdEnd < se ?
                                                                  req.ItemIdEnd : se,
                                                                  SoldItem));
38.                                           }
39.                                        pointer = se + 1;
40.                                    }
41.                                    if (pointer < req.ItemIdEnd)
42.                                    {
43.                                        resultBag.Add(new ResultHolder(req.RequestId, pointer, req.ItemIdEnd,
44.                                                                OnShelfItem));
45.                                    }
46.                                });
47.     return resultBag;
```

Line Number	Description
Line 6	Parallel.ForEach() loop can be used to enable data parallelism over any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable(T) data source. It is like a Foreach loop but in this case the source collection is partitioned and the work is scheduled on multiple threads based on the system environment. The more processors on the system, the faster the parallel method runs. Please refer How to: Write a Simple Parallel.ForEach Loop to understand the syntax of the API
Line 14	Here the body of our ForEach loop starts
Line 16	Here we have a normal Linq query to filter the records returned from the database which map to the Request node being processed. Here we again avoided parallelism as nested parallelism did not provide any benefits.

Approach 3: Using PLINQ without using ConcurrentBag

In the above examples a lot of synchronization occurs since we used Concurrent Bag. Multiple calls to ConcurrentBag.Add() method inside the ForEach loop or in the PLINQ query requires synchronization. The following example shows how to avoid synchronization costs by using PLINQ select operator. In the example only merging the results into a list will add an overhead of sequential processing. But this overhead is far less than the cost of synchronization involved in this particular scenario.

```
 1.  private List<ResultHolder> ProcessData_PLINQOnly(DataTable dataTable, XDocument inputDoc)
     {
 2.      var query = inputDoc.Root.Elements("Request").AsParallel().WithMergeOptions(ParallelMergeOptions.FullyBuffered)
 3.                          .Select(
 4.                               x =>
 5.                                {
 6.
 7.                                    int RequestId = Int32.Parse(x.Element("RequestID").Value);
 8.                                    int ItemId = Int32.Parse(x.Element("ItemID").Value);
 9.                                    int RangeStart = Int32.Parse(x.Element("RangeStart").Value);
10.                                    int RangeEnd = Int32.Parse(x.Element("RangeEnd").Value);
11.
12.                                    List<ResultHolder> tempList = new List<ResultHolder>();
13.                                    //Get a List of all DataRows that Map to this Request.
14.                                    var tempResultList = (from dr in dataTable.AsEnumerable()
15.                                                          where ((int)dr["RequestID"] == RequestId)
16.                                                          select dr).OrderBy(
17.                                                               (dr) => { return (int)dr["RangeStart"]; }
18.                                                               ).ToList<DataRow>();
19.                                    //Main Logic to evaluate the SoldItems and OnShelfItems
20.                                       //Populate the ResultBag datastructure with Objects of type ResultHolder with the
21.                                    //corresponding string in the Comment Property - SoldItem or OnShelfItem
22.                                    //(Public const strings)
23.                                    int pointer = RangeStart;
24.
25.                                    for (int j = 0; j < tempResultList.Count; j++)
26.                                    {
27.                                        int ss = (int)tempResultList[j]["RangeStart"];
28.                                        int se = (int)tempResultList[j]["RangeEnd"];
29.                                        if (ss <= pointer)
30.                                        {
31.                                            templist.Add(new ResultHolder(RequestId, pointer, RangeEnd<se?RangeEnd:se,
                                                                          SoldItem));
32.                                        }
33.                                        else
34.                                        {
35.                                            templist.Add(new ResultHolder(RequestId, pointer, ss - 1, OnShelfItem));
36.                                            templist.Add(new ResultHolder(RequestId, ss, RangeEnd<se?RangeEnd:se,
                                                                          SoldItem));
37.                                        }
38.                                        pointer = se + 1;
39.                                    }
40.                                    if (pointer < RangeEnd)
41.                                    {
42.                                        templist.Add(new ResultHolder(RequestId, pointer, RangeEnd, OnShelfItem));
43.                                    }
44.                                    return templist;
45.                                });
46.                  List<ResultHolder> returnList = new List<ResultHolder>();
47.                  foreach (var r in query)
48.                  {
49.                      returnList.AddRange(r);
50.                  }
51.                  return returnList;
52.
53.  }
```

Line Number	Description
Line 2	Here we use the AsParallel API which converts the IEnumerable source into ParallelQuery. We also use a Select operator which is an extension method implemented by the ParallelEnumerable class.It has a lambda expression which returns List<ResultHolder>.
Line 47	PLINQ like Linq uses deferred execution. Hence we force the execution of the query using the foreach loop.
Line 51	returns all the ResultHolder objects returned by the PLINQ.

Based on the three implementations it is hard to generalize which approach is better than the rest. Depending on the scenarios, the amount of data to be processed and the type of processing each method has its benefits and limitations.

Performance Results

I did a benchmark test on a machine following configuration:

Processor: Intel® Core™ 2 Duo CPU T7500 @ 2.20GHz 2.20GHz

RAM: 4:00 GB

OS: 64 bit Windows 7 Enterprise.

Figure 6 shows the time taken from by the middle tier.

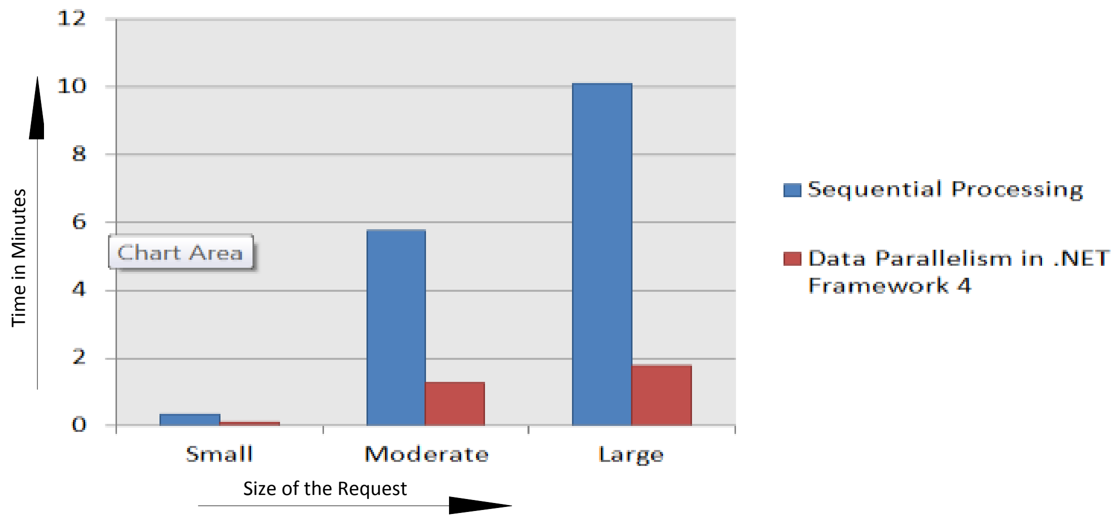


Figure 6

Note: Figure 6 shows results for only two approaches – *Sequential Processing and Data Parallelism in .NET Framework 4.* The approach, Parallelizing database access is not included since the approach was deemed as infeasible.

Conclusion

Developers should hunt for scenarios where they can efficiently use the new .NET 4 parallel constructs to leverage concurrency. Some key points a developer should keep in mind while writing concurrent code include:

- Look out for embarrassingly parallel scenarios in your code. An embarrassingly parallel problem "is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks" [1].
- For non-embarrassingly parallel scenarios, look for avenues to parallelize. Care has to be taken such that parallel loops/tasks do not modify/write to shared memory. When shared memory has to be modified, use thread safe data structures. When using thread safe data structures, one must keep in mind that the threads accessing the data structure needs

synchronization and might slow down the performance in some scenarios. Further, developers should be careful when trying to parallelize unsafe code as incorrect implementation can lead to disasters.

- Developers should study the cost incurred in threading, such as context switching which happens at operating system level. One should keep in mind that there will be other applications running on the same machine fighting for CPU and other system resources. Over-subscription to parallelism may have deteriorating effects. Use parallelism only if the cost of parallelizing is less than cost of executing the CPU intensive operation.

Also when I said before that the code becomes scalable over available cores if using .NET Framework 4, it is not 100% true. There is a upper limit to which the application can scale ie 64/32 for Task Parallel Library. And 63/31 for PLINQ as in PLINQ one thread will be needed to iterate through the results. Along with using parallel programming I would also like to stress upon the key changes that we made in the entire architecture. When developing multi-tier applications certain key points have to be kept in time for a good design

- It is always a good practice to divide the work load between different systems/tiers. The more uniform the work division will be between the tiers, the better will be the overall throughput of the operation. It is a good practice to have Tiers dedicated for specific taks like Presentation Tier, Business Logic Tier, Data Access Tier etc.
- When considering division of work across tiers, keep in mind that for CPU intensive tasks, .NET code is better than SQL queries. Hence CPU intensive tasks like data processing should be done at middle tier. While the database is master in data manipulation operations like Create/Read/Update/Delete (CRUD) operations.
- Further due to limited SQL Server connections and cost considerations (SQL Server licenses is per CPU sockets) computational intensive tasks should be done at middle or front end and access database for CRUD operations only.
- Since we are transferring data over the network, some overhead in terms of running time will be added because of network latency. When using ADO.NET to query a database the “packet size” defines the maximum amount of data that can be accommodated in a single call to database. If the data size increases this bound then multiple calls will be made to the database. Every call to the database will add network latency to it. When evaluating the benefits of parallelizing the computational work, this overhead must be considered.
- There will be an additional cost in terms of space needed like RAM on the middle tier machine where the data is being pulled into. This should also be considered when evaluating the benefits of parallelizing. A different approach to minimize the cost of space can be utilized here: ADO.NET allows data to be streamed using a DataReader object. This can be used to pull data in batches, do the CPU intensive tasks on it, clear the memory and then get the next batch. When streaming data, “packet size” plays a vital role. Consider the following example:

  Suppose the packet size is 200bytes. Total data to be fetched is 840 bytes and the batch size is 210bytes. Then two network calls will be needed to fetch one batch. The first batch is full but the second batch doesn’t utilize the complete packet size. Total there will be 8 calls needed to fetch complete data. But in the same case if the batch size was 400 bytes then entire data would have been fetched in 5 calls over network.

Hence select the batch size considering both the factors, Space(RAM) and packet size of SQL connection.

- PLINQ and TPL provides an option of writing custom partitioners to partition the source into multiple segments that can be accessed concurrently by multiple threads. When pulling data from database some additional data can be pulled that can help identify related data. Custom Partitioners can then be written to partition related data into same segments so that same thread can work on it. This will help increase efficiency of the parallel constructs. Refer the MSDN article Custom Partitioners for PLINQ and TPL for more details.

In short, I would say that the new Parallel features in .NET Framework 4 helps you to write concurrent code faster and with less complexity. However developers have to ensure the correctness of their concurrent code. Refer to Patterns_of_Parallel_Programming_CSharp book for a deep insight into various parallel programming patterns and how they are implemented in .NET Framework 4.

References:

1. Embarrassingly parallel, Wikipedia.
   URL: http://en.wikipedia.org/wiki/Embarrassingly_parallel

2. PATTERNS OF PARALLEL PROGRAMMING by Stephen Toub.
   URL:http://download.microsoft.com/download/3/4/D/34D13993-2132-4E04-AE48-53D3150057BD/Patterns_of_Parallel_Programming_CSharp.pdf

3. MSDN – Parallel Programming in .NET Framework.
   URL: http://msdn.microsoft.com/en-us/library/dd460693(VS.100).aspx