

White Paper



Multicore Programming With .NET 4 Parallel Extensions

Humayun Khan Pathan

Abstract

Most of the personal computers and workstations today have multicore processors. However most software programs are not designed to make use of multicore processors and hence even though we run these programs on the new machines equipped with multicore processors, we don't see sizable improvements in application performance. The key to improved performance is in parallelizing the code and distributing the work amongst multiple cores, but writing programming logic to achieve this is complex. In the past, parallelization required low-level manipulation of threads and locks, but now with the help of Visual Studio 2010 and the .NET Framework 4 Parallel Extensions, programmers have enhanced support for parallel programming.

This paper is aimed at giving a brief overview of the .NET 4 Parallel Extensions using which programmers can write efficient, fine-grained and scalable code without having to work directly with threads and thread pool.

1. Introduction

Majority of the programs written today primarily target single core CPUs and don't have in built intelligence to benefit from multi core processors. Prior to the release of .NET 3.5, in order to take advantage of the multicore processors, developers had to write complex multithreaded code and also had to meticulously handle all the synchronization primitives. Thus, writing thread safe parallel programs using worker threads and delegates has become a challenging task for developers. The Parallel Extension libraries have been designed specifically to help programmers to overcome these difficulties. The Parallel Extensions contains sophisticated algorithms for dynamic work distribution which can intelligently distribute varying workloads to different processor cores. Programs developed using Parallel Extensions will run on a multi core machine and they can allocate work to the processor cores depending on their availability. In absence of multiple cores these programs can run on a single core machine without any changes.

In this paper some preliminary knowledge of Delegates and Lambda Expressions is expected from the reader. [1][2]

2. Parallel Extensions

To the core libraries of the Microsoft .NET Framework, Parallel Extensions are a new addition. Parallel Extensions makes it easier for the developers to write programs for the multi core processors by taking off the burden of dealing explicitly with the complexities of threads and locks.

The multithreading API's new to .NET Framework 4 for leveraging multicore processors are:

- Parallel LINQ (PLINQ)
- Task Parallelism Constructs
- The Parallel Class
- Concurrent Collections & Coordination Data Structures

These API's are collectively known as Parallel Extensions or The Parallel Framework.

The following illustration provides an overview of the .NET Framework 4 stack:

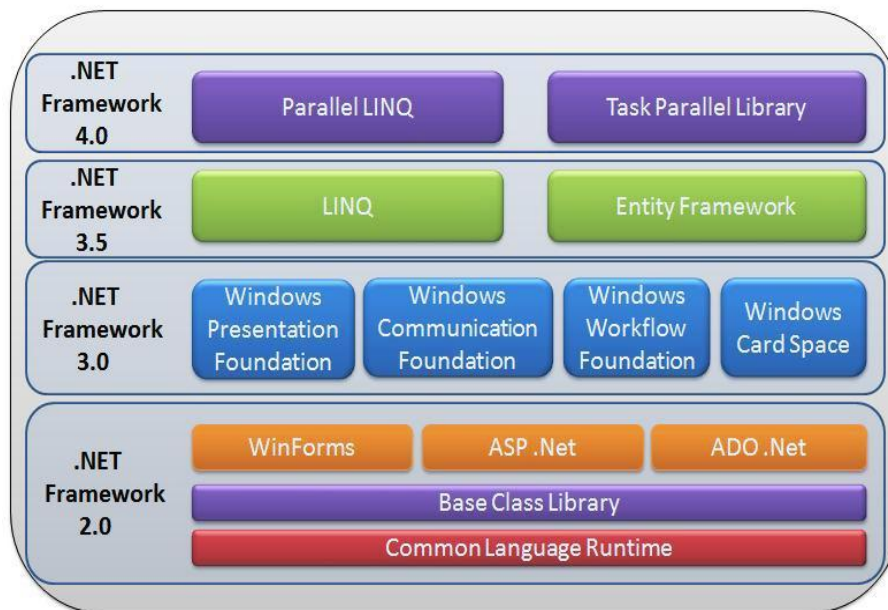


Fig-1: The .NET Framework Stack

The following illustration provides an overview of the Parallel Programming architecture in the .NET Framework 4:

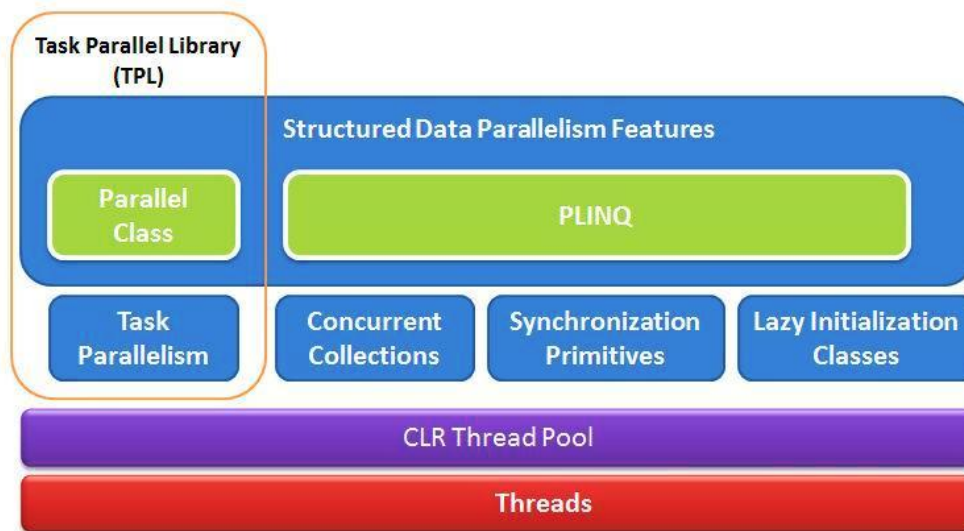


Fig-2: Architecture For Parallel Programming in the .NET Framework 4

3. What is “Task”?

We can roughly say that a “Task” is an asynchronous operation. In an abstract way a Task resembles the creation of a new thread or Thread Pool work item. There are two primary benefits of Tasks:

- **Scalable and Efficient use of the system resources.**

Tasks get queued to the Thread Pool, which has been enhanced with algorithms like “hill-climbing” [3] that determine and adjust the number of threads to maximize the throughput. This makes tasks relatively lightweight, and to enable fine grained parallelism we can create as many of them. To provide load balancing, widely known work stealing algorithms [4] [5] are implemented.

- **More programmatic control is possible with “Tasks” than with a thread or work item.**

Tasks and the framework built around them provide us a rich set of APIs that support Task waiting, Task cancellation, Task continuations, exception handling, custom Task scheduling, and much more.

4. Task Parallel Library (TPL)

The “Parallel” Class and the “Task Parallelism Constructs” are collectively called the Task Parallel Library or TPL.

The Task Parallel Library (TPL) is the task parallelism part of the Parallel Extensions to .NET. It exposes parallel constructs such as “*Parallel.For ()*” and “*Parallel.ForEach ()*” loops, using regular method calls and delegates. TPL boosts the developer’s productivity by simplifying the process of parallelization of applications. TPL can dynamically scale the degree of concurrency to efficiently use all the processor cores that are available in the machine. In addition to this, TPL also handles the partitioning of work, the scheduling of threads on the Thread Pool, manages the state, and handles many other low level details.

Partitioning the work among threads could be done in two ways:

- Task parallelism
- Data parallelism

The Task Parallel Library supports both types of parallelism.

4.1 Task Parallelism

The term “Task parallelism” refers to a group of tasks running concurrently. With task parallelism, each thread performs a different task.

The creation of tasks and executing them could be done in two ways:

- Creating and Executing Tasks Implicitly
- Creating and Executing Tasks Explicitly

4.1.1 Creating and Executing Tasks Implicitly

The “*Parallel.Invoke()*” method provides a convenient way of running any number of arbitrary statements concurrently. *Parallel.Invoke()* tries to execute a collection of Action delegates concurrently, and waits for the completion of all of them. The following example shows how a “*Parallel.Invoke()*” call creates and starts two tasks which can run in parallel:

```
// Perform Two Tasks In Parallel On The “words” Source Array
Parallel.Invoke(
    () =>
    {
        Console.WriteLine("Beginning the first task...");
        GetWordCount(words);
    }, // End of First Action
    () =>
    {
        Console.WriteLine("Beginning the second task...");
        GetMostCommonWord(words);
    } //End of Second Action
); //End of parallel.invoke
```

On the surface of it the above example looks like a convenient shortcut for creating, running and waiting on two “Task” objects. But there’s an important difference, “Parallel.Invoke” works efficiently even if we pass in an array of one million delegates. This is because “*Parallel.Invoke()*” will not create a separate Task for each delegate, instead it partitions the large number of input elements into batches and then it assigns each batch to a handful of underlying Tasks.

Try out this simple program:

```
static void Main(string[] args)
{
    Action[] tasks = new Action[100000];
    for (int i = 0; i < 100000; i++)
    {
```

```

        tasks[i] = new Action(() => DoSomeWork());
    }
    Parallel.Invoke(tasks);
}
private static void DoSomeWork()
{
    Console.WriteLine("Task No:{0} Started...", Task.CurrentId);
    Thread.SpinWait(100);
}

```

When the above code was executed on a dual core machine only 5 tasks were created to execute all the 1 million action delegates. However the number of tasks which will get created can vary depending on the number of cores available and also on various other factors.

4.1.2 Creating and Executing Tasks Explicitly

Task parallelism is the low level approach to parallelization with .NET 4 Parallel Extensions. The classes for working at low level with the “Tasks” are defined in the “System.Threading.Tasks” namespace:

| Class | Purpose |
|----------------------|---|
| Task | For managing a unit of work |
| Task<TResult> | For managing a unit of work with a return value |
| TaskFactory | For creating tasks |
| TaskFactory<TResult> | For creating tasks and continuations with some same return type |
| TaskScheduler | For managing the scheduling of tasks |
| TaskCompletionSource | For manually controlling a task’s workflow |

For greater control over task execution, we have to work with “Task” objects more explicitly. Tasks provide many powerful features to manage units of work, including the ability to:

- Tune a task’s scheduling
- Establish a parent/child relationship when a “Task” is started from another “Task”
- Implement cooperative cancellation
- Wait on a set of tasks without a signaling construct
- Attach “continuation” tasks
- Scheduling of a continuation
- Propagate exceptions to parents.

The following example shows some explicit task controlling features:

```

// Waiting On A Task.
Task task1 = Task.Factory.StartNew(() => DoSomeWork(1000000));

```

```

task1.Wait();
Console.WriteLine("Task1 Has completed Execution.");

// Waiting On A Task With A Specific Timeout.
Task task2 = Task.Factory.StartNew(() => DoSomeWork(1000000));
task2.Wait(100); //Wait for 100 ms.

if (task2.IsCompleted)
    Console.WriteLine("Task2 Has Completed Execution.");
else
    Console.WriteLine("Timed Out Before Task2 Can Complete.");

// Wait For All The Tasks To Complete.
Task[] tasks = new Task[10];
for (int i = 0; i < 10; i++)
{
    tasks[i] = Task.Factory.StartNew(() => DoSomeWork(1000000));
}
Task.WaitAll(tasks);

```

There is one very interesting method in the “Task” class, “*Task.WaitAny()*”. This method takes an array of “Task” objects and tries to run each of these tasks in parallel and it will return the index of that Task which has completed first among all the other Tasks. This feature is particularly useful in scenarios where we need to compare different algorithms and pick the one that performs the best.

```

List<int> unsortedElements = new List<int>();
Random randNumGen = new Random();
for (int i = 0; i < 10; i++)
    unsortedElements.Add(randNumGen.Next());

Task<List<int>>[] sortBenchMark = new Task<List<int>>[3];

sortBenchMark[0] = Task<List<int>>.Factory.StartNew(() => BubbleSort(unsortedElements));

sortBenchMark[1] = Task<List<int>>.Factory.StartNew(() => QuickSort(unsortedElements));

sortBenchMark[2] = Task<List<int>>.Factory.StartNew(() =>
InsertionSort(unsortedElements));

//Try Three Different Sorting Techniques And Take Result For The One Which Completes
First
int index = Task.WaitAny(sortBenchMark);
List<int> sortedElements = sortBenchMark[index].Result;

```

```
Console.WriteLine("sortBenchmark[{0}] completed first.", index);
```

In the above example we tried to sort the “*unsortedElements*” list using 3 different sorting techniques and took the result from the sorting technique which completes first. Then the program displays which sorting algorithm completed first.

4.2 Data Parallelism

The programming technique of partitioning a large data set into small chunks that can be processed in parallel is called Data parallelism. The smaller pieces of data set which are processed in parallel are combined back into a final result data set. The technique of Data Parallelism enables the developers to convert a long sequential data processing job which was not capable of utilizing multicore processing power, into one which can efficiently use multicore processing power available.

Fig- 3 and Fig- 4 will pictorially explain the idea of Data Parallelism.

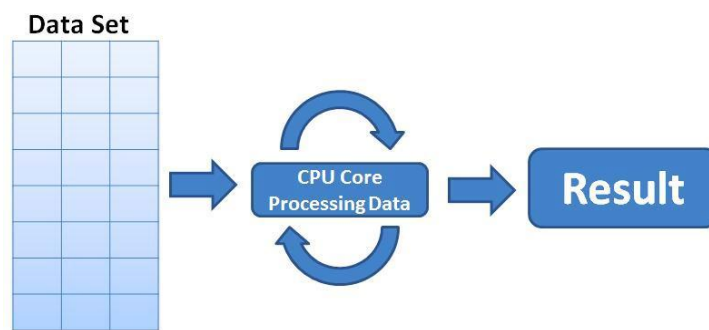


Fig-3: Single CPU Core Processing Huge Data Set

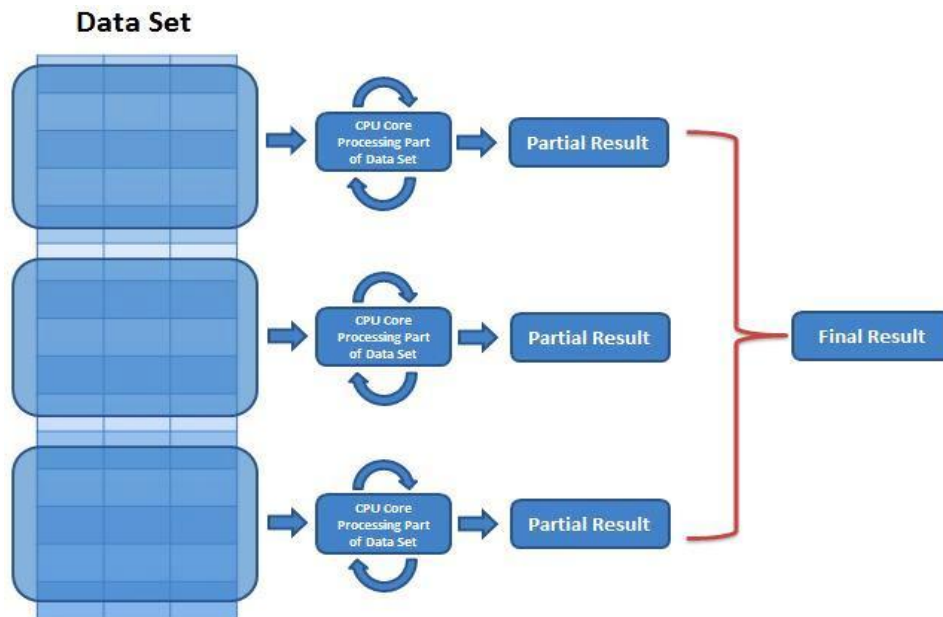


Fig-4: Multiple CPU Cores Processing Subset of Huge Data Set

TPL exposes data parallelism through the “*System.Threading.Tasks.Parallel*” class. This class has parallel implementations of “*foreach ()*” and “*for ()*” loops. The TPL handles all the low-level work.

The following example shows a simple matrix multiplication code and its parallel equivalent:

```

//Simple Matrix Multiplication
for (int i = 0; i < matARows; i++)
{
    for (int j = 0; j < matBCols; j++)
    {
        for (int k = 0; k < matACols; k++)
        {
            result[i, j] += matA[i, k] * matB[k, j];
        }
    }
}

// A Parallel Equivalent of Matrix Multiplication.
//Parallelizing The Outer Loop
Parallel.For(0, matARows, i =>
{
    for (int j = 0; j < matBCols; j++)
    {
        double temp = 0;
        for (int k = 0; k < matACols; k++)
        {
            temp += matA[i, k] * matB[k, j];
        }
        result[i, j] = temp;
    }
}); //Close Parallel.For

```

Both “*Parallel.ForEach()*” and “*Parallel.For()*” methods return a “*ParallelLoopResult*” object which exposes the “*IsCompleted*” and “*LowestBreakIteration*” properties. These properties will help the developer to know whether the loop ran to completion or not. They also tell the developer about the iteration in which the loop was broken.

4.3 Parallelizing Outer Loops VS Parallelizing Inner Loops

“*Parallel.For()*” and “*Parallel.ForEach()*” usually perform well on outer loops rather than inner loops. This is because by parallelizing outer loops, we are offering larger chunks of work to parallelize, thus decreasing the Task management overhead.

For example consider the code shown below:

```
//Parallelize The Inner Loop.
for (int i = 0; i < matARows; i++)
{
    Parallel.For(0, matBCols, j =>
    {
        double temp = 0;
        for (int k = 0; k < matACols; k++)
        {
            temp += matA[i, k] * matB[k, j];
        }
        result[i, j] = temp;
    });
}
```

In the above code we have parallelized the inner loop and the matrix multiplication of two 1000 x 1000 matrices on a dual core machine took 22.032 Seconds to complete. The execution time may vary depending on the processor clock speed, number of cores available, the system load at that time and several other parameters.

Now consider this code shown below:

```
//Parallelize The Outer Loop.
Parallel.For(0, matARows, i =>
{
    for (int j = 0; j < matBCols; j++)
    {
        double tmp = 0;
        for (int k = 0; k < matACols; k++)
        {
            tmp += matA[i, k] * matB[k, j];
        }
        result[i, j] = tmp;
    }
}); //Close Parallel.For
```

In the above code we have parallelized the outer loop and the matrix multiplication of two 1000 x 1000 matrices on a dual core machine took 20.249 Seconds to complete. The execution time can differ depending on the number of cores available, the system load at that time and several other parameters.

There may be some special cases where parallelizing inner loops will give better performance than parallelizing outer loops, but in general it is observed that parallelizing outer loops gives more performance than parallelizing inner loops, which is clearly evident from the above code examples.

5. Parallel LINQ

Parallel LINQ (PLINQ) is the concurrent query execution engine for LINQ. PLINQ exposes data parallelism with the help of queries. PLINQ helps in parallelizing the execution of LINQ queries on objects and XML data. Computations implemented as queries on objects can be parallelized using PLINQ. PLINQ works on those objects which implement the “*IParallelEnumerable*” interface. Internally PLINQ uses TPL for execution. To use PLINQ, we need to invoke the “*AsParallel()*” extension method on the input sequence of a normal LINQ query.

The following example shows a simple usage of PLINQ:

```
var numberSource = Enumerable.Range(1, 10000);
var evenNumbers = from number in numberSource.AsParallel()
                 where CheckEven(number) == 0
                 select number;
```

In the above code snippet we are first creating an enumerable range of numbers from 1 to 10000 and then writing a PLINQ query to select all even numbers from the enumerable range, making full use of all cores on the target machine. The “*CheckEven(number)*” is an user defined function that returns 0 if the number is even and returns 1 if it is odd.

Note that we have used “*AsParallel()*” in the above PLINQ query. “*AsParallel*” is an extension method in the “*System.Linq.ParallelEnumerable*” class. The “*ParallelEnumerable*” wraps the input into a sequence based on “*ParallelQuery<TSource>*”. This wrapping of the normal sequence into a sequence of “*ParallelQuery<TSource>*” type will cause the LINQ query operators that we subsequently call to get bound to alternate extension methods defined in “*ParallelEnumerable*” class. The extension methods inside the “*ParallelEnumerable*” class have parallel implementations of each of the standard query operators. These parallel implementations work by partitioning the given input sequence into chunks which can execute on different threads and then collates the results into single output sequence for final consumption. Similarly there are some more operators in the “*ParallelEnumerable*” class which facilitate in writing PLINQ queries.

Some important operators are mentioned below:

| ParallelEnumerable Operator | Description |
|----------------------------------|--|
| AsParallel() | This is the starting point for PLINQ. This indicates that the rest of the query must be parallelized, if it is possible. |
| AsSequential() | This specifies that the rest of the query must be run as a sequential LINQ query. |
| WithDegreeOfParallelism() | This specifies the maximum number of processors a PLINQ query can use for parallelization. |

Table-1: IParallelEnumerable Operators

PLINQ queries can scale up the degree of concurrency depending on the capabilities of the host computer. In many scenarios, PLINQ can significantly improve the speed of LINQ to Objects queries by optimally utilizing the available CPU cores on the host computer. PLINQ queries can sometimes operate sequentially if PLINQ suspects that the overhead of parallelization may slow down a particular query.

Some of the query operators cannot be parallelized efficiently. For those query operators that cannot be parallelized, PLINQ implements the operator sequentially. We can override this behavior of PLINQ and force parallelism by calling the “*WithExecutionMode*” extension method like this:

“*AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism)*”.

We recommend using the feature of forcing parallelism judiciously only when the expected performance is not seen in the query execution. The hidden hazard of forcing parallelism is that we are indirectly limiting the portability of our application. When we force parallelism in the query, we may see immediate benefits on multi-core machines but our application may perform very badly on a machine not equipped with multi-core processor because, the query in which we forced parallelism can take more time to execute on a single core processor because of forced parallelism.

PLINQ is only for local collections and it doesn't work with LINQ to SQL or with the Entity Framework because in these cases the LINQ queries get translated into SQL queries which will be executed on a database server. However, PLINQ can be used on the result sets obtained from database queries for performing additional local querying.

5.1 Side Effects of PLINQ

One of the side effects of parallelizing the query operators is that the results may not get collated in the order as they were submitted. In other words, LINQ's normal order preservation is not guaranteed. However, we can force order preservation by calling `AsOrdered()` after `AsParallel()`:

Example:

```
inputCollection.AsParallel().AsOrdered()
```

When we have large number of elements, calling `AsOrdered` incurs a performance hit because; now PLINQ has to keep track of each element's original position. The effect of `AsOrdered` can be negated later in a query by calling `AsUnordered` which introduces a random shuffle point which allows the query to execute efficiently from that point on. So if we wanted to preserve input ordering of sequence only for the first two query operators, we can do this:

```
sourceSequence.AsParallel().AsOrdered()  
.Operator1()  
.Operator2()  
.AsUnordered() //Ordering Of Sequence Not Preserved From Here On  
.Operator3()  
...
```

`AsOrdered` is not applied by default because for most of the queries, the original input ordering does not matter. In other words, if `AsOrdered` was the default, we would have to apply the `AsUnordered` extension method to most of our parallel queries, which is burdensome.

5.2 Limitations of PLINQ

Currently there are some limitations on what PLINQ can parallelize. Microsoft may overcome these limitations in future with new service packs and Framework versions.

Unless the source elements are in their original indexing position, the following operators prevent a PLINQ query from being parallelized:

- Take, Skip, TakeWhile, SkipWhile
- ElementAt, Indexed versions of Select and SelectMany

Most of the query operators change the indexing position of elements. If we want to use the above mentioned operators in a PLINQ query, we must use these query operators before using the other operators which are parallelizable irrespective of the original indexing position.

The following query operators are parallelizable, but the partitioning strategy that they use can sometimes be expensive and it can be slower than sequential processing:

- Join, GroupBy, GroupJoin, Distinct, Union, Intersect, and Except

5.3 When to Use PLINQ?

Many of the programmers start tweaking with the existing LINQ queries and convert them to PLINQ queries. This approach may help in some cases but not all. Most of the problems for which LINQ is the obvious and the best solution will not get any benefit from parallelization. A better approach to find the best use of PLINQ is to find a CPU intensive work and then try to express that work as a PLINQ query.

PLINQ is well suited to embarrassingly parallel problems. Block and wait tasks like calling multiple web services at once can perform well with PLINQ.

Let's consider a small problem: suppose we want sum of the square roots of numbers from 1 through 10,000,000. We can easily parallelize the calculation of 10 million square roots, but summing up all the square root values is painful because we must put a lock around the variable which holds the total before updating it. The usual way of doing this is shown below:

```
class TotalSum
{
    public double total = 0;
}
static void Main(string[] args)
{
    TotalSum sum = new TotalSum();
    Parallel.For(1, 10000000, i => { lock (sum) sum.total += Math.Sqrt(i); });
}
```

In the above example the performance gain that we obtain from parallelization is very less when compared to the cost of acquiring 10 million locks plus the resultant blocking.

Often PLINQ is a good fit for these scenarios. The above example can be parallelized with PLINQ simply like this:

```
ParallelEnumerable.Range(1, 10000000).Sum(i => Math.Sqrt(i));
```

6. Exception Handling

The “Parallel” class, “PLINQ”, and “Tasks” automatically marshal exceptions to the consumer. To see why this is essential, consider the following LINQ query, which throws a “DivideByZeroException” on the first iteration:

```
try
{
    var source = from number in Enumerable.Range (0, 1000000) select 100 / number;
    source.Count(); //Get The Count Of Elements
}
catch (DivideByZeroException)
{
    ...
}
```

If we write a PLINQ query to parallelize the above LINQ query and ignore the exceptions, a “*DivideByZeroException*” would be thrown on a separate thread, bypassing our catch block and causing the application to terminate.

To handle this situation, exceptions are caught automatically and rethrown to the caller. Because these libraries leverage many threads, it is quite possible that two or more exceptions to be thrown simultaneously. To ensure that all exceptions are reported, exceptions are wrapped in an “*AggregateException*” container, which exposes an “*InnerExceptions*” property containing all the exceptions caught.

The following code explains exception handling:

```
try
{
    var source = from number in Enumerable.Range(0, 1000000) select 100 / number;
    source.Count(); //Get The Count Of Elements
}
catch (AggregateException aggException)
{
    foreach (Exception exception in aggException.InnerExceptions)
        Console.WriteLine(exception.Message);
}
```

“*PLINQ*” and the “*Parallel*” class will end the query or loop execution upon encountering the first exception and will stop the processing of any further elements or loop bodies. But it is possible that more exceptions might be thrown before the current cycle of all the threads which are running is completed. The first exception in “*AggregateException*” is visible in the “*InnerException*” property.

Similarly when we wait for the completion of a “*Task*” either by calling its “*Wait*” method or by accessing the *Task*’s “*Result*” property, any unhandled exceptions will be wrapped in an “*AggregateException*” object and will be re-thrown to the caller. The exception re-throwing to the caller avoids the need of writing code within task blocks for handling the unexpected exceptions.

The code written below shows one way of handling exceptions in *Tasks*:

```
int x = 0;
//We Are Not Handle The Exception Here
Task<int> calculation = Task.Factory.StartNew(() => 7 / x);
try
{
    Console.WriteLine(calculation.Result);
}
catch (AggregateException aggException)
{
    //Attempted A Divide By Zero
    Console.Write(aggException.InnerException.Message);
}
```

In the above example notice that the exception will arise when the division is performed but it is caught and handled after the task completion.

For the Tasks which have Child Tasks, waiting on the parent causes an implicit wait on the children and any exceptions that are raised in the child tasks will bubble up.

The following code shows this phenomenon:

```
TaskCreationOptions attachToParent = TaskCreationOptions.AttachedToParent;
var parentTask = Task.Factory.StartNew(() =>
{
    Task.Factory.StartNew(() => //Child
    {
        //Grandchild
        Task.Factory.StartNew(() => { throw null; }, attachToParent);
    }, attachToParent);
});
//This Call Will Throw A NullReferenceException Wrapped In //AggregateException
parentTask.Wait();
```

Unhandled exceptions on Tasks will not terminate the application immediately instead the termination is delayed until the finalize method of the Task is not called by the garbage collector. Termination of the application is delayed because it is uncertain until the “Task” is garbage collected whether the developer will call the “Wait” method or we will check the “Result” or “Exception” property. This delay can mislead developers from the root cause of the error.

Every time after the execution of a Task if we check the Task’s Exception property, the virtue of reading that property will prevent the exception from terminating our application. The reason is that .NET 4 Parallel Extensions designers did not want the developers to ignore exceptions. As long as the developers check the exceptions, they will not get punished with program terminations.

7. Data Structures for Parallel Programming

One reason why people didn’t write lot of parallel code is that it brings with it problems like Race conditions, Dead Locks, etc. which are difficult to track and fix. To tackle these problems we need special data structures which are thread-safe. The .NET Framework 4 introduces several new types which are helpful for parallel programming. These new types include concurrent collection classes, lightweight synchronization primitives, and a few types for lazy initialization. We can use these types with managed multithreaded code as well as the Task Parallel Library and PLINQ.

Data Structures for Parallel Programming:

- Concurrent Collection Classes
- Synchronization Primitives
- Lazy Initialization Classes

7.1 Concurrent Collection Classes

Unlike the collections introduced of the .NET Framework 1.0 and 2.0, the new concurrent collection classes introduced in .NET 4 which can be found in the “System.Collections.Concurrent” namespace requires no explicit locks in the user code to accesses the items of these concurrent collections. Thread-safe “add” and “remove” operations are provided in these new concurrent collections that can avoid locks wherever possible and can acquire fine-grained locks where locks are necessary.

Using the new concurrent collections can significantly improve the performance in scenarios where multiple threads try to add and remove items from a collection.

Some important Concurrent Collection Classes are listed below:

- System.Collections.Concurrent.BlockingCollection(T)
- System.Collections.Concurrent.ConcurrentBag(T)
- System.Collections.Concurrent.ConcurrentDictionary(TKey, TValue)
- System.Collections.Concurrent.ConcurrentQueue(T)
- System.Collections.Concurrent.ConcurrentStack(T)

7.2 Synchronization Primitives

For improving the performance of multithreaded applications by enabling fine-grained concurrency and by avoiding expensive locking mechanisms, Microsoft .NET Framework 4 comes with new synchronization primitives in the “System.Threading” namespace. “System.Threading.CountdownEvent” and “System.Threading.Barrier” are new types having no equivalents in earlier versions of the .NET Framework.

“System.Threading.Barrier” class provides various methods which allow developers to bring all the parallel tasks to a synchronization point. In earlier versions of .NET Framework, developers had to write special code to do the barrier synchronization. Using .NET Framework 4, developers can efficiently and easily write synchronization code with the help of “Barrier” class methods.

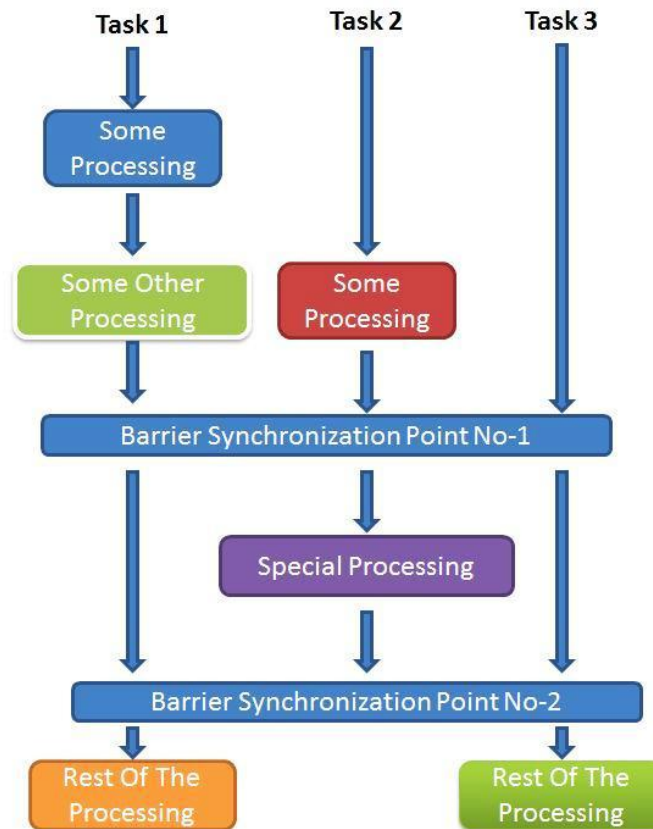


Fig-5: Pictorial Representation of Barrier Synchronization

With the help of “Barrier” class methods, a group of tasks cooperatively move through a series of phases shown as barrier synchronization points in the above figure. In each phase a task signals its arrival at the Barrier which causes it to implicitly wait for all others to arrive to that Barrier. The same instance of the “Barrier” class can be used for multiple phases.

“*System.Threading.CountdownEvent*” is designed for scenarios in which we would want to block a particular Task till the occurrence of an event on some or all of the remaining tasks running concurrently. Before the .NET Framework 4 was released, developers had to use a “*ManualResetEventSlim*” or “*ManualResetEvent*” and had to explicitly decrement a count variable before signaling the event. The “*CountdownEvent*” makes things easier for developers by letting them initialize the “*CountdownEvent*” with an initial signal count and subsequently call its “*Wait()*” method. The Task on which the “*Wait()*” is called will be blocked until the signal count becomes zero. The “*CountdownEvent*” class provides a “*Signal()*” method which decrements the signal count by 1 when called. After the “*CountdownEvent*” has been signaled a certain number of times, the Task which called the “*Wait()*” method will be unblocked.

The figure given below will pictorially show the functionality of “*CountdownEvent*”:

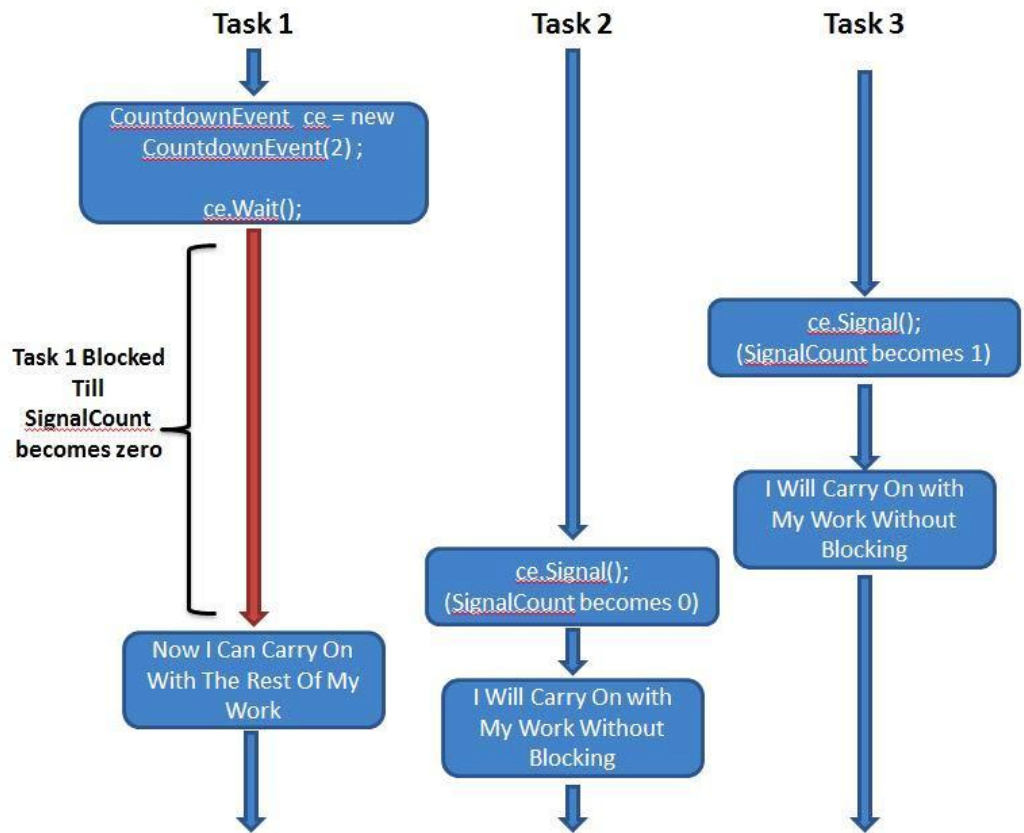


Fig-6: Pictorial Representation of CountdownEvent

Some of the other important Synchronization Primitives are listed below:

- `System.Threading.ManualResetEventSlim`
- `System.Threading.SemaphoreSlim`
- `System.Threading.SpinLock`
- `System.Threading.SpinWait`

7.3 Lazy Initialization Classes

With lazy initialization, the memory required for an object is allocated only when it is needed. By spreading object allocations evenly across the entire lifetime of a program, lazy initialization can drastically improve performance of the application. We can enable lazy initialization for any custom type by wrapping the type Lazy(T).

Some important Lazy Initialization Classes are listed below:

- System.Lazy(T)
- System.Threading.ThreadLocal(T)
- System.Threading.LazyInitializer

8. Performance Analysis & Debugging Tools

Visual Studio 2010 has several new enhanced tools to help developers in debugging the parallel applications as well as analyze their performance.

8.1 Debugging

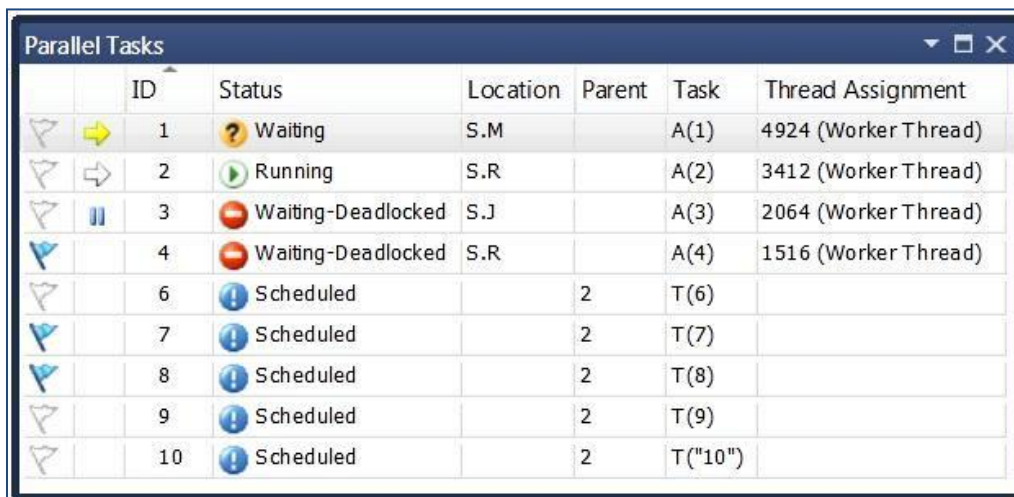
For debugging the code written using TPL it did not make sense to force a developer to think in terms of threads so Visual Studio 2010 introduced new debugging tools which can show debugging information in terms of “Tasks” but not in terms of “Threads”. Two new tool windows are added to Visual Studio 2010 that makes the “Task” a first-class citizen in debugging. The new debugger tool windows support both managed and native task models along with the traditional threading programming models.

The two new debugger tool windows added are:

- Parallel Tasks
- Parallel Stacks

8.1.1 Parallel Tasks

The Parallel Tasks tool window displays the tasks created by the application and shows whether they are running, or waiting on a resource, or scheduled for running. It can even show additional information like information on the thread that is executing a task, parent/child relationships between tasks, and a task’s call stack. This view helps the developer in understanding the current execution patterns and it also gives information on the system load conditions.



| ID | Status | Location | Parent | Task | Thread Assignment |
|----|--------------------|----------|--------|---------|----------------------|
| 1 | Waiting | S.M | | A(1) | 4924 (Worker Thread) |
| 2 | Running | S.R | | A(2) | 3412 (Worker Thread) |
| 3 | Waiting-Deadlocked | S.J | | A(3) | 2064 (Worker Thread) |
| 4 | Waiting-Deadlocked | S.R | | A(4) | 1516 (Worker Thread) |
| 6 | Scheduled | | 2 | T(6) | |
| 7 | Scheduled | | 2 | T(7) | |
| 8 | Scheduled | | 2 | T(8) | |
| 9 | Scheduled | | 2 | T(9) | |
| 10 | Scheduled | | 2 | T("10") | |

Fig-7: Sample Parallel Tasks Window

8.1.2 Parallel Stacks

With the Parallel Stacks tool window, a developer can visualize the call stacks of the different Tasks of the application. This graphical view is very similar to the call stack window but it is enhanced by expanding the focus from one execution context to multiple execution contexts and by visually indicating the sharing of different methods among different tasks.

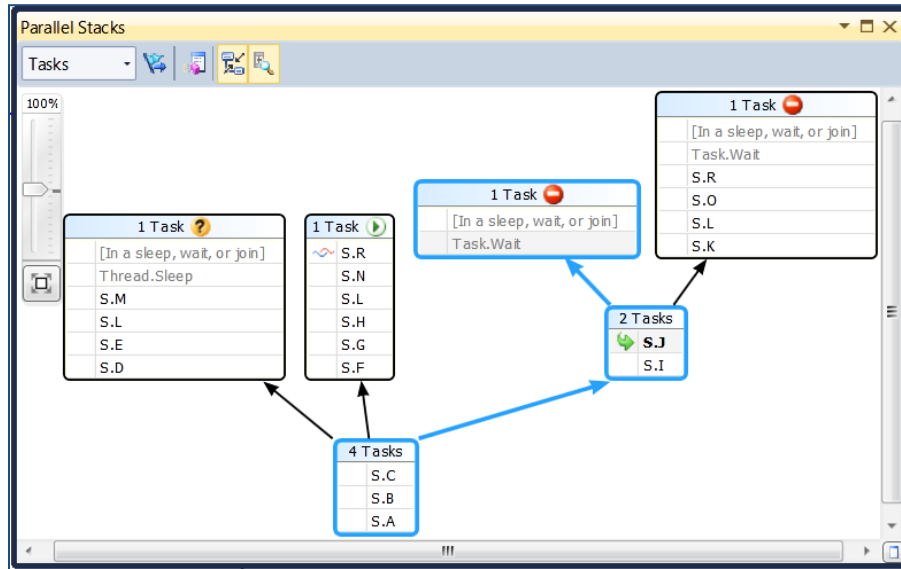


Fig-8: Sample Parallel Stacks Window

Note: The profiling features discussed from here onwards require kernel support; these features are available on Windows Vista and other higher versions of Windows only.

8.2 Profiling

To help the developers in analyzing parallel applications, Visual Studio 2010 has new profiling tools which can measure the degree of parallelism within the application, discover contention for resources, visualize the distribution of threads across cores, know the amount of time spent within the application for executing program code, time spent for waiting on synchronization objects, time spent on performing I/O, and more. Microsoft Visual Studio 2010 includes three new profiler views for concurrency:

- CPU utilization view
- Cores view
- Threads view

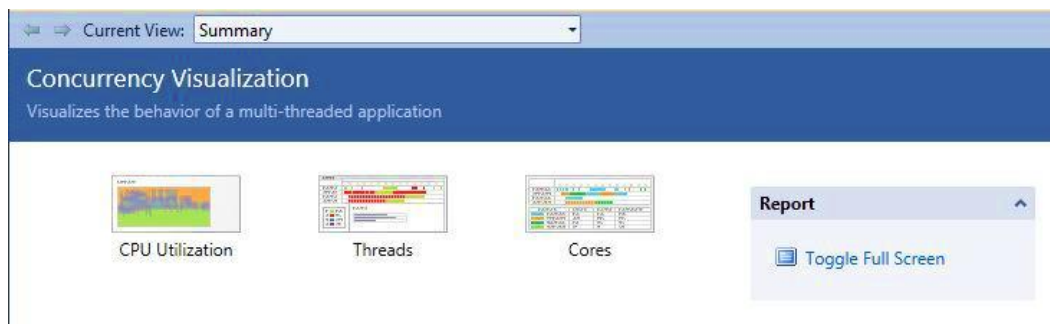


Fig-9: Concurrency Visualization Modes

8.2.1 CPU Utilization View

With the help of CPU utilization view developers can determine the phases of the application which are CPU-bound and those that are not CPU-bound. It can even help in finding the duration of execution of different portions of the application code. It also helps in visualizing the utilization of the CPU Cores available on the machine. Using the CPU Utilization view we can zoom in on a particular phase of execution to examine the parallelism that exists in that phase.



Fig-10: Sample CPU Utilization View

8.2.2 Cores View

The purpose of the cores view is to show the mapping of the Tasks to different CPU cores. This information will help us visualize the execution of application over time and detect thread migrations. The act of moving a thread from one CPU core to another is called Thread migration. Thread migration is a resource intensive operation that can degrade overall performance of the application. By examining the thread state across the entire duration of the trace, developers can spot thread migration and map it back to specific delays using the Threads view.

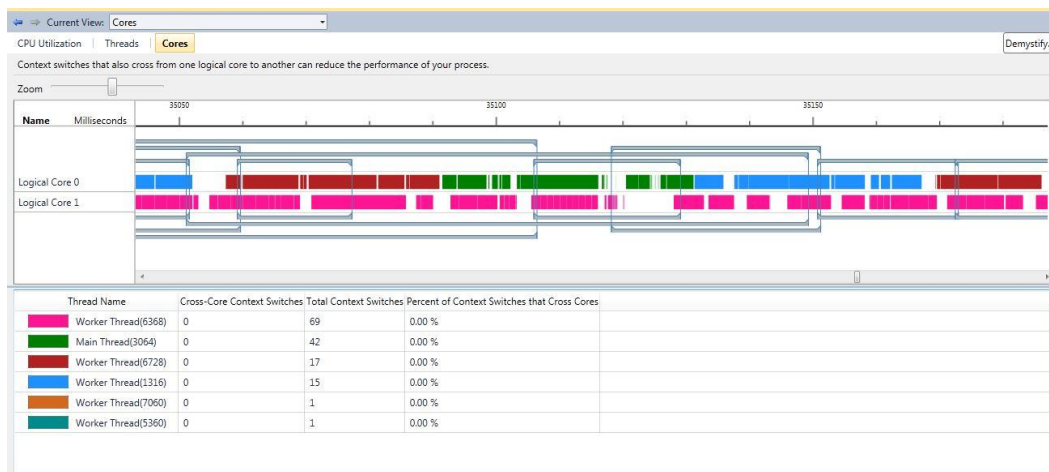


Fig-11: Sample Cores View

8.2.3 Threads View

After finding a region of interest using either the CPU utilization view or the Cores view, a developer can further scrutinize the behavior of an application using the Threads view. This view gives the developer a wealth of information on the behavior of various threads in an application.

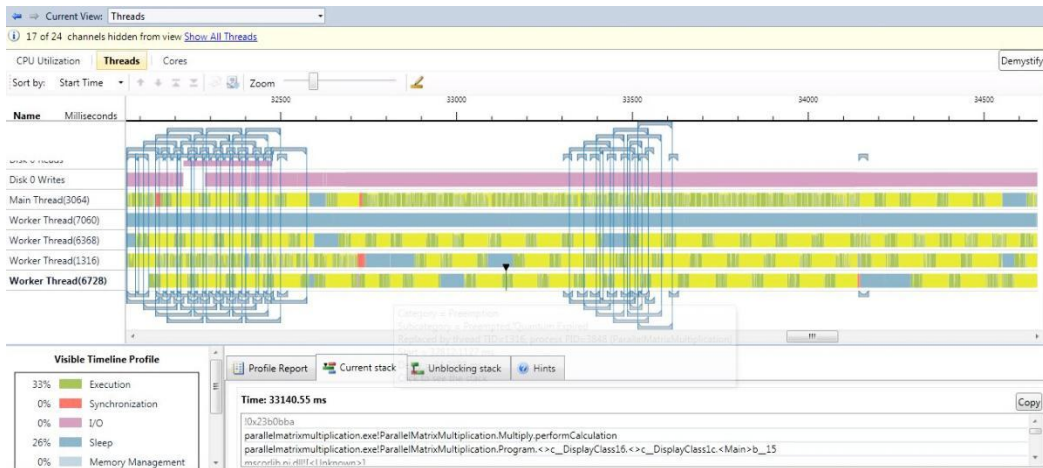


Fig-12: Sample Threads View

9. Scheduled Funds Transfer Sample Implementation

9.1 Purpose

- To showcase improvement in performance of applications using .Net 4 Parallel Extensions.
- To showcase the capabilities of the parallel programming API's provided in .NET 4
- To showcase how multithreaded applications were programmed earlier and how easily we can program them now.

9.2 Scenario

9.2.1 Problem Statement

- Increasing the performance of Batch Execution of Scheduled Funds Transfer Requests.

9.2.2 Proposed Solution

- Using the Parallel Programming API's available in .NET 4; execute the Fund Transfer requests in parallel.

9.3 Description of Application

The “Scheduled Funds Transfer” Application was implemented to demonstrate the power of .NET 4 parallel programming API's. The business case that we have taken for building this application is a common bank operation called the “Funds Transfer” which allows an account holder to transfer funds into another account. Using this application we simulated millions of customers trying to make a Funds Transfer request and a back office bank application which executes all the millions of Funds Transfer requests which are received.

A screen-shot of the application is shown below:

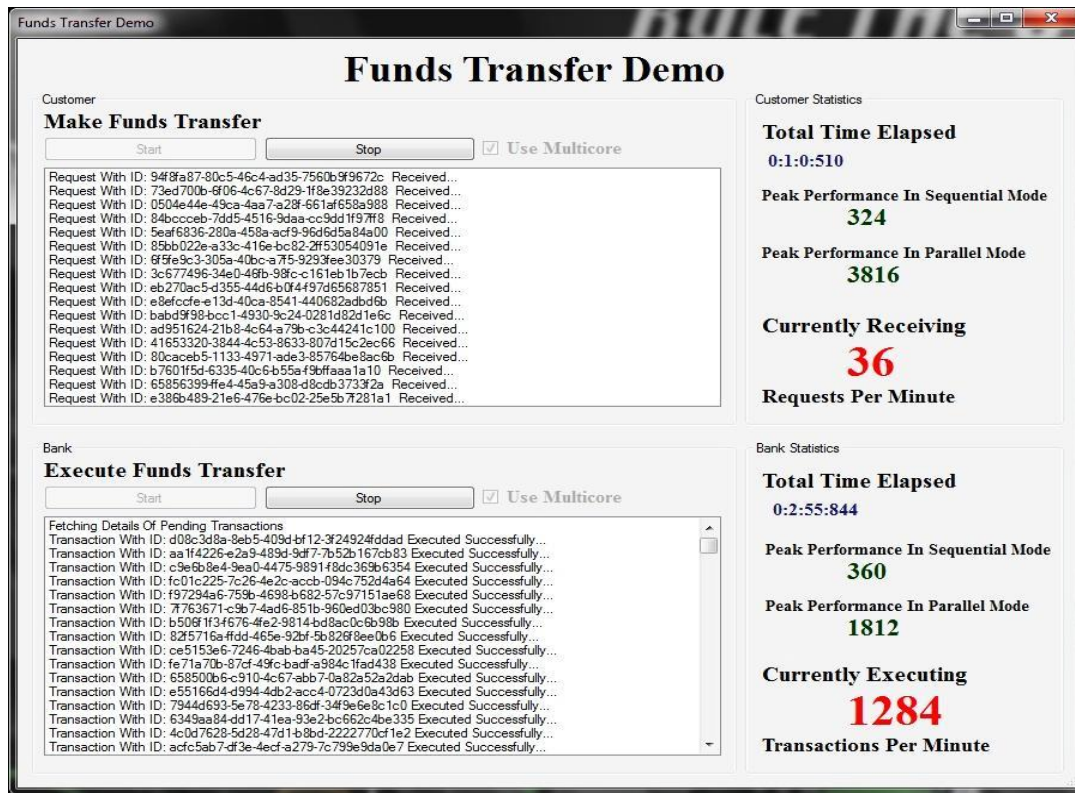


Fig-13: Screenshot of the application

In the screen-shot we can see that the application has two modules. The first module is the “Customer” module and the one which is below the Customer module is the “Bank” module. Both the modules are designed to run in “Sequential Mode” as well as “Parallel Mode”. To run the module in sequential mode (Normal sequential code) we must leave the “Use Multicore” check box of that module “Unchecked” whereas to run the module in “Parallel Mode” (Using .Net 4 Parallel Programming API’s) we must “Check” it.

Beside each module there is a “Statistics” section. These statistics sections are used in tracking the performance of the module adjacent to which they are placed. The statistics section is updated for every 5 seconds. Different parameters taken into consideration are:

Total Time Elapsed: This will show the total time for which the particular module was running.

Peak Performance in Sequential Mode: Shows the “Peak Performance Number” in sequential mode.

Peak Performance in Parallel Mode: Shows the “Peak Performance Number” in parallel Mode.

Peak Performance Number: In case of “Customer Module” the peak performance number is the highest number of “Funds Transfer Requests” that the application was able to receive in a time span of 1 minute.

In case of “Bank Module” the peak performance number is the highest number of “Funds Transfer” requests executed in a span of 1 minute.

A brief description of the two modules is given below:

Customer Module:

This module is designed to simulate millions of customers making “Funds Transfer” request. If we run this module in sequential mode, it can only receive one request at a time. But if we run this module in parallel mode, it can receive multiple requests simultaneously.

Bank Module:

This module is designed to simulate a back office bank application which executes the millions of “Funds Transfer” requests that it has received. If we run this module in sequential mode, it will fetch all the pending transactions from the database and then it starts executing the “Funds Transfer Requests” one after the other. But if we run this module in parallel mode, it will fetch all the pending transactions and then it will try to execute as many transactions as possible in parallel. The number of transactions that can be executed in parallel is decided by the TPL.

9.4 Benchmarking Environment

| | |
|------------------------|---|
| Processor | AMD Athlon™ II X2 245 Processor 2.90 GHz(Dual Core Processor) |
| Installed Memory (RAM) | 4 GB (3.75 GB Usable) |
| System Type | 64-bit Operating System |
| Operating System | Windows 7 Enterprise |

Table-2: Details of Machine Chosen for Benchmarking

9.5 Code Snippets

9.5.1: Code snippet showing Simulation of 1 Million customers

```
//Simulating 1 Million Customers
Parallel.For(0, 100000, (i, state) =>
{
    //Generating New Transaction
    businessLogicLayer.InsertScheduledPayment()
    Interlocked.Increment(ref transactionCount);
});
```

From the code you can see that we have used a “Parallel For” loop to simulate 1 million customers. The “`businessLogicLayer.InsertScheduledPayment()`” is equivalent to a single “Funds Transfer Request” made. You can also see how we have sampled the number of requests for every 5 seconds.

9.5.2: Code Showing Sequential Execution of Funds Transfer Requests

```
//Executing Transactions Sequentially
for (int i = 0; i < listOfTransactions.Count(); i++)
{
    businessLogicLayer.ExecuteScheduledPayment(listOfTransactions.ElementAt(i));
    transactionCount++;
}
```

```
}
```

In the above code snippet all the pending transactions are executed one after the other.

`businessLogicLayer.ExecuteScheduledPayment(listOfTransactions.ElementAt(i))` is equivalent of executing a single “Funds Transfer” request.

9.5.3: Code showing Parallel Execution of Funds Transfer Requests

```
//Executing Transactions Parallely
Parallel.For(0, listOfTransactions.Count(), (i, state) =>
{
    businessLogicLayer.ExecuteScheduledPayment(listOfTransactions.ElementAt(i));
    Interlocked.Increment(ref transactionCount);
});
```

Notice that we have not changed much code compared to sequential execution. We have just used a “Parallel For” loop instead of sequential loop.

9.6 Performance Numbers

| Action | Sequential Execution | Parallel Execution (Using .NET 4 Parallel Programming API's) | % Improvement |
|--|-----------------------------|---|-------------------------------|
| Time taken for Executing 1 Million Transactions | 2358 Minutes (1.64 Days) | 501 Minutes (0.35 Days) | 78.75% (reduction in time) |
| Average No. of Transactions Executed Per Minute | 451 | 2810 | 523.06% |
| Maximum No. of Transactions Executed Per Minute | 468 | 3396 | 625.64% |
| Average CPU Utilization | 12% | 35% | 191.66% |

Table-3: Performance Numbers

These performance numbers were collected by load testing the application by simulating 1 Million customers in the Customer Module, each one making one scheduled funds transfer request. The Bank Module executed all the 1 Million funds transfer requests. The above table shows the performance numbers of the Bank Module.

10. Conclusion

Enterprises are investing in high performance servers with multicore processors in order to meet the performance demands of the dynamic business environment. Parallel Extensions in .NET Framework 4 will help us develop applications that meet these performance demands. .NET architects and developers will be able to easily design and develop programs for the multicore processors using the parallel extensions.

This paper introduced the readers to Parallel Extensions in .NET 4 Framework. Paper also described the PLINQ and its suitability. Exception handling in Parallel extension needs a careful attention. Sample implementation scenarios with TPL features got a remarkable performance gain over serial code.

The TPL is the preferred way of writing parallel code. The wealth of explicit Task handling features like “*Task.WaitAny()*”, “*Task.WaitAll()*” will help the developers in fine tuning their applications. Not all code is suitable for parallelization; many a times the overhead of parallelization can slowdown the application. Careful evaluation of the scenario should be done before using TPL.

This paper will help the .NET community to appreciate and understand the Parallel Extensions in the .NET Framework and it will help the developers in building applications which can target multicore processors using Parallel Extensions.

References

1. [Lambda Expressions in PLINQ and TPL.](http://msdn.microsoft.com/en-us/library/dd460699(VS.100).aspx)
[http://msdn.microsoft.com/en-us/library/dd460699\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd460699(VS.100).aspx)
2. [Lambda Expressions.](http://msdn.microsoft.com/en-us/library/bb531253(VS.100).aspx)
[http://msdn.microsoft.com/en-us/library/bb531253\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/bb531253(VS.100).aspx)
3. [Comparative Analysis of Hill Climbing Mapping Algorithms.](http://repository.upenn.edu/cis_reports/653/)
http://repository.upenn.edu/cis_reports/653/
4. [Scheduling Multithreaded Computations by Work Stealing.](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.139.2618)
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.139.2618>
5. [Building a custom thread pool: A work stealing queue.](http://www.bluebytesoftware.com/blog/2008/08/12/BuildingACustomThreadPoolSeriesPart2AWorkStealingQueue.aspx)
<http://www.bluebytesoftware.com/blog/2008/08/12/BuildingACustomThreadPoolSeriesPart2AWorkStealingQueue.aspx>
6. [Data Structures for Parallel Programming.](http://msdn.microsoft.com/en-us/library/dd460718(VS.100).aspx)
[http://msdn.microsoft.com/en-us/library/dd460718\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd460718(VS.100).aspx)
7. [Introduction to PLINQ.](http://msdn.microsoft.com/en-us/library/dd997425(VS.100).aspx)
[http://msdn.microsoft.com/en-us/library/dd997425\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd997425(VS.100).aspx)
8. "Parallelism in .NET" Series by Reed Copsey, Jr.
<http://blogs.msdn.com/pfxteam/archive/2010/02/10/9961019.aspx>

9. [Introduction to Task Parallel Library.](#)
[http://msdn.microsoft.com/en-us/library/dd537609\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd537609(VS.100).aspx)
10. [Parallel Programming in the .Net Framework.](#)
[http://msdn.microsoft.com/en-us/library/dd460693\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd460693(VS.100).aspx)
11. [VS2010 Parallel Computing Features Tour.](#)
<http://channel9.msdn.com/posts/DanielMoth/VS2010-Parallel-Computing-Features-Tour/>
12. [Concurrency vs. Parallelism - What is the difference?](#)
<http://stackoverflow.com/questions/1050222/concurrency-vs-parallelism-what-is-the-difference>
13. [Debugging features available in VS2010 for concurrent programs.](#)
<http://channel9.msdn.com/posts/DanielMoth/VS2010-Parallel-Computing-Features-Tour/>

About the Author

[Humayun Khan Pathan](#) is a Systems Engineer in the High performance computing (HPC) group with Infosys Microsoft Technology Center (MTC).

Acknowledgements

I profusely thank **S. Naveen Kumar** (Principal Architect, MTC) and Sudhanshu M. Hate (Senior Technical Architect, MTC) for their support, help and valuable guidance in bringing out the paper in the current form.

I would like to thank **Atul Gupta** (Principal Architect, MTC) and Sripriya Thothadri (Technical Architect, MTC) for helping me at various stages during the review process.



Infosys among the world's top 50 most respected companies

Reputation Institute's Global Reputation Pulse 2009 ranked Infosys among the world's top 50 most respected companies.



About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit www.infosys.com.

Global presence

Americas

Brazil
Nova Lima
Canada
Calgary
Montreal
Toronto
Mexico
Monterrey
United States
Atlanta
Bellevue
Bentonville
Bridgewater
Charlotte
Detroit
Fremont
Hartford
Houston
Lake Forest
Lisle
New York
Phoenix
Plano
Quincy
Reston

Asia Pacific

Australia
Brisbane
Melbourne
Perth
Sydney
China
Shanghai
Hangzhou
Hong Kong
Central
India
Bangalore
Bhubaneshwar
Chandigarh
Chennai
Gurgaon
Hyderabad
Jaipur
Mangalore
Mumbai
Mysore
New Delhi
Pune
Thiruvananthapuram
Japan
Tokyo
New Zealand
Wellington
Philippines
Metro Manila
Singapore
Singapore
London

Europe

Belgium
Brussels
Czech Republic
Brno
Prague
Denmark
Copenhagen
Finland
Helsinki
France
Paris
Germany
Frankfurt
Stuttgart
Walldorf
Ireland
Dublin
Italy
Milano
Norway
Oslo
Poland
Lodz
Spain
Madrid
Sweden
Stockholm
Switzerland
Geneva
Zurich
The Netherlands
Amsterdam
United Kingdom (UK)
London

Middle East and Africa

Kingdom of Saudi Arabia
Riyadh
Mauritius
Reduit
UAE
Dubai
Sharjah

For more information, contact askus@infosys.com

www.infosys.com

© 2011 Infosys Technologies Limited, Bangalore, India. Infosys believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. Infosys acknowledges the proprietary rights of the trademarks and product names of other companies mentioned in this document.