*International Journal on Emerging Technologies* (Special Issue NCETST-2017) **8**(1): 406-415(2017)
*(Published by Research Trend, Website:* www.researchtrend.net*)*

**ISSN No. (Print) : 0975-8364**

**ISSN No. (Online) : 2249-3255**

# Parallel Programming in the .NET Framework by using wrapper classes for a COM object

***Mohit Giri Goswami, Prashant Rajput and Upasana Arya***
*Assistant Professor, Amrapali Institute of Technology and Sciences Haldwani (U.K.), INDIA*

**ABSTRACT: I have to admit that I'm not an expert in multithreading or parallel computing. However, people often ask me about easy introductions and beginner's samples for new features. And I have an enormous advantage over most newbie's in this area. I have a simple goal this time. I want to parallelize a long-running console application and add a responsive WPF UI. By the way, I'm not going to concentrate too much on measuring performance. I'll try to show the most common caveats, but in most cases just seeing that the application runs faster is good enough for me. Many personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously. Computers in the near future are expected to have significantly more cores. To take advantage of the hardware of today and tomorrow, you can parallelize your code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio 2010 and the .NET Framework 4 enhance support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools. These features simplify parallel development so that you can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. The following illustration provides a high-level overview of the parallel programming architecture in the .NET Framework 4.**

## I. INTRODUCTION

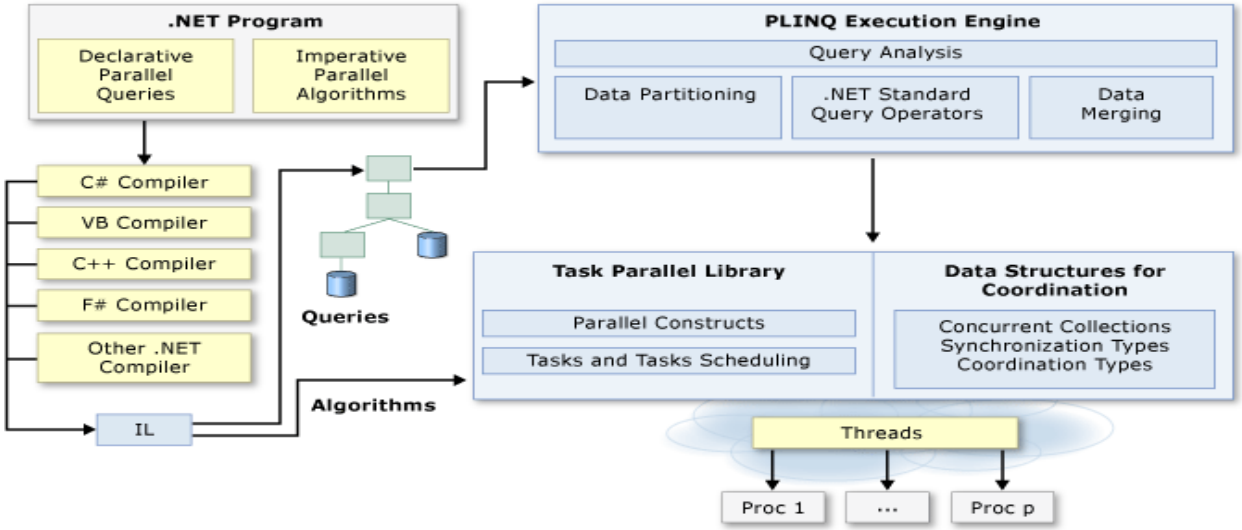
"With great power comes great responsibility"... As the hardware world is booming and it's processing power increases, the responsibility lies with developers to utilize such power. OK, so the comparison is lame, but I am a fan of the Spider Man movie and was always looking for a way to use its slogan... guess I should keep looking :)

Seriously though, multi-core (and multi-processer) machines are dominant these days. Soon enough, as reported by some lead vendors, you will not be able to find machines with less than 8 cores. As such, the traditional way of writing serial programs will not be able to make use of such power, by default. These programs should be written in a way which utilizes the power of multi-cores. Multi-threaded programming is the way which allows developers to take advantage of multiple threads concurrently. And while it has been available for many years, multi-threaded programming has often been avoided due to its complexity. Developers had to put most of their effort in writing clean multi-threaded code rather than focusing on the business problem. What does multithreading have to do with multi-cores? Can't we write multi-threaded code if we have a single core? Yes, we can. But again, as long as we have a single core, only one thread at a time will work since all threads belong to that same single core... logical enough. When your multi-threaded program runs on a multi-core machine, threads belonging to different cores will be able to run together and hence your program will take advantage of the hardware power. One final note before going on: multithreaded programming does not differ if you have a multi-core machine or multi-processer machine. This is a hardware separation; underneath, your code will be working with threads all the same.

**What's Useful in the FCL:**

Version 1.0 of the .NET Framework Class Library provides extensive functionality for the following areas:

- New data types, such as GUID for working with Global Unique Identifiers, and TimeSpan, for working with a relative times (5 minutes instead of 12:05 AM 1/1/2005).




- Environment information—OS version, environment variables, command line arguments, machine name, user name, user domain, special folder paths, logical drives.
- Functionality for creating console (command line) applications.

- Functionality for writing Windows services.
- Registry access.
- Power events.
- Database access.
- Numerous types of collections, including sorted lists, dictionaries, and queues.
- Configuration file management.
- Operating system interaction—Processes, event logs, performance counters, services, tracing, and debugging.
- Active Directory access.
- Advanced graphics capabilities—Drawing, gradients, images, printing.
- Access to COM+ services such as distributed transactions.
- File system interaction and IO.
- Windows Management Interface access.
- Access to message queuing.
- Low level network access—DNS lookups, IP Addresses, sockets, HTTP communication.
- Interoperability with non .NET code (known as unmanaged code).
- Reflection—Allows late binding, and inspection of types at runtime.
- Remote object access—Creating instances of remote classes and calling their methods.
- Serialization—The ability to easily import/export the data of a class as XML.
- Cryptography—Hashing, symmetric and asymmetric encryption, X509 certificates.
- Text formats—Conversions for ASCII, UTF-7, UTF-8, Unicode.
- Regular Expressions—The ability to insure that a string conforms to a certain pattern (the string looks like a phone number, IP address, e-mail address, and so on) The ability to extract sub-strings from a string (for example, extract all the HTML <A> tags).
- Threading—The ability to have an application do work in the background.
- Web Services.
- XML support—Superior functionality compared to MSXML.

Version 2.0 adds the following new functionality:

- Programmatic access to access control lists (ACLs).
- Data Protection API (DPAPI).
- New networking classes (Ping, network connectivity changed notification).
- Better programmatic support for certificates.
- FTP classes.
- Compression and decompression classes.
- Access to more local computer information.
- Serial port classes.
- SMTP classes.
- New windows forms controls—**SoundPlayer**, **BackgroundWorker**.
- Enhancements to many existing windows forms controls.

This list isn't exhaustive, but it should give you an idea that the FCL is extensive, and contains functionality that wasn't available in Visual Basic 6. Some of this functionality is available directly, and some is accessible through the creation of simple wrappers (literally, only a couple lines of code). In this article, you'll see the functionality that can be used directly. In following articles, you'll learn how to create wrappers to access additional FCL functionality.

Parallel vs. Concurrent:

Well, both concepts are related but not the same: in its simplest definition, Concurrency is about executing multiple un-related tasks in a concurrent mode. What these un-related tasks share are system resources, but they do not share a common problem to solve; meaning, they are logically independent. Now, think of these tasks as threads. And, since these threads share system resources, you face the globally common problems of concurrent programming such as deadlocks and data races. This makes concurrent programming extremely difficult and debugging complicated.

What is Parallelism? Parallelism is taking a certain task and dividing it into a set of related tasks to be executed concurrently. Sweet! So again, thinking of these tasks as threads, Parallel Programming still has the same problems of concurrent programming (deadlocks, data races, etc...), and introduces new challenges, most notably, sharing and partitioning data across these related threads. This makes Parallel Programming even more difficult and debugging even more complicated :(

However, it is not all bad news. .NET 4.0 parallel programming is a great step onwards. The new API solves a lot of problems (but not all) of Parallel Programming, and greatly eases up parallel debugging..

.NET 4.0 Parallel Programming

The Parallel Programming model of .NET 4.0 is composed of the following:

- The Task Parallel Library (TPL): this is the base of the Task-based programming discussed in the previous section. It consists of:
  - Task class: the unit of work you will code against instead of the previous thread model.
  - Parallel class: a static class that exposes a task-based version of some parallel-nature problems. More specifically, it contains the following methods:
    - For
    - Foreach
    - Invoke
- Parallel LINQ (PLINQ): built on top of the TPL and exposes the familiar LINQ as Parallel extensions.

Parallel programming for specific Tasks

The first example shows the new Tasks programming model. As explained previously, Tasks - as opposed to the Thread Pool - grants more control over thread programming. In fact, you now deal with Tasks and not threads.

```
static void Main(string[] args)
{
    Task t = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("I am the first task");
    });

    var t2 = t.ContinueWith(delegate
    {
        //simulate compute intensive
        Thread.Sleep(5000);
        return "Tasks Example";
    });

    //block1
    //string result = t2.Result;
    //Console.WriteLine("result of second task is: " + result);
    //end block1

    //block2
    //t2.ContinueWith(delegate
    //    {
    //        Console.WriteLine("Here i am");
    //    });
    //Console.WriteLine("Waiting my task");
    Console.ReadLine();
```

In the above code, we first create a new Task using the Task class, and we pass to the lambda expression the code to run inside
Parallel

```
static void Main(string[] args)
{
    Stopwatch watch;
    watch = new Stopwatch();
    watch.Start();

    //serial implementation
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(1000);
        //Do stuff
    }
    watch.Stop();
    Console.WriteLine("Serial Time: " + watch.Elapsed.Seconds.ToString());

    //parallel implementation
    watch = new Stopwatch();
    watch.Start();
```

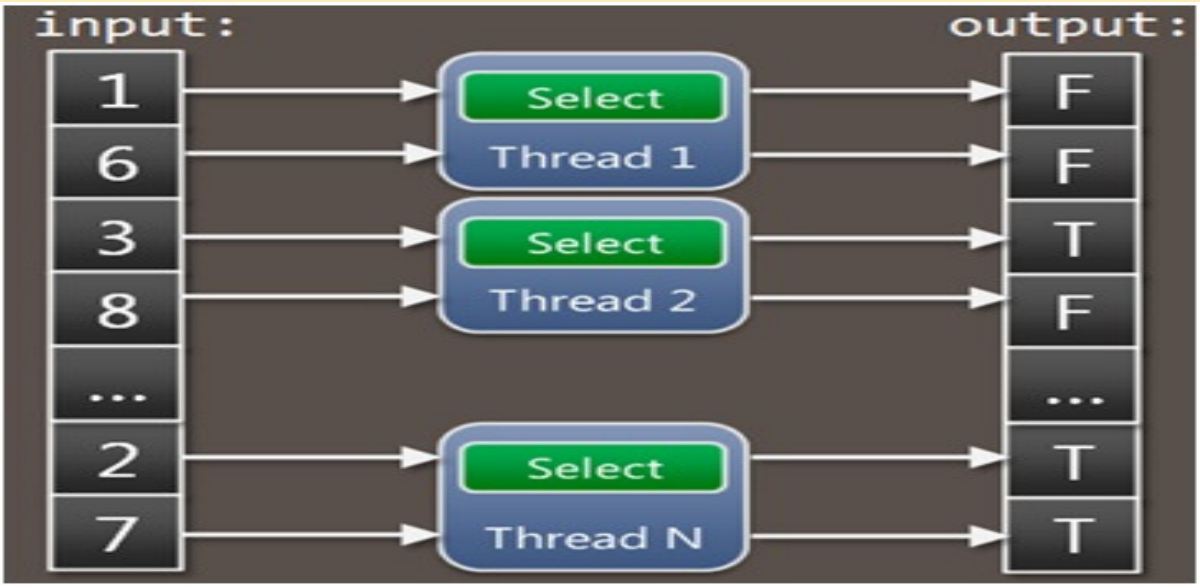
```
    System.Threading.Tasks.Parallel.For(0, 10, i =>
    {
        Thread.Sleep(1000);
        //Do stuff with i
    }
    );
    watch.Stop();
    Console.WriteLine("Parallel Time: " + watch.Elapsed.Seconds.ToString());

    Console.ReadLine();
}
Parallel.For(0,1000,(int i, ParallelLoopState loopState) =>
{
    If(i==500)
      loopState.Break();
      //or loopState.Stop();
     return; // to ensure that this iteration also returns
     //do stuff
});
```

The difference between the Break and Stop methods is that Break ensures that all earlier iterations that started in the loop are executed before exiting the loop.The Stop method makes no such guarantees; it basically says that this loop is done and should exit as soon as possible.Both methods will stop future iterations from running.

Parallel programming for PLINQ
PLINQ is a LINQ Provider, so you still use the familiar LINQ model. In the underlying model, however, PLINQ uses multiple threads to evaluate queries.

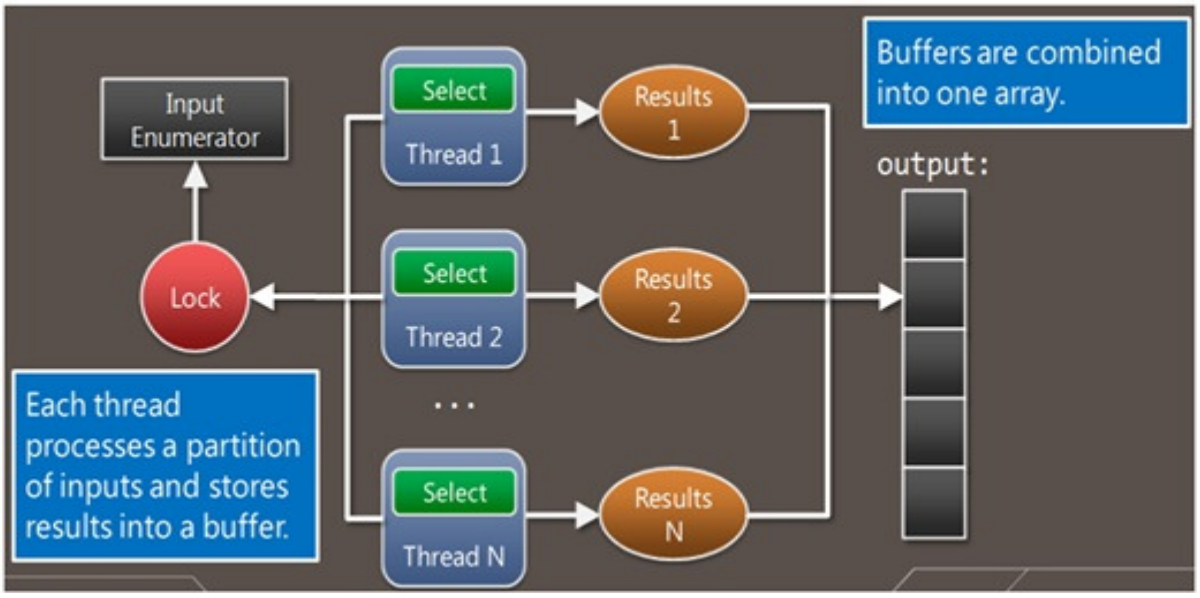
```
static void Main(string[] args)
{
    Stopwatch watch;

    //for loop
    watch = new Stopwatch();
    watch.Start();
    bool[] results = new bool[arr.Length];
    for (int i = 0; i < arr.Length; i++)
    {
        results[i] = IsPrime(arr[i]);
    }
    watch.Stop();
    Console.WriteLine("For Loop took: " + watch.Elapsed.Seconds);

    //LINQ to Objects
    watch = new Stopwatch();
    watch.Start();
    bool[] results1 = arr.Select(x => IsPrime(x))
        .ToArray();
    watch.Stop();
    Console.WriteLine("LINQ took: " + watch.Elapsed.Seconds);
```

```
    //PLINQ
    watch = new Stopwatch();
    watch.Start();
    bool[] results2 = arr.AsParallel().Select(x => IsPrime(x))
        .ToArray();
    watch.Stop();
    Console.WriteLine("PLINQ took: " + watch.Elapsed.Seconds);

    Console.ReadLine();
}
```




Now, edit the code and use the AsOrdered extension as follows:

```
var query = src.AsParallel().AsOrdered()
                    .Select(x => ExpensiveFunc(x));
```




What number do you expect to see if you run the program on a multi-core machine? 100000 is the logical result. Now, run the program and you will notice that the number is less than that (if you do get 100000, you are just being lucky, so try again). So what happened here? Why did we get this wrong when it seemed to be so obvious? The answer is data race occurred.

Let's explain this. The trick here is in understanding how the statement counter++ gets executed. When at Intermediate Language (IL) level, counter++ is not a single statement anymore. Actually, it gets executed in 4 IL steps by the JIT (Just In-time Compiler):

1. Get the current value of the counter
2. Get the value of 1
3. Add the two numbers together
4. Save the result back to the counter

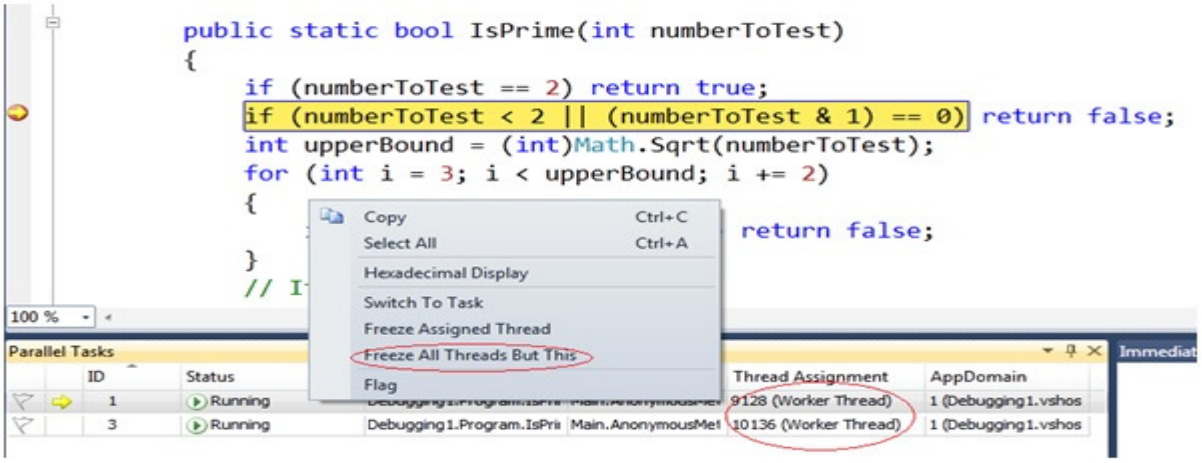
| Thread | Action |
|---|---|
| T1 | Get value of counter (gets 2) |
| T1 | Get value of 1 |
| T2 | Get value of counter (gets 2) |
| T2 | Get value of 1 |
| T1 | Add the two numbers together (2+1) |
| T1 | Save the result back into counter (saves 3) |
| T2 | Add the two numbers together (2+1) |
| T2 | Save the result back into counter (saves 3) |

As a result, one step increment was overlooked due to the fact that two threads were granted access to the variable counter at the same time. The above scenario occurs many times, and you will get the final value wrong, as we did when we ran the program. This is a classical case of Data Race.

Parallel programming for Debugging

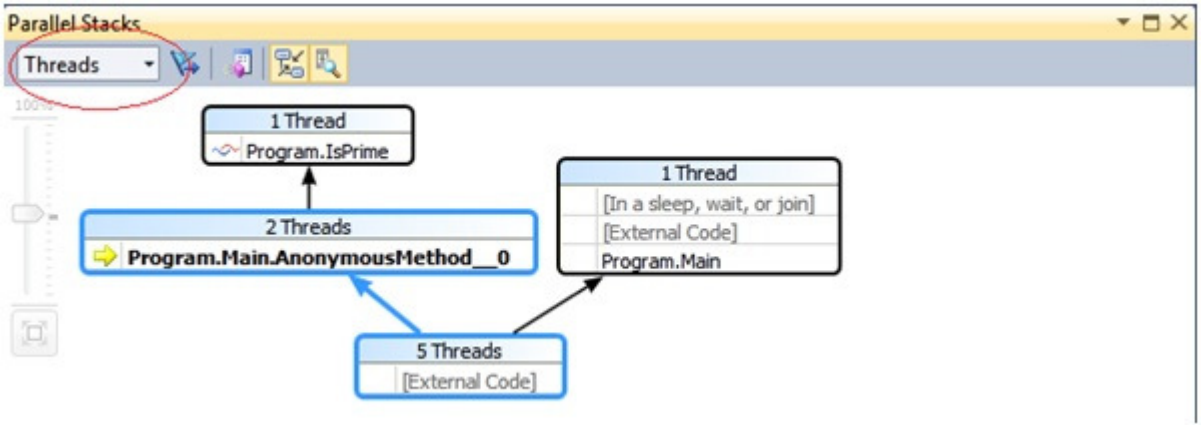
```
static void Main(string[] args)
{
   var primes =
      from n in Enumerable.Range(1, 10000000)
      .AsParallel()
      .AsOrdered()
      .WithMergeOptions(ParallelMergeOptions.NotBuffered)
      where IsPrime(n)
      select n;
   foreach (var prime in primes)
      Console.Write(prime + ", ");

}

public static bool IsPrime(int numberToTest)
{
   if (numberToTest == 2) return true;
   if (numberToTest < 2 || (numberToTest & 1) == 0) return false;
   int upperBound = (int)Math.Sqrt(numberToTest);
   for (int i = 3; i < upperBound; i += 2)
   {
      if ((numberToTest % i) == 0) return false;
   }
   // It's prime!
   return true;
}
```

The above code uses PLINQ to print all prime numbers of a given range.




Through this window, you can see all the running threads in Debug mode. You can select to freeze all threads except the one that you want to actually debug, and once done, you can run the other threads.

Another useful window is the Parallel Stacks window (Debug menu -> Windows -> Parallel Stacks):



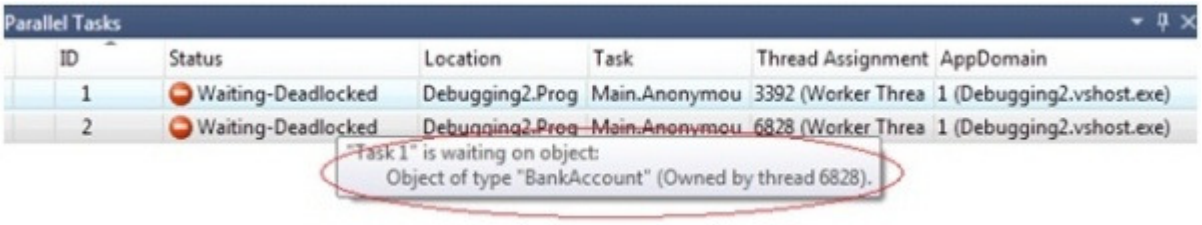

Through this window, you can see a graphical representation of the running threads of your program and how they originated.

Parallel programming for Deadlocks

Example 5 explained Data Races and how to deal with them through locks. However, you should take care that locking can bite you in the ugly form of Deadlocks. Simply put, a deadlock is a case where a thread TA is locking resource R1 and a thread TB is locking resource R2, and now, TA needs R2 to continue while TB needs R1 to continue. Both threads are stuck and cannot continue, thus causing a deadlock.

```
static void Main(string[] args)
{
   int transfersCompleted = 0;
   WatchDeadlock.BreakIfRepeats(() => transfersCompleted, 500);
   BankAccount a = new BankAccount { Balance = 1000 };
   BankAccount b = new BankAccount { Balance = 1000 };
   while (true)
   {
      Parallel.Invoke(
      () => Transfer(a, b, 100),
      () => Transfer(b, a, 100));
      transfersCompleted += 2;
   }

}

class BankAccount { public int Balance; }
static void Transfer(BankAccount one, BankAccount two, int amount)
{
   lock (one)
   {
      lock (two)
      {
         one.Balance -= amount;
         two.Balance += amount;
      }
   }
}
```

The above code uses Parallel.Invoke to concurrently perform money transfer between bank accounts "a" to "b" and vice




So how do you avoid deadlocks? The answer is: simply do not do them. When doing parallel programming and locks, always program in a way that deadlocks will not occur!

Result:

A Wrapper class is one that "encapsulates" a resource, or another class in order to simplify or restrict the interface between the outside world and the encapsulated object. For example, a wrapper class might encapsulate a COM object, providing .NET access methods and properties and translating them internally to the function calls to the actual COM interface. Or it might "hide" a full Dictionary object, allowing you to just add, query and remove items from it, without knowing what the underlying storage is.

If you want to compare it to the real world, the "drive a car" class wraps the complexity of the engine by providing an Accelerator object with "press" and "release" methods that hide the complexity of the fuel injection mapping from the user.

**REFERENCES**

[1]. http://www.codeguru.com/csharp/article.php/c18689/NET-Framework-Parallel-Programming-Design-Patterns.htm

[2]. https://blogs.msdn.microsoft.com/csharpfaq/2010/06/01/parallel-programming-in-net-framework-4-getting-started/

[3]. http://www.codeguru.com/columns/experts/article.php/c17197/Understanding-Tasks-in-NET-Framework-40-Task-Parallel-Library.htm

[4]. http://www.diva-portal.org/smash/get/diva2:652176/FULLTEXT01.pdf

[5]. http://www.codeguru.com/columns/experts/article.php/c17197/Understanding-Tasks-in-NET-Framework-40-Task-Parallel-Library.htm

[6]. https://books.google.co.in/books?id=IgBx6RRO0WcC&pg=PA36&lpg=PA36&dq=advantage+of+wrapper+class+in+.net+framework&source=bl&ots=XhR4SnlbCa&sig=Tzu_nZxdkiMiSWG7GRsOJVo3vu8&hl=en&sa=X&ved=0ahUKEwjuobeN6pvTAhVLwI8KHdGFBN8Q6AEITzAH#v=onepage&q=advantage%20of%20wrapper%20class%20in%20.net%20framework&f=false