Parallel Processing Capabilities of C#

Peter Gissel
Computer Science Department
San Jose State University
530-277-3156

ABSTRACT

This document is a study in the parallel processing capabilities of C#. There are a variety of different methodologies to approach parallelism in the C# language. This is intended to give a brief overview of the various approaches and how the language has evolved to handle the problem more and more effectively over the years. The language has evolved significantly, and indeed, the latest evolution was mainly to introduce the addition of two very powerful key words to facilitate parallel programming.

1. INTRODUCTION

C# is Microsoft's flagship language among their .Net language offerings. It is an industrial strength language and offered as an alternative to Java and C++, although it is a strictly Windows based language. There are implementations of C# for Linux and Unix but they are incomplete and not as well performing as Microsoft's native implementation on Windows. Microsoft has added advanced capability for parallel programming in the more recent versions of C# that allow the developer to take advantage of multi-core and parallel processing computing platforms in a far easier manner than in previous versions. The new features have the advantage of requiring less code and allow the developer to program at a higher level than the original thread based approach. Their most recent asynchronous additions to the language allow programmers to take advantage of parallel and asynchronous programming and retain a simpler sequential programming approach in their implementation of parallel programming.

2. HISTORY

Microsoft originally developed their own Java Virtual Machine (JVM) a couple of decades ago. It was by far the fastest implementation of Java on Windows at the time. Microsoft had developed their own version of the Java language for use with their JVM called J++. It implemented the Java standard mostly, but left out some key API's and added other Windows-only custom extensions. Sun Microsystems, the creator of Java 1.1 was not happy about Microsoft's divergent Java. In 1997 they sued Microsoft for incompletely implementing the Java 1.1 standard. In 2001 Sun and Microsoft settled the lawsuit. Microsoft paid Sun $20 million and the two agreed to a plan for Microsoft to phase out products that included the older version of Microsoft Java that allegedly infringed on Sun's Java copyrights and trademarks. As a part of their agreement Microsoft discontinued development of their JVM and J++. [2]

Microsoft still had a vision of a better programming language built directly on Windows that would enable easier development of high performing applications. The result of this was the creation of C# which was initially released with Visual Studio 2002. It was a very similar language to Java and initially criticized for being a Java knock-off. With each successive release of C#, however, the two have diverged more and more. Over the years the parallel capabilities have increased and the latest release of C# 5.0 has some new unique features built into the language that make writing parallel processing applications much easier and straight forward.

3. PARALLEL CAPABILITIES

3.1 First Generation – C# 1.0

The power of C# is based on Microsoft's .Net libraries and Common Language Runtime (CLR), which is common to other Microsoft languages as well, such as Visual Basic and F#. The .Net libraries for their initial release included the most common parallel capabilities at the time which were based on a set of threading classes and libraries included in .Net. In addition there were also a couple of features built into the language that simplified controlling resource contention. These were the Lock() statement and the [Synchronized] attribute. The paradigms were similar to the current version of Java and other languages at the time giving the developer the ability to create and manage threads, control resource access with Monitors and Semaphores, and communicate between threads with classes such the ManualResetEvent and AutoResetEvent classes. The libraries offered a host of classes to facilitate parallel programming including mutexes, semaphores, different kinds of timers, and thread pool management. It also included the AppDomain class which allowed a developer to create completely isolated independent processes within an application.

3.1.1 Threads

In its most simplistic form creating a parallel processing application requires the creation of a thread for each task to be run in parallel. A method is defined to be called by the thread and passed to the thread object as a parameter. The thread will continue executing until it exits the method. It is common for the method to contain a loop that will continue to loop until some application condition is met and exits the loop and then exits the method terminating the thread.

The .Net Common Language Runtime (CLR) will handle the distribution of the threads to the different logical cores so this process is transparent to the developer. It can be demonstrated that two threads running the same method on multiple cores will execute in approximately the same time as a single threaded application running the same method a single time.

This first example does exactly that. It runs a method directly and displays the time it took to execute the method. Then it creates two

threads that each individually run the same method and displays the time it took for each thread to execute the method.

Here is the code for implementing this first example:

```
private void RunTwoThreads(object sender,
                                routedEventArgs e)
{
   m_threadOneTime.Content = "";
   m_threadTwoTime.Content = "";
   Thread threadOne = new Thread(RunTask);
   Thread threadTwo = new Thread(RunTask);
   threadOne.Start(m_threadOneTime);
   threadTwo.Start(m_threadTwoTime);
}

private void RunTask(object labelParam)
{
   var timer = new TimeToLabel(labelParam as Label);
   Tasks tasks = new Tasks();
   tasks.LongTask();
   timer.Complete();
}
```

When these are in a sample application the timing of the two different methods is very similar, demonstrating that the threads are running on separate cores with no resource contention. On the computers I tested on it was about five seconds.

The C# method called by the threads is similar to the example from CS 159 which shows the difference between column-major and row-major memory access. The second example calls a very similar method, but instead of each method having its own private instance of a matrix, the method operates on a shared matrix and demonstrates locking. The overhead of locking and the bottleneck effect contributes a significant amount of time for the thread execution time such that it is actually greater than two processes running sequentially and in fact usually runs from four to six times as long. It is a very good example of why critical sections should be minimized both in size and in number.

This code shows the long task implemented with locking:

```
public void LongTaskWithLocking()
{
  for (int i = 0; i < MAX_ITERATIONS; i++)
  {
     for (int j = 0; j < MAX_ITERATIONS; j++)
     {
        for (int k = 0; k < MAX_ITERATIONS; k++)
        {
           lock (m_data)
           {
              m_data[j, k] = i;
           }
        }
     }
  }
}
```

The task without locking is identical but without the lock() statement encapsulating the assignment statement.

3.1.2 ThreadPool

Another option available to the developer besides manipulating threads directly is using the thread pool. This object takes care of a lot of the programming overhead of managing threads directly and simplifies the implementation if more than a couple of threads are to be used. This is particularly applicable on a server application with multiple clients. The ThreadPool object itself takes care of creating and managing all the threads and also manages a queue of tasks to allocate to the pool. The maximum number of threads is set automatically depending on the system resources including the number of logical cores and the size of the virtual address space, but the developer can change the number to suite the application. Normally the ThreadPool will re-use threads to run large numbers of tasks more efficiently. It will also create and destroy threads to optimize throughput. Too few threads might not make optimal use of available resources and too many threads would increase resource contention. The developer simply creates tasks and queues them on the pool to take advantage of this built in optimization.

This ThreadPool example also uses the ManualResetEvent object so the application can determine when the threads are complete. By utilizing this object one per task and waiting for all tasks to complete it computes the amount of time it took for all the tasks to run on the thread pool.

This code shows the tasks run using a ThreadPool:

```
void RunTasksWithThreadPoo(int numThreads)
{
   ManualResetEvent[] doneEvents =
                     new ManualResetEvent[numThreads];
   for (int i = 0; i < numThreads; i++)
   {
      doneEvents[i] = new ManualResetEvent(false);
      var resetEvent = doneEvents[i];
      ThreadPool.QueueUserWorkItem((o) =>
                        RunTaskFromThreadPool(resetEvent));
   }
   WaitHandle.WaitAll(doneEvents);
}

private void RunTaskFromThreadPool(
                                ManualResetEvent resetEvent)
{
   ParallelProcessing.Tasks tasks =
                           new ParallelProcessing.Tasks();
   tasks.LongTask();
   resetEvent.Set();
}
```

There is a little extra overhead in this example in order to add the extra management aspect of waiting until threads are complete in order to time the process. Otherwise it could be even simpler by using a simple method as in the threading example rather than creating an entire class.

It shows that it takes about the same amount of time running as when creating the threads individually, but is simpler to code and manage.

3.2 Next Generation - C# 4.0

The release of C# and .Net 4.0 introduced a host of new features to further simplify parallel development and enable the developer to write more efficient and scalable code in a more natural way without have to resort to working directly with threads or thread

pools. It included the Task Parallel Library, Parallel LINQ (PLINQ), and new data structures for parallel programming. [4]

3.2.1 Task Parallel Library (TPL)

3.2.1.1 Data Parallelism

Data parallelism refers to scenarios in which the same operation is performed concurrently on individual elements of a source collection or array. This would be the same concept as the Data Partitioning discussed in CS 159. The TPL supports data parallelism by providing method based parallel implementations of *for* and *foreach* loops. It allows the developer to write loop logic much the same way as they would write a sequential loop but take advantage of parallel processing in a very simple way. It handles all the low level work so no threads or queue work is explicitly required. It still gives the developer a great deal of control if needed, you can stop or break loop execution, monitor the state of the loop on other threads, control the degree of concurrency, etc.

This next example uses a *foreach* loop and shows a process similar to the previous example using a thread pool.

```
private static void RunForEachInner(int numTasks)
{
   Tasks[] tasks = new Tasks[numTasks];
   Parallel.For(0, numTasks, i =>
   {
      tasks[i] = new Tasks();
   });
   Parallel.ForEach(tasks, task =>
   {
      task.LongTask();
   });
}
```

There are two loops, one to create the objects that run the task, and one to run the tasks. Each of these looks very similar to sequential code, but each loop is implemented in a parallel manner so the loop iterations are run in parallel instead of sequentially. This gives a tremendous speed increase if more cores are available with very little effort, and the resulting code is still very straight forward and easy to understand and maintain.

Although writing a simple parallel foreach loop is very straight forward and similar to a normal sequential loop, adding more sophisticated functionality such as a loop cancellation requires a little more work. The effect of breaking a loop is not quite the same. When you cancel a loop it will wait until all currently executing iterations are complete and then cancel execution. Only the remaining iterations that have not yet begun are actually cancelled.

Here is an example with the ability to cancel, it requires considerably more code:

```
private int RunForEachWithCancelInner()
{
   m_cts = new CancellationTokenSource();
   ParallelOptions po = new ParallelOptions();
   po.CancellationToken = m_cts.Token;
   po.MaxDegreeOfParallelism =
                  System.Environment.ProcessorCount;

   Tasks[] tasks = new Tasks[m_numTasks];
   Parallel.For(0, m_numTasks, i =>
   {
      tasks[i] = new Tasks();
   });
   int count = 0;
   try
   {
      Parallel.ForEach(tasks, po, task =>
      {
         count++;
         task.LongTask();
         try
         {
            po.CancellationToken.
                        ThrowIfCancellationRequested();
         }
         catch { }
      });
   }
   catch { }
   return count;
}

private void Cancel(object sender, RoutedEventArgs e)
{
   m_cts.Cancel();
}
```

This is still not overly cumbersome considering the speed increase possible, but still leaves room for plenty of improvement.

3.2.1.2 Task Parallelism

The Task Parallel Library (TPL) is based on the concept of the Task. Tasks represent asynchronous operations and are similar to threads in some ways but at a higher level of abstraction. Tasks provide two primary benefits:

1. More efficient and scalable use of system resources. Internally, tasks are implemented with threads and the ThreadPool which has been enhanced and optimized with a number of different algorithms to determine and adjust the number of threads that provide load balancing and maximum throughput. Tasks are relatively lightweight so you can create many of them to enable fine-grained parallelism.
2. More programmatic control than is possible with a thread or work item. Tasks and the entire framework supporting them provide a rich set of functionality that includes waiting, cancellation, continuations, exception handling, detailed status, custom scheduling, and a lot more.

Both of these reasons make it the preferred method of writing multi-threaded, asynchronous, and parallel code over using the Thread API directly.

To use the Task objects directly and run methods in parallel the most common and recommended method is to use the TaskFactory to create and handle new tasks.

This Task example is similar to the previous example using a ThreadPool, but the code is even simpler, and it takes only about half the work.

```
private static void RunTasksInner(int numTasks)
{
   Task[] taskArray = new Task[numTasks];
   for (int i = 0;i < numTasks; i++)
   {
      Tasks tasks = new Tasks();
      taskArray[i] = Task.Factory.StartNew(() =>
                                          tasks.LongTask());
   }
   Task.WaitAll(taskArray);
}
```

The performance of this code is very comparable to implementing it directly with the Thread object, so the overhead of using a higher level of abstraction to benefit from less code, and more maintainable and clearer code is inconsequential.

3.2.1.3 Parallel.Invoke

The TPL also has a Parallel.Invoke() method that allows the developer to execute any number of statements arbitrarily. All that is needed is to pass delegates into the Invoke method for each item to be run in parallel.

Here is a line of code that will execute two tasks in parallel:

```
Tasks tasks = new Tasks();
Parallel.Invoke(() => tasks.LongTask(),
                          () => tasks.AnotherLongTask());
```

It is a very simple way to run multiple tasks in parallel without incurring the overhead of complicated code.

3.2.1.4 Data Flow Library

The Data Flow Library provides data flow components to help increase the robustness of parallel processing applications. It provides actor-based programming providing in-process message passing. The library is based on the other previously mentioned components of the TPL. They are used when there are multiple operations that must communicate with each other asynchronously or when data needs to be processed as it comes available rather than all at once.

This library is divided into three different types of data flow blocks: source blocks, target blocks, and propagator blocks. Sources blocks are data sources and can be read from, target blocks are receivers of data and can be written to, and propagator blocks act as both and can be read from and written to. These blocks can be connected together to form pipelines. These pipelines propagate data through the blocks asynchronously as data becomes available.

They provide a common pattern and approach to data processing programming and provide powerful tools to develop certain types of parallel processing applications. It allows the developer to develop high throughput data processing applications by utilizing the parallel capabilities of this library that would take considerably more effort without it. This paper doesn't go into any detail on this part of the TPL as it would require too extensive of an example to demonstrate.

3.2.1.5 PLINQ – Parallel Language-Integrated Query

PLINQ is a parallel implementation of the LINQ pattern introduced in the previous version of C# 3.5. LINQ queries run against in-memory collections such as List<T> and other types of arrays and generic collections. Parallel LINQ is a parallel implementation of LINQ. PLINQ queries are very similar to LINQ, the main difference being that they attempt to make full use of all logical cores on the system. It does this by partitioning the data source into segments and then executing the query on each segment on separate worker threads in parallel on multiple processors.

For a simple LINQ query, the addition of one simple method call, AsParallel() can substantially improve the performance. As simple as PLINQ is, it is also very powerful and flexible with a multitude of options available to the developer.

PLINQ is implemented to be very conservative, parallel processing-wise. It will analyze the structure of the overall query at runtime. If it determines that parallelism is likely to speed up the query it will partition the data into tasks that can be run concurrently. If it finds it is not safe to parallelize a query it just runs it sequentially. The developer still has the control to ensure it uses parallel processing and the type of algorithm to use if testing determines an optimal method that may be faster than the defaults that PLINQ would choose. But the default implementation is very smart and very fast in most case.

By default PLINQ implementation will use up to 64 processors on a computer, but WithDegreeOfParallelism can be used to limit processors in case the developer wants to ensure some processors are left available for other tasks. This is just one example of how the developer can tailor the performance to suit a particular application.

This example show a very simple query that is sped up substantially by the use of PLINQ instead of just regular LINQ.

```
void RunPLinq(object sender, RoutedEventArgs e)
{
   var timer = new TimeToLabel(m_plinqTime);
   var source = Enumerable.Range(1, 200000000);
   var evenNums = from num in source.AsParallel() where
                                 Calc(num) % 2 == 0 select num;
   int count = evenNums.Count();
   timer.Complete();
}

private int Calc(int num)
{
   return (int)Math.Round(Math.Sqrt(num));
}
```

It generates a list of even square roots of a list of integers in substantially less time than if the AsParallel() feature were not used.

3.3 Latest Generation – C# 5.0

The latest version of C# introduced some really advanced ways of implementing asynchronous parallel processing code. The previous examples have all been very straight forward but asynchronous parallel programming can get very complex very quickly. This is especially true if you are returning data from parallel processes, triggering more asynchronous processes when some processes finish, have nested asynchronous methods, etc. The new *async* and a*wait* keyword additions make this kind of programming far simpler. To implement the same functionality that *async* and *await* give you using threads, it can take many times more code and far greater complexity than the simple straight forward approach that the new keywords combined with the Task library from TPL can achieve.

This example uses *async* and *await* in an implementation that is very similar to the standard Asynchronous Programming Model (APM) pattern. It only uses a fraction of the code however:

```
private async void RunAsyncAwait(object sender,
                                 RoutedEventArgs e)
{
   var timer = new TimeToLabel(m_asyncAwaitTime);

   var longAsyncTask = RunLongTaskAsync();
   Tasks tasks = new Tasks();
   tasks.LongTask();
   var data = await longAsyncTask;

   timer.Complete();
}

private async Task<int[,]> RunLongTaskAsync()
{
   int[,] data = new int[1000, 1000];
   Tasks tasks = new Tasks();
   await Task.Run(new Action(() =>
                       tasks.LongTask(data,1000, 1000)));
   return data;
}
```

The call to RunLongTaskAsync is run asynchronously while the method continues execution until it encounters the *await* keyword and so is really running the two tasks in parallel. You can see how this compares to code written implementing the APM pattern as in the example above but without the *async* and *await* keywords. The main calling method is very similar but the code to implement the async task has grown considerably.

To show the great advantages provided by these new keywords here is the classic Async code implemented using the standard Thread object and the APM pattern to show the difference in how much code is necessary:

```
private void RunAsyncAwaitThreaded(object sender,
                                   RoutedEventArgs e)
{
   var timer = new TimeToLabel(m_asyncAwaitTime2);

   var longTask = new LongTaskThread();
   var longAsyncTask = longTask.BeginLongTask();
   Tasks tasks = new Tasks();
   tasks.LongTask();
   var data = longTask.EndLongTask(longAsyncTask);

   timer.Complete();
}

public class AsyncResult : IAsyncResult
{
   public ManualResetEvent ResetEvent { get; set; }
   public WaitHandle AsyncWaitHandle
   {
      get { return ResetEvent; }
      set { }
   }
   public bool IsCompleted { get; set; }
   public bool CompletedSynchronously { get; set; }
   public object AsyncState { get; set; }

   public AsyncResult()
   {
      ResetEvent = new ManualResetEvent(false);
   }
}

public class LongTaskThread
{
   Thread m_thread;
   AsyncResult m_asyncResult = new AsyncResult();
   int[,] data = new int[1000, 1000];

   public IAsyncResult BeginLongTask()
   {
      m_thread = new Thread(RunThread);
      m_asyncResult = new AsyncResult();
      return m_asyncResult;
   }
   private void RunThread()
   {
      Tasks tasks = new Tasks();
      m_asyncResult.ResetEvent.Reset();
      tasks.LongTask(data, 1000, 1000);
      m_asyncResult.ResetEvent.Set();
   }
   public int[,] EndLongTask(IAsyncResult result)
   {
      result.AsyncWaitHandle.WaitOne();
      result.AsyncWaitHandle.Close();
      return data;
   }
}
```

It is clearly visible what a tremendous advantage using the new keywords can have in creating parallel processing code.

Microsoft also added many new asynchronous capabilities to the .Net framework that utilize this new methodology. These new methods are mainly concentrated in web communications, reading and writing to files, capturing media from devices, and communications using their Windows Communication Foundation (WCF).

4. Grid Computing

There are no grid computing capabilities built into C# or .Net, but the aforementioned tools give the capability to build grid computing frameworks. As a result, there are third party grid computing tools available to C# developers.

Digipede[6] is one company that offers a solution. This is a platform written in C# that enables developers to extend their own applications and add grid computing capabilities in-house that can speed up applications orders of magnitude if computing resources are available. The framework offers a very simple way to harness the power of grid computing using a very easy to implement framework based on object oriented and polymorphic principles.

The parallel processing capabilities of a grid computing framework offer tremendous advantages but still require a solid understanding of writing parallel processing applications in C# and all the risks and potential problems associated with it.

5. Cloud and Grid Computing – Azure

Windows Azure is a platform for building scalable, highly reliable, multi-tiered web service applications that offers both compute and data resources.

Cloud computing is where highly distributed computing resources are offered as a service. Applications can take as much or as little advantage as is applicable by utilizing these services. These services are offered through the internet or an intranet and the application communicates to this cloud remotely and does not need to know anything about the physical resources used to implement the cloud.

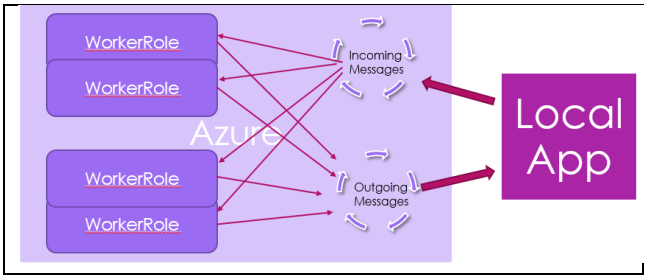
Grid computing is a where a large network of interconnected computers are connected together and designed to run applications or tasks in parallel.

With Azure you can combine the two concepts together and do both. Azure offers persistent message queues for passing messages to and from the cloud. It also offers the ability to define WorkerRoles that can send and receive messages from these queues and apply some processing to them. These WorkerRoles can be configured as processes that run within the Azure cloud.

The power of scalable parallel computing (grid computing) can easily be applied by configuring the number of WorkerRoles. Each of these worker roles are run as completely independent processes. This gives developers a very simple to way to harness the power of grid computing.

This example shows a very simple implementation of creating a WorkerRole, a couple of Service Bus Queues and sending and receiving messages. This example runs one long task for each message received. It shows that with one WorkerRole if multiple messages are sent, they are processed sequentially. If multiple WorkerRoles are created, it can be seen that they will process their tasks in parallel and return messages far quicker. This is easily done by changing the configuration of the service to define how many instances of the WorkerRole are created.

This is the archecture of the Azure example:



The local application sends messages to a service bus queue, a worker role then receives a message from the queue, does its processing, then sends a message to another service bus queue. The local application then receives this message from the queue.

To create a worker a subclass of the Windows Azuer API object RoleEntryPoint must be created. Minimally, the OnStart, OnStop, and Run virtual methods must be overridden.

Here is the startup and shut down code for the worker:

```
public override bool OnStart()
{
    string connectionString =
            CloudConfigurationManager.GetSetting(
                    "Microsoft.ServiceBus.ConnectionString");
    m_queue =
            QueueClient.CreateFromConnectionString(
                    connectionString, m_queueName);
    m_responseQueue =
            QueueClient.CreateFromConnectionString(
                    connectionString, m_responseQueueName);
    return base.OnStart();
}
public override void OnStop()
{
    m_queue.Close();
    m_responseQueue.Close();
    CompletedEvent.Set();
    base.OnStop();
}
```

It creates proxies of the queues to send and receive messages from, and shuts things down when done.

This is the WorkerRole code that actually runs on startup. It sets up the code that runs when a message is received:

```
public class WorkerRole : RoleEntryPoint
{

public override void Run()
{
    m_queue.OnMessage((receivedMessage) =>
      {
            Tasks tasks = new Tasks();
            tasks.LongTask();
            var message = new BrokeredMessage("parallel");
            m_responseQueue.Send(message);
      });
    CompletedEvent.WaitOne();
}
....
}
```

This is very similar to the long tasks run in the previous examples and in fact employs the very same code for running the long task.

Here is the code from the client application to send and receive messages to the WorkerRole through the message queues:

```
private void InitAzure()
{
    m_queue =
        QueueClient.CreateFromConnectionString(endPoint);
     m_replyQueue =
                QueueClient.Create(m_replyQueueName);
     m_replyQueue.OnMessage(ReceiveMessages);
}
private void SendMessages(object sender, RoutedEventArgs e)
{
    m_timer = new TimeToLabel(m_azureTime);
    int numTasks = Int16.Parse(m_azureCount.Text);
    m_sentMessages = numTasks;
    m_receivedMessages = 0;
    for (int i = 0; i < numTasks; i++)
    {
        var message = new BrokeredMessage("parallel");
        m_queue.Send(message);
```

```
    }
}
private void ReceiveMessages(BrokeredMessage message)
{
    m_receivedMessages++;
    if (m_receivedMessages == m_sentMessages)
    {
        m_timer.Complete();
    }
}
```

The proxy creation of the queues is almost identical to the WorkerRole code for doing the same thing. The code to send the messages and then receive the messages is pretty trivial.

In the example application with one WorkerRole configured it takes about four seconds to send a message to the cloud, process the task, and send a message back. It takes about five seconds to run the task locally on a laptop. When the service is configured to scale up to 16 instances there is no noticeable speed degradation when 16 messages are sent to trigger the 16 tasks in parallel. The full 16 tasks can still be performed in less time than it takes one task to be process locally.

It is amazing how much the power of parallel computing in the cloud can be harnessed by such little code!

6. SUMMARY/CONCLUSION

C# offers a variety of ways both old and new, including additions to the language and .Net libraries, that enable the developer to more easily implement asynchronous parallel programming with more straight forward logic that is easier to understand, debug, and maintain. The Task Parallel Libraries combined with the new language features *async* and *await* offer powerful advanced parallel programming capabilities. Third party frameworks like Digipede and Microsoft's Azure cloud offer powerful grid computing solutions that are simple to implement. These tools offer great power and simplify the process of creating parallel processing applications, but still require the developer to have a sound understanding of parallel programming problems and pitfalls. The continuing evolution of C# and the .Net libraries give developers the power to design, write, and maintain code in a more sequential manner while taking full advantage of parallel processing capabilities to offer a wide array of ways to offer consumers better performing applications for their unquenchable thirst and unending desire for more speed.

7. REFERENCES

[1] Parallel Extensions. http://en.wikipedia.org/wiki/Parallel_Extensions

[2] Microsoft Java Virtual Machine. http://en.wikipedia.org/wiki/Microsoft_Java_Virtual_Machine

[3] Parallel Programming in the .Net Framework. http://msdn.microsoft.com/en-us/library/dd460693(v=vs.100).aspx.

[4] Task Parallel Library (TPL) http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx

[5] Azure for Research http://research.microsoft.com/en-us/projects/azure/faq.aspx

[6] Digipede http://www.digipede.net/products/digipede-network.html