

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

Department of Information Engineering



Diploma thesis

**Asynchronous and parallel programming in .NET
framework 4 and 4.5 using C#**

Author: Milan Manasievski

Supervisor: Ing. Jiří Brožek, Ph.D

Prague 2015

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Department of Information Engineering

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Milan Manasievski, BBA, BA

Informatics

Thesis title

Asynchronous and parallel programming in .NET framework 4 and 4.5 using C#

Objectives of thesis

The diploma thesis focuses on ways and means of parallel and asynchronous programming in .NET Framework using the C# language. The main aim of the thesis is to introduce various methods for asynchronous programming in C# and to demonstrate and compare their application on example problems.

Methodology

The methodology of this thesis is based on analytical and synthetical approach. The student will do an analysis of various scientific and technical information sources. Based on the synthesis of the gained knowledge and his own practical experience he will prepare various example tasks that will be used to demonstrate different approaches to asynchronous programming in C# and will compare them based on selected measurements.

The proposed extent of the thesis

60-80 pages

Keywords

.NET Framework, C#, parallel process, asynchronous process, programming, algorithm, concurrency

Recommended information sources

Cleary, Stephen. Concurrency in C# Cookbook, Sebastopol: O'Reilly Media, 2014

Herlihy, Maurice. Shavit, Nir. The art of multiprocessor programming, San Francisco: Morgan Kaufmann, 2012

Razdan, Sanjay. Fundamentals of parallel computing, Oxford: Alpha Science International Ltd, 2014

Skeet, John. C# in depth, 3rd edition, Connecticut: Manning Publications, 2013



Expected date of thesis defence

2015/06 (June)

The Diploma Thesis Supervisor

Ing. Jiří Brožek, Ph.D.

Electronic approval: 25. 3. 2015

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 30. 03. 2015

Statutory Declaration

I hereby declare that I wrote my diploma thesis “Asynchronous and parallel programming in .NET framework 4 and 4.5 using C#” on my own with help of the listed bibliography.

In Prague, Czech Republic 30.03.2015

.....
Milan Manasievski

Acknowledgments

I would like to thank my friends and family for support in this endeavor and also my colleagues and employer for being flexible, understanding and for giving me time and space to complete this diploma thesis in a timely manner. I would also like to thank my mentor, professor Jiří Brožek, for his guidance and professional expertise in the subject matter.

**Asynchronní a paralelní programování v .NET
Framework verze 4 a 4,5 pomocí C#**

**Asynchronous and parallel programming in .NET
framework 4 and 4.5 using C#**

Souhrn

Tato diplomová práce se bude zabývat na asynchronní a paralelní programování v rozhraní .NET framework verze 4 a verze 4.5. Cílem této práce bude prokázat a poskytovat lepší přehled o modelu task-programing aplikace který Microsoft představil a porovnávat různé aplikace z hlediska rychlosti a řádky kódu slouží k zápisu pak a rozdíly mezi nimi pomocí jednoduché statistiky. Pomocí literatury jsem shromáždil, vám vysvětlí, co by bylo nejlepší způsoby, jak dosáhnout rovnoběžnosti o žádostech, psát o návrhové vzory použité a poskytovat výstřižky kódu, které vám pomohou čtenáři získat lepší celkové pochopení úloh paralelní knihovny a výhody, které dává ve srovnání starších metod a sekvenční programování.

Klíčová slova

Paralelní programování, paralelní výpočty, .NET, C #, vícejádrové procesory.

Summary

In this diploma thesis the author will elaborate on asynchronous and parallel programming in the .NET framework version 4 and version 4.5. The aim of this thesis will be to prove and provide better insight on the task-programming model that Microsoft introduced and compare different applications in terms of speed and lines of code used to write them and the differences between them using simple statistics. Using the literature gathered, the author will explain what would be the best ways to achieve parallelism on applications, write about design patterns used, and provide code snippets that will help the reader get better overall understanding of the Task Parallel Library and the benefits it gives in comparison of older methods and sequential programming.

Keywords

Parallel programming, parallel computing, .NET, C#, multicore processors.

Table of Contents

1. Introduction	5
2. Objectives and methodology	7
3. Theoretical part.....	10
3.1. Introduction to C# and the .NET framework.....	10
3.2. The task-programming model.....	14
3.2.1. Task completion.....	17
3.2.2. Language support in the Task Parallel library.....	20
3.3. Creating, working and waiting on Tasks.....	23
3.3.1. Coordinating, cancelling and exception handling of Task	26
3.3.2. Exception handling	26
3.3.3. Task cancellation	29
3.3.4. Task priorities.....	30
3.4. Dangers of concurrency	31
3.5. Execution model and types of parallelism.....	36
3.5.1. Common types of parallelism.....	39
3.6. Exception handling and the Parallel class	42
3.6.1. Breaking out of parallel for loop	42
3.7. Designs and patters for parallel programming	43
3.8. Asynchronous programming	48
4. Practical part	53
4.1. The Mandelbrot application with implemented cancellation	63
4.2. The stock history application	66
4.3. Comparisons between two applications	74
5. Conclusions.....	75
6. Bibliography	78

1. Introduction

In this diploma thesis the reader will get acquainted and get a deeper understanding of the Microsoft's .NET framework and the new task based library that is offered with the .NET 4 and Visual Studio 2012.

In the thesis the author will also elaborate on why would one choose to develop applications using the new task based model and not the old and traditional event based programming as it was in the previous version of the .NET framework and the C# language. The motivation for learning and using the Task based library in .NET mainly falls into two different categories and those are:

- Responsiveness

The Task parallel library from Microsoft provides solution for long running tasks and blocking operations such as updating the user interface or input and output operations when one is working with files from the file system. It also hides latency of these long running and blocking tasks and this becomes increasingly important to start these tasks in the background and returning to the user interface immediately even before the completion of the task (Campbell, Johnson, Miller, & Toub, 2010).

- Performance

When it comes to performance, it can reduce the time of CPU bound computations by balancing the workload and also executing the tasks in parallel. In todays computing the database size and network operations are that are performed on these data sets are increasing, therefore the performance of this execution is equally important (Razdan, 2014).

These two categories can be put in different categories too, and those are parallel and asynchronous programming, where the responsiveness goes to asynchronous programming and the performance goes to parallel programming. In order to make use of these principles with the .NET framework 4, the author will be using the Task Parallel library or

the TPL, which is only available from .NET 4 and above, and it came out recently to Silverlight 5 (Campbell, Johnson, Miller, & Toub, 2010).

There are other and older existing techniques that deal with parallel and asynchronous programming and one might wonder why the world needs another library for these same features since before the .NET 4, there was the existence of threads and also there was something called Asynchronous Programming Model for achieving similar things? The answer for the above question is that the new Task Parallel Library from Microsoft is an evolutionary model for creating background tasks, which involves much more features than the old event and based model for instance. The new Task based model is evolutionary and the model consists of cancelling tasks, easier exception handling, high-level constructs (in the older model there were only threads) and more. Now, there are much better capabilities in the new library than there were in the old API's (Campbell, Johnson, Miller, & Toub, 2010).

2. Objectives and methodology

In this part of the thesis, the author will elaborate on the methodology used and objectives that the author tries to achieve. A clear intention and aspiration will be presented and also shown how the author is planning to achieve the aim set forth and the methods that will be used in order to achieve this aim. Since this project will be about the task parallel library in .NET framework 4 and 4.5 versions, the focus of the thesis will be on the benefits of adopting the methods that this library offers and also focus on the differences between these methods and the way it was done before, more specifically, comparisons on how parallel programming was done before the release of the task parallel library and what is the difference after will be presented.

The author will try to prove that the methods that are used in the new task parallel library are easier to understand and learn and also prove that they are based on higher abstraction and in programming terms that means that programmers or developers do not need to worry about any low level details such as memory management, task execution or task priority.

The task parallel library does all this internally and most of the implementations are even hidden from the developer. Moreover, the author will try to prove that the task parallel library help us to efficiently and effectively utilize all the physical cores on a multicores architecture and do that in a manner that will not affect the execution of the program, but it will improve its responsiveness and perceived speed. At the end of this project the author will try to prove that with less code, code easier to understand, read and write will be able to write high performing application that are using 100% of all the CPU cores.

The author will also try to prove that there are differences in application designs and the creators of the task parallel library identified this and provided the consumers with different classes for writing parallel code and also making sure that the threads in the application are not sitting idle without doing anything.

A principle called SMART will be used for setting objectives for the thesis and the SMART abbreviation stands for:

- Specific

In order to reach the first point of SMART, the author needs to know exactly what is need to do in order to achieve the goal of the thesis and in this case the author will present two different types of applications, one is done using an older technology for presentation called Windows Forms however, its computationally very expensive and will make sure that the thread that is executing it have no idle time. After that the author will present different versions of the application where he will try to make it run in parallel see the difference in the result and also compare. The second application is a console based application that has no user interface, this application will know how to connect to the internet and download some data about stocks for any company that the user requests. This application does not require high CPU power, however it makes threads to go to sleep and be idle for the time of the downloading is done. With the task parallel library the author will try to prevent this.

- Measurable

The application will have built in mechanisms for timing and also for counting the number of threads that are executing in parallel, the author will try to present the number of CPU time to the reader and provide clear and concise differences between the different versions of the applications and also the benefits from adopting a certain approach over the other.

- Achievable

The first application that will be presented on this thesis will be based on a Mandelbrot set and in the most basic sense the Mandelbrot set is a group of numbers that display a certain unusual properties and as per definition the Mandelbrot set is the visual representation of an iterated function on the complex plane. In this application, deeper levels of the Mandelbrot set will not be explored as it can be quite complex and its computation can

take even several minutes. For the application, for each pixel on the screen the program will check if that particular pixel is within the Mandelbrot set or not, as an input the program will give a random number between -2 and 2. The second application will be slightly simpler and the user who will be running the application will only need an Internet connection. There are classes in the .NET framework that know how to connect to the Internet and download some content from a certain website.

- Realistic

The objectives set forth must also be realistic, in this term the author needs to think in terms of time, resources and skills. Both application to write and the actual thesis to write will be time consuming and challenging, however if everything starts on time, probably four or five months before, the author will be able to write the necessary applications and before doing that the theoretic part should be already done. Since the author is working as a web developer in a company in Prague, Czech Republic and also covered most of the material in the Czech University of Life Sciences, safely can be assumed that the necessary skills are available for utilization.

- Time constrained

The literature has to be gathered well before starting the thesis and that should be around a year before starting to actually write anything. After the literature is gathered, which will consist mostly of books, the author will start classifying them and start writing the theoretical part. The theoretical part will be based on parallel programming in the .NET framework and while writing and researching additional knowledge can be accumulated and this will be beneficial for the applications needed for the practical part of this thesis. Last month before due date, everything should be pretty much finished and finalized. This time will be scheduled for last consultations with the thesis mentor to make sure that the author is on track and doing the things right. Last two weeks, the author will go throughout the literature once again and make sure the thesis is formatted correctly, its in presentable form and ready for submission.

3. Theoretical part

3.1. Introduction to C# and the .NET framework

In this section the author will be taking the reader on a fast pace introduction to C# syntax and the CLR (the common language runtime). The .NET framework is also known as virtual machine and some people in the development community refer it to. The .NET framework contains many of the components that developers need in order to run a program successfully and do not worry about on which CPU architecture is running, how much RAM memory it has or that fact that computers do not understand code, they only understand binary language (Skeet, 2013). What the .NET framework does is takes our code and transforms it to the binary language using the common language runtime depending on which processor its being executed. The .NET framework also has classes and objects that are written by someone else but at a disposal of the consumer to use if found fit and necessary. The task parallel library that the author will focus on in this thesis is written by someone else in Microsoft, but shipped with the framework to consumers. Lets look into the most common data types that are used in the .NET framework.

Type name	C# keyword equivalent (if any)
System.Int32	<code>int</code>
System.Double	<code>double</code>
System.Enum	<code>enum</code>
System.Object	<code>object</code>
System.String	<code>string</code>
System.Text.StringBuilder	
System.DateTime	

Table 1: Common types in C# and .NET. Source: self-authored

C# is an object oriented programming language and it follow a hierarchical approach in inheritance. The top-level object is the System.Object from which all other objects inherit and that's how they share similar functionality (Skeet, 2013).

The below code shows us some simple usages of these data types:

```
//Assigning values of 42 and 54 to variables of type integer
int a = 42;
Int32 b = 54;

//Assigning value of 67.0 to variable of type double
double x = 67.0;

//Assigning variable a to x
x = a;

//Casting and assigning variable x to a
a = (int)x;
```

Above, the value 42 is assigned to a variable of type integer called “a”, after that this is done with another variable called “b”, but in this case the integer value of 54 is assigned. Now, lets create an array of values. Array is a sequence of values with same data type and it has fixed length. In C# the below mentioned syntax needs to be used to declare arrays:

```
//Assigning length of three to array of integers
a = new int[3];
```

The above syntax means that array of integers is created but there are no values assigned to it. The array of integers called “a” is pointing to null. However, after some time later in the program same array can be pointed to allocate space in RAM and set a value.

```
//Assigning a null
int[] a = null;
//... and then assigning length of three to array
a = new int[3];
```


The above syntax means that the author is creating an array of integers with a length of three, however there are no values assigned to it yet. To set the various integers in the array the below syntax is used:

```
//Assigning values to array
int[] a = { 40, 41, 42 };
```

The below code snippet is slightly more complicated. The program walks through every member of the array and prints it on the screen for the user to see. This can be achieved with the below code:

```
int[] a = { 40, 41, 42 };

//Looping through length of array and write values to screen
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine(a[i]);
}
```

In the above for-loop, the author is indexing the array with the “i” variable and using the Console.WriteLine() method to write to the console.

Another common type of loop is the while loop in C#, the syntax is presented below:

```
int i = 0;

//Looping through length of array in while loop
while (i < a.Length)
{
    Console.WriteLine(i);
    i++;
}
```

Above, value of zero is assigned to variable called “i” and then a while-loop is entered to print the variable and then increment it. This is the “++” operator in C#. After some condition on top becomes false, the while loop is exited.

Next, the author will present classes in the .NET library and how one can create and instantiate classes with the C# language. Classes are logical grouping of code and they are normally thought as object that contain state and behavior. By adding what its called a field in the .NET framework a state is added to our class, and the behavior in the classes is achieved by creating methods or also known as functions. The difference between a method and a function is that the method belongs to an object where as a function can be considered global. Below there is a class called Calculator, this has a simple state and behavior:

```
public class Calculator
{
    private double sum;

    //Method that adds number to sum variable
    public void Add(double a)
    {
        sum += a;
    }

    //Method that returns the sum variable to caller
    public double GetSum()
    {
        return sum;
    }
}
```

Now there is state and behavior in our class and this is the typical way classes are created in the .NET framework.

3.2. The task-programming model

The programming model is based on the concept of a task and task can be regarded as a unit of work or an object denoting an ongoing operation or computation. In order to create this object of work the library needs to be included in the project and this is done by the below statement:

```
using System.Threading.Tasks;
```

After the library imported the new object of work or the Task can be created as below:

```
//Instantiation of a new task  
Task T = new Task(code);
```

As in the code above, it is shown how to create an object of type Task. However in order to inform .NET on the tasks execution, a method that belongs to this task object needs to be called as in the example below:

```
T.Start();
```

Same method for creating and starting a task can be done in a different way and this is shown below:

```
Task T = new Task  
{  
    //some code goes here...  
};  
T.Start();
```

More graphically explained way is shown below:

```
using System.Threading.Tasks;
```

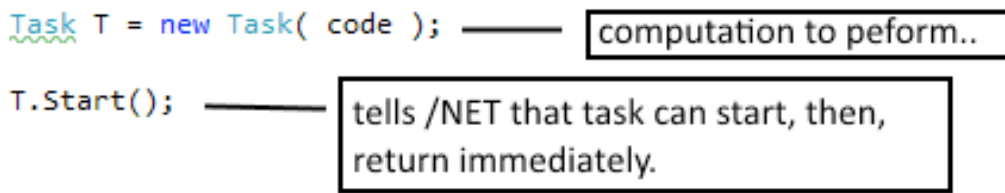


Image 1: Creating task and starting it. Source: self-authored

The important aspect on what is happening next, is a bit difficult to understand. At the above code, there are two code threads that are potentially executing in parallel and that is the original thread, which is the main thread or the UI thread, and a new thread that was created with starting of this task T. This process is called “forking”. For more details see the below image:

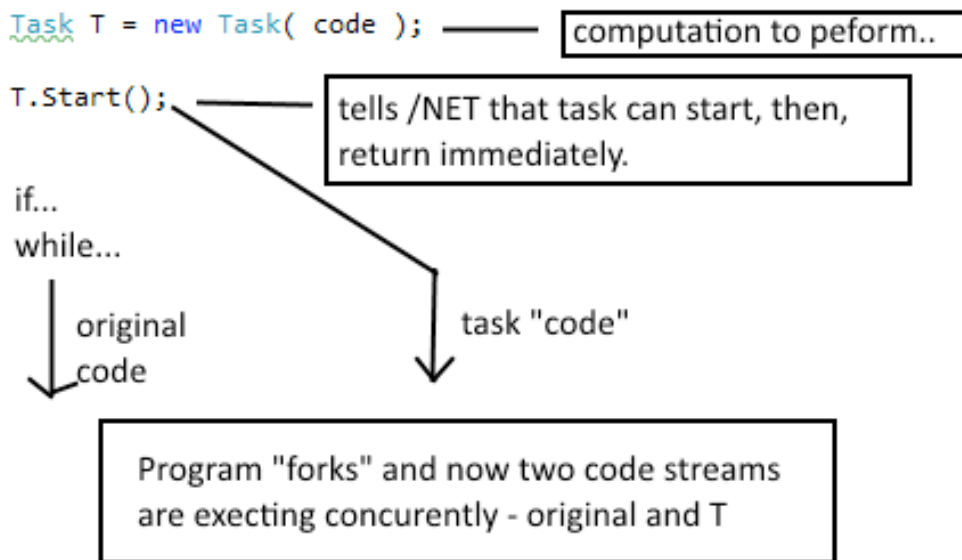


Image 2: Graphical representation of forking. Source: self-authored

The above image shows how forking is done, this is slightly more difficult to understand than a normal program execution flow, and it cannot be seen in the debugger as it will be shown later (Cleary, 2014), but this is definitely happening and there is a parallel execution happening. For instance, any type of code can be added that is not dependent on the main thread code in the “computation to perform” and this will be done in parallel while the other code which is the origin code will continue its way.

If one looks in the execution mode very quickly in the Task Parallel Library, one will see that code based tasks are executed by a thread on some processor running on some machine and the operating system assigns a thread to a task and this thread “sticks” to the task until that task is completed (Campbell, Johnson, Miller, & Toub, 2010).

Lets look at this from two different perspectives. First, the author will present the parallel programming model from a perspective of a single code CPU machine and see how this is performed there. When the main thread creates a worker thread to complete some job on a single core CPU, the tasks cannot go in parallel as there is only one core CPU present. However, the operating system has something called a “scheduler” and this tool does something that is called “context switching”. What happens is the execution head jumps from one thread to another very fast in order to seem like the operations are done in parallel, however they are not. They are not even done faster. In best scenario they are done in same way as in the traditional single threaded programming model. However, the ability to jump from one thread to another, it give the application more flexibility and it can respond to any other tasks that are raised by the user (Pacheco, 2011).

The image below gives us a graphical representation of how the single core CPU’s handle multiple tasks and asynchronous programming.

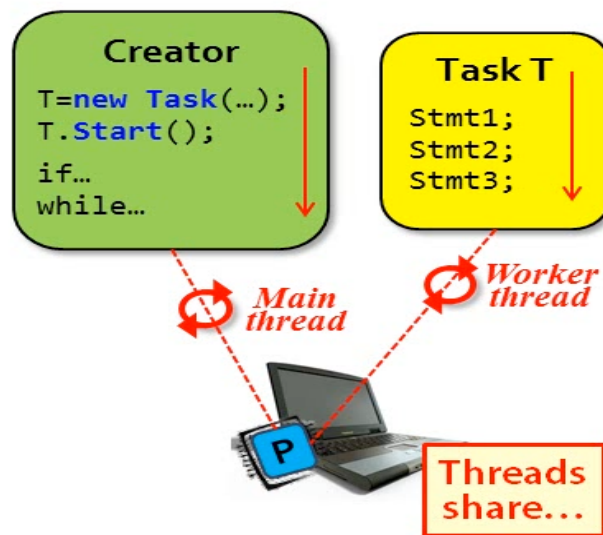


Image 3: Two threads sharing same CPU. Source: <https://msdn.microsoft.com>

From the model above, it's shown that the threads are sharing the CPU cycles and the Windows scheduler is telling which task to run and where, and it also takes into consideration user inputs and raising events. In the second perspective the author will show how a multi-processor machine is dealing with multiple tasks and multiple threads running. If hypothetically a program is executing on an eight-core machine, which could be a web server for instance that is taking in many requests simultaneously, on this multiple-core machine these requests and tasks created can run in parallel because there is enough CPU power to do that. This example is illustrated graphically below:

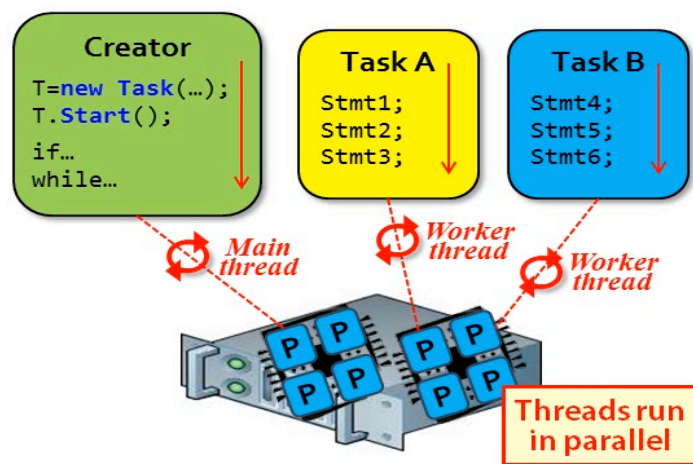


Image 4: Tasks on multiple cores. Source: <https://msdn.microsoft.com>

Since the speed of the CPU's is no longer increasing as per Moore's law (Watson, 2014), the goal of parallel programming is to take advantage of every single core in the processor and make sure it's busy, in case the machine does not have more than one core it does not matter. The Windows Scheduler with its context switching feature will make sure that to the user it seems like the tasks are running in parallel.

3.2.1. Task completion

Now that the tasks are executing, what happens when the task is done? Well, the answer is simple, this is done when the code block is finished either naturally by completing the task or throwing an exception.

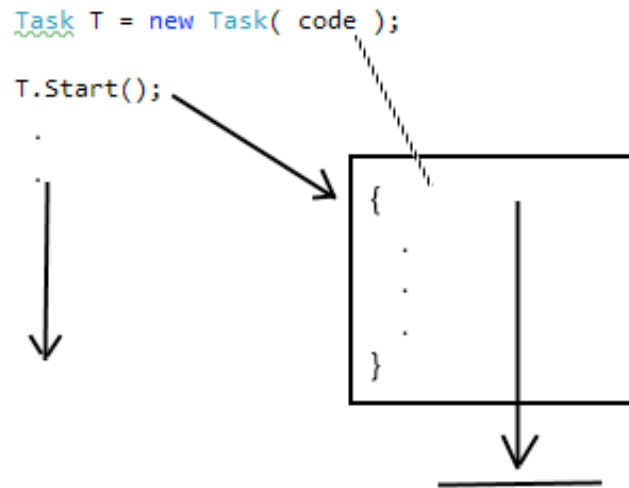


Image 5: Parallel execution. Source: self-authored

At this point it is important to note that even though one can create as many threads as they like and as many tasks as they want, the only thread that is able to touch and update the user interface is the main thread (Cleary, 2014). If the developer wants to update the user interface with a worker thread the application will crash and it will throw an exception. Later in this diploma thesis the author will show how to deal with exceptions in the Task Parallel Library and how to resolve them and potentially avoid.

For now two different approaches will be mentioned that can be used to resolve issues that arise with updating the user interface. The first resolution would be to update the user interface in the context of the main task that is executing the program. The code for this is below:

```
Task T = new Task( code );

//Tells the scheduler this needs to be run in same context
T.Start(TaskScheduler.FromCurrentSynchronizationContext());
```

The code snippet above is telling the task that the synchronization context is the same thread that created it; this is only done when the main thread created the task and not a worker thread. After this call, it is safe to update the user interface using the background worker task without the application crashing on us (Cleary, 2014).

This approach however can make the user interface unresponsive, and this is not what the user wants. The other way of doing this is the more preferred way and this approach does not block the user interface.

```
Task T = new Task( code );

//Task T2 continues after T1 is done.
Task T2 = T.ContinueWith(task1 =>
{
    // . . . code that updates the user interface. . .
}, TaskScheduler.FromCurrentSynchronizationContext());

T.Start();
```

Now, when the tasks starts, it is divided into two smaller parts and the second task is updating the user interface and this is the task that the consumer should run in the user interface context. This works just fine.

Since a lot of material is already covered about tasks in the .NET framework it good to mentioned that there is even more efficient way in starting and running a task in the .NET framework and that is using the `Task.Factory.StartNew(code)`; and therefore this is the preferred technique when creating new tasks (Esposito & Saltarello, 2014). This has exactly the same process, but it is slightly more efficient:

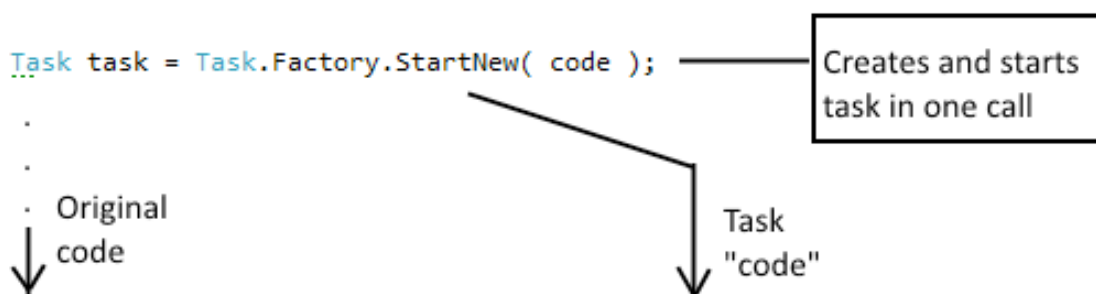


Image 6: Parallel execution with immediate start. Source: self-authored

3.2.2. Language support in the Task Parallel library

Some of the built in language features in the Task Parallel Library are already mentioned, however its worth mentioning that the TPL support lambda expression and closures (Campbell, Johnson, Miller, & Toub, 2010).

Lambda expressions in the .NET framework is just an unnamed block of code which means consumers usually write methods that have a name and can be called from different places in our application, however when it comes to lambda expressions consumers can only call them from one particular place and not from any other without re-writing it (Sharp, 2013). The lambda expressions are also known as inline methods.

The reason why lambda expressions are mentioned in the thesis is because the TPL is designed to accept lambda expressions as parameters and this makes it very easy and convenient in creating tasks (Campbell, Johnson, Miller, & Toub, 2010). Behind the scenes, the lambda expressions are generated as a custom class plus a delegate. The .NET framework and the way its created is managed by the .NET framework create the custom class, and the delegate is a “function pointer” in programming terms which serves as a wrapper around the lambda that consumers are creating (Sharp, 2013).

Below a code snippet is shown of what is happening in the .NET compiler when it comes across a lambda expression passed into a Task object.

```
// Parallel code
Task task = Task.Factory.StartNew(=>
{
    something1();
    something2();
    something3();
});
```

Above a Task executing three different statements passed in as lambda expressions is shown. Since there is the “=>” sign after the parameters this means that the body is passed as lambda expression along with the arguments (which in this case is none) and then the three different statements “something1()”, “something2()” and “something3()”.

```

// Class that compiler generates from above parallel code
[CompilerGenerated]
private sealed class c_DisplayClass1
{
    public void b_0()
    {
        something1();
        something2();
        something3();
    }
}

```

The class generated by the .NET framework is shown above. Notice that even though a name for the lambda expression is not specified the compiler creates a class and a method within that class that matches our lambda expression. The method does not take any parameters and contains the three statements that it was initially passed to the lambda expression. The compiler generates both the class and the method names. The second language support that exists in the TPL is the notion of closures. Closures are blocks of code plus the supporting environment that they live in. When one is passing data to the Task object outside from its environment, the .NET compiler needs to do some additional work to make sure those variables are included in the class that it generated. This is known as closures (Pacheco, 2011). The image below depicts this scenario.

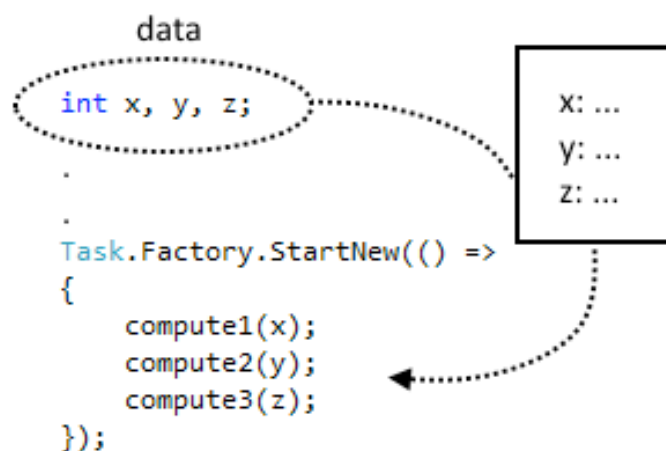


Image 7: Data passed in a parallel task. Source: self-authored

Important to mention is that the variables that are passed in the Task object are passed by reference. This means that the compiler does not copy the variable but it only creates a pointer to it so it can reference it later when needed. These variables passed by reference now become shared variables and if the consumer wants more threads executing at same time using these variables, they potentially have unsafe code. The variables can be updated and executed using multiple threads and the results may be inconsistent (Esposito & Saltarello, 2014). This is known as race condition. In continuation, closure variables will be presented to the reader. Closure variable are stored in a compiler-generated class, as its shown with the lambda expressions. Before continuing in more depth on the asynchronous and parallel programming its important to distinguish between these two concepts and make sure the reader understands what this is. The asynchronous model helps the application to stay more responsive which means consumer can start an operation and immediately return to the UI and the parallel model helps us to divide the workload so the job is finished sooner, the asynchronous model is all about responsiveness and the parallel model is about performance (Herlihy & Shavit, 2012).

Passing data correctly into a task is important, because in case data is not passed currently it may have unpredictable results and errors. Lets say consumers need to create ten tasks that are running in parallel, in tasks are put into a loop and try to name them as the index of the loop, no errors or warnings will be thrown at first but the running of the code will provide the caller with some unpredicted and unexpected results. The below graph shows the incorrect way of passing data and keeping track of the index of the loop:

```
for (int i = 0; i < 10; i++)
{
    Task.Factory.StartNew(() =>
    {
        int taskId = i;
        .
        .
        .
    });
}
```




Image 8: Passing argument from for-loop (incorrect). Source: self-authored

The variable in the above image will be always ten even though one might think that the variable “taskId” will increment accordingly in the loop. The reason for this is that there is a lag in the task creation and the starting of the task. It does not happen immediately and that’s why the value of the variable will be ten. This means the tasks will start executing after the loop is done. Now the correct approach to do this is to write the loop as below:

```
for (int i = 0; i < 10; i++)
{
    Task.Factory.StartNew((arg) =>
    {
        int taskId = (int) arg;
        .
        .
        Console.WriteLine(taskId);
    }, i);
}
```

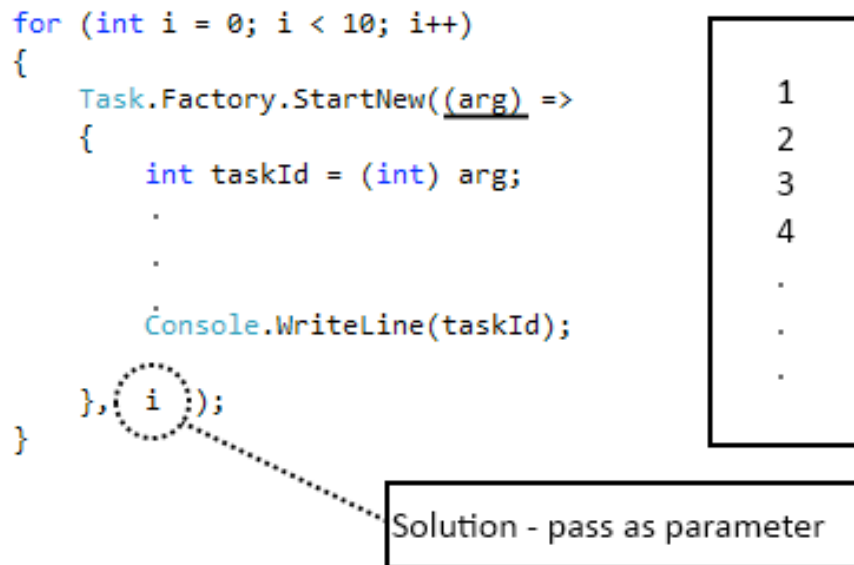


Image 9: Passing argument from for-loop (correct). Source: self-authored

The “arg” variable is passed into the task and the “i” index is passed as an argument to the task itself. In this way the task can monitor the value of “i” before its starts.

3.3. Creating, working and waiting on Tasks

In the TPL there are several technologies that come into play that enable all this asynchronous and parallel programming. These technologies consist of the Task parallel library and this provides the Task object. The task scheduler is responsible for mapping tasks to the available worker threads in the thread pool and the resource manager is responsible for managing the pool of worker thread. On top of this is are the concurrent data structures that are thread safe, which means thread can access them simultaneously and this provide the expected result and also the parallel LINQ that is used for accessing

and retrieving data from these concurrent data structures. They're also few other technologies as well, including support for parallel loops, data parallelism, parallel invoke etc. and these technologies are referred to collectively as the asynchronous and parallel components of .NET 4 (Campbell, Johnson, Miller, & Toub, 2010).

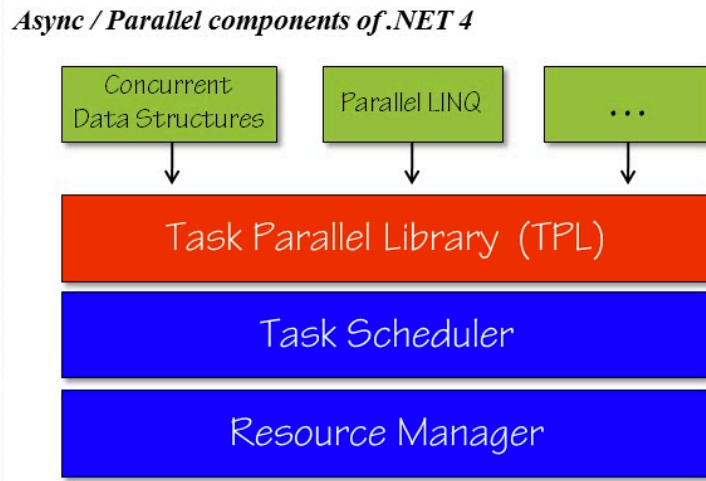


Image 10: “Async” and parallel components. Source: <https://msdn.microsoft.com/>

In this particular part of the diploma thesis the reader will be familiarized on how to wait for a task to finish, how to return a value from a task, how to compose tasks, how to handle exception and how to cancel a task in case its stuck somewhere and its not returning.

Waiting for a task to finish is one of the most common requirements. The whole point in this is that consumers have to wait for the operation to complete before outputting a tasks result to the user or before they start the next phase of the computation. Good news for this is that the waiting is easy. Consumer just needs to call the Wait() method on the task object (Campbell, Johnson, Miller, & Toub, 2010).

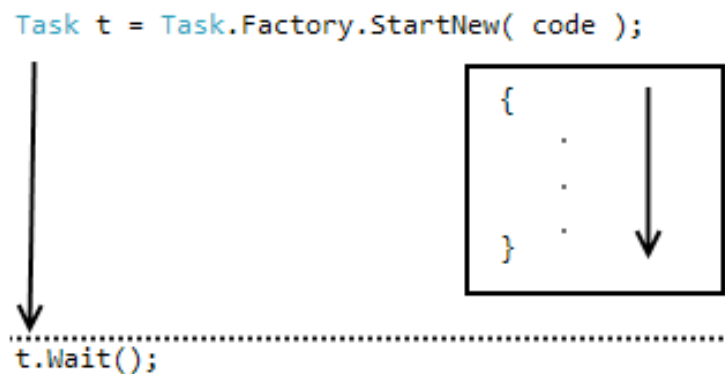


Image 11: Explicit call to Wait(). Source: self-authored

In case the task is not finished until the method is called, the Wait() method blocks the execution of the program and waits for the particular task. When the execution returns from the Wait() method, the status property of the Task object will be one of the three possible statuses and those are: RanToCompletion, Cancelled or Faulted.

The Wait() method is one of the options for waiting on tasks, this can be handy when there is no result coming out of a task, just some computation is done without an end result. However, in case the task is returning a result, consumers can make use of a much better way in harvesting this result without the explicit call to Wait(). Every task has a Result property where the result of the computation is stored. Therefore, if the task is returning a value, consumer can call the Task.Result property and this will be only available when the task is finished, if the task is not finished the Result property will call the Wait() method implicitly and make it wait (Watson, 2014).

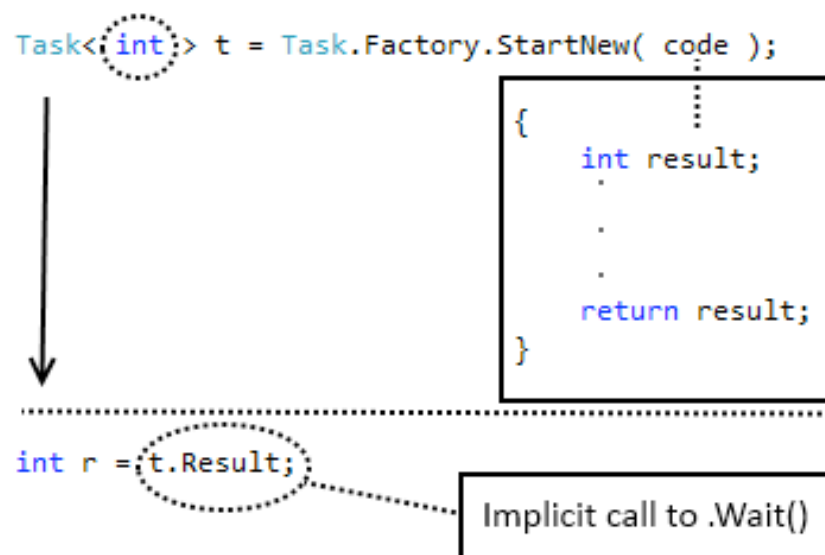


Image 12: Implicit call to Wait(). Source: self-authored

Sometimes there is the need for creating multiple tasks that are executing in parallel and consumer does not know specifically which task will finish first. Consumers of the library would not want to wait for a task while its still executing, the best scenario would be to wait on a tasks as they finish one by one, however, since consumers don't know which one will finish first what should we they? The best option here is to put all those tasks in an array and call the WaitAll() method. This will make sure these tasks are being waited on as they finish and there is no blocking on threads involved (Watson, 2014).

```
Task t1 = Task.Factory.StartNew( code );

Task t2 = Task.Factory.StartNew( code );

Task t3 = Task.Factory.StartNew( code );

Task[] tasks { t1, t2, t3 };

.
.
.

// Waits for all tasks to complete
Task.WaitAll();
```

3.3.1. Coordinating, cancelling and exception handling of Task

In this part of the diploma thesis the reader will be familiarized with the exception handling when it comes to Task and Task based programming, also the author will elaborate on Task cancellation for when there are long running task and it might be stuck somewhere, also task priorities, parent-child tasks and parameter passing into these task objects.

3.3.2. Exception handling

In regards to exception handling, in the TPL in .NET when a task throws an exception that goes unhandled then:

- Task is terminated
- Exception is caught by .NET, saved as part of an Aggregate Exception AT, and stored in the task objects Exception property.
- Aggregate Exception is re-thrown upon Wait() method, Result property or WaitAll() method (Campbell, Johnson, Miller, & Toub, 2010).

The author will present examples and also he will look into the implications for these. Therefore, when a task throws an exception, is the exception is unhandled, the .NET framework will re-throw the exception when consumer calls the Result property and that is why the best way to handle exception in the TPL is to wrap the Result property with a try-catch block such as below:

```
//Creating task that returns integer result
Task<int> t = Task.Factory.StartNew(code);
.
.
.
// Wrap unsafe code in try block
try
{
    int r = t.Result;
}
// Catch exception if thrown
catch (AggregateException ae)
{
    Console.WriteLine(ae.InnerException.Message);
}
```

Note that all the exceptions that are thrown by the TPL are of type `AggregateException`, however the actual message for the exception can be accessed through the `InnerException.Message` property where one can see additional information on the error. Sometimes, now it many case though, the task can throw multiple exception depending on how many threads are fired out by that particular task, if this is the case the result of the task would be a tree of exceptions (Cleary, 2014).

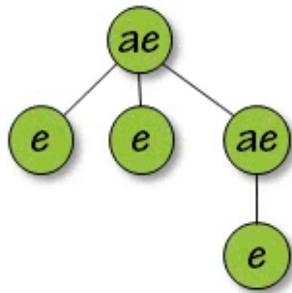


Image 13: Exception three. Source: <https://msdn.microsoft.com/>

The tree at the root has a single AggregateException but the sub-nodes of the tree can be of type Exception or of type AggregateException depending on the task object. In order to get to all of the exceptions in the tree, the tree needs to be traversed using an algorithm, however these types of algorithms take time to write and therefore the .NET framework provides a method which flattens the tree and then it can be accessed using a “foreach” loop. In this way all the exceptions in the tree can be accessed and analyzed (Cleary, 2014).

```

catch (AggregateException ae)
{
    //Method that linearizes binary tree
    ae = ae.Flatten();

    foreach (Exception exception in ae.InnerExceptions)
    {
        Console.WriteLine(exception.Message);
    }
}

```

When it comes to designing exception-handling techniques, the developer has to observe all unhandled exceptions, if this is not the case the exceptions are re-thrown when the task is garbage collected. Garbage collection is not in scope of this thesis however, in short it’s a feature of .NET application where after the lifetime of an object it gets garbage collected and the memory is freed ready to be used by another object. This is not a good time to

catch exception; therefore its good practice to touch on the exception before the garbage collector sees it (Sharp, 2013). There are few practices that programmer can follow and Microsoft suggests these practices:

1. Call Wait() method of touch on the Result property of the task since the exceptions are re-thrown at this point.
2. Call Task.WaitAll() since here as well the exceptions are re-thrown when tasks are finished.
3. Touch the Exception property of the task after the task has completed
4. Subscribe to TaskScheduler.UnobservedTaskException event

3.3.3. Task cancellation

When it comes to task cancellation .NET provides the users with the notion of cancelling tasks. This makes sense when it comes to long running task that consumers of the library do not wait for, since they take long time to execute and long running tasks are consuming more system resources.

Microsoft has chosen the cooperative model for task cancellation where both the creator and the task play a role. In this approach the creator of the task passes a “cancellation token”, starts the task running and later signals to be cancelled if necessary. The job of the task is to monitor the token while it’s executing and if cancelled is signaled it performs cleanup and throws an exception. The exception type that is thrown in this case is OperationCancelledException. If one is using this particular model and the task is cancelled the Status property of the task is set to “Cancelled” and everyone that was monitoring or observing this task will know what happened It also provides the task a chance to cleanup before exiting (Campbell, Johnson, Miller, & Toub, 2010). When it comes to task cancellation the responsibility of the creator is to create a new CancellationTokenSource() object, grab a reference of the Token property and pass that token to the new task as parameter.

```

//Create object with cancellation options
CancellationTokenSource cts = new CancellationTokenSource();
//Grab reference of the token
CancellationToken token = cts.Token();

Task t = Task.Factory.StartNew(() =>
{
.
.
.
}, token); // token passed as parameter

```

The token is being sent to the .NET and the task it's monitoring the token using the closure mechanism. Now, if some condition occurs, the user can call the `Cancel()` method on the `CancellationTokenSource` object and cancel the task. For now the reader was only acquainted with the creator's side of responsibilities, now author will present the responsibilities of a task in this cooperative model approach.

The job of the task is to monitor the token and if cancelled, to clean up and throw the appropriate exception (Campbell, Johnson, Miller, & Toub, 2010).

3.3.4. Task priorities

When it comes to the notion on priorities, the .NET framework does not provide an out of the box support for scheduling tasks according to priority, however the developer is able to create their custom priorities via the task scheduler. However, task may form a parent-child relationship. The idea behind this is that the child task attaches to the parent when the `TaskCreationOptions` object specifies its created and this. The parent task does not complete until all the child tasks are completed. Even if one child fails the remaining children run to completion. Any exceptions that are thrown by the children can be caught using the parent as a single point of exception handling.

3.4. Dangers of concurrency

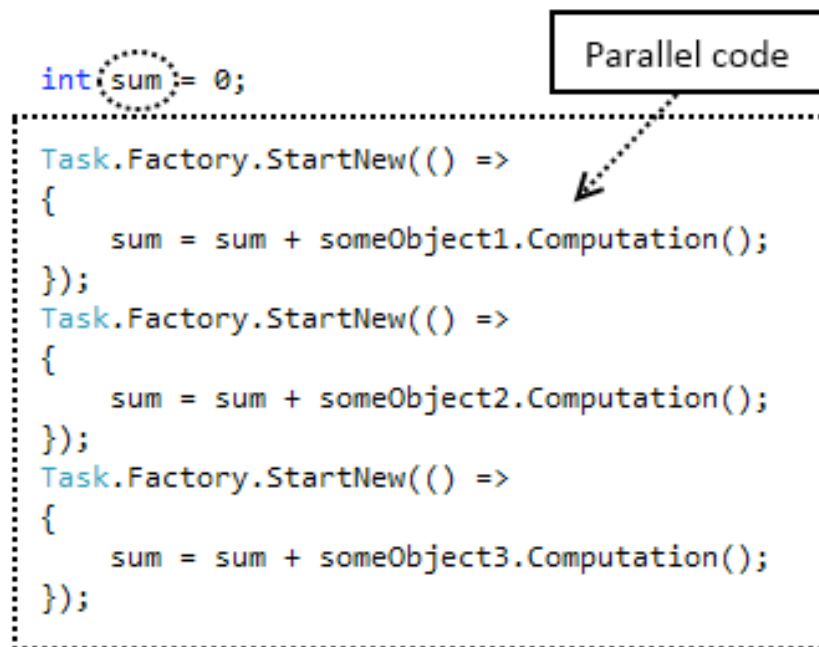
The asynchronous and parallel programming model of .NET can have many pitfalls. It is very powerful model and therefore it has many dangers. The most popular problems that can arrive while writing parallel software and those are:

1. Race conditions
2. Starvation
3. Livelock
4. Deadlock
5. Optimizing compilers
6. Optimizing hardware

However, even though this list is not short, many of these issues are not connected with the developers code and they are the responsibility of the hardware manufacturers, compiler provides etc. The only real threat that can be caused by developers is the race condition. Correctness in the world of parallel programming is important and also a problem sometimes since the parallel code that is written has infinite number of way that can execute and interact. The goal is to minimize these interactions and provide two guarantees. The Microsoft documentation has two simple ways in achieving this and those two ways are: safety and live-ness. Safety means developers have to strive so that nothing bad happens in the code and live-ness means that eventually something good happens. There will be some elaboration on these two ways later in the diploma thesis. The race condition in parallel programming exists when the outcome depends on the timing of events (Cleary, 2014). For example if user runs the program today and get one answer and if they run the program some other day they get another answer or outcome, this is happening because two of more threads are reading and updating a single shared resource. Race conditions are something that developers are trying to avoid. Following the definition for race condition, it's important to mention that a critical section is. A critical section is the smallest region of code involved in a race condition and the goal for developers is to identify the critical section and to resolve the race condition. A shared resource can be anything really, it can be variable in code, objects, collections files etc. with one sentence,

anything that multiple threads are trying to read and update is considered as shared resource (Cleary, 2014).

Consider the below code snippet:



```
int sum = 0;

Task.Factory.StartNew(() =>
{
    sum = sum + someObject1.Computation();
});
Task.Factory.StartNew(() =>
{
    sum = sum + someObject2.Computation();
});
Task.Factory.StartNew(() =>
{
    sum = sum + someObject3.Computation();
});
```

Image 14: Tasks updating global resource (incorrect). Source: self-authored

Notice that each task that its created, it accesses the same resource and the shared resource in this case is the variable “sum”. The problem is that all tasks access “sum” but the problem is that every task reads and writes to “sum” and this is when the problems occur. This code is unsafe and it’s the most common mistake in the concurrent software today. In this particular case the variable “sum” will not contain the sum of the expressions but it will contain the value of the last task that was able to write to it. This is very dangerous and it’s very common in the software today (Pacheco, 2011).

There are at least three solutions to race conditions. First, the application needs to be redesigned to eliminate shared resources (critical sections) for example by adding local variables that are accessed only by one thread. Or by using thread safe entities within these critical sections, for example the task parallel library offers thread safe data structures and the third possible solution to race condition is to use the synchronization to control access within critical sections. Locking for instance is one of the most used primitive feature to synchronize thread access to shared resources, it works by surrounding the critical section

with a lock on a common object and this restricts entry to one task at a time thus eliminating race conditions and parallel access. Below there is sequential code that stores all the results from two different computations to a variable “sum”. This is working fine, and there are no race conditions or any issues with concurrency in this case since there are no multiple threads accessing the same shared resource (Pacheco, 2011).

```
// . . . Sequential code . . .  
int sum = 0;  
  
sum = sum + object1.Computation();  
sum = sum + object2.Computation();
```

In the code below, in order to speed up execution of this method the code above can be parallelized into the example that is shown below:

```
// . . . Parallel code . . .  
int sum = 0;  
var l = new object(); // common object  
  
Task.Factory.StartNew(=>  
{  
    int t = object1.Computation();  
  
    lock (l) // locking code block for other threads  
    {  
        sum += t;  
    }  
});
```

In the above example with the parallel code, the shared variable “sum” is declared but on top of that a common object “l” is also declared. Then, task is written to perform the computation, but when it comes time to read and write the shared variable sum (the critical

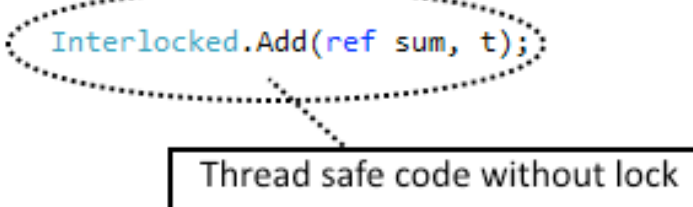
section) the region is surrounded with a lock on “l”. If “l” is unlocked, a single thread is able to access the region and safely update sum. After entering the region, the thread locks object “l” and no other thread is able to enter the region. All the other threads that theoretically should access the “l” region are waiting until the first thread that enters is finished and is not locking the region anymore. For best performance the critical section should be as small as possible so that the section is locked as briefly as possible. Note that no other threads can enter the locked sections therefore those threads are waiting and are idle, which is not best practice (Herlihy & Shavit, 2012).

Another solution to race conditions is interlocking; this is a simple hardware based lock for simple arithmetic critical sections. This method is generally more efficient than locking, since its software based, but the trade of is that the interlocking only applies to certain conditions.

Parallel code

```
int sum = 0;

Task.Factory.StartNew( () =>
{
    int t = object1.Computation();
    Interlocked.Add(ref sum, t);
}
}
```



Thread safe code without lock

Image 15: Tasks updating global resource (correct). Source: self-authored

The above example is the same as it was used before, but it's using the “interlocked” class from the .NET framework and its hardware based whereas the previous lock was software based solution (Cleary, 2014).

As a last example a lock free approach to eliminating race conditions will be presented. Note that locking is a blocking code that makes all the other threads wait before they update a shared resource. Wherever possible, lock-free approach should be used in solving these types of issues. In the below example a local variable will be used in our tasks and make the task return the result of its computation instead of updating the shared resource.

```

Local memory ("task local state")
Task< int > t1 = Task.Factory.StartNew( () =>
{
    int t = object1.Computation();
    return t;
});

Task<int> t2 = Task.Factory.StartNew(() =>
{
    int t = object2.Computation();
    return t;
});

```

Image 16: Task returning value. Source: self-authored

In this example there is no access to a shared resource from any of the tasks, instead, they are return their individual result of the computation. This code can and will be executed in parallel with no concurrent issues. After retrieving the result from both tasks one can combine it using another thread or using the main thread. Lock free designs are not always possible, but when they are, they eliminate race conditions and also improve performance. Regarding the lock free design, there are several schools of thought. Note that the most common opinion is that there “lock free” does necessarily mean freedom from locks, but it means freedom from unnecessary waiting. The below table shows all the primitives that are available in the .NET framework that solve the issue of thread unsafe code:

Primitive	Purpose
Monitor	General-purpose .NET synchronization class
Lock	Enforces one-at-a-time semantics using Monitor class
Mutex	Win32 lock suitable for inter-process sync (“mutual exclusion”)
Intelocked	Hardware-based lock for simple arithmetic operation
Sempahore	Enforces N at-a-time semantics via Win32
SpinLock / SpinWait	Lock-like mechanisms that loop (“spin”) instead of yield CPU
Barrier	Allows tasks to synchronize (“sync-up”) before start of next phase
CountdownEven, ManualResetEvent, AutoResetEvent	These three events are additional ways for tasks to synchronize with one another.

Table 2: .NET primitives for solving race-conditions

These are primitive types of making sure the code consumers write is thread safe. The .NET framework 4 and above provides us with a higher-level abstraction of how to deal with these issues and its generally safer and more reliable. For instance, the .NET library provides concurrent data structures and therefore, consumers of the library do not need to use these primitives that are difficult to maintain and write and since the concurrent data structures were already mentioned, here is somewhat more extensive list of these:

- ConcurrentBag
- ConcurrentQueue
- ConcurrentStack
- ConcurrentDictionary
- BlockingCollection

All these data structures will be discussed in more detail later in this thesis.

In conclusion for this part of the diploma thesis, the author presented various issues that can rise with there is a race condition in our application and the only reason a developer is motivated to deal with these issues is because of performance. In fact the whole topic of asynchronous and parallel programming revolves around performance and ways consumers can improve it in their applications (Esposito & Saltarello, 2014).

3.5. Execution model and types of parallelism

In the first part of this thesis, the reader was familiarized with the dangers of concurrency and in this part the author will look into how parallel programs execute in the .NET framework and the types of parallelism one is likely to identify and focus in our application. Here the author will look into the execution model in the .NET framework, the difference between tasks, threads and cores will be presented and also author will look into the different types of parallel code that one can write and identify the possible parallelism in our problem domain.

Our job as developers is express work as tasks and the .NET's framework job is to execute these tasks intelligently. The tasks in .NET are not expensive to run or to create, however

there is some overhead involved when creating tasks and one should now about the pitfalls that can occur. A question that often emerges is what is the minimum granularity to offset the cost of creating a task? It turns out that there is a straightforward answer to this question and the answer is: the task execution should take at least 200-300 hundred CPU cycles before it's worth creating (Cleary, 2014). For instance if a task is only having few multiplies and adds that is not enough work to offset the cost in executing these tasks in parallel, so the performance will be no better than the sequential performance. Now the author will present how .NET executes these tasks and how threads and cores are introduced into the picture: Lets assume that there are many windows processes running in .NET and each process has its own application domain. The app domains are using the available threads that are in the thread pool provided by the operating system. This pool contains one or more worker threads that execute on the CPU core. The thread pool is shared across all the application domains. The threads are executing on the available CPU cores and typically the task parallel library creates two threads per CPU core. The task parallel library includes a task scheduler, a resource manager and a global work queue. When tasks are created it's the job of the task scheduler what happens to those tasks and by default new tasks are added to the end of the global queue. The resource manager manages the thread pool and its creating and removing threads as demand changes or when certain kind of tasks are created and completed. Underneath all this there is the operating system, which is ultimately responsible for deciding when threads run and on which CPU cores. As the number of threads grows, as single global queue for task distribution can become significant point of contention. A better approach then is to associate a local queue with each of the worker threads. When a task creates additional sub tasks, those tasks are added to the workers local queue instead of the global queue since worker threads are checking their local queue before checking the global queue. For example if one has a loop in the global program for creating tasks, each of the tasks created is added to the global queue. The worker threads eventually retrieve these tasks and when work comes along, the threads unblock and begin execution on one of the CPU cores. This is the default behavior of tasks and task scheduling (Cleary, 2014).

The scheduling of tasks can also be customizable, that will involve customizing the TaskScheduler class in the .NET framework and doing this is easy, but doing it in the right way its very tricky and not in scope with this thesis. Even if one is not touching the

TaskScheduler class to overwrite behavior, one can expect very good results from the default behavior (Mattson, Sanders, & Massingill, 2013).

There are several observations that can be made about the default behavior of the task scheduler and those are:

1. A single global queue is not a good approach, so the task parallel library adds local queues to reduce contention.
2. The task scheduler responds to demand by dynamically creating tasks and by default a thread is dedicated to a particular task until the task completes.
3. If task blocks and waits, its corresponding thread blocks and waits too, therefore the task scheduler creates new worker threads to maintain task completion rate.
4. Allowing threads to steal tasks for better load balancing and more efficient execution does dynamically balancing the task workload and this.

Besides observations there are also couple of assumptions that one needs to know about the task scheduler and those assumptions are:

1. Tasks are short lived (at most 2 seconds)
2. Execution order does not matter (random order of execution is ok)

These assumptions can be overridden; in fact if random order for instance is not OK, consumer can make the scheduler to create tasks with the “Fairness” option enabled and what happens here, is .NET will add the task to the global queue of the thread pool and never to the local worker’s queue and in this way tasks will be queued in the order in which they were created, however there is not guarantee that they will be finished in the same order.

Even though it was mentioned that tasks are usually short lived, there is the opportunity to create long running tasks that perhaps are running while the program is running and they are logging exceptions or maybe write logs to a file. In this case one can create the tasks with the “LongRunning” option. In response to this option when the task is created .NET framework creates a non-worker pool thread and dedicates a special thread to this task. However, one needs to keep in mind that this option greatly increases the cost of task

creation, since it takes roughly 200 000 CPU cycles to create, 100 000 cycles to retired and every context switch is 6 000 to 8 000 cycles, plus the memory cost for stack space.

3.5.1. Common types of parallelism

When it comes to parallel programming, there are several types of parallel code that is written out there, and this does not only apply for .NET framework but for all programming platforms out there. The most common types of parallelism in programming are:

1. Data parallelism

The notion behind the data parallelism is that the same operation is executed across different data (Mattson, Sanders, & Massingill, 2013). Note that the same operation (UpdatePortfolio) is called for different customers:

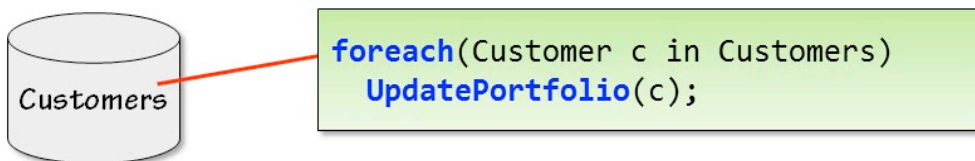


Image 17: Calling method for each of customers. Source: <https://msdn.microsoft.com/>

The example above is data parallelism; below the author will contrast this with task parallelism.

2. Task parallelism

Task parallelism is when you have different operation executing on the same or different data. For instance, consumers have financial services application that updates portfolios, predicts tomorrow markets and assesses risks. Now, these are three different operations operating across variety of data (Mattson, Sanders, & Massingill, 2013). A graphical representation of this example is shown below:

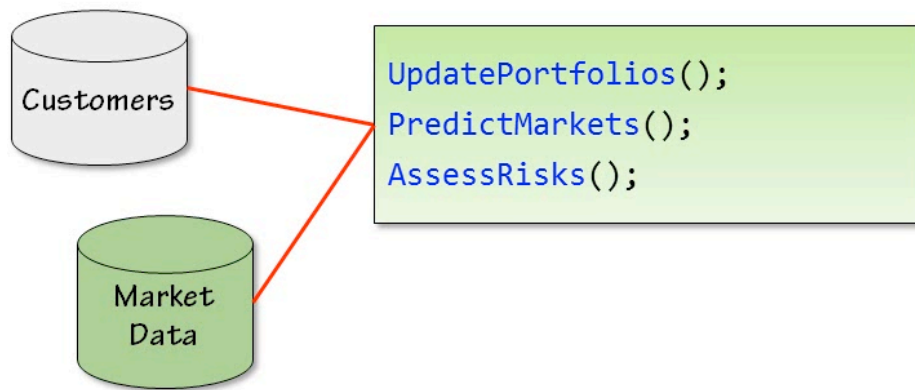


Image 18: Operations executing on different data. Source: <https://msdn.microsoft.com/>

3. Dataflow

The third type of parallelism is the dataflow, this occurs when the operations depend on one another i.e. data flows from one operation to another. One example is the classic pipeline where the result from one task feeds the next. One needs to be aware that the data flow scenarios do not always parallelize well since some tasks can finish faster than others and since tasks are dependent on one another in this case, the parallelism does not always yields the results one expects. In the dataflow type, the speed of execution is as fast as the slowest task (Mattson, Sanders, & Massingill, 2013).

4. Embarrassingly parallel

This type of parallelism is the most wanted approach and this is what everyone needs to be focusing when writing software. This method is also known as delightfully parallel and whatever you call it; this is the best type of parallelism to have in any type of software. To define this type of parallelism, one can say, “A problem is embarrassingly parallel if the computations are entirely independent of one another”. This means there is no data flow between operations (Mattson, Sanders, & Massingill, 2013).

Now, most frequently used classes will be presented in the .NET framework that enable all these types of parallel programming that were mentioned above.

Hypothetically consumers are all familiar with the classic for-loop in computer programming; the .NET framework is making use of this concept to create a parallel for loop for executing the method faster and in parallel. See below:

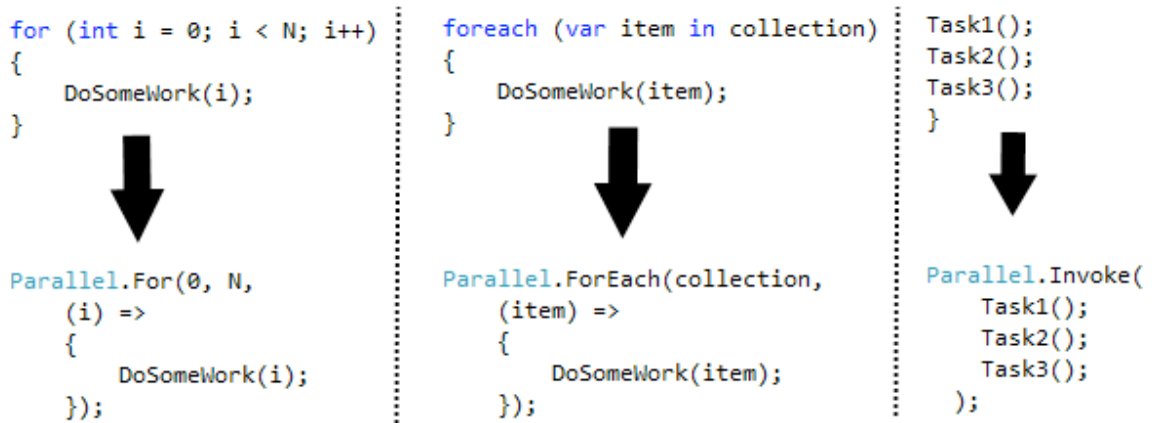


Image 19: Sequential code with parallel equivalent. Source: self-authored

If one looks more carefully, one can notice that both `Parallel.ForEach()` and `Parallel.For()` are a method calls and they take three parameters, which are essentially copies of the values in the original loop. Internally, what the task parallel library is going to do is create one or more tasks to execute the iteration of the loop. The only difference is the `Parallel.Invoke()` call, which takes N number of arguments and invokes them in parallel. Those arguments can be another tasks or primitive parameters (Cleary, 2014).

Important to note is that the `Parallel` class is using the fork-join pattern, which is well established and time tested approach to application level parallelism. By default the application is running sequentially, until it reaches `Parallel.For()`, `Parallel.ForEach()` or `Parallel.Invoke()`, at which point the program forks and starts running multiple tasks concurrently. Now, when these tasks finish, the program is said to join and returns to the sequential stage (Cleary, 2014).

The three methods mentioned above, chunk the data for good load balancing. This is a dynamic technique, where the chunks start small and then grow in size. This type of data partitioning is not good for small or simple loop bodies. There are four types of data partitioning in order to achieve maximum use of the parallelism in the program. Those types are:

1. Range (divides the data that needs to be processed into the number of CPU cores available).
2. Stride (creates one task per core to stride through the data. This increases the likelihood of a balanced distribution)
3. Hash (This is dynamic technique that separates the data to be processed based on their hash code)
4. Chunk (Is used by the three methods mentioned above, where chunks start small and they grow in size as time passes)

Custom data partitioning is supported by the `Parallel.ForEach()` method call however this is out of scope for this diploma thesis.

3.6. Exception handling and the Parallel class

When it comes to exception handling in the Parallel class, in case an exception is thrown and its unhandled by any method in the Parallel class, then tasks are allowed to finish their current iteration of work and then exceptions are deferred and re-thrown as aggregate exception and the bottom line here is that the unsafe code can be surrounded in a try-catch block and all exceptions will be caught successfully that are thrown by the Parallel class.

3.6.1. Breaking out of parallel for loop

Much like C# lets you break out of a for-loop, the Parallel class lets you break out of `Parallel.For()` or `Parallel.ForEach()` loop. Note that the method will have additional parameter called “loopControl” which is the key piece that allows you to break out of a loop, for instance by calling the `Stop()` method. On top of that, the methods that have the “loopControl” object are now returning `ParallelLoopResult` object and this is used to see if the loop ran to completion or was interrupted in the middle. There are couple of choices if one wants to break out of a loop; you can call the `Stop()` method to stop the loop immediately or call the `Break()` method if one wants to be sure that the earlier iterations

complete successfully. In other words, if a task calls `Break()` on iteration 10, the loop continues to run until the iterations 0 to 9 finish, and then it stops (Cleary, 2014).

In summary of this part, one has to understand that our job as developers is to create tasks, and .NET's job is to execute those tasks as efficiently as possible. In executing tasks it was shown that the task parallel library provides us with a versatile and efficient execution engine with dynamic load balancing task scheduler, work stealing tasks and intelligent resource manager for our threads. Of course as any system, there are tradeoffs in its design, so one also needs to understand the task parallel library's execution model so one can work around it or customize it when necessary. The reader was also familiarized on how to identify the types of parallelism in our programs such as; data, task, dataflow and embarrassingly parallel and if possible to use one of the higher level abstractions of the task parallel library to exploit this parallelism (`Parallel.For()`, `Parallel.Foreach()`, `Parallel.Invoke()`).

3.7. Designs and patters for parallel programming

In this section the author will focus on high-level designs for correct high performance software. The author will start by presenting a few design problems for us to think about, then will dive into the well known parallel patterns, such as; pipeline and dataflow, concurrent data structures provided by the task parallel library, the famous producer-consumer patterns, map reduce and task local state, parallel LINQ (language integrate query), speculative execution and finish with the asynchronous programming model (Mattson, Sanders, & Massingill, 2013).

As a change of pace, it's better to start with some design problems that one can talk about. Lets say one needs to execute 100+ CPU intensive operations and each operation takes roughly 3-5 minutes, where the execution order does not matter. What would be the best solution for this particular problem?

It turns out that there is couple of solutions in this case; the first would be to create 100 tasks where each task will map to the particular operation with no special creation options. However, this is not a good solution, since the tasks are long running, .NET will inject more threads into the thread pool and eventually consumer will have 100+ CPU threads competing for the CPU cores and since all the tasks are CPU intensive, the system will

spend more time in context switching and less time in working. The second solution is to create 100 tasks (one per operation, but flag them with the long running option and this solution also turns out not to be effective for the same reasons as the first one. A good solution in this case is to create one task per core and as the tasks finish, one creates additional tasks and to do this one can use the wait-all-one-by-one pattern or one of the parallel for-loops with option of max degree of parallelism.

The second problem, where the application needs to do 20+ I/O operations for instance it could be web page downloads. Order does not matter and it can take anywhere from few milliseconds to few seconds to complete. So, what is the best option for downloading these webpages?

One of the solutions is to create one task per download with no special creation options and it turns out that this is a good solution and ends up starting one download for each worker thread in the pool whereas for the previous problem this was a bad solution and the difference is that the downloads are expected to finish within few seconds and this is before .NET starts injecting more threads into the pool and the second difference is that these are I/O operation and do not require much CPU power this the threads to not compete for the CPU nearly as much as the CPU intensive tasks. The second solution to this problem is to create façade task per download using the FromAsync class plus the APM patter. The advantage with the second solution is that the when the download is started, the thread is returned to the thread pool and its ready to do other work since its no longer needed for the downloading of the web content. The third solution is to create one task per core as in the previous problem and the final approach is to use the producer-consumer pattern where consumer can have one task to download the content and the remainder to be used for processing of the downloaded data.

The third problem is where one requires logging facilities to be enabled on our application and we want the logging to run every 30 seconds. What would be the best design for this type of application?

This problem is probably hardest of the previous ones that we've looked into. To solve this problem consumer will need to consider the following design guidelines (Mattson, Sanders, & Massingill, 2013).

Here are the guidelines that consumer of the TPL needs to consider:

1. Create the task with the long-running option
2. Before the app closes, first task needs to join with the second task and catch any exceptions.
3. Design clean shutdown of task so application closes (perhaps use task parallel library cancellation?)
4. If the task can crash, consumer needs to design a way to monitor and restart (check task's status property via a timer or application's message loop)

The part that follows is more in depth explanation on how the design patterns look like and when is the best idea to implement these in our application. The patterns that were already mentioned briefly above are:

1. Pipeline

The pipeline method is a good choice when consumer has a linear flow of data forms one task to another. Example applications include image processing, user interface updating and standard workflows. The coding is very straightforward; one creates and starts the first task normally and then the `ContinueWith()` method is used to create the second task that waits until the first one finishes. The `ContinueWith()` method takes a single argument, which one can name, as they want, denoting the task that finished. This allows us to consume the result from stage one and then use that value as the input to stage two, which can be computed and returned to the caller. Likewise, consumers can use `ContinueWith()` again to fire up the third task that waits until the second task finishes, consumes the result and produces result for stage three of the pipeline. Interestingly, note that there is no parallelism here since the tasks run sequentially one after the other. Parallelism comes from repeating this pattern over and over again for instance inside a loop that processes set of data (Mattson, Sanders, & Massingill, 2013).

2. Dataflow

The dataflow pattern is a generalization of the pipeline allowing many to one and one to many relationships. The coding is similar, except with the many to one relationship in which case, the `ContinueWhenAll()` method is used to wait for all the other tasks to finish (Mattson, Sanders, & Massingill, 2013).

3. Concurrent data structures

Another way to increase parallelism is by using concurrent data structures. These data structures are thread-safe and they can run in parallel. These data structures can be used in conjunction with the dataflow and pipeline method by inserting the data structures between stages. Consumers of the TPL need to keep in mind that the data structures must be thread-safe since they are being accessed in parallel. Now, another more subtle consideration is whether the data structures should be bounded or unbounded in size. By default most collections are unbounded, which is dangerous in the parallel world, where one task may run faster than another to the point where one task is consuming most of the memory filling its collection with data or the reverse situation where tasks are chewing up CPU cycles checking for data to arrive into their generally empty collection (Mattson, Sanders, & Massingill, 2013).

4. Producer-Consumer

The producer-consumer pattern is a good pattern to use for long running workloads where the speed of data generation (the producer) is very different from speed of data consumption (the consumer). The general pattern allows one or more producers connected to one or more consumers by way of a shared data structure. This data structure serves to throttle the faster component and keep the system better balanced. A classic example is where the producer reads data from the disk or network and the consumer is processing this data. Now, the producer-consumer pattern is based on the blocking collection, and this collection make easy work on implementing the pattern. The blocking collection is a fixed size collection in a parallel world. Blocking producer if the collection is full and block

consumers if the collection is empty. Fixed size collections are much more realistic and typically improve performance by throttling the faster component so it does not consume too much memory or spend much time in checking for data in empty collection (Mattson, Sanders, & Massingill, 2013).

5. Map-reduce

Map reduce is a well-known and commonly used pattern for embarrassingly parallel for search and data mining application. Map reduce has been around for a long time, but became very popular with Google revealed this was the technology driving their search engine. The idea is to split your application into two phases, map and reduce. In the case of search the map phase applies search function to a portion of the data, producing a set of local intermediate results, then, in the reduce phase the intermediate results are combined to produce the final results set R. In the case of Google search R is the ranked set of page hits. There are various strategies for implement map reduce in the task parallel library. The first is to fire off N mapping tasks that return data of type T and then have the parent thread do a wait-all-one-by-one to reduce each of those results as the tasks finish. Another approach is to use the Parallel task combined with the Task Local Storage (TLS) (Mattson, Sanders, & Massingill, 2013).

6. Parallel LINQ

LINQ is a query like language at the level of C# of Visual Basic. Its very similar to SQL and the designers of LINQ had the idea to create it based on LINQ. LINQ is very easy to learn and understand and the task parallel library provides us with a parallel version of this language. All the querying can now be done in parallel using multiple threads. The parallel version of LINQ is enabled just by adding the AsParallel() method to any LINQ query. LINQ enables us querying of all enumerable data sources (Mattson, Sanders, & Massingill, 2013).

7. Speculative execution

Speculative execution is good pattern when consumers have multiple sources for generating a result. An example would be calling various web services, such as primary and back up services when some value is needed. The idea is pretty simple, consumers create a set of tasks and the first one to finish wins. The results are harvested and the rest of the tasks are cancelled (Mattson, Sanders, & Massingill, 2013).

8. Asynchronous programming model

The last pattern is the APM pattern. This is commonly used for asynchronous operations such as file and network I/O. The pattern provides two methods; a begin method to start the operation and an End() to complete the operation and to harvest the result. Good news is that various .NET classes already support the APM pattern, including file stream and http web requests. The advantage of using this pattern and in particular these .NET classes (FileStream and HTTPWebRequest) is that the Begin() method starts the operation on a thread, but then returns that thread to the worker pool until the operation completes. This allows that thread to be used for other work during the I/O operation. To take advantage of classes that implement this pattern, the TPL provides first class support by way of façade tasks. In particular the TPL's FromAsync() method is used to create the task that wraps calls to Begin() and End() and then simply use the standard task mechanism to wait, continue and/or harvest the result (Mattson, Sanders, & Massingill, 2013).

3.8. Asynchronous programming

In this part the reader will be familiarized with the asynchronous programming model in C# and the .NET framework 4.5. Note that there is a difference between parallel and asynchronous programming and by parallel is meant that different threads or tasks executing code in parallel and by asynchronous is meant that the code executed can be performed by multiple tasks as well, but it's not necessarily in parallel. The distinction between the synchronous and asynchronous programming is not new. The .NET framework has supported this since the version one, but it was very tedious and difficult to

write and many developers were reluctant in writing asynchronous code. Synchronous API's are the easiest to understand since it's a code that happens one thing after another. Lets look into a code snippet. This code below is downloading string from a website and puts it into a text box for the user to see. The method DownloadString() is synchronous which means that the thread waits until the work is completed before it displays is to the user (Campbell, Johnson, Miller, & Toub, 2010).

```
// Create web client that connects to the internet
WebClient webClient = new WebClient();
.
.
.

// Download text from website
string text = webClient.DownloadString("http://www.milan.com/");

Console.WriteLine(text);
```

These methods are also sometimes described as blocking, because they block the progress of the code until the work is done. DownloadString() has to block because its return value is the data caller asked it to fetch. It cannot return the data until the download is complete, and this could be a problem if the download takes a long time. This code updates the user interface element once the download finishes which means it must be running on a user interface thread and consumer are not allowed to touch UI element from worker threads and since all UI output happens on user interface threads the user interface is not able to respond on user input. It will be unresponsive for as long as the thread is stuck in waiting for the download to finish and if it takes too long, Windows gets impatient, it greys out the application's window and adds a "Not Responding" message to the header of the application:

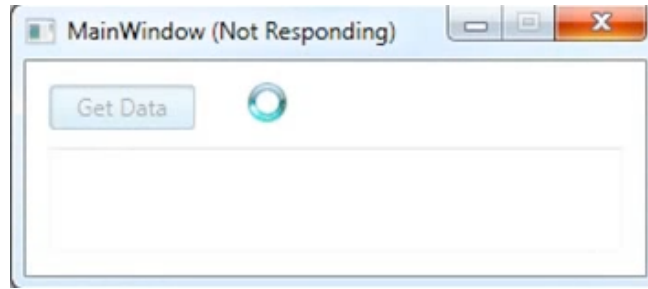


Image 20: Application not responding. Source: self-authored

The image above constitutes a common problem with synchronous programming and it happens when consumer needs to run slow responding code such as the method that was mentioned earlier and this is not a problem only with the Windows applications, this is also a problem with web applications too. So, multithreading will not necessarily help us here as one could run the work on a separate thread and that could work fine on the client side, but it may not help on the server, it will just add complication without increasing performance (Campbell, Johnson, Miller, & Toub, 2010). Now, the WebClient class offers additional method for downloading content from a website, called `DownloadStringAsync()`:

```
// Asynchronous method for downloading text from website  
string text = webClient.DownloadStringAsync(new Uri ("http://milan.com/"));
```

This method return immediately because it does not wait for the download to complete before returning, so this is asynchronous API, meaning that the progress of the thread is no longer synchronized with the work being done. This solves our responsiveness issue but it creates a new problem and that is how to get the data once the downloading eventually completes (Watson, 2014). Below is the modified version of `DownloadString()` method that is waiting on the download to complete but it does not block the user interface or the execution of the program:

```

private static async Task SomeFunction()
{
    WebClient webClient = new WebClient();
    .
    .
    string text = await webClient.DownloadStringAsync(new Uri("http://www.milan.com/"));
    Console.WriteLine(text);
}

```

keywords "async" and "await" must be used together

Image 21: “Async” and “await” used in method. Source: self-authored

When the thread comes to the await keyword it will start the asynchronous work and immediately return, in this case to the windows application.

Any method marked with the new “async” keyword can contain away expressions. The await expressions kick off the asynchronous work, but they don’t wait for it to complete and rather than blocking the thread return early and continues from the await keyword once the operation completes. It is also to worth mentioning that the methods that can be “awaited” are almost always returning a Task object, and that’s why the “async” and “await” keywords are part of the task parallel library.

People often think that asynchronous work means multithreading, but that is not necessarily true since if consumers for example are fetching data over http, this does not mean that they require an actual free thread to do that, but this is not true, even though asynchronous API’s are used, sometimes the thread object does not exist. The truth is that the request needs to get a thread involved only at a specific point in the process and its technically possible to do everything on one thread, without blocking that thread and there are two reasons for this (Herlihy & Shavit, 2012).

1. Most of the time spent is waiting for a response from the server, once the client machine sends a request caller cannot do anything until response is sent, so what would a thread even be doing in this case while the request is in progress. The synchronous API’s are putting the thread to sleep until there is a response from the server because there is no work for the thread to do.
2. There is also a more subtle issue, even while the client computer is sending a request over the network or receiving a response it does not need to use a thread

most of the time for that since the network hardware is capable of doing that work. When consumer wants to send data over the network the device driver for the network interface card programs the network hardware, telling it what data to send and most network hardware is capable of fetching the network bytes out of main memory directly so the driver only needs to tell the interface where the data to be sent in the computer memory (Watson, 2014).

Thread only needs to be involved when the request is send and when the request is received from the server, everything in the middle is done by the actual hardware and the time that thread spends telling the hardware what to send and getting the returned data is much less than the actual work that is being done over the wire and most I/O works this way, this is the same with disk access for example, it takes miniscule amount of time to tell the disk controller which data to load from disk and much longer for the moving parts of the disk drive to actually load the data and this is true even with solid state driver that do not have moving parts.

If one needs to build a highly scalable web application for example, then one of the goals is to keep the thread in our server as busy as possible and in an environment like Asp.NET for example, there are only limited number of threads for processing http requests and if these threads need to wait for long input output operations to complete you end up putting those threads sitting idle and doing no work (Watson, 2014). A web service call that travels over the network is often measured in milliseconds and if the call takes 250 milliseconds than the processing thread needs to sit idle for 250 milliseconds and if all threads are tied up and idle then any new http requests that are arriving to the server, AP.NET will queue them up and make them to wait until one of the processing threads frees up (Watson, 2014).

4. Practical part

In the practical part of the thesis, the reader will understand how to best implement the task parallelism and also what are the benefits of implementing this feature of the .NET framework 4 and later. Also the author will look into the differences between the old way of doing it and the new way using the framework. Moreover, the author will prove that the new method is less time consuming, more efficient and more effective with some good example applications that the author developed as part of the practical part of this thesis.

First application that will be presented is the Mandelbrot set application. The Mandelbrot set is related to chaos theory and the application is drawing a fractal image that is very recognizable.

The Mandelbrot set “is a visual representation of an iterated function on a complex plane” where the function is any function that takes an input and produces a certain output. In the Mandelbrot set, the said function is repeated and called using an iterative approach, the iterative function performs the same as any other function but it’s called over and over again, with each output used as the next input. On top of that, the Mandelbrot set can also be defined as a set of all numbers that do not grow exponentially in the function $x^2 + c$ and the whole graphical representation of the Mandelbrot set is represented in a cycle with a radius of 2.

Briefly, the application computes whether or not a given point falls within a mathematical definition of the Mandelbrot set. It’s a good test case because the work is computationally expensive. As it will be shown later, lighter colors on the screen represent an area that requires very little computation, and the darker colors represent more intensive computation. The app can be challenging to parallelize, because the workflow is very uneven, since from the start, there are very few computations and as we get to the middle the computations are getting more and more complex.

The image that the application is going to compute is shown below:

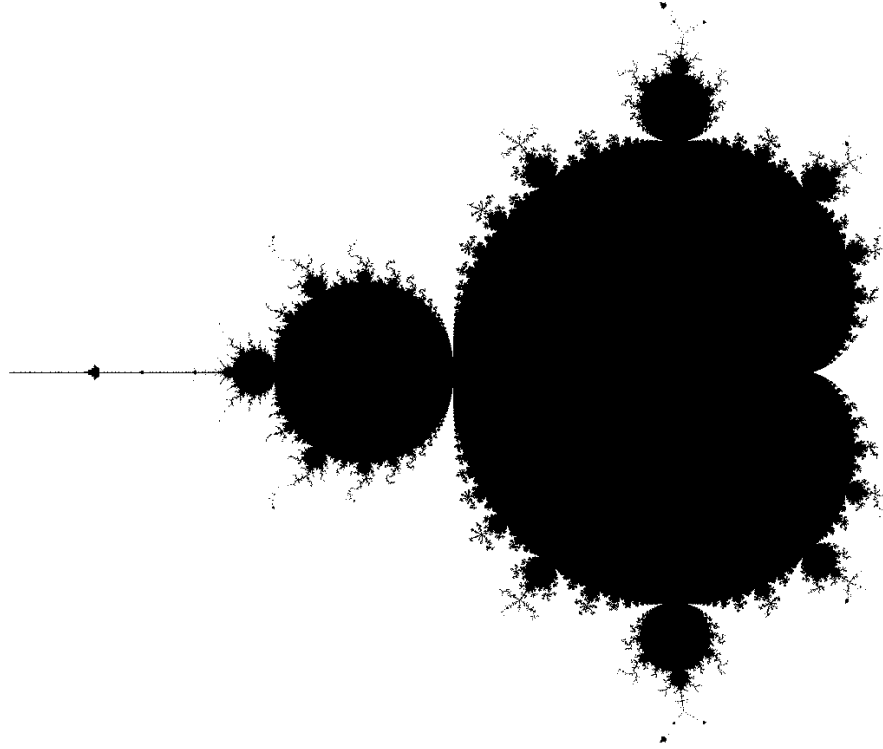


Image 22: Mandelbrot image. Source: <http://home.olympus.net/~dewey/mandelbrot.html>

This is a classic data parallel problem, so the `Parallel.For()` loop can be used to solve this issue and see how well it performs. In the application, there will be different colors for different threads executing a particular portion of the app. The colors will help convey how many tasks were created by the TPL and what portion of the image they compute.

The way this application is architected is that is using the background worker class to do the image computation on a background worker thread, that way the application remains responsive. The background worker class has a report progress event that is used to the UI thread so it can draw the image on the screen. This class also supports cancellation. This was the way developers did parallel programming prior to .NET 4, however, with the .NET framework 4, this way became obsolete. Another pitfall of this method is that there was no way to include more than 1 core in the execution engine, thus all the other cores remain idle. In the first sequential version of the app, is not using all the possible cores on the computer and the CPU usage is not at 100%, its on about 40%, because its using only one

core and the rest are idle. This is due to the fact that the application is using older method for long running and parallel task and its not making use of the new methods in .NET framework 4 and 4.5.

Now, lets see in the code behind for this application and note that most of the code will stay the same in the next parallel version as well, the author will point out which code is different and where the difference is. As part of the class, there are several fields that are instantiated and then used in the application and those fields are:

```
private int _startTime = 0;
private BackgroundWorker _worker;

private double _x; // parameters of Mandelbrot computation:
private double _y;
private double _size;
private int _pixels;

public Mandelbrot(double x, double y, double size, int pixels)
{
    _x = x; // assigning parameters
    _y = y;
    _size = size;
    _pixels = pixels;
}
```

On the code snippet is shown a variable that measures the time in order to check how long it takes for application to execute and draw the image on screen, also there are variables for x and y coordinates and also the size of the image want on the screen and at the end there is the distance in pixels. The distance is set as 600 pixels, so for each pixel on the image the program makes a computation and that 600x600. The time is displayed on the screen with the help of this TimeTaken() function which body consist of:

```
//Method that measures time
public double TimeTaken()
{
    int curTime = Environment.TickCount;
    double time = (curTime - _startTime) / 1000.0;

    return time;
}
```

Lets run the application as it is, using the old method and harvest the result:

Just to have a better overview of the whole process, here is how the application looks like with half of its work finished:

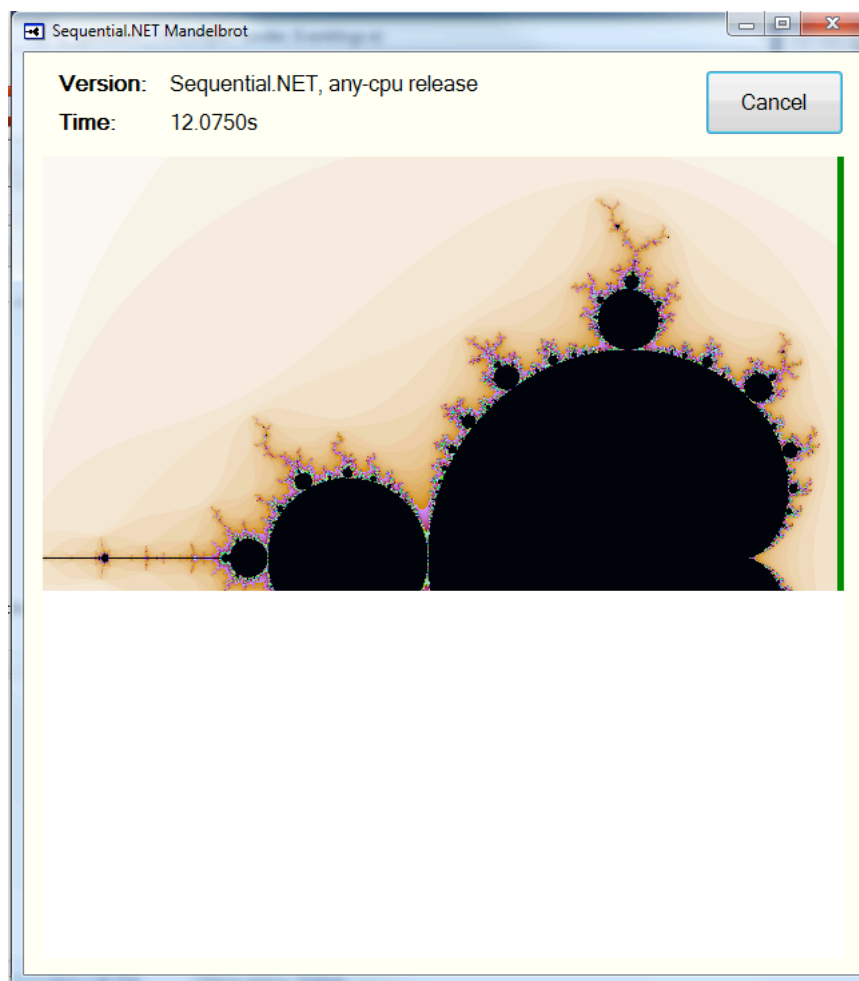


Image 23: Drawing of image in progress (sequential). Source: self-authored

The image is being drawn from top to bottom executing calculations sequentially. Above, it is shown how the application looks like when it's at around half way completed.

Below the final result is shown:

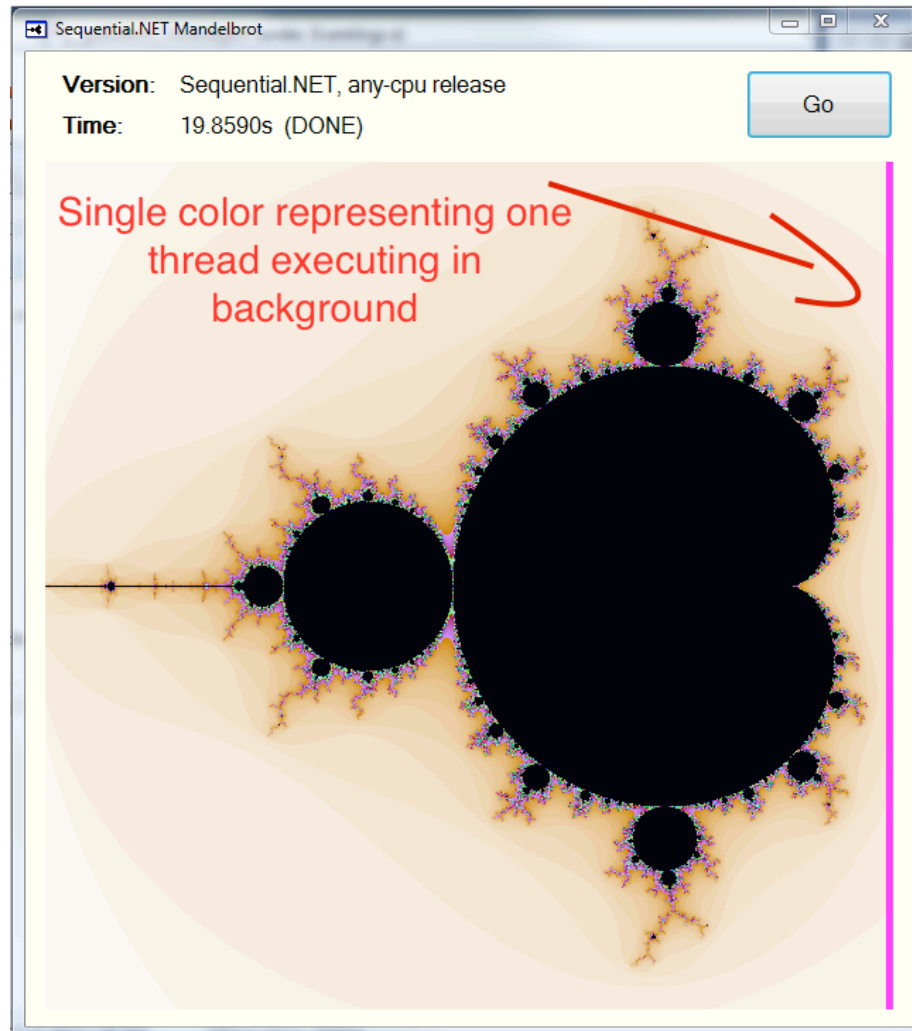


Image 24: Drawing of image completed (sequential). Source: self-authored

The calculations finished in twenty seconds on a quad core architecture using the old sequential method for multithreading, where there is one background process and one process is responsible for the user interface. Now let's see how this can be improved and hopefully made more efficient.

If one looks a bit deeper and sees the code behind for the application one will see that the most of computation is done in two "for" loops that belong to the Calculate() method, the method is taking the sender of the event and the event arguments passed at the "e"

parameter to the function, it then continues to assign the worker background thread as the sender and after that the program starts counting the elapsed time which is then displayed to the application for the user to see. The code is mentioned below:

// The method for Mandelbrot set calculation

```
public void Calculate(Object sender, DoWorkEventArgs e)
{
    _worker = (BackgroundWorker)sender; //draws the image
    _startTime = Environment.TickCount; //keeps track of time
    //for each pixel does the Mandelbrot set calculation
    for (int r = 0; r < _pixels; r++)
    {
        if (_worker.CancellationPending)
            break;
        int[] values = new int[_pixels];
        //creates color based on calculation
        for (int c = 0; c < _pixels; ++c)
        {
            values[c] = MandelbrotColor
                (r, c, _y, _x, _size, _pixels);
        }
        int threadID = Thread.CurrentThread.ManagedThreadId;
        for (int c = _pixels - 5; c < _pixels; c++)
            values[c] = -threadID;
        //worker threads reports the progress if the image
        _worker.ReportProgress
            (r, new object[] { r, values });
    }
    if (_worker.CancellationPending)
        e.Cancel = true;
}
```

In the code snippet above, 99% of the entire computation happening is shown. This is where the dots are drawn by the application. What are shown above are two loops, outer loop and inner loop, and the inner loop for each for iterates along the columns. The image they produce is square and its 600 pixels by 600 pixels. The two loops are processing each pixel on the image. Now, normally when one parallelizes these types of applications, one needs to identify what type of parallelism can be applied, and it turns out that in this case, this application can be “embarrassingly parallel” so its possible to do all the points of the image in parallel. Normally one takes the outer-most loop and makes that parallel. That gives us larger grain tasks to offset the cost of task creation. On the other hand though, consumer wants to try to expose as much parallelism as possible, so what is best to do here is run both loops in parallel using `Parallel.For()` and remember that experimentation is big part of the game in this world, so consumer should try different things and find out what works best.

When redesigned the method to run in parallel will get the below code portion:

```
public void Calculate(Object sender, DoWorkEventArgs e)
{
    _worker = (BackgroundWorker)sender;
    _startTime = Environment.TickCount;
    Parallel.For(0, _pixels, (r) =>
    {
        if (_worker.CancellationPending)
            return;
        int[] values = new int[_pixels];
        Parallel.For(0, _pixels, c =>
        {
            values[c] = MandelbrotColor(r, c, _y, _x, _size, _pixels);
        });
        int threadID = Thread.CurrentThread.ManagedThreadId;
        for (int c = _pixels - 5; c < _pixels; c++)
            values[c] = -threadID;
    });
}
```



```

        _worker.ReportProgress(r, new object[] { r, values });
    });

    if (_worker.CancellationPending)
        e.Cancel = true;
}

```

As it was mentioned before, the `Parallel.For()` is a static method on a static class that takes three arguments that needs to be passed appropriately. In this case there are two parallel loops created so both the inner and outer loop will run in parallel.

Also, at the beginning of this method there is cancellation code implemented as well, in case the user does not want to continue in calculating the image they can cancel it and all tasks will be cancelled. For this purpose, the `Parallel.For()` method is used and it is firing threads and tasks in parallel depending on how many cores consumer has available on the machine. The application will be run on the same machine. Lets see the results:

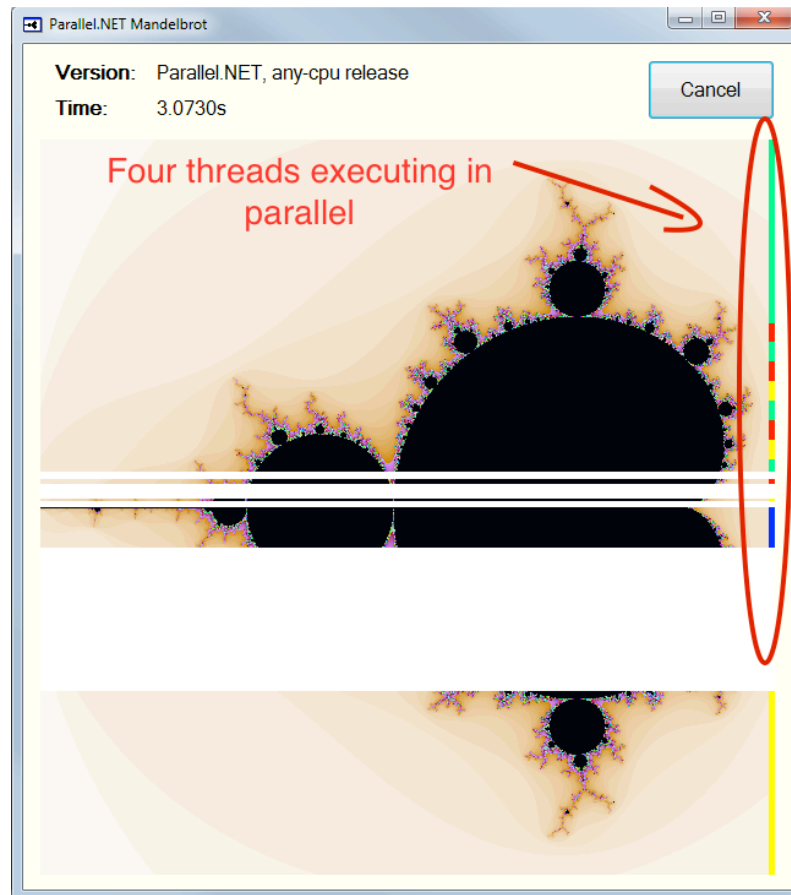


Image 25: Drawing of image in progress (parallel). Source: self-authored

In the above image the state of the application is shown just before finishing. There are four threads executing in parallel making use of all the available resources on the machine. The machine has four cores and all the cores were in usage, none of them were idle as in previous versions of the application.

And here is the final finished computation of the application:

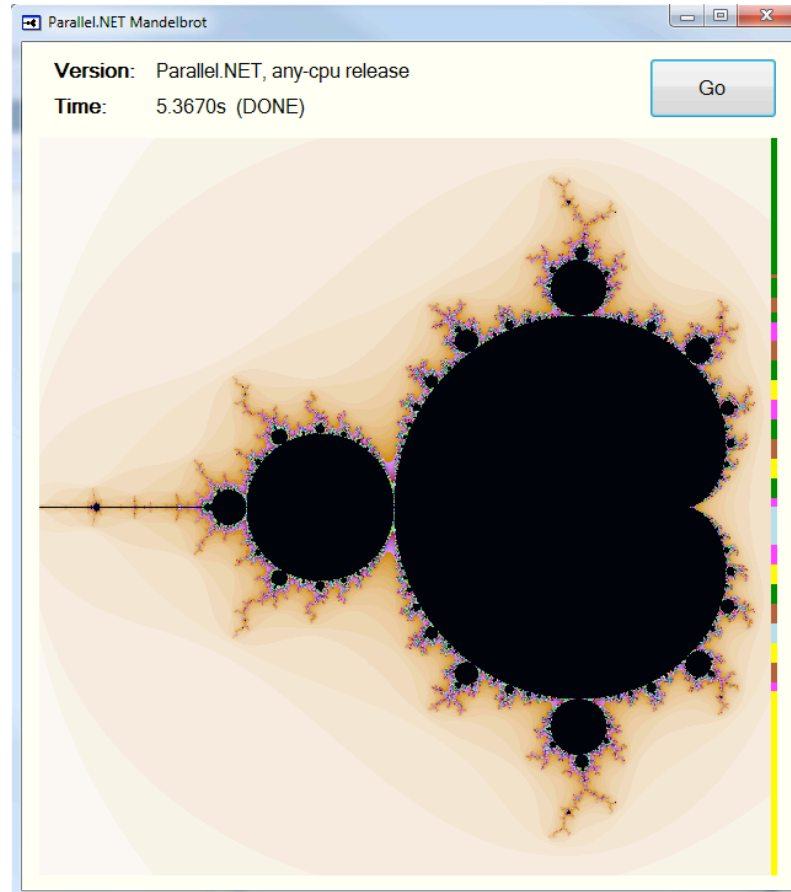


Image 26: Drawing of image completed (parallel). Source: self-authored

From the time stamp on the application, it is shown that it finished in 5.3670 seconds which roughly four times faster than in our initial version of the application. Here is a table with the specifics:

Sequential version		Parallel version	
Speed	Threads executing	Speed	Threads executing
19.9590	2	5.3670	4 (max)

Table 3: Comparison of sequential and parallel versions (speed and threads used)

Also in the parallel version there is cancellation of the tasks implemented in case it's taking too much time or the users of the application decides not to wait for it. After the tasks are cancelled, all the threads are immediately destroyed and returned to the operating system in the thread pool for further usage.

In order to make sure that the calculations are correct and reliable, both versions of the application are ran for one hundred times to make sure that the results are consistent and can be used in practice. Therefore, from the one hundred times the application ran, the application did not crash even once, and that goes both for the sequential version and the parallel version, however in regards to time to run there are significant differences. The full excel spreadsheet can be found on a CD attached to the appendix of this diploma thesis and the summary is shown below, the values are calculated in seconds:

Sequential version summary:

Average	19,60751
Minimum	19,008
Maximum	20,936

Parallel version summary:

Average	5,14415
Minimum	4,836
Maximum	5,897

From the one hundred tests done and the summary stated above it is shown that the average time for the sequential version of the application is 19, 60751 seconds and 19, 008 and 20, 936 for minimum and maximum values respectively. For the parallel version of the application there are much different results, the author got 5, 14415 seconds for average time in seconds and 4,836 and 5, 897 seconds respectively for minimum and maximum values.

4.1. The Mandelbrot application with implemented cancellation

Right now the Mandelbrot application does not use any cancellation features, the only way to cancel the execution of this parallel program is to terminate the application and even then, some of the resources will not be put back to the operating system as it will take some extra time the operating system to realize that the threads are not being used anymore so they are returned to the system. The cancel button appears on the application while running but its implemented using the old way and this cannot cancel multiple tasks at same time. In this section the cancellation support offered by the task parallel library will be implemented, so lets open the Mandelbrot application and start implementing the cancellation feature.

So, first of all the CancellationTokenSource object needs to be added as a private field to the application and the field will be instantiated whenever necessary:

```
private int _startTime = 0;
private BackgroundWorker _worker = null;

private readonly double _x;
private readonly double _y;
private readonly double _size;
private readonly int _pixels;
private CancellationTokenSource _cts;
```

After adding this, the code that was actually doing this extensive computation and implement the cancellation feature needs to be updated.

First consumer instantiates the CancellationTokenSource object and assign it a value, after that a new ParallelOptions class is created that takes in a cancellation token. After that pass in this class to the Parallel.For() loop as additional argument.

This new object that is passed in knows how to cancel a Parallel.For() loop efficiently and effectively releasing all the un-used memory back to the operating system and all the threads that are idle into the threads pool ready for usage by another application or process.

```

public void Calculate(Object sender, DoWorkEventArgs e)
{
    _worker = (BackgroundWorker)sender;
    _startTime = System.Environment.TickCount;
    _cts = new CancellationTokenSource(); //Create cancellation options object
    var options = new ParallelOptions
    {
        //Assign token to parallel options class
        CancellationToken = _cts.Token
    };
    try
    {
        Parallel.For(0, _pixels, options, (r) => //pass in options as parameter
        {
            int[] values = new int[_pixels];
            Parallel.For(0, _pixels, (c) =>
            {
                values[c] = MandelbrotColor(r, c, _y, _x, _size, _pixels);
            });
            // .NET thread id:
            int threadID = System.Threading.Thread.CurrentThread.ManagedThreadId;
            for (int c = _pixels - 5; c < _pixels; c++)
                values[c] = -threadID;
            _worker.ReportProgress(r, new object[] { r, values });
        });
    }
    catch (OperationCanceledException)
    {
        // tell background worker that work was cancelled:
        e.Cancel = true;
    }
}

```

As it was shown in the theoretical part with exception handling, it's enough to wrap the unsafe code in a try-catch block to catch any unhandled exception and this is what it's in this application. In case the application crashes, the task parallel library is designed to release any un-used memory back to the operating system without any input from the user. The only thing that needs to be done in the catch block is to tell the background worker thread that is reporting for the progress of this application that is cancelled and no additional work is required by it.

The cancel button is implemented by the below code snippet:

```
// Call to cancel operation:  
public void CancelCalculate()  
{  
    _cts.Cancel();  
}
```

The above code snippet simply calls the `Cancel()` method of the `CancellationTokenSource` object that knows how to cancel the work in the background for all the threads and release back resources to the operating system.

Now, if the application is ran one more time and try to cancel it we will see the correct behavior. Note this was not working in the previous versions.



Image 27: Execution cancelled. Source: self-authored

4.2. The stock history application

The second application as part of the practical part the author will present an application that requires creating code using façade tasks. An application that processes historical stock data will be presented, which then computes some statistics on the results such as minimum stock price, maximum stock price, average price, standard deviation and standard error. This is a simple application with just two main components. Program.cs file represents the front end and performs the computations and the back-end DownloadData.cs retrieves live data from the Internet. Now, right now the front end performs the computations sequentially, one after the other while the backend makes three asynchronous calls to msn, NASDAQ and yahoo and then it uses the data from the first

site that responds. The application is a simple console based application that has no user interface and the code behind this application is also pretty simple. Let's start investigating what is in the code behind. First we have the method that processes the command line arguments that are entered by the user, the code looks like this:

```
static void Main(string[] args)
{
    String version, platform, symbol;
    int numYearsOfHistory;

    ProcessCmdLineArgs(args, out version, out platform, out symbol, out
numYearsOfHistory);

    ProcessStockSymbol(symbol, numYearsOfHistory);

    Console.WriteLine();
    Console.WriteLine("** Done **");
    Console.WriteLine();

    Console.Write("\n\nPress a key to exit...");
    Console.ReadKey();
}
```

The main method is written above, and this method basically processes command line arguments and then processes the stock symbol. The method called `ProcessStockSymbol()` and this is the method that computes the minimum, maximum and average from the stock prices as shown below:

```
StockData data = DownloadData.GetHistoricalData(symbol, numYearsOfHistory);

int n = data.Prices.Count;
```



```
decimal min = data.Prices.Min();
decimal max = data.Prices.Max();
decimal avg = data.Prices.Average();
```

The `GetDataFromInternet()` method returns a `StockData` object that contains the information that one needs from the specific company. The method is written below:

```
private static StockData GetDataFromInternet(string symbol, int numYearsOfHistory)
{
    try
    {
        StockData yahoo = GetDataFromYahoo(symbol, numYearsOfHistory);
        return yahoo;
    }
    catch { /*ignore, try next one*/ }

    try
    {
        StockData nasdaq = GetDataFromNasdaq(symbol, numYearsOfHistory);
        return nasdaq;
    }
    catch { /*ignore, try next one*/ }

    try
    {
        StockData msn = GetDataFromMsn(symbol, numYearsOfHistory);
        return msn;
    }
    catch { /*ignore, try next one*/ }

    // all failed:
    throw new ApplicationException("all web sites failed");
}
```

The method takes two parameters, one is the stock symbol that the user enters and second one is the number of years in past they need the data for. In case they don't specify this, the default number of years is 10. After that the thread enters the method and sequentially is trying to access yahoo, msn and NASDAQ websites to retrieve live data from the internet. In case the first site fails, the error is ignored and it goes to the second one and if this one fails as well it goes to third one. In case all sites fails to return the information the application throws an error to the user informing them that all web sites have failed. For completeness, this is the stock data class that is returned from the websites:

```
class StockData
{
    public string DataSource { get; private set; }
    public List<decimal> Prices { get; private set; }

    public StockData(string dataSource, List<decimal> prices)
    {
        this.DataSource = dataSource;
        this.Prices = prices;
    }
}
```

Lets run the application and see how it behaves.

The application starts with a prompt screen asking the user to input a stock symbol:

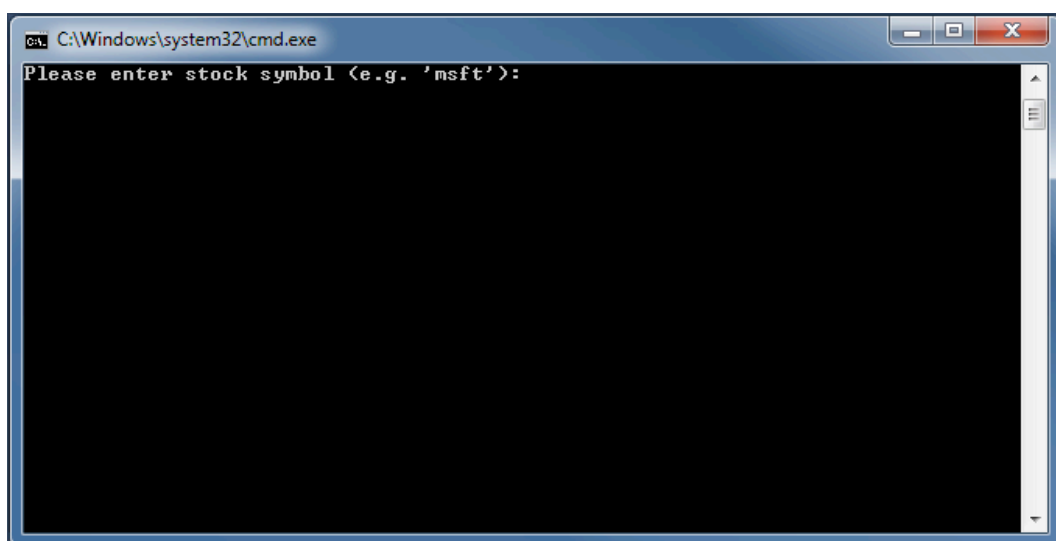
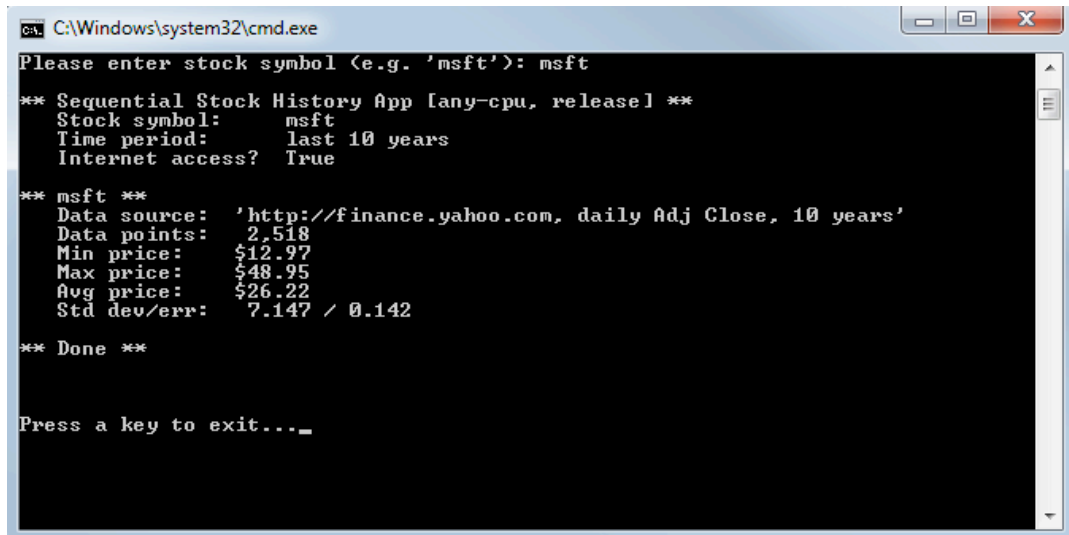


Image 28: Waiting for user input. Source: self-authored

If user provides any stock symbol it will go to three different website, and the first that will respond with the data it will display it to the screen for the user to see as such:



```
C:\Windows\system32\cmd.exe
Please enter stock symbol (e.g. 'msft'): msft
** Sequential Stock History App [any-cpu, release] **
Stock symbol:      msft
Time period:      last 10 years
Internet access?  True

** msft **
Data source:      'http://finance.yahoo.com, daily Adj Close, 10 years'
Data points:      2,518
Min price:        $12.97
Max price:        $48.95
Avg price:        $26.22
Std dev/err:      7.147 / 0.142

** Done **

Press a key to exit..._
```

Image 29: Response from website and calculations done. Source: self-authored

As it is the application is performing OK at best. The problem with this application is that while the user is waiting for the data to be downloaded from the Internet and the application to connect to the Internet the thread that is executing the code is actually sleeping.

This could be a problem for applications that are used by many users since there is only certain number of threads available in the thread pool and one cannot afford threads to be sleeping while waiting for some result form the server. Now the application will be modified to make it more responsive and have better performance. However, this is a different kind of application and the solution that was done in the Mandelbrot application will not apply here, because here there is no complex computation, just the problem with threads being idle and waiting for the data to come from the internet. We want these threads to be doing some other work and then continue with the results that are returned from the Internet.

First, lets create separate tasks for the average, minimum and maximum computations:

```
StockData data = DownloadData.GetDataFromInternet(symbol, numYearsOfHistory);
```

```
int n = data.Prices.Count;

// Create tasks for each of calculation
Task<decimal> min = Task.Factory.StartNew(() => data.Prices.Min());

Task<decimal> max = Task.Factory.StartNew(() => data.Prices.Max());

Task<decimal> avg = Task.Factory.StartNew(() => data.Prices.Average());
```

Now, since there are tasks created for every computation users can utilize every physical core on the machine and the results from these tasks are stored in the Result property and displaying the result to the user is simply calling min.Result for instance:

```
// Display values to screen for user to see
Console.WriteLine(" Min price:  {0:C}", min.Result);
Console.WriteLine(" Max price:  {0:C}", max.Result);
Console.WriteLine(" Avg price:  {0:C}", avg.Result);
```

In the previous versions only one core was used for all the computations and now if machine has four physical cores all four cores will be able to work and none of them stay idle in the background. The benefit of this approach is not speed in this case, but in case lets say one hundred users are connecting to the application at same time, all the cores will be active and there will be no delay or lag on the application. In the previous version if more users were using the application they will experience lag and slow response. However, this is not done yet. In the previous version the application was going through three websites and waiting for response from the first one, in case that one fails it waits from second etc. Now, the first website to response is what is displayed to the user.

This will improve the performance, utilize all cores on the machine and make sure that none of the threads are idle in waiting or sleeping in the background. Now, lets look at the final version of the application where the application is executing in parallel, making use of all threads and is able to scale to the number of users that are using it. The previously sequential method for downloading data from the Internet is now running in parallel and its

displaying to the user that first website that returns a value. The code for the parallel version of the method is less than the one for sequential version, its more readable and easier to understand and also has better performance:

```
public static StockData GetDataFromInternet(string symbol, int numYearsOfHistory)
{
    // Create tasks for each of the downloads
    Task<StockData> yahoo = Task.Factory.StartNew( () =>
        GetDataFromYahoo(symbol, numYearsOfHistory));

    Task<StockData> nasdaq = Task.Factory.StartNew(() =>
        GetDataFromNasdaq(symbol, numYearsOfHistory));

    Task<StockData> msn = Task.Factory.StartNew(() => GetDataFromNasdaq(symbol,
        numYearsOfHistory));

    Task<StockData>[] tasks = { yahoo, nasdaq, msn };
    int index = Task.WaitAny(tasks);
    Task<StockData> winner = tasks[index];
    return winner.Result;
}
```

The application is connecting to the Internet in parallel and its waiting on the first task to finish, and this is done depending on which website will first return result. After that consumer waits for any of the tasks to finish and the first one to finish is the winner. The result of the winner is returned to the caller and displayed to the user. In conclusions for this application, best pattern used here is the speculative execution, since as we've seen earlier, this pattern is best when consumers have multiple sources of data and the first source that return a value is the winner and it's the one they use, in this case the first website that returns data that they need is the one that it is displayed to the user.

This is general used when calling web services. The benefits from this shown in the below table:

Sequential version		Parallel version	
Data source	Threads executing	Data source	Threads executing
1	1	3	3

Table 4: Comparison of sequential and parallel versions (speed and threads used)

The sequential version of the application as it is shown on the table above is using only one thread for execution. The execution is done from top to bottom and when it comes to downloading data from the internet its waiting on every single source to return data, if a specific source does not return data it moves to the other one and then to third one. In case none of them return value, exception is thrown. In the improved version where the task parallel library is used, the speculative pattern is sued where consumers fire separate tasks for each of the data sources and when any if them is finish they return the value to the caller and cancel other tasks. Moreover, the calculations are also done in parallel (the minimum, maximum and average) so we do not have any threads idle. This way, more users are able to connect to the application simultaneously.

If the application is modified to download the data from all the three sources that are listed there and wait for all, the sequential version has to download all the data one by one and wait until each of them is finished. The parallel version on the other hand will trigger the download in all three threads at once and run the download in parallel. After one hundred runs of the application versions both sequential and parallel, a data is gathered that can help the author make conclusions. The full data for the tests can be found on the CD attached to the appendix of this thesis and the summary is listed below:

Sequential summary

Average	6,23107
Minimum	5,827
Maximum	6,687

Parallel summary

Average	2,22679
Minimum	1,901
Maximum	2,499

Once again on the summary it is shown that the parallel version is almost three times faster and more efficient than the sequential version, and in this type of application there was no expensive computation but this application was connecting to the Internet and it was download data.

4.3. Comparisons between two applications

As we've seen from the Mandelbrot application, the computation that this app is doing is more computationally expensive and the best pattern to use is the embarrassingly parallel, where consumer fires off tasks for each CPU core and they make sure the core is used in maximum. All four cores of the CPU were running at 100% when optimized for best performance whereas with only two cores as it was in the old version there was CPU usage of 40%. The history stock application is slightly different. There are no expensive computations here but still there is some time lag in downloading the data from the Internet. This is known as I/O operation is when there are no multiple threads involved the application is not responsive and the single thread that is doing all the work is actually sleeping when the data is being downloaded from the Internet. This issue is solved by creating task for each of the downloads and make sure the result is only displayed to the user form the first task that is finished.

5. Conclusions

Throughout this diploma thesis the author has explained about the new task parallel library in .NET framework 4 and 4.5, the author has also compiled a theory based on newly released material in form of books and articles on the internet. As it was mentioned and seen the task parallel library or the TPL has been around in the .NET framework since version 4 and its design to make it easy for us to work with threads and asynchronous operation so consumers of the TPL can run work in parallel or asynchronously and not to worry about partitioning work and scheduling threads and some of the low level details that they had to worry about in the past.

The task parallel library is both good and provides high level abstractions for parallel programming and also for asynchronous programming and as it was mentioned in the theoretical part these two notions are related to each other but they are different and the applications that the author built during the practical part and the theory explained this. From the task parallel library it was explained how to create a task that represents a unit of work in the .NET framework, how one can start a task for that unit of work to start executing, also it was elaborated on how to create and start a task at the same time without making an explicit call to the Start() method.

On top of that, the author explained how a user can cancel a long-running task in case its taking too long with the support of the task parallel library and in case a task is cancelled, the thread that was assigned to that task is return to the thread pool and is ready for some additional work that will be assigned by the operating system. The author also elaborated on exception handling and saw that there is not much difference in this area when it comes to synchronous code, the easiest way is to wrap the un-safe code into a try-catch block and if something goes wrong the exception will be caught and handled gracefully. Since, it is not guaranteed for the task to start immediately when the user requests, there is a way to specify a priority to a task saying that this task has high priority and the task scheduler will be able to start that particular task first and make all the others wait. In the part where the theory was presented, we have seen the best design techniques to write parallel applicants that are responsive and not hanging or crashing the user interface. Moreover, the author looked into the most common types of parallelism and also showed implementation of

some of them in the practical part. Some built in data structures as presented as well that allow concurrent work and are thread safe and when talking about thread safe operations and data structures, its important to note that there are some dangers of parallel programming and dangers of concurrent work. A developer must be very careful in implementing parallel code since quite few subtle bugs can be introduced in the code and some of the dangers and the bugs were also presented in this thesis. Moreover, the author explained when its good idea to use the task parallel library and when its not and also when it can help and when it cannot, in fact in some cases it can hinder performance.

In the practical part of the diploma thesis, two very different applications were presented that are making use of the parallel and asynchronous programming model. The first application that it was mentioned was the Mandelbrot set application that was drawing the Mandelbrot image on the screen for the user to see. The application is computationally very expensive and as it was shown before without using the task parallel library the application was running on a single core and it was finished with drawing the image in about twenty seconds and after doing small changes and made the application run in parallel it was noticed significant improvement on performance for just few lines of extra code written. The application was using 100% of the CPU power and finished in five seconds whereas in the sequential version it was using only 30% of the CPU power. The measurements are done directly in the application using some of the features of the .NET framework. Also with experimenting the author has ascertained that the best pattern for parallelism in this application was the “Embarrassingly parallel” where the application is firing off tasks for each of the loops and the operating system is assigning threads to those tasks accordingly. In the second application, the history stock application, the same methods for parallelism cannot be employed there and since the computation that the application is doing is not expensive but for best performance it still required creation of additional tasks. Note that in case the .NET framework thinks that no thread is required for particular work it will not assign a new thread, that’s the beauty in the task parallel library, it takes responsibility of the low level code that consumers do not need to write. In the history stock application we created separate tasks for the three methods that were downloading data from the Internet and the first one to finish was the winner. The results from the winner task were then displayed to the user to see. In the previous versions of the application there was not any type of parallelism and the call to the websites was a

blocking call where the user had to wait and the thread had to sleep. The previous version did not support multiple users and was not making use of all the cores on the machine that the user might have and even though the user might have a single core, the .NET framework will coordinate the work in such a way that no extra threads are created and no unnecessary work is being done in the background.

Therefore, bottom line is that the task parallel library can be implemented to multicore but also in single core architectures. In the multicore architectures consumers can take advantage of the parallel programming and on a single core machines, the asynchronous programming model can be used where a thread is not sitting idle waiting for an I/O operation to complete but is listening for new requests and doing some other work.

6. Bibliography

Watson, B. (2014). *Writing High-Performance .NET code*. Norfolk, USA: Ben Watson.

Campbell, C., Johnson, R., Miller, A., & Toub, S. (2010). *Parallel Programming with Microsoft® .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Redmond, USA: Microsoft Press.

Cleary, S. (2014). *Concurrency in C# Cookbook*. Sebastopol, USA: O'Reilly Media.

Esposito, D., & Saltarello, A. (2014). *Microsoft .NET - Architecting Applications for the Enterprise*. Redmond, USA: Microsoft Press.

Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming*. Burlington: Morgan Kaufmann.

Mattson, T., Sanders, B., & Massingill, B. (2013). *Patterns for Parallel Programming* (1st Edition ed.). Boston, USA: Addison-Wesley Professional.

Pacheco, P. (2011). *An Introduction to Parallel Programming* (1st Edition ed.).

Burlington, Massachusetts, USA: Morgan Kaufmann.

Sharp, J. (2013). *Microsoft Visual C#*. Redmond, USA: Microsoft Press.

Skeet, J. (2013). *C# in Depth* (3rd Edition ed.). Greenwich, USA: Manning Publications.

Razdan, S. (2014). *Fundamentals of Parallel Computing*. Oxford, UK: Alpha Science International Ltd.