Czech University of Life Sciences

Faculty of Economics and Management

Department of Information Engineering

Bachelor thesis

Multi-threaded programming

Author: Oleksii Lahushchenko

Supervisor: Ing. Petr Hanzlík, Ph.D.

BACHELOR THESIS ASSIGNMENT

Oleksii Lahushchenko

Systems Engineering and Informatics
Informatics

Thesis title

Multi-threaded programming

Objectives of thesis

The objective of this thesis is to find the use cases when code written to be executed by multiple threads will improve the performance of an application, and hence may replace the classic code that is written for single-threaded execution. The supplementary goal is to research what are the disadvantages of multi-threaded programming, and when this approach should be avoided.

Methodology

The methodology of the thesis is based on analysis of technical and scientific sources focusing on multi-threaded and single-threaded programming. Based on the synthesis of the gained knowledge, a series of experimental programs will be written in C# and the differences between the single-threaded and multi-threaded implementation described and measured in terms of performance.

The proposed extent of the thesis

40-50 pages

Keywords

Multi-threaded programming, C#,

Recommended information sources

ALBAHARI, Joseph and Ben ALBAHARI, 2017. C# 7.0 in a Nutshell: The Definitive Reference. O’Reilly Media. ISBN 9781491987650.

DEITEL, Paul J. and Harvey DEITEL, 2016. C# 6 for Programmers. 6th Edition. Prentice Hall. ISBN 9780134596327.

Expected date of thesis defence

2020/21 SS – FEM

The Bachelor Thesis Supervisor

Ing. Petr Hanzlík, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 19. 11. 2020	Electronic approval: 19. 11. 2020
Ing. Martin Pelikán, Ph.D.	**Ing. Martin Pelikán, Ph.D.**
Head of department	Dean

Prague on 10. 03. 2021

Declaration

I declare that I have worked on my bachelor thesis titled "Multi-threaded Programming" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break copyrights of any their person.

In Prague on date of submission ____________________________

Acknowledgement

I would like to thank Petr Hanzlík, for their advice and support during my work on this thesis.

Multi-threaded programming

Abstract

This thesis describes different multithreading techniques and how they are realized in modern environment, specifically C# and .NET. The goal of this thesis is to research how a modern application could run on multiple threads, what are the tools that are necessary for creating a multithreaded application and what are the benefits that threading can provide. These principles are demonstrated on a C# application, which provides a benchmark for selected implementation scenarios on a set of complex tasks.

Keywords: Threading, C#, .NET, Asynchronous programming, Parallel programming, CPU, Multi-core processor, async, await, Task.

Vicevlaknové programování.

Abstrakt

Tato bakalářská práce popisují různé techniky vícevláknoveho programováni a jejich použití v moderním aplikačním prostředí, konkrétně v C# i. NET. Cílem práce je prozkoumat fungování moderních vícevláknových aplikací, jaké nástroje jsou nezbytné pro vývoj takových aplikací, a popis výhod, která poskytují vicevlaknové programování. Tyto principy jsou demonstrovány na aplikaci, která srovnává vybrané implementace pomocí množiny komplexních úloh.

Klíčová slova: Vlákno, C#, .NET, Asynchronné programování, Souběžné programování, CPU, Vícejádrový procesor, async, await, Task.

Table of contents

List of figures

Introduction

Nowadays the term programming has a lot of meanings. To know how to program does not mean only to have the knowledge of programming language and its syntax, but also the ability to use a lot of other important tools, such as frameworks, software, hardware, etc. Among such tools are the different program execution methods. Apart from the “classical” line-after-line code execution, there exist another way to execute the code – multithreaded execution.

The objective of this thesis is to determine what is multithreaded programming and where it is appropriate to use it. It is generally assumed, especially among less experienced developers, that creating multithreaded application is an extremely challenging task which requires thorough knowledge of the programming language and frameworks which are used for threading. By writing this thesis I want to investigate how threading is implemented in a modern .NET framework. Also, I want to research if there are cases when code written to be executed by multiple threads will improve the performance and user experience of the application, and hence may replace the classic code that is written for a single threaded execution.

The first part of the thesis consists of theoretical background of multithreaded programming. Such terms as CPU threads, implementation of multithreaded programming in C# and .NET framework is described in this part. The second part consists of practical showcase of the application, designed to complete same tasks using different execution methods, and compare the results of the execution.

Objectives and Methodology

Objectives

The objective of this thesis is to explore different multithreading techniques and how they affect an application. In this thesis I want to discuss how multithreading is implemented in modern development environments and frameworks, what are the benefits of running application on multiple threads and is there any practical benefits and advantages of multithreaded application over the single threaded application.

Methodology

This thesis is focused on implementation of threading in C# programming language and .NET framework. I used multiple books and Microsoft documentation website in order to research how threading is implemented in .NET and to describe it in multiple sections in theoretical part of the thesis.

Practical part of the thesis is focused on how the theoretical concepts of threading could be implemented in a real-world application and how exactly they affect the application. For practical part I created a WPF application using Visual Studio 2019 along with .NET framework for implementing multithreading in the application.

The application consists of 4 execution modes – Synchronous execution, Asynchronous execution, Parallel execution and Asynchronous + Parallel execution. When a user clicks the respective to the execution mode button, the application starts its execution in the selected execution mode. and create multiple web requests. During each cycle of execution, the application is performing 3 different tests which are designed to load CPU and create multiple web requests. Additionally, the application allows to cancel current execution cycle and display the results of all executions.

Part 1 – Theoretical background

1.1. Introduction to CPU organization

Before getting to multithreaded programming, it is important to establish an understanding of what is going on "behind the scenes". Taking a closer look at hardware implementation of threads would help us later to understand how the multithreading actually works, when taking a look at the multithreaded programming itself.

CPU (Central processing unit) is a "brain" of computer. It is responsible for performing computational work, which basically means that CPU runs the programs, performs calculations, reads instructions and data from memory, communicates with other parts of hardware using system bus and gives commands to other hardware parts of the computer. CPU itself consists of multiple sub-components which are:

- ALU (Arithmetic Logic Unit) – this component is responsible for all mathematical, logical and decisional processes within the CPU. Inside the CPU there is also an FPU (Floating Point Unit) which is not exactly a standalone unit since the purpose of this unit is to support ALU and perform tasks with floating point.
- Registers – these are the storage components inside of the CPU. Registers store small amounts of data that is used in processes that are currently being executed. Since CPU needs to constantly perform tasks with this data, it is necessarily that the data is being delivered to ALU at the maximum speed possible, that is why registers are needed.
- Control unit – this is the unit that gives commands to all other units. Instructions first come to the control unit, and then it distributes the tasks between all other components.
- Cache – apart from registers, modern CPUs also have another unit to store data, and that is cache. Cache stores data fetched from the operating memory, in order to be accessed as fast as possible by other components of CPU. Only the most necessary data about the upcoming processes are stored inside of cache, since even the modern processors have only about 15 megabytes of cache memory.
- Front-side-bus – as was already mentioned before, CPU receives data and instructions from operating memory (RAM) where this data is being stored during the current session. Front-side bus is responsible for transferring the data from RAM and processed results into RAM. [1]

[1] Das, D. (2018, January 1). *COMPONENTS OF CPU AND THEIR FUNCTIONS | DIAGRAM*.

1.2. CPU cores

Aside from the basic components described in the previous section, CPU also contains cores. They play vital role in the multi-threaded execution of the programs - basically multi-threading is possible only because there are multi-core processors. The first thing that needs to be established is what exactly the core is.

Core consists of components mentioned earlier – ALU, FPU, Registers, Control unit and lower-level cache. All those components combined form a core. So why is it even needed to combine components and make cores?

Now it all comes back to the Moore's law, according to which the number of transistors in a dense integrated circuit doubles about every two years. Consequently, it means that the size of transistors becomes smaller, larger amounts of transistors could be put on the circuit and hence the performance of the CPU is increased. However, the increase in number of transistors comes with its drawbacks - mainly overheating, also single CPU unit consumes large amounts of power and is bad at multitasking. When CPU manufacturers faced this problem in early 2000s, they had 2 options – either place multiple CPUs on motherboard or pack a set of basic CPU components into single core and put multiple cores inside of the CPU. [2]

Multiple CPUs method proved to be inefficient, since in this case the overall speed of the system would be limited by the speed of the front-side-bus, that is why the majority of modern motherboards, apart from certain server-grade models, supports only one CPU, but each CPU nowadays have at least 4 cores or more. [3]

1.3. Principle of operation of the cores. Hyper-Threading.

As was described in the previous section, CPU cores quickly became an important part of any CPU and an industry standard. What is important for this thesis, is that with multiple cores it became possible to truly execute multiple tasks and applications at the same time. To understand how multiple cores are executing tasks in multiple threads, let's first take a look at how multiple cores actually work together. In the beginning, all instructions are stored in the operating memory. When CPU needs to execute an instruction, it starts the instruction execution cycle. This cycle can be separated into 6 steps: [4]

1. The memory address where the first instruction is located is copied to the program counter.

2 Hope, C. (2019, July 10). *Computer processor history*.
3 White, A. (2019, August 19). *What is Core in Computer? Here is What it Does*.
4 Plantz, R. G. *Introduction to Computer Organization*.

2. CPU sends the instruction address that is stored in the program counter to the operating memory via the bus.
3. Memory responds by sending the state of bits at the requested memory location back to CPU, which then copies it into its own registers and cache.
4. The instruction pointer that is stored inside the program counter is incremented, in order to get new instructions from memory in the next cycle.
5. CPU executes the instructions that came from RAM and send the results back to RAM or gives instructions to other hardware parts.
6. Return to the step 2 with the incremented program counter.

Each core functions as a separate CPU, so every one of them could perform execution cycle described above. This approach provides many benefits, the most obvious of which is that there is no longer need for a single CPU to execute all instructions one by one. Instead, now there are multiple cores, the instructions could be distributed between the cores which would not only speed up the execution process, but also provide us with the opportunity to execute multiple instructions simultaneously.

In addition to just adding more cores in order to speed up the system there's also a technology that is called Hyper-Threading, that allows us to increase the computational capabilities of the system even further. Hyper-Threading is a technology that was first introduced by Intel back in 2002 when they released their Pentium 4 HT. The Pentium 4 at that time featured only a single core, so as was mentioned before, it suffered from the drawbacks of a single core CPU. However, Intel implemented a technology that would allow the single core to be registered as multiple logical cores by the operating system. Technically it's "cheating", since the processor only pretends to have multiple cores, while in reality CPU uses its own logic to distribute the tasks between the logical cores in order to speed up the execution. This approach is obviously nowhere near as good as having true multiple cores, but it is still used in modern processors as a nice bonus. For example, AMD's Ryzen 7 3700 features 8 physical cores that are represented as 16 logical cores via AMD's analog of hyper-threading, called Simultaneous multithreading. [5]

[5] Hoffman, C. (2018, October 12). *CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained*.

Part 2 – Implementation of multithreading in C#

2.1 Introduction to concurrency

Concurrency is a very important aspect of the modern applications. It is a concept of executing multiple tasks at the same time. It is extremely beneficial to use concurrency in any kinds of applications. End-user applications may execute some calculations or any other time-consuming work, while still being able to maintain the UI of the application and respond to the user. Server applications use concurrency to respond to one request while finishing another request.

Concurrency itself is not a tool that allows computer to execute multiple tasks simultaneously. It is rather a set of concepts which should be implemented to achieve a structure that could potentially benefit from parallelism. Parallelism is a term that means doing multiple things at the same time, while concurrency is about dealing with multiple things at once.

There are multiple tools in C# that developers could use in order to write a concurrent application, for example multithreading, parallel processing, asynchronous programming, reactive programming. These concepts will be revealed in detail in further sections.

As was mentioned earlier, the main goal of concurrency is to build a proper structure, that means the implementation of concurrent programming techniques. When done correctly such applications may benefit from parallelism. For example, an application that would implement multithreading and asynchronous programming and would be executed on a computer with multiple cores, would most likely perform better than an application where tasks are being executed one by one. However, if such application would run on a machine with only 1 core, the benefits of concurrency would be less significant, since it is physically not possible to do multiple tasks simultaneously with only 1 core. [6]

2.2 Threading

2.2.1 Creating a new thread

A thread is an execution pass that can proceed independently of others. Each thread runs within the operating system, which provides an isolated environment for the thread within the program where it runs. A process is called single-threaded if it has only one execution thread. In such case computer executes all instructions one by one. With a multithreaded program, multiple threads run in the same process and share the same execution environment (memory).

[6] Cleary, S. (2019). *Concurrency in C# Cookbook* (Second ed.).

In C# each program starts as a single-threaded application with the “main” thread. New threads could be created as follows:

```
using System;
//In order to create new threads it is necessary import the following
namespace
using System.Threading;

namespace test
{
    class Program
    {
        static void Main()
        {
            //Initializing a new thread
            Thread t = new Thread(WriteInConsole);
            //Starting the execution of the new thread
            t.Start();

            //While the new thread is running, do some work in the main
thread
            for (int i = 0; i < 10; i++)
                Console.Write("1");
        }

        static void WriteInConsole()
        {
            for (int i = 0; i < 10; i++)
                Console.Write("0");
        }
    }
}
```

The typical output of such program would be: 11000110000011110011.

The main thread creates a new thread by creating a new `Thread` object, and assigns it to the variable `t`. The Thread class constructor can take either of two delegates, depending on whether the method that would be executed by the thread can take the arguments or not. [7]

- If the method that is being executed in the new thread takes no arguments, just like in the example above, `ThreadStart` delegate should be passed to the constructor of the Thread. It has the following signature:

  ```
  public delegate void ThreadStart()
  ```

- If the method takes an argument, `ParameterizedThreadStart` delegate that has the following signature should be passed:

  ```
  public delegate void ParameterizedThreadStart(object obj)
  ```

[7] Microsoft Corporation. (2020). *Thread Class Documentation*.

Using ParameterizedThreadStart delegate an argument could be passed to the method before executing it on a new thread, for example:

```
static void Main()
        {
            Thread t = new Thread(WriteInConsole);
            t.Start(2);                            //pass the argument to
the method

...

static void WriteInConsole(object amtOfIterations)
        {
            for (int i = 0; i < (int)amtOfIterations; i++)
                Console.Write("0");
        }
```

In this case 0 would be printed out twice and the output would be: 111111100111

Another technique to pass the argument to the method is lambda expression:

```
Thread t = new Thread(() => WriteInConsole(0,10)); //Lambda expression to
pass mutiple arguments to the method
t.Start();

...

static void WriteInConsole(int a, int b)
        {
            for (int i = a; i < b; i++)
                Console.Write("0");
            Console.Write(";");
        }
```

Passing arguments via lambda expression is generally a better option, since it allows to pass unlimited number of arguments to the method, while Start method accepts only 1 argument.

After the thread is started its IsAlive property returns true, until the point at which the thread ends. A thread ends when the delegate that was passed to the constructor of the thread finishes its execution. In case of the example above, the new thread ended when the method WriteInConsole was executed. After the thread is ended, it cannot restart. [8]

2.2.2 Threads management

The example above shows the first issue with threading. Without a proper management threads may start to behave unexpectedly. For example, the following expressions are the results of

[8] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

executing the example program several times: 00000000001111111111; 11111111110000000000; 11111000000000011111.

This behavior is caused by the fact that both threads are writing the output to the console simultaneously. The thread is said to be "preempted" at the points at which the execution of the thread interferes with another thread. This term often appears in explaining why something gone wrong with a multithreaded application.

In order to eliminate the unexpected behavior of the application, multiple threads managing techniques can be applied.

`Join` and `Sleep` methods could be used to control and synchronize the threads. `Join` blocks the thread that called this method until the thread represented by the instance finishes its execution or the specified time period elapses.

The following code will always output the same data:

```
static void Main()
        {
            Thread t = new Thread(WriteInConsole);
            t.Start();
            t.Join(); //Blocking the main thread until thread t ends


            for (int i = 0; i < 10; i++)
                Console.Write("1");
            Console.Write(";");
        }

        static void WriteInConsole()
        {
            for (int i = 0; i < 10; i++)
                Console.Write("0");
            Console.Write(";");
        }
```

The output is: 0000000000;1111111111;

While joining the threads, caution should be exercised. It is possible to lock the application forever. For example, calling `Thread.CurrentThread.Join();` in the main thread would block the main thread which would freeze the whole application.

`Sleep` suspends the thread for a specified amount of time. Calling `Thread.Sleep(1000);` In the `WriteInConsole` method would lock the `t` thread for 1 second.

When thread is sleeping or waiting for other thread to end via `Join`, it voluntarily yields its time-slice back to the CPU. Time slice is the amount of time the thread is allowed to use the CPU before operating system suspend the CPU usage and hands the resources over to another thread or process.

`ThreadState` property can output the following values depending on the state of the thread:

- Unstarted – the execution of the thread is not started yet,
- Running – thread is currently running,
- WaitSleepJoin – thread is sleeping or waiting for other thread,
- Stopped – thread has ended.

`ThreadState` property is useful for diagnostics but it is unreliable for synchronization of the threads since the state of the thread might change between testing ThreadState and acting on that information. [9]

2.2.3 Sharing data between threads. Threads safety

The CLR (Common Language Runtime) allocates each thread its own memory stack, so that local variables are stored separately. For example, if there is a method with a local value and that method is called from 2 different threads, each thread would have its own copy of the local value.

Threads may share the data if they are both referencing the same object instance:

```
class Program
    {
        bool done = false;
        static void Main()
        {
            Program p = new Program(); //Creating an instance of the Program
class in order to use it as an object reference
            Thread t = new Thread(p.WriteInConsole);
            p.WriteInConsole();
        }

        void WriteInConsole() //This is an instance method, not a static one
        {
            if (!done)
            {   done = true; Console.WriteLine("Done"); }
        }
    }
```

In this case the value `done` is shared between the threads, so "Done" is printed out only once.

Another way to share data between threads are static values:

[9] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

```
class Program
    {
        static bool done = false;
        static void Main()
        {
            Thread t = new Thread(WriteInConsole);
            WriteInConsole();
        }

        static void WriteInConsole()
        {
            if (!done)
            { done = true; Console.WriteLine("Done"); }
        }
    }
```

Though it is unlikely, it is possible that "Done" could be printed out twice. However, just by switching the order of the statements in the `if` statement,

```
{ Console.WriteLine("Done"); done = true; }
```

the chances of printing out "Done" two times rise dramatically, since one thread could be evaluating the `if` statement exactly at the same time as the other thread is writing into console, before it had a chance to set the value of `done` to true. In order to prevent preemption of threads, threads safety principles should be exercised.

One of such principles are threads locking. C# provides specific statement for this called `lock`. Following is the example of lock statement usage:

```
class Program
    {
        static bool done = false;
        static readonly object threadLock = new object(); //Creating a new
object to use it later in the lock statement
        static void Main()
        {
            Thread t = new Thread(WriteInConsole);
            WriteInConsole();
        }

        static void WriteInConsole()
        {
            lock (threadLock) //Setting up a lock for threads
            {
                if (!done)
                { done = true; Console.WriteLine("Done"); }
            }
        }
    }
```

When a thread executes the lock statement, any other threads that would try to access the code within the lock are going to be blocked. When the thread that activated the lock finishes the execution of the code within the statement, the lock is unlocked and another thread is able to access the code. This ensures that "Done" is printed out only once. The code that is protected in such manner from indeterminacy in multithreaded context is called thread-safe code.

The downside of this approach is easily determined – it is possible to forget to put the locks in all necessary places, especially in a program with multiple thousands of lines of code, since there would be no compile-time errors thrown about the absence of the lock. [10]

2.2.4 Classification of threads

Threads are classified into two groups: foreground and background threads. By default, all new threads are of foreground type. The difference between foreground and background threads is that foreground threads keep the application alive while at least 1 of them is running (for example the "main" thread), while background threads do not. If all foreground threads finish their execution, the application would be shut down and abruptly terminate all background threads.

```
static void Main(string[] args)
        {
            Thread t = new Thread(() => Console.ReadLine());
            if (args.Length > 0)
                t.IsBackground = true;
            Console.WriteLine("Main thread is running");
        }
```

In the example above thread t assumes foreground status, it would keep the application running and waiting for the user input, even if main thread has finished its execution. However, if there are arguments passed to the `Main` method, the application would exit after the main thread is ended, and thread t would be terminated.

While using background threads it is important to specify the timeout for such threads, since if there are background threads that for some reason refuse to finish, the application would not close without having the user to close it manually using Task Manager or kill command in Linux. Foreground threads do not require such treatment. However, it is important to avoid bugs that may cause foreground threads to not end. Application would fail to exit properly if there is a presence of active foreground threads.

Threads also have `Priority` property, which determine how much of execution time is allocated for the specified thread relative to other threads. The priority is specified on the following scale: Lowest, BelowNormal, Normal, AboveNormal, Highest. Assigning priority to threads is relevant when there are multiple threads active. Priority increase works well for threads that execute non-

[10] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

UI processes that require minimal computational work and high latency (ability to quickly respond) in the work they do. However, assigning high priority to a thread that execute processes that require a lot of computational work, such as the UI updating, may starve the other threads and lead to the decrease in performance of the whole computer. [11]

2.2.5 Threads synchronization

During the development of an application that implements multithreading, it is likely that there would be a need for threads synchronization. There are multiple techniques that are used to synchronize threads. Two of such techniques – joining and locking the threads were described in previous sections. Those methods are reliable but have their downsides, mainly threads are not really communicating with each other, they rather wait for the other thread to catch up. In order to resolve this issue C# provides multiple constructs that allow one thread to send a notification to another. It is called signaling. One of the simplest signaling construct is `ManualResetEvent`. The following is the usage example of this event: [12]

[11] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[12] Microsoft Corporation. (2020). *ManualResetEvent Class*.

```
class Program
    {
        //Creating an instance of ManualResetEvent class and passing false
        //to the consturctor to initially disable the signal
        static ManualResetEvent signal = new ManualResetEvent(false);
        static void Main(string[] args)
        {
            for (int i = 0; i <= 2; i++)
            {
                Thread t = new Thread(() =>
                {
                    string name = Thread.CurrentThread.Name;
                    Console.WriteLine("{0} has started", name);

                    //Current thread locks
                    //and awaits the signal
                    signal.WaitOne();

                    Console.WriteLine("{0} has ended", name);
                });

                t.Name = "Thread " + i;
                t.Start();
            }

            Thread.Sleep(500); //Awaiting the initialization of all threads

            Console.WriteLine("all threads have started");

            signal.Set(); //Activating the signal and unlocking all threads

            //Awaiting for all threads
            //to finish their execution
            Thread.Sleep(500);

            Console.WriteLine("all threads have ended");
        }
    }
```

The typical output of this program is:

```
Thread 0 has started
Thread 1 has started
Thread 2 has started
All threads have started
Thread 2 has ended
Thread 0 has ended
Thread 1 has ended
All threads have ended
```

Another simple event that is used for signaling is `AutoResetEvent`. While the signal from `ManualResetEvent` stays active until the developer disables it, `AutoResetEvent` automatically disable the signal after it unlocks a single thread. [13]

2.2.6 Thread pool

When a thread is starting it takes a few hundreds of milliseconds to initialize the thread, allocate space for local variables stack etc. While it may not be an issue for a thread that is planned to run for considerable amount of time, it is possible to overwhelm small operations which run on short-life threads with the overhead from initialization of those threads. It means that the performance of these small operations would suffer from the overhead of each thread initialization. Thread pool allows to prevent such performance issues. It is a pool of pre-created recyclable worker threads that doesn't require as much time for initialization. Thread pool is essential in applications that aim for the best performance and the finest concurrency.

There are a few rules that apply to threads in thread pool:

- It is not possible to set the `Name` property of a thread from the pool, which makes it difficult to debug the application.
- Threads from the pool are always background threads.
- Locking threads from the pool could degrade performance of the application.

In order to run a method on a thread from the pool the following construct could be used:

```
//Access the thread pool via ThredPool class and assign a free worker thread
ThreadPool.QueueUserWorkItem(_ => Console.WriteLine("Hello"));
```

Another way of accessing thread pool is the `Task` class which is the core of asynchronous programming.

Thread pool serves another vital function for modern application– it ensures that temporary excess in compute-bound operations does not cause CPU oversubscription. Oversubscription is the condition of there being more active threads than CPU cores. This would force operating system to time-slice threads (assign to each thread only a short period of time when it is allowed to run). Time-slicing causes severe degrade in performance, since it requires expensive context switching and could invalidate the CPU caches that are essential in delivering performance to modern processors.

[13] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

The CLR avoids oversubscription by queueing tasks to worker threads in the thread pool and then throttling their startup. CLR then tunes the level of concurrency via hill-climbing algorithm, continually adjusting the workload in particular direction. If throughput improves, the direction stays the same, otherwise it reverses. This ensures that it always tracks the most optimal performance curve and preventing further oversubscription. This is why locking the thread from the pool could lead to performance issues – it gives the CLR the false idea that it is loading up the CPU. CLR can compensate the locked threads by injecting new threads in the pool, however, it could introduce latency because the CLR throttles the rate at which it injects new threads. Maintaining hygiene in the thread pool (ensuring that threads are not locked explicitly or by the bug) is particularly relevant for the applications that fully utilize the CPU. [14]

2.3 Asynchronous programming

2.3.1 Introduction to asynchronous programming

Concepts and principles that are described in previous sections are the low level of threading. It is important to understand what is going on "behind the scenes", but modern applications almost never have manual initialization of threads in their source code. The main reason for that is the complexity of manual thread usage – it is necessary to always keep track of threads, lock them, signal to other threads, maintain threads security. And while it is relatively easy to do in simple programs, as program grows, so does its thread usage and when there are hundreds or even thousands of threads running simultaneously, it becomes very hard to manually manage them.

Fortunately, there are multiple programming concepts that simplify and automate threads initialization and management, making the code cleaner and much easier to maintain. One of such concepts is asynchronous programming. The main difference between a synchronous and an asynchronous operation is that the synchronous operation does its work before returning to the caller, while the asynchronous operation can do most of its work after returning to the caller. Majority of operation are synchronous. Examples of synchronous operations are the following functions: `List<T>.Add;` `Console.WriteLine;` `Thread.Sleep;` Asynchronous operations are less common and initiate concurrency because work continiues in parallel to the caller. An example of asyncronous operatin is `Thread.Start;` function.

Asyncronous programming provides the same benefits as the manual threading – for end-user GUI applications it provides the ability to run calculations and other operations in background, while still being responsive to the user. And for the server-side programs asynchronous programming

[14] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

enables scalability. However, in contrast to the manual multithreading, asynchronous programing is a higher level of abstruction and thus provides access to multiple keywords and functions that simplify multithreading.

2.3.2 `Task` class

The `Task` class represents a concurrent operation that might be backed by thread. Tasks are usually executed asynchronously and are one of the fundamental components of the task-based asynchronous pattern. Tasks are compositional – they could be chained together through the use of continuations. They can also use thread pool to lessen startup latency.

The easiest way to start a task is to call a static method `Task.Run();` This method creates a new Task backed by a thread and accepts an Action delegate as follows:

```
Task.Run (() => Console.WriteLine ("Hello World!"));
```

`Task.Run();` returns a Task object that could be used for tracking the execution of the task via its Status property.

Tasks are created as background threads by default. It means that if the application terminates its execution before all tasks are completed, all background threads are abruptly terminated. Sometimes it is necessary to wait for some or for all background threads before application is allowed to terminate. It is possible to synchronize thread that called the task with the asynchronous threads by using the `Task.Wait();` method. If no arguments to the method is passed, it will wait unconditionally until a task completes. There are multiple ways to specify the conditions to stop waiting. The first way is to pass the `Int32` or `TimeSpan` value to the `Task.Wait();` In this case Wait method will block the calling thread until the task finishes or the timeout interval elapses, whichever comes first. Alternative option to interrupt the Wait method is to pass `CancellationToken`, it acts as a boolean flag and if tokens `IsCancellationRequested` property changes to `true` the method throws `OperationCanceledException.` It is also possible to wait for series of executing tasks to complete by using `Task.WaitAll();` method, or for only one task by using `Task.WaitAny();` method. [15]

Task also has a generic subclass called `Task<TResult>` which allows task to return a value. To obtain a return value it is necessary to run a Task with a `Func<TResult>` delegate. The obtained result could later be accessed by the `Result` property. If the Task hasn't yet finished,

[15] Microsoft Corporation. (2020). *Task Class.*

accessing this property will block the current thread until the task finishes. The following is the example of obtaining the result from the task:

```
//Start a task that returns a value
Task<int> task = Task.Run(() =>
{
    Console.WriteLine("Task is running");
    return 1;
});
//Block the current thread if task is not yet finished
int result = task.Result;
Console.WriteLine(result);
```

Another important feature of Tasks is continuation. A continuation is a command to the task to do some other operations when the original operation is completed.

```
//Create a new task
Task<int[]> arrayTask = Task.Run(() =>
{
    int[] array = new int[10];
    for (int i = 0; i < 10; i++)
    {
        array[i] = i;
    }
    return array;
});

//Create a new awaiter object whose OnCompleted method tells
//the task to execute the new delegate once it is finishes the current
operation.
var awaiter = arrayTask.GetAwaiter();
awaiter.OnCompleted(() =>
{
    foreach (var i in awaiter.GetResult())
    {
        Console.WriteLine(i);
    }
});
```

The `GetAwaiter();` function returns the awaiter object of the task which is then used to track the progress of the task and give it the new delegate. It is also possible to attach continuation to a task which already completed its work. In this case, continuation will be scheduled to execute right away.

Another way to create continuation is to create a new `Task` and use original, also called "antecedent" tasks values in the delegate of the new task.

```
Task continuationTask = arrayTask.ContinueWith(antecedent =>
{
    int[] result = antecedent.Result;
    foreach (var i in result)
    {
        Console.WriteLine(i);
    }
});
```

The advantage of using `ContinueWith();` method is that it returns a task, which means it is possible to chain multiple continuations one after another.

Apart from manually creating tasks and running them, there is also another way to create tasks with `TaskCompletionSource`. It provides an option to create a task out of any operation that starts and finishes sometime later. `TaskCompletionSource` provides a "slave" task which then could be used to determine when the operation finishes or faults. This is ideal for Input/Output bound work: `TaskCompletionSource` provides all benefits of tasks without the necessity of blocking a thread for the duration of operation. The following is the example of usage of `TaskCompletionSource`:

```
//Create an instance of TaskCompletionSource class
TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();

new Thread(() => { Thread.Sleep(1000); tcs.SetResult(25); }).Start();

//Simulate some I/O work while thread is running
Console.ReadLine();
//Get access to the "slave" task created by TaskCompletionSource
Task<int> task = tcs.Task;
Console.WriteLine(task.Result);
```

First it is necessary to create an instance of `TaskCompletionSource` class. Then by using `SetResult`, `SetException` or `SetCancelled` method the thread that executes the work signals to the slave task and puts it into a completed, faulted or canceled state. Only one of these methods is supposed to be called, calling any of those methods again will throw an exception.

The true power of `TaskCompletionSource` is that it can create and run tasks without even creating a new thread:

```
Task<int> TaskWithoutThread()
{
    var tcs = new TaskCompletionSource<int>();
    //create a new timer to simulate work
    var timer = new System.Timers.Timer(1000) { AutoReset = false };
    timer.Elapsed += delegate { timer.Dispose(); tcs.SetResult(20); };
    timer.Start();
    return tcs.Task;
}
```

```
var awaiter = TaskWithoutThread().GetAwaiter();
awaiter.OnCompleted(() => Console.WriteLine(awaiter.GetResult()));
//Simulate some I/O work while task is running
Console.ReadLine();
```

In preceding example, the task is running in background without creating nor blocking any threads. This means that main thread is engaged only when the task launches and 1 second later when the task continuation starts. All the work in task within this 1 second is running on the background threads, while main thread can handle the I/O bound operations. [16]

2.3.3 Asynchronous functions

To create asynchronous programs modern .NET applications use two keywords: `async` and `await`. These keywords are used to create asynchronous functions. `async` is added to the function declaration and serves double purpose: it enables the use of `await` keyword within the function and also signals the compiler to generate a state machine for this method. The state machine is used when the execution encounters the `await` keyword, in this case it normally returns to the caller, similarly to `yield return` in an iterator. But before returning, the runtime attaches a continuation on the awaited task, ensuring that when it is finished, execution jumps back to the method and continues where it left off. An `async` method may return `Task` if it doesn't return a value, `Task<TResult>` if it returns a value and also `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` if it returns multiple values in an enumeration. The following is example of an asynchronous function:

```
async Task DoSomethingAsync()
{
    int value = 13;
    // Asynchronously wait 1 second.
    await Task.Delay(TimeSpan.FromSeconds(1));
    value *= 2;
    // Asynchronously wait 1 second.
    await Task.Delay(TimeSpan.FromSeconds(1));
    Console.WriteLine(value);
}
```

The async method begins executing synchronously. Within an async method, the `await` keyword performs an asynchronous wait on its argument. First it checks whether the operation is already completed. If it is, the execution continues synchronously, otherwise, it will pause the async method and return an incomplete task. When that operation finishes some time later, the async

[16] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

method will resume executing. This is where the main advantage of asynchronous programming appears – when the operation is awaited the main thread remains responsive and can, for example, update the UI, which means that the application does not appear "frozen" to the user while it performs awaited tasks.

The other important feature of async functions is the ability to capture the current synchronization context, which includes local variables, loop counters and the information about the thread which started the awaited task. The task starts executing on a pooled thread and when it is finished, the method will continue executing within the captured context. So, if the UI thread executes async method, the synchronization context ensures that execution resumes on the same UI thread. This behavior can be overridden by awaiting the result of the `ConfigureAwait` extension method and passing "false" as a parameter, for example:

```
await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false)
```

This will ensure that after it is paused, the method will resume on a threadpool thread where the task was running, instead of the UI thread. [17]

2.3.4 Parallelism

Parallelism is a concept of executing multiple tasks simultaneously. It could greatly improve the performance of the application, especially if there are multiple independent tasks that are running at the same time. In asynchronous programming it is possible to create parallelism by starting multiple asynchronous methods without awaiting them, for example:

```
async Task DoSomethingParallelAsync()
{
    var tasks = new List<Task<int>>();
    //start multiple tasks, each task takes different time to finish
    tasks.Add(Task.Run(async () => { await Task.Delay(2000); return 1; }));
    tasks.Add(Task.Run(async () => { await Task.Delay(3000); return 2; }));
    tasks.Add(Task.Run(async () => { await Task.Delay(1000); return 3; }));
    //await until all tasks are finished and then report results
    var results = await Task.WhenAll(tasks);
    for (int i = 0; i < results.Length; i++)
    {
        Console.WriteLine(results[i]);
    }
}
```

[17] Cleary, S. (2019). *Concurrency in C# Cookbook* (Second ed.).

In the preceding example, if each task were awaited individually, the whole execution would have been taken approximately 6 seconds, but because all 3 tasks are executing simultaneously, it takes only 3 seconds to complete all tasks (the time it takes to complete the longest task). [18]

2.3.5 Cancellation

Sometimes it is necessary to cancel a concurrent operation that is already started. For example, the operation might take too long to execute, so the timeout must be thrown, or the user may want to cancel the operation. For this purpose, the CLR provides `CancellationTokenSource` Class and `CancellationToken` Struct. `CancellationToken` works similarly to the boolean flag – when it is set, it will throw a `OperationCanceledException`. However, the token lacks the `Cancel` method that is used to set its state. Instead, this method is exposed by the `CancellationTokenSource` type. This separation is done for the security – a method that has the token can only check the state of the token but not initiate the cancellation. The following is the example of cancellation:

```
static void Main()
{
    //Create a new instance of CancellationTokenSource
    //The class constructor may take a specified amount of milliseconds
    //to initiate cancellation after a set period of time
    //which is useful for timeouts
    CancellationTokenSource cts = new CancellationTokenSource(3000);
    CancellationToken token = cts.Token;
    try
    {
        //Wait(); is used to not let the application terminate
        //this task before it is finised
        Task.Run(async () => { await DoSomethingAsync(token); }).Wait();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Operation was timed out");
    }
}
static async Task DoSomethingAsync(CancellationToken token)
{
    await Task.Run(async () =>
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            await Task.Delay(1000);
            //Check if token was set in cancelled state
            token.ThrowIfCancellationRequested();
        }
    });
}
```

[18] Cleary, S. (2019). *Concurrency in C# Cookbook*

In the preceding example the `CancellationTokenSource` is set to initiate cancellation in 3 seconds after the class is instantiated.

Most asynchronous methods in the CLR support cancellation tokens by default, for example: [19]

```
Task.Delay(3000, cancellationToken);
Task.Run(() => { return 0; }, cancellationToken);
```

2.3.6 Progress reporting

Another feature that greatly improves the experience of the user and facilitate application development and debugging is progress reporting. A simple example is a progress bar that shows the user the progress of the running operation. The CLR provides `IProgress<T>` interface and `Progress<T>` class to implement progress reporting. The following is the example of progress reporting:

```
static void Main()
{
    //Create an instance of Progress class and pass a delegate to its
constructor
    Progress<int> progress = new Progress<int>(i =>
    { Console.WriteLine("The current progress is {0}%", i); });

    Task.Run(async () => { await DoSomethingAsync(progress); }).Wait();
}
static async Task DoSomethingAsync(IProgress<int> progress)
{
    await Task.Run(async () =>
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(100);
            //Report the progress every 10%
            if (i % 10 == 0)
                progress.Report(i);
        }
    });
}
```

Upon istnantiating `Progress<T>` class captures the current synchronization context. Then its constructor takes an `Action<T>` delegate, which is invoked through that context every time `IProgress` interface calls `Report` method. `Progress<T>` also has a

19 Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

ProgressChanged event that could be subscribed to instead of (or in addition to) passing the action delegate to the constructor.

Since Progress<T> and IProgress<T> are both generic types they could take a more sophisticated report structure instead of int. For example it is possible to create a class with multiple report values and then pass this class as a parametrer to the generic types. [20]

2.3.7 Task combinators

Because there is a consistent protocol for asynchronous functions (they consistently return tasks), it is possible to write and use task combinators – functions that combine multiple tasks, regardless of what those tasks do. The CLR provides two task combinators: Task.WhenAny and Task.WhenAll.

Task.WhenAny returns a task that completes when any one of a set of tasks complete, for example:

```
static async Task QuickestTask()
{
    var tasks = new List<Task<string>>();
    //start multiple tasks, each task takes different time to finish
    tasks.Add(Task.Run(async () =>
    { await Task.Delay(2000); return "Task 1"; }));

    tasks.Add(Task.Run(async () =>
    { await Task.Delay(3000); return "Task 2"; }));

    tasks.Add(Task.Run(async () =>
    { await Task.Delay(1000); return "Task 3"; }));

    //await the task that finishes first
    var winner = await Task.WhenAny(tasks);
    Console.WriteLine("{0} finished first", winner.Result);
}
```

Because Task.WhenAny itself returns a task it is necessary to await it, after that it returns the task that finished first. Then it is possible to access the Result property of this task, use continuation, etc. The preceding example prints "Task 3 finished first".

Task.WhenAll completes when all tasks that are passed to it are completed. This is useful in situations where one of the tasks throws an exception. In this case the remaining tasks are still

[20] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

being executed even when there is a fault. The example in the 2.3.5 Cancellation sections utilizes the `Task.WhenAll` combinator. [21]

2.4 Parallel programming

2.4.1 Introduction to parallel programming

Programming with a goal to utilize multiple cores is called parallel programming. It is a subset of broader concept of multithreading that was discussed in previous sections. In section 2.3.7 `Task.WhenAll` was used in order to retrieve results in parallel. Parallel programming and specifically the Parallel framework provide classes and libraries with tools to write code that is executed in parallel.

Parallel programming should be used any time when there is a fair amount of computational work that could be split in multiple chunks. The more independent those chunks are, the better is the performance of the parallel framework functions. If pieces of work are dependent on each other or have some shared state sometime during the execution, it is likely that one thread would have to wait or synchronize with other and it would ultimately decrease the effectiveness of the parallel framework functions. Parallel programming temporarily increases CPU usage, specifically it increases load on multiple CPU cores to increase throughput. It is desirable on client systems (such as personal computers) where some CPU cores are often idle. Servers usually have some parallel code built-in, so generally it is not desirable to execute code that utilizes parallel programming on the server, since it may work against built-in parallelism and would not provide real benefit.

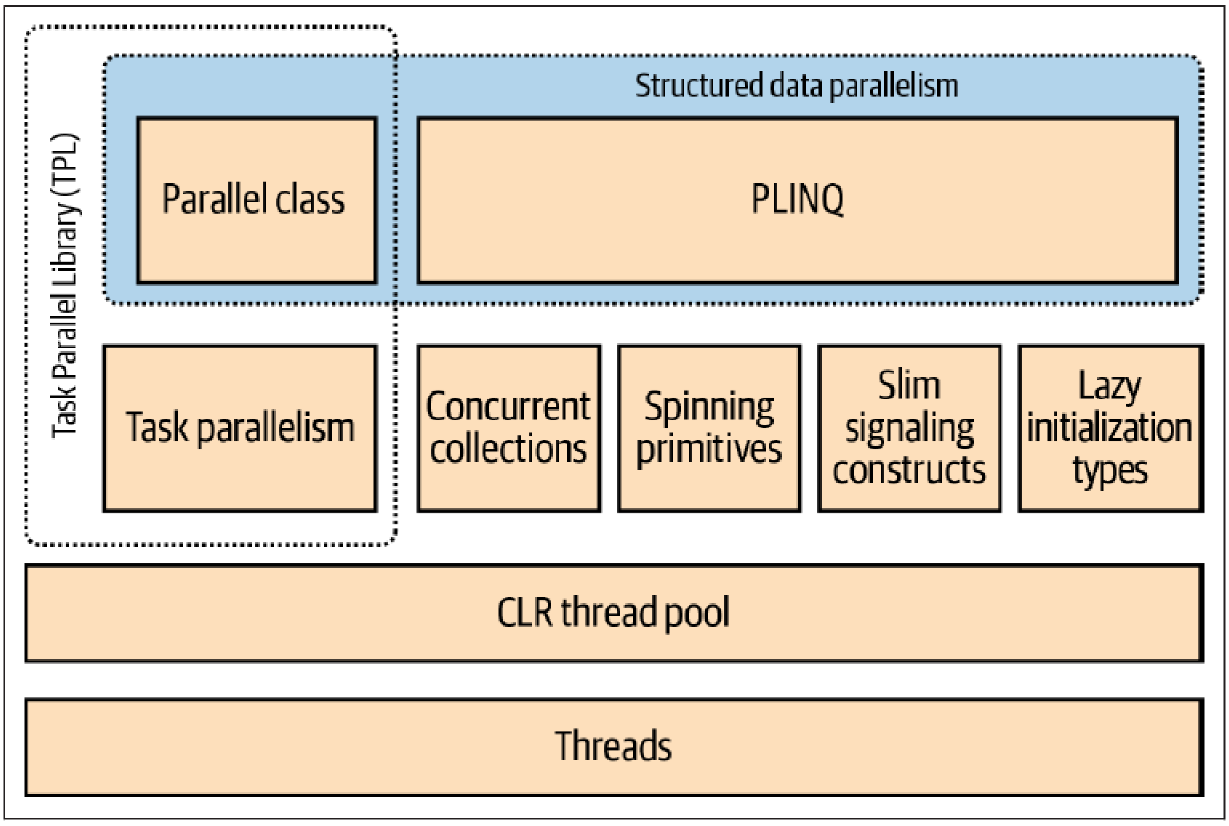
There are 2 forms of parallelism: data parallelism and task parallelism. When a set of tasks or operations must be performed on many data values it is possible to split those tasks between threads, hence the name – data parallelism. Task parallelism on the other hand is about splitting the tasks between threads, so each thread executes specific task. In general, data parallelism is easier to implement because it reduces or eliminates shared data so thread-safety and synchronization issues are of no concern. Also, because there are usually more data values than tasks the potential of parallel programming is higher while implementing data parallelism. Task parallelism is harder to implement because tasks may start and finish at places scattered across the program, making it harder to debug code and increasing possibility of errors to occur. [22]

[21] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[22] Cleary, S. (2019). *Concurrency in C# Cookbook* (Second ed.)

2.4.2 Overview of the Parallel framework

In the past, parallelization required low level manipulations with threads and locks, which means that it was difficult to create and maintain programs that utilized parallel execution. Modern .Net and Visual Studio provide support for parallel programming by adding a runtime, class library types, diagnostics tools. These tools together form the Parallel framework which simplifies the development of parallelized applications. Parallel framework was introduced in .Net framework 4.

Parallel framework consists of 2 layers of functionality, which is shown on the figure below. The higher level consists of structured data parallelism APIs: PLINQ and the `Parallel` class. The lower level consists of the task parallelism classes and a set of additional constructs which helps with parallel development.


Task Parallel Library (TPL)
Structured data parallelism
Parallel class
PLINQ
Task parallelism
Concurrent collections
Spinning primitives
Slim signaling constructs
Lazy initialization types
CLR thread pool
Threads

1. *Parallel framework components* [23]

PLINQ offers the automatization of parallelization including partitioning of the work into tasks, executing those tasks on thread pool threads, and combining results into a single output sequence. It is called declarative parallelism because it is required to only declare the work that needs to be parallelized, the rest is done by framework. In contrast, the other approaches (`Parallel` class

[23] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

and Task parallelism) are imperative, which means that it is required to manually write code to partition the work and combine the results. Because `Parallel` class is a part of structured data parallelism, it is necessary to only combine the results. Task parallelism also requires manual partition of the work.

Concurrent collections and spinning primitives that are featured on the figure above are also important parts of the parallel framework. They help with managing lower-level parallel programming activities. Parallel framework has been designed to support processors with multiple cores, including future generations of CPUs which would contain 24-32 cores and more. To benefit from a large number of cores it is necessary to divide the algorithm between them, otherwise if ordinary locks are used to protect common resources, the resultant blocking can lead to the fact that only a fraction of those cores are actually busy at once. Concurrent collections are tuned specifically for maximizing concurrent access and minimizing potential blocking. PLINQ and `Parallel` class rely on concurrent collections and spinning primitives for efficient management of work.

This thesis covers only Task parallel library in detail (`Parallel` class and task parallelism) since PLINQ is mostly rely on LINQ which is not related to this thesis. [24] [25]

2.4.3 `Parallel` class

`Parallel` class provides basic form of structured parallelism by 3 static methods:

- `Parallel.Invoke` – this method executes an array of delegates in parallel,
- `Parallel.For` – parallel equivalent of the regular `For` loop,
- `Parallel.Foreach` – parallel equivalent of the regular `Foreach` loop.

All these methods have a large number of overloads for different configurations of inputs. All 3 methods block the program execution until the work is complete. If an unhandled exception occurs during the execution of parallel work all worker threads are stopped after their current iteration and the exception (or multiple exceptions) are thrown back to the caller, wrapped in an `AggregateException` object.

`Parallel.Invoke` executes an array of Action delegates in parallel, for example:

[24] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[25] Microsoft Corporation. (2020). *Parallel Programming in .NET*.

```
static void Main()
{
    int[] arr = { 3, 2, 6, 7, 9, 10, 2, 1, 1, 0 };
    RaiseArraytoPow(arr, 2);
    foreach (int i in arr)
        Console.Write(i + ", ");
}
//This function raises all numbers in an array to the specified power
static int[] RaiseArraytoPow(int[] input, double pow)
{
    //Create 2 Action delegates and split the work in half to execute it in
parallel
    Parallel.Invoke(
        () =>
        {
            for (int i = 0; i < input.Length / 2; i++)
                input[i] = (int)Math.Pow(input[i], pow);
        },
        () =>
        {
            for (int i = input.Length / 2; i < input.Length; i++)
                input[i] = (int)Math.Pow(input[i], pow);
        }
    );
    return input;
}
```

`Parallel.Invoke` and `Parallel.Foreach` work similarly to the regular loops. Apart from slightly different syntax the main differenrce is that each iteration is being executed in parallel instead of sequentially, for example:

```
static void Main()
{
    //Execute SomeFunction 10 times in parallel
    Parallel.For(0, 10, (i) => SomeFunction(i));

    //Print strings from array in parallel
    string[] arr = { "String 1", "String 2", "String 3", "String 4" };
    Parallel.ForEach(arr, (str) => Console.WriteLine(str));
}
```

All functions in `Parallel` class are optimized for compute-bound work, so they would work efficiently even if an array of thousands of delegates is passed. This is because they partition large number of elements into batches and then assign them to a several underlying tasks instead of creating separate task for each delegate.

All functions in `Parallel` class are overloaded to accept `ParallelOptions` object. With this object it is possible to override default task scheduler, limit the maximum concurrency and pass a cancellation token.

With all `Parallel` class methods, it is necessary to manually implement collating of the results in order to keep threads safety. The following code is thread-unsafe:

```
static void Main()
{
    List<string> list = new List<string>();

    Parallel.Invoke(
        ()=>list.Add("Hello"),
        ()=>list.Add("World!")
        );

    foreach (string s in list)
        Console.Write(s + " ");
}
```

This code could lead to the unexpected output as was discussed in section 2.2.3. One possible solution is to add locks around adding to the list but this could introduce performance bottlenecks in case of large lists. Another solution is to use thread-safe collections which is a part of concurrent collections. [26] [27]

2.4.4 Parallel functions overloads

Parallel `For` and `Foreach` loops have multiple overloads in order to provide extended functionality. Both loops are overloaded to accept instance of `ParallelLoopState` class, which exposes multiple functions and properties that provide extended control over the loops. For example, in order to break out of a regular loop a simple `break;` statement is enough, however, it is impossible to break out of parallel loop with this statement because the body of both `For` and `Foreach` loops is a delegate. Instead, it is necessary to overload the parallel loop with `ParallelLoopState` object, which then provides a `break();` function, for example:

```
static void Main()
{
    int[] arr = { 0, 4, 9, 1, -3, 20 };
    //Overload loop with ParallelLoopState object
    Parallel.ForEach(arr, (i, loopState) =>
    {
        //Break out of the loop if the condition is met
        if (i < 0)
            loopState.Break();
        else
        {
            Console.Write(Math.Sqrt(i) + ", ");
        }
    });
```

[26] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[27] Microsoft Corporation. (2020). *Parallel Class*.

```
}
```

Because all loop iterations in the previous example are being executed in parallel, the output is usually unordered, for example the output might be 0, 2, 1, 4. Despite that, break function of `ParallelLoopState` object always yields at least the results of iterations that were preceding the break point, similarly to the regular loops. In this example at least first 4 elements of the array are always processed and the results are printed.

`ParallelLoopState` class also provides the `Stop();` function which works similarly to the `Break();` function with the main difference being that `Stop();` forces all threads to finish immediately after their current iteration. It means that in previous example only a subset of 0, 1, 2, 4 may be printed out if one of the threads is slow or lagging behind other threads (although it is unlikely in this scenario because of its simplicity). Calling `Stop();` is useful when something is found and there is no need for further processing or something has gone wrong and the results are irrelevant.

If the loop body is long and it is necessary to break out of the loop partway through `ShouldExitCurrentIteration` property should be used. This property is a boolean flag which is switched to true once `Break();` or `Stop();` function is called. It is meant to be used together with another property of `ParallelLoopState` class which is called `LowestBreakIteration`. This property indicates at what cycle the loop was broken by `Break();` function, or it returns null if `Stop();` function was called. The following is the example of usage of both these properties:

```
static void Main()
{
    int[] arr = { 0, 4, 9, 1, -3, 16, 25, 20 };
    //add loopIteration overload which indicates loop iteration index
    Parallel.ForEach(arr, (i, loopState, loopIteration) =>
    {
        if (i < 0)
        {
            loopState.Break();
            return;
        }

        //add small delay to ensure that
        //ShouldExitCurrentIteration is switched to true for all iterations
        Thread.Sleep(100);

        if (loopState.ShouldExitCurrentIteration)
        {
            if (loopIteration > loopState.LowestBreakIteration)
                return;
        }

        Console.Write(Math.Sqrt(i) + ", ");
    });
}
```

This example also showcases the usage of additional overload of the `Foreach` loop. `loopIteration` allows to access loop iteration index and is used for comparison with `loopState.LowestBreakIteration` in order to determine whether it is necessary to stop current iteration execution or continue the execution. [28] [29]

2.4.5 Task parallelism

Task parallelism is the low-level approach of parallelization in parallel framework. There are multiple classes in `System.Threading.Tasks` namespace that are used for creating task parallelism:

- `Task` – for managing a unit of work,
- `Task<Tresult>` – for managing a unit of work and returning a value,
- `TaskFactory` – for creating new tasks,
- `TaskFactory<Tresult>` – for creating tasks and continuations with the same return type,
- `TaskScheduler` – for managing the scheduling of tasks,
- `TaskCompletitionSource` – for manually controlling a task's workflow.

`Task`, `Task<Tresult>` and `TaskCompletitionSource` classes were already mentioned in asynchronous programming chapters. Even though these classes are widely used in asynchronous programming, originally, they were created for parallel programming.

`TaskFactory` object is created after calling `Task.Factory` property of `Task`. The purpose of `TaskFactory` object is to create new tasks, specifically of these types: "ordinary" tasks, continuations with multiple antecedents.

There are multiple ways to create new tasks. `Task.Run` and `Task.Start` create new tasks but they are limited to creating only the "ordinary" tasks. The more advanced method of creating new tasks is `Task.Factory.StartNew` method which provides multiple useful overloads that

[28] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[29] Microsoft Corporation. (2020). *ParallelLoopState Class.*

allows to tune the creation of new tasks. In fact, `Task.Run` is a shortcut for creating `Task.Factory.StartNew` with default overloads.

One of such overloads is `TaskCreationOptions` enum, which allows to modify tasks execution. `TaskCreationOptions` has 3 combinable values: `LongRunning`, `PreferFairness` and `AttachedToParent`. `LongRunning` suggest the scheduler to dedicate a thread to a task. This is beneficial for long running tasks and for I/O bound tasks that might otherwise force short running task to wait for an unreasonable amount of time before being scheduled. `PreferFairness` suggest to scheduler to try to ensure that tasks are scheduled in the same order in which they were started. If `PreferFairness` is not specified, scheduler will internally optimize the scheduling of the tasks using local work-stealing queues – an optimization that allows the creation of child tasks without creating the contention overhead that would otherwise arise with a single work queue. A child task is created by specifying `AttachedToParent` overload.

Child tasks could be created as follows:

```
static void Main()
{
    Task parent = Task.Factory.StartNew(() =>
    {
        Console.WriteLine("I am a parent");

        Task.Factory.StartNew(() =>
            { Console.WriteLine("I am an independent task"); });

        Task.Factory.StartNew(() =>
            { Console.WriteLine("I am a child"); },
            TaskCreationOptions.AttachedToParent);
    });
    parent.Wait();
}
```

Child tasks are special in that when the parent task is awaited, it waits for any child tasks to complete. This can be particularly useful for propagating exceptions and for task continuations. [30]

2.4.6 Continuations

Continuations were mentioned in asynchronous programming chapters, essentially a continuation is another task that starts when previous task (also called antecedent) completes, fails or is being

[30] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

cancelled. This chapter will expand the concepts of continuation. Here is a simple example of the continuation:

```
static void Main()
{
    Task task1 = Task.Factory.StartNew(() => { Console.Write("Hello, "); });
    Task task2 = task1.ContinueWith(ant => { Console.WriteLine("World!");
});
    task2.Wait();

    Task<float> task3 = Task.Factory.StartNew(() => { return MathF.Pow(4,
2); });
    Task task4 = task3.ContinueWith(ant => { Console.WriteLine(ant.Result);
});
    task4.Wait();
}
```

This code snippet includes 2 examples of task continuation. First example with `task1` and `task2` is a regular continuation. Once `task1` finished its execution, `task2` starts to execute its own delegate. It is important to notice the `ant` object, which is a reference to the antecedent task (`task1` in this case). Second example with `task3` and `task4` demonstrates how the result of the antecedent task, which is accessed by the `ant` object, could be used in the continuation task.

Another powerful feature of continuations is exceptions handling. Continuation task can know whether the antecedent task has faulted by querying antecedent task's `Exception` property or by specifying `TaskContinuationOptions`. Combining this with the parent-child tasks relation is an effective way of creating durable algorithms, for example:

```
static void Main()
{
    Task task1 = Task.Factory.StartNew(() =>
    {
        //Start multiple child tasks
        Task.Factory.StartNew(() =>
        { Console.WriteLine("Child 1 finished successfully!"); },
            TaskCreationOptions.AttachedToParent);

        //Start a task that throws exception
        Task.Factory.StartNew(() =>
        { throw null; },
            TaskCreationOptions.AttachedToParent);

        Task.Factory.StartNew(() =>
        { Console.WriteLine("Child 3 finished successfully!"); },
            TaskCreationOptions.AttachedToParent);
    });

    //Catch all exceptions thrown by antecedent
    Task CatchExceptions = task1.ContinueWith(ant =>
    {
       Console.WriteLine(ant.Exception);
    }, TaskContinuationOptions.OnlyOnFaulted);
    CatchExceptions.Wait();
}
```
In the previous example one of the tasks intentionally throws an exception, but the application does not crash, instead the exception is handled (in this case the exception is written out in the console).

By default, continuation is executed unconditionally, whether the antecedent completes, throws an exception or is cancelled. This behavior could be modified with the set of flags included within `TaskContinuationOptions` enum, as shown in previous example. These flags are combinable (meaning it is possible to combine multiple flags), here are the core flags: `NotOnRunToCompletition` (The continuation should start only if the antecedent task was cancelled or threw an exception), `NotOnFaulted` (start continuation if the antecedent task did not throw an exception), `NotOnCanceled` (start continuation if the antecedent task was not cancelled). There are also multiple precombined flags:

- OnlyOnRunToCompletition = NotOnFaulted | NotOnCanceled,
- OnlyOnFaulted = NotOnRunToCompletition | NotOnCanceled,
- OnlyOnCanceled = NotOnRunToCompletition | NotOnFaulted.

Another important aspect of continuations with flags is that if they are not executed due to a flag, they are not abandoned or forgotten, instead they are cancelled. This means that any continuations on the continuation itself would still run, for example:

```
static void Main()
{
    Task task1 = Task.Factory.StartNew(() =>
        { Console.Write("Hello, "); });

    Task CatchExceptions = task1.ContinueWith(ant =>
        { Console.WriteLine("An error occured!"); },
            TaskContinuationOptions.OnlyOnFaulted);

    Task task2 = CatchExceptions.ContinueWith(ant =>
        { Console.WriteLine("World!"); });

    task2.Wait();
}
```

The output of the preceding example would be "Hello, World!", because even though `task1` doesn't throw an error, `CatchExceptions` is simply cancelled, but the continuations on this task (in this example `task2`) are still executed, since by default all continuations are executed unconditionally. In order to prevent `task2` from execution if `CatchException` is cancelled, it is necessary to specify flag `TaskContinuationOptions.NotOnCanceled` during the initialization of the task.

Task combinators such as `WhenAll` and `WhenAny` which were discussed in previous chapters are also applicable to continuations, for example:

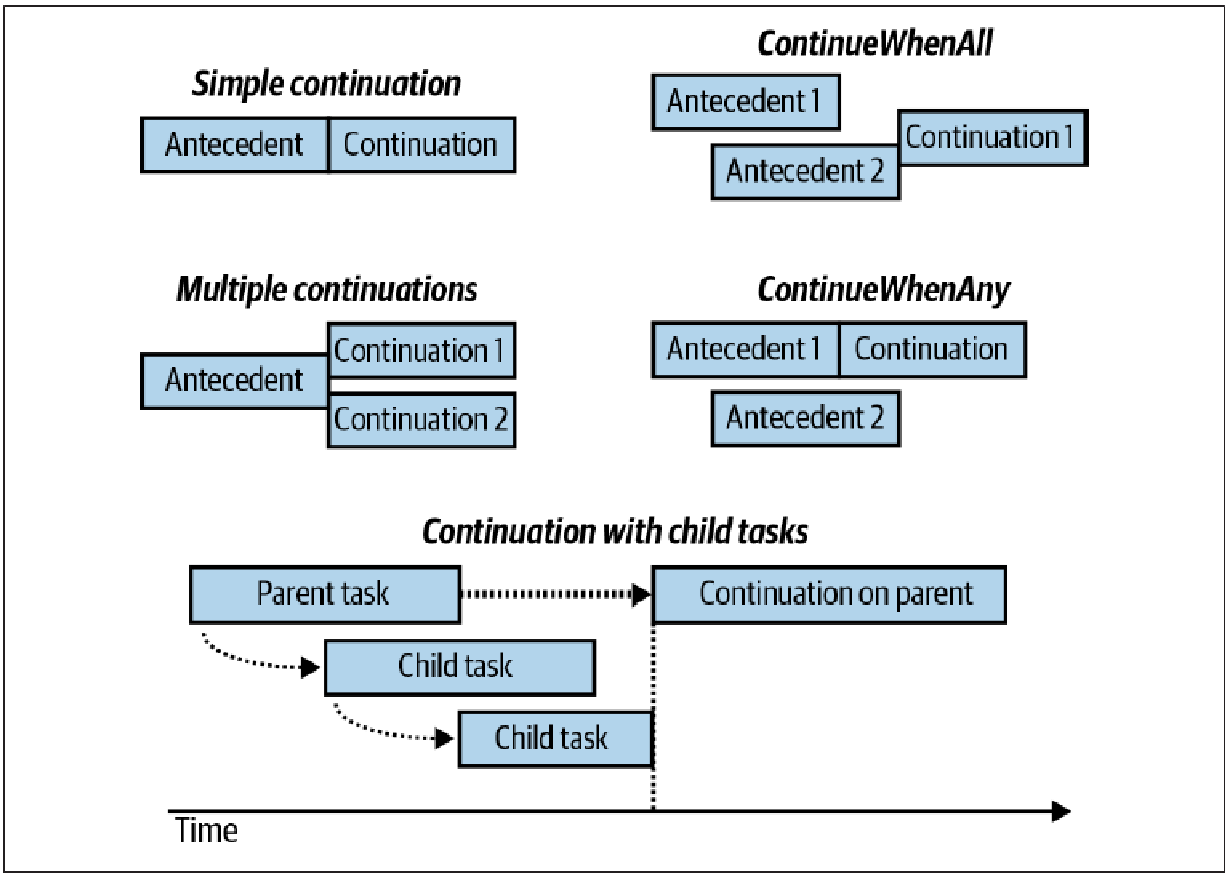
```
static void Main()
{
    Task task1 = Task.Factory.StartNew(() =>
        { Console.WriteLine("Task 1 finished."); });

    Task task2 = Task.Factory.StartNew(() =>
        { Console.WriteLine("Task 2 finished."); });

    Task continuantion = Task.WhenAll(task1, task2).ContinueWith(ant =>
        { Console.WriteLine("Done!"); });
    continuantion.Wait();
}
```

In this example `task1` and `task2` start their execution in parallel, `continuation` task waits for both tasks to finish and then prints "Done!", this technique allows to create continuations on multiple antecedents. It is also possible to start multiple continuations on a single antecedent, for example:

```
static void Main()
{
    Task task1 = Task.Factory.StartNew(() =>
        { Console.WriteLine("Task 1 finished."); });

    Task continuation1 = task1.ContinueWith(ant =>
        { Console.WriteLine("Starting continuation 1"); });

    Task continuation2 = task1.ContinueWith(ant =>
        { Console.WriteLine("Starting continuation 2"); });

    Task.WaitAll(continuation1, continuation2);
}
```

In this example both `continuation1` and `continuation2` start to execute in parallel after `task1` is finished. It is also possible to modify continuations so that they start their execution synchronously by specifying `TaskContinuationOptions.ExecuteSynchronously` on both continuations. In this case both continuations would start on the same thread as their antecedent (`continiuation1` would always be executed first and `continuation2` – second).

Multiple forms of continuations are graphically represented on the following figure: [31]



2. Different forms of continuation [32]

2.4.7 Concurrent collections

It is not thread-safe to use regular collections in code which utilizes parallelization. Instead, .NET offers several thread-safe concurrent collections: `ConcurrentStack<T>`, `ConcurrentQueue<T>`, `ConcurrentBag<T>`, `ConcurrentDictionary<TKey, TValue>`. These collections are optimized for highly concurrent scenarios and they also could be used when there is a need for thread-safe collection (as an alternative to locking around a regular collection). However, there are multiple downsides of concurrent collections that needs to be mentioned:

- The regular collections outperform concurrent collections in all but highly concurrent scenarios.

[31] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*
[32] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

- A thread-safe collection does not mean that code that would use this collection is also thread-safe.
- Enumerating over a concurrent collection while other thread is modifying it does not throw an exception. It means that the result of enumeration might be a mixture of old and new data.
- There is no concurrent version of `List<T>`.
- The concurrent stack, queue and bag classes are implemented internally with linked list. It means that they are less memory-efficient then the regular stack and queue classes, but better for concurrent access, since inserting into a linked list require only modification of a few references, while inserting an element into a `List<T>` might cause the shifting of thousands of elements.

A producer/consumer collection is one for which the two primary use cases are adding an element (producing) and removing an element (consuming). The examples of this collection are stack and queue. Producer/consumer collections are significant in parallel programming because they have efficient lock-free implementations. Concurrent collections expose several lock-free methods to perform test-and-act operations. Most of these methods are unified via the `IProducerConsumerCollection<T>` interface, for example bool `TryAdd (T item);` and `TryTake (out T item);` These methods test whether an add/remove operation could be performed and if so, the perform the add/remove. The testing and acting are atomically performed, which means it cannot be divided, thus eliminating the need to lock around the collection.

The following classes implement `IProducerConsumerCollection<T>` interface: `ConcurrentStack<T>`, `ConcurrentQueue<T>`, `ConcurrentBag<T>`. The particular element that `TryTake` removes is defined by the class: with a stack `TryTake` removes the most recently added element, with a queue `TryTake` removes the least recently added element and with a bag `TryTake` removes whatever element it could modify most efficiently.

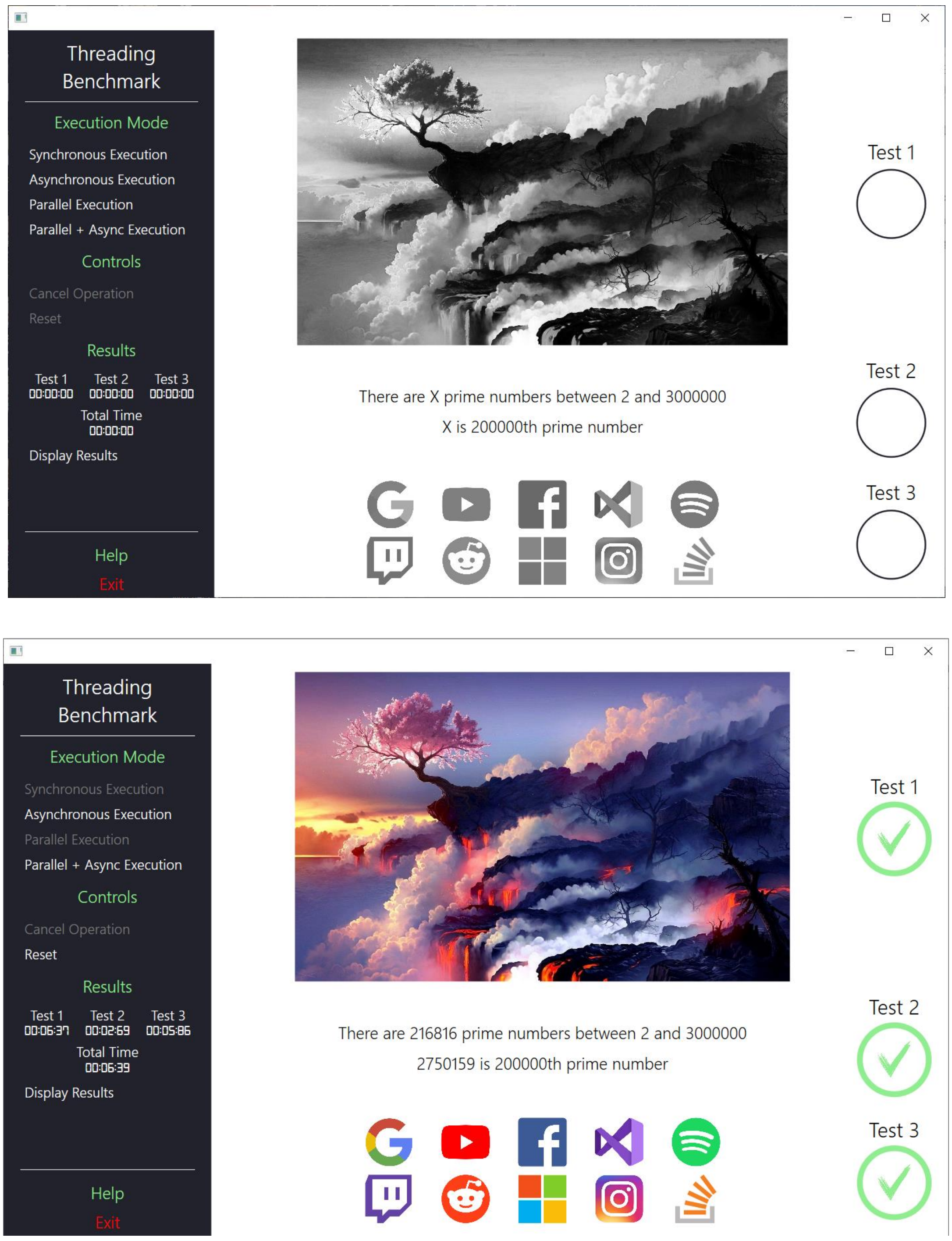
`ConcurrentBag<T>` stores an unordered collection of objects with duplicates permitted. Bag is suitable for situations when it doesn't matter which element is retrieved by `Take` or `TryTake`. The benefit of the bag over a concurrent stack or queue is that a bag's `Add` method suffers no contention when called by multiple threads at once. In contrast, calling `Add` in parallel on a concurrent queue or stack causes some contention, although a lot less then locking around regular collection. Bag is ideal when the concurrent operation on the collection consists mostly of adding new elements to the collection. A concurrent bag would be a poor choice for a producer/consumer queue because elements are added and removed by different threads. [33]

[33] Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference*

Part 3 – Practical part. Threading benchmark application

3.1 Application description. MVVM design pattern.

In order to showcase different threading techniques that were discussed in previous chapters I've made a threading benchmark, which is an application that can execute a series of tasks in different threading modes. The application is made with C# and WPF. Below is a figure which demonstrates the UI of the application.



Test 1 00:00:00 Test 2 00:00:00 Test 3 00:00:00
Total Time 00:00:00

There are X prime numbers between 2 and 3000000

X is 200000th prime number

Test 1 00:06:37 Test 2 00:02:69 Test 3 00:05:86
Total Time 00:06:39

There are 216816 prime numbers between 2 and 3000000

2750159 is 200000th prime number

3. UI of the threading benchmark application

Model-View-ViewModel (MVVM) design pattern was used to implement the UI and functionality of the application. Although it is not necessary to implement this pattern, it is a common practice to develop a WPF application while utilizing this pattern. As the name of the pattern suggests, the application is divided into 3 independent modules – Views, ViewModels and Models.

Views define how the application looks to the end user. To create views for the application I used XAML – it is a markup language which is very similar to CSS. XAML only defines how the application looks, it doesn't define how the application behaves (except for some runtime style changes, such as changes of button colors while they are pressed etc.). Below is a part of XAML code which defines how Help and Exit buttons look and where they are located.

```
<!-- StackPanel for the bottom buttons-->
<StackPanel Orientation="Vertical" DockPanel.Dock="Bottom" VerticalAlignment="Bottom">

    <Separator Background="White" Margin="20,0" VerticalAlignment="Stretch"/>

    <Button Content="Help"
            Style="{StaticResource capitalBtnStyle}"
            Margin="0,10,0,0"
            Command="{Binding HelpCommand}"
            FontSize="20"
            Foreground="LightGreen"/>

    <Button Content="Exit"
            Style="{StaticResource capitalBtnStyle}"
            Command="{Binding ExitCommand}"
            FontSize="20"
            Foreground="Red"/>

</StackPanel>
```

Each element in XAML (such as `<StackPanel>`, `<Separator>`, `<Button>`) has different properties which may contain different values, for example `Margin="0,10,0,0"` and `FontSize="20"`. Apart from assigning numerical values to properties it is also possible to bind them to a property in the ViewModel, for example `Command="{Binding HelpCommand}"`. Bindings are the essential part of the MVVM, because of them the property in a View could be controlled from the ViewModel. Almost every property could be bound to a property in the ViewModel.

Models are objects which define data structures of the application. They do not have a visual representation and may not even be directly related to the application. An example of model is an object which is passed from the function that executes the Image test to the MainWindow ViewModel during the asynchronous test in order to update the UI. Below is the definition of the Image Test Report Model.

```
/// <summary>
/// Data model which is used in progress reports for the Image test
/// </summary>
public class ImageTestReportModel
{
    public BitmapImage ReportImage { get; set; } = null;
    public int ProgressArcAngle { get; set; } = 0;
}
```

During the execution of the Image test an instance of the report data model is created, populated with values and then passed to the ViewModel.

Finally, the ViewModels are objects which model the GUI (Graphical User Interface) of the application, meaning they provide data to the Views and define their functionality. Each ViewModel has to implement `INotifyPropertyChanged` interface as described below.[34]

```
public class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    //Implementation of PropertyChanged event from Microsoft documentation
    protected void OnPropertyChanged([CallerMemberName] string name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

When a property of the ViewModel is updated, the `PropertyChanged` event is raised and the property of the View, which is bounded to the property of the ViewModel is updated. Below is an example of the ViewModel property which represents the text of the Image test timer.

```
private string _imageTestTimerText = "00:00:00";
public string ImageTestTimerText
{
    get { return _imageTestTimerText; }
    set
    {
        _imageTestTimerText = value;
        OnPropertyChanged();
    }
}
```

To summarize how the MVVM works:

1. From the View user starts a new execution of the benchmark,
2. ViewModel of this View calls the function which executes the benchmark and starts a timer,

[34] George, A. D. (2017, March 30). *How to: Implement Property Change Notification*.

3. Timer updates the value of the `ImageTestTimerText` string,
4. When the value of the string is updated the `PropertyChanged` event is raised,
5. View receives a notification from the event and updates the text which is displayed to the user.

The same principle (with some minor changes) is applied to every property of the View which is bounded to the ViewModel.

The main objective of the MVVM is to separate UI into several independent modules. This allows to easily implement the application on different platforms. For example, if there would be a need to implement the application for Linux or macOS, or even Android and iOS, it would be necessary to create only a new version of the GUI elements (the Views) for the specific platform. After this, the GUI elements could be bounded to the ViewModels and the application would work in the same way it does on Windows.

3.2 Application execution modes

The application consists of 4 execution modes – Synchronous execution, Asynchronous execution, Parallel execution and Asynchronous + Parallel execution. When the user clicks the respective to the execution mode button, the application starts its execution in the selected execution mode. Below is detailed description of each execution mode:

- Synchronous execution - in this mode benchmark is executed synchronously – no threading techniques are applied. During synchronous execution UI is not updated, meaning that application will appear "frozen" until all tests are finished.
- Asynchronous execution - in this mode benchmark is executed asynchronously. While all tests are running application remains responsive and UI is being constantly updated.
- Parallel execution - similarly to synchronous execution UI is not updated while benchmark is running in parallel mode, however, because all 3 tasks are executed in parallel (at the same time), overall execution time is lower than during synchronous execution.
- Asynchronous + Parallel execution - this mode combines the best of parallel and asynchronous executions. While all 3 tasks are executed at the same time, application remains responsive like in asynchronous mode.

The MainWindow ViewModel contains the definitions of 4 functions, each representing the respective execution mode. Following the MVVM algorithm which was described earlier, when the user clicks a button to start the benchmark in a specific mode, ViewModel starts executing the appropriate function. During the function execution multiple properties of the ViewModel are changed. Because these properties are bounded to the elements in the View, user sees the updates of the GUI. Below is the definition of the `AsyncTest` function, which is executed during the asynchronous run.

```
private async void AsyncTest()
{
    //disable UI elements
    ResetUI();
    SyncBtnIsEnabled = AsyncBtnIsEnabled = ParallelBtnIsEnabled =
ParallelAsyncBtnIsEnabled = false;
    CancelBtnIsEnabled = true;
    AsyncBtnTag = "Clicked";

    DTime.Start();
    TotalTimeWatch.Start();

    //Initialize instances of Progress class that are later used to trigger
    //the events that update UI
    Progress<ImageTestReportModel> imageTestProgress = new
Progress<ImageTestReportModel>();
    imageTestProgress.ProgressChanged += ImageTestProgressEvent;
    Progress<PNumTestReportModel> pNumTestProgress = new
Progress<PNumTestReportModel>();
    pNumTestProgress.ProgressChanged += PNumTestProgressEvent;
    Progress<WebsitesTestReportModel> websitesTestProgress = new
Progress<WebsitesTestReportModel>();
    websitesTestProgress.ProgressChanged += WebsitesTestProgressEvent;



//Use try catch statement for cancellation Tokens
    try
    {
        ImageTestWatch.Start();
        await ImageTest.StartAsync(imageTestProgress, cts.Token);
        ImageTestWatch.Stop();
        ImageTestCheckmarkVisibility = Visibility.Visible;
    }
    catch (OperationCanceledException)
    {
        ImageTestProgressBarColour = new SolidColorBrush(Colors.Red);
        ImageTestCrossmarkVisibility = Visibility.Visible;
        CancelBtnTag = null;
    }

    try
    {
        PNumTestWatch.Start();
        await PrimeNumbersTest.StartAsync(pNumTestProgress, pNumRangeTest,
pNumNthTest, cts.Token);
        PNumTestWatch.Stop();
        PNumTestCheckmarkVisibility = Visibility.Visible;
    }
    catch (OperationCanceledException)
    {
        PNumTestProgressBarColour = new SolidColorBrush(Colors.Red);
        PNumTestCrossmarkVisibility = Visibility.Visible;
```

```
            CancelBtnTag = null;
        }

        try
        {
            WebsitesTestWatch.Start();
            await WebsitesTest.StartAsync(websitesTestProgress, cts.Token);
            WebsitesTestWatch.Stop();
            WebsitesTestCheckmarkVisibility = Visibility.Visible;
        }
        catch (Exception ex)
        {
            if (ex is OperationCanceledException || ex is WebException)
            {
                WebsitesTestProgressBarColour = new SolidColorBrush(Colors.Red);
                WebsitesTestCrossmarkVisibility = Visibility.Visible;
                CancelBtnTag = null;
            }
        }

        //enable UI elements
        AsyncBtnIsEnabled = ParallelAsyncBtnIsEnabled = ResetBtnIsEnabled =
    true;
        CancelBtnIsEnabled = false;
        AsyncBtnTag = null;

        //Force update UI to enable disabled buttons
        CommandManager.InvalidateRequerySuggested();

        TotalTimeWatch.Stop();
        DTime.Stop();

        //Push the results of execution to the results manager
        ResultsDataManager.SetResults("Asynchronous", ImageTestTimerText,
    PNumTestTimerText, WebsitesTestTimerText, TotalTimeText,
    !cts.IsCancellationRequested);
    }
```

Additionally, application also features multiple control buttons which allow to do the following:

- Cancel operation - allows to prematurely cancel current benchmark run. Cancellation is available only in Asynchronous and Parallel + Asynchronous modes.
- Reset - used to reset tests and UI to their default state. It is necessary to manually reset UI only before running application in Synchronous and Parallel modes, since Asynchronous execution modes will reset UI automatically.
- Display results - used to display the results of all benchmark executions in a separate window. Results window consists of a table which contains all results and line charts for visual representation of results. Line charts display only the results of executions which were successfully completed (e.g. not cancelled with "Cancel Operation" button).

3.3 Benchmark tests

During each cycle of execution, the application is performing 3 different tests which are designed to load CPU and create multiple web requests. Each test exists in its own static class. When the execution of the benchmark starts, the MainWindow ViewModel calls the respective function in the test class. Each test function can communicate with the ViewModel to send the results of the test execution, which are later displayed to the user. Below is detailed description of each test:

Test 1 "Image test" - during this test, the image that is originally desaturated gradually becomes colorized. The aim of this test is to load CPU with a large amount of array operations. At first, coordinate of every pixel of the colorized picture is assigned to a jagged array, then the array is shuffled and finally the image is "restored" by replacing colors of resulting bitmap image, which originally contains desaturated picture, with colors of colorized bitmap image, following the pattern of the shuffled array. While running benchmark in asynchronous mode it is possible to observe how picture gradually becomes colorized in a random pattern. Because GUI is not updated during synchronous and parallel executions, picture becomes colorized at once, once the test is finished.

Below are private fields used in the Test Image class

```
private static readonly Random rnd = new Random();
//Bitmaps to store colorized and desaturated versions of the Image
private static readonly Bitmap ColorizedImage = new
Bitmap(Properties.Resources.TestImageColorized);
private static readonly Bitmap BWImage = new
Bitmap(Properties.Resources.TestImageBW);
private static Bitmap ResultImg = new Bitmap(BWImage.Width, BWImage.Height);
//Array to reference the coordinates of pixels
private static readonly int[][] ColorCoordinates = new
int[ColorizedImage.Width * ColorizedImage.Height][];
//Set how frequently the report is sent to the UI
private static readonly int ProgressReportFrequency = 20;
```

Test Image class consists of 2 public functions – `StartSync` and `StartAsync` which are called to start the execution of the test in the respective mode. Notably, Image Test class doesn't implement the execution in parallel mode. That is because functions and variables in the Image test are highly dependent on each other, therefore no efficient ways were found to implement parallel execution for this test. Also, Test Image class contains 2 private functions `SetCoordinates` and `ToWpfBitmap`.

`SetCoordinates` function consists of 2 loops. The first loop iterates through all pixels in `ColorizedImage` Bitmap and assigns the pixel coordinates to the `ColorCoordinates` array. The second loop shuffles the `ColorCoordinates` array.

`ToWpfBitmap` is used to convert `Bitmap` to `BitmapImage`. `Bitmap` is a legacy type which was used in Windows Forms, while `BitmapImage` is a new type which is used in WPF. `Bitmap` is used because it allows to access and manipulate every pixel of the image. After performing the image manipulations, the `Bitmap` is converted to the `BitmapImage` which is later displayed in the View. Below is the implementation of the `StartAsync` function:

```
//Execute Image test in asynchronous mode
public static async Task StartAsync(IProgress<ImageTestReportModel>
progress, CancellationToken cancellationToken)
{
    cancellationToken.ThrowIfCancellationRequested();
    //Create an instance of the report model object which will be later sent
in progress report
    ImageTestReportModel report = new ImageTestReportModel();
    ResultImg = new Bitmap(Properties.Resources.TestImageBW);
    int reportIndex = 0;

    //Send initial progress report
    reportIndex++;
    report.ProgressArcAngle = (reportIndex * 360) / ProgressReportFrequency;
    progress.Report(report);

    await Task.Run(() => SetCooridnates());
    cancellationToken.ThrowIfCancellationRequested();

    await Task.Run(() =>
    {
        for (int i = 0; i < ColorCoordinates.Length; i++)
        {
            ResultImg.SetPixel(ColorCoordinates[i][0],
ColorCoordinates[i][1], ColorizedImage.GetPixel(ColorCoordinates[i][0],
ColorCoordinates[i][1]));

            //Send the report to the UI thread to update progress bar
            if (i % (ColorCoordinates.Length / ProgressReportFrequency ) ==
0 && i > 0 || i == ColorCoordinates.Length - 1)
            {
                reportIndex++;
                report.ReportImage = ToWpfBitmap(ResultImg);
                //Adjust the value of progress bar angle (multiply by 343
instead of 360)
                //because the first report was already sent earlier
                report.ProgressArcAngle = (reportIndex * 343) /
ProgressReportFrequency;
                progress.Report(report);
            }

            if (cancellationToken.IsCancellationRequested)
                break;
        }
    });
    cancellationToken.ThrowIfCancellationRequested();
}
```

Test 2 "Prime numbers test" - the aim of this test is to give CPU a large number of arithmetic operations. Test consists of 2 parts – in first part computer calculates how many prime numbers there are in a given range (in this case from 2 to 3000000). In the second part computer calculates what is the Nth prime number (in this case 200000th prime number).

Prime Number Test class doesn’t have any private fields and consists of 8 functions. 4 public functions – `StartSync`, `StartAsync`, `StartParallel` and `StartParallelAsync` are called by the ViewModel during the benchmark execution in the respective mode. 4 private functions – `PrimeNumbersRange`, `PrimeNumbersRangeAsync`, `NthPrimeNumber` and `NthPrimeNumberAsync` are used to calculate the prime numbers. The first two functions calculate how many prime numbers there are in a given range. The last two functions calculate the Nth prime number. There exists two version of each functions (regular and Async) both versions perform identical calculations, while Async version can also send the progress reports.

Below is the implementation of the `StartParallelAsync` functions

```
//Execute Prime Numbers test in parallel async mode
public static async Task StartParallelAsync(IProgress<PNumTestReportModel>
progress, int range, int nthPos, CancellationToken cancellationToken)
{
    cancellationToken.ThrowIfCancellationRequested();
    //Create an instance of the report model object which will be later sent
in progress report
    PNumTestReportModel report = new PNumTestReportModel();

    //Start both tasks at the same time
    Task rangeTask = Task.Run(() => { return PrimeNumbersRangeAsync(range,
ref progress, ref report, ref cancellationToken); })
        .ContinueWith(ant=> { report.PNumberRangeResult = ant.Result; });

    Task nthTask = Task.Run(() => { return NthPrimeNumberAsync(nthPos, ref
progress, ref report, ref cancellationToken); })
        .ContinueWith(ant => { report.NthPNumberResult = ant.Result; });

    //Asynchronously wait for both task to complete
    await Task.WhenAll(rangeTask, nthTask);
    cancellationToken.ThrowIfCancellationRequested();

    report.ProgressArcAngle = 360;
    progress.Report(report);
}
```

Test 3 "Websites test" - in this test computer loads 10 websites in the background. The aim of this test is to showcase how different threading techniques can improve application performance while executing web requests. On the application GUI there are 10 desaturated icons, each representing a website. Once corresponding website is downloaded, icon becomes colorized. Similarly to the Image test, icons will become colorized all at once after synchronous and parallel execution, while during asynchronous executions icons will become colorized gradually, once corresponding website is downloaded.

Websites test class consists of 7 functions. Similarly to the Prime Numbers Test class, it has 4 public functions which are called by the MainWindow ViewModel. The last 3 private functions are `DownloadWebsite`, `DownloadWebsiteAsync` and `PrepareWebsites`. `PrepareWebsites` function creates a new list of 10 strings, where each string is a URL of a website. The other two functions create a new instance of the `WebClient` class and then download each website from the list which was prepared by `PrepareWebsites` function.

Below is the implementation of `StartParallel` function of the Websites Test class

```
//Execute Websites test in parallel mode
public static void StartParallel(IProgress<WebsitesTestReportModel>
progress)
{
    //Create an instance of the report model object which will be later sent
in progress report
    WebsitesTestReportModel report = new WebsitesTestReportModel();
    List<string> websites = PrepareWebsites();
    var temp = new object();

    Parallel.ForEach<string>(websites, (site) =>
    {
        //Cretae an instance of the website model which is later passed into
the report
        WebsiteDataModel result = new WebsiteDataModel();

        DownloadWebsite(site);

        result.URI = site;
        result.isLoaded = true;

        //lock around access to shared object for thread safity
        lock (temp)
        {
            report.LoadedWebsites.Add(result);
        }
    });

    //send the report to update the UI
    report.ProgressArcAngle = 360;
    progress.Report(report);
}
```

3.4 Results and discussion

In order to collect an accurate statistic of the application performance, every execution mode has been run for 50 times each. Benchmark was executed on my personal computer. It has the following technical specifications:

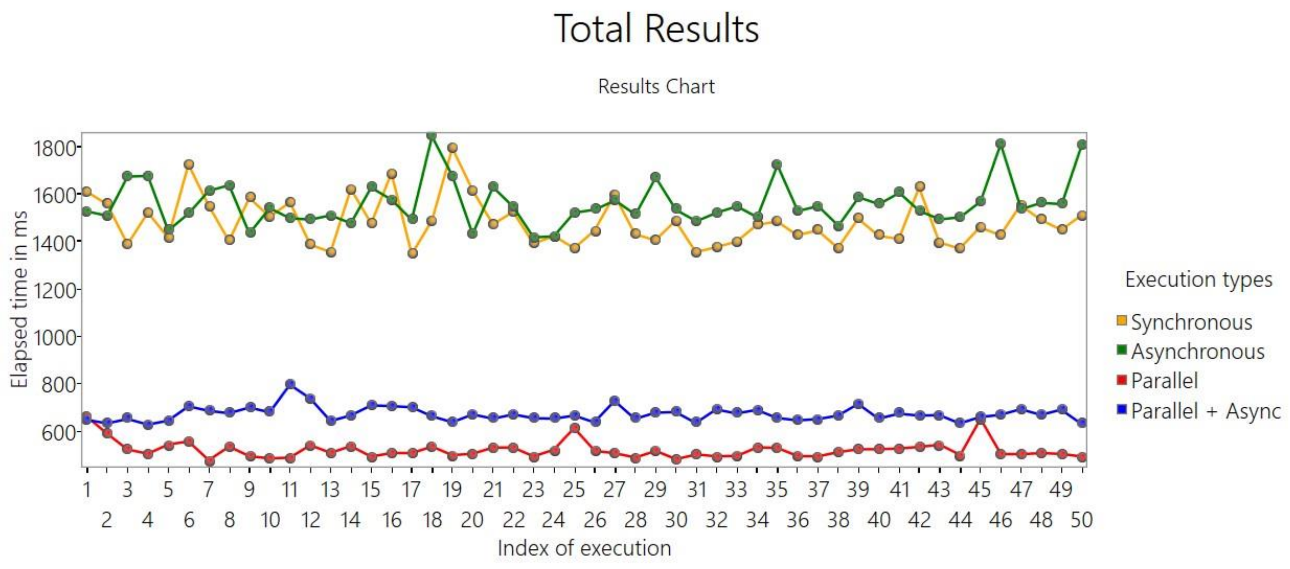
- Operating System: Windows 10
- CPU: AMD Ryzen 3 2200G 3.5 GHz
- Motherboard: ASUS ROG Strix B450-F
- GPU: AMD Vega 56
- RAM: G.Skill RipjawsV 8GB DDR4 3200 MHz

Because tests are designed to load the computer, it would take more time to execute the tests on a lower-end machine (such as laptop), and oppositely, computer with better technical specifications then my machine could execute the tests faster. While running the tests there was no processes running in background, since additional processes may require resources from the computer and thus may affect the results of the benchmark.

The time results of the executions are following:

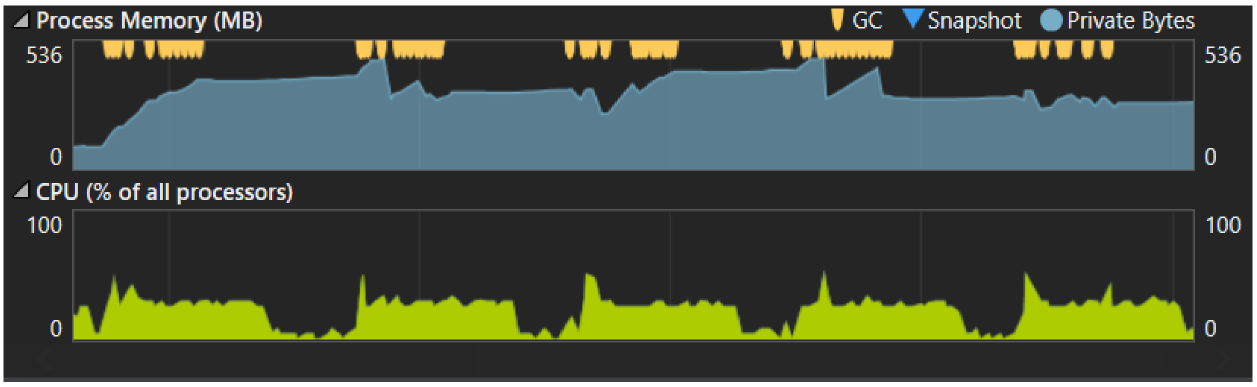
- Synchronous execution: 14.75 seconds
- Asynchronous execution: 15.54 seconds
- Parallel execution: 5.13 seconds
- Parallel + Asynchronous execution: 6.66 seconds

Below is the chart that visually represents the results:

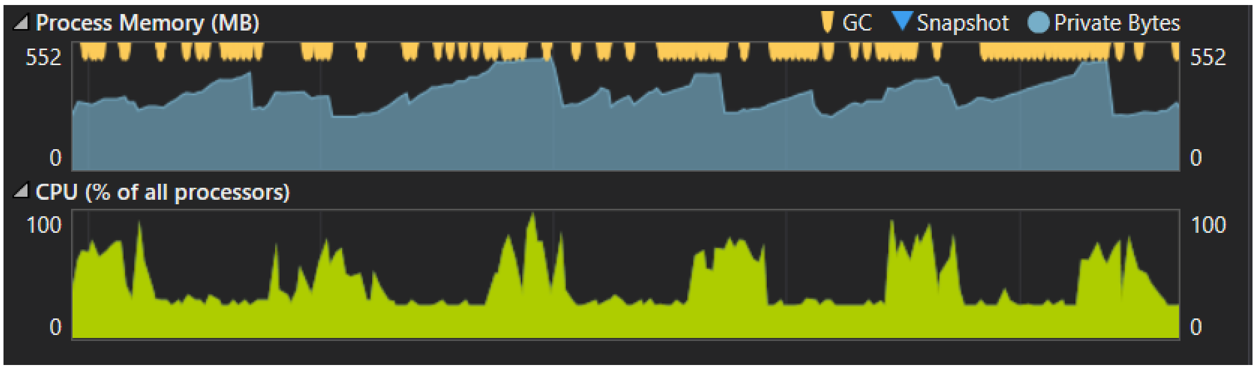


4. *Total results chart*

Both parallel execution modes greatly outperform sequential execution modes in terms of total execution time, however, while application is running in parallel mode it requires more resources from the CPU and slightly more operating memory, as shown on figures below.



5. *Benchmark execution in Asynchronous (sequential) mode*



6. *Benchmark execution in Parallel + Asynchronous mode*

As a result, while application takes much less time to complete all tasks, it appears "slow" or "less responsive" for the user and thus is worse for the overall user experience. Another issue are the potential failures during the execution of the tests. Because of high CPU load, application has a higher chance to crash while performing tasks in parallel than sequential mode (although the chance is still low).

Another trend which is observed from the total results is that the execution in asynchronous modes takes on average more time to complete than non-asynchronous modes. It is expected since the application also has to continuously update the GUI of the application and keep the application responsive. However, when looking at the application from user experience perspective, the advantages of asynchronous execution modes outweigh the increase of overall execution time by approx. 1 second.

Conclusion

The main objectives of this thesis were to explore how multithreading is implemented in modern programming languages and frameworks, and if there is any substantial impact of threading on application performance and also is it practical to implement multithreading in modern applications.

Theoretical part of the thesis covers in detail the implementation of multithreading in C# and .NET. In order to gain the necessary information regarding the theoretical topics of the thesis I used multiple books and websites (mainly Microsoft Documentation (https://docs.microsoft.com/en-us/dotnet/csharp/) and Stackoverflow (https://stackoverflow.com/) websites). Despite the fact that threading techniques have a slightly steep learning curve, and require in depth knowledge of the framework to fully utilize them, modern frameworks are greatly optimized for multithreaded development, making it easy to implement basic threading and already improve the application.

The practical part of the thesis has demonstrated the threading techniques and compared for possible approaches to application development. In order to test the principles discussed in the theoretical part of the thesis, mainly asynchronous and parallel execution of the application, I made a benchmark which is used for testing of different threading techniques. Based on the results from repeated test, it is possible to conclude…

Asynchronous programming significantly improves user experience at a slight performance cost related to regular UI updates, meaning that asynchronous execution should be implemented for any operation that is longer than a few seconds in order to not leave the user with an unresponsive application. Parallel execution is also an extremely useful technique, which allows to dramatically improve the application performance, especially in cases where there are multiple independent tasks which could be executed simultaneously.

Although the application that is shown in the practical part is not nearly as large and complicated as a "real-world" released application, it still demonstrates that threading is a very desired technique for modern applications, especially for those that process large amounts of data, handle multiple web requests and are used by regular people not related to the IT industry.

Finally, I choose this topic for my bachelor thesis because I wanted to challenge my programming skills and gain new ones along the way. During the research and writing of this thesis I've learnt a lot of new concepts and techniques related to different areas of programming, which would definitely help in my future development as a software engineer and my professional career.

Bibliography

Albahari, J., & Johannsen, E. (2020). *C# 8.0 in a Nutshell: The Definitive Reference* (First ed.). O'Reilly Media Copyright 2020, 978-1-492-05113-8.

Cleary, S. (2019). *Concurrency in C# Cookbook* (Second ed.). O'Reilly Media Copyright 2019 Stephen Cleary, 978-1-492-05450-4.

Das, D. (2018, January 1). *COMPONENTS OF CPU AND THEIR FUNCTIONS | DIAGRAM*. Retrieved from csetutor.com: https://www.csetutor.com/components-of-cpu-and-their-functions/

George, A. D. (2017, March 30). *How to: Implement Property Change Notification*. Retrieved from Microsoft Documentation: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/how-to-implement-property-change-notification?view=netframeworkdesktop-4.8

Hoffman, C. (2018, October 12). *CPU Basics: Multiple CPUs, Cores, and Hyper-Threading Explained*. Retrieved from howtogeek.com: https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/

Hope, C. (2019, July 10). *Computer processor history*. Retrieved from computerhope.com: https://www.computerhope.com/history/processor.htm

Microsoft Corporation. (2020). *ManualResetEvent Class*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.manualresetevent?view=netcore-3.1

Microsoft Corporation. (2020). *Parallel Class*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel?view=net-5.0

Microsoft Corporation. (2020). *Parallel Programming in .NET*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/

Microsoft Corporation. (2020). *ParallelLoopState Class*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallelloopstate?view=net-5.0

Microsoft Corporation. (2020). *Task Class*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=netcore-3.1

Microsoft Corporation. (2020). *Thread Class Documentation*. Retrieved from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=netcore-3.1

Plantz, R. G. (n.d.). *Introduction to Computer Organization*. Retrieved from bob.cs.sonoma.edu: https://bob.cs.sonoma.edu/IntroCompOrg-RPi/sec-progexec.html

White, A. (2019, August 19). *What is Core in Computer? Here is What it Does*. Retrieved from techgearoid.com: https://techgearoid.com/articles/what-is-core-in-computer/