Kaunas University of Technology
Faculty of Informatics

Concurrent Programming

Engeneering Project

Student name, surname, academical group

Student **Zahi El Helou**, IFU-1

Position

Instructor **Ryselis Karolis**

Kaunas, 2023

1. Description:

Goal — The aim of my program is to simplify and make a program that counts the number of paragraphs in a data file using the concurrency tools and methods we learned this semester.

2. Task analysis and solution

I have created a program that reads a text, so I have prepared a total of 8 different data sets. We then execute the code in parallel to read a single data file and count the number of paragraphs in the data file. We are simultaneously calculating the time taken to execute this code by varying the number of threads being used which is set by user input, to find out the most optimum number of threads to use for executing this code.

The tool we are using to achieve this is C# parallel LINQ.

The data concurrency tools were applied as such:

- PLINQ Query Creation: The array of lines is converted into a PLINQ query using lines.AsParallel(). This enables the parallel processing of these lines in the subsequent steps. By invoking AsParallel() on the array of lines, your program enables the PLINQ framework to process these lines in parallel. It's the key step that shifts from sequential to parallel execution.
- Customizing Degree of Parallelism: The program allows the user to specify the maximum degree of parallelism (i.e., the maximum number of concurrent tasks, or number of threads). This is set using WithDegreeOfParallelism(maxDegreeOfParallelism) if the user provides a valid number. This setting controls how many concurrent tasks PLINQ should use for processing, allowing for fine-tuning performance based on available resources.

3. Testing Methods

All 8 data files already exist and are placed inside the program, which can be changed to the user's requirements.

We are performing speed tests with every data file to calculate the optimum number of threads to use.

We have done multiple tests by implementing multiple conditions, such as inserting a small number of threads and applying it for a large data file, and inserting a large number of threads and applying it for a small data file.

Here we can see that we have 4 paragraphs.

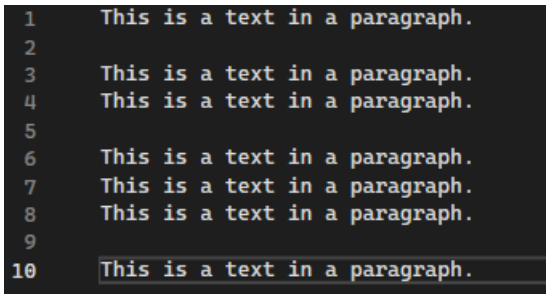

Figure 1 datafile1.txt

Here's the result confirming having 4 paragraphs.

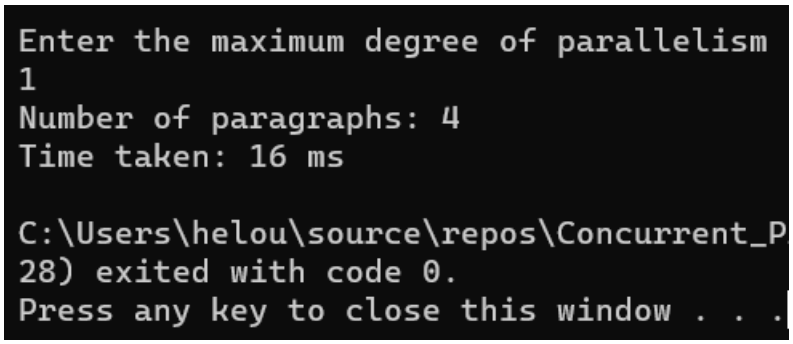

Figure 2 Console for datafllle1.txt

4. Performance Analysis:

Analysis:

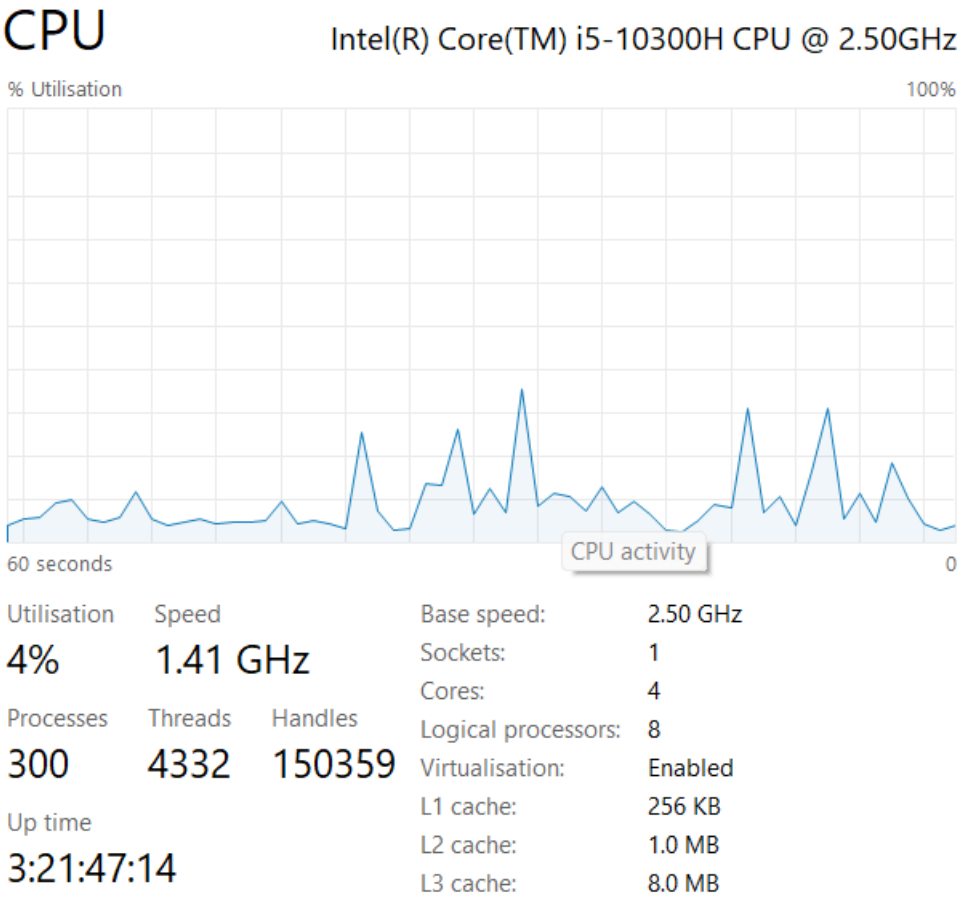

Figure 3 CPU activity

Device specifications:

Device name	LAPTOP-K3HU8513
Processor	Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz
Installed RAM	8.00 GB (7.83 GB usable)
Device ID	86B4AA4E-2DBC-4A21-86A5-B6705CF8BF55
Product ID	00327-31016-58988-AAOEM
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

The number of threads the processor can run concurrently depends on the number of physical cores and the number of threads per core that your processor supports. The Intel Core i5-10300H processor has 4 cores and 8 threads, so it can run up to 8 threads concurrently.

The amount of memory (RAM) that the device has can also affect processing time, especially if the program requires a lot of memory. With 8 GB of RAM, I should be able to run multiple programs at the same time without running into memory issues. The type of tasks that your program is performing can also impact processing time. Some tasks, such as those that involve a lot of math or data manipulation, can be more CPU-intensive and take longer to complete. Finally, the overall speed and performance of your device can also affect processing time. The Intel Core i5-10300H processor is a mid-range processor with a base clock speed of 2.50 GHz and a boost speed of 4.20 GHz, so it should be able to handle most tasks quickly.

Source Code:

```
using System;
using System.IO;
using System.Linq;
using System.Diagnostics;

class ParallelParagraphCounter
{
    static void Main(string[] args)
    {
        string filePath = args.Length > 0 ? args[0] :
@"C:\Users\helou\source\repos\Concurrent_Project\Concurrent_Project\datafile1.txt";

        Console.WriteLine("Enter the maximum degree of parallelism (-1 for unlimited or a
positive integer): ");
        if (!int.TryParse(Console.ReadLine(), out int maxDegreeOfParallelism) ||
maxDegreeOfParallelism < -1)
        {
            Console.WriteLine("Invalid input. Using default value: -1 (unlimited)");
            maxDegreeOfParallelism = -1;
        }

        try
        {
            string[] lines = File.ReadAllLines(filePath);

            var stopwatch = Stopwatch.StartNew();

            var lineQuery = lines.AsParallel();
            if (maxDegreeOfParallelism != -1)
            {
                lineQuery = lineQuery.WithDegreeOfParallelism(maxDegreeOfParallelism);
            }

            var isBoundary = lineQuery.Select(line => string.IsNullOrWhiteSpace(line))
                                      .ToArray();

            int paragraphCount = CountParagraphs(isBoundary);

            stopwatch.Stop();

            Console.WriteLine($"Number of paragraphs: {paragraphCount}");
            Console.WriteLine($"Time taken: {stopwatch.ElapsedMilliseconds} ms");
        }
        catch (IOException e)
        {
            Console.WriteLine($"An error occurred while reading the file: {e.Message}");
        }
```

```
    }

    static int CountParagraphs(bool[] isBoundary)
    {
        int paragraphCount = 0;
        bool inParagraph = false;

        foreach (var boundary in isBoundary)
        {
            if (boundary)
            {
                if (inParagraph)
                {
                    paragraphCount++;
                    inParagraph = false;
                }
            }
            else
            {
                inParagraph = true;
            }
        }

        if (inParagraph)
        {
            paragraphCount++;
        }

        return paragraphCount;
    }
}
```

Output:

Test 1:

```
1     This is a text in a paragraph.
2
3     This is a text in a paragraph.
4     This is a text in a paragraph.
5
6     This is a text in a paragraph.
7     This is a text in a paragraph.
8     This is a text in a paragraph.
9
10    This is a text in a paragraph.
```

Figure 4 datafile1.txt

```
Enter the maximum degree of parallelism
1
Number of paragraphs: 4
Time taken: 16 ms

C:\Users\helou\source\repos\Concurrent_P
92) exited with code 0.
Press any key to close this window . . .
```

Figure 5 Console with 1 thread

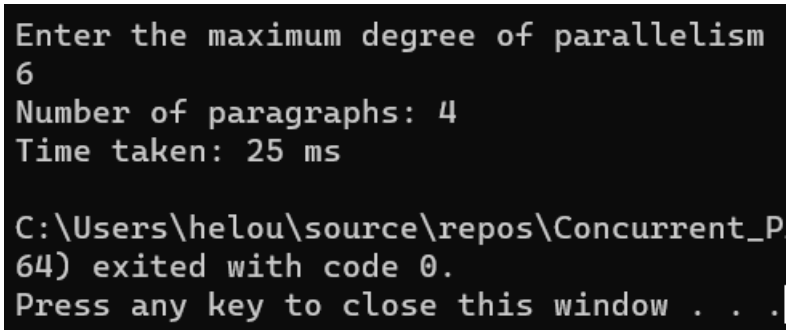

Figure 6 Console with 6 threads

Here we can see that is it less efficient to use a bigger number of threads on a small data file, due to the overhead of thread management outweighing the benefits of parallel processing for a small workload.

So we can realize that it is taking more when the number of threads is bigger.

Test 2:

Now we will try a different data file that has a bigger size, it has 1 000 000 lines of data:

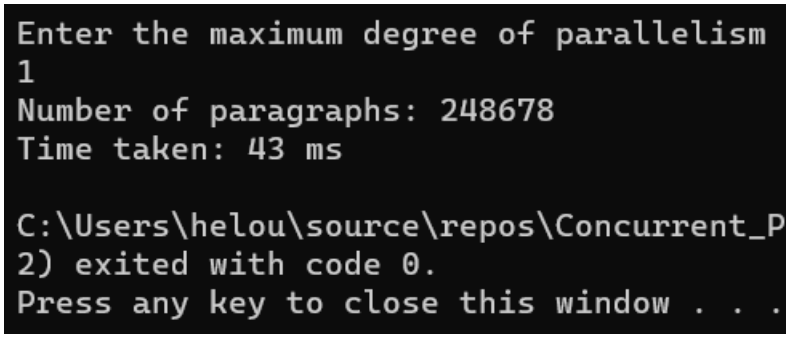

Figure 7 Console for datafile6.txt

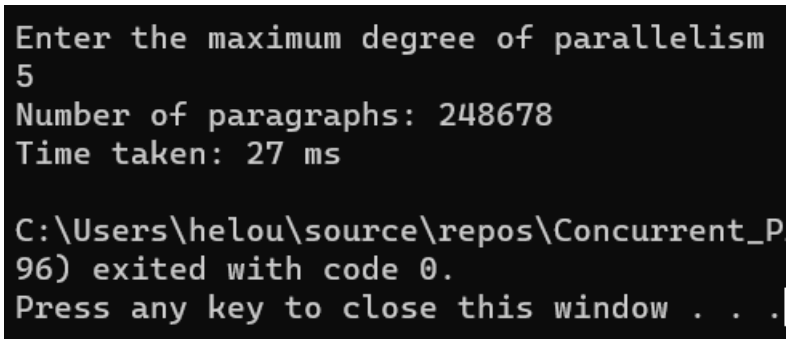

Figure 8 Console for datafile6.txt

Here we can realize that when we have a large file and, in that case, we have 248678 paragraphs, and when we input a small number of threads, the program is taking a lot more time comparing to when we used 5 threads.

This is where parallel processing shines. With a large data file, distributing the workload across multiple threads can significantly reduce processing time.

So using a large number of threads (up to a certain limit, specifically the system's logical amount of processors) generally offers substantial performance improvements due to more efficient workload distribution and better utilization of the CPU's capabilities.

Test 3

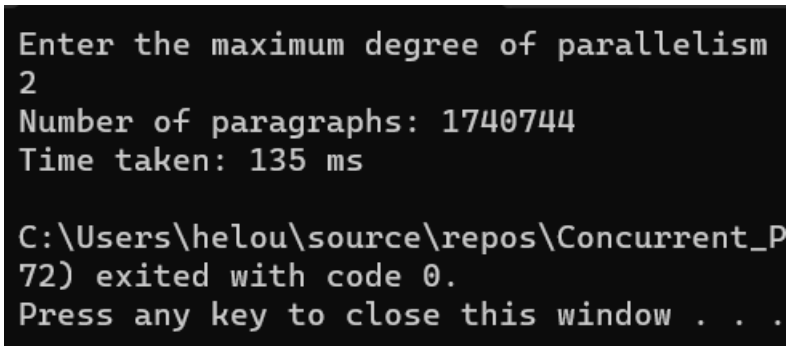

Figure 9 Console for datafile8.txt

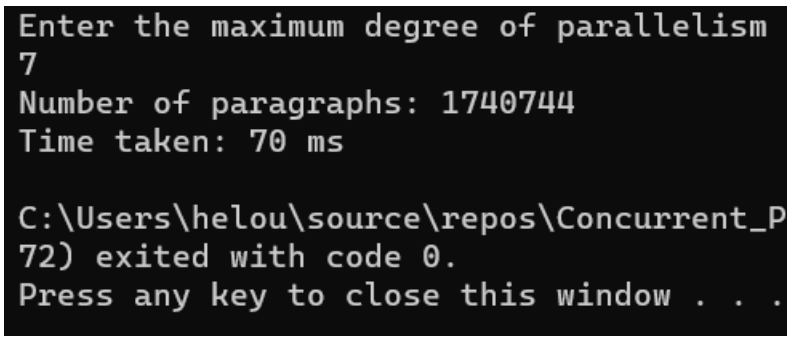

Figure 10 Console for datafile8.txt

Here we can realize that when we have a large file and, in that case, we have 1740744 paragraphs, and when we input a small number of thread which is 2, the program is taking a lot more time comparing to when we used 7 threads.

This is where parallel processing shines. With a large data file, distributing the workload across multiple threads can significantly reduce processing time.

So using a large number of threads (up to a certain limit, specifically the system's logical amount of processors) generally offers substantial performance improvements due to more efficient workload distribution and better utilization of the CPU's capabilities.

Analysis and Visualization:

Test 1:

We will test in the case where we have the smallest data file 1 which has 10 lines (4 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	16	20	22	23	24	22	21	22	21.25

Figure 8 Test for the smallest data file

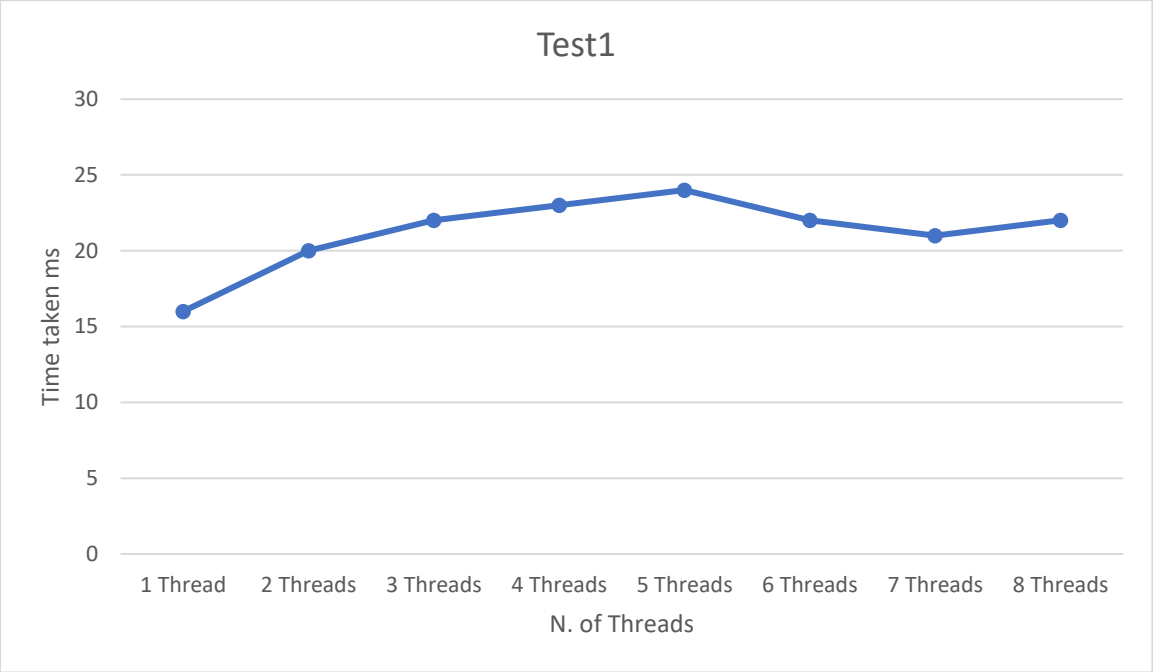

Figure 9 Graph for testing

Here we can see that is it less efficient to use a bigger number of threads on a small data file, due to the overhead of thread management outweighing the benefits of parallel processing for a small workload.

So we can realize that it is taking more when the number of threads is bigger.

Test 2:

We will test in the case where we have the smallest data file 2 which has 100 lines (25 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	20	19	18	19	21	23	22	22	20.5

Figure 10 Test 2

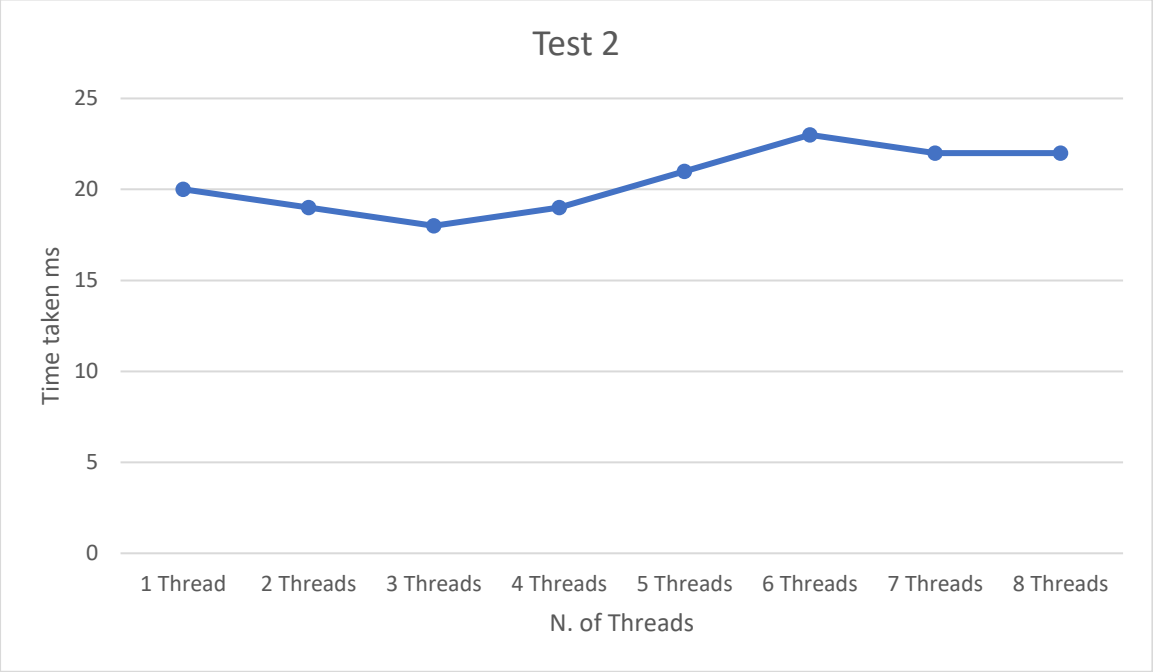

Figure 11 Graph for testing

Here we can see that is it less efficient to use a bigger number of threads on a considerably small data file, due to the overhead of thread management outweighing the benefits of parallel processing for a small workload.

So we can realize that it is taking more when the number of threads is bigger.

Test 3:

We will test in the case where we have the smallest data file 3 which has 1000 lines

(249 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	20	19	18	19	22	20	26	22	20.75

Figure 12 Test 3

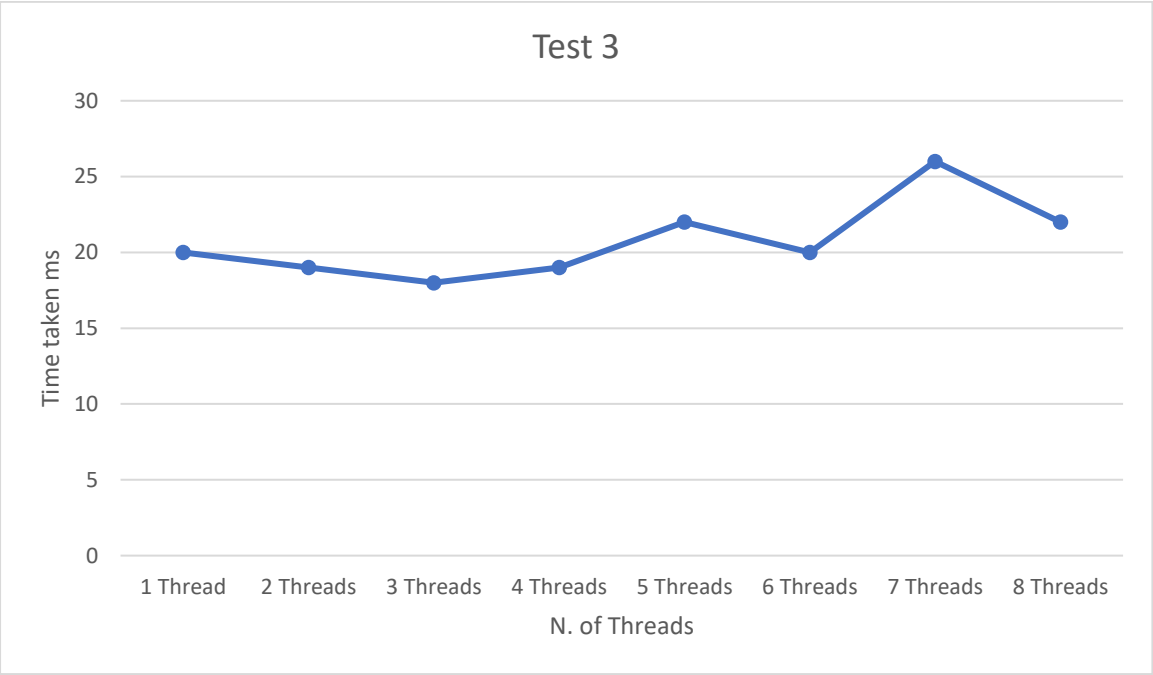

Figure 13 Graph for testing

Here we can see that is it less efficient to use a bigger number of threads on a considerably small data file, due to the overhead of thread management outweighing the benefits of parallel processing for a small workload.

So we can realize that it is taking more when the number of threads is bigger.

Test 4:

We will test in the case where we have the smallest data file 4 which has 10 000 lines

(2487 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	20	20	19	21	20	21	21	21	20.3

Figure 14 Test 4

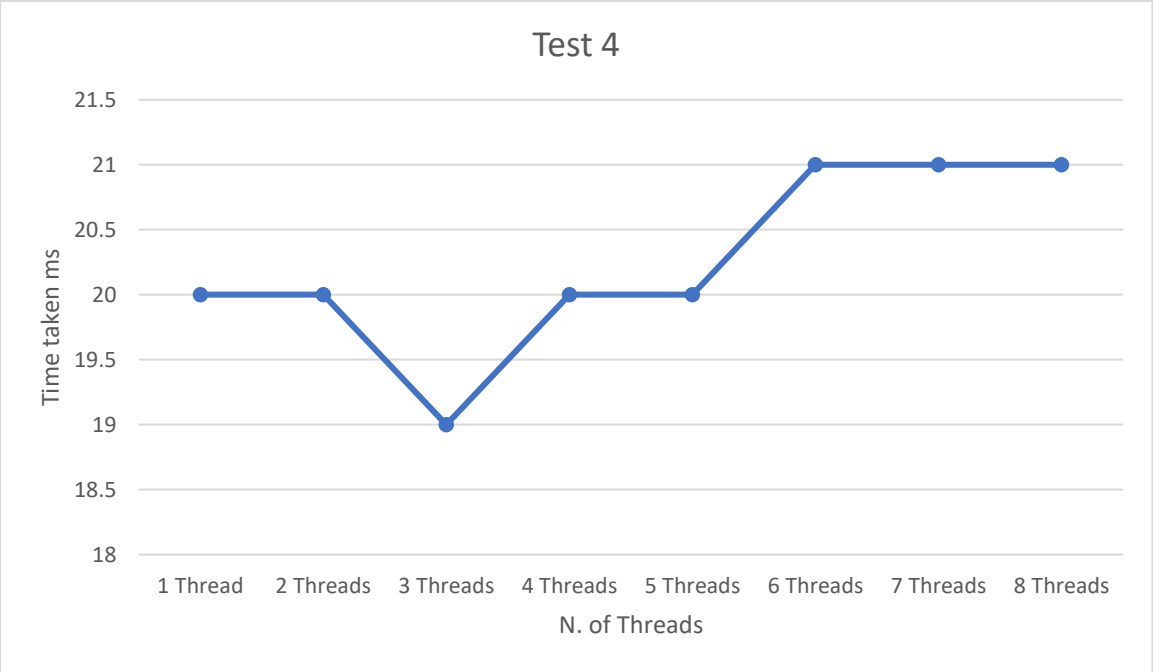

Figure 15 Graph for testing

This curve flattens when we increase the number of threads to 3 and more. We are able to observe a flatter curve and more consistent times.

Test 5:

We will test in the case where we have a moderate data file 5 which has 100 000 lines

(24868 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	25	22	20	19	21	21	22	22	22.75

Figure 16 Test 5

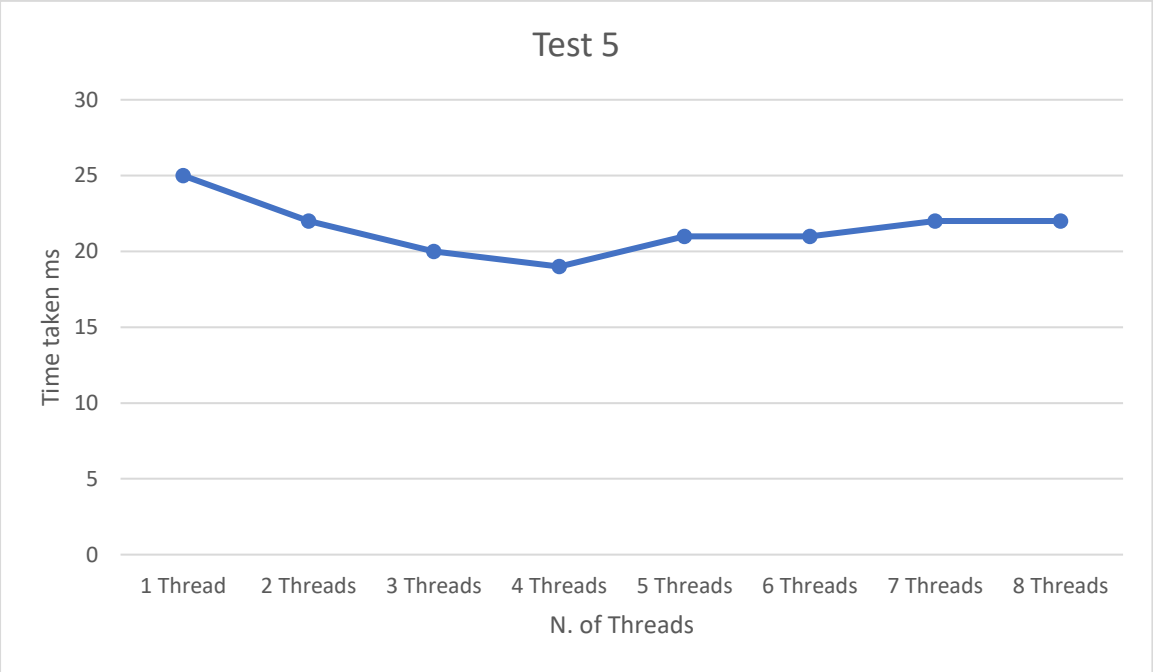

Figure 17 Graph for testing

This curve flattens when we increase the number of threads to 3 and more. We are able to observe a flatter curve and more consistent times.

Test 6:

We will test in the case where we have a moderate data file 6 which has 1 000 000 lines

(248678 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	45	35	29	28	27	33	32	32	32.6

Figure 18 Test 6

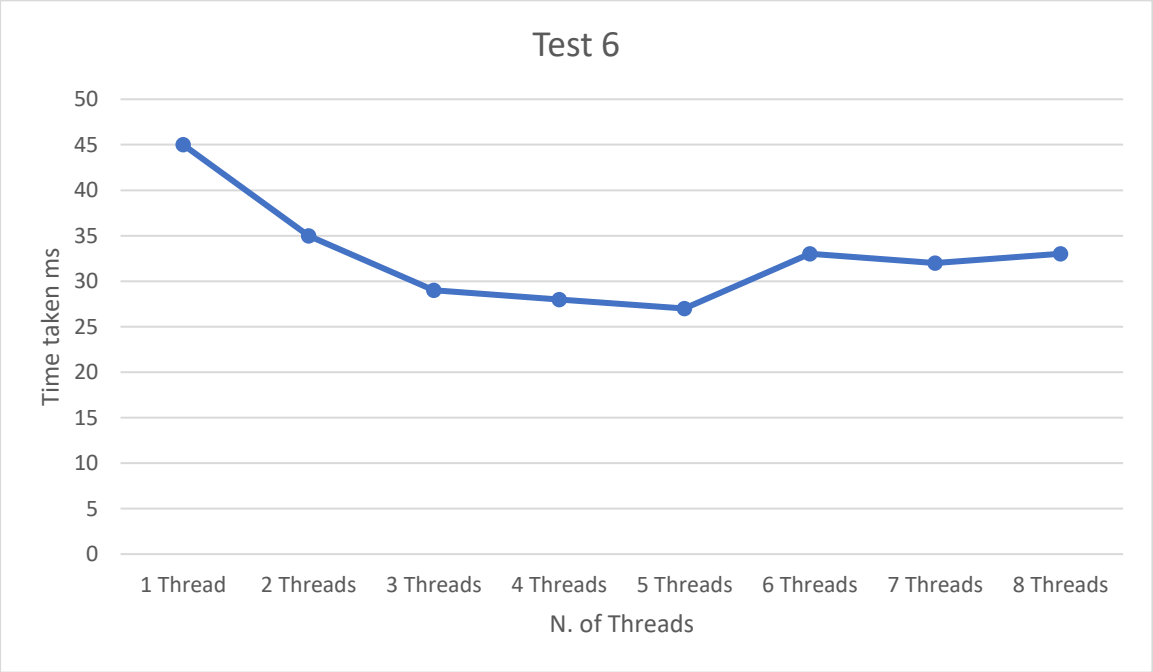

Figure 19 Graph for testing

Here we can realize that when we have a large file and, in that case, we have 248678 paragraphs, and when we input a small number of threads, the program is taking a lot more time comparing to when we used 5 threads.

This is where parallel processing shines. With a large data file, distributing the workload across multiple threads can significantly reduce processing time.

So using a large number of threads (up to a certain limit, specifically the system's logical amount of processors) generally offers substantial performance improvements due to more efficient workload distribution and better utilization of the CPU's capabilities.

Test 7:

We will test in the case where we have a moderate data file 7 which has 5 000 000 lines

(1243388 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	188	102	80	74	69	59	57	57	85.75

Figure 20 Test 7

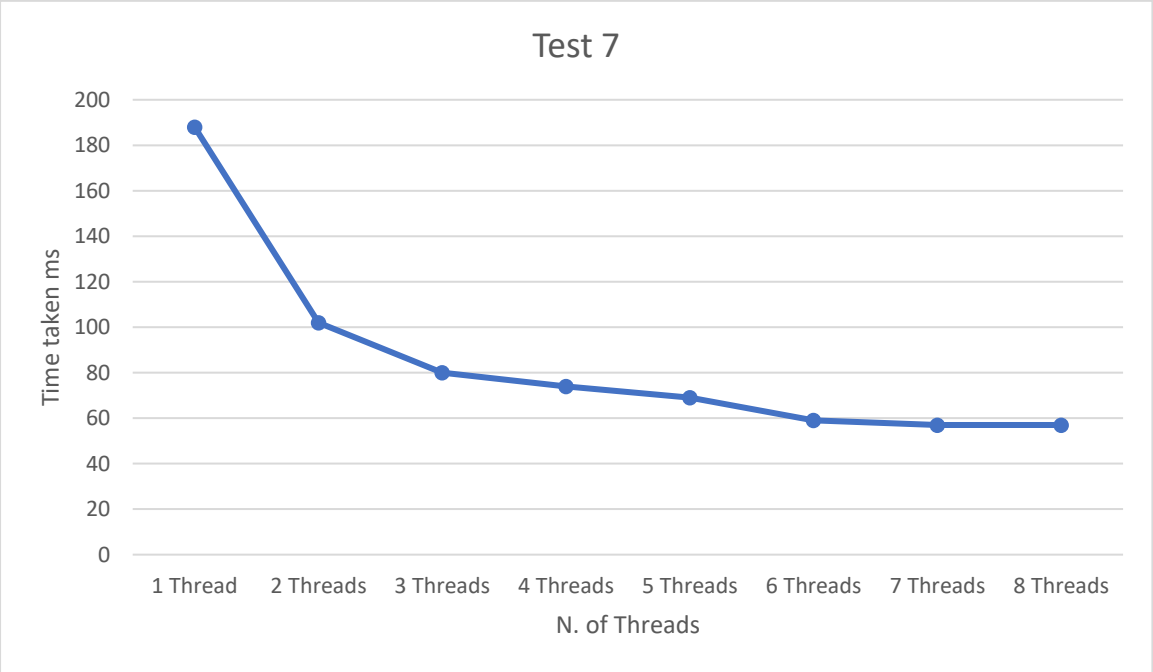

Figure 21 Graph for testing

Here we can realize that when we have a large file and, in that case, we have 248678 paragraphs, and when we input a small number of threads, the program is taking a lot more time comparing to when we used 6 threads.

This is where parallel processing shines. With a large data file, distributing the workload across multiple threads can significantly reduce processing time.

So using a large number of threads (up to a certain limit, specifically the system's logical amount of processors) generally offers substantial performance improvements due to more efficient workload distribution and better utilization of the CPU's capabilities.

Test 8:

We will test in the case where we have a moderate data file 8 which has 10 000 000 lines

(2486777 paragraphs), and we will visualize the results.

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads	Avg
Time taken Ms	452	370	180	261	112	97	190	220	260.25

Figure 22 Test 8

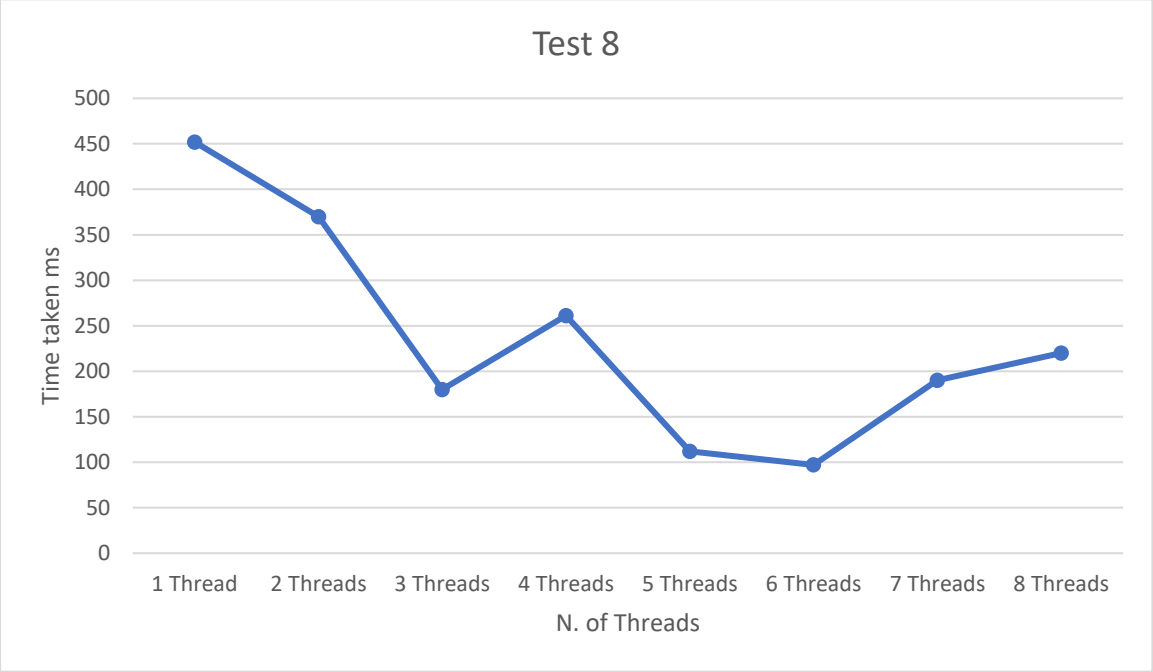

Figure 23 Graph for testing

Despite being faster than 5 threads, this small improvement suggests we might be approaching the limits of how much parallelism the task can effectively use.

With 7 threads, the performance degrades from the optimal time at 5 and 6 threads, possibly due to the overhead of context switching or other system limitations.

The fluctuations in time taken with different numbers of threads indicate that finding the optimal number of threads requires careful testing and consideration of the specific task and system. It's also a reminder that parallel processing performance is not always intuitive and can be influenced by many factors.

Average Table:

N. of Threads	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
Average Time taken	98.25	75.87	48.25	58	39.5	37	48.87	52.25

Figure 24 Table for Average Time

Average Graph:

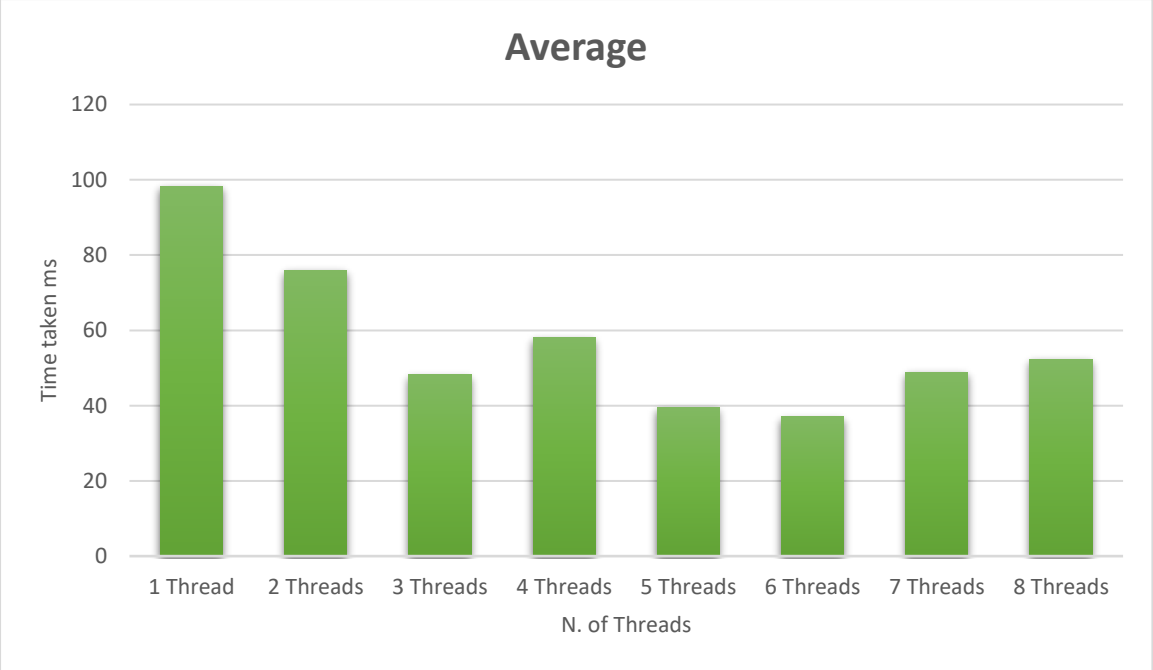

Figure 25 Graph for Average Time

Based on the average execution timetable, it looks like using 6 threads may be the optimal number of threads to use for this program.

- **1 Thread**
    - **Average Time Taken**: 98.25 ms
    - It's the slowest.
- **2 Threads**:
    - **Average Time Taken**: 75.87 ms
    - Introducing parallelism reduces the time taken, showing a benefit over 1 Thread.
- **3 Threads**:
    - **Average Time Taken**: 48.25 ms
    - Further reduction in time suggests that the task scales well with more threads up to this point.
- **4 Threads**:
    - **Average Time Taken**: 58 ms
    - Interestingly, there's an increase in time compared to three threads, which might indicate that the overhead of managing an additional thread outweighs the benefit it provides for this task.
- **5 Threads**:
    - **Average Time Taken**: 39.5 ms
    - This is the second fastest average time, suggesting that this is an optimal number of threads for this particular task on the given hardware.
- **6 Threads**:
    - **Average Time Taken**: 37 ms
    - This is the fastest average time, suggesting that this is the optimal number of threads for this particular task on the given hardware.
- **7 Threads**:
    - **Average Time Taken**: 48.87 ms
    - The time increases again compared to five threads, indicating that adding more threads beyond the optimal number starts to have diminishing returns.
- **8 Threads**:
    - **Average Time Taken**: 52.25 ms
    - Continuing the trend, time increases as more threads are added, likely due to the overhead of context switching

Conclusion:

The program in question performs paragraph counting across multiple text files, leveraging Parallel LINQ to parallelize the process. A performance analysis was conducted to ascertain the optimal threading conditions for the most efficient execution time. Increasing the number of threads has the potential to expedite the program's operations by distributing tasks concurrently. Nevertheless, there are instances where an increase in threads might paradoxically prolong the execution time. Several factors contribute to this:

- **Overhead**: The initiation and management of additional threads can bring about overhead that may undermine performance. This effect is particularly noticeable if threads have a short lifespan or if their quantity is exceedingly high.
- **Contention**: When multiple threads vie for the same resources, such as file I/O operations, contention can occur, resulting in a slowdown.
- **Context Switching**: Performance can suffer from frequent context switches, which occur when the CPU alternates between multiple threads, especially if the workload is unevenly distributed among them.
- **Limited Resources**: A program's demand for resources like CPU cycles or memory—if it surpasses the available hardware capabilities—can mean that adding more threads deteriorates performance rather than improving it.

It is vital to recognize that the ideal thread count is not a constant but varies depending on numerous factors, including the specific nature of the program, the architecture of the system on which it's running, and the characteristics of the workload. Therefore, determining the most effective number of threads requires careful consideration and testing within the context of the particular application in question.

Sources:

- https://www.baeldung.com/cs/servers-threads-number
- https://www.techtarget.com/searchitoperations/tip/Improve-application-performance-with-multithreaded-applications
- https://stackoverflow.com/questions/6086111/plinq-on-concurrentqueue-isnt-multithreading
- https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq
- https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/december/concurrent-affairs-data-parallel-patterns-and-plinq