



KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATICS

CONCURRENT PROGRAMMING PROJECT

Abdelrhman_Ahmed_IFU1

2023

Description:

This C# program performs matrix multiplication using parallel processing to enhance performance. The matrices are read from a file named "vector_data#.txt," and each line in the file contains two matrices separated by a semicolon. The program calculates the product of these matrices using parallel processing with a specified degree of parallelism. The degree of parallelism is determined by user input. The matrices, along with their multiplication result, are displayed, and the total time taken for processing is measured.

Objective:

The objective of this code is to demonstrate and implement parallel matrix multiplication to optimize performance. The program achieves this by:

1. User Input:

- Takes user input for the degree of parallelism, ensuring it is a positive integer.

2. Matrix Data Reading:

- Reads matrix data from the "vector_data#.txt" file.
- Each line in the file represents two matrices separated by a semicolon.

3. Matrix Multiplication:

- Iterates through each line in the file.
- Parses the matrices from the line and displays them.
- Performs matrix multiplication using parallel processing, considering the specified degree of parallelism.
- Displays the resulting matrix.

4. Performance Measurement:

- Records the total time taken for processing using a `Stopwatch`.

5. Output:

- Outputs the matrices (A, B, and the result) along with the total processing time.
- Prints the number of multiplication processes performed.
- Displays the total number of input matrices and the number of result matrices.

6. Exception Handling:

- Handles exceptions, such as errors in reading matrix data from the file.
- Handel cases when the matrices cannot be multiplied.

Source code :

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

class Program
{
    static void Main()
    {
        int count = 0;
        string filepath = "vector_data2.txt";
        Console.Write("Enter the degree of parallelism: ");
        if (int.TryParse(Console.ReadLine(), out int degreeOfParallelism) &&
            degreeOfParallelism > 0)
        {
            Stopwatch totalStopwatch = Stopwatch.StartNew();

            // Call the ReadMatrixData method to read matrix data from a file
            if (ReadMatrixData(filepath, out int[,] matrixA, out int[,] matrixB))
            {
                // Iterate through each line
                foreach (string line in ReadAllLines(filepath))
                {
                    count++;
                    // Split the line into two parts using ';'
                    string[] parts = line.Split(';');

                    // Create matrices
                    matrixA = ParseMatrix(parts[0]);
                    matrixB = ParseMatrix(parts[1]);

                    Console.WriteLine("\nMatrix A:");
                    PrintMatrix(matrixA);

                    Console.WriteLine("\nMatrix B:");
                    PrintMatrix(matrixB);

                    Stopwatch stopwatch = Stopwatch.StartNew();
                    // Perform matrix multiplication
                    int[,] resultMatrix = MultiplyMatrices(matrixA, matrixB,
                        degreeOfParallelism);
                    stopwatch.Stop();

                    Console.WriteLine("\nResult Matrix:");
                    PrintMatrix(resultMatrix);
                }
            }
        }
    }
}
```

```

    }
    else
    {
        Console.WriteLine("Failed to read matrix data from the file.");
    }

    totalStopwatch.Stop();
    Console.WriteLine($"Total time taken for processing:
{totalStopwatch.ElapsedMilliseconds} ms");
}
else
{
    Console.WriteLine("Invalid input for the degree of parallelism.");
}
Console.WriteLine("The number of multiplication process = {0} ",count);
Console.WriteLine("The number of input Matrices = {0} ", count * 2);
Console.WriteLine("The number of result Matrices = {0} ", count);

}

static bool ReadMatrixData(string filePath, out int[,] matrixA, out int[,]
matrixB)
{
    matrixA = null;
    matrixB = null;

    try
    {
        // Read all lines from the file
        foreach (string line in ReadAllLines(filePath))
        {
            // Split the line into two parts using ';'
            string[] parts = line.Split(';');

            // Create matrices
            matrixA = ParseMatrix(parts[0]);
            matrixB = ParseMatrix(parts[1]);
        }

        return true;
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error reading matrix data: {ex.Message}");
        return false;
    }
}

static IEnumerable<string> ReadAllLines(string filePath)
{
    using (var reader = new StreamReader(filePath))
    {
        while (!reader.EndOfStream)
        {
            yield return reader.ReadLine();
        }
    }
}

```

```

    }
}

static int[,] ParseMatrix(string line)
{
    var values = line.Split(new char[] { ',', ' ' },
StringSplitOptions.RemoveEmptyEntries);
    int rows = values.Length / 3; // Assuming each vector has 3 components
(change if needed)
    int cols = 3; // Assuming each vector has 3 components (change if needed)

    int[,] matrix = new int[rows, cols];

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            matrix[i, j] = int.Parse(values[i * cols + j]);
        }
    }

    return matrix;
}

static int[,] MultiplyMatrices(int[,] matrixA, int[,] matrixB, int
degreeOfParallelism)
{
    int rowsA = matrixA.GetLength(0);
    int colsA = matrixA.GetLength(1);
    int colsB = matrixB.GetLength(1);

    if (colsA != matrixB.GetLength(0))
    {
        throw new InvalidOperationException("Matrices cannot be multiplied.
Number of columns in Matrix A must be equal to the number of rows in Matrix B.");
    }

    int[,] resultMatrix = new int[rowsA, colsB];

    IEnumerable<int> Data(int start, int count) => Enumerable.Range(start,
count);

    var query = from i in Data(0, rowsA)
                from j in Data(0, colsB)
                select new
                {
                    i,
                    j,
                    value = Data(0, colsA)
                        .AsParallel()
                        .WithDegreeOfParallelism(degreeOfParallelism)
                        .Sum(k => matrixA[i, k] * matrixB[k, j])
                };

    foreach (var item in query)
    {
        resultMatrix[item.i, item.j] = item.value;
    }
}

```

```

        return resultMatrix;
    }

    static void PrintMatrix(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);

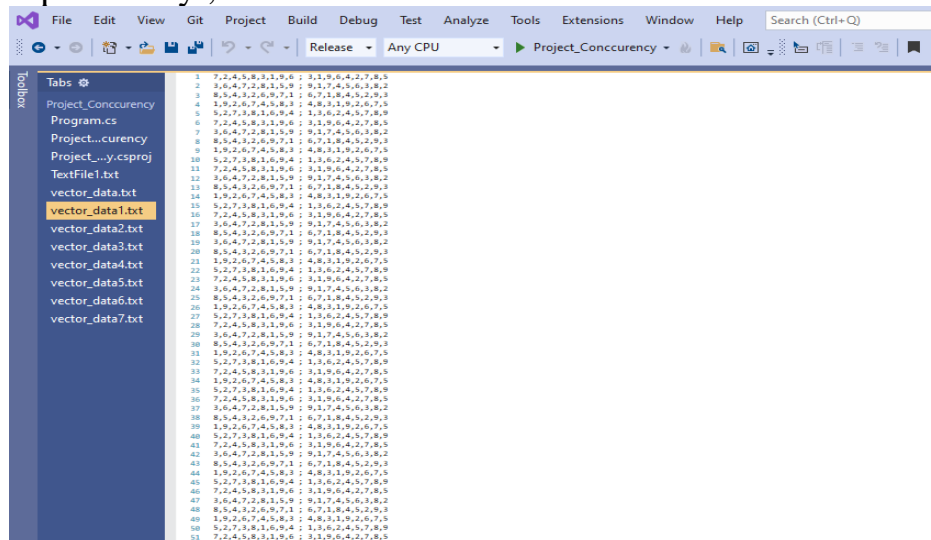
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                Console.Write($"{matrix[i, j]} ");
            }
            Console.WriteLine();
        }
    }
}

```

The code is designed to showcase the benefits of parallel processing in matrix multiplication, with a focus on readability and user interaction.

Input :

The input is a group of vectors that form a matrix and every two matrices are separated by ;



Output:

```
Enter the degree of parallelism: 3

Matrix A:
3 6 4
7 2 8
1 5 9

Matrix B:
9 1 7
4 5 6
3 8 2

Result Matrix:
63 65 65
95 81 77
56 98 55

Matrix A:
8 5 4
3 2 6
9 7 1

Matrix B:
6 7 1
8 4 5
2 9 3

Result Matrix:
96 112 45
46 83 31
112 100 47

Matrix A:
1 9 2
6 7 4
5 8 3

Matrix B:

Result Matrix:
58 79 103
26 49 67
52 86 117

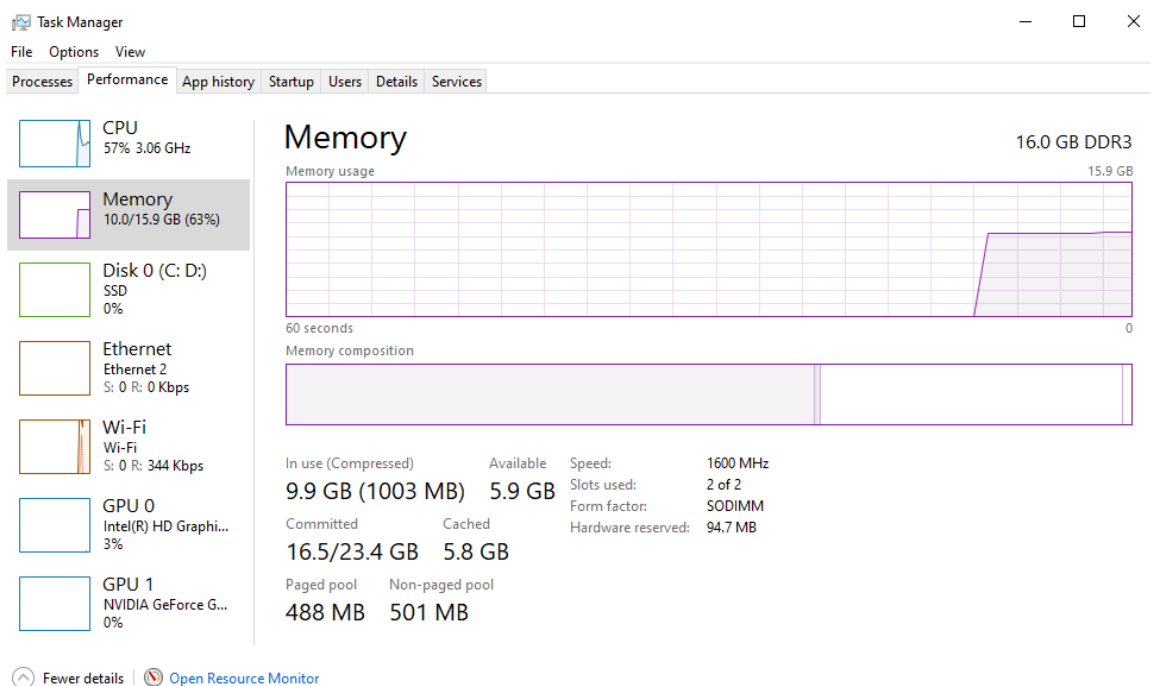
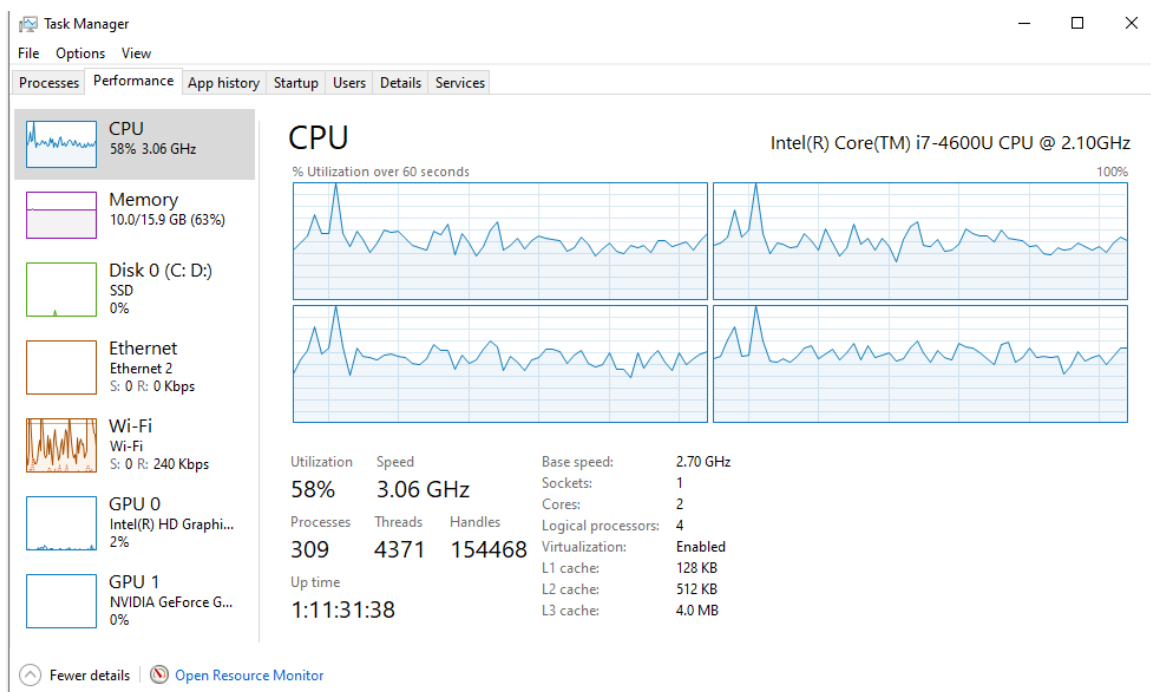
Total time taken for processing: 1055 ms
The number of multiplication process = 114
The number of input Matrices = 228
The number of result Matrices = 114

C:\Users\pc\source\repos\Project_Concurrency\Project_Concurrency\bin\Release\net5.0\Project_Concurrency.exe (process 24320) exited with code 0.
Press any key to close this window . . .
```

5. Performance Analysis:

Below I have mentioned all the speed tests conducted to determine the optimum number of threads to use to perform this task.

Computer Specifications:



Device name PC

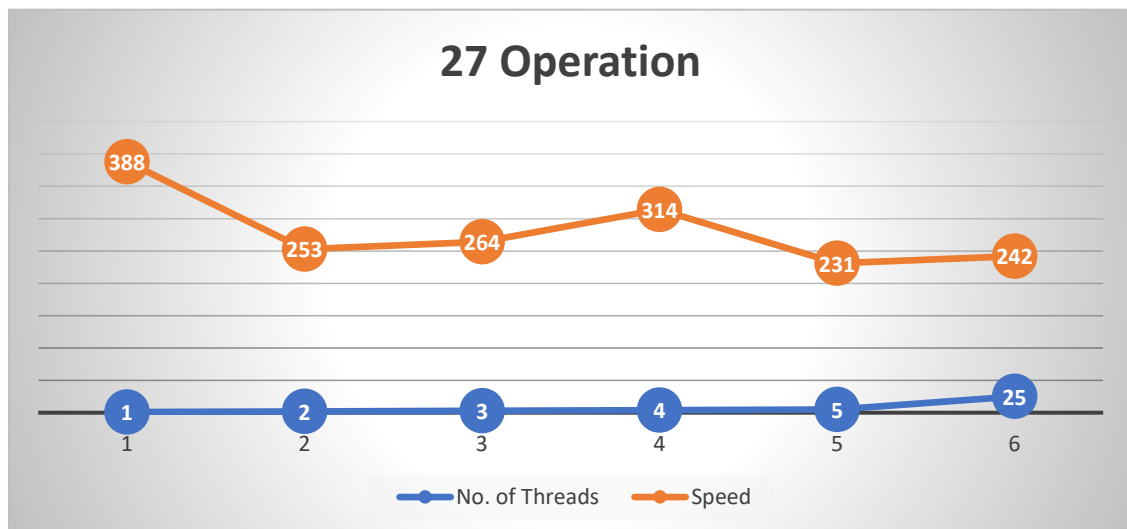
Processor Intel(R) Core(TM) i7-4600U CPU @ 2.10 GHz

Installed RAM 16.00 GB (9.8 GB usable)

System type64-bit operating system, x64-based processor

The Intel Core i7-4600U processor operates as a dual-core processor featuring hyper-threading technology, enabling it to handle four threads concurrently. When determining the optimal degree of parallelism for tasks involving parallel processing, it is advisable to focus on the number of physical cores rather than the total thread count.

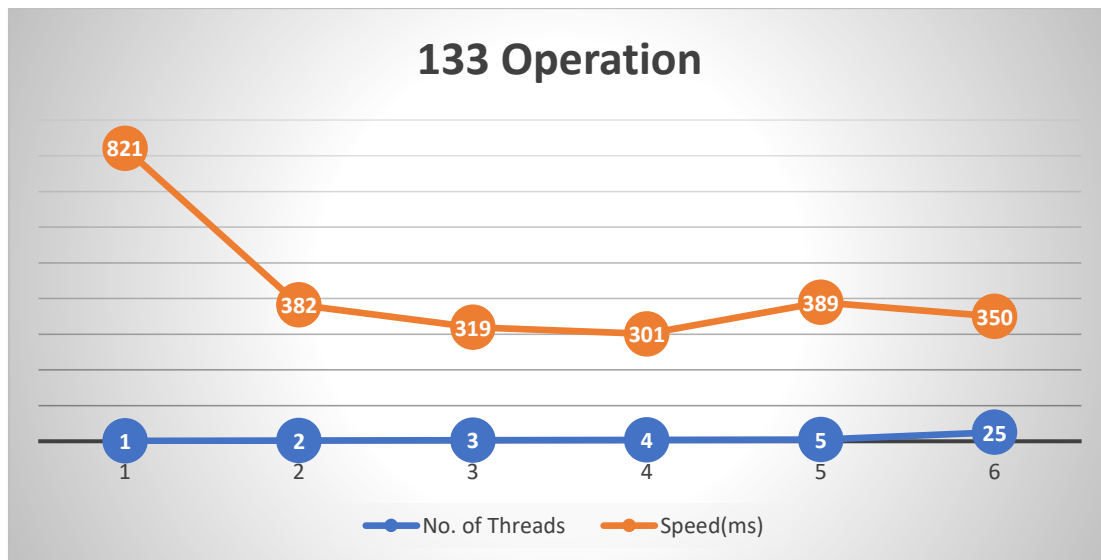
1st Test:



No. of Threads	Speed	Number of operations
1	388	27
2	253	//
3	264	//
4	314	//
5	231	//
25	242	//

Observation: The program process the input data faster while using more number of threads

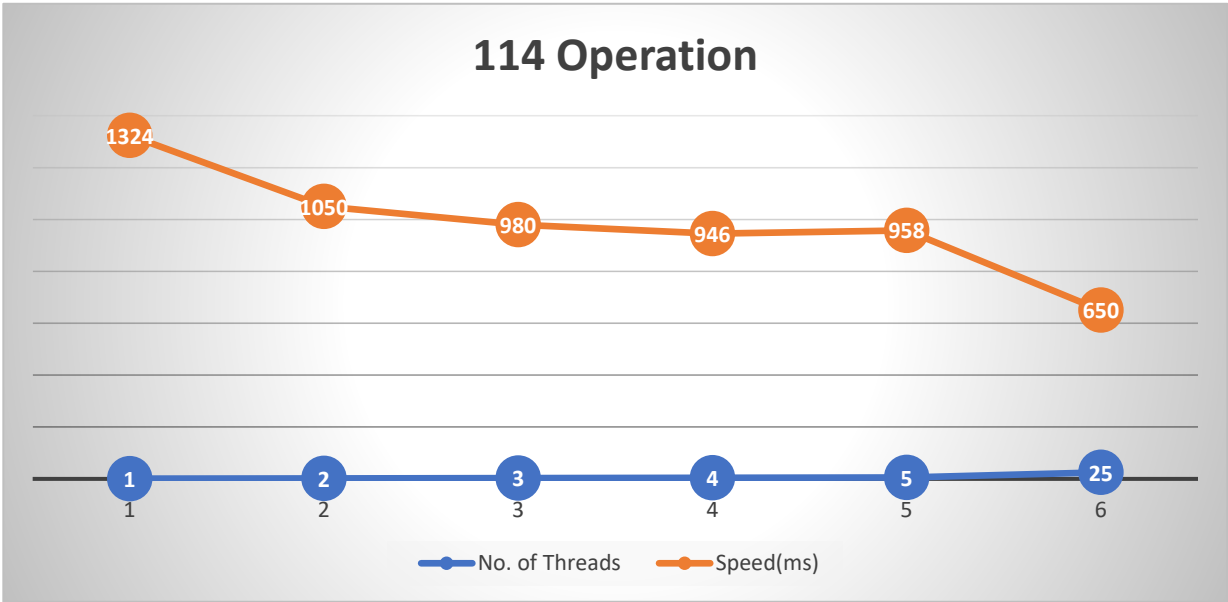
2nd Test:



No. of Threads	Speed(ms)	Number of operations
1	821	133
2	382	//
3	319	//
4	301	//
5	389	//
25	350	//

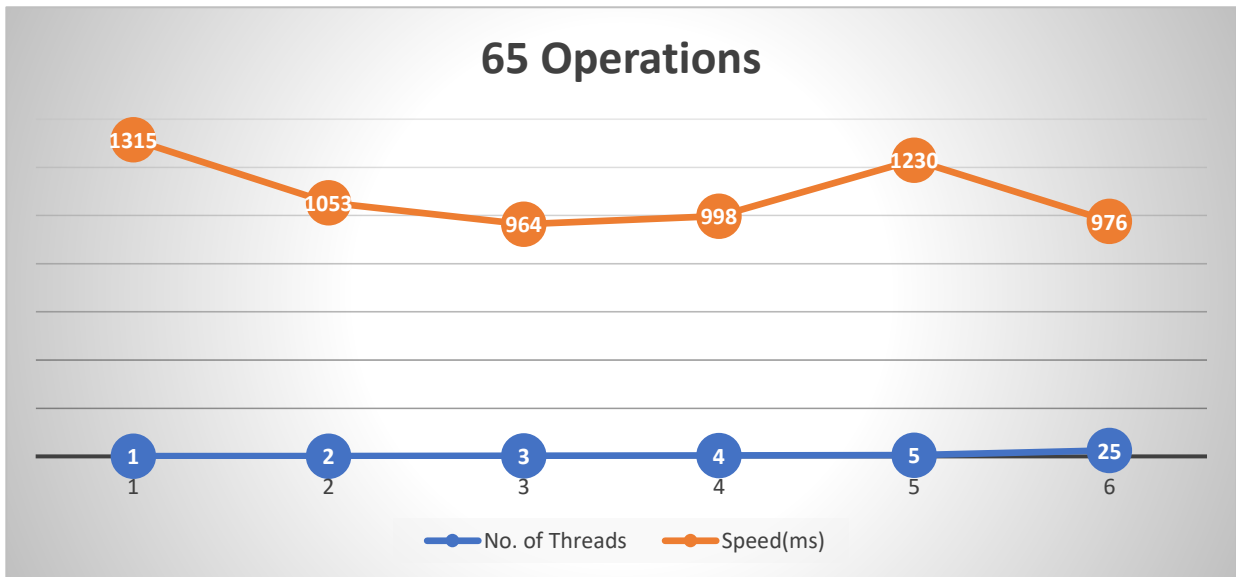
When we increase the number of files in the process, we can observe changes in the processing times the program is getting more time to finish processing

3rd Test:



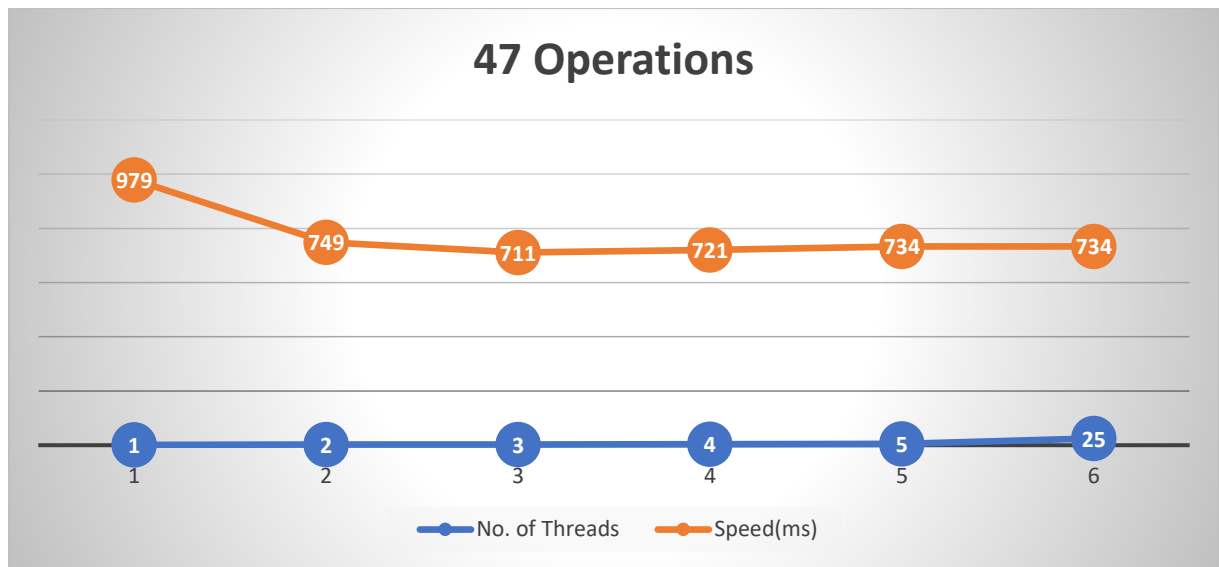
No. of Threads	Speed(ms)	Number of operations
1	1324	114
2	1050	//
3	980	//
4	946	//
5	958	//
25	650	//

4th Test:



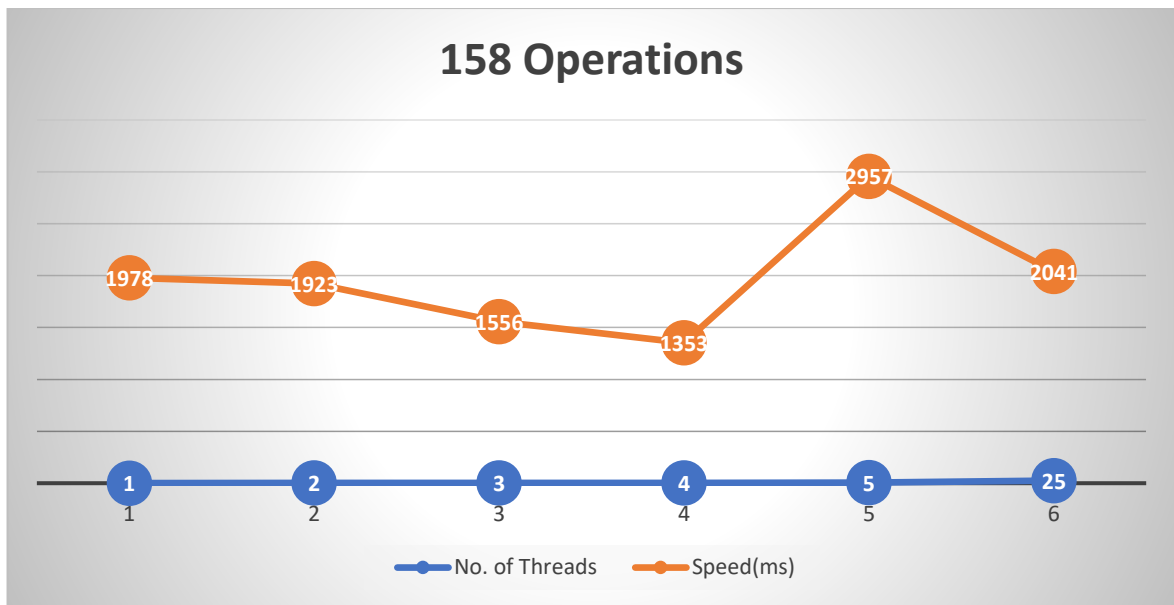
No. of Threads	Speed(ms)	Number of operations
1	1315	65
2	1053	//
3	964	//
4	998	//
5	1230	//
25	976	//

5th Test:



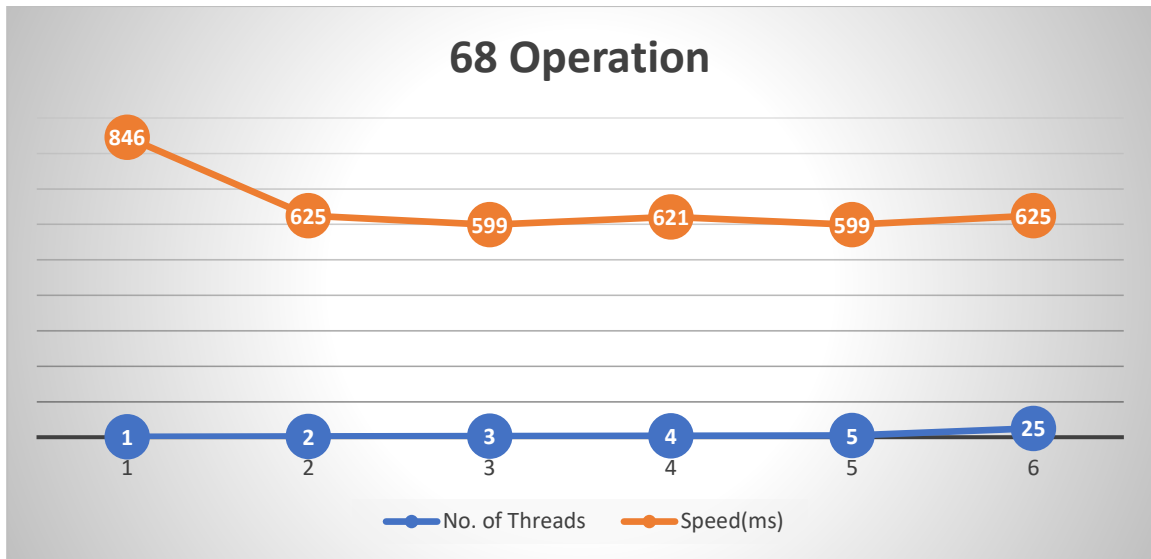
No. of Threads	Speed(ms)	Number of operations
1	979	47
2	749	//
3	711	//
4	721	//
5	734	//
25	734	//

6th Test:



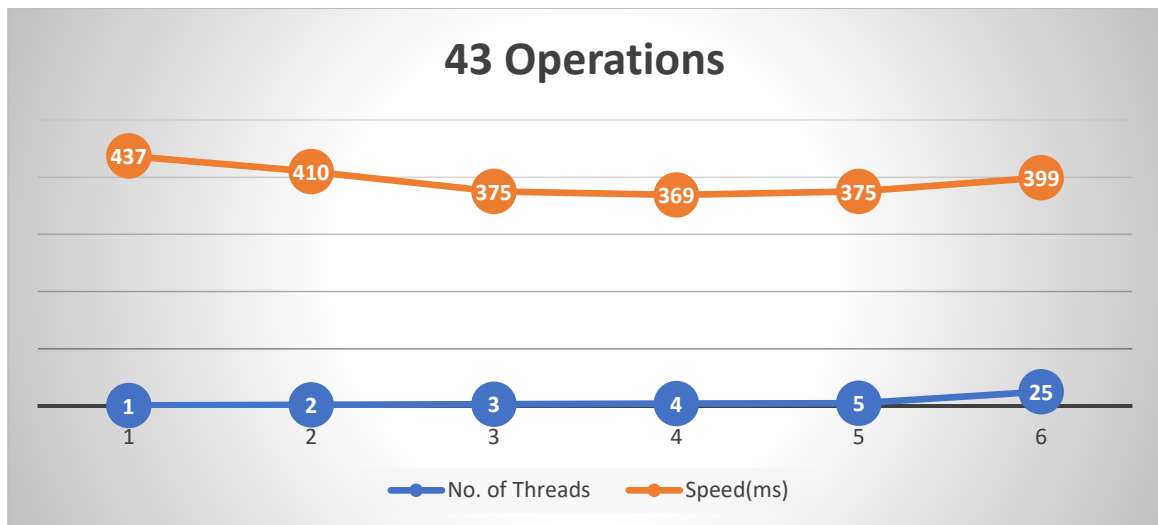
No. of Threads	Speed(ms)	Number of operations
1	1978	158
2	1923	//
3	1556	//
4	1353	//
5	2957	//
25	2041	//

7th Test:



No. of Threads	Speed(ms)	Number of operations
1	846	68
2	625	//
3	599	//
4	621	//
5	599	//
25	625	//

8th Test:

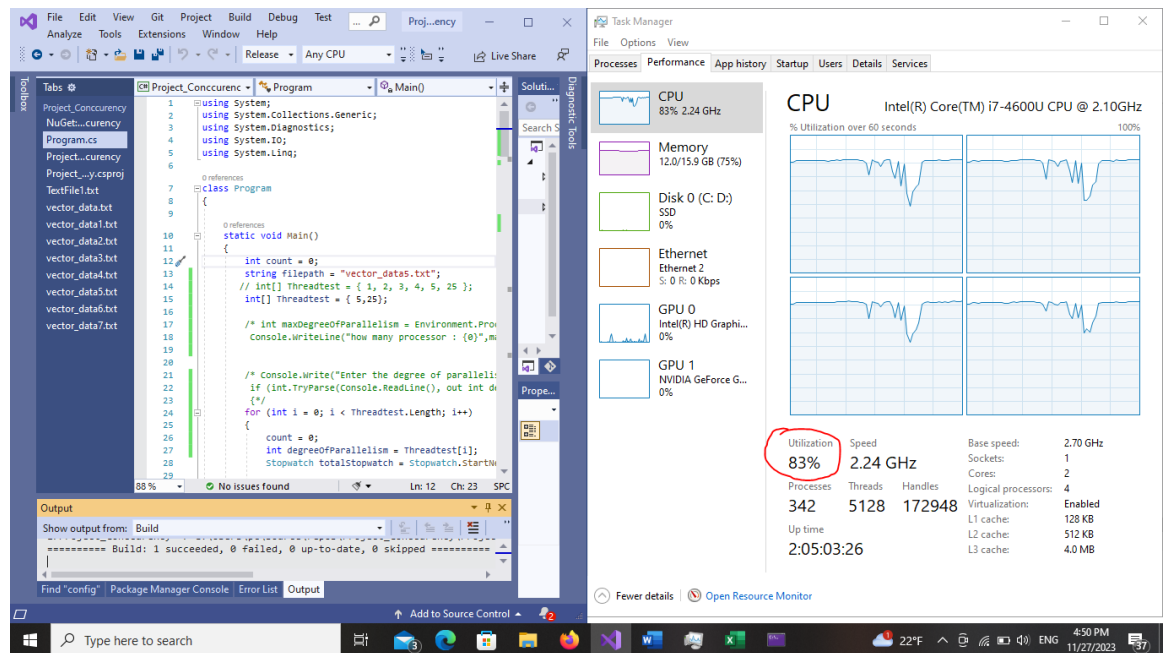


No. of Threads	Speed(ms)	Number of operations
1	437	43
2	410	//
3	375	//
4	369	//
5	375	//
25	399	//

Conclusion:

When you use an appropriate number of threads, you can achieve optimal parallelism and performance. Here are the key benefits:

1. **Maximum CPU Utilization:** The number of threads matches the number of available processors or cores, allowing each thread to execute on a separate core simultaneously. This maximizes CPU utilization, leading to efficient use of computational resources.

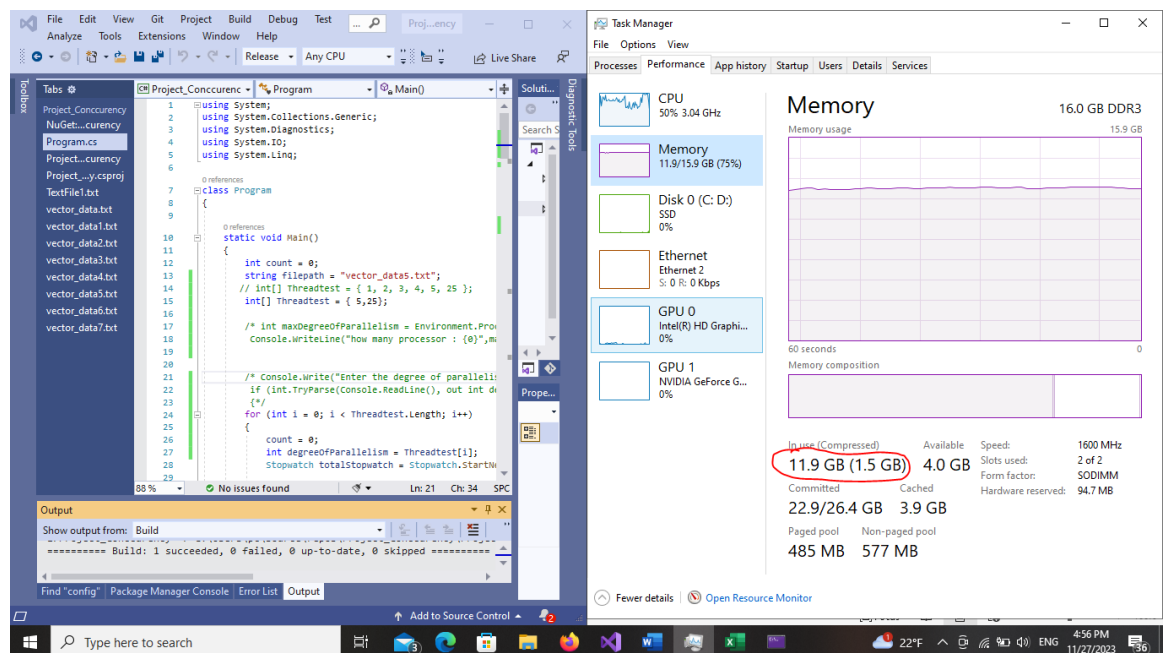


2. **Reduced Contention:** With an optimal number of threads, there is less contention for shared resources like the CPU, memory, and cache. This reduces the likelihood of contention delays and allows each thread to execute without unnecessary waiting.
3. **Improved Throughput:** The parallel execution of threads can lead to improved throughput, especially for CPU-bound tasks. Tasks that can be divided into independent subtasks can be executed concurrently, leading to faster completion times.
4. **Scalability:** Properly tuning the number of threads ensures scalability, meaning that as the workload increases, the application can efficiently use additional threads to maintain or improve performance. Scalability is crucial for handling larger datasets or more significant computational loads.

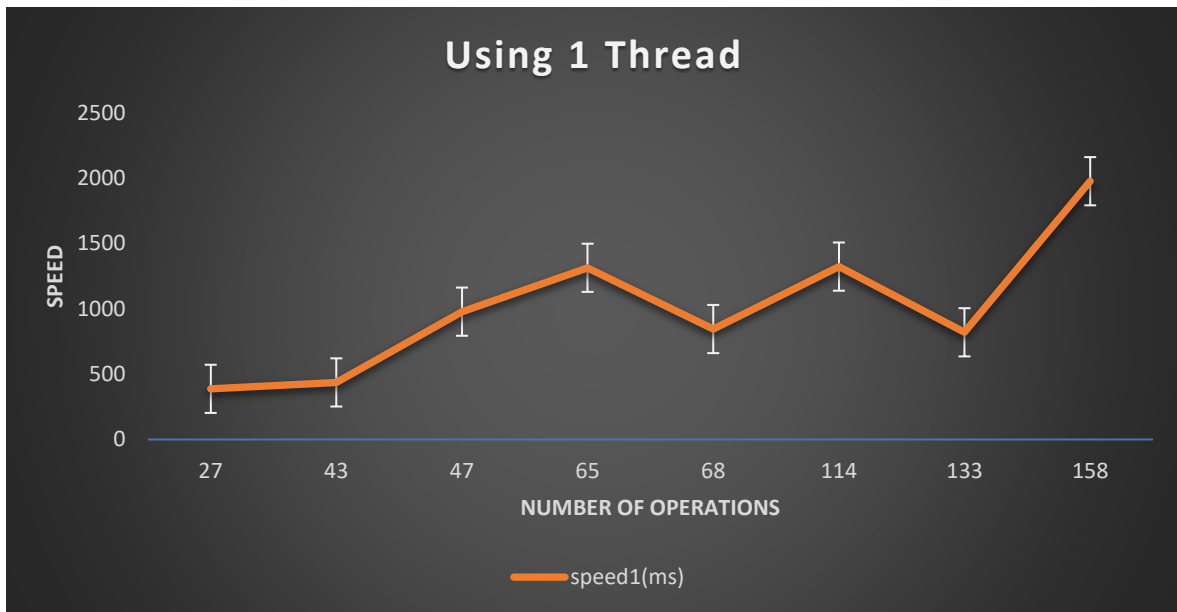
When you use a number of threads more than the available processors, you may experience increased contention for CPU resources, which can lead to performance degradation instead of improvement. This is known as oversubscription. Here are some key points to consider:

1. **Context Switching Overhead:** When there are more threads than processors, the operating system needs to frequently switch between threads to give each one a turn to execute. This context switching has an associated overhead, and if it happens too frequently, it can result in slower overall performance.

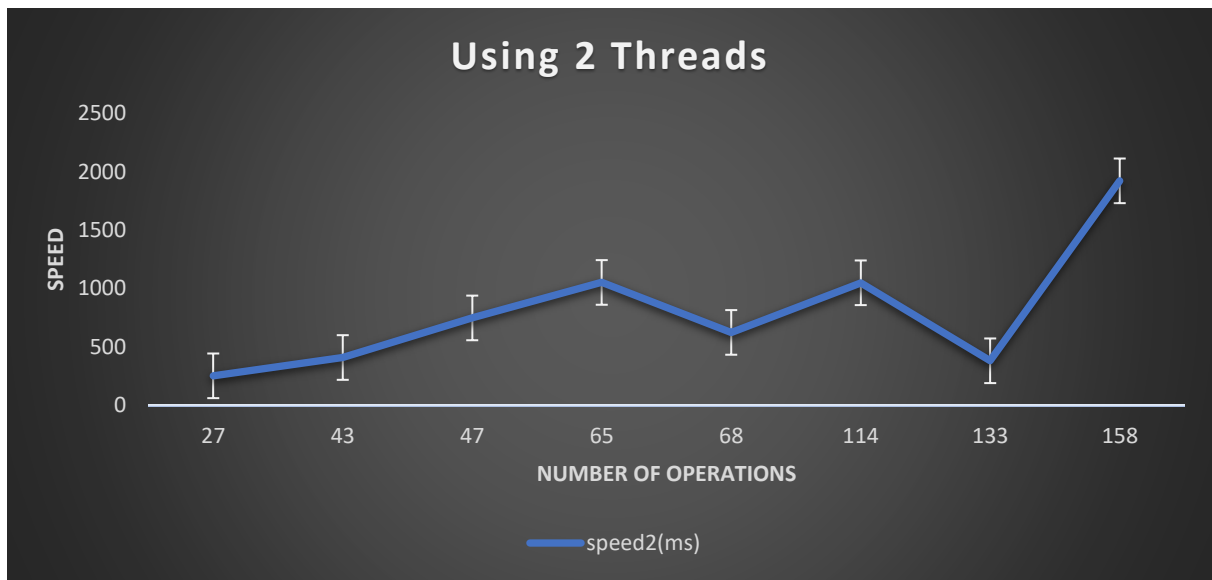
2. **Contention for Resources:** Threads compete for shared resources such as the CPU, memory, and cache. With more threads than processors, contention for these resources increases, leading to contention delays and potential inefficiencies.
3. **Diminished Parallelism:** While the goal of parallel programming is to execute tasks concurrently for performance improvement, oversubscription can lead to diminished parallelism due to increased contention and context switching.
4. **Increased Memory Usage:** Each thread has its own stack and other associated resources. With a large number of threads, memory usage can increase, potentially leading to increased paging and decreased performance.



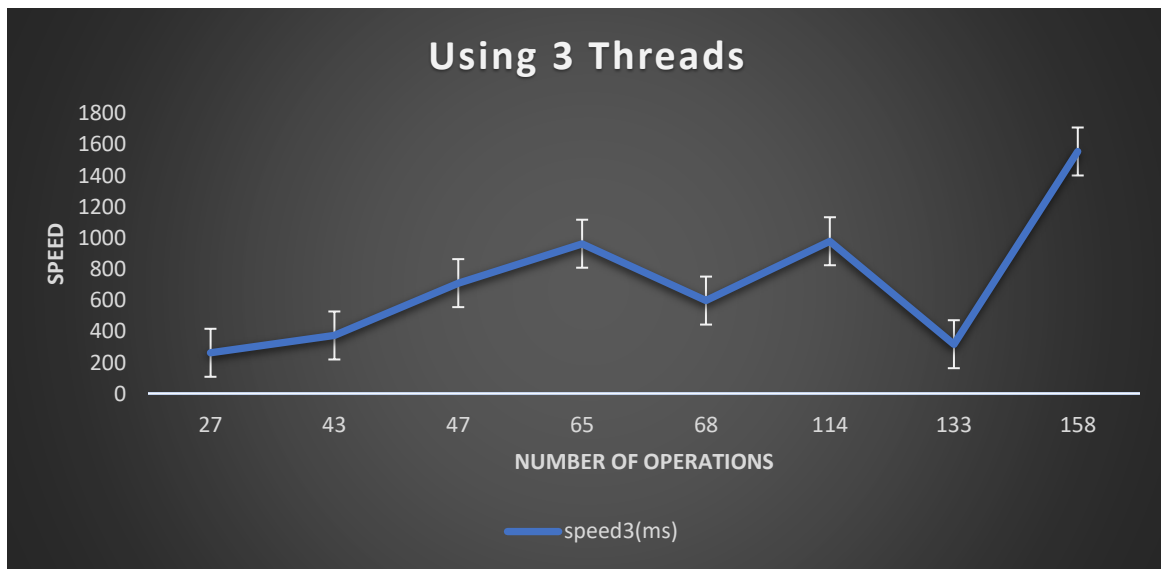
Threads graph:



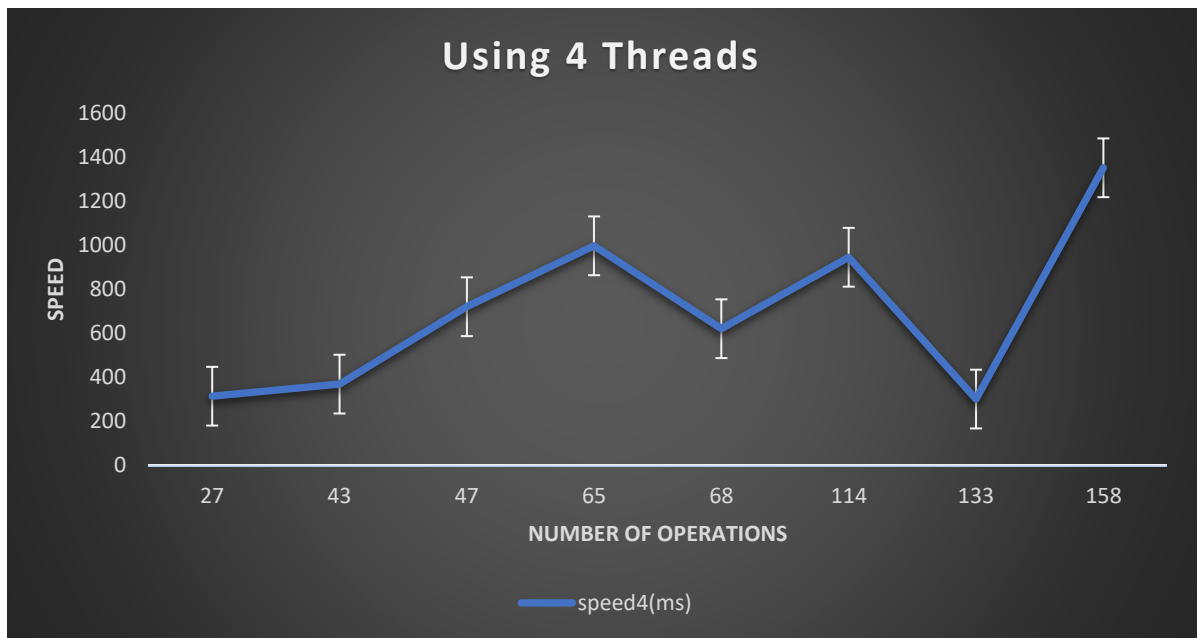
No of operations	speed(ms)
27	388
43	437
47	979
65	1315
68	846
114	1324
133	821
158	1978



No of operations	speed(ms)
27	253
43	410
47	749
65	1053
68	625
114	1050
133	382
158	1923

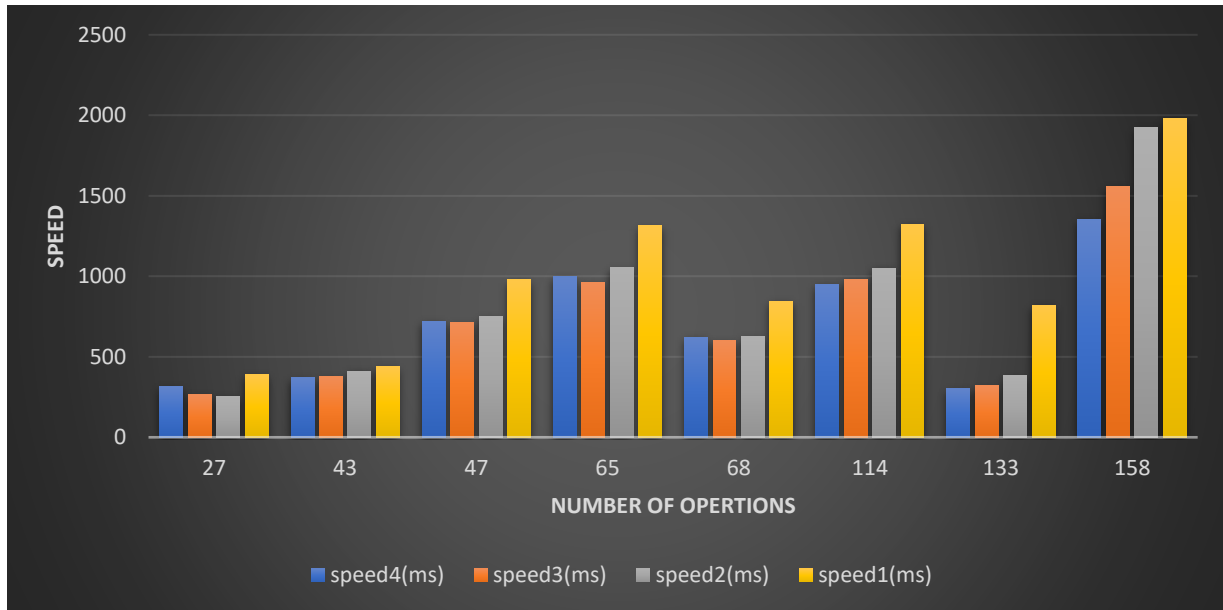


No of operations	speed(ms)
27	264
43	375
47	711
65	964
68	599
114	980
133	319
158	1556



No of operations	speed(ms)
27	314
43	369
47	721
65	998
68	621
114	946
133	301
158	1353

Combined graph :



Conclusion:

1. The appropriate number of threads in a parallelized program should generally be less than or equal to the number of processors or cores available on the device. This ensures optimal utilization of the available hardware resources and helps prevent potential performance degradation due to excessive thread contention.
2. When conducting multiple tests or experiments involving parallel programming, it's crucial to consider the thermal implications. Intensive parallel computations can lead to increased CPU usage, causing the device to generate more heat. In such scenarios, the device's cooling system, often including fans, may operate at maximum capacity to dissipate heat. Prolonged exposure to high temperatures can negatively impact overall system stability and longevity (personal experiment).
3. Using a number of threads greater than the number of processors may not provide proportional performance gains and, in fact, could lead to diminishing returns or even performance degradation. Additionally, it may exacerbate thermal issues, as the system attempts to manage the increased computational load.

In summary, it is advisable to carefully choose the number of threads in parallel programming to align with the available hardware resources, considering both performance optimization and thermal considerations to ensure a balanced and efficient execution of parallel tasks in my program the appropriate number of threads is 3.