Contents

Objective

This assignment had the following objectives:

- Implementing programs in C# language
- Using threading and multithreading operations
- Measuring the time complexity of programs
- Implementing the algorithm of 0/1 Knapsack problem in C#
- Writing clean and efficient code to minimize time complexity

Problem Statement

Serial vs. Multithreaded Applications

The 0/1 Knapsack Problem

Given n objects, with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, with Knapsack capacity M,

Maximize $\sum_{1 \le i \le n} v_i x_i$ ***subject to*** $\sum_{1 \le i \le n} w_i x_i \le M$, *for* $x_i = 0$ **or 1**, $1 \le i \le n$

Implement a C# application to find **optimal** solution of the 0/1 Knapsack problem using following three program design techniques:

a) Serial (single threaded) application
b) Multithreaded application (using Thread objects)
c) Multithreaded application (using a ThreadPool)

Compute the execution time of these three applications using the following data:

There are **10** different items and the Knapsack capacity is **67**.
w_1=23, w_2=26, w_3=20, w_4=18, w_5=32, w_6=27, w_7=29, w_8=26, w_9=30, w_{10}=27
v_1=505, v_2=352, v_3=458, v_4=220, v_5=354, v_6=414, v_7=498, v_8=545, v_9=473, v_{10}=543

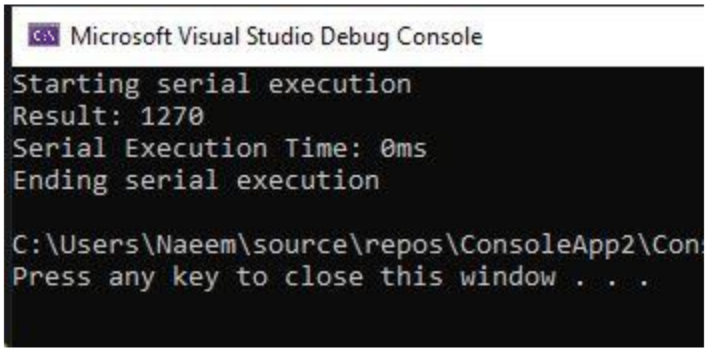
Prepare a **comparison table** to compare the performance of these three techniques.
Conclude your findings.
(Hint: Stopwatch class can be used to log execution time in C#).

Source Code

A) Serial application code:

```
using System;
using System.Diagnostics;
using System.Threading;

class PDCAssignment
{
    static int knapsack(int maxWeight, int[] weights, int[] values, int n)
    {

        if (maxWeight == 0 || n == 0)
        {
            return 0;
        }
        if (weights[n - 1] > maxWeight)
        {
            return knapsack(maxWeight, weights, values, (n - 1));
        }
        else
        {
            return Math.Max(values[n - 1] + knapsack(maxWeight - weights[n - 1],
weights, values, (n - 1)),
                knapsack(maxWeight, weights, values, (n - 1)));
        }
    }

    public static void Main(string[] args)
    {
        Console.WriteLine("Starting serial execution");
        Stopwatch watch;
        watch = new Stopwatch();
        watch.Start();

        int[] weights = new int[] { 23, 26, 20, 18, 32, 27, 29, 26, 30, 27 };
        int[] values = new int[] { 505, 352, 458, 220, 354, 414, 498, 545, 473, 543
};
        int maxWeight = 67;
        int n = values.Length;

        int result = knapsack(maxWeight, weights, values, n);
        Console.WriteLine("Result: " + result);
        watch.Stop();
        Console.WriteLine("Serial Execution Time: " +
        watch.Elapsed.Milliseconds.ToString() + "ms");
        Console.WriteLine("Ending serial execution");

    }
}
```

Output:



```
Starting serial execution
Result: 1270
Serial Execution Time: 0ms
Ending serial execution

C:\Users\Naeem\source\repos\ConsoleApp2\Con
Press any key to close this window . . .
```

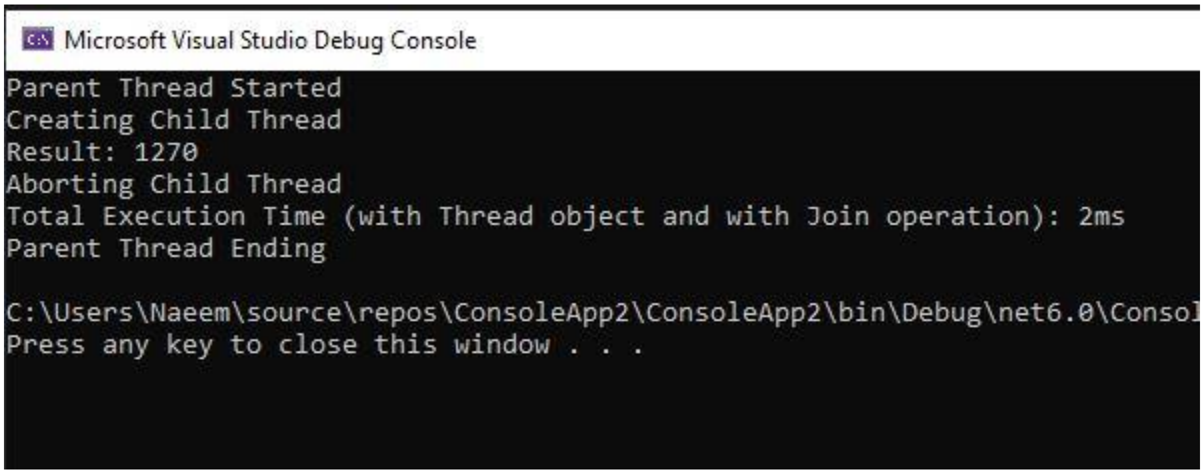
B) Multithreaded application (using Thread object) (with Join operation):

```
using System.Diagnostics;


class PDCAssignment
{
    static int knapsack(int maxWeight, int[] weights, int[] values, int n)
    {
        if (maxWeight == 0 || n == 0)
        {
            return 0;
        }
        if (weights[n - 1] > maxWeight)
        {
            return knapsack(maxWeight, weights, values, (n - 1));
        }
        else
        {
            return Math.Max(values[n - 1] + knapsack(maxWeight - weights[n - 1],
weights, values, (n - 1)),
                knapsack(maxWeight, weights, values, (n - 1)));
        }
    }

    static void driverProgram()
    {
        object result = 0;
        int[] weights = new int[] { 23, 26, 20, 18, 32, 27, 29, 26, 30, 27 };
        int[] values = new int[] { 505, 352, 458, 220, 354, 414, 498, 545, 473, 543
};
        int maxWeight = 67;
        int n = values.Length;

        result = knapsack(maxWeight, weights, values, n);
        Console.WriteLine("Result: " + result);
        Console.WriteLine("Aborting Child Thread");
        Thread.CurrentThread.Interrupt();                              //Abort()
function has become obsolete
    }


    public static void Main(string[] args)
    {
        Console.WriteLine("Parent Thread Started");
        Stopwatch watch;
        watch = new Stopwatch();
        watch.Start();
        Thread.CurrentThread.Name = "Parent";

        Console.WriteLine("Creating Child Thread");
        Thread knapsackSolution = new Thread(driverProgram);
        knapsackSolution.Start();
        if (Thread.CurrentThread.Name == "Parent")
        {
            knapsackSolution.Join();
        }


        watch.Stop();
        Console.WriteLine("Total Execution Time (with Thread object and with Join
operation): " +
        watch.Elapsed.Milliseconds.ToString() + "ms");
        Console.WriteLine("Parent Thread Ending");

    }
}
```

Note: Firstly, I implemented another function (driverProgram) to call the knapsack function instead of directly calling it from Main because I had to store the result in a variable first. Calling the knapsack function directly only returns a value and obviously, threads can't return a value, they must write it in a variable. Implementing the driverProgram function also allowed me the freedom to completely separate the data needed for knapsack function from Main which makes the app clean and well-organized, making sure the Parent thread is not executing any of the child thread's code.

Secondly, implementing the Abort() operation was throwing a runtime error. The main issue is that this function has become completely obsolete now. I tried to disable the warning using pragma but that did not work either. Also, try-catch blocks were also implemented with the Abort function but obviously, since the Abort function wasn't working, I removed those blocks as well. The next best thing that I found was the Interrupt function as it had been mentioned on a lot of forums so I chose to replace the Abort function with that.

Output:



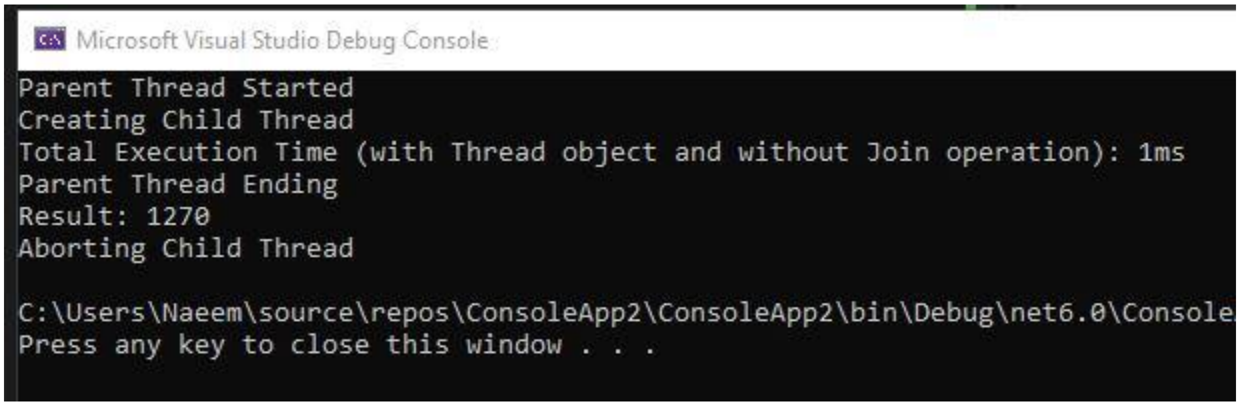
```
Parent Thread Started
Creating Child Thread
Result: 1270
Aborting Child Thread
Total Execution Time (with Thread object and with Join operation): 2ms
Parent Thread Ending

C:\Users\Naeem\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net6.0\Consol
Press any key to close this window . . .
```

B) Multithreaded application (using Thread object) (without Join operation):

```
using System.Diagnostics;

class PDCAssignment
{
    static int knapsack(int maxWeight, int[] weights, int[] values, int n)
    {
        if (maxWeight == 0 || n == 0)
        {
            return 0;
        }
        if (weights[n - 1] > maxWeight)
        {
            return knapsack(maxWeight, weights, values, (n - 1));
        }
        else
        {
            return Math.Max(values[n - 1] + knapsack(maxWeight - weights[n - 1],
weights, values, (n - 1)),
                knapsack(maxWeight, weights, values, (n - 1)));
        }
    }

    static void driverProgram()
    {
        object result = 0;
        int[] weights = new int[] { 23, 26, 20, 18, 32, 27, 29, 26, 30, 27 };
        int[] values = new int[] { 505, 352, 458, 220, 354, 414, 498, 545, 473, 543
};
        int maxWeight = 67;
        int n = values.Length;

        result = knapsack(maxWeight, weights, values, n);
        Console.WriteLine("Result: " + result);
        Console.WriteLine("Aborting Child Thread");
        Thread.CurrentThread.Interrupt();                              //Abort()
function has become obsolete
    }


    public static void Main(string[] args)
    {
        Console.WriteLine("Parent Thread Started");
        Stopwatch watch;
        watch = new Stopwatch();
        watch.Start();

        Console.WriteLine("Creating Child Thread");
        Thread knapsackSolution = new Thread(driverProgram);
        knapsackSolution.Start();

        watch.Stop();
        Console.WriteLine("Total Execution Time (with Thread object and with Join
operation): " +
        watch.Elapsed.Milliseconds.ToString() + "ms");
        Console.WriteLine("Parent Thread Ending");

    }
}
```

Note: Since there's no Join operation being performed in this one, there's no need to assign a name to the parent thread either.

Output:



```
Parent Thread Started
Creating Child Thread
Total Execution Time (with Thread object and without Join operation): 1ms
Parent Thread Ending
Result: 1270
Aborting Child Thread

C:\Users\Naeem\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net6.0\Console
Press any key to close this window . . .
```

C) Multithreaded application (using ThreadPool) (without Join operation):

```
using System.Diagnostics;

class PDCAssignment
{
    static int knapsack(int maxWeight, int[] weights, int[] values, int n)
    {
        if (maxWeight == 0 || n == 0)
        {
            return 0;
        }
        if (weights[n - 1] > maxWeight)
        {
            return knapsack(maxWeight, weights, values, (n - 1));
        }
        else
        {
            return Math.Max(values[n - 1] + knapsack(maxWeight - weights[n - 1],
weights, values, (n - 1)),
                knapsack(maxWeight, weights, values, (n - 1)));
        }
    }

    static void driverProgram(object state)
    {
        object result = 0;
        int[] weights = new int[] { 23, 26, 20, 18, 32, 27, 29, 26, 30, 27 };
        int[] values = new int[] { 505, 352, 458, 220, 354, 414, 498, 545, 473, 543
};
        int maxWeight = 67;
        int n = values.Length;

        result = knapsack(maxWeight, weights, values, n);
        Console.WriteLine("Result: " + result);
        Console.WriteLine("Thread From ThreadPool Released");
        //Thread.CurrentThread.Interrupt();                    //Interrupting
thread from threadpool stopped execution of method

    }


    public static void Main(string[] args)
    {
        Console.WriteLine("Parent Thread Started");
        Stopwatch watch;
        watch = new Stopwatch();
        watch.Start();

        Console.WriteLine("Assigning Task To Thread From ThreadPool");
        ThreadPool.QueueUserWorkItem(new WaitCallback(driverProgram));

        watch.Stop();
        Console.WriteLine("Total Execution Time (with ThreadPool Thread and without 
Join operation): " +
        watch.Elapsed.Milliseconds.ToString() + "ms");
        Console.WriteLine("Parent Thread Ending");

    }
}
```

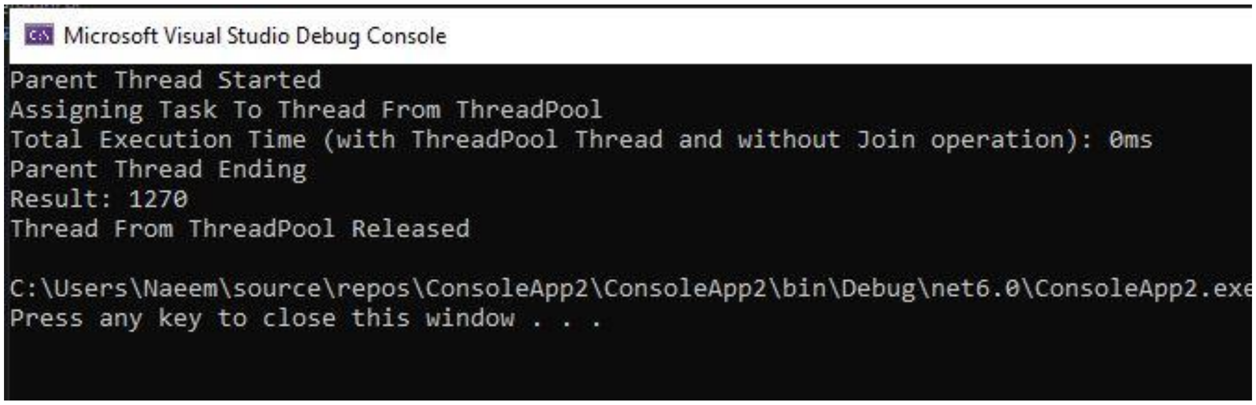
Note: Firstly, interrupting the ThreadPool thread resulted in no output from that thread and only the parent thread was showing results. Hence the Interrupt function has been commented out.

Secondly, I had to add the "object state" argument for driverProgram function when using ThreadPool because without it, the IDE was showing an error.

Thirdly, since this implementation is without the Join operation, the parent thread usually ended before the ThreadPool thread due to which I think the time measurement might be a little off. I tried implementing an object of the Stopwatch inside the driverProgram function but when I did that, the ThreadPool thread would output no results at all.

Lastly, I tried researching how to implement Join on a ThreadPool thread and I found people using quite advanced concepts to achieve that. Since, I'm completely new to C#, I was unable to implement those in the given time.

Output:



```
Parent Thread Started
Assigning Task To Thread From ThreadPool
Total Execution Time (with ThreadPool Thread and without Join operation): 0ms
Parent Thread Ending
Result: 1270
Thread From ThreadPool Released

C:\Users\Naeem\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net6.0\ConsoleApp2.exe
Press any key to close this window . . .
```

Comparison Table

Single thread	0ms
Multithread (Thread object) with Join	2ms
Multithread (Thread object) without Join	1ms
Multithread (ThreadPool) without Join	0ms (slightly off)

Multithreaded applications would likely have defeated single-threaded application had the problem been quite large. In this case, the problem was pretty small and the overhead time of thread creation and task assignment slowed things down.

As for ThreadPool application, there's no overhead for thread creation and interruption but there's still a statement in which we have to assign the function to the ThreadPool thread. That costs some time too, but less than the Thread objects application.

Again, if the problem had been larger, we would have seen multithreaded applications win. Specially the ones without the Join operation.

Conclusion

It can be confirmed that, at least in this case, the single threaded application was faster than the multithreaded application using Thread object (both with and without Join operation).

But, due to ThreadPool implementation's time measurements being slightly off, it cannot be said with certainty whether it performed better or worse than the single threaded application.