Laboratory Exercise 02 Part 02

Runtime-efficient Threaded Interpolating elevation

Kurt Brian Daine B. Punzalan

March 30, 2023

Table of Contents

Introduction

As a programmer, our goal would always be to create a program that runs efficiently. We can achieve this by using appropriate data structures for our program [3], choosing the best algorithms [1], and optimizing loops like what was done in the previous exercise. We can also achieve better running time for our program by deviating from the norms of serial programming and start designing parallel algorithm.

A parallel algorithm is an algorithm designed to solve a particular problem by breaking it down into sub-problems while independently computing these smaller problems. Parallel programs typically run faster than serial algorithm because parts of the program are being concurrently computed, unlike in the former where computations are done sequentially [2].

Parallel program works by assigning tasks – that are often independent from one another – to different processors or nodes. Parallel programming can be used in climate research, advanced computer graphics, and quantum mechanics, etc. [4]

In this exercise, we were tasked to implement parallelism and run our program on interpolating elevations given various number of processors t and compare the running time of the program given these processors.

Objectives

The objectives of this exercise are as follows:

1. To partition matrix M into t sub-matrices;
2. To independently interpolate the sub-matrices and regroup them in a single matrix;
3. To examine and evaluate the running time of the program given an input and various processors; and
4. Identify factors that might affect the running time and complexity of the program.

Methodology

In this exercise, we were tasked to reuse our program on interpolating elevations, partition it into t submatrices which will also serve as the number of processors, and independently interpolate these submatrices. Python programming language was used in the exercise and Federal Communications Commission (FCC) as the method of interpolation. The processing module was used in order to interpolate in parallel.

In the program, an n by n matrix was initialized where n is a non-zero integer divisible by 10. All rows and columns that are both divisible by 10 were filled with randomized values from 1 to 1000 which served as the basis for the FCC.

When the program is run, it will prompt the user to provide values for n and t. Assuming that the user correctly inputted the data, the program will then proceed to interpolating per row and partitioning them into submatrices. The program will then proceed to calculating the remaining rows.

The running time of the program from the initialized until the filled matrix was calculated. Three runs were made for each input. This was repeatedly done for n = 8000 with t ranging from 1 to 64. The number of processors were plotted against their respective average running time of the program.

The program was ran in the laboratory PC with Ubuntu 22.04.2 as its operating system having an Intel Core i7-8700 processor with 16.0GB of RAM and clock rate of 3.2GHz.

Results and Discussions

The program has a loop within a loop, which indicates that it runs in a quadratic time complexity or $O(n^2)$. After running the program, the number of processors were tabluated with their corresponding average runtime.

Table 1: Average Runtime of different processors (t) given input size (n) 8000

n	t	Time Elapsed (seconds)			Average Runtime (seconds)
		Run 1	Run 2	Run 3	
8000	1	36.02228975	35.00237465	34.58222628	35.20229689
8000	2	23.34000087	22.78530884	22.96395826	23.02975599
8000	4	16.97979498	17.87165809	16.79051924	17.21399077
8000	8	14.43450546	14.39202523	14.26860452	14.36504507
8000	16	14.6655376	15.11849761	15.15313673	14.97905731
8000	32	14.25943065	14.13332129	14.91182899	14.43486031
8000	64	14.28606868	14.22788358	14.1799829	14.23131172

Table 2: Average Runtime of the program without parallelism on input n equals 8000

n	t	Time Elapsed (seconds)			Average Runtime (seconds)
		Run 1	Run 2	Run 3	
8000	1	25.0845	25.085	25.1549	25.1082

Table 1 shows the average time of different processors given the same input size 8000, while Table 2 describes the average run time of the program without parallelism (single processor).

Since Python was used in this exercise, it was expected to execute more slowly than in C. Although, comparing the running time from Table 1 and Table 2, we can see significant changes as the number of processors is increased.

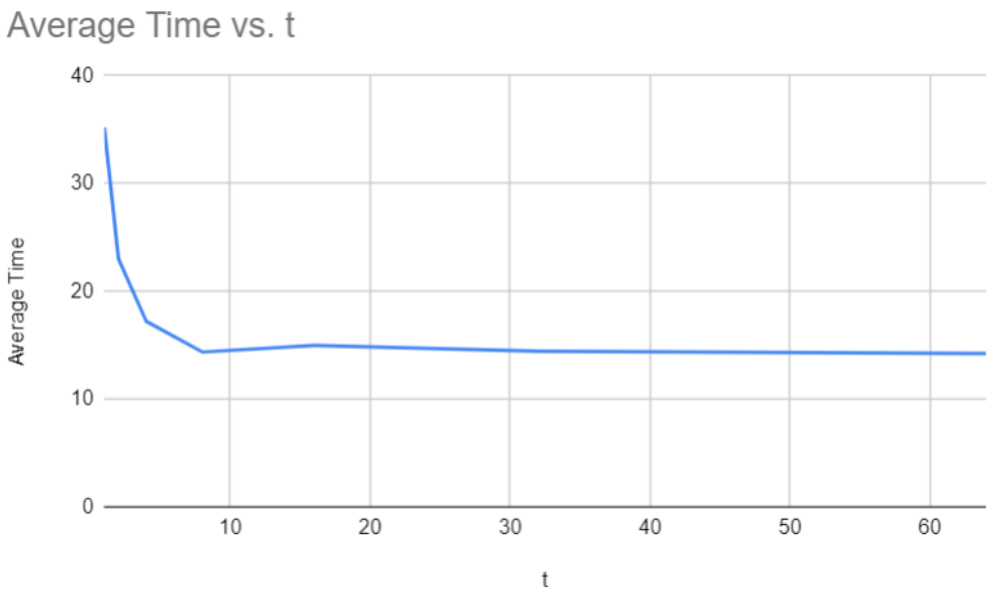

Figure 1: Average Running Time of Various Processors Given n = 8000

Figure 1 shows represents the graph of the Table 1 where the number of processors is plotted against their respective average run time in seconds for input n = 8000. In this graph, we can see that there is a significant change in the running time of the program when the number of processors. It can also be seen that the the running time was almost the same starting from t = 8 up to t = 64.

The complexity of estimating the point elevation of a n x n square matrix with given/randomized values at grid points divisible by 10 when using n concurrent processors is O(n2). This also holds true for processors n/2, n/4, or n/8 because the algorithm was not altered and that a loop within a loop was still observed just like in the previous exercise.

Comparing the average running time of the program with threads being implemented (Table 1) with the sequential program (Table 2), we can

say that the latter is faster. It ran for 25.1082 seconds while the former ran for 32.20229689 seconds. This is due to an additional overhead in creating the threads. The program with threads must also take note of resource allocation, starting and joining the threads, and terminating them, which makes it run slower.

Although, the program can go as far as t = n, even t = n/2, t= n/4, or t = n/8. This is possible since we can still partition our matrix in this way. It will not work, however, if t is greater than n since there will be threads that are not assigned to a row in the program.

Lastly, it can be noted that if the program that runs sequentially is capable of executing n=16000 and n=20000, then it would also run for the threaded exercise. However, I was not able to do it because the PC that I used became unresponsive during the running process. It is assumed that the running time of the threaded program will be faster than in the sequential one just like when n=8000 due to the aforementioned reasons. Having Python as the programming language used in this exercise, I assume that it will not run (or, if it would, then it would run slowly) for n=5000 and higher because of hardware limitations.

Conclusion

Creating a program that runs in parallel is beneficial when the things to be calculated are independent from one another. In the long run, it makes our program efficient and run faster. It was observed that the running time of the threaded program is slower than in the sequential one due to some overhead. Although, the running time rapidly decreased as the number of processors was increased and it stayed the same after a few processors. The system hardware also plays an important role since in the program, it was not able to run higher inputs. Lastly, it is important that we understand how parallel program works since more often than not, it will provide faster performance and enhance our programming skills as well on a new paradigm.

List of Collaborators

These are the individuals who have collaborated on the program:

1. **Van Paul Dayag**: importing and using the multiprocessing library

References

[1] Pythonista Planet. Do you need to know algorithms? *Pythonista Planet*, March 25 2023.

[2] Tutorials Point. Parallel algorithm - introduction, 2022.

[3] Masai School. 5 reasons why you should learn data structures and algorithms. https://blog.masaischool.com/5-reasons-why-you-should-learn-data-structures-and-algorithms/, September 20 2022. Accessed on March 25, 2023.

[4] TotalView Technologies. What is parallel programming?, 2021.

Appendices

Appendix A: Source Code

```
# Punzalan, Kurt Brian Daine B. Punzalan
# 2020-00772
# CMSC 180 - T-6L
# Exercise 02

import os
import random
import time
import multiprocessing as mp

os.system("clear")

#! FUNCTIONS
def printMatrix(M, n):
    for i in range(n):
        for j in range(n):
            print(M[i][j], end="\t")

        print()

def randomizeDivisible(M, n):
    # randomize values of all divisible by 10
    for row in range(n):
        for col in range(n):
            if (row % 10 == 0) and (col % 10 == 0):
                M[row][col] = float(random.randint(1,1000))

    return M

def terrain_inter_row(M, n, row):
    for index in range(n):
        if M[row][index] == 0:
```

```
            x = index # provide x based on the x-coordinate

            # apply the formula for FCC
            M[row][index] = round((y1 + ((x-x1)/(x2-x1)) * (y2-y1)), 2)
        else:
            # print(index, "index")
            x1 = index
            x2 = index + 10

            try:
                y1 = M[row][x1]
                y2 = M[row][x2]
            except:
                pass

def terrain_inter_col(M, n, col):
    for index in range(n):
        if M[index][col] == 0:
            x = index
            M[index][col] = round((y1 + ((x-x1)/(x2-x1)) * (y2-y1)), 2)

        else:
            x1 = index
            x2 = index + 10

            try:
                y1 = M[x1][col]
                y2 = M[x2][col]
            except:
                pass

    return M

def elevate(M, n, index, index2, i, q):
```

```
67      for row in range(index, index2):
68          terrain_inter_row(M, n, row)
69      # printMatrix(M, n)
70      # print()
71
72      q[i] = M[index:index2]
73
74      # for i in range(index, index2):
75      #     for j in range(n):
76      #         print(M[i][j], end="\t")
77
78      #     print()
79      # print()
80
81
82
83  #! MAIN
84  if __name__ == "__main__":
85      # user input
86      n = int(input("Input: ")) + 1
87      t = int(input("Number of submatrices: "))
88
89
90      # matrix
91      M = [[0 for column in range(n)]for row in range(n)]
92      M = randomizeDivisible(M, n)
93
94      #! FCC
95      # calculate
96      time_before = time.time()
97
98
99      # fill the columns whose rows are divisible by 10
100     col = 0
101     while col < n:
102         M = terrain_inter_col(M, n, col)
```

```
        col += 10

    # printMatrix(M, n)
    # print()

    #! get submatrices (per row)

    num_per_group = n // t
    remainder = n % t
    elements = [num_per_group] * t

    # Distribute the remainder evenly
    for i in range(remainder):
        elements[i] += 1


    # print(elements)

    # put into a list the starting indexes of the submatrices
    start_index = 0
    start_list = [0] # starting will always be 0th index
    for item in range(len(elements) - 1):
        start_list.append(start_index + elements[item])
        start_index += elements[item]

    threads = []

    q = mp.Manager().dict()
    for i in range(t):
        index = start_list[i]
        if i == t-1: # last row
            index2 = n
        else:
            index2 = start_list[i+1]

        my_thread = mp.Process(target=elevate, args=(M, n,
```

```
    index, index2, i, q, ))
139         my_thread.start()
140         threads.append(my_thread)
141 
142     for p in threads:
143         p.join()
144 
145     # printMatrix(M, n)
146 
147     list = []
148     for i in range(t):
149         for row in q[i]:
150             list.append(row)
151     M = list
152 
153     # for row in list:
154     #     print(row)
155     # print(list)
156     # printMatrix(list, n)
157 
158     # for i in range(n):
159     #     for j in range(n):
160     #         print(f"({i}, {j})", end="\t")
161     #     print()
162 
163     # print()
164 
165     # printMatrix(list, n)
166 
167     time_elapsed = time.time() - time_before
168 
169     print("time elapsed:", time_elapsed)
```

Appendix B: PC Specifications