# Laboratory Exercise 03 Lab Report

Runtime-efficient Threaded Interpolating elevation and
Core-affine Threaded Computation of Interpolating Elevation

Kurt Brian Daine B. Punzalan

April 30, 2023

# Table of Contents


Introduction . . . . . 2
Objectives . . . . . 3
Methodology . . . . . 4
Results and Discussions . . . . . 6
Conclusion . . . . . 10
List of Collaborators . . . . . 11
References . . . . . 12
Appendices . . . . . 13
    Appendix A: Source Code . . . . . 13
    Appendix B: PC Specifications . . . . . 18

# Introduction

Both multiprocessing and multithreading are techniques used in Python to execute tasks concurrently. However, it is important to take note that the two are different concepts. According to Brownlee, "multiprocessing" module facilitates processed-based concurrency, while the module "threading" enables concurrency based on threads. Furthermore, in multiprocessing, each process runs independently and and has its own memory space. Multithreading, however, involves using multiple threads within a single process to execute concurrently, wherein the CPU juggles from one task to another with this processor. Unlike in multiprocessing, threads share the same memory space and can access the same resources. [2]

In this exercise, we were tasked to implement parallelism and compare multithreading and multiprocessing as well, 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. Create threads to independently interpolate the sub-matrices and re-group them in a single matrix;
3. Apply multi-threading in the program; and
4. Compare multi-threading and multiprocessing in Python.

# Methodology

In this exercise, we were tasked to restructure our program on interpolating elevations, partition it into t submatrices and independently interpolate these submatrices using core-affine threaded computations. However, it is important to note that this is a report on multi-threading, not in core-affinity, for the reason that core affinity was already accomplished in the previous exercise (Exercise 2). Thus, the concept of multithreading was implemented instead this exercise.

Python programming language was used in the exercise with Federal Communications Commission (FCC) as the method of interpolation. The Threading module was used in order to interpolate.

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 result of the multi-threaded program was then compared to the results of the program that is core-affined (utilizing multiprocessing library).

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 shows the average time of different processors given the same input size 8000 using the multithreading library. We can see only minimal changes in the average running time as t increases.

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

| n | t | Time Elapsed (seconds) | | | Average Runtime (seconds) |
|---|---|---|---|---|---|
| | | Run 1 | Run 2 | Run 3 | |
| 8000 | 1 | 25.12501907 | 25.79419994 | 25.4202745 | 25.44649784 |
| 8000 | 2 | 26.25427413 | 26.56773281 | 25.7060504 | 26.17601911 |
| 8000 | 4 | 26.27286744 | 26.12017703 | 26.86624217 | 26.41976221 |
| 8000 | 8 | 26.2011404 | 26.39999676 | 27.78777719 | 26.79630478 |
| 8000 | 16 | 27.94791985 | 29.95100427 | 27.97935843 | 28.62609418 |
| 8000 | 32 | 26.80592394 | 26.95694685 | 27.25011873 | 27.00432984 |
| 8000 | 64 | 27.04390645 | 28.09926963 | 28.09926963 | 27.7474819 |

Table 2 shows the average time of different processors given the same input size 8000 using the multiprocessing library. It was observed in this table that the average running time became lower as the number of processors is increased.

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 results from the program that uses multiprocessing rather than multiprocessing.

Table 2: Average Runtime of different processors (t) given input size (n) 8000 Using Multirpocessing

| 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 |

Likewise, the average running time of the multi-threaded program as compared to the original program with no threads (from the Exercise 1) was almost the same for t = 1 to t = 64.

Table 3: 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 |

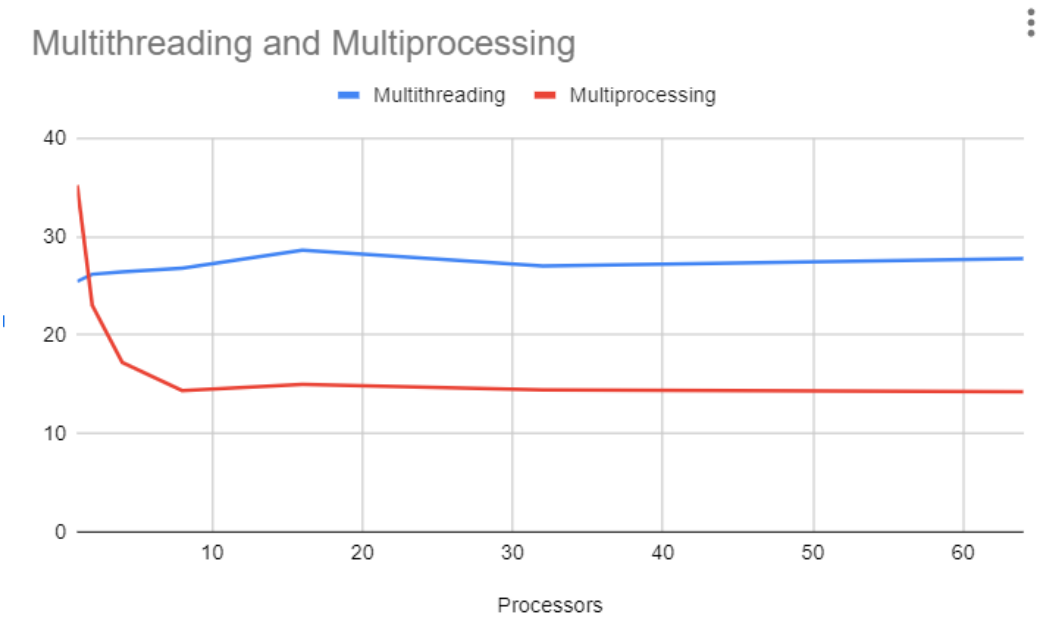


Figure 1: Average Running Time of Various Processors for Multiprocessing and Multithreading Given n = 8000

Figure 1 represents the graph of both Tables 1 and 2 where the number of processors is plotted against their respective average running time in seconds for input n = 8000. In this graph, we can see a downward trend for the multiprocessing. It can also be seen that the the running time was almost the same starting from t = 8 up to t = 64 in multithreading. However, a stagnant trend was observed in the program for multiprocessing for t = 1 up to t = 64.

The stagnant data from the multithreaded program might be caused by the Python Global Interpreter Lock (GIL) being the bottleneck preventing threads from running concurrently. Even if you have multiple threads running at the same time in Python, only one thread can execute Python code at any given time, regardless of how many cores your processor has. In other words, multiple threads cannot execute Python code simultaneously, which limits the performance gains you can achieve through parallelism. Increasing

the number of threads might also pave way for slower running times due to thread management overhead, as well as increased contention for shared resources.

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.

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 is important to note that if a program can handle executing n=16000 and n=20000 sequentially, it should be able to run the threaded exercise as well. However, I was unable to do so because the computer I used became unresponsive during the process. Based on the reasons previously discussed, it is assumed that the threaded program would run the same as the sequential one when n=8000. Likewise, applying multiprocessing instead would provide significant changes in the average running time of the program as compared with both sequential and multithreaded program. Considering that Python was used in this exercise, it is expected that the program may not run (or run slowly) for n=5000 and higher due to hardware limitations.

# Conclusion

A program that runs in parallel is beneficial when the partitions created to be calculated are independent from one another. In the long run, it makes our program efficient and run faster. However, it is important to take note of both multithreading and multiprocessing, which are two different concepts in Python programming language. In the provided data, multiprocessing yielded better results than the program that uses multithreading. [1]

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 no collaborators for this exercise.

# References

[1] DevGenius. Why is multi-threaded python so slow?, 2020.

[2] Super Fast Python. Threading vs multiprocessing in python, 2021.

# Appendices

## Appendix A: Source Code

```
# Punzalan, Kurt Brian Daine B. Punzalan
# 2020-00772
# CMSC 180 - T-6L
# Exercise 03

import os
import random
import time
import threading

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:
```

```
33              x = index # provide x based on the x-coordinate
34
35              # apply the formula for FCC
36              M[row][index] = round((y1 + ((x-x1)/(x2-x1)) * (
    y2-y1)), 2)
37          else:
38              # print(index, "index")
39              x1 = index
40              x2 = index + 10
41
42              try:
43                  y1 = M[row][x1]
44                  y2 = M[row][x2]
45              except:
46                  pass
47
48  def terrain_inter_col(M, n, col):
49      for index in range(n):
50          if M[index][col] == 0:
51              x = index
52              M[index][col] = round((y1 + ((x-x1)/(x2-x1)) * (
    y2-y1)), 2)
53
54          else:
55              x1 = index
56              x2 = index + 10
57
58              try:
59                  y1 = M[x1][col]
60                  y2 = M[x2][col]
61              except:
62                  pass
63
64      return M
65
66  def elevate(M, n, index, index2):
```

```
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 = []

    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 = threading.Thread(target=elevate, args=(M,
    n, index, index2, ))
```

```
138         threads.append(my_thread)
139 
140     for p in threads:
141         p.start()
142 
143     for p in threads:
144         p.join()
145 
146 
147     time_elapsed = time.time() - time_before
148 
149 
150     print("time elapsed:", time_elapsed)
```

## Appendix B: PC Specifications