Laboratory Exercise 04 Lab Report

Distributing Parts of a Matrix over Sockets

Kurt Brian Daine B. Punzalan

May 27, 2023

Table of Contents

Introduction

Performance is a crucial aspect in various fields of computer science, particularly if we want to implement efficient algorithms and software systems. So far, what was introduced in the laboratory were multithreading and multiprocessing, both of which provided us with insights on how to improve the running time of an algorithm. Another viable approach would be distributed computing.

Distributed computing refers to a model where different computers or nodes collectively share software components of a system. Despite these components being distributed across various computers and locations, they function as a unified system. The purpose of this approach is to enhance efficiency and performance [3].

To facilitate distributed computing, socket programming is commonly used. In socket programming, a socket refers to and acts as an endpoint in a distributed communication system, utilizing the client-server model to address various network applications. For distributed communication to occur, two sockets are necessary, with one acting as the endpoint in the client and the other in the server [2].

In this exercise, we were tasked to implement socket programming in our Interpolating Elevation problems and distribute partitions of these matrix over sockets.

Objectives

The objectives of this exercise are as follows:

1. To partition matrix M into t sub-matrices;
2. Create a master (server) and distribute these partitions into different slaves (client);
3. Elevate the partitions on each slave;
4. Apply socket programming in a single computer but different terminals; and
5. Apply socket programming in different computers.

Methodology

In this exercise, we were tasked to restructure our program on interpolating elevations, partition it into t submatrices and independently interpolate these submatrices by having the master (server) send these partitions to the slave (client) and each slave interpolate. Note that in this context, the terms master and slaves are used interchangeably with server and clients, respectively.

Python programming language was used in the exercise with Federal Communications Commission (FCC) as the method of interpolation. The socket module was used in this exercise to facilitate socket programming.

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. The port number as well as the host were used as inputs as well.

The user will be prompted whether they would use the master (0) or the slave (1). If the user chose the master, a server will be started for the clients to connect to. The master will then create the partition of the matrix depending on the number of clients and each partition will be sent to the clients. The clients will then evaluate and interpolate the submatrix.

This was run in all instances within one PC only but on different terminals and all slave instances on different PCs as well.

The running time of the algorithm on varying matrix size and number of slaves for both single PC and different PCs were tabulated.

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

Table 1 shows the average running time of the program on different terminals with varying sizes and number of slaves.

Table 1: Average Running Time of the Program on Same PC but Different Terminals

n	t	Time Elapsed (seconds)			Average Runtime (seconds)
		Run 1	Run 2	Run 3	
1000	2	3.887878656	3.462329865	2.825969934	3.392059485
1000	4	5.666114092	4.412817478	4.252913475	4.777281682
1000	8	11.01140761	10.91570258	7.030566931	9.652559042
1000	16	13.03499246	13.58272719	13.03594518	13.21788828
4000	2	6.978392363	7.025309086	7.070826292	7.02484258
4000	4	9.14272809	9.567723036	10.6107626	9.773737907
4000	8	12.1652863	12.63525724	11.93952847	12.24669067
4000	16	20.75235605	17.38881755	16.69650197	18.27922519
8000	2	26.75223708	24.06841135	23.74662066	24.85575636
8000	4	24.44364452	24.08331037	24.28868246	24.27187912
8000	8	25.29796624	25.11438894	24.67533374	25.02922964
8000	16	29.51941824	29.60834694	29.56071877	29.56282798

Based on the running times above, we can say that as the number of terminals increases, the average time increases as well. For n = 1000 and 4000, we can see that the average time increases for several seconds as the number of slaves are increased. In n = 8000, however, there are little to no changes for t = 2, 4, and 8, while there are a 4-second gap between t = 8 and t = 16.

Table 2 shows the average time of the program on different PCs for varying input sizes and slaves.

From the table above, we can deduce that as the number of PCs increases,

Table 2: Average Running Time of the Program on Different PCs

n	t	Time Elapsed (seconds)			Average Runtime (seconds)
		Run 1	Run 2	Run 3	
1000	2	1.254256487	1.347983837	1.714601994	1.438947439
1000	4	4.037964106	2.703391075	3.591584921	3.444313367
4000	2	10.20779538	9.613873243	9.618762255	9.81347696
4000	4	9.715957642	9.68091011	9.67411375	9.690327168
8000	2	38.14207792	38.19124532	38.89095664	38.40809329
8000	4	38.4219203	38.5423491	38.43476439	38.46634459

the running time also increases for n = 1000. However, for n = 4000 and 8000, there are no differences in the average running time for t = 2 and t = 4 for both sizes.

Since Python was used in this exercise, it was expected to execute more slowly than in C.

Conclusion

Distributed computing and parallel computing are frequently used together. Parallel computing involves utilizing multiple processors on a single computer to execute tasks simultaneously, while distributed parallel computing utilizes multiple computing devices to process those tasks [1]. In this exercise, I have learned sufficient knowledge on distributed computing and how it can be used to effectively solve the problem at hand.

Furthermore, we can say that the program is not efficient since as the number of t increases, the running time also increases. However, we can say that the running time for the multiple PCs is significantly slower than in single PC with different terminals. Lastly, we can say that one-to-many broadcast was used because the master specifies different messages or data for each recipient. This happens when n is not divisible by t and it distributes excess rows to the client.

List of Collaborators

The list of collaborators for this exercise are as follows:

1. Van Paul Dayag
2. Earl Samuel Capuchino

References

[1] Khan Academy. Distributed Computing, Year of publication (if available). Accessed: May 27, 2023.

[2] Baeldung. Socket vs RPC: A Comparison, Year of publication (if available). Accessed: May 27, 2023.

[3] TechTarget. Distributed Computing. https://www.techtarget.com/whatis/definition/distributed-computing. Accessed: May 27, 2023.

Appendices

Appendix A: Source Code

Main File

```
import master
import slave
import os
os.system('cls')

s = int(input("Enter s: "))
if __name__ == "__main__":
    with open("config.txt") as f:
        n = int(f.readline()) + 1
        t_minus_1 = int(f.readline())
        port = int(f.readline())
        host = f.readline()

    if s == 0:
        print("Sever")
        master.main(n, t_minus_1, host, port)
    else:
        print("Client")
        slave.main(n, host, port)
```

slave

```
# Import socket module
import socket
import pickle
import os

#! FUNCTIONS
def printMatrix(M, n):
    for i in range(n):
```

```
        for j in range(n):
            print(M[i][j], end="\t")

        print()

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 main(n, host, port):
    # os.system('clear')

    # Create a socket object
    client_socket = socket.socket()

    # Define the port on which you want to connect

    # connect to the server on local computer
    client_socket.connect((host, port))

```

```
44
45      data_size = int.from_bytes(client_socket.recv(4), 'big')
46
47      data = bytearray()
48      while len(data) < data_size:
49          packet = client_socket.recv(4096)
50          data.extend(packet)
51
52      M = pickle.loads(data)
53
54      message = "ack"
55      client_socket.send(message.encode())
56      print(message)
57
58      # fill remaining rows (inner box)
59      for row in range(len(M)):
60          terrain_inter_row(M, n, row)
61
62
63      print("Done Interpolation")
64
65      # print()
66      # for i in M:
67      #     print(i)
68      # print()
69
70
71      # close the connection
72      client_socket.close()
```

master

```
1 import socket
2 import pickle
3 import os
4 import random
```

```
import time

os.system('clear')

#! FUNCTIONS
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 createMatrix(n):
  M = [[0 for column in range(n)]for row in range(n)]
  randomizeDivisible(M, n)

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

        print()

def main(n, client_num, host, port):

    server_socket = socket.socket()
    print ("Socket successfully created")

    server_socket.bind((host, port))
    print("socket binded to %s" %(port))
```

```
75
76      # put the socket into listening mode
77      print ("socket is listening")
78      server_socket.listen(5)
79
80
81      M = createMatrix(n)
82      # printMatrix(M, n)
83      col = 0
84      while col < n:
85          M = terrain_inter_col(M, n, col)
86          col += 10
87
88      num_per_group = n // client_num
89      remainder = n % client_num
90      elements = [num_per_group] * client_num
91
92      # Distribute the remainder evenly
93      for i in range(remainder):
94          elements[i] += 1
95
96      print(elements)
97
98      # put into a list the starting indexes of the submatrices
99      start_index = 0
100     start_list = [0] # starting will always be 0th index
101     for item in range(len(elements)):
102         start_list.append(start_index + elements[item])
103         start_index += elements[item]
104
105     print(start_list)
106
107     for index in range(len(start_list)-1):
108         # print(start_list[index], start_list[index+1])
109         temp = M[start_list[index]:start_list[index+1]]
110
```

```
111
112         # for i in temp:
113         #     print(i)
114         # print()
115
116
117     start_time = time.time()
118
119
120     counter = 0
121     while counter < client_num:
122         client_socket, addr = server_socket.accept()
123         print('Got connection from', addr)
124
125         temp = M[start_list[counter]:start_list[counter+1]]
126         data=pickle.dumps(temp)
127         client_socket.sendall(len(data).to_bytes(4, 'big'))
128         client_socket.sendall(data)
129
130         ack = client_socket.recv(4096)
131         print(ack.decode())
132
133
134         data_size = int.from_bytes(client_socket.recv(4),'big
    ')
135         data = bytearray()
136
137
138
139
140         client_socket.close()
141         counter += 1
142
143     end_time = time.time()  # Record the end time
144     elapsed_time = end_time - start_time  # Calculate the
    elapsed time
```

```
145
146     print("Elapsed time:", elapsed_time, "seconds")
147
148
149     print("You have reached the maximum number of clients.")
```

Appendix B: PC Specifications