**BCSE205L – COMPUTER ARCHITECTURE**

**DIGITAL ASSIGNMENT II**

**TITLE – N x N Matrix Multiplication**

**Team No- 1**

| **Member Name1:** | **Revanth Srinivasa Reddy Bommareddy** |
| --- | --- |
| **Member Regno1:** | **22BAI1168** |
| **Contribution:** | **Implemented major parts of the program** |
| **Member Name2:** | **Nannapuraju Hemanth Raju** |
| **Member Regno2:** | **22BAI1257** |
| **Contribution:** | **Researched about the implementation of multiprocessing and applied in the code** |
| **Member Name3:** | **Batta Venkata Rahul** |
| **Member Regno3:** | **22BAI1245** |
| **Contribution:** | **Tested and analyzed the code with several examples and modified the Program whenever there is an error** |

**GITHUB LINK:**

**SUMMARY OF THE PROJECT**

This Python program showcases the power of parallel processing for matrix multiplication. It starts by importing essential libraries, including NumPy for matrix operations, time for measuring execution time, multiprocessing for parallelization, and random for generating random matrices.

The program defines two crucial functions:

‘generate\_random\_matrix’ generates random matrices with user-defined dimensions, and “compare\_to\_our\_approach” compares the execution time and results of parallel matrix multiplication (our approach) with a naive matrix multiplication approach.

A worker function, “worker”, is introduced to perform matrix multiplication on a portion of the matrix. This function is crucial for parallel processing, which is a key aspect of this program.

The "input\_matrix" function is responsible for taking user input to create matrices. Users specify the number of rows and columns, and the program populates the matrix with user-defined elements.

The main part of the program lies in two matrix multiplication functions: "parallel\_mul” and "naive\_matrix\_mult". The former performs matrix multiplication using parallel processing, splitting the first matrix into smaller parts, processing them concurrently with multiple processes, and then combining the results to produce the final outcome. The latter, "naive\_matrix\_mult", computes matrix multiplication using a traditional, iterative approach.

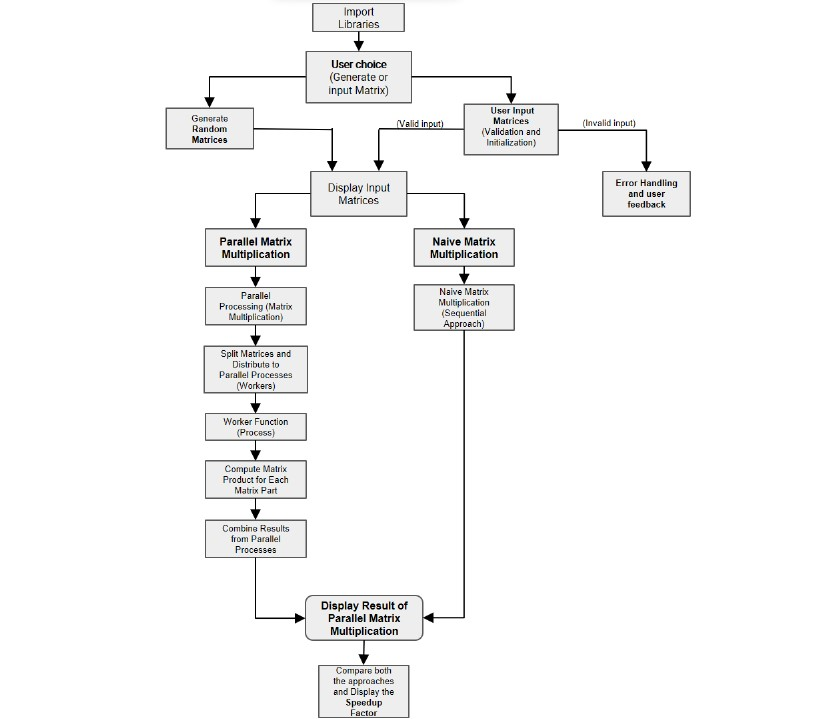
Users are given the choice to either generate random matrices or input their own matrices. Based on this choice, the program either generates random matrices or takes user input for two matrices.

After collecting the matrix inputs, the program prints the input matrices and the final result obtained by multiplying them using the "parallel\_mul" function.

Finally, the program calculates the execution time and speedup factor, comparing the results to the naive matrix multiplication approach through the "compare\_to\_our\_approach" function. This speedup factor quantifies the efficiency gains achieved by the parallel approach, highlighting its advantage over the traditional approach.

In summary, this program serves as a practical demonstration of the benefits of parallel processing in accelerating matrix multiplication. It offers flexibility for users to input their own matrices or utilize randomly generated ones, making it a valuable tool for assessing the efficiency of parallel processing in matrix operations.

**ARCHITECTURE DIAGRAM**

****

**CODE**

import numpy

import time

from multiprocessing import Process, Queue, cpu\_count

import random

def generate\_random\_matrix(rows, cols):

"""Generate a matrix with random numbers"""

return numpy.random.randint(100, size=(rows, cols))

def compare\_to\_our\_approach(matrix1, matrix2):

"""Compare the result and execution time to multiprocessing approach and naive matrix multiplication"""

# Start the timer for our approach

start\_time\_our = time.perf\_counter()

our\_result = parallel\_mul(matrix1, matrix2)

end\_time\_our = time.perf\_counter()

# Start the timer for naive matrix multiplication

start\_time\_naive = time.perf\_counter()

naive\_result = naive\_matrix\_mult(matrix1, matrix2)

end\_time\_naive = time.perf\_counter()

# Calculate and print the time taken by each process

time\_our = end\_time\_our - start\_time\_our

time\_naive = end\_time\_naive - start\_time\_naive

print(f"\nTime taken by our approach: {time\_our} seconds")

print(f"Time taken by naive matrix multiplication: {time\_naive} seconds")

# Calculate the speedup factor

speedup = time\_naive / time\_our

print(f"\nOur approach is {speedup:.2f} times faster than naive matrix multiplication.\n")

# worker function

def worker(matrix1Part, matrix2, result\_queue):

result\_queue.put(numpy.matmul(matrix1Part, matrix2))

def input\_matrix():

"""Take a matrix as input from the user."""

# Get the number of rows and columns

while True:

try:

rows = int(input("Enter the number of rows: "))

cols = int(input("Enter the number of columns: "))

if rows <= 0 or cols <= 0:

print("Number of rows and columns should be greater than 0. Please try again.")

continue

break

except ValueError:

print("Invalid input! Please enter a positive integer.")

# Initialize an empty matrix

matrix = numpy.zeros((rows, cols))

# Get the matrix elements

for i in range(rows):

for j in range(cols):

while True:

try:

matrix[i, j] = float(input(f"Enter element ({i+1}, {j+1}): "))

break

except ValueError:

print("Invalid input, Please enter a number.")

return matrix

#parallel muliplication using parallel processing

def parallel\_mul(matrix1,matrix2):

# Splitting the first matrix

num\_processes = cpu\_count()

split\_matrix\_1 = numpy.array\_split(matrix1,num\_processes)

# Queue to store the results

resultQueue = Queue()

# Start the processes

processes = []

for i in split\_matrix\_1:

p = Process(target=worker,args=(i,matrix2,resultQueue))

p.start()

processes.append(p)

# Wait for all processes to finish

for j in processes:

j.join()

# Combine the results

result\_parts = []

while not resultQueue.empty():

result\_parts.append(resultQueue.get())

finalResult = numpy.concatenate(result\_parts)

return finalResult

#naive matrix multiplication approach

def naive\_matrix\_mult(matrix1, matrix2):

"""Perform naive matrix multiplication"""

n1, m1 = matrix1.shape

n2, m2 = matrix2.shape

assert m1 == n2, "Matrix dimensions do not match for multiplication"

result = numpy.zeros((n1, m2))

for i in range(n1):

for j in range(m2):

for k in range(m1):

result[i][j] += matrix1[i][k] \* matrix2[k][j]

return result

# Input two matrices n\*n matrices'

option = input("Enter 0 to generate random matrices or 1 to input matrices: ")

option = int(option) # Convert the input to an integer.

if option == 0:

n = int(input("Enter the size of the matrices: "))

matrix1 = generate\_random\_matrix(n, n)

matrix2 = generate\_random\_matrix(n, n)

elif option == 1:

matrix1 = input\_matrix()

matrix2 = input\_matrix()

else:

print("Enter a valid choice (0 or 1)!")

# Print the inputs and the product of the two matrices

print("\nInput matrix 1: \n",matrix1)

print("\nInput matrix 2: \n",matrix2)

print("\nfinal result after multiplying the above matrices:\n",parallel\_mul(matrix1,matrix2))

# Compare the result to naive matrix multiplication approach

compare\_to\_our\_approach(matrix1, matrix2)

**OUTPUT**

