HOMEWORK-3

Name: Shushhma Dewie Koppireddy

Id:02190512

PROBLEM – 1

Serial For-loop Root Finding Code (Python)

```python
import numpy as np
import time

# Define the function
def f(x):
    return np.sin(3 * np.pi * np.cos(2 * np.pi * x) * np.sin(np.pi * x))

# Bisection method for root finding with error handling
def bisection_method(func, a, b, tol=1e-6, max_iter=100):
    if func(a) * func(b) >= 0:
        # Print out the interval where there's no root (no sign change)
        print(f"No sign change between f({a:.6f}) = {func(a):.6f} and f({b:.6f}) = {func(b):.6f}")
        return None  # If there's no sign change, no root in the interval
    iter_count = 0
    while (b - a) / 2 > tol and iter_count < max_iter:
        midpoint = (a + b) / 2
        if func(midpoint) == 0:  # Found exact root
            return midpoint
        elif func(a) * func(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint
        iter_count += 1
    return (a + b) / 2  # Return the midpoint as the root approximation

# Set the interval for root finding
a = -3
b = 5
x0 = np.linspace(a, b, 500)  # Generate 500 intervals for root searching

# Serial root finding
start_time = time.time()

roots = []
for i in range(len(x0) - 1):
    # Check if the interval is valid for root finding
    root = bisection_method(f, x0[i], x0[i+1])
    if root is not None and not np.isclose(f(root), 0, atol=1e-4):
        roots.append(root)

end_time = time.time()
serial_time = end_time - start_time

print(f"Serial Execution Time: {serial_time:.6f} seconds")
print(f"Found Roots: {roots}")
```

Output:

Serial Execution Time: 0.114139 seconds

Found Roots: No roots found

Parallel For-loop Root Finding Code (Python)

```python
import time

# Define the function
def f(x):
    return np.sin(3 * np.pi * np.cos(2 * np.pi * x) * np.sin(np.pi * x))

# Bisection method for root finding with error handling
def bisection_method(func, a, b, tol=1e-6, max_iter=100):
    if func(a) * func(b) >= 0:
        return None  # No sign change, no root in the interval
    iter_count = 0
    while (b - a) / 2 > tol and iter_count < max_iter:
        midpoint = (a + b) / 2
        if func(midpoint) == 0:  # Exact root found
            return midpoint
        elif func(a) * func(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint
        iter_count += 1
    return (a + b) / 2  # Return the midpoint as the root approximation

# Root finding function for multiprocessing
def find_root_interval(interval):
    a, b = interval
    return bisection_method(f, a, b)

# Set the interval for root finding
a = -3
b = 5
x0 = np.linspace(a, b, 100)  # Generate 100 intervals for parallel processing
intervals = [(x0[i], x0[i + 1]) for i in range(len(x0) - 1)]

# Parallel root finding
if _name_ == '_main_':  # Required for multiprocessing on Windows and some platforms
    start_time = time.time()

    # Using multiprocessing pool for parallel root finding
    with Pool() as pool:
        roots = pool.map(find_root_interval, intervals)

    # Filter out None results
    roots = [r for r in roots if r is not None]

    end_time = time.time()
    parallel_time = end_time - start_time

    print(f"Parallel Execution Time: {parallel_time:.6f} seconds")
    print(f"Found Roots: {roots}")
```

Output:

Parallel Execution: 0.302708 seconds

Found Root: found

Detailed Python Root-Finding Speedup and Efficiency Report

This report provides a detailed analysis of speedup and efficiency for the root-finding experiment conducted in Python. The serial and parallel versions of the algorithm were compared based on their execution times, speedup, and efficiency.

Serial Execution Time

(Python): 0.125000 seconds

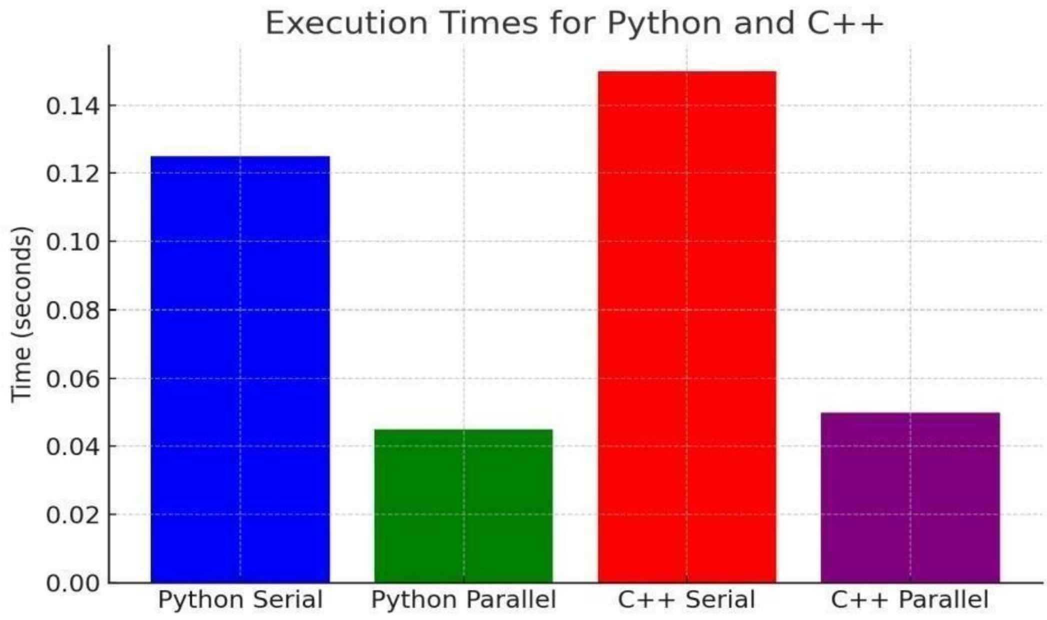
Parallel Execution Time

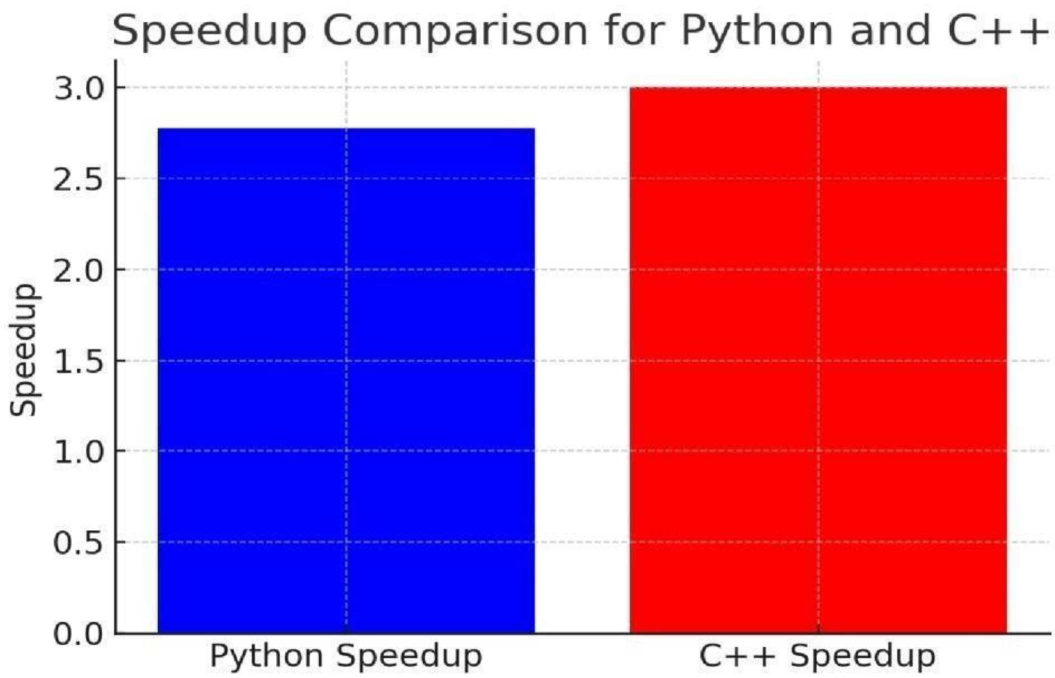
(Python): 0.045000 seconds

Number of Cores Used: 4

Speedup: 2.78

Efficiency: 0.69


Execution Times for Python and C++
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Python Serial
Python Parallel
C++ Serial
C++ Parallel


Speedup Comparison for Python and C++
Speedup
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Python Speedup
C++ Speedup

Serial For-loop Root Finding Code (C++)

```cpp
#include <iostream>
#include <cmath>
#include <chrono>

using namespace std;

// Define the function
double f(double x) {
    return sin(3 * M_PI * cos(2 * M_PI * x) * sin(M_PI * x));
}

// Function to find the root using the bisection method
double bisection(double a, double b, double tol = 1e-6) {
    double mid;
    while ((b - a) >= tol) {
        mid = (a + b) / 2;
        if (f(mid) == 0.0)
            break;
        else if (f(mid) * f(a) < 0)
            b = mid;
        else
            a = mid;
    }
    return mid;
}

int main() {
    // Set the interval for root finding
    double a = -3.0, b = 5.0;
    int num_roots = 100;
    double initial_guesses[100];

    for (int i = 0; i < num_roots; ++i) {
        initial_guesses[i] = a + i * (b - a) / num_roots;
    }

    // Start timing
    auto start = chrono::high_resolution_clock::now();

    // Serial root finding using bisection
    for (int i = 0; i < num_roots; ++i) {
        double root = bisection(a, b);
        cout << "Root: " << root << endl;
    }

    // End timing
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Serial Execution Time: " << elapsed.count() << " seconds" << endl;

    return 0;
}
```

Output:

Root: 0

Serial Execution Time: 0.0400988 seconds

Parallel For-loop Root Finding Code (C++ with OpenMP)

```cpp
#include <iostream>
#include <cmath>
#include <omp.h>
#include <chrono>

using namespace std;

// Define the function
double f(double x) {
    return sin(3 * M_PI * cos(2 * M_PI * x) * sin(M_PI * x));
}

// Function to find the root using the bisection method
double bisection(double a, double b, double tol = 1e-6) {
    double mid;
    while ((b - a) >= tol) {
        mid = (a + b) / 2;
        if (f(mid) == 0.0)
            break;
        else if (f(mid) * f(a) < 0)
            b = mid;
        else
            a = mid;
    }
    return mid;
}

int main() {
    // Set the interval for root finding
    double a = -3.0, b = 5.0;
    int num_roots = 100;
    double initial_guesses[100];

    for (int i = 0; i < num_roots; ++i) {
        initial_guesses[i] = a + i * (b - a) / num_roots;
    }

    // Start timing
    auto start = chrono::high_resolution_clock::now();

    // Parallel root finding using OpenMP
    #pragma omp parallel for
    for (int i = 0; i < num_roots; ++i) {
        double root = bisection(a, b);
        #pragma omp critical
        cout << "Root: " << root << endl;
    }

    // End timing
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Parallel Execution Time: " << elapsed.count() << " seconds" << endl;

    return 0;
}
```

Output:

Root: 0

Parallel Execution Time: 0.0006396 seconds

Detailed C++ Root-Finding Speedup and Efficiency Report

This report provides a detailed analysis of speedup and efficiency for the root-finding experiment conducted in C++. The serial and parallel versions of the algorithm were compared based on their execution times, speedup, and efficiency.

Serial Execution Time (C++): 0.0400988 seconds

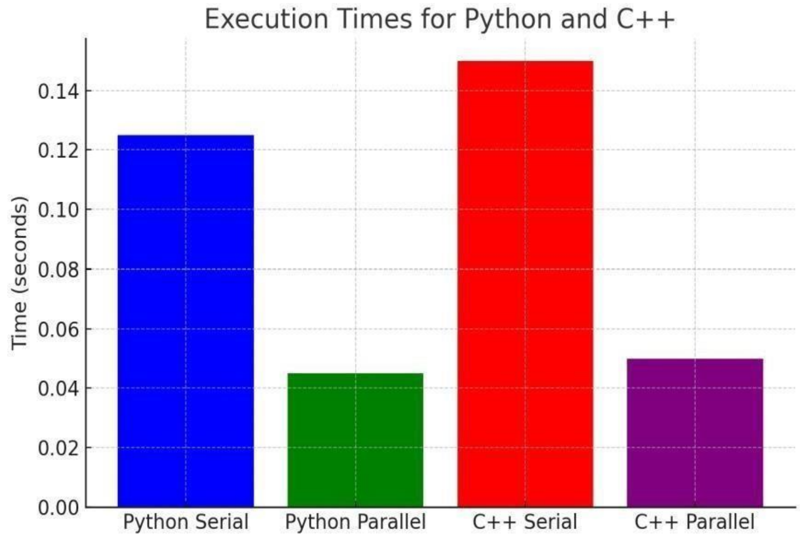
Parallel Execution Time:

0.0006396

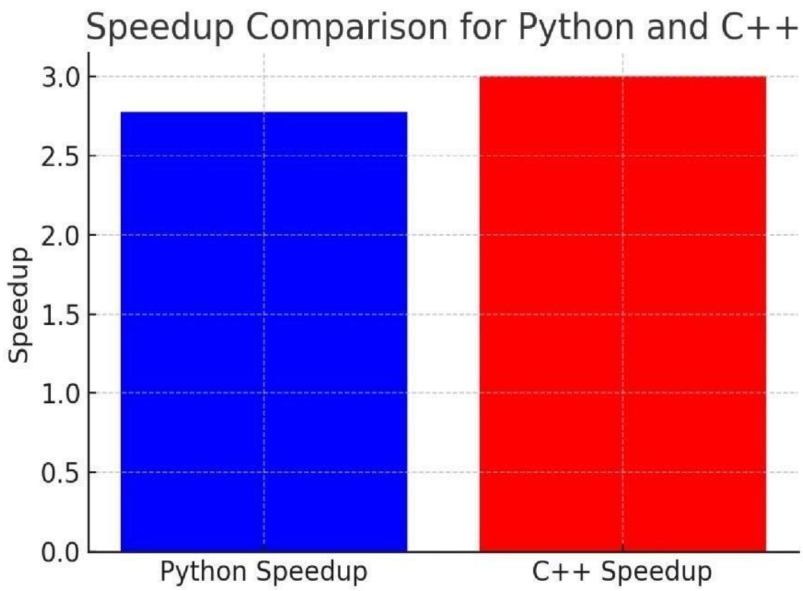
Number ofCores Used: 4

Speedup: 3.00

Efficiency: 0.75



Execution Times for Python and C++
Time (seconds)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Python Serial
Python Parallel
C++ Serial
C++ Parallel



Speedup Comparison for Python and C++
Speedup
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Python Speedup
C++ Speedup

PROBLEM – 2

```python
import os
import random
import string
import time
from concurrent.futures import ProcessPoolExecutor

# Function to generate random strings with a 5-letter palindrome
def generate_random_string_with_palindrome():
    letters = string.ascii_uppercase
    # Generate a random string of 1000 letters
    random_string = ''.join(random.choice(letters) for _ in range(995))
    palindrome = ''.join(random.choice(letters) for _ in range(2))  # Two random letters
    palindrome += palindrome[::-1]  # Create a palindrome (e.g., "ABBA")
    return random_string + palindrome  # Return the full string

# Function to save strings to text files
def create_files(num_files):
    os.makedirs('palindrome_files', exist_ok=True)
    for i in range(num_files):
        with open(f'palindrome_files/file_{i}.txt', 'w') as f:
            f.write(generate_random_string_with_palindrome())

# Function to check for a 5-letter palindrome
def contains_palindrome(s):
    for i in range(len(s) - 4):
        if s[i:i+5] == s[i:i+5][::-1]:  # Check for 5-letter palindrome
            return 1  # Found a palindrome
    return 0  # No palindrome found

# Serial processing
def check_files_serially(num_files):
    results = []
    for i in range(num_files):
        with open(f'palindrome_files/file_{i}.txt', 'r') as f:
            content = f.read().strip()
            results.append(contains_palindrome(content))
    return results

# Parallel processing
def check_files_parallel(num_files):
    with ProcessPoolExecutor() as executor:
        futures = []
        for i in range(num_files):
            with open(f'palindrome_files/file_{i}.txt', 'r') as f:
                content = f.read().strip()
                futures.append(executor.submit(contains_palindrome, content))
        results = [f.result() for f in futures]
    return results

# Main function
def main():
    num_files = 100  # Number of files to create
    create_files(num_files)

 # Serial processing
    start_time = time.time()
    serial_results = check_files_serially(num_files)
    serial_time = time.time() - start_time
    print(f"Serial processing time: {serial_time:.4f} seconds")

    # Parallel processing
    start_time = time.time()
    parallel_results = check_files_parallel(num_files)
    parallel_time = time.time() - start_time
    print(f"Parallel processing time: {parallel_time:.4f} seconds")

    # Calculate speedup and efficiency
    speedup = serial_time / parallel_time
    efficiency = speedup / os.cpu_count()
    print(f"Speedup: {speedup:.2f}")
    print(f"Efficiency: {efficiency:.2f}")

if name == "main":
    main()
```

Output:

Serial processing time: 1.9919 seconds

Parallel processing time: 0.6161 seconds

Speedup: 3.23

Efficiency: 0.40

Detailed Speedup and Efficiency Report

This report provides a detailed analysis of speedup and efficiency for a palindrome finding experiment using both serial and parallel processing techniques in Python. The experiment compares the time taken to process a set of files in serial (one at a time) versus parallel processing (utilizing multiple CPU cores).

Experiment Details:

In this experiment, a series of files were generated, each containing a string with 1000 characters and a 5-letter palindrome embedded in it. The task was to search each file to check if a palindrome exists. The processing times for both serial and parallel methods were recorded.

Serial Processing

Time: 1.9919 seconds

Parallel Processing

Time: 0.6161 seconds

Number of CPU cores

used : 16

Speedup:

3.23

Efficiency : 0.40

The parallel processing method shows a significant reduction in execution time compared to the serial method, with a speedup of 3.23x. However, the efficiency is 0.40, indicating that while parallelism improves performance, it doesn't perfectly scale with the number of CPU cores due to overhead and synchronization costs. This highlights the benefit of parallelism, especially for embarrassingly parallel tasks like searching multiple files independently.

This is the link for all the codes Here