Operating Systems

Complex Computing Problem

Muhammad Shakaib Arsalan (F2022266626)

Muhammad Husnain (F2022266533)

Zaid Rasool (F20222666)

Course Code: CC3011L

Section: V5

Course Instructor: Habiba Habib

School of Systems and Technology

UMT Lahore Pakistan

Estimating Pi using the Maclaurin Series

We have to calculate the value of π using the Maclaurin series for $tan^{-1}(x)$. This is the mathematical computational problem. Find π using Maclaurin series for is one the effective approach.

We can express π as:

$$\pi = 4 * \sum_{n=0}^{\infty} \frac{-1^n}{2n+1}$$

$$\pi = 4 \text{ x } (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots)$$

Eq shows an optimized version of the Maclaurin Series for $\tan^{-1}(1)$ and $\tan^{-1}(1) = \frac{\pi}{4}$

The goal is to implement parallel computing approach to approximate π using series. By ensuring the equal distribution of workload using multithread and multiprocessing.

1. Sequential Implementation

Algorithm:

The sequential implementation uses a single process and thread, and estimates PI in a single loop.

Code:

Refer to ‘sequential.cpp’ for the complete implementation.

Steps:

1. Read the number of terms, n, from the command line.
2. Initialize a global variable PI to store the computed result.
3. Loop from 0 to n - 1 and compute the sum using the formula.
4. Multiply the result by 4 to get the final value of π.

Discussion:

This solution is simple but lacks parallelism, resulting in high execution time for larger values of n. It does not efficiently utilize available system resources.

2. Multiprocessor Implementation Details

Algorithm:

The multiprocessor implementation uses shared memory with processes to distribute computation.

Steps:

Create shared memory for storing the global PI result.

1. Fork child processes based on the number of requested processes.
2. Each process computes a partial sum of the Maclaurin series and writes to shared memory.
3. Use locks to synchronize access to the shared π variable.
4. Parent process waits for all child processes to complete.

Code:

Refer to ‘shared-memory.cpp’ for the complete implementation.

Discussion:

This solution achieves better performance than the sequential implementation by utilizing multiple processes. However, synchronization through shared memory and locks introduces some complexity and potential contention.

3. Multithreading-based Solution

Algorithm:

The multithreading approach divides the computation across multiple threads, synchronizing access to a shared π variable using mutex locks.

Steps:

1. Initialize a global variable π and a mutex lock.
2. Spawn threads based on the user-specified thread count.
3. Each thread computes a partial sum of the Maclaurin series.
4. Threads acquire the lock to update the shared π variable and release it afterward.
5. Wait for all threads to finish computation.
6. Compute the final value by multiplying π by 4.

Code:

See `thread.cpp` for the complete implementation.

Discussion:

This solution provides faster execution and better resource utilization compared to the process-based approach. Proper lock management ensures accurate results.

4. Results and Performance Analysis

Test Environment:

Processor: Intel Core i7 11th gen.

Operating System: Kali-Linus-2024.3

Performance Comparison:

Metric	Sequential	Multiprocessing	Multithreading
Execution Time	Slow	Moderate	Fast
Sync. Overhead	None	High	Low
Resource Utilization	Low	High	High
Scalability	Poor	Moderate	Excellent

Observations:

1. The multithreading solution provided the best performance due to reduced context-switching overhead compared to processes.
2. Synchronization was handled efficiently using mutex locks.
3. The sequential approach was significantly slower, demonstrating the necessity for parallel solutions.

5. Flowchart for Implementation Method

Refer to the attached flowchart figures showing the steps for each implementation.

6. Discussion on Locks and Synchronization

Mutex locks were used to protect critical sections when updating the global π variable.
Accurate placement of locks ensured correctness without introducing unnecessary contention.
Shared memory with atomic operations was used in the process-based solution to synchronize access.

7. Code Files

1. **sequential.cpp**: Sequential Implementation
2. **shared-memory.cpp**: Multiprocessor Implementation using Shared Memory
3. **pipes.c**pp: Process Communication using Pipes
4. **thread.cpp**: Multithreading-based Solution

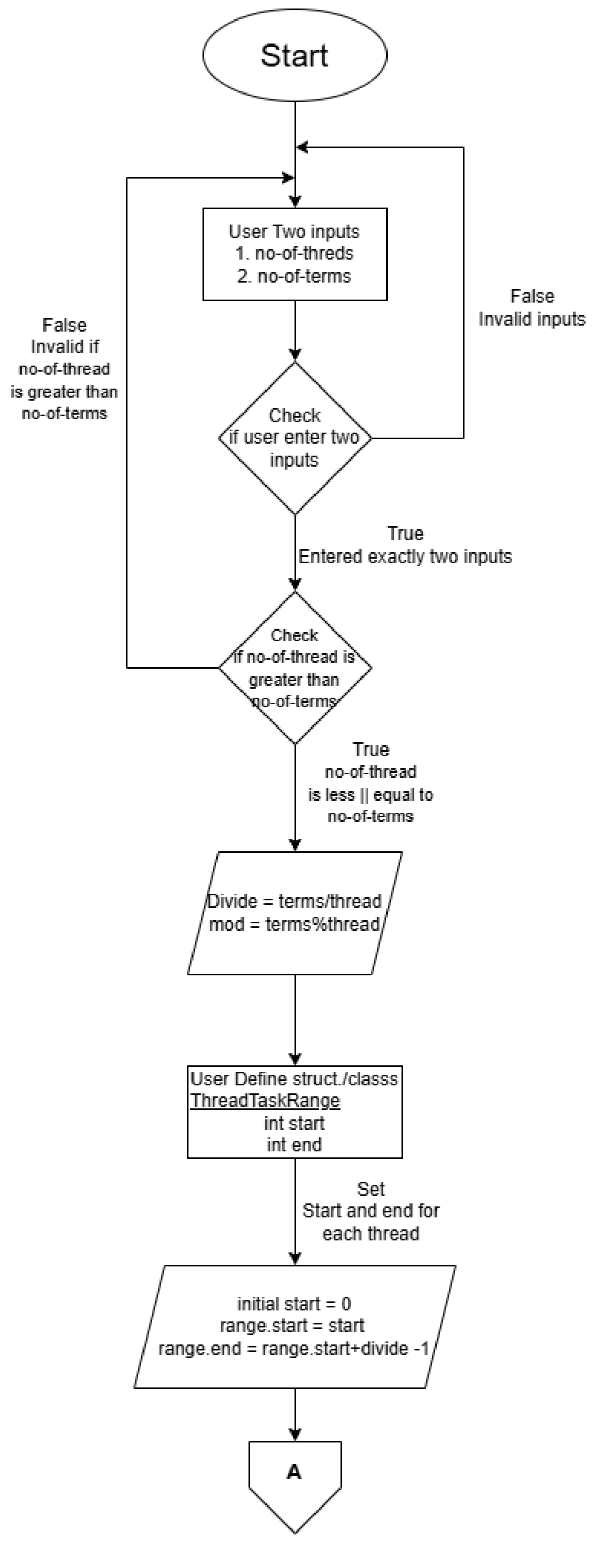
8. Performance Analysis Discussion

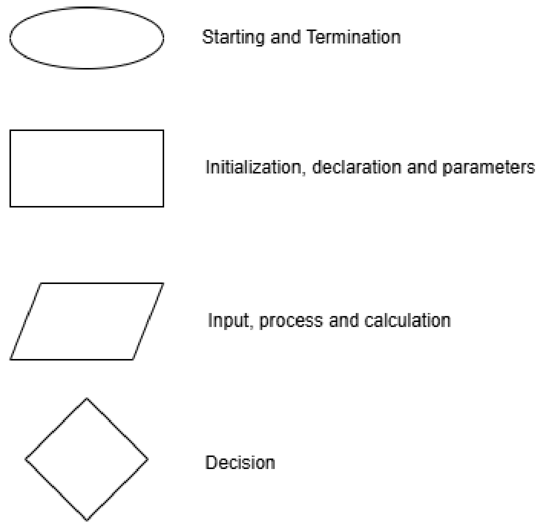
1. The parallel solutions showed significant improvement over the sequential implementation.
2. Proper workload distribution and synchronization played a crucial role in achieving accurate and efficient results.
3. Load balancing was achieved by evenly dividing the terms among threads and processes.

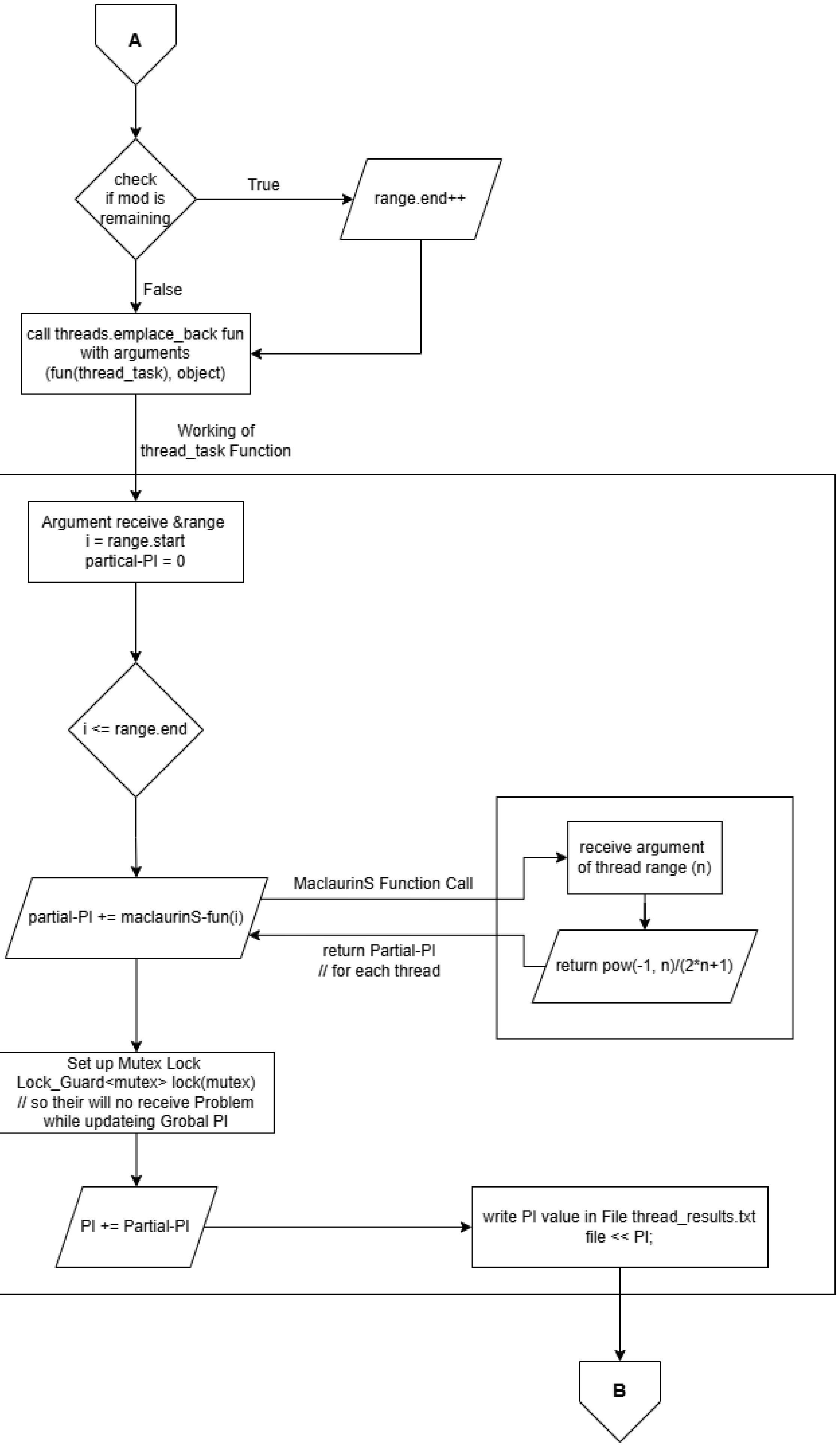
9. Conclusion

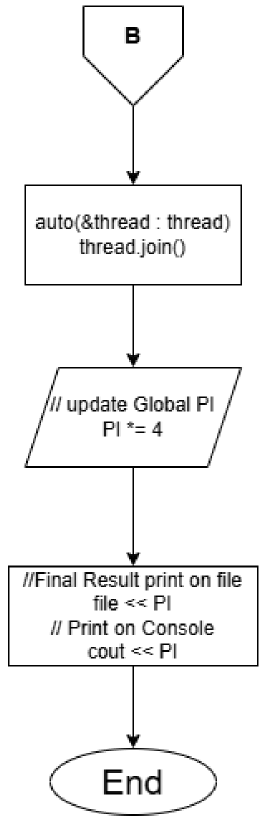
This project successfully demonstrated the computation of Pi using the Maclaurin series. Different parallel approaches were implemented and compared, highlighting the advantages of multithreading for computationally intensive tasks. Effective lock management and synchronization ensured accuracy and efficiency.

Flowchart – Multithreaded




Starting and Termination
Initialization, declaration and parameters
Input, process and calculation
Decision


A
check if mod is remaining
True
range.end++
False
call threads.emplace_back fun with arguments (fun(thread_task), object)
Working of thread_task Function
Argument receive &range
i = range.start
partical-PI = 0
i <= range.end
partial-PI += maclaurinS-fun(i)
MaclaurinS Function Call
receive argument of thread range (n)
return pow(-1, n)/(2*n+1)
return Partial-PI
// for each thread
Set up Mutex Lock
Lock_Guard<mutex> lock(mutex)
// so their will no receive Problem while updateing Grobal PI
PI += Partial-PI
write PI value in File thread_results.txt
file << PI;
B



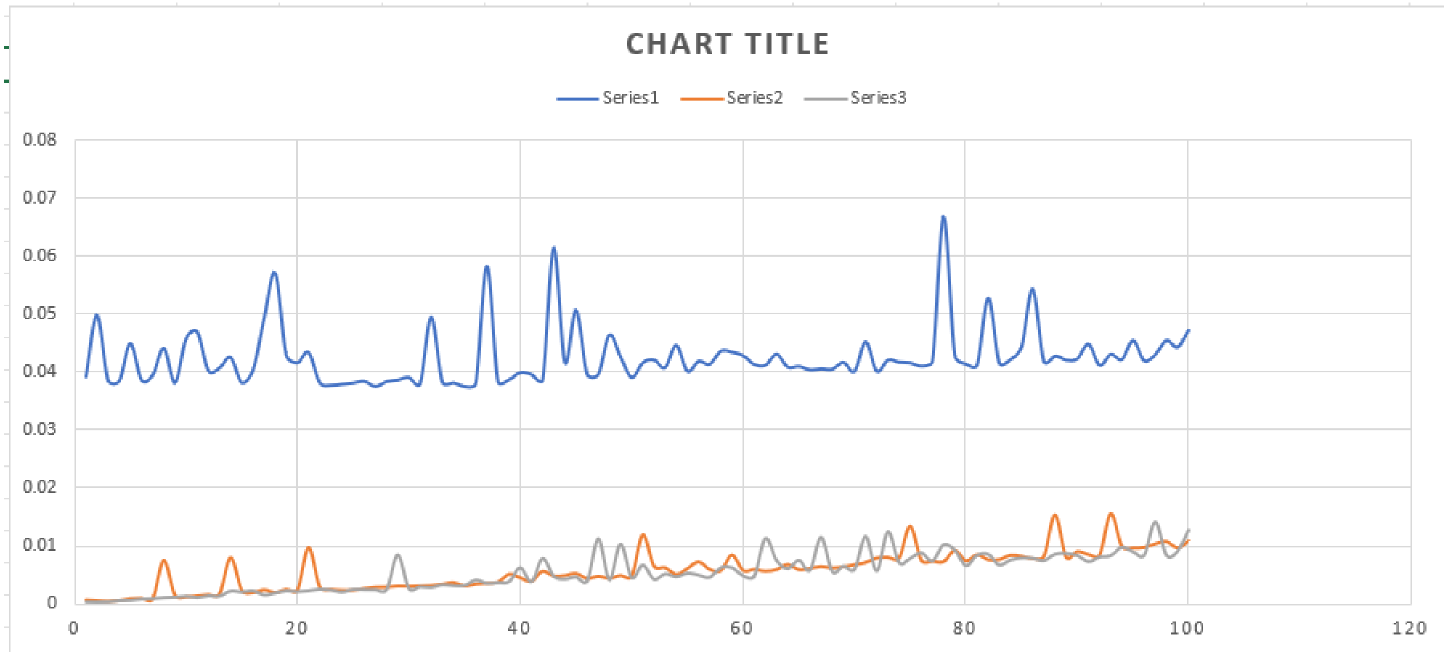
Performance Comparison:

X is instances from 1 to 100

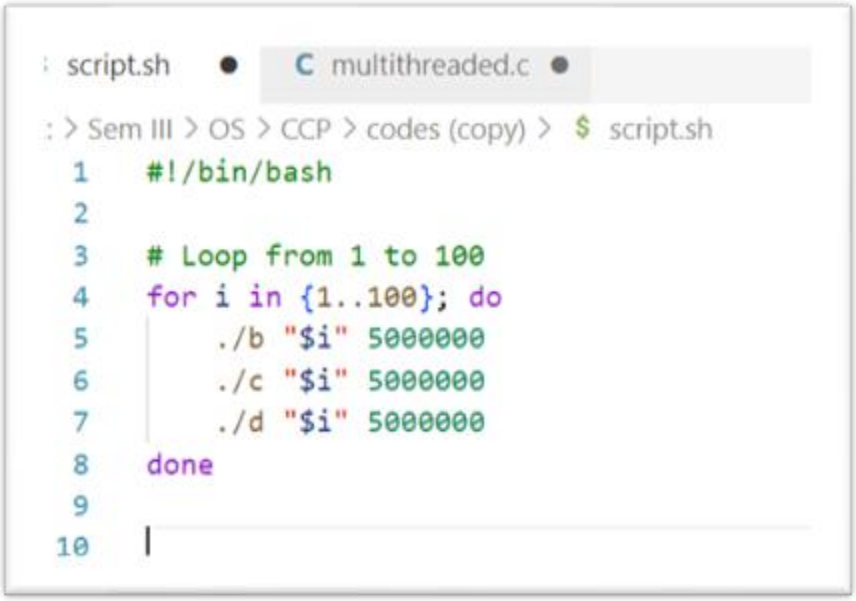
Series 1: Multithreaded

Series 2: Shared Memory

Series 3: Pipes



Bash Script:



```bash
#!/bin/bash

# Loop from 1 to 100
for i in {1..100}; do
    ./b "$i" 5000000
    ./c "$i" 5000000
    ./d "$i" 5000000
done

```