Operating Systems

Complex Computing Problem

Muhammad Shakaib Arsalan (F2022266626)

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:

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

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

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:
Click to access 🔗 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:
Click to access 🔗 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: Click to access 🔗 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 5th 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

🔗 Performance Comparison Graph

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. Discussion on Locks and Synchronization

Mutex locks are used to enter in critical sections when updating the global PI variable.
exact placement of locks ensured correctness of value of updated PI
Shared memory with atomic operations was used in the process-based solution to synchronize access.

6. Code Files

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

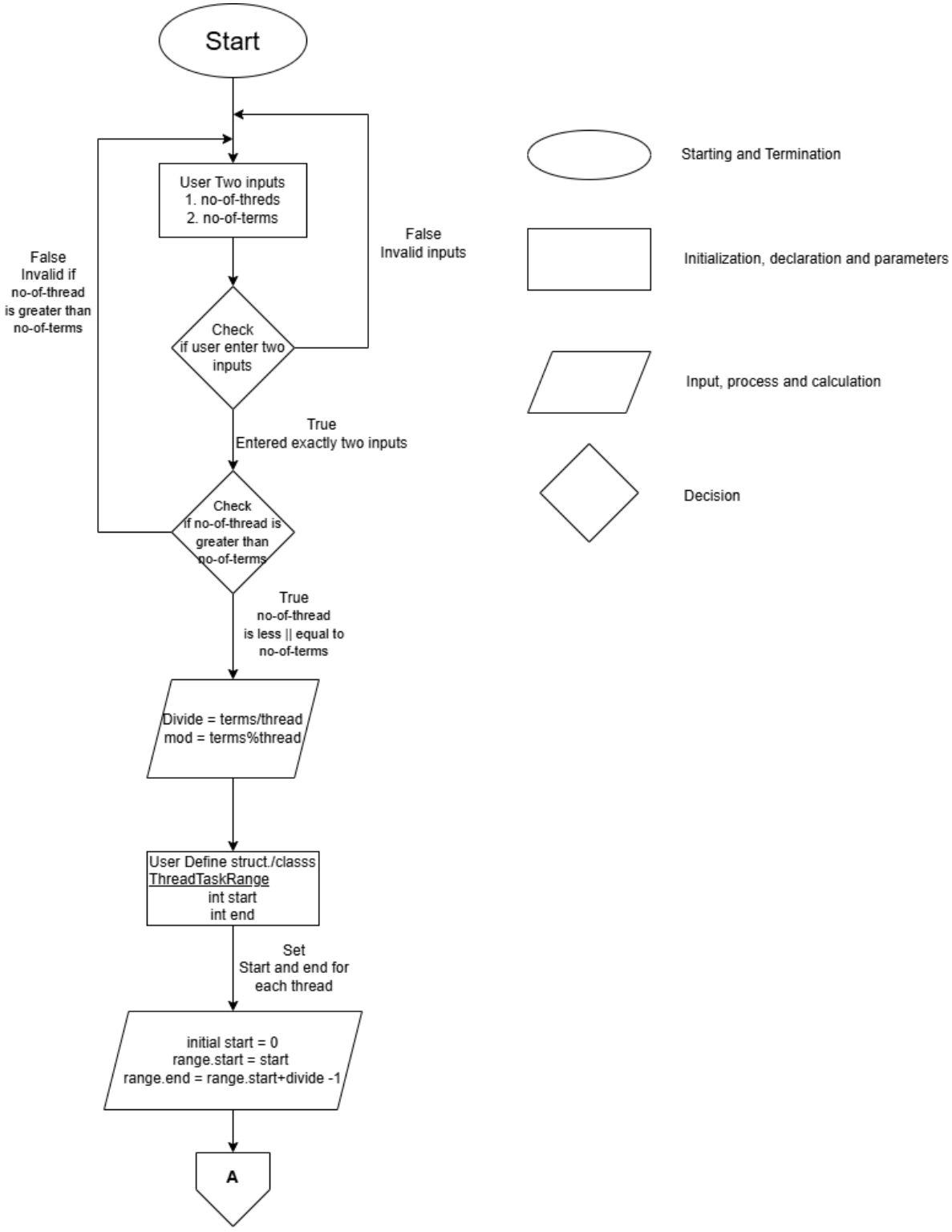
7. Performance Analysis Discussion

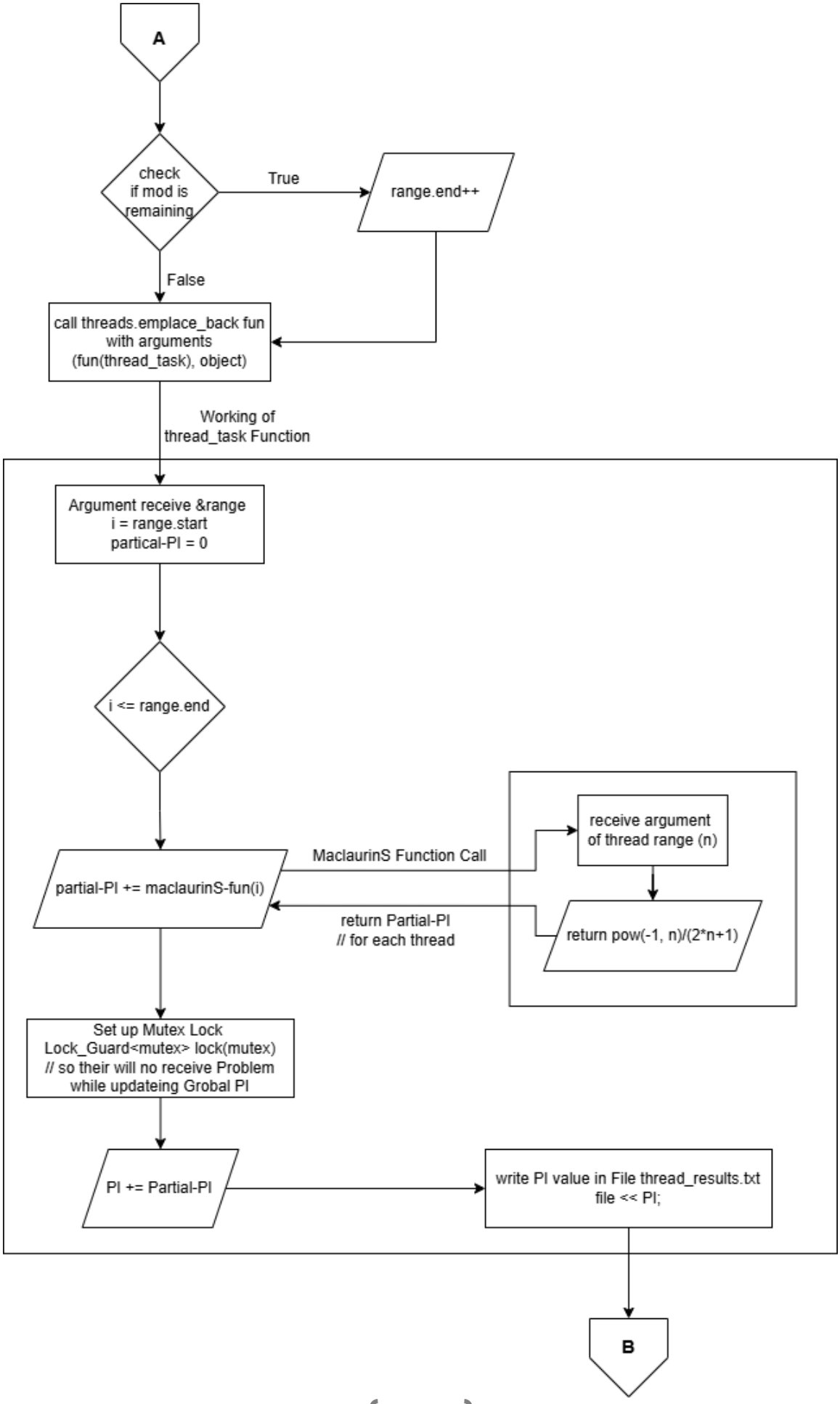
1. The parallel solutions showed significant improvement over the sequential implementation.
2. Proper workload distribution and synchronization played a crucial role in achieving accurate and efficient results.
3. Load balancing was achieved by evenly dividing the terms among threads and processes.

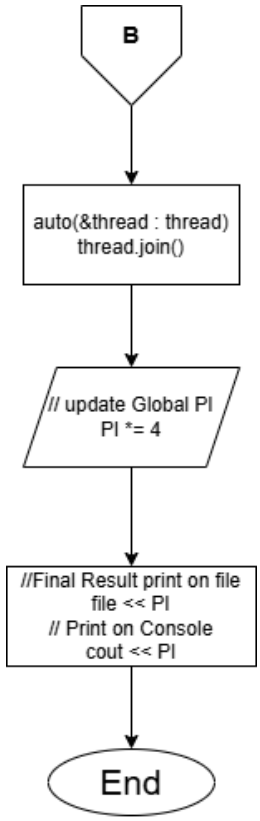
8. Conclusion

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


Start
User Two inputs
1. no-of-threds
2. no-of-terms
Check
if user enter two
inputs
False
Invalid inputs
False
Invalid if
no-of-thread
is greater than
no-of-terms
True
Entered exactly two inputs
Check
if no-of-thread is
greater than
no-of-terms
True
no-of-thread
is less || equal to
no-of-terms
Divide = terms/thread
mod = terms%thread
User Define struct./classs
ThreadTaskRange
int start
int end
Set
Start and end for
each thread
initial start = 0
range.start = start
range.end = range.start+divide -1
A
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 Partial-PI
// for each thread
return pow(-1, n)/(2*n+1)
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



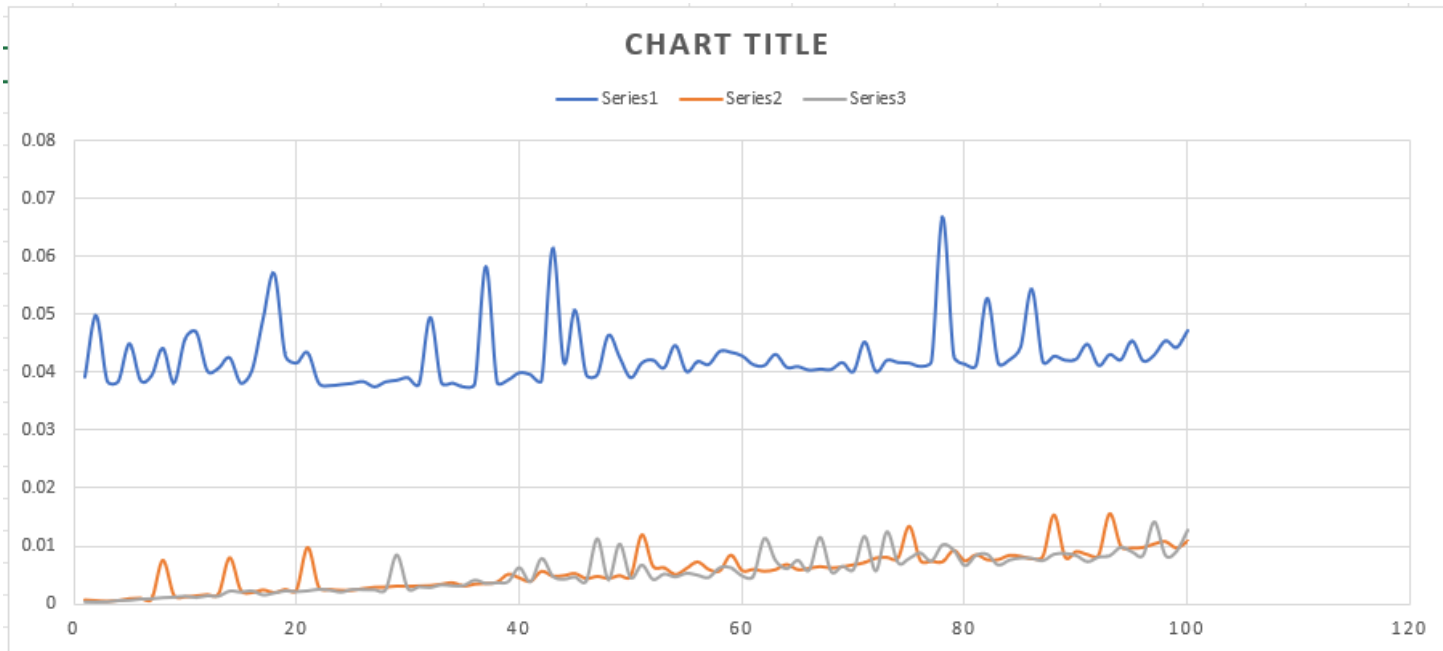
Performance Comparison:

X is instances from 1 to 100

Series 1: Multithreaded

Series 2: Shared Memory

Series 3: Pipes



Bash Script:

script.sh ● C multithreaded.c ●

: > Sem III > OS > CCP > codes (copy) > $ script.sh

```bash
#!/bin/bash

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

```

Code

sequential.cpp

```cpp

#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

long double maclaurin_polynomial_for_PI(unsigned int n)
{
    return pow(-1, n) / (2 * n + 1);
}

double compute_pi(unsigned int n)
{
    double pi = 0;
    for (unsigned int i = 0; i < n; i++)
    {
        pi += maclaurin_polynomial_for_PI(i);
    }
    return pi * 4;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Invalid argument count\n";
        return 1;
    }

    unsigned int n = static_cast<unsigned int>(atoi(argv[1]));
    double PI = compute_pi(n);

    cout << "PI: " << PI << endl;

    return 0;
}
```

pipes.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include <sys/wait.h>
#include <cmath>

using namespace std;

long double maclaurin_polynomial_for_PI(unsigned int n) {
    return ((n % 2) ? -1.0L : 1.0L) / (2 * n + 1);
}

long double child_func(int start, int end) {
    long double local_sum = 0.0L;
    for (unsigned int i = start; i <= end; i++) {
        local_sum += maclaurin_polynomial_for_PI(i);
    }
    return local_sum;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        cerr << "Usage: " << argv[0] << " <num_processes> <num_terms>" << endl;
        return 1;
    }

    int no_of_processes = atoi(argv[1]);
    int no_of_terms = atoi(argv[2]);

    if (no_of_processes < 1 || no_of_terms < 1) {
        cerr << "Number of processes and terms must be at least 1." << endl;
        return 1;
    }

    int pipes[no_of_processes][2];
    int div_each = no_of_terms / no_of_processes;
    int mod_remain = no_of_terms % no_of_processes;
    int start = 0;

    for (int i = 0; i < no_of_processes; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("Pipe error");
            return 1;
        }
```

```
        int end = start + div_each - 1;
        if (i < mod_remain) end++; // Distribute remainder

        pid_t pid = fork();

        if (pid == -1) {
            perror("Fork error");
            return 1;
        }
        else if (pid == 0) { // Child process
            close(pipes[i][0]); // Close read end
            long double partial_sum = child_func(start, end);
            write(pipes[i][1], &partial_sum, sizeof(long double));
            close(pipes[i][1]);
            exit(0);
        }

        start = end + 1; // Update for next process
    }

    long double PI = 0.0L;

    // Parent process collects results
    for (int i = 0; i < no_of_processes; i++) {
        close(pipes[i][1]); // Close write end
        long double partial_sum;
        read(pipes[i][0], &partial_sum, sizeof(long double));
        close(pipes[i][0]);
        wait(NULL); // Wait for child
        PI += partial_sum;
    }

    PI *= 4;
    cout.precision(20);
    cout << "\nPI is: " << PI << endl;

    return 0;
}
```

shared-memory.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <cmath>
#include <atomic>
using namespace std;

typedef atomic_flag lock_t;

// #define LOCK_INIT ATOMIC_FLAG_INIT

static inline void acquire(lock_t *lock)
{
    while (lock->test_and_set(memory_order_acquire));
}

static inline void release(lock_t *lock)
{
    lock->clear(memory_order_release);
}

long double maclaurin_polynomial_for_PI(unsigned int n)
{
    return ((n % 2) ? -1.0L : 1.0L) / (2 * n + 1);
}

void child_func(int start, int end, long double *PI, lock_t *mutex)
{
    long double local_sum = 0.0L;
    for (int i = start; i <= end; i++)
    {
        local_sum += maclaurin_polynomial_for_PI(i);
    }

    acquire(mutex);
    *PI += local_sum; // Write to shared memory
    release(mutex);
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        cerr << "Invalid argument count" << endl;
        return 1;
    }

    int no_of_child = atoi(argv[1]);
    int no_of_terms = atoi(argv[2]);

    if (no_of_child < 1 || no_of_terms < 1)
    {
        cerr << "Number of processes and terms should be at least 1." << endl;
        return 1;
    }
```

```
60      long double *PI = (long double *)mmap(NULL, sizeof(long double), PROT_READ | PROT_WRITE, MAP_SHARED |
        MAP_ANONYMOUS, -1, 0);
61      lock_t *mutex = (lock_t *)mmap(NULL, sizeof(lock_t), PROT_READ | PROT_WRITE, MAP_SHARED |
        MAP_ANONYMOUS, -1, 0);
62
63      if (PI == MAP_FAILED || mutex == MAP_FAILED)
64      {
65          perror("mmap error");
66          return 1;
67      }
68
69      *PI = 0.0L;
70      // *mutex = LOCK_INIT;
71      atomic_flag_clear(mutex);
72
73      int div_each = no_of_terms / no_of_child;
74      int mod_remain = no_of_terms % no_of_child;
75      int start = 0;
76
77      for (int i = 0; i < no_of_child; i++)
78      {
79          int end = start + div_each - 1;
80          if (i < mod_remain)
81              end++;
82
83          pid_t pid = fork();
84          if (pid == -1)
85          {
86              perror("fork error");
87              return 1;
88          }
89          else if (pid == 0)
90          { // Child process
91              child_func(start, end, PI, mutex);
92              exit(0);
93          }
94          start = end + 1;
95      }
96
97      // Parent waits for all children to finish
98      for (int i = 0; i < no_of_child; i++)
99      {
100         wait(NULL);
101     }
102
103     *PI *= 4;
104     long double error = 100 - ((*PI / 3.1415926535) * 100);
105     cout.precision(20);
106     cout << "\n\nPI is: " << fixed << *PI << endl;
107     cout << "error: " << error << endl;
108
109     munmap(PI, sizeof(long double));
110     munmap(mutex, sizeof(lock_t));
111
112     return 0;
113 }
```

thread.cpp

```cpp
#include <iostream>
#include <cmath>
#include <vector>
#include <thread>
#include <mutex>
#include <fstream>

long double PI = 0.0;
std::mutex mutex:
std::mutex file_mutex:

double maclaurin_polynomial_for_PI(unsigned int n)
{
    return std::pow(-1, n) / (2 * n + 1);
}

class ThreadTaskRange
{
public:
    unsigned int start;
    unsigned int end;
};

void thread_task(const ThreadTaskRange &range, std::ofstream &file)
{
    long double partialPI = 0.0;

    for (unsigned int i = range.start; i <= range.end; ++i)
    {
        partialPI += maclaurin_polynomial_for_PI(i);
    }

    {
        std::lock_guard<std::mutex> lock(mutex);
        PI += partialPI;
    }

    {
        std::lock_guard<std::mutex> file lock(file_mutex);
        file << "Thread Range: " << range.start << " to " << range.end << ", Partial PI: " << partialPI
    }
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        std::cout << "Invalid argument count\n";
        return 1;
    }

    unsigned int no_of_threads = std::stoul(argv[1]);
    unsigned int no_of_terms = std::stoul(argv[2]);

    std::cout << "Thread count: " << no_of_threads << "\n";
    std::cout << "# of terms: " << no_of_terms << "\n";
```

```
    if (no_of_threads > no_of_terms || no_of_terms < 1 || no_of_threads < 1)
    {
        std::cout << "Invalid inputs. Ensure no_of_terms >= no_of_threads and both are > 0.\n";
        return 1;
    }

    unsigned int div_each = no_of_terms / no_of_threads;
    unsigned int mod_remain = no_of_terms % no_of_threads;

    unsigned int start = 0;

    std::vector<std::thread> threads;
    std::ofstream file("thread_results.txt");

    if (!file.is_open())
    {
        std::cout << "Error opening file\n";
        return 1;
    }

    {
        std::lock_guard<std::mutex> file_lock(file_mutex);
        file << "Calculating PI using threads:\n";
    }

    for (unsigned int i = 0; i < no_of_threads; ++i)
    {
        ThreadTaskRange range;
        range.start = start;
        range.end = range.start + div_each - 1;

        if (i < mod_remain)
        {
            range.end++;
        }

        threads.emplace_back([=, &file]()
                             { thread_task(range, file); });

        start = range.end + 1;
    }

    for (auto &thread : threads)
    {
        thread.join();
    }

    {
        std::lock_guard<std::mutex> file_lock(file_mutex);
        file << "\nValue of PI before *4: " << PI;
    }

    PI *= 4;

    {
        std::lock_guard<std::mutex> file_lock(file_mutex);
        file << "\nFinal PI value: " << PI << "\n";
    }

    file.close();
    std::cout << "\nPI is: " << PI << "\n";

    return 0;
}
```

Turnitin Plagiarism Report

0% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Detection Groups

1 AI-generated only 0%
Likely AI-generated text from a large-language model.

2 AI-generated text that was AI-paraphrased 0%
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.