**Pi Estimation using Maclaurin Series**

**Abstract**  
In this project we use multithreading synchronization technique to find the value of pi using Maclaurin series for arctan(1). This program computes using multiple threads because of thread safe updates to a shared result using mutex locks. We use parallel implementation rather than the sequential for the better performance and efficiency. This program ensures that shared variable should produce a correct result.

**Introduction**  
Computing Pi is a classical problem in computation. This project uses the Maclaurin expansion series for arctan(1) to calculate Pi. Since the computation requires summations repeated many times, parallelism can greatly reduce the execution time for the program. In this implementation, multithreading is used together with proper synchronizations to handle the shared resource. All work adheres to load balancing constraints and allows a scalable solution for large computations (>1,000,000).

**Problem Analysis**  
The Maclaurin Series for determining Pi is as follows:  
**arctan(1) = pi/4, we can compute pi = 4\*[1 - 1/3 + 1/5 - 1/7 + 1/9 - . . . ]**

**Conflicting Requirements:**

* Give correct balance to all threads
* Best solution for synchronization to avoid race condition

**Constraints:**

1. n>1,000,00
2. The final shared result must be safely updated with correct value
3. Balance Distribution of numbers between threads for better performance and efficiency

**Sequential Implementation:**  
It is the process in which only a single loop runs and calculated all terms of the series and them sum every term on by on, it is the simplest method but it is an inefficient for large values.

**Sequential Code:**

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

int n = 1000000; // value of n

double pi = 0.0;

cout << "n: " << n << "\n";

for (int i = 0; i < n; ++i)

{

double term;

if (i % 2 == 0)

{

term = 1.0 / (2 \* i + 1);

}

else

{

term = -1.0 / (2 \* i + 1);

}

pi += term;

}

pi \*= 4;

cout << "Value of pi: " << pi << "\n";

return 0;

}

**Parallel Implementation (Multithreading):**   
In this process, the terms are divided to the multiple threads. Each thread then calculated given terms and then store the result in shared variable result using mutex locks to be on a safe side for avoiding race condition. It also do the computation more efficiently.

**Parallel Code:**

#include <iostream>

#include <pthread.h>

#include <cmath>

#include <mutex>

using namespace std;

double pi = 0.0;

mutex mx;

struct ThreadArgs //Construction Thread

{

int start;

int end;

};

void \*calculate\_pi(void \*args) //Calculation pi function

{

ThreadArgs \*threadArgs = (ThreadArgs \*)args;

double sum = 0.0;

for (int i = threadArgs->start; i < threadArgs->end; ++i)

{

double term = (i % 2 == 0 ? 1.0 : -1.0) / (2 \* i + 1);

sum += term;

}

{

lock\_guard<mutex> lock(mx);

pi += sum;

}

pthread\_exit(nullptr);

}

int main()

{

int num\_threads = 4;

int num\_values = 1000000;

cout << "number of threads: " << num\_threads << "\n";

cout << "number of values: " << num\_values << "\n";

pthread\_t threads[num\_threads];

pthread\_attr\_t attrs[num\_threads];

ThreadArgs args[num\_threads];

int values\_per\_thread = num\_values / num\_threads;

for (int t = 0; t < num\_threads; ++t)

{

args[t].start = t \* values\_per\_thread;

if (t == num\_threads - 1)

{

args[t].end = num\_values;

} else

{

args[t].end = args[t].start + values\_per\_thread;

}

pthread\_attr\_init(&attrs[t]);

pthread\_create(&threads[t], &attrs[t], calculate\_pi, &args[t]);

}

for (int t = 0; t < num\_threads; ++t)

{

pthread\_join(threads[t], nullptr);

}

pi \*= 4;

cout << "value of pi: " << pi << "\n";

return 0;

}

**Results and Performance Analysis**

**Result:**  
Number of threads: 4  
Value of n: 100,000  
Estimated Values of Pi: 3.14159

**Performance Comparison:**

The execution time using Multithreading approach is much lesser than the Sequential Approach also the result is much accurate in multithreading and it is more efficient

**Analysis:**

* Multiprogramming decreases the computation time for higher values of n.
* Mutex ensures correctness and prevent race condition

**Challenges and Measures**

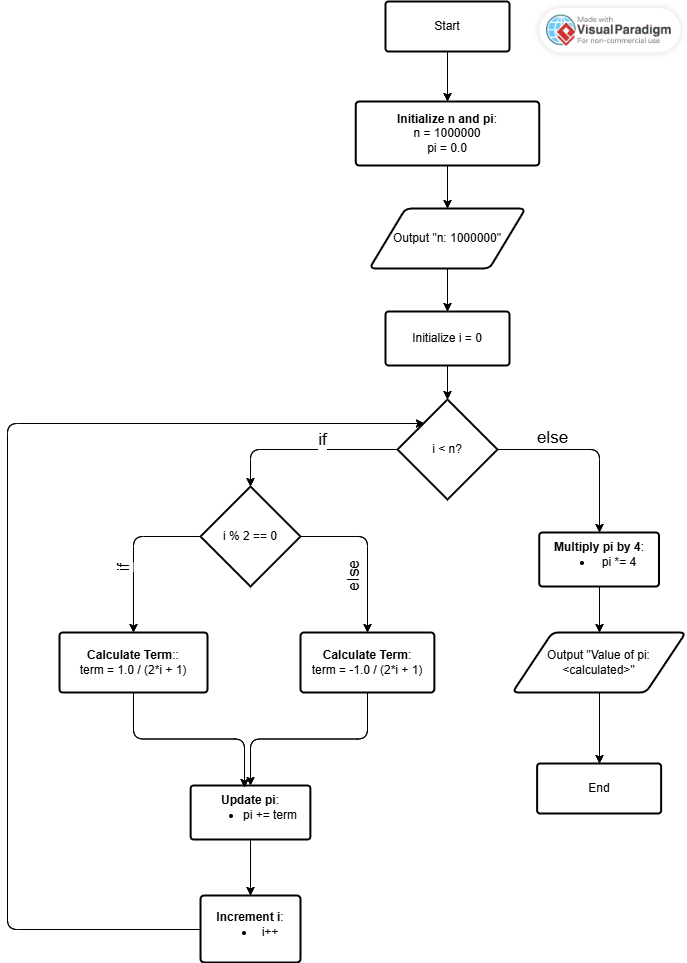
**Challenge 1:** Data gets incorrect due to updating the value of pi again and again.

**Solution:** Mutex is used to prevent this.

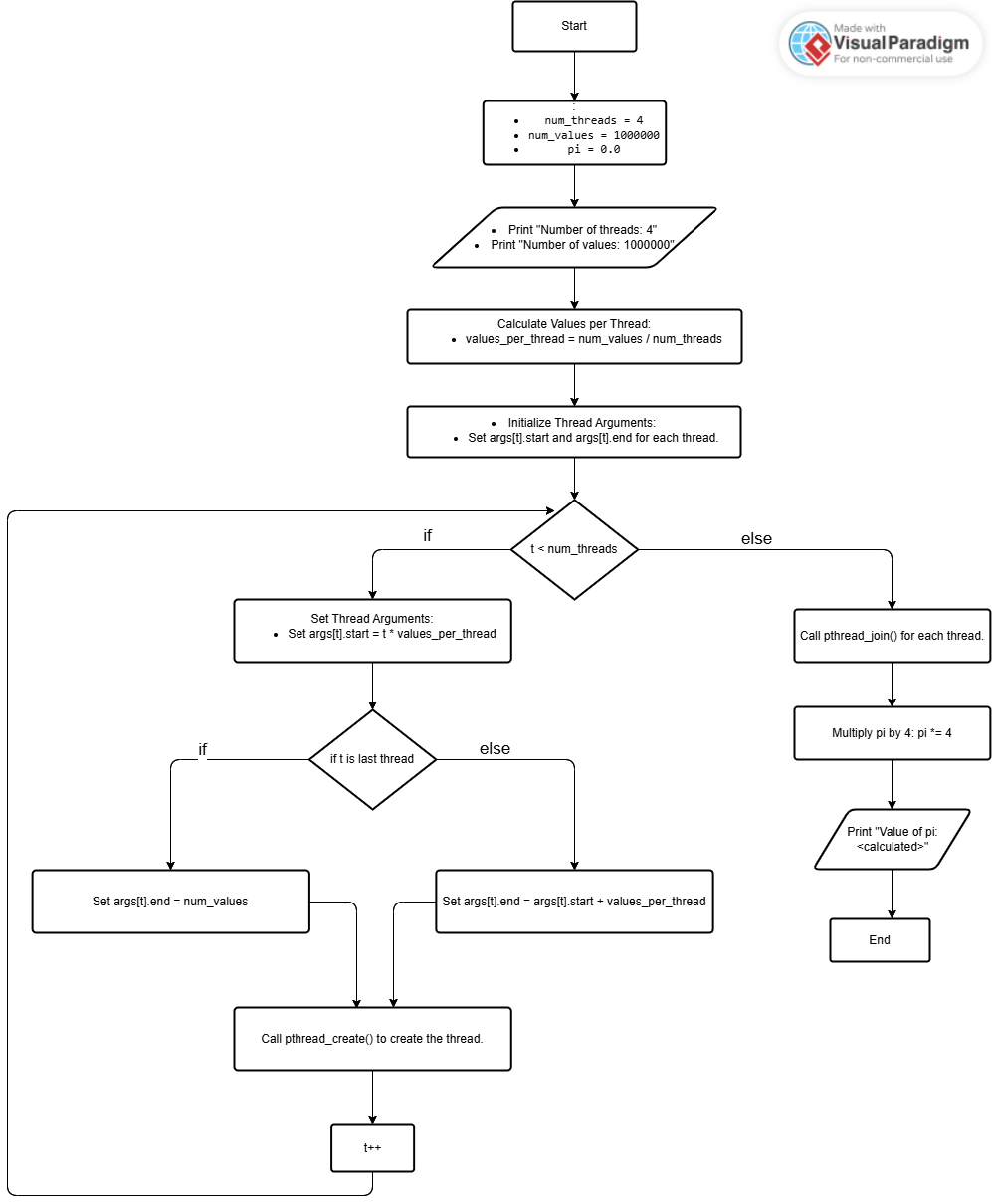
**Challenge 2:** Number of terms not evenly divided among threads

**Solution:** The last thread should take the left terms and do computation on them

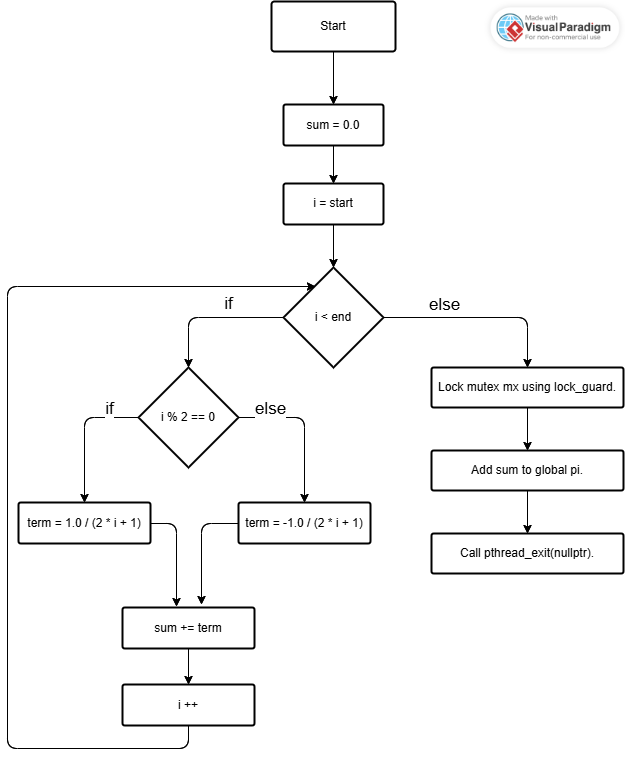
**Sequential Diagram:**

****

**Parallel Thread Diagram:**

****

**Pi Calculation Diagram:**

****

**Conclusion**  
In this project we use multithreading to calculate the value of pi using the McLaurin series. Because of multithreading and synchronization, we achieve the correct required result with a lesser computation time.