**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

# **Кафедра Вычислительной техники**

**ОТЧЕТ**

# **по лабораторной работе № 3.1**

**по дисциплине «Операционные системы» Тема: «Процессы и потоки»**

Студент гр. 3311 Пасечный Л.В.

Преподаватель Тимофеев А. В.

Санкт-Петербург

2025

# **Введение**

**Цель работы:**

исследовать механизмы создания и управления

процессами и потоками в ОС Windows.

# **Постановка задачи:**

# 1. Создайте приложение, которое вычисляет число pi с точностью

# N знаков после запятой по следующей формуле

# pi = (4/(1+x0^2) + 4/(1+x1^2) + ... + 4/(1+xN-1^2))\*(1/N);

# xi = (i + 0.5)\*(1/N), i = 0,999999;

# где N=10000000.

# Используйте распределение итераций блоками (размер блока = 10

# \* 3311) по потокам. Сначала каждый поток по очереди получает

# свой блок итераций, затем тот поток, который заканчивает

# выполнение своего блока, получает следующий свободный блок

# итераций. Освободившиеся потоки получают новые блоки

# итераций до тех пор, пока все блоки не будут исчерпаны.

# Создание потоков выполняйте с помощью функции Win32 API

# CreateThread.

# Для реализации механизма распределения блоков итераций

# необходимо сразу в начале программы создать необходимое

# количество потоков в приостановленном состоянии, для

# освобождения потока из приостановленного состояния

# используйте функцию Win32 API ResumeThread.

# По окончании обработки текущего блока итераций поток не должен

# завершаться, а должен быть приостановлен с помощью функции

# Win32 API SuspendThread. Затем потоку должна быть

# предоставлена следующий свободный блок итераций, и поток

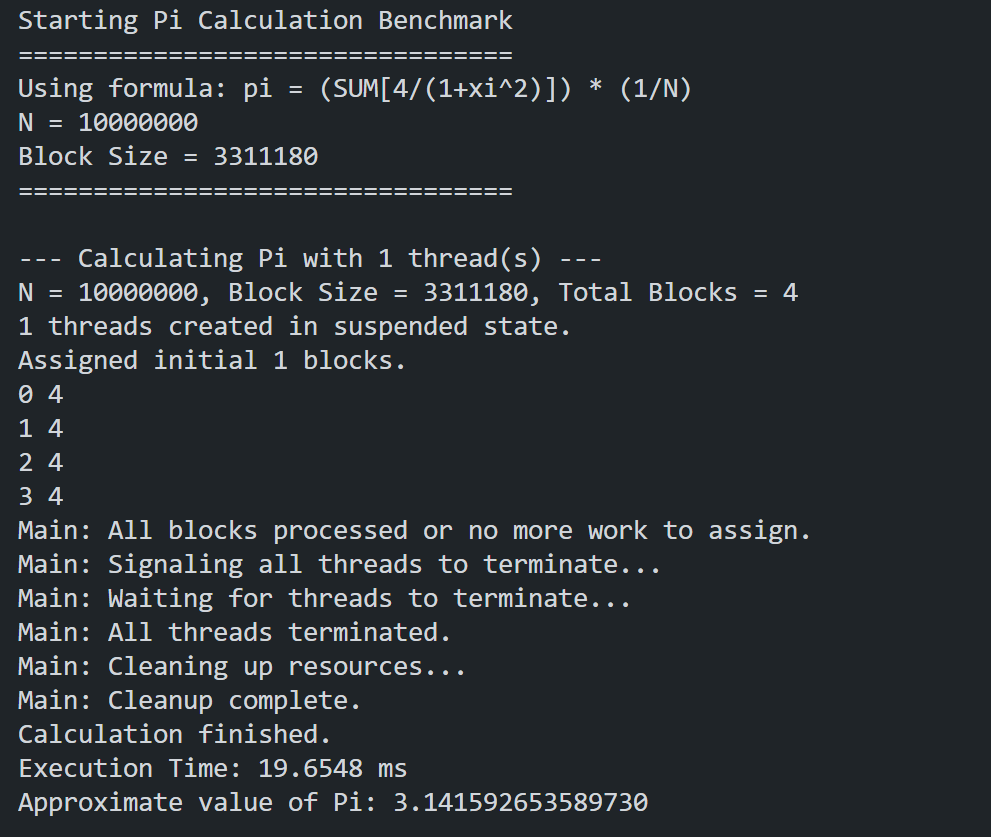
# должен быть освобожден (ResumeThread).

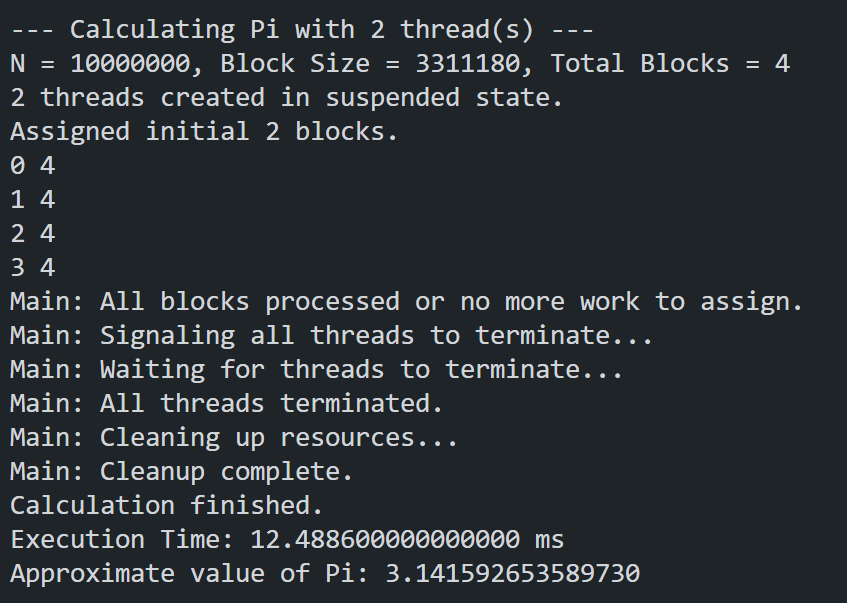
# 2

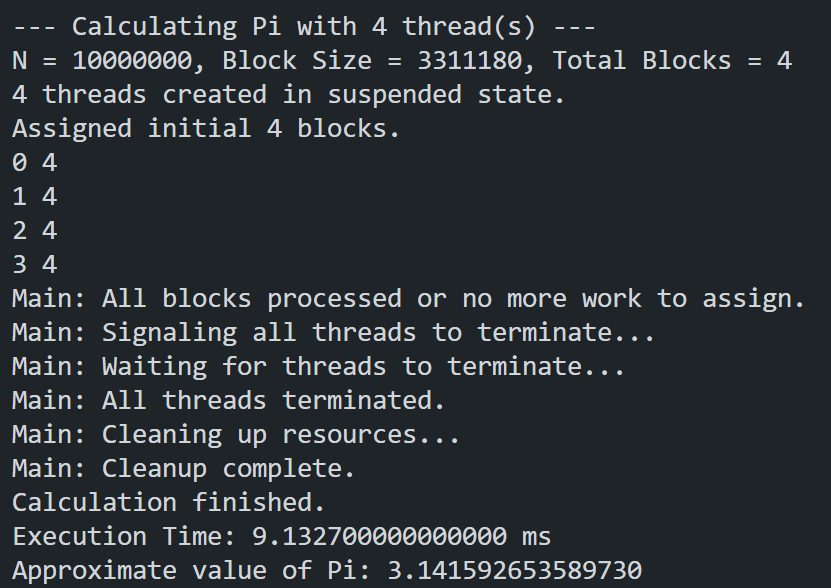
# 2. Произведите замеры времени выполнения приложения

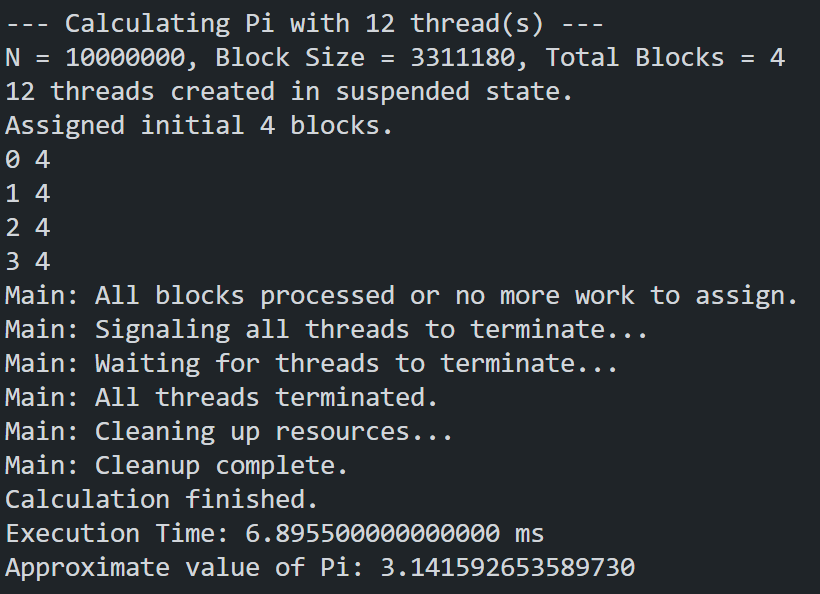
# для разного числа потоков (1, 2, 4, 8, 12, 16).

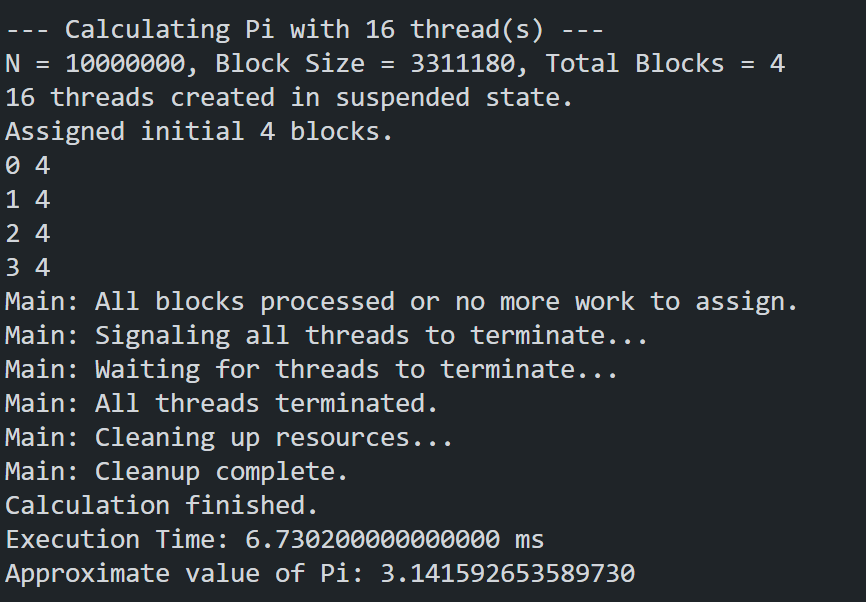
# **Результаты:**

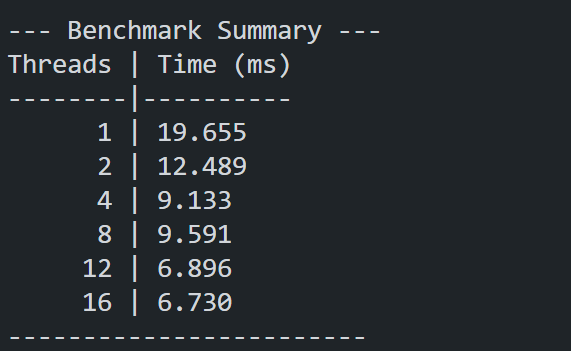












**Заключение**

В работе разработано многопоточное приложение для вычисления числа π с высокой точностью (N = 10,000,000 знаков после запятой) с использованием Win32 API (CreateThread, ResumeThread, SuspendThread). Задача была распараллелена путем динамического распределения блоков итераций (по 33110 итераций) между потоками: каждый поток брал блок, обрабатывал, приостанавливался (SuspendThread) и получал новый свободный блок при возобновлении (ResumeThread).

# **Код программы**

#include <windows.h>

#include <iostream>

#include <vector>

#include <cmath>

#include <iomanip>

#include <chrono>

#include <atomic>

#include <numeric> // Potentially for std::iota if needed, not strictly required here

using namespace std;

// --- Constants ---

const long long N = 10000000;

const int BLOCK\_SIZE\_FACTOR = 10;

const int BLOCK\_SIZE\_BASE = 331118;

const int BLOCK\_SIZE = BLOCK\_SIZE\_FACTOR \* BLOCK\_SIZE\_BASE; // 33110

const double N\_INV = 1.0 / N;

// --- Shared Data ---

std::atomic<int> nextBlockIndex;

CRITICAL\_SECTION sumCritSec;

double totalSum = 0.0;

int numTotalBlocks;

// Structure to pass data to each thread

struct ThreadData {

int threadId;

HANDLE hEvent; // Event handle for this thread to signal completion

HANDLE hThread; // Handle to the thread itself

int assignedBlock; // Block index assigned to this thread

bool shouldTerminate; // Flag to signal thread termination

};

// --- Thread Function ---

DWORD WINAPI ThreadFunction(LPVOID lpParam) {

ThreadData\* data = static\_cast<ThreadData\*>(lpParam);

int threadId = data->threadId;

HANDLE hEvent = data->hEvent;

double partialSum = 0.0;

// std::cout << "Thread " << threadId << " started and suspended (virtually, before first resume)." << std::endl;

while (true) {

// 1. Suspend until resumed by the main thread

// Note: The first time, it's already suspended from CreateThread.

// This call is for subsequent blocks.

// We'll Suspend \*after\* processing, before potentially looping again.

// Let's rethink: The main thread resumes, thread works, thread signals, thread suspends.

// --- Wait to be Resumed ---

// (Implicitly waiting after SuspendThread or initial creation state)

// Check if the main thread signaled termination \*before\* resuming

if (data->shouldTerminate) {

// std::cout << "Thread " << threadId << " received termination signal." << std::endl;

break;

}

// --- Process Assigned Block ---

int blockIdx = data->assignedBlock;

if (blockIdx < 0) {

// This should technically be caught by shouldTerminate, but as a safeguard

//std::cout << "Thread " << threadId << " received invalid block index, terminating." << std::endl;

break;

}

// std::cout << "Thread " << threadId << " resumed, processing block " << blockIdx << std::endl;

long long start\_iter = (long long)blockIdx \* BLOCK\_SIZE;

// Calculate end\_iter, ensuring it doesn't exceed N

long long end\_iter = min((long long)(blockIdx + 1) \* BLOCK\_SIZE, N);

partialSum = 0.0; // Reset partial sum for the new block

for (long long i = start\_iter; i < end\_iter; ++i) {

double x\_i = (static\_cast<double>(i) + 0.5) \* N\_INV;

partialSum += 4.0 / (1.0 + x\_i \* x\_i);

}

// --- Add partial sum to total (synchronized) ---

EnterCriticalSection(&sumCritSec);

totalSum += partialSum;

LeaveCriticalSection(&sumCritSec);

// --- Signal Completion and Suspend ---

// std::cout << "Thread " << threadId << " finished block " << blockIdx << ". Signaling and suspending." << std::endl;

// Signal main thread that this block is done

SetEvent(hEvent);

// Suspend self, waiting for potential next block or termination

SuspendThread(GetCurrentThread());

// --- Check for termination signal \*after\* being resumed ---

// (This happens at the top of the loop on the next iteration)

}

// std::cout << "Thread " << threadId << " exiting." << std::endl;

return 0; // Thread finished

}

// --- Main Calculation Function ---

double calculate\_pi\_parallel(int numThreads) {

if (numThreads <= 0) {

std::cerr << "Error: Number of threads must be positive." << std::endl;

return 0.0;

}

std::cout << "\n--- Calculating Pi with " << numThreads << " thread(s) ---" << std::endl;

// --- Initialization ---

totalSum = 0.0;

nextBlockIndex.store(0); // Reset block index counter

InitializeCriticalSection(&sumCritSec);

// Calculate total number of blocks needed

// Equivalent to ceil(N / (double)BLOCK\_SIZE)

numTotalBlocks = (N + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

std::cout << "N = " << N << ", Block Size = " << BLOCK\_SIZE << ", Total Blocks = " << numTotalBlocks << std::endl;

std::vector<ThreadData> threadDataList(numThreads);

std::vector<HANDLE> threadHandles(numThreads);

std::vector<HANDLE> eventHandles(numThreads);

// --- Create Events and Threads (Suspended) ---

for (int i = 0; i < numThreads; ++i) {

// Create an auto-reset event for signaling block completion

// Auto-reset: Automatically resets to non-signaled after a wait is satisfied.

// Manual-reset: Stays signaled until explicitly reset. Let's try manual.

// CreateEvent(SecurityAttributes, bManualReset, bInitialState, lpName)

eventHandles[i] = CreateEvent(NULL, TRUE, FALSE, NULL); // Manual-reset, initially non-signaled

if (eventHandles[i] == NULL) {

std::cerr << "Error: CreateEvent failed for thread " << i << ". Error code: " << GetLastError() << std::endl;

// Cleanup previously created events/threads if necessary

for(int j=0; j<i; ++j) CloseHandle(eventHandles[j]);

DeleteCriticalSection(&sumCritSec);

return 0.0;

}

threadDataList[i].threadId = i;

threadDataList[i].hEvent = eventHandles[i];

threadDataList[i].assignedBlock = -1; // No block assigned yet

threadDataList[i].shouldTerminate = false;

// Create thread in suspended state

threadHandles[i] = CreateThread(

NULL, // Default security attributes

0, // Default stack size

ThreadFunction, // Thread function

&threadDataList[i], // Argument to thread function

CREATE\_SUSPENDED, // Creation flags: Start suspended

NULL // Thread identifier (not needed)

);

if (threadHandles[i] == NULL) {

std::cerr << "Error: CreateThread failed for thread " << i << ". Error code: " << GetLastError() << std::endl;

// Cleanup

threadDataList[i].hEvent = NULL; // Mark event as invalid

CloseHandle(eventHandles[i]);

for(int j=0; j<i; ++j) {

// Try to signal termination to already created threads? Complex, maybe just exit.

CloseHandle(eventHandles[j]);

CloseHandle(threadHandles[j]);

}

DeleteCriticalSection(&sumCritSec);

return 0.0;

}

threadDataList[i].hThread = threadHandles[i]; // Store thread handle in data

}

std::cout << numThreads << " threads created in suspended state." << std::endl;

// --- Start Timing ---

auto startTime = std::chrono::high\_resolution\_clock::now();

int blocksProcessedCount = 0;

int blocksAssignedCount = 0;

std::vector<bool> threadWorking(numThreads, false); // Track if a thread is currently processing

// --- Initial Block Assignment ---

// Assign the first batch of blocks, up to the number of threads

int initialAssignments = 0;

for (int i = 0; i < numThreads && nextBlockIndex < numTotalBlocks; ++i) {

int blockIdx = nextBlockIndex.fetch\_add(1);

if (blockIdx < numTotalBlocks) {

threadDataList[i].assignedBlock = blockIdx;

threadDataList[i].shouldTerminate = false;

ResetEvent(eventHandles[i]); // Ensure event is non-signaled before starting

ResumeThread(threadHandles[i]);

// std::cout << "Main: Assigned initial block " << blockIdx << " to thread " << i << " and resumed." << std::endl;

threadWorking[i] = true;

blocksAssignedCount++;

initialAssignments++;

} else {

// Should not happen if nextBlockIndex started at 0, but as a safeguard

nextBlockIndex.store(numTotalBlocks); // Ensure no more blocks are fetched

break;

}

}

std::cout << "Assigned initial " << initialAssignments << " blocks." << std::endl;

// --- Main Management Loop ---

// Continue as long as not all blocks have been processed

while (blocksProcessedCount < numTotalBlocks) {

// Wait for \*any\* thread to signal completion

// We only wait on events of threads that we know are working

std::vector<HANDLE> workingEvents;

for(int i=0; i< numThreads; ++i) {

if(threadWorking[i]) {

workingEvents.push\_back(eventHandles[i]);

}

}

cout << blocksProcessedCount << " " << numTotalBlocks << endl;

if (workingEvents.empty()) {

// This might happen if numThreads > numTotalBlocks and all blocks are done

// or if something went wrong.

if (blocksProcessedCount < numTotalBlocks) {

std::cerr << "Warning: No threads seem to be working, but not all blocks are processed." << std::endl;

}

break; // Exit loop if no threads are working

}

DWORD waitResult = WaitForMultipleObjects(

workingEvents.size(), // Number of handles to wait on

workingEvents.data(), // Array of handles

FALSE, // Wait for ANY object (not all)

INFINITE // Wait indefinitely

);

if (waitResult >= WAIT\_OBJECT\_0 && waitResult < (WAIT\_OBJECT\_0 + workingEvents.size())) {

// An event was signaled

int signaledEventIndex = waitResult - WAIT\_OBJECT\_0;

HANDLE signaledEvent = workingEvents[signaledEventIndex];

// Find which thread corresponds to this event

int finishedThreadId = -1;

for (int i = 0; i < numThreads; ++i) {

if (eventHandles[i] == signaledEvent) {

finishedThreadId = i;

break;

}

}

if (finishedThreadId != -1) {

blocksProcessedCount++;

//std::cout << "Main: Thread " << finishedThreadId << " finished a block. Blocks processed: " << blocksProcessedCount << "/" << numTotalBlocks << std::endl;

// --- Assign Next Block (if available) ---

int blockIdx = nextBlockIndex.fetch\_add(1);

if (blockIdx < numTotalBlocks) {

// Assign new block and resume

threadDataList[finishedThreadId].assignedBlock = blockIdx;

threadDataList[finishedThreadId].shouldTerminate = false;

ResetEvent(eventHandles[finishedThreadId]); // Reset event before resuming

ResumeThread(threadHandles[finishedThreadId]);

// std::cout << "Main: Assigned block " << blockIdx << " to thread " << finishedThreadId << " and resumed." << std::endl;

threadWorking[finishedThreadId] = true; // Still working

blocksAssignedCount++;

} else {

// No more blocks, thread is done with work

// std::cout << "Main: No more blocks to assign to thread " << finishedThreadId << ". It remains suspended for now." << std::endl;

threadWorking[finishedThreadId] = false; // Mark as not working

// We don't necessarily need to signal termination yet,

// just don't give it more work. It's already suspended.

}

} else {

std::cerr << "Error: Could not find thread for signaled event." << std::endl;

// Potentially problematic state, maybe break?

break;

}

} else if (waitResult >= WAIT\_ABANDONED\_0 && waitResult < (WAIT\_ABANDONED\_0 + workingEvents.size())) {

std::cerr << "Error: Wait abandoned for an event. Thread might have terminated unexpectedly." << std::endl;

// Handle error - maybe try to find which thread and mark it as not working

break; // Exit loop on error

}

else {

// WAIT\_TIMEOUT or WAIT\_FAILED

std::cerr << "Error: WaitForMultipleObjects failed or timed out (INFINITE shouldn't timeout). Error code: " << GetLastError() << std::endl;

break; // Exit loop on error

}

}

std::cout << "Main: All blocks processed or no more work to assign." << std::endl;

// --- Stop Timing ---

auto endTime = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> duration = endTime - startTime;

// --- Shutdown Threads ---

std::cout << "Main: Signaling all threads to terminate..." << std::endl;

for (int i = 0; i < numThreads; ++i) {

threadDataList[i].shouldTerminate = true;

// We need to resume any thread that might be suspended

// so it can check the termination flag and exit.

// It's generally safe to resume a thread that's already running.

// If a thread already exited, ResumeThread might return an error, which is okay.

ResumeThread(threadHandles[i]);

}

std::cout << "Main: Waiting for threads to terminate..." << std::endl;

// Wait for all thread \*handles\* (not events) to signal termination

WaitForMultipleObjects(numThreads, threadHandles.data(), TRUE, INFINITE); // Wait for ALL to complete

std::cout << "Main: All threads terminated." << std::endl;

// --- Cleanup ---

std::cout << "Main: Cleaning up resources..." << std::endl;

for (int i = 0; i < numThreads; ++i) {

CloseHandle(threadHandles[i]);

CloseHandle(eventHandles[i]);

}

DeleteCriticalSection(&sumCritSec);

std::cout << "Main: Cleanup complete." << std::endl;

// --- Calculate Final Pi and Return ---

double pi\_approx = totalSum \* N\_INV;

std::cout << "Calculation finished." << std::endl;

std::cout << "Execution Time: " << duration.count() << " ms" << std::endl;

std::cout << "Approximate value of Pi: " << std::fixed << std::setprecision(15) << pi\_approx << std::endl;

return duration.count(); // Return time for reporting

}

// --- Main Function ---

int main() {

// List of thread counts to test

std::vector<int> threadCounts = {1, 2, 4, 8, 12, 16};

std::cout << "Starting Pi Calculation Benchmark" << std::endl;

std::cout << "=================================" << std::endl;

std::cout << "Using formula: pi = (SUM[4/(1+xi^2)]) \* (1/N)" << std::endl;

std::cout << "N = " << N << std::endl;

std::cout << "Block Size = " << BLOCK\_SIZE << std::endl;

std::cout << "=================================" << std::endl;

std::vector<double> executionTimes;

executionTimes.reserve(threadCounts.size());

for (int count : threadCounts) {

double time\_ms = calculate\_pi\_parallel(count);

if (time\_ms > 0) { // Store time if calculation was successful

executionTimes.push\_back(time\_ms);

} else {

// Handle error case if needed, maybe push a sentinel value or skip

executionTimes.push\_back(-1.0); // Indicate failure for this count

std::cerr << "Calculation failed for " << count << " threads. Skipping." << std::endl;

}

}

// --- Print Summary ---

std::cout << "\n--- Benchmark Summary ---" << std::endl;

std::cout << "Threads | Time (ms)" << std::endl;

std::cout << "--------|----------" << std::endl;

for (size\_t i = 0; i < threadCounts.size(); ++i) {

std::cout << std::setw(7) << threadCounts[i] << " | ";

if (executionTimes[i] >= 0) {

std::cout << std::fixed << std::setprecision(3) << executionTimes[i] << std::endl;

} else {

std::cout << "Failed" << std::endl;

}

}

std::cout << "------------------------" << std::endl;

return 0;

}