**Lab 2: Parallel Programming Using C++ Threads**

ECE 455: GPU Algorithm and System Design

Due: Submit completed PDF to Canvas by 23:59 PM 9/26

# Overview

In this lab you will practice parallel programming with the C++ standard threading library

(<thread>, <mutex>, <condition\_variable>, <atomic>). You will start with thread creation and joining, then move through synchronization and producer–consumer patterns, and finish with a compute-heavy but conceptually simple matrix multiplication.

# Learning Objectives

By the end of this lab, you should be able to:

* Create and manage threads in C++.
* Partition work among threads and measure performance.
* Identify and fix data races using mutexes and atomics.
* Coordinate threads using condition variables. • Apply threading to a nontrivial computation.

# Euler Instruction

~$ ssh your\_CAE\_account@euler.engr.wisc.edu

~$ sbatch your\_slurm\_scrip.slurm

You should NEVER run your program on the log-in node with the interactive mode. Doing so will risk your account being blocked by the IT. Instead, you should work on your local machine and set up a GitHub repo to transfer code from your local machine to your Euler node, and then compile and run it using a proper sbatch script.

# Submission Instruction

Specify your GitHub link here: <https://github.com/avins0114/ECE455/tree/main/HW02>

Note that your link should be of this format: https://github.com/YourGitHubName/ECE455/HW02

# Problem 1 — Hello, Multithreaded World

**Task.** Spawn *N* threads. Each thread prints "Hello from thread X of N" where X is the thread’s ID (0-based). Join all threads.

## Hints

Pass the thread ID as a function argument; store threads in a std::vector<std::thread> and call join() on each.

**Solution** hello\_threads.cpp

|  |
| --- |
| **#include** <iostream>  **#include** <**thread**>  **#include** <**vector**>  **void** hello(**int** id, **int** total) {  std::cout << "Hello from thread " << id << " of " << total << "\n";  }  **int** main() {  **const int** N = 5;  std::**vector**<std::**thread**> threads; threads.reserve(N);  **for** (**int** i = 0; i < N; ++i)  threads.emplace\_back(hello, i, N);  **for** (**auto** &t : threads) t.join(); **return** 0;  } |

p1.slurm

|  |
| --- |
| *#!/usr/bin/env zsh*  *#SBATCH --partition=instruction*  *#SBATCH --time=00:02:00*  *#SBATCH --ntasks=1*  *#SBATCH --cpus-per-task=4*  *#SBATCH --output=hello\_threads.output* **cd** $SLURM\_SUBMIT\_DIR  g++ -O2 -std=c++17 hello\_threads.cpp -o hello\_threads -pthread  ./hello\_threads |

# Problem 2 — Parallel Array Sum

**Task.** Create a large array of 107 ints, split into *T* segments, and sum in parallel. Combine partial sums and compare timing with a single-threaded baseline.

## Hints

Use std::accumulate for segments and std::chrono for timing. Store partials in a std::vector<long long>.

**Solution** parallel\_sum.cpp

|  |
| --- |
| **#include** <iostream>  **#include** <**vector**>  **#include** <**thread**>  **#include** <numeric>  **#include** <random>  **#include** <chrono>  **void** partial\_sum(**const** std::**vector**<**int**> &data, **size\_t** start, **size\_t** end, **long long** &out) {  out = std::accumulate(data.begin() + start, data.begin() + end, 0LL);  }  **int** main() { **const size\_t** N = 10000000; **const int** T = std::**thread**::hardware\_concurrency() ? std::**thread**::hardware\_concurrency() : 4;  std::**vector**<**int**> data(N); std::mt19937 rng(42); std::uniform\_int\_distribution<**int**> dist(1, 100); **for** (**auto** &x : data) x = dist(rng);  *// Baseline (single-threaded)*  **auto** t0 = std::chrono::high\_resolution\_clock::now();  **long long** baseline = std::accumulate(data.begin(), data.end(), 0LL); **auto** t1 = std::chrono::high\_resolution\_clock::now();  *// Parallel*  std::**vector**<**long long**> partials(T, 0); std::**vector**<std::**thread**> threads; threads.reserve(T);  **size\_t** chunk = N / T;  **auto** p0 = std::chrono::high\_resolution\_clock::now(); **for** (**int** i = 0; i < T; ++i) {  **size\_t** s = i \* chunk; **size\_t** e = (i == T - 1) ? N : s + chunk; |
| threads.emplace\_back(partial\_sum, std::cref(data), s, e, std::ref( partials[i]));  }  **for** (**auto** &th : threads) th.join(); **long long** total = std::accumulate(partials.begin(), partials.end(), 0LL)  ;  **auto** p1 = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<**double**> t\_base = t1 - t0; std::chrono::duration<**double**> t\_par = p1 - p0;  std::cout << "Baseline sum: " << baseline  << " Time: " << t\_base.count() << " s\n";  std::cout << "Parallel sum: " << total  << " Time: " << t\_par.count() << " s\n";  **return** 0;  } |

p2.slurm

*#!/usr/bin/env zsh*

*#SBATCH --partition=instruction*

*#SBATCH --time=00:02:00*

*#SBATCH --ntasks=1*

*#SBATCH --cpus-per-task=4*

*#SBATCH --output=parallel\_sums.output* **cd** $SLURM\_SUBMIT\_DIR g++ -O2 -std=c++17 parallel\_sum.cpp -o parallel\_sum -pthread

./parallel\_sum

# Problem 3 — Race Condition Demonstration

**Task.** Have *T* threads increment a shared counter 100*,*000 times each. First run *without* synchronization (expect wrong result), then fix using (1) a mutex and (2) an atomic. Compare results and timing.

**Solution** race\_conditions.cpp

|  |
| --- |
| **#include** <iostream>  **#include** <**thread**>  **#include** <**vector**>  **#include** <**mutex**>  **#include** <**atomic**>  **#include** <chrono>  **constexpr int** ITER = 100000;  **void** inc\_no\_lock(**int** &counter) {  **for** (**int** i = 0; i < ITER; ++i) counter++; *// data race!*  }  **void** inc\_with\_mutex(**int** &counter, std::**mutex** &m) {  **for** (**int** i = 0; i < ITER; ++i) {  std::lock\_guard<std::**mutex**> lk(m);  ++counter;  }  }  **void** inc\_atomic(std::**atomic**<**int**> &counter) {  **for** (**int** i = 0; i < ITER; ++i) counter.fetch\_add(1, std:: memory\_order\_relaxed);  }  **template** <**typename** F> **int** run\_and\_time(**int** T, F &&fn) { **auto** t0 = std::chrono::high\_resolution\_clock::now(); std::**vector**<std::**thread**> ths; ths.reserve(T); **for** (**int** i = 0; i < T; ++i) ths.emplace\_back(fn); **for** (**auto** &t : ths) t.join();  **auto** t1 = std::chrono::high\_resolution\_clock::now(); **return** std::chrono::duration<**double**, std::milli>(t1 - t0).count();  }  **int** main() { **const int** T = std::**thread**::hardware\_concurrency() ?  std::**thread**::hardware\_concurrency() : 4;  **const int** expected = T \* ITER;  { *// No lock (incorrect)* |
| **int** counter = 0; **auto** ms = run\_and\_time(T, [&]{ inc\_no\_lock(counter); }); std::cout << "[No lock] counter=" << counter << " (expected " << expected << "), "  << ms << " ms\n";  }  { *// Mutex* **int** counter = 0; std::**mutex** m; **auto** ms = run\_and\_time(T, [&]{ inc\_with\_mutex(counter, m); }); std::cout << "[Mutex] counter=" << counter << " (expected " << expected << "), "  << ms << " ms\n";  }  { *// Atomic* std::**atomic**<**int**> counter{0}; **auto** ms = run\_and\_time(T, [&]{ inc\_atomic(counter); }); std::cout << "[Atomic] counter=" << counter.load()  << " (expected " << expected << "), "  << ms << " ms\n";  }  **return** 0;  } |

p3.slurm

*#!/usr/bin/env zsh*

*#SBATCH --partition=instruction*

*#SBATCH --time=00:02:00*

*#SBATCH --ntasks=1*

*#SBATCH --cpus-per-task=4*

*#SBATCH --output=race\_conditions.output* **cd** $SLURM\_SUBMIT\_DIR g++ -O2 -std=c++17 race\_conditions.cpp -o race\_conditions -pthread

./race\_conditions

# Problem 4 — Producer–Consumer with Condition Variables

**Task.** Implement one producer and one consumer sharing a bounded queue. The producer pushes integers 0..99. The consumer pops and processes them. Use a std::condition\_variable; the producer waits when the queue is full, and the consumer waits when empty. Cleanly terminate.

**Solution** producer\_consumer.cpp

|  |
| --- |
| **#include** <iostream>  **#include** <queue>  **#include** <**thread**>  **#include** <**mutex**>  **#include** <**condition\_variable**>  **constexpr int** MAX\_ITEMS = 10; std::queue<**int**> q; std::**mutex** m; std::**condition\_variable** cv; **bool** done = **false**;  **void** producer() {  **for** (**int** i = 0; i < 100; ++i) {  std::unique\_lock<std::**mutex**> lk(m); cv.wait(lk, []{ **return** (**int**)q.size() < MAX\_ITEMS; }); q.push(i); std::cout << "Produced: " << i << "\n"; lk.unlock(); cv.notify\_all();  }  { std::lock\_guard<std::**mutex**> lk(m); done = **true**;  } cv.notify\_all();  }  **void** consumer() { **while** (**true**) { std::unique\_lock<std::**mutex**> lk(m); cv.wait(lk, []{ **return** !q.empty() || done; }); **if** (q.empty() && done) **break**; **int** item = q.front(); q.pop(); lk.unlock();  std::cout << "Consumed: " << item << "\n"; cv.notify\_all();  }  }  **int** main() { std::**thread** p(producer); |
| std::**thread** c(consumer); p.join();  c.join(); **return** 0;  } |

p4.slurm

*#!/usr/bin/env zsh*

*#SBATCH --partition=instruction*

*#SBATCH --time=00:02:00*

*#SBATCH --ntasks=1*

*#SBATCH --cpus-per-task=4*

*#SBATCH --output=producer\_consumer.output* **cd** $SLURM\_SUBMIT\_DIR g++ -O2 -std=c++17 producer\_consumer.cpp -o producer\_consumer -pthread

./producer\_consumer

# Problem 5 — Parallel Matrix Multiplication

**Task.** Given two square matrices *A* and *B* of size *N × N* (e.g., *N* = 800), compute *C* = *A × B* in parallel.

* Split rows of *C* among threads.
* Measure execution time and compare with single-threaded.
* Use std::vector<double> in row-major layout.

**Solution** parallel\_matmul.cpp

|  |
| --- |
| **#include** <iostream>  **#include** <**vector**>  **#include** <**thread**>  **#include** <random>  **#include** <chrono>  **void** multiply\_block(**const** std::**vector**<**double**> &A,  **const** std::**vector**<**double**> &B, std::**vector**<**double**> &C, **int** N, **int** row\_start, **int** row\_end) {  **for** (**int** i = row\_start; i < row\_end; ++i) { **for** (**int** j = 0; j < N; ++j) { **double** sum = 0.0; **for** (**int** k = 0; k < N; ++k) { sum += A[i\*N + k] \* B[k\*N + j];  }  C[i\*N + j] = sum;  }  }  }  **int** main() { **const int** N = 800; **const int** T = std::**thread**::hardware\_concurrency() ? std::**thread**::hardware\_concurrency() : 4;  std::**vector**<**double**> A(N\*N), B(N\*N), C(N\*N); std::mt19937 rng(42); std::uniform\_real\_distribution<**double**> dist(0.0, 1.0);  **for** (**auto** &x : A) x = dist(rng); **for** (**auto** &x : B) x = dist(rng);  std::**vector**<std::**thread**> threads; **int** chunk = N / T;  **auto** start\_time = std::chrono::high\_resolution\_clock::now(); **for** (**int** t = 0; t < T; ++t) { **int** rs = t \* chunk; **int** re = (t == T-1) ? N : rs + chunk; |
| threads.emplace\_back(multiply\_block,  std::cref(A), std::cref(B), std::ref(C), N, rs, re);  }  **for** (**auto** &th : threads) th.join(); **auto** end\_time = std::chrono::high\_resolution\_clock::now();  std::cout << "Parallel multiplication took "  << std::chrono::duration<**double**>(end\_time - start\_time).  count()  << " s\n";  } |

p5.slurm

*#!/usr/bin/env zsh*

*#SBATCH --partition=instruction*

*#SBATCH --time=00:02:00*

*#SBATCH --ntasks=1*

*#SBATCH --cpus-per-task=4*

*#SBATCH --output=parallel\_matmul.output* **cd** $SLURM\_SUBMIT\_DIR g++ -O2 -std=c++17 parallel\_matmul.cpp -o parallel\_matmul -pthread

./parallel\_matmul

# Problem 6

Describe the challenges you encounter when completing this lab assignment and how you overcome these challenges.

**Challenges:**

* SLURM scripts failed with "Invalid --time specification" - had spaces around equals signs. Fixed by removing spaces.
* Git repo was broken on Euler. Just deleted and re-cloned from GitHub.
* Matrix multiplication used 256 threads (hardware\_concurrency) which killed performance. Fixed to 4 threads.

**What I learned about expected outputs:**

* Garbled output in hello\_threads is normal - shows race conditions on stdout
* Getting "correct" race condition results can be misleading - might just be lucky timing
* Too many threads hurts performance due to overhead
* Parallel programming behavior is counterintuitive - things that look broken might be correct demonstrations

Basically learned that parallel code doesn't behave like sequential code and debugging requires understanding what the program is supposed to show, not just whether output looks clean.