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:

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`

```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`

```bash
#!/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 10^7 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`

```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);
```

```cpp
  std::thread c(consumer);
  p.join();
  c.join();
  return 0;
}
```

p4.slurm

```bash
#!/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 $N \times N$ (e.g., $N = 800$), compute $C = A \times B$ 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

```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;
```

```cpp
        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.