Звіт

Лабораторна робота №6

студента групи ТТП-32

Ходакова Максима

**Умова**

Використати механізми ОС для розпаралелення обчислень та організації взаємодії з критичною секцією.  
   
1. Помножити матриці A[n x m] та B[m x k] (автоматично заповнені). Для обчислення створити 1..n\*k однотипних потоків для обчислень (множення векторів або сукупності векторів).  
1.1. (2 бала) Продемонструвати паралелізм (непослідовність) обчислень через виведення результату (трійками [x,y]=result) “по ходу обчислень”.  
1.2.\* (+3 бала) Дослідити швидкодію A\*B залежно від кількості потоків для розпаралелення множення. Продемонструвати та пояснити цю залежність. За якої кількості потоків множення буде найшвидшим? Підтвердити експериментально.  
   
2. Змоделювати паралельну роботу двох потоків (threads) зі спільною коміркою пам’яті (shared variable):  
а) з використанням критичного сегменту (або atomic, mutex, lock, і т.п.)  
б) без використання критичного сегменту.   
2.1. (2 бала) Продемонструвати різницю результатів обчислень у цих двох випадках. (*Наприклад*, збільшувати значення спільної комірки на 1: v=v+1; 10^9 разів в кожному потоці і дивитись результуюче значення v.)  
2.2. (+1 бал) Проаналізувати часову різницю різних варіантів реалізації та пояснити, чому іноді можливе отримання некоректного кінцевого результату (race condition).  
2.3. (+1 бал) Спробувати досягти якомога швидшого результату при збереженні коректності обчислень (правильного фінального значення).  
2.3.\* (+3 балів) Досягти варіанту, коли таке паралельне додавання виконується повністю синхронно, тобто, *наприклад*, 1000 додавань виконуються паралельно двома потоками крок-в-крок і збільшують значення спільної змінної від 0 до 1000. Тобто, не тільки кожний з двох паралельних потоків збільшує значення від 0 до 1000, а й обидва, запущені в паралель, також збільшують від 0 до 1000 (а не до 2000, як очікувалось би).

**Завдання 1**

**Реалізація**

**Код без статистики (фокус на обчисленнях пункти 1- 1.1)**

#include **<iostream>**#include **<vector>**#include **<mutex>**#include **<fstream>**#include **<thread>  
  
  
namespace** {  
 **constexpr** size\_t kMaxElement{100};  
 **template**<**typename** T>  
 **using** Matrix = std::vector<std::vector<T>>;  
  
 std::mutex cout\_mtx{};  
  
 Matrix<**int**> generate\_rand\_matrix(size\_t rows, size\_t columns) {  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 el = std::rand() % kMaxElement;  
 }  
 }  
 **return** res;  
 }  
  
 **void** print\_matrix(**const** Matrix<**int**> &matrix) {  
 **for** (**const auto** &row: matrix) {  
 **for** (**const auto** &el: row) {  
 std::lock\_guard lck{cout\_mtx};  
 std::cout << el << **" "**;  
 }  
 std::cout << **"\n"**;  
 }  
 }  
  
 Matrix<**int**> read\_matrix\_from\_file(**const** std::string &path, size\_t rows, size\_t columns) {  
 std::ifstream fin(path);  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **int** tmp{};  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 fin >> tmp;  
 el = tmp;  
 }  
 }  
 **return** res;  
 }  
  
 **void** compute\_el(**int** &res, **const** Matrix<**int**> &a, **const** Matrix<**int**> &b, size\_t i, size\_t j) {  
 **if** (a[i].size() != b.size()) {  
 exit(-2);  
 }  
 res = 0;  
 size\_t m = a[i].size();  
 **for** (size\_t cntr = 0; cntr < m; ++cntr) {  
 res += a[i][cntr] \* b[cntr][j];  
 }  
  
 {  
 std::lock\_guard lck{cout\_mtx};  
 std::cout << **"["** << i << **","** << j << **"] = "** << res << **"\n"**;  
 }  
 }  
  
}  
  
  
*// ./a.out n m k isRand pathA pathB***int** main(**int** argc, **char** \*\*argv) {  
  
 **if** (!(argc == 5 || (argc == 7 && std::stoi(argv[4]) == 0))) {  
 **return** -1;  
 }  
 Matrix<**int**> first;  
 Matrix<**int**> second;  
  
 size\_t n = std::stoul(argv[1]);  
 size\_t m = std::stoul(argv[2]);  
 size\_t k = std::stoul(argv[3]);  
  
 **if** (argc == 5) {  
 first = generate\_rand\_matrix(n, m);  
 second = generate\_rand\_matrix(m, k);  
 }  
  
 **if** (argc == 7) {  
 first = read\_matrix\_from\_file(argv[5], n, m);  
 second = read\_matrix\_from\_file(argv[6], m, k);  
 }  
  
  
 print\_matrix(first);  
 print\_matrix(second);  
  
 std::vector<std::thread> threads;  
  
 Matrix<**int**> res(n, std::vector<**int**>(k));  
  
 **for** (size\_t i = 0; i < n; ++i) {  
 **for** (size\_t j = 0; j < k; ++j) {  
 threads.emplace\_back(compute\_el, std::ref(res[i][j]), first, second, i, j);  
 }  
 }  
  
 **for** (**auto** &thrd: threads) {  
 thrd.join();  
 }  
  
 print\_matrix(res);  
  
 **return** 0;  
}

**A screenshot of a computer

Description automatically generated**

**Опис роботи**

Перевірка аргументів командного рядка: Програма перевіряє кількість аргументів, що передані їй. Якщо аргументи не відповідають очікуваному формату (або їх кількість), програма завершує роботу з кодом помилки -1. Читання або генерація матриць: У режимі генерації (argc == 5) програма створює дві матриці з випадковими числами за допомогою функції generate\_rand\_matrix. У режимі читання з файлу (argc == 7) матриці читаються з файлів, шлях до яких передається як аргументи командного рядка, за допомогою функції read\_matrix\_from\_file. Виведення матриць на екран: Перша і друга матриці виводяться на екран за допомогою функції print\_matrix. Обчислення елементів результуючої матриці: Для кожного елемента результуючої матриці створюється окремий потік, в якому виконується множення відповідного рядка на стовпець. Функція compute\_el обчислює значення одного елемента результуючої матриці, а також виводить інформацію про проміжний результат у форматі "[і,j] = результат". Синхронізація потоків: Основний потік чекає завершення всіх створених потоків за допомогою методу join. Виведення результуючої матриці: Після обчислення всіх елементів і завершення всіх потоків, результуюча матриця виводиться на екран.

**Код для статистики(пункт 1.2)**

#include **<iostream>**#include **<vector>**#include **<mutex>**#include **<fstream>**#include **<thread>**#include **<chrono>  
  
namespace** {  
 **constexpr** size\_t kMaxElement{100};  
 **template**<**typename** T>  
 **using** Matrix = std::vector<std::vector<T>>;  
  
 std::mutex cout\_mtx{};  
  
 Matrix<**int**> generate\_rand\_matrix(size\_t rows, size\_t columns) {  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 el = std::rand() % kMaxElement;  
 }  
 }  
 **return** res;  
 }  
  
 **void** print\_matrix(**const** Matrix<**int**> &matrix) {  
 std::lock\_guard<std::mutex> lck{cout\_mtx};  
 **for** (**const auto** &row: matrix) {  
 **for** (**const auto** &el: row) {  
 std::cout << el << **" "**;  
 }  
 std::cout << **"\n"**;  
 }  
 }  
  
 Matrix<**int**> read\_matrix\_from\_file(**const** std::string &path, size\_t rows, size\_t columns) {  
 std::ifstream fin(path);  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **int** tmp{};  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 fin >> tmp;  
 el = tmp;  
 }  
 }  
 **return** res;  
 }  
  
 **void** compute\_el(**int** &res, **const** Matrix<**int**> &a, **const** Matrix<**int**> &b, size\_t i, size\_t j) {  
 **if** (a[i].size() != b.size()) {  
 exit(-2);  
 }  
 res = 0;  
 size\_t m = a[i].size();  
 **for** (size\_t cntr = 0; cntr < m; ++cntr) {  
 res += a[i][cntr] \* b[cntr][j];  
 }  
 }  
}  
  
**int** main(**int** argc, **char** \*\*argv) {  
 **if** (!(argc == 6 || (argc == 8 && std::stoi(argv[4]) == 0))) {  
 std::cerr << **"Usage: "** << argv[0] << **" n m k isRand pathA pathB maxThreads\n"**;  
 **return** -1;  
 }  
  
 size\_t n = std::stoul(argv[1]);  
 size\_t m = std::stoul(argv[2]);  
 size\_t k = std::stoul(argv[3]);  
 size\_t maxThreads = std::stoul(argv[argc - 1]); *// New argument for max threads* Matrix<**int**> first, second;  
  
 **if** (std::stoi(argv[4]) == 1) {  
 first = generate\_rand\_matrix(n, m);  
 second = generate\_rand\_matrix(m, k);  
 } **else if** (argc == 8) {  
 first = read\_matrix\_from\_file(argv[5], n, m);  
 second = read\_matrix\_from\_file(argv[6], m, k);  
 }  
  
 **auto** start = std::chrono::high\_resolution\_clock::now();  
  
 std::vector<std::thread> threads;  
 Matrix<**int**> res(n, std::vector<**int**>(k));  
  
 size\_t threadsToUse = std::min(maxThreads, n \* k);  
  
 **for** (size\_t i = 0; i < n; ++i) {  
 **for** (size\_t j = 0; j < k; ++j) {  
 **if** (threads.size() < threadsToUse) {  
 threads.emplace\_back(compute\_el, std::ref(res[i][j]), std::cref(first), std::cref(second), i, j);  
 } **else** {  
 compute\_el(res[i][j], first, second, i, j); *// Compute directly if max threads reached* }  
 }  
 }  
  
 **for** (**auto** &thrd: threads) {  
 **if** (thrd.joinable()) {  
 thrd.join();  
 }  
 }  
  
 **auto** end = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<**double**, std::milli> duration = end - start;  
  
 std::cout << **"Computation with "** << threadsToUse << **" threads took "** << duration.count() << **" milliseconds.\n"**;  
  
 print\_matrix(res);  
  
 **return** 0;  
}

**Виконаємо команди для 4 потоків**

**g++ -o mainStat mainStat.cpp -std=c++17 -pthread**

**./mainStat 100 100 100 1 4 | tee output.txt**

**A screenshot of a computer screen

Description automatically generated**

**Опис роботи**

Перевірка валідності аргументів командного рядка: Програма перевіряє, чи правильна кількість аргументів була передана і чи вони відповідають очікуваному формату. Якщо аргументи неправильні, програма виводить повідомлення про помилку та завершується з кодом -1. Читання або генерація вхідних матриць: Якщо четвертий аргумент (isRand) дорівнює 1, програма генерує дві випадкові матриці з розмірами, вказаними в аргументах командного рядка. Якщо isRand дорівнює 0 та передано шляхи до файлів, матриці читаються з цих файлів. Множення матриць з використанням многопоточності: Програма створює потоки для обчислення елементів результуючої матриці, але кількість одночасно працюючих потоків обмежується максимальною кількістю, вказаною користувачем. Якщо кількість потрібних обчислень перевищує максимально дозволену кількість потоків, додаткові обчислення виконуються послідовно в основному потоці. Обробка та синхронізація потоків: Програма чекає завершення всіх створених потоків перед тим, як продовжити. Це гарантує, що вся обробка даних завершена перед виведенням результату. Вимірювання часу виконання: Програма фіксує час початку та кінця обчислень, обчислюючи загальну тривалість виконання в мілісекундах, що дозволяє оцінити ефективність многопоточності. Виведення результату: Після завершення обчислень програма виводить тривалість виконання та виводить результуючу матрицю на екран.

**Завдання 2**

**Код програми для (2-2.3)**

#include **<iostream>**#include **<vector>**#include **<mutex>**#include **<fstream>**#include **<thread>  
  
  
namespace** {  
 std::mutex shared\_var\_mutex;  
 **long long** shared\_var = 0;  
  
 **void** safe\_increment() {  
 **for** (**int** i = 0; i < 1000000000; ++i) {  
 std::lock\_guard<std::mutex> lock(shared\_var\_mutex);  
 ++shared\_var;  
 }  
 }  
  
 **long long** unsafe\_shared\_var = 0;  
  
 **void** unsafe\_increment() {  
 **for** (**int** i = 0; i < 1000000000; ++i) {  
 ++unsafe\_shared\_var;  
 }  
 }  
  
  
 **constexpr** size\_t kMaxElement{100};  
 **template**<**typename** T>  
 **using** Matrix = std::vector<std::vector<T>>;  
  
 std::mutex cout\_mtx{};  
  
 Matrix<**int**> generate\_rand\_matrix(size\_t rows, size\_t columns) {  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 el = std::rand() % kMaxElement;  
 }  
 }  
 **return** res;  
 }  
  
 **void** print\_matrix(**const** Matrix<**int**> &matrix) {  
 **for** (**const auto** &row: matrix) {  
 **for** (**const auto** &el: row) {  
 std::lock\_guard lck{cout\_mtx};  
 std::cout << el << **" "**;  
 }  
 std::cout << **"\n"**;  
 }  
 }  
  
 Matrix<**int**> read\_matrix\_from\_file(**const** std::string &path, size\_t rows, size\_t columns) {  
 std::ifstream fin(path);  
 Matrix<**int**> res(rows, std::vector<**int**>(columns));  
 **int** tmp{};  
 **for** (**auto** &row: res) {  
 **for** (**auto** &el: row) {  
 fin >> tmp;  
 el = tmp;  
 }  
 }  
 **return** res;  
 }  
  
 **void** compute\_el(**int** &res, **const** Matrix<**int**> &a, **const** Matrix<**int**> &b, size\_t i, size\_t j) {  
 **if** (a[i].size() != b.size()) {  
 exit(-2);  
 }  
 res = 0;  
 size\_t m = a[i].size();  
 **for** (size\_t cntr = 0; cntr < m; ++cntr) {  
 res += a[i][cntr] \* b[cntr][j];  
 }  
  
 {  
 std::lock\_guard lck{cout\_mtx};  
 std::cout << **"["** << i << **","** << j << **"] = "** << res << **"\n"**;  
 }  
 }  
  
}  
  
  
*// ./a.out n m k isRand pathA pathB***int** main(**int** argc, **char** \*\*argv) {  
  
 **if** (!(argc == 5 || (argc == 7 && std::stoi(argv[4]) == 0))) {  
 **return** -1;  
 }  
 Matrix<**int**> first;  
 Matrix<**int**> second;  
  
 size\_t n = std::stoul(argv[1]);  
 size\_t m = std::stoul(argv[2]);  
 size\_t k = std::stoul(argv[3]);  
  
 **if** (argc == 5) {  
 first = generate\_rand\_matrix(n, m);  
 second = generate\_rand\_matrix(m, k);  
 }  
  
 **if** (argc == 7) {  
 first = read\_matrix\_from\_file(argv[5], n, m);  
 second = read\_matrix\_from\_file(argv[6], m, k);  
 }  
  
  
 *// Демонстрація з мьютексом* **auto** start = std::chrono::high\_resolution\_clock::now();  
 std::thread t1(safe\_increment);  
 std::thread t2(safe\_increment);  
 t1.join();  
 t2.join();  
 **auto** end = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<**double**> diff = end - start;  
 std::cout << **"Safe increment result: "** << shared\_var << **", Time: "** << diff.count() << **" s\n"**;  
  
 *// Скидання спільної змінної* shared\_var = 0;  
  
 *// Демонстрація без мьютексу* start = std::chrono::high\_resolution\_clock::now();  
 std::thread t3(unsafe\_increment);  
 std::thread t4(unsafe\_increment);  
 t3.join();  
 t4.join();  
 end = std::chrono::high\_resolution\_clock::now();  
 diff = end - start;  
 std::cout << **"Unsafe increment result: "** << unsafe\_shared\_var << **", Time: "** << diff.count() << **" s\n"**;  
  
 **return** 0;  
}

**A screenshot of a computer

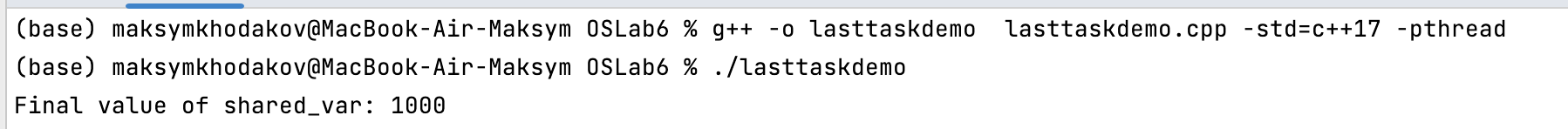
Description automatically generated**

2.2. Аналіз часової різниці та ризику race condition Часова різниця Використання мьютекса (безпечний інкремент) додає додаткову накладну витрату на операції через необхідність блокування та розблокування мьютекса кожен раз при зміні змінної. Це забезпечує взаємне виключення і запобігає одночасному доступу до змінної з боку кількох потоків, але робить виконання повільнішим порівняно з безпечним інкрементом, де така синхронізація відсутня. У варіанті без мьютекса (небезпечний інкремент) операції інкременту виконуються швидше, але при цьому виникає ризик race condition, коли кілька потоків одночасно читають та змінюють змінну, що може призвести до втрати деяких інкрементів і неправильного кінцевого результату. Race condition Race condition виникає у небезпечному інкременті, коли два або більше потоків конкурують за зміну спільної змінної без належної синхронізації. Це означає, що потоки можуть "переплутати" свої операції, результатом чого буде некоректний кінцевий результат. Наприклад, якщо два потоки одночасно прочитають значення змінної як n, обидва інкрементують його до n+1 і запишуть назад, замість очікуваного n+2 ви отримаєте n+1.

2.3. Покращення швидкості з забезпеченням коректності Щоб досягти швидкості з одночасним забезпеченням коректності обчислень, можна використовувати атомарні операції замість мьютекса для інкременту. Атомарні операції забезпечують взаємне виключення на рівні окремих операцій і є більш ефективними за використання мьютексів, коли мова йде про прості операції, такі як інкремент або декремент.

Код для демонстрації завдання 2.3\*

#include **<iostream>**#include **<vector>**#include **<mutex>**#include **<fstream>**#include **<thread>**#include **<condition\_variable>  
  
namespace** {  
 std::mutex shared\_var\_mutex;  
 std::condition\_variable cv;  
 **bool** turn = **false**; *// false for t1's turn, true for t2's turn* **long long** shared\_var = 0;  
  
 **void** safe\_increment(**bool** myTurn) {  
 **for** (**int** i = 0; i < 500; ++i) { *// Each thread only needs to increment 500 times to reach 1000 together* std::unique\_lock<std::mutex> lock(shared\_var\_mutex);  
 cv.wait(lock, [myTurn]{ **return** turn == myTurn; }); *// Wait for my turn* ++shared\_var;  
 turn = !turn; *// Toggle turn* cv.notify\_one(); *// Notify the other thread* }  
 }  
}  
  
**int** main() {  
 std::thread t1(safe\_increment, **false**); *// Thread 1 starts first* std::thread t2(safe\_increment, **true**); *// Thread 2 waits for its turn* t1.join();  
 t2.join();  
  
 std::cout << **"Final value of shared\_var: "** << shared\_var << std::endl;  
  
 **return** 0;  
}



**Висновок**

Було реалізовано програми згідно пунктів з вимогами. GitHub: [лінк](https://github.com/maksymkhodakov/OSLab6).