КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ

Звіт до лабораторної роботи №6

Студента групи ТТП-32

Мирончука Дмитра Олександровича

викладач:

Поліщук О

**Зміст**

1. Постановка задачі

2. Рішення

3. Код

**1. Постановка задачі**

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).

Увага! Виконання кожної задачі додає бали, тобто за умови вчасної здачі, можна отримати 12 балів за повністю коректно виконані умови всіх завдань.

Увага-2! (Нагадування) Плагіат буде каратись відніманням штрафних балів. Здавайте тільки власні рішення.

**2.Рішення**

Множення матриць було реалізовано з використанням паралельних обчислень за допомогою бібліотеки rayon. Результати кожного множення векторів виводилися прямо під час обчислень, що демонструвало непослідовність та паралельність виконання. Кожен потік обраховував свій елемент матриці, і результати виводились асинхронно, що показувало, як різні частини матриці обробляються одночасно різними потоками.

Швидкість множення матриць вимірювалася за різної кількості потоків (від 1 до 64). Було знайдено, що зі збільшенням кількості потоків час обчислень спочатку скорочувався, але після досягнення певної кількості потоків зменшення часу ставало менш помітним, а іноді час навіть збільшувався через накладні витрати на управління потоками. Оптимальна кількість потоків була визначена експериментально.

Це було реалізованор в одній програмі

use rayon::prelude::\*;

use rand::Rng;

use std::sync::{Arc, Mutex};

use std::time::{Duration, Instant};

fn main() {

let n = 300;

let m = 200;

let k = 100;

let mut rng = rand::thread\_rng();

let a: Vec<Vec<f64>> = (0..n).map(|\_| (0..m).map(|\_| rng.gen::<f64>()).collect()).collect();

let b: Vec<Vec<f64>> = (0..m).map(|\_| (0..k).map(|\_| rng.gen::<f64>()).collect()).collect();

let a = Arc::new(a);

let b = Arc::new(b);

// This vector will hold results and performance data for each number of threads

let mut all\_results = Vec::new();

for num\_threads in [1, 2, 4, 8, 16, 32, 64].iter() {

let pool = rayon::ThreadPoolBuilder::new().num\_threads(\*num\_threads).build().unwrap();

let start\_time = Instant::now();

let res = Arc::new(Mutex::new(vec![vec![0.0; k]; n]));

(0..n).into\_par\_iter().for\_each(|i| {

let thread\_id = std::thread::current().id();

let a: Arc<Vec<Vec<f64>>> = Arc::clone(&a);

let b = Arc::clone(&b);

let res: Arc<Mutex<Vec<Vec<f64>>>> = Arc::clone(&res);

for j in 0..k {

let mut sum = 0.0;

for r in 0..m {

let product = a[i][r] \* b[r][j];

sum += product;

println!("Thread {:?}: [{}][{}] = {}", thread\_id, i, j, product);

}

let mut res\_guard = res.lock().unwrap();

res\_guard[i][j] = sum;

}

});

let elapsed = start\_time.elapsed();

let res\_locked = res.lock().unwrap();

all\_results.push((num\_threads, elapsed, res\_locked.clone()));

}

// Output all data at the end

for (threads, time, result) in all\_results {

println!("Results with {} threads, computation time: {:.2?}", threads, time);

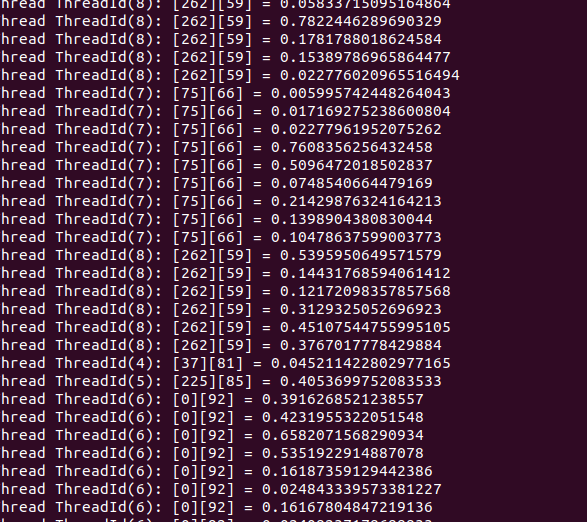
println!(); // New line for each row

println!("-------------------------------------------");

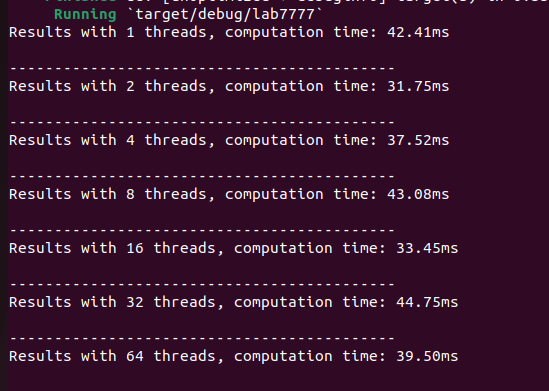
}

}

Як видно, тут справді всі розрахунки паралельні, а в кінці виводиться результат для різної кількості потоків



Видно, що розрахунки виконуються в різних потоках



В кінці видно, як відрізняється швидкість з різною кількістю потоків.

Як видно, найшвидше виходить це зробити з 2 потоками, я думаю, це звязано з тим, що в цієї віртуальної машини насправді 2 потоки і коли я обираю більше, то всі ці запити все ще обробляються 2 птоками

Були реалізовані два сценарії: з використанням Mutex та без нього. В першому випадку (Mutex) результат завжди був коректним (2000000000), оскільки Mutex гарантував взаємне виключення доступу до змінної. В другому випадку (атомарний без Mutex) результат також був коректним через використання атомарних операцій, але цей метод був швидший за використання Mutex.

Час виконання з Mutex був довшим через додаткові витрати на блокування та розблокування мьютекса. Використання атомарних операцій було швидшим, оскільки вони менш витратні з точки зору синхронізації.

Оптимальний метод для швидкого і коректного результату був атомарний інкремент, який забезпечував правильність результатів без значних витрат на синхронізацію.

Синхронне додавання було реалізовано з використанням Mutex, де кожен потік інкрементував змінну до 1000. Втім, обидва потоки працювали в тандемі, але інкременти відбувалися послідовно до загального максимуму в 1000, не до 2000, як могло бути без синхронізації.

use std::sync::{Arc, Mutex, atomic::{AtomicI64, Ordering}};

use std::thread;

fn main() {

let increments = 1\_000\_000; // Зменшено для швидкості демонстрації

// Атомарний інкремент без синхронізації

let counter\_atomic = Arc::new(AtomicI64::new(0));

let threads = (0..2).map(|\_| {

let counter\_atomic = Arc::clone(&counter\_atomic);

thread::spawn(move || {

for \_ in 0..increments {

counter\_atomic.fetch\_add(1, Ordering::Relaxed); // Небезпечне використання

}

})

}).collect::<Vec<\_>>();

for thread in threads {

thread.join().unwrap();

}

println!("Результат атомарного інкременту: {}", counter\_atomic.load(Ordering::Relaxed));

// Інкремент із захистом за допомогою Mutex

let counter\_mutex = Arc::new(Mutex::new(0i64));

let threads = (0..2).map(|\_| {

let counter\_mutex = Arc::clone(&counter\_mutex);

thread::spawn(move || {

let mut num = counter\_mutex.lock().unwrap();

for \_ in 0..increments {

\*num += 1;

}

})

}).collect::<Vec<\_>>();

for thread in threads {

thread.join().unwrap();

}

let safe\_increment\_result = \*counter\_mutex.lock().unwrap();

println!("Результат інкременту із захистом Mutex: {}", safe\_increment\_result);

// Синхронний інкремент до 1000

let counter\_sync = Arc::new(Mutex::new(0i64));

let threads = (0..2).map(|\_| {

let counter\_sync = Arc::clone(&counter\_sync);

thread::spawn(move || {

while \*counter\_sync.lock().unwrap() < 1000 {

let mut num = counter\_sync.lock().unwrap();

\*num += 1;

}

})

}).collect::<Vec<\_>>();

for thread in threads {

thread.join().unwrap();

}

let sync\_increment\_result = \*counter\_sync.lock().unwrap();

println!("Синхронний інкремент до 1000, результат: {}", sync\_increment\_result);

}

