

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
 НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу  
Кафедра системного проектування**

**Звіт**

**про виконання практичної роботи №2  
з дисципліни «Паралельні обчислення»**

Виконав:  
студент III курсу, групи ДА-22  
Жадько Микита Сергійович

Прийняв:

асистент Яременко В. С.

Київ – 2025

**ПОСТАНОВКА ЗАДАЧІ**

1. Ознайомитися з визначенням: атомарна змінна, атомарна операція, неблокуючий алгоритм. Ознайомитися з деталями атомарності в обраній мові програмування.
2. Надати в протоколі роботи опис того, як саме досягається справжня атомарність операцій в обраній студентом мові програмування.
3. Виконати завдання за варіантом без використанням паралелізації. Заміряти час виконання завдання.
4. Виконати завдання за варіантом з використанням блокуючих примітивів синхронізації. Заміряти час виконання завдання.
5. Виконати завдання за варіантом з використанням атомарних змінних та CAS\CMPXCHNG операцій (більш високорівнені функції, що абстрагують дані операції, не приймаються для використання в даній роботі). Заміряти час виконання завдання.
6. Повторити пункти 2 – 4 з використанням різної розмірності даних та фіксованою кількістю потоків виконання.
7. В протокол занести отримані для пункту 5 результати у вигляді графіків залежності часу від кількості даних, надати порівняльний аналіз блокуючого та неблокуючого алгоритму з використання атомарних операцій.
8. Надати висновок, що повинен містити аналіз отриманих результатів.

Завдання:

21. Знайти суму трьох мінімальних елементів, і самі елементи.

**ХІД РОБОТИ**

У мові C++ атомарність досягається за допомогою стандартної бібліотеки <atomic>, яка надає класи та функції для роботи з атомарними змінними. Основним засобом для цього є шаблонний клас std::atomic<T>, що дозволяє виконувати операції без блокувань, запобігаючи станам гонки у багатопотокових додатках

**Операції читання та запису**

Одними з найпростіших атомарних операцій є load() та store(value).

* **load()** — здійснює атомарне читання значення змінної, гарантуючи, що інші потоки не отримають неповних або частково змінених даних.
* **store(value)** — атомарно записує нове значення у змінну, забезпечуючи правильне оновлення без впливу на інші операції потоків.

**Операції обміну та порівняння**

Для ефективного управління ресурсами часто використовуються операції обміну значень та порівняння з обміном **(CAS, Compare-And-Swap)**.

* **exchange(value)** — атомарно змінює значення змінної, повертаючи її попереднє значення.
* **compare\_exchange\_weak(expected, desired)** та **compare\_exchange\_strong(expected, desired)** — реалізують механізм CAS, що дозволяє оновлювати значення змінної лише у разі, якщо воно дорівнює очікуваному значенню (expected). Якщо умова не виконується, операція повертає false, дозволяючи повторювати спробу оновлення у циклі.

Операції compare\_exchange\_weak і compare\_exchange\_strong відрізняються тим, що weak може іноді повертати false навіть при збігу значень (через спуріозні відмови), тоді як strong гарантує успішне виконання без зайвих повторних спроб.

**Операції модифікації значень**

Крім базових операцій, std::atomic підтримує набір атомарних арифметичних та побітових операцій, які можуть використовуватися для ефективного оновлення змінних у багатопотоковому середовищі:

* **fetch\_add(value)** — атомарно додає value до поточного значення змінної, повертаючи її попереднє значення.
* **fetch\_sub(value)** — аналогічно виконує віднімання.
* **fetch\_or(value), fetch\_and(value), fetch\_xor(value)** — виконують відповідні побітові операції (OR, AND, XOR).

Ці операції використовуються у ситуаціях, коли необхідно оновлювати спільний лічильник або здійснювати контроль доступу до ресурсів без блокування потоків.

**Моделі пам’яті та їх вплив на синхронізацію**

Атомарні операції у C++ можуть використовувати різні **моделі пам’яті (memory orderings)**, які визначають, як зміни у змінній впливають на послідовність виконання операцій у потоках:

* **memory\_order\_relaxed** — не накладає обмежень на порядок виконання операцій, що забезпечує максимальну продуктивність, але не гарантує узгодженості між потоками.
* **memory\_order\_acquire** — гарантує, що всі операції після цієї будуть виконані лише після завершення цієї операції. Використовується при читанні змінних, щоб забезпечити видимість попередніх записів.
* **memory\_order\_release** — гарантує, що всі операції перед цією будуть завершені до її виконання. Застосовується під час запису змінної, щоб забезпечити її коректне оновлення для інших потоків.
* **memory\_order\_seq\_cst** (за замовчуванням) — забезпечує **строгу послідовну консистентність**, тобто всі операції виконуються в єдиному впорядкованому порядку для всіх потоків.

Реалізуємо програму для знаходження трьох найменших елементів у масиві, використовуючи як послідовні, так і паралельні методи обчислення. Паралельне обчислення буде виконуватися за допомогою потоків, яким розподіляються частини масиву.

У паралельних версіях використовуватимемо два підходи до обробки даних:

1. **Блокуюча версія з м'ютексом** – кожен потік знаходить свої локальні мінімальні значення, після чого результати синхронізуються через м’ютекс при оновленні загальних змінних.
2. **Lock-free версія з атомарними змінними** – кожен потік паралельно обробляє свою частину масиву, а потім оновлює глобальні мінімальні значення, використовуючи атомарні операції.

Лістинг коду:

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <atomic>

#include <chrono>

#include <cstdlib>

#include <random>

#include <algorithm>

constexpr int BITS\_PER\_NUMBER = 21;

constexpr int64\_t MASK = (1LL << BITS\_PER\_NUMBER) - 1;

constexpr int MAX\_VAL = MASK;

int64\_t encodeTriple(int a, int b, int c) {

return (int64\_t(a) << (2 \* BITS\_PER\_NUMBER)) | (int64\_t(b) << BITS\_PER\_NUMBER) | c;

}

void decodeTriple(int64\_t encoded, int& a, int& b, int& c) {

a = encoded >> (2 \* BITS\_PER\_NUMBER);

b = (encoded >> BITS\_PER\_NUMBER) & MASK;

c = encoded & MASK;

}

int64\_t combineEncodedTriple(int64\_t encoded1, int64\_t encoded2) {

int a1, b1, c1, a2, b2, c2;

decodeTriple(encoded1, a1, b1, c1);

decodeTriple(encoded2, a2, b2, c2);

std::vector<int> combined = { a1, b1, c1, a2, b2, c2 };

std::sort(combined.begin(), combined.end());

return encodeTriple(combined[0], combined[1], combined[2]);

}

struct Triple {

int m1;

int m2;

int m3;

};

void printTriple(const std::string& label, const Triple& tr) {

std::cout << label << " minimal elements: "

<< tr.m1 << ", " << tr.m2 << ", " << tr.m3

<< " | sum = " << (tr.m1 + tr.m2 + tr.m3) << std::endl;

}

// 1. Sequential version

Triple findThreeMinSerial(const std::vector<int>& arr) {

Triple result{ MAX\_VAL, MAX\_VAL, MAX\_VAL };

for (int x : arr) {

if (x < result.m1) {

result.m3 = result.m2;

result.m2 = result.m1;

result.m1 = x;

}

else if (x < result.m2) {

result.m3 = result.m2;

result.m2 = x;

}

else if (x < result.m3) {

result.m3 = x;

}

}

return result;

}

// 2. Paralel Mutex Version

Triple findThreeMinBlocking(const std::vector<int>& arr, int numThreads) {

Triple global{ MAX\_VAL, MAX\_VAL, MAX\_VAL };

std::mutex mtx;

auto worker = [&](int start, int end) {

Triple local{ MAX\_VAL, MAX\_VAL, MAX\_VAL };

for (int i = start; i < end; i++) {

int x = arr[i];

if (x < local.m1) {

local.m3 = local.m2;

local.m2 = local.m1;

local.m1 = x;

}

else if (x < local.m2) {

local.m3 = local.m2;

local.m2 = x;

}

else if (x < local.m3) {

local.m3 = x;

}

}

std::lock\_guard<std::mutex> lock(mtx);

std::vector<int> combined = { global.m1, global.m2, global.m3, local.m1, local.m2, local.m3 };

std::sort(combined.begin(), combined.end());

global.m1 = combined[0];

global.m2 = combined[1];

global.m3 = combined[2];

};

std::vector<std::thread> threads;

int n = static\_cast<int>(arr.size());

int chunk = n / numThreads;

int start = 0;

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

int end = (i == numThreads - 1) ? n : start + chunk;

threads.emplace\_back(worker, start, end);

start = end;

}

for (auto& t : threads) {

t.join();

}

return global;

}

// 3. Paralel Atomic version

Triple findThreeMinNonBlocking(const std::vector<int>& arr, int numThreads) {

std::atomic<int64\_t> atomicGlobal;

atomicGlobal.store(encodeTriple(MAX\_VAL, MAX\_VAL, MAX\_VAL));

auto worker = [&](int start, int end) {

int local1 = MAX\_VAL, local2 = MAX\_VAL, local3 = MAX\_VAL;

for (int i = start; i < end; i++) {

int x = arr[i];

if (x < local1) {

local3 = local2;

local2 = local1;

local1 = x;

}

else if (x < local2) {

local3 = local2;

local2 = x;

}

else if (x < local3) {

local3 = x;

}

}

int64\_t localEncoded = encodeTriple(local1, local2, local3);

int64\_t oldVal = atomicGlobal.load();

while (true) {

int64\_t newVal = combineEncodedTriple(oldVal, localEncoded);

if (atomicGlobal.compare\_exchange\_weak(oldVal, newVal)) {

break;

}

}

};

std::vector<std::thread> threads;

int n = static\_cast<int>(arr.size());

int chunk = n / numThreads;

int start = 0;

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

int end = (i == numThreads - 1) ? n : start + chunk;

threads.emplace\_back(worker, start, end);

start = end;

}

for (auto& t : threads) {

t.join();

}

int64\_t finalEncoded = atomicGlobal.load();

int a, b, c;

decodeTriple(finalEncoded, a, b, c);

return Triple{ a, b, c };

}

int main() {

const int dataSize = 1000;

const int numThreads = 8;

// Filling array with normal distribution in range [0, MAX\_VAL)

std::random\_device rd;

std::mt19937 gen(rd());

std::normal\_distribution<double> dist(MAX\_VAL / 2.0, MAX\_VAL / 6.0);

std::vector<int> data(dataSize);

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

int val = static\_cast<int>(dist(gen));

data[i] = std::clamp(val, 0, MAX\_VAL);

}

// 1. Sequential version

auto startSerial = std::chrono::steady\_clock::now();

Triple resultSerial = findThreeMinSerial(data);

auto endSerial = std::chrono::steady\_clock::now();

auto timeSerial = std::chrono::duration\_cast<std::chrono::microseconds>(endSerial - startSerial).count();

// 2. Paralel Mutex Version

auto startBlocking = std::chrono::steady\_clock::now();

Triple resultBlocking = findThreeMinBlocking(data, numThreads);

auto endBlocking = std::chrono::steady\_clock::now();

auto timeBlocking = std::chrono::duration\_cast<std::chrono::microseconds>(endBlocking - startBlocking).count();

// 3. Paralel Atomic version

auto startNonBlocking = std::chrono::steady\_clock::now();

Triple resultNonBlocking = findThreeMinNonBlocking(data, numThreads);

auto endNonBlocking = std::chrono::steady\_clock::now();

auto timeNonBlocking = std::chrono::duration\_cast<std::chrono::microseconds>(endNonBlocking - startNonBlocking).count();

std::cout << "Sequential version:" << std::endl;

printTriple("Sequential", resultSerial);

std::cout << "Execution time: " << timeSerial << " microseconds" << std::endl << std::endl;

std::cout << "Paralel Mutex Version:" << std::endl;

printTriple("Lock", resultBlocking);

std::cout << "Execution time: " << timeBlocking << " microseconds" << std::endl << std::endl;

std::cout << "Paralel Atomic version:" << std::endl;

printTriple("Lock-free", resultNonBlocking);

std::cout << "Execution time: " << timeNonBlocking << " microseconds" << std::endl;

return 0;

}

Запустимо програму та занесемо результати роботи в окремі таблиці.



Рисунок 1 — Час роботи програми з масивом розмірності 1000.



Рисунок 2 — Час роботи програми з масивом розмірності 10000.



Рисунок 3 — Час роботи програми з масивом розмірності 100000.



Рисунок 4 — Час роботи програми з масивом розмірності 1000000.



Рисунок 5 — Час роботи програми з масивом розмірності 10000000.



Рисунок 6 — Час роботи програми з масивом розмірності 100000000.



Рисунок 7 — Час роботи програми з масивом розмірності 200000000.

Побудуємо графіки:

Рисунок 8 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 1 000.

Рисунок 9 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 10 000.

Рисунок 10 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 100 000.

Рисунок 11 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 1 000 000.

Рисунок 12 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 10 000 000.

Рисунок 13 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 100 000 000.

Рисунок 14 — Графік залежності часу роботи від кількості потоків для програми з масивом розмірності 200 000 000.

**ВИСНОВКИ**

В ході виконання лабораторної роботи було проведено порівняльний аналіз трьох підходів до розв’язання задачі знаходження суми трьох мінімальних елементів масиву: послідовного алгоритму, паралельного алгоритму з блокуючими примітивами (м’ютекс) та неблокуючого алгоритму з використанням атомарних операцій (CAS). Отримані результати демонструють, що при відсутності синхронізації (послідовна версія) час виконання залишається мінімальним незалежно від кількості потоків – наприклад, для масиву розмірності 1000 час становив лише 1 мікросекунду, а для масиву розмірності 10 000 – від 9 до 14 мікросекунд. Однак для масивів більших розмірів, час знаходження трьох мінімальних елементів суттєво зростає, наприклад, для масиву розмірності 100 000 000 час становив приблизно 200 000 мікросекунд.

Застосування блокуючих примітивів суттєво збільшує час виконання для масивів меншого розміру через накладні витрати на блокування та розблокування. Так, для масиву з 1000 елементів при використанні 1 потоку час склав 845 мікросекунд, а при збільшенні кількості потоків до 32 – вже 2605 мікросекунд. Аналогічна тенденція спостерігається і для більших масивів: для розмірності 100 000 результат варіював від 1187 мікросекунд (1 потік) до 3795 мікросекунд (32 потоки), що свідчить про зростання накладних витрат у разі високої конкуренції доступу до критичної секції. Однак для розмірів дуже великого розміру, ми можемо побачити досить непогані результати у порівнянні з послідовною версією – для масиву розміром 100 000 000 час становив 27 843 мікросекунди для версії з 16 потоками, більш ніж в 8 разів швидше за послідовну версію.

Неблокуючий підхід з використанням атомарних операцій дозволив знизити затримки, пов’язані із блокуванням. Наприклад, для масиву з 1000 елементів при 1 потоці час склав 273 мікросекунди, при 2 – 279 мікросекунд, а при 8 потоках – 1318 мікросекунд. Для розмірності 10 000 елементів час у варіанті з атомарними операціями коливався від 222 мікросекунд (1 потік) до 2500 мікросекунд (32 потоки). При великих обсягах даних переваги атомарних операцій особливо помітні: для масиву з 100 000 000 елементів при 8 потоках час виконання склав 15 513 мікросекунд порівняно з 19 728 мікросекундами для версії з м’ютексами, а для масиву з 200 000 000 елементів – 33 909 мікросекунд для атомарного підходу проти 39 564 мікросекунд для блокуючого у версії з 8 потоками.

Особливу увагу варто приділити проблемі використання атомарних операцій для роботи з кількома змінними. Як видно з коду, спроба використовувати окремі атомарні змінні для зберігання трьох мінімальних чисел не забезпечує їхньої спільної атомарності, тобто операції над ними можуть виконуватися незалежно одна від одної, що може призвести до некоректного оновлення загального стану. Тому було прийнято рішення кодувати три мінімальних числа в один 64-бітний тип (int64\_t), що дозволяє здійснювати операції оновлення над усією трійкою як над єдиним атомарним блоком за допомогою операції CAS. Такий підхід забезпечує цілісність даних, але водночас ускладнює реалізацію алгоритму.

Таким чином, результати дослідження свідчать про те, що для невеликих масивів послідовний алгоритм забезпечує найкращу продуктивність завдяки відсутності додаткових накладних витрат. Проте, із збільшенням розмірності даних та кількості потоків переваги паралельних рішень стають очевидними. Серед паралельних підходів, неблокуючий алгоритм з використанням атомарних операцій демонструє кращу масштабованість і нижчі загальні затримки в порівнянні з блокуючим варіантом, хоча потребує спеціальних рішень для забезпечення комплексної атомарності (як, наприклад, кодування трьох чисел в один int64\_t). Вибір оптимального механізму синхронізації повинен базуватися на конкретних вимогах задачі та характеристиках апаратного забезпечення, що дозволяє забезпечити баланс між мінімізацією накладних витрат та ефективним використанням ресурсів