**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**

**імені Ігоря Сікорського»**

**Лабораторна робота № 2**

**з курсу «Паралельні обчислення»**

**на тему «Дослідження атомарних змінних та атомарних операцій»**

**Виконав студент ІПСА групи ДА-22**

**Котляр Єгор Ярославович**

Київ – 2025

**Мета роботи**

Розглянути поняття атомарності, навчитися працювати з атомарними змінними, а також ознайомитися з підходом написання паралельного коду без блокування.

**Завдання**

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.

**Варіант 14**: Знайти суми по модулю два всіх елементів масиву кратних 7.

Посилання на [репозиторій](https://github.com/Kott1/PC).

**Теоретичні відомості**

**Атомарна змінна** – це змінна, над якою можна виконувати атомарні операції, тобто такі, що не можуть бути перервані іншим потоком під час виконання. У мові C++ атомарні змінні представлені стандартом std::atomic.

**Атомарна операція** – операція, яка виконується повністю або не виконується взагалі, без можливості втручання інших потоків. Наприклад, атомарне інкрементування змінної гарантує, що жоден потік не зможе побачити її проміжний стан.

**Неблокуючий алгоритм** – це алгоритм, який дозволяє кільком потокам отримувати доступ до спільних ресурсів без використання блокувань (наприклад м’ютексів). До таких алгоритмів належать lock-free та wait-free алгоритми. Вони мінімізують затримки та підвищують продуктивність у багатопотокових середовищах.

У мові C++ атомарність підтримується на рівні стандартної бібліотеки, починаючи з C++11. Для цього використовується шаблон std::atomic<T>, який забезпечує атомарні операції над об’єктами типу T.

Операції, що підтримуються std::atomic, включають:

* Завантаження та збереження – load(), store().
* Читання з оновленням – fetch\_add, fetch\_sub, exchange.
* Порівняння та обмін значенням – compare\_exchange\_weak, compare\_exchange\_strong.

Також можна вказувати модель пам’яті для контролю порядку видимості змін між потоками, наприклад: memory\_order\_relaxed, memory\_order\_acquire, memory\_order\_release).

У C++ справжня атомарність операцій досягається за рахунок:

* Використання апаратної підтримки процесора — стандартна бібліотека C++ реалізує атомарні операції за допомогою низькорівневих інструкцій, таких як LOCK CMPXCHG, XCHG, LL/SC, які підтримуються більшістю сучасних процесорів. Це дозволяє виконати операцію так, що вона є неділимою навіть на рівні процесора.
* Інтерфейсів з компілятором — компілятори реалізують атомарні операції через вбудовані функції (\_\_atomic, \_\_sync) або ж оптимізують використання std::atomic відповідно до архітектури.
* Шаблонного класу std::atomic — цей клас гарантує, що всі операції над об’єктом виконуються атомарно. Він також може використовуватись для побудови неблокуючих структур даних та алгоритмів.

**Хід виконання роботи**

За варіантом потрібно знайти суму по модулю два (% 2) всіх елементів, кратних 7.

У випадку, де ми не використовуємо паралелізацію – будемо просто обробляти масив чисел по одному числу.

Якщо використовуємо м’ютекси, тоді можна розділити масив на, наприклад, 2 потоки, де для кожного потоку буде своя частина масиву. Також, за допомогою м’ютексів буде гарантовано, що одночасних записів у змінну не буде.

При використання атомарних значень знову розділимо дані масиву для 2 потоків. У цьому випадку ми дозволяємо кільком потокам одночасно оновлювати змінну, тобто використовуємо операцію CAS (compare‑and‑swap).

**Код програми:**

#include <iostream>  
#include <windows.h>  
#include <vector>  
#include <thread>  
#include <mutex>  
#include <atomic>  
#include <chrono>  
#include <cstdlib>  
#include <ctime>  
  
using namespace std;  
using namespace chrono;  
  
vector<int> generate\_data(int size) {  
 vector<int> data;  
 for (int i = 0; i < size; ++i) {  
 data.push\_back(rand() % 1001);  
 }  
 return data;  
}  
  
int sequential(const vector<int>& data) {  
 int result = 0;  
 for (int val : data) {  
 if (val % 7 == 0) {  
 result ^= val;  
 }  
 }  
 return result;  
}  
  
void process\_mutex(const vector<int>& data, int start, int end, int& result, mutex& mtx) {  
 for (int i = start; i < end; ++i) {  
 if (data[i] % 7 == 0) {  
 lock\_guard<mutex> lock(mtx);  
 result ^= data[i];  
 }  
 }  
}  
  
int parallel\_mutex(const vector<int>& data) {  
 int result = 0;  
 mutex mtx;  
 int mid = data.size() / 2;  
  
 thread t1(process\_mutex, cref(data), 0, mid, ref(result), ref(mtx));  
 thread t2(process\_mutex, cref(data), mid, data.size(), ref(result), ref(mtx));  
 t1.join(); t2.join();  
  
 return result;  
}  
  
void process\_atomic(const vector<int>& data, int start, int end, atomic<int>& result) {  
 for (int i = start; i < end; ++i) {  
 if (data[i] % 7 == 0) {  
 int current = result.load(memory\_order\_relaxed);  
 while (!result.compare\_exchange\_weak(current, current ^ data[i], memory\_order\_relaxed));  
 }  
 }  
}  
  
int parallel\_atomic(const vector<int>& data) {  
 atomic<int> result(0);  
 int mid = data.size() / 2;  
  
 thread t1(process\_atomic, cref(data), 0, mid, ref(result));  
 thread t2(process\_atomic, cref(data), mid, data.size(), ref(result));  
 t1.join(); t2.join();  
  
 return result.load();  
}  
  
void test\_size(int size) {  
 cout << "\nРозмір масиву: " << size << " елементів\n";  
 vector<int> data = generate\_data(size);  
  
 auto start = high\_resolution\_clock::*now*();  
 int res1 = sequential(data);  
 auto end = high\_resolution\_clock::*now*();  
 cout << "Послідовно:\tXOR = " << res1  
 << ", час = " << duration<double>(end - start).count() << " с\n";  
  
 start = high\_resolution\_clock::*now*();  
 int res2 = parallel\_mutex(data);  
 end = high\_resolution\_clock::*now*();  
 cout << "З м'ютексом:\tXOR = " << res2  
 << ", час = " << duration<double>(end - start).count() << " с\n";  
  
 start = high\_resolution\_clock::*now*();  
 int res3 = parallel\_atomic(data);  
 end = high\_resolution\_clock::*now*();  
 cout << "З CAS:\t\tXOR = " << res3  
 << ", час = " << duration<double>(end - start).count() << " с\n";  
}  
  
int main() {  
 SetConsoleOutputCP(CP\_UTF8);  
 srand(time(0));  
  
 vector<int> sizes = {10000, 100000, 1000000, 10000000, 100000000};  
 for (int i = 0; i < sizes.size(); ++i) {  
 test\_size(sizes[i]);  
 }  
  
 return 0;  
}

Вхідні дані – це рандомні числа у діапазоні від 0 до 1000. Для тестування написаних алгоритмів використано масиви різного розміру (10000, 100000, 1000000, 10000000, 100000000) та заміряно час виконання кожного з рішень:

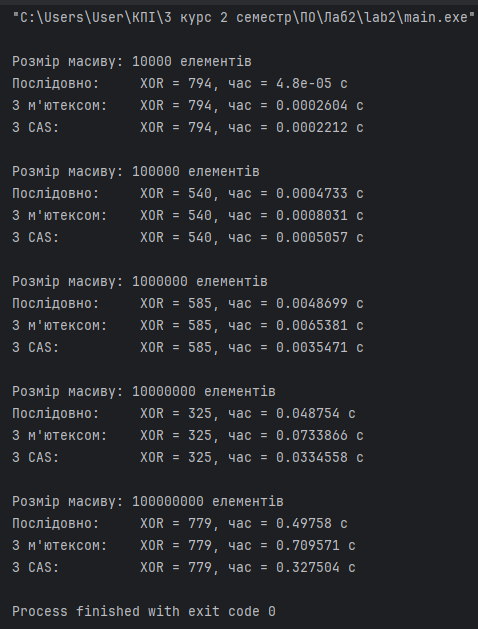


Рисунок 1 – Результат виконання програми.

Створимо графіки для кращого порівняння отриманого результату:

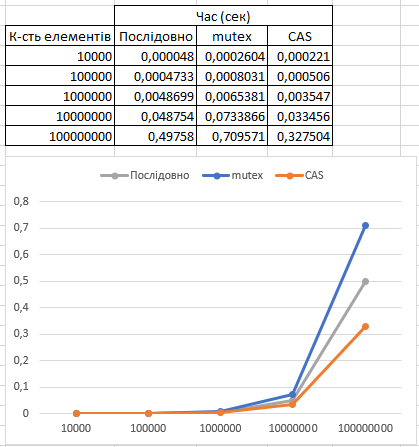


Рисунок 2 – Графік результату виконання програми.

**Висновок**

У результаті експериментів було виконано три підходи до обчислення XOR всіх елементів масиву, кратних 7: послідовний, паралельний із м’ютексом та паралельний із атомарними змінними на основі CAS. Для невеликих розмірів масиву (10 000–100 000 елементів) послідовний алгоритм виявився найшвидшим, оскільки витрати на створення потоків та синхронізацію переважали переваги розпаралелювання. Зі збільшенням обсягу даних (1 млн–10 млн елементів) алгоритм із CAS показав кращу продуктивність порівняно з варіантом із м’ютексом завдяки відсутності блокувань у критичних секціях. Для найбільшого масиву (100 млн елементів) неблокуючий підхід виявився майже вдвічі швидшим за послідовний і значно випередив м’ютексну реалізацію, що пояснюється зменшенням витрат на lock‑unlock операції і відносно невеликою кількістю повторних спроб у CAS-циклі. Таким чином, для великих обсягів даних використання атомарних змінних і CAS є більш ефективним, тоді як для малих обсягів простіший послідовний підхід залишається оптимальним.