

**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  
Байдюк Антон Олексійович

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.

Варіант 1: Знайти максимальний елемент масиву, а також кількість цих елементів.

**ХІД РОБОТИ**

Мета цієї лабораторної роботи — ознайомлення з поняттями **атомарних змінних**, **атомарних операцій** та **неблокуючих алгоритмів**, а також практична робота з атомарними операціями в обраній мові програмування (**C++**).

**Атомарність** — це властивість операції бути повністю виконаною або не виконаною взагалі без проміжних станів, видимих іншим потокам. Це гарантує цілісність даних при паралельній обробці. Іншими словами, атомарна операція або відбувається повністю, або не відбувається зовсім, що виключає можливість спотворення стану змінних у процесі роботи декількох потоків.

**Атомарні змінні** — це спеціальні типи даних, які дозволяють виконувати операції над собою без блокувань і ризику гонки даних. Вони підтримують атомарне зчитування, запис, інкремент, декремент та основні арифметичні операції. У C++ для роботи з атомарними змінними використовується стандартна бібліотека std::atomic, яка забезпечує простий доступ до апаратних механізмів атомарності.

**Неблокуючі алгоритми** — це алгоритми, які дозволяють потокам продовжувати роботу без необхідності очікування блокування ресурсів. Основою неблокуючих алгоритмів є атомарні операції, зокрема **CAS (Compare-And-Swap)**. Вони підвищують продуктивність програм і зменшують ризик появи "взаємного блокування" (deadlock) у багатопоточних додатках.

У мові **C++** справжня атомарність досягається спеціальними інструкціями процесора (наприклад, CMPXCHG) через std::atomic, існує гарантовання атомарності базових операцій (load, store, fetch\_add, compare\_exchange). Використовуються атомарні операції без використання блокуючих механізмів синхронізації (таких як м’ютекси).

В мові **C++** атомарність досягається завдяки використанню бібліотеки <atomic>, яка надає спеціальні типи змінних (std::atomic<T>) та операцій над ними. Ці змінні і операції гарантують виконання без втручання інших потоків, тобто без "розриву" операцій.

У поточній лабораторній атомарність реалізується у функції find\_with\_atomic. Тут застосовано:

std::atomic<int> global\_max та std::atomic<int> global\_count — це атомарні змінні для збереження максимального значення і кількості його появ у масиві.

Операція global\_max.load() читає значення атомарно.

Операція compare\_exchange\_weak(current\_max, val) реалізує механізм **CAS (Compare-And-Swap)**:  
Вона порівнює поточне значення змінної з очікуваним (current\_max) і, якщо вони однакові, замінює значення змінної на нове (val) атомарно.

Якщо нове значення більше, то воно записується, а global\_count перезаписується на 1 через store(1).

Якщо нове значення таке саме, лічильник збільшується через атомарну операцію fetch\_add(1).

Отже, жоден інший потік не може змінити global\_max або global\_count в процесі виконання цих атомарних операцій, що забезпечує **справжню атомарність без блокувань**.

Коротко особливості методів реалізації можна описати так:

| **Метод** | **Що робить** | **Особливості** |
| --- | --- | --- |
| **CAS** | Атомарно змінює дані тільки якщо вони не змінилися іншим потоком | Без блокування, швидше при великій кількості потоків |
| **Mutex** | Забороняє одночасний доступ до ресурсу, примушує інші потоки чекати | Блокування потоків, простіша логіка |

Код програми:

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <cstdlib>

#include <ctime>

#include <iomanip>

#include <sstream>

#include <thread>

#include <mutex>

#include <atomic>

using namespace std;

using namespace std::chrono;

struct answer {

int max\_value;

int count\_max;

answer() : max\_value(0), count\_max(0) {}

answer(int max\_value, int count\_max) : max\_value(max\_value), count\_max(count\_max) {}

};

string int\_to\_string(int value) {

ostringstream oss;

oss << value;

return oss.str();

}

answer find\_slow(vector<int>& arr, long long& duration, int num\_threads = 1) {

auto start = high\_resolution\_clock::now();

int max\_value = \*max\_element(arr.begin(), arr.end());

int count\_max = count(arr.begin(), arr.end(), max\_value);

auto end = high\_resolution\_clock::now();

duration = duration\_cast<nanoseconds>(end - start).count();

return answer(max\_value, count\_max);

}

answer find\_with\_mutex(vector<int>& arr, long long& duration, int num\_threads) {

auto start = high\_resolution\_clock::now();

if (num\_threads == 0) num\_threads = thread::hardware\_concurrency();

vector<thread> threads;

mutex mtx;

int global\_max = 0;

int global\_count = 0;

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

int local\_max = 0;

int local\_count = 0;

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

if (arr[i] > local\_max) {

local\_max = arr[i];

local\_count = 1;

}

else if (arr[i] == local\_max) {

local\_count++;

}

}

lock\_guard<mutex> lock(mtx);

if (local\_max > global\_max) {

global\_max = local\_max;

global\_count = local\_count;

}

else if (local\_max == global\_max) {

global\_count += local\_count;

}

};

int chunk\_size = arr.size() / num\_threads;

for (int i = 0; i < num\_threads; ++i) {

int start\_idx = i \* chunk\_size;

int end\_idx = (i == num\_threads - 1) ? arr.size() : start\_idx + chunk\_size;

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

}

for (auto& th : threads) {

th.join();

}

auto end = high\_resolution\_clock::now();

duration = duration\_cast<nanoseconds>(end - start).count();

return answer(global\_max, global\_count);

}

answer find\_with\_atomic(vector<int>& arr, long long& duration, int num\_threads) {

auto start = high\_resolution\_clock::now();

if (num\_threads == 0) num\_threads = thread::hardware\_concurrency();

vector<thread> threads;

atomic<int> global\_max(0);

atomic<int> global\_count(0);

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

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

int val = arr[i];

while (true) {

int current\_max = global\_max.load();

if (val > current\_max) {

if (global\_max.compare\_exchange\_weak(current\_max, val)) {

global\_count.store(1);

break;

}

}

else if (val == current\_max) {

global\_count.fetch\_add(1);

break;

}

else {

break;

}

}

}

};

int chunk\_size = arr.size() / num\_threads;

for (int i = 0; i < num\_threads; ++i) {

int start\_idx = i \* chunk\_size;

int end\_idx = (i == num\_threads - 1) ? arr.size() : start\_idx + chunk\_size;

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

}

for (auto& th : threads) {

th.join();

}

auto end = high\_resolution\_clock::now();

duration = duration\_cast<nanoseconds>(end - start).count();

return answer(global\_max.load(), global\_count.load());

}

void print\_results(vector<int>& arr, answer(\*find\_func)(vector<int>&, long long&, int), const string& method\_label, int num\_threads) {

long long duration = 0;

answer result = find\_func(arr, duration, num\_threads);

cout << left

<< setw(12) << arr.size()

<< setw(20) << result.max\_value

<< setw(20) << result.count\_max

<< setw(20) << duration

<< setw(25) << method\_label

<< setw(15) << num\_threads

<< endl;

}

int main() {

srand(static\_cast<unsigned int>(time(0)));

const int sizes[] = {100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000 };

const int numSizes = sizeof(sizes) / sizeof(sizes[0]);

cout << left

<< setw(12) << "Size"

<< setw(20) << "Max Value"

<< setw(20) << "Count Max"

<< setw(20) << "Execution Time (ns)"

<< setw(25) << "Method"

<< setw(15) << "Threads"

<< endl;

cout << string(97, '-') << endl;

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

vector<int> arr(sizes[i]);

for (int j = 0; j < sizes[i]; j++) {

arr[j] = rand() % 1000 + 1;

}

print\_results(arr, find\_slow, "Single-threaded (slow)", 1);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 2);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 4);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 8);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 16);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 32);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 64);

print\_results(arr, find\_with\_mutex, "Multi-threaded (mutex)", 128);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 2);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 4);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 8);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 16);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 32);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 64);

print\_results(arr, find\_with\_atomic, "Multi-threaded (CAS)", 128);

cout << string(97, '-') << endl;

}

return 0;

}

Зафіксуємо результат роботи коду (рис 1 - перший запуск, рис 2- другий запуск) та продумонструємо консольний вивід результату на рис 3 - 10):

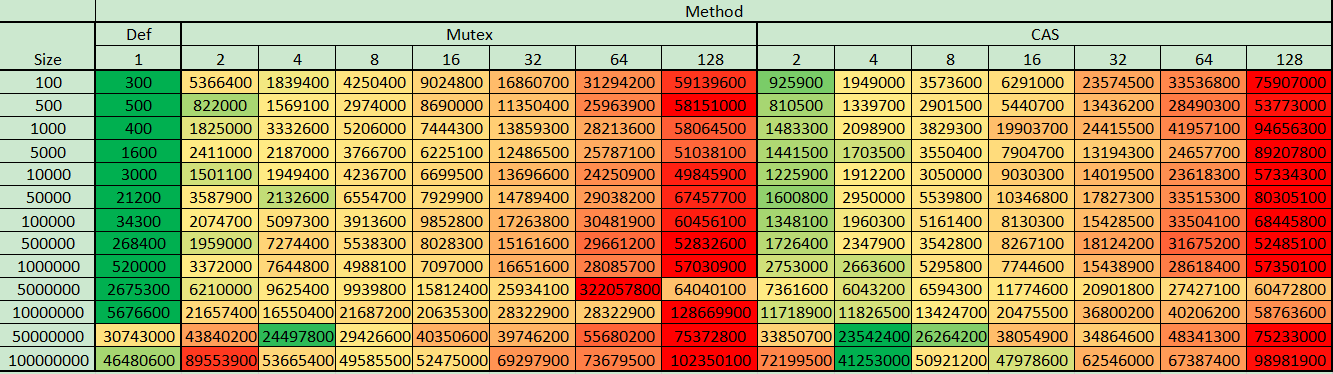
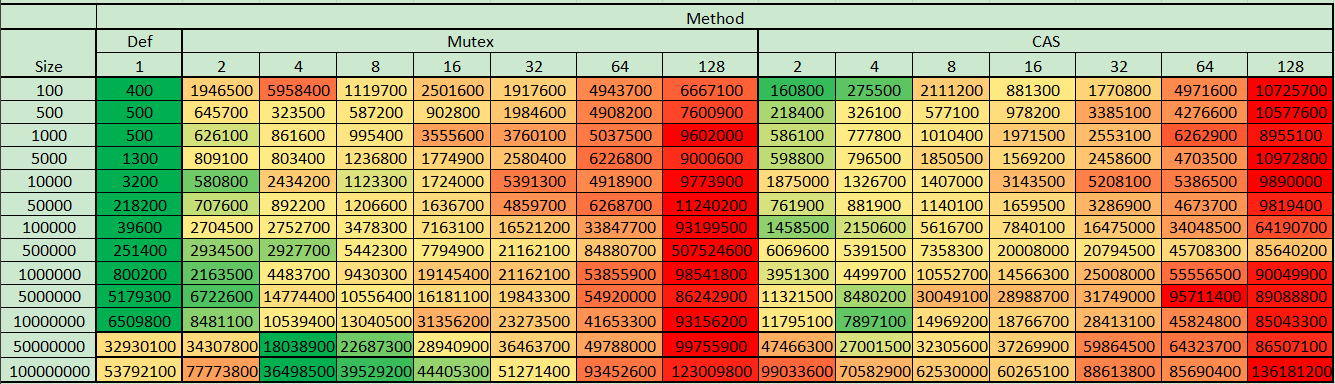


Рис. 1 - Перший результат виконання програми

Рис. 2 - Другий результат виконання програми

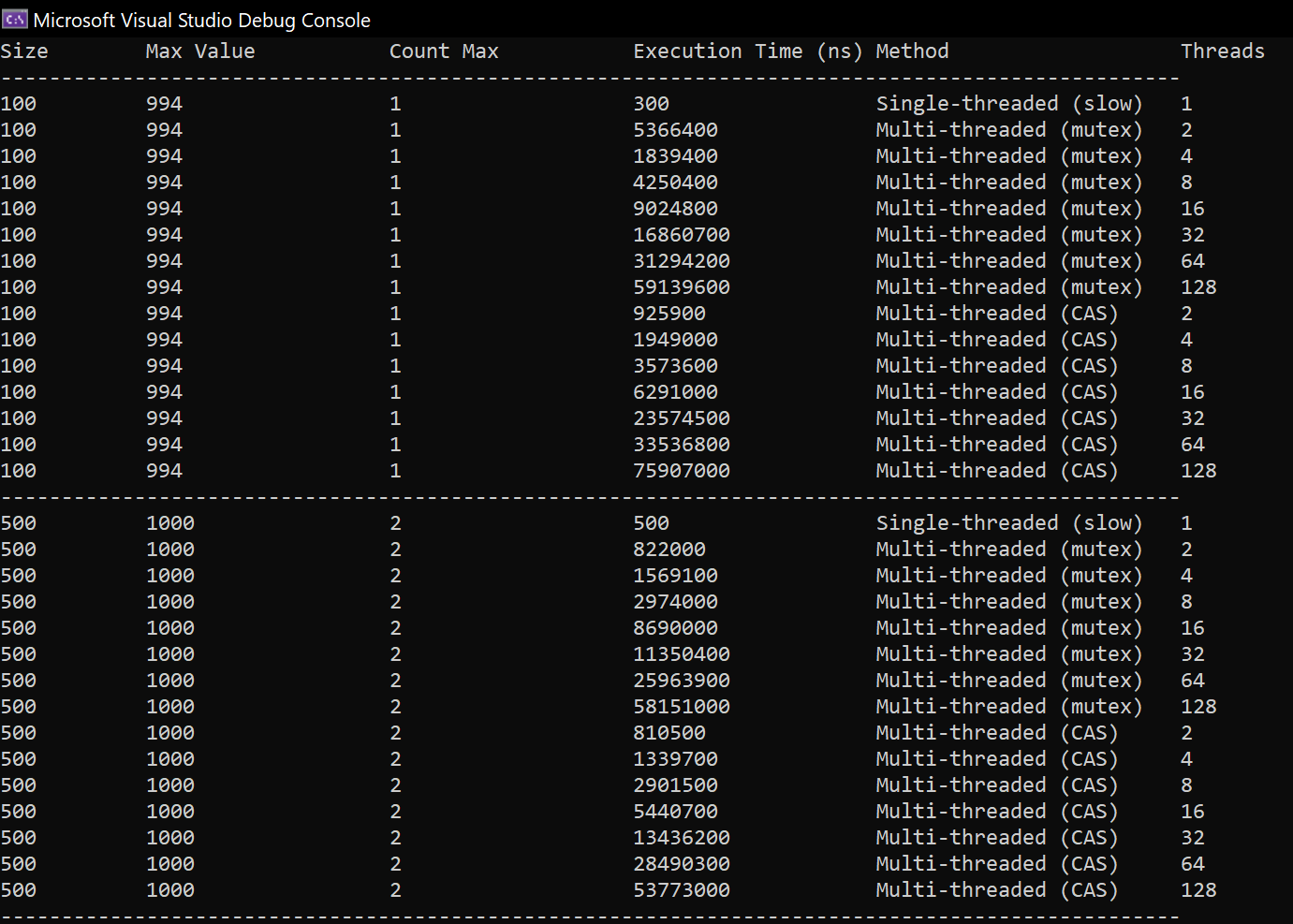
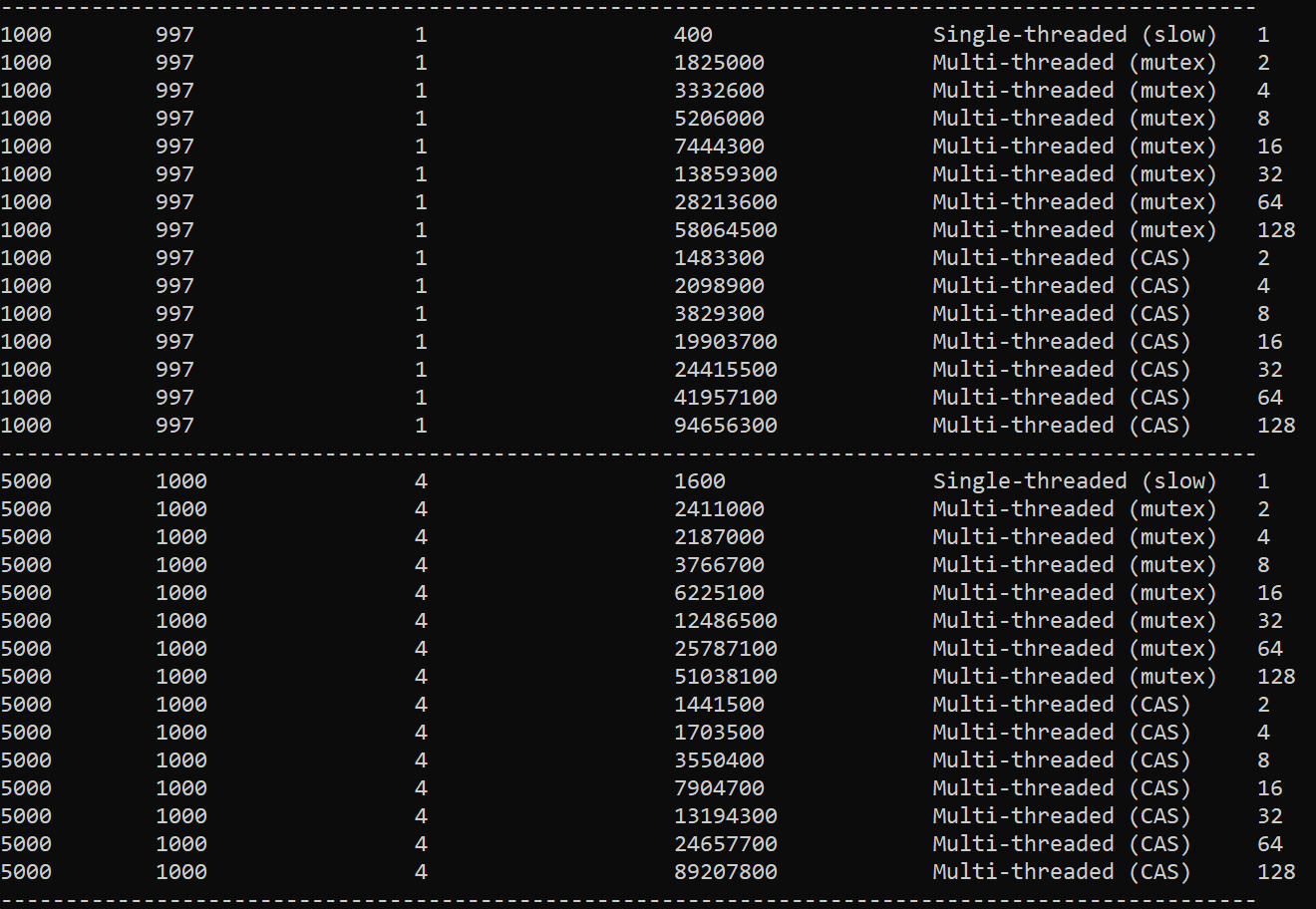


Рис. 3 - Результат виконання програми у консолі



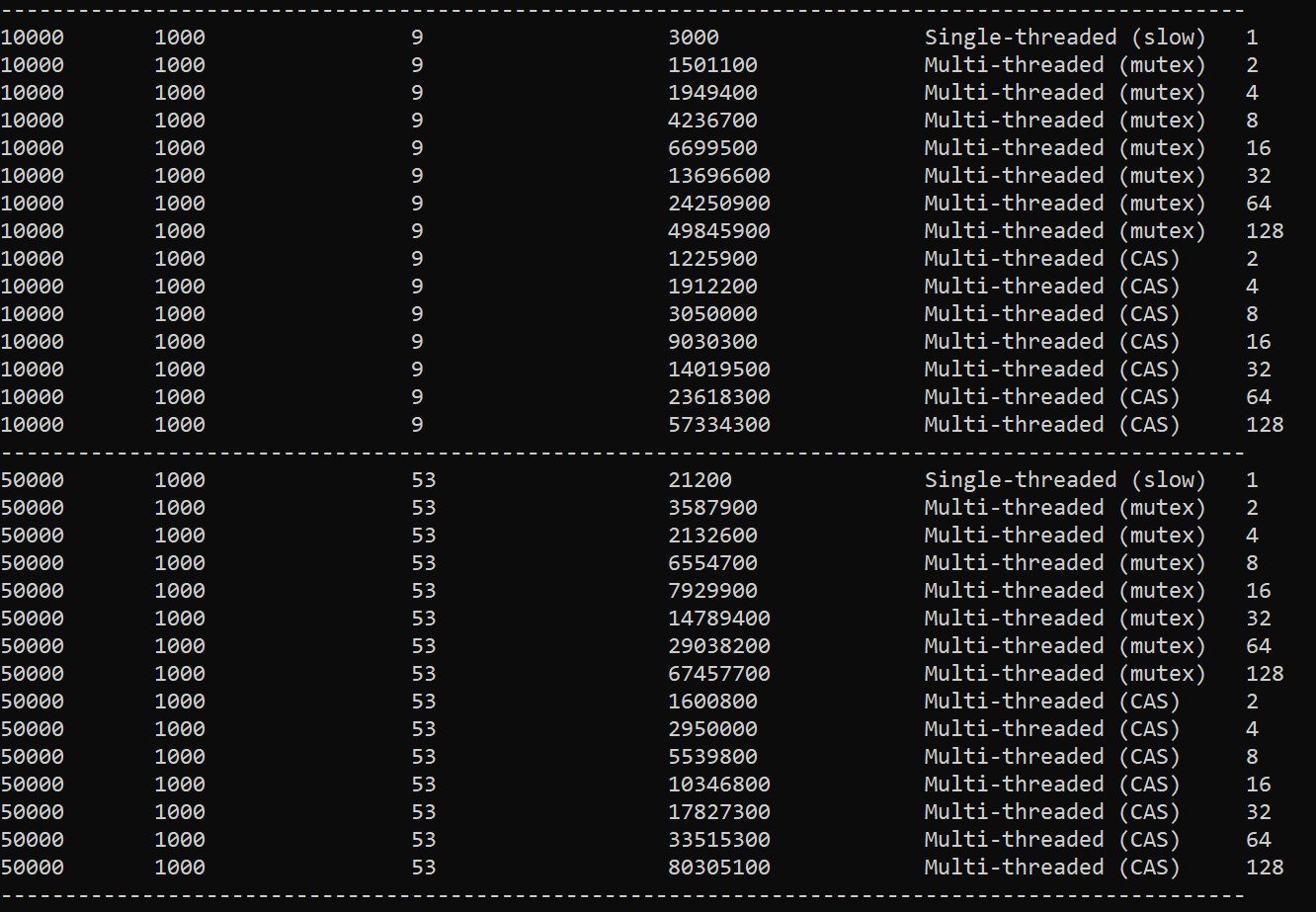
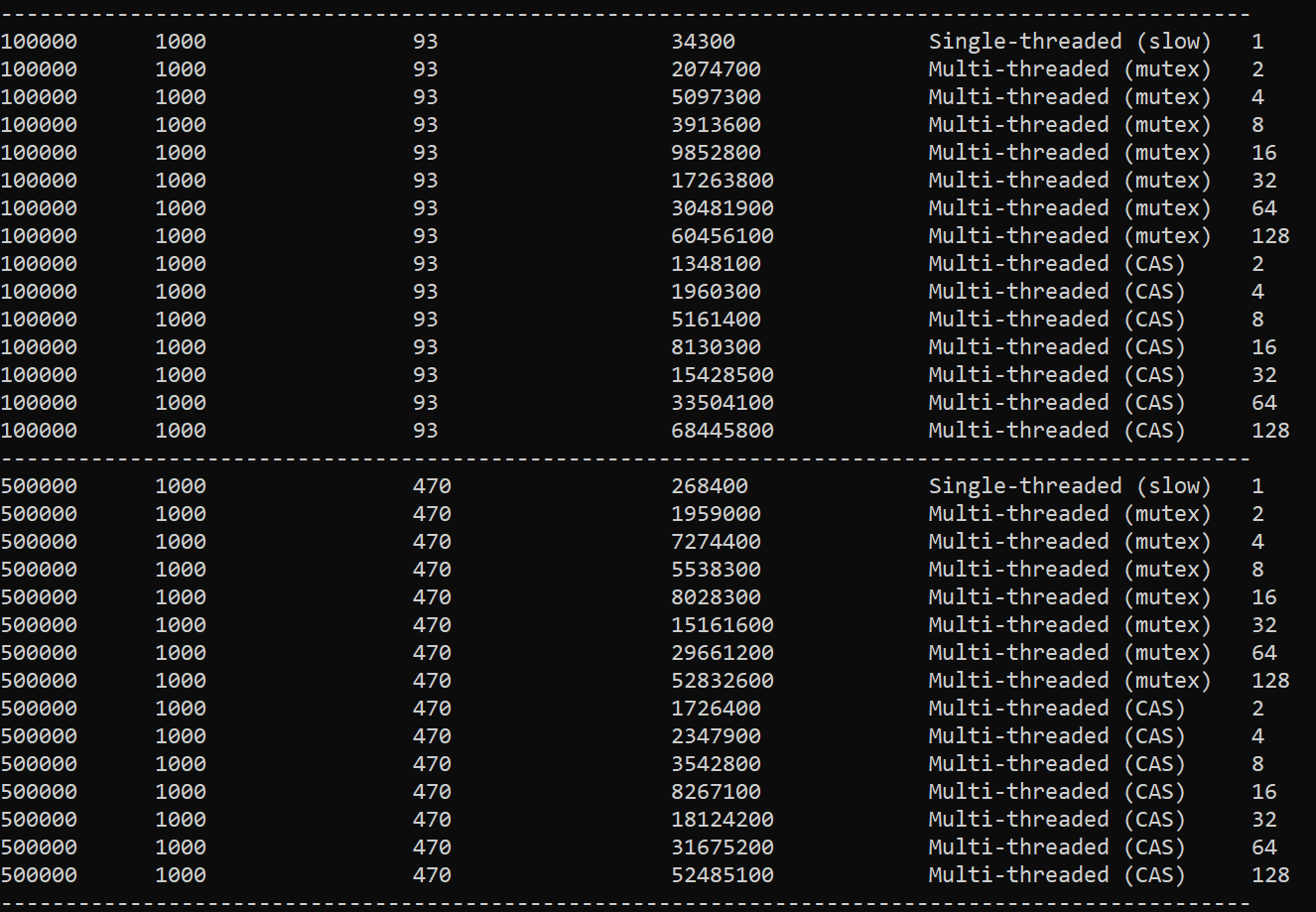


Рис. 4-5 - Результат виконання програми у консолі



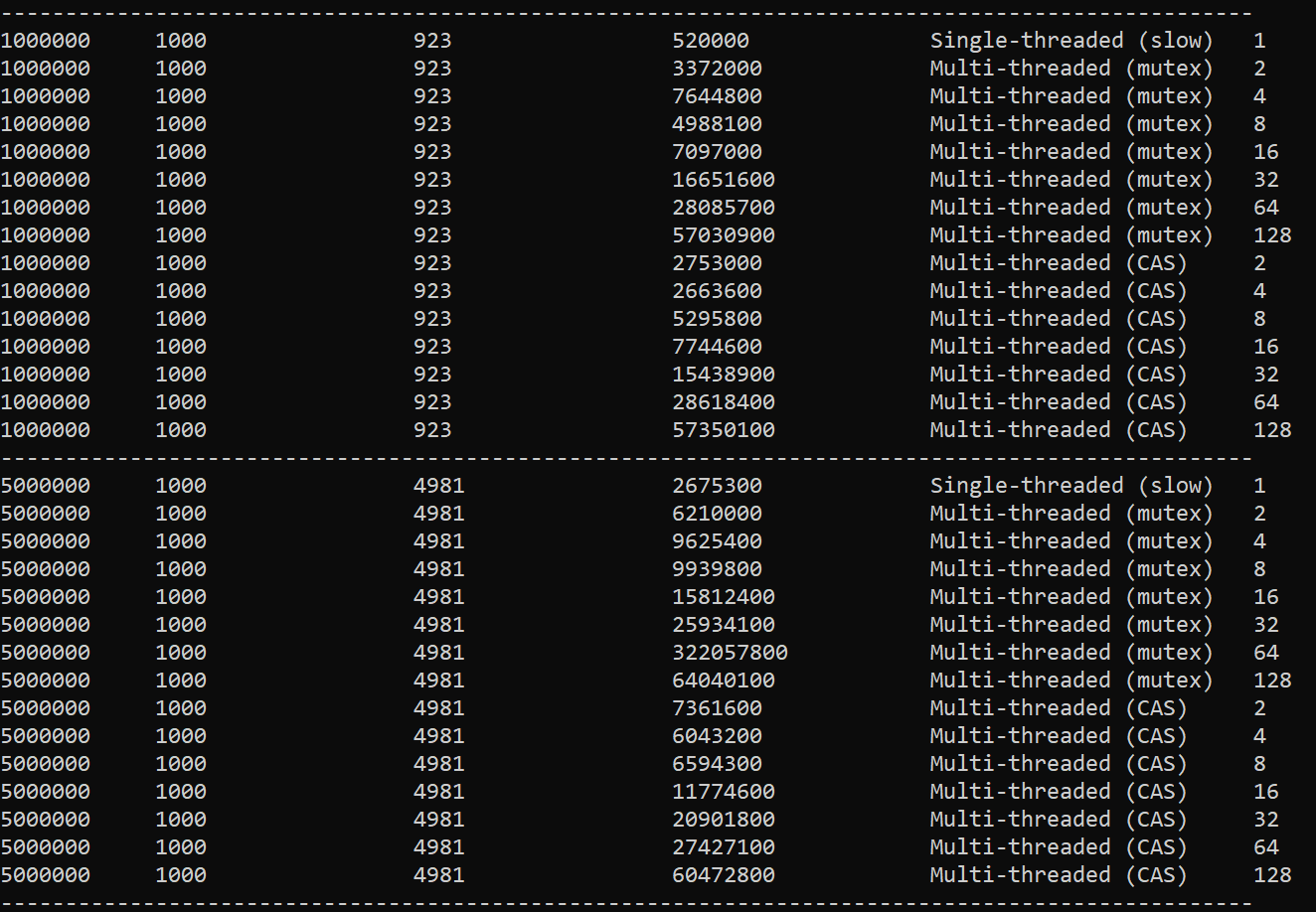
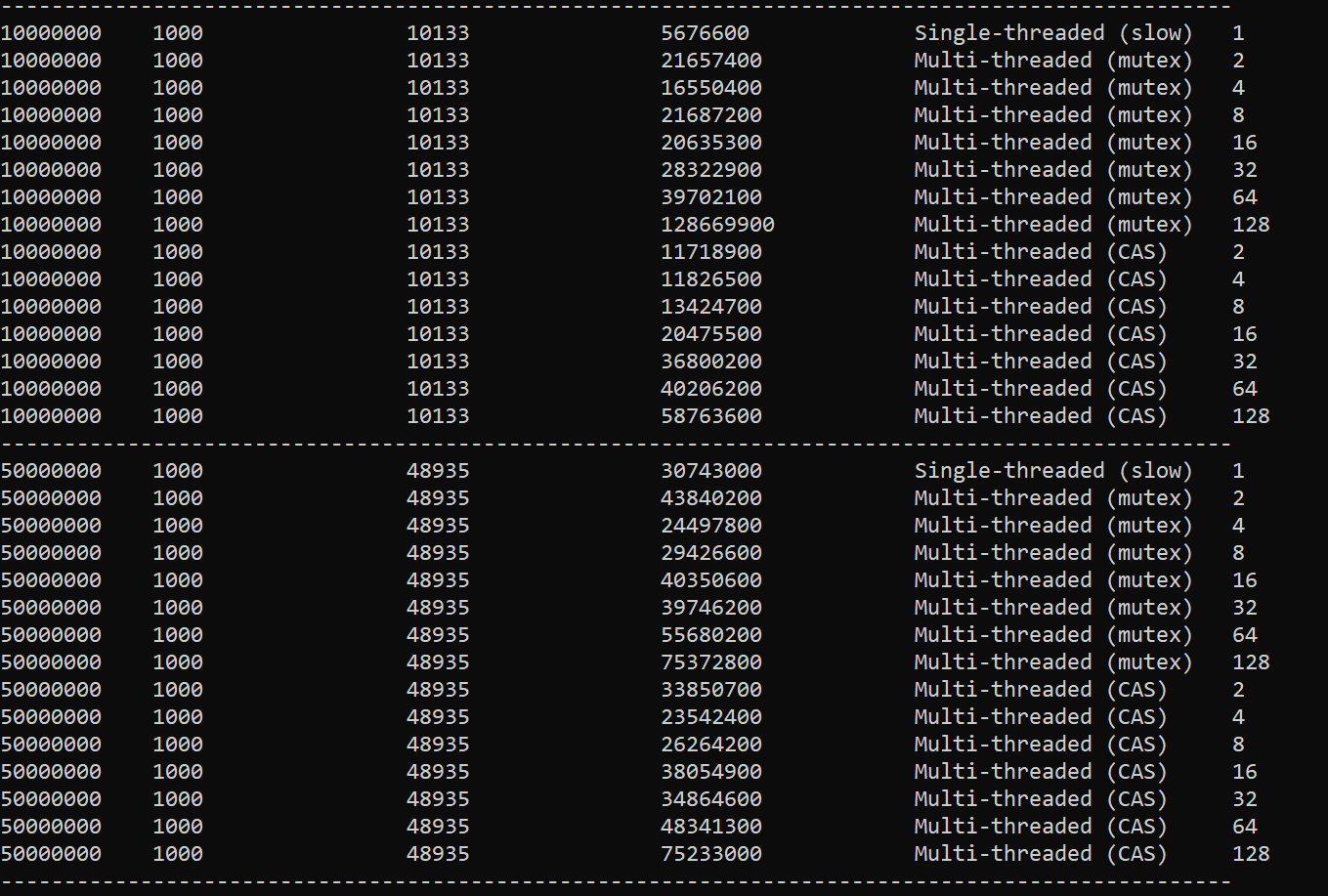


Рис. 6-7 - Результат виконання програми у консолі



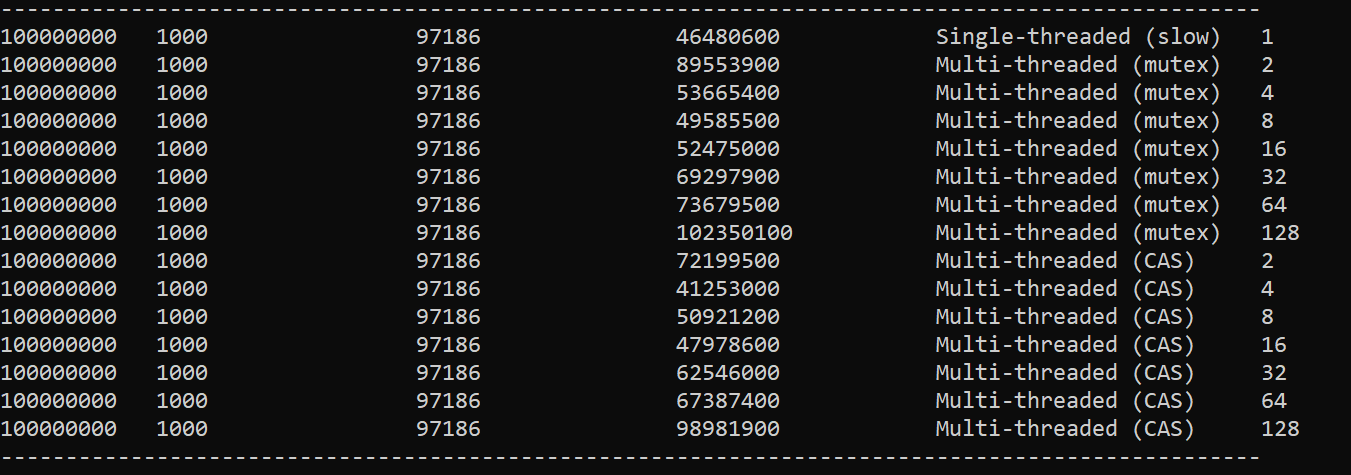


Рис. 8-9 - Результат виконання програми у консолі

**ВИСНОВОК**

У ході виконання роботи було реалізовано знаходження максимального елемента у векторі та підрахунок кількості його появ із використанням різних підходів: послідовного обчислення, багатопотокового обчислення з використанням м'ютекса та багатопотокового обчислення з використанням атомарних змінних (CAS-операцій).

Для кожного методу було заміряно:

**Size** — розмір вектора

**Max Value** — максимальне значення у векторі

**Count Max** — кількість появ максимального значення

**Execution Time** — час виконання обчислення

**Method** — метод обчислення

**Threads** — кількість потоків

Було протестовано такі розміри векторів: 100, 500, 1000, 5000, 10k, 50k, 100k, 500k, 1m, 5m, 10m, 50m та 100m.

Результати показали, що:

Для невеликих розмірів векторів найбільш ефективним є послідовне обчислення.

При зростанні розміру даних вигідним стає багатопотоковий підхід, особливо при використанні 2–8 потоків (для використання Mutex) та 2-4 потоки (для CAS).

Використання атомарних змінних замість м'ютексів дозволяє уникнути блокування та підвищити продуктивність за рахунок зменшення накладних витрат на синхронізацію.

Створення занадто великої кількості потоків (понад 8) не дає значного виграшу в продуктивності й навпаки може погіршити час виконання через перевантаження системи керування потоками.

Використання мьютексу в контексті даної лабораторної трохи програє у швидкості використанні CAS, проте це змінюється при великому об’ємі даних, де mutex працює швидше.

Під час тестування спостерігались окремі випадки аномального збільшення часу обчислення, що можна пояснити впливом сторонніх процесів операційної системи та особливостями апаратної платформи.

У моєму випадку обчислення були виконані у “release” режимі visual studio 2022 із вимкненими сторонніми програмами та із рівнем використованої операційної пам’яті 38%

Посилання на репозиторій із матеріалами - [GitHub](https://github.com/anton265463/Labs_Parellel_Computing)