

**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 за варіантом (№9)

Знайти суму всіх елементів кратних 10 та найменше число кратне 10.

**ХІД РОБОТИ**

У мові C++ атомарність досягається за допомогою стандартної бібліотеки <atomic>, яка надає класи та функції для роботи з атомарними змінними та операціями.

Шаблонний клас std::atomic<T> дозволяє створювати атомарні змінні для будь-якого типу T.

**Атомарні операції** :

* + **load()** — атомарне читання значення.
  + **store(value)** — атомарний запис значення.
  + **exchange(value)** — атомарна заміна значення з поверненням попереднього значення.
  + **compare\_exchange\_weak(expected, desired)** і **compare\_exchange\_strong(expected, desired)** — атомарна операція порівняння та обміну (CAS).

**Операції з модифікацією** :

* + **fetch\_add(value)** — атомарне додавання значення.
  + **fetch\_sub(value)** — атомарне віднімання значення.
  + **fetch\_or(value)**, **fetch\_and(value)**, **fetch\_xor(value)** — побітові операції.

У C++ можна вказати модель пам'яті для атомарних операцій, що впливає на те, як операції синхронізуються між потоками.

**Моделі пам'яті (memory orderings)** :

* **memory\_order\_relaxed**: Відсутність синхронізації.
* **memory\_order\_acquire**: Гарантує, що всі операції після цієї будуть виконані після завершення цієї операції.
* **memory\_order\_release**: Гарантує, що всі операції перед цією будуть завершені до її виконання.
* **memory\_order\_seq\_cst**: Строга послідовна консистентність (за замовчуванням).

Вирішимо всі завдання однією програмою, яка виконуватиме лінійні та паралельні обчислення на масивах різної величини.

Для паралельних обчислень розділимо масив на рівні частини та роздамо їх потокам для виконання. Нам знадобиться всього 2 спільних змінних: сума та мінімум. Для кожного числа, кратного 10, виконуватимемо додавання та перевірку на мінімум (організувавши спільний доступ для кількох потоків).

Код програми:

#include <iostream>

#include <chrono>

#include <thread>

#include <vector>

#include <atomic>

#include <mutex>

#include <cstdlib>

#include <ctime>

#include <iomanip>

**using** **namespace** std**;**

**using** chrono**::**nanoseconds**;**

**using** chrono**::**duration\_cast**;**

**using** chrono**::**high\_resolution\_clock**;**

void linearExecution**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**);**

void parallelWithMutex**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**,** int numThreads**);**

void parallelWithCAS**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**,** int numThreads**);**

int main**()** **{**

vector matrixSizes **=** **{**10000**,** 1000000**,** 100000000**,** 2000000000**};**

vector threadCounts **=** **{**8**,** 16**,** 32**,** 64**,** 128**,** 256**};**

cout **<<** "\nTest Results:" **<<** endl**;**

cout **<<** "Matrix Size\tThreads\tMode\tTime (seconds)\tSum\tMin Value" **<<** endl**;**

**for** **(**int matrixSize**:** matrixSizes**)** **{**

vector**<**int**>** data**(**matrixSize**);**

srand**(static\_cast<**unsigned**>(**time**(nullptr)));**

**for** **(**int i **=** 0**;** i **<** matrixSize**;** **++**i**)** **{**

data**[**i**]** **=** rand**()** **%** 1001**;**

**}**

long long sum **=** 0**;**

int minVal **=** INT32\_MAX**;**

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

linearExecution**(**data**,** sum**,** minVal**);**

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

double elapsed **=** duration\_cast**<**nanoseconds**>(**end **-** start**).**count**()** **\*** 1e-9**;**

cout **<<** matrixSize **<<** "\t\t-\tLinear\t" **<<** fixed **<<** setprecision**(**6**)** **<<** elapsed **<<** "\t" **<<** sum **<<** "\t" **<<** minVal **<<** endl**;**

cout **<<** endl**;**

**for** **(**int numThreads**:** threadCounts**)** **{**

long long sum **=** 0**;**

int minVal **=** INT32\_MAX**;**

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

parallelWithMutex**(**data**,** sum**,** minVal**,** numThreads**);**

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

double elapsed **=** duration\_cast**<**nanoseconds**>(**end **-** start**).**count**()** **\*** 1e-9**;**

cout **<<** matrixSize **<<** "\t\t" **<<** numThreads **<<** "\tMutex\t" **<<** fixed **<<** setprecision**(**6**)** **<<** elapsed **<<** "\t" **<<**

sum **<<** "\t" **<<** minVal **<<** endl**;**

**}**

cout **<<** endl**;**

**for** **(**int numThreads**:** threadCounts**)** **{**

long long sum **=** 0**;**

int minVal **=** INT32\_MAX**;**

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

parallelWithCAS**(**data**,** sum**,** minVal**,** numThreads**);**

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

double elapsed **=** duration\_cast**<**nanoseconds**>(**end **-** start**).**count**()** **\*** 1e-9**;**

cout **<<** matrixSize **<<** "\t\t" **<<** numThreads **<<** "\tCAS\t" **<<** fixed **<<** setprecision**(**6**)** **<<** elapsed **<<** "\t" **<<**

sum **<<** "\t" **<<** minVal **<<** endl**;**

**}**

cout **<<** endl **<<** endl**;**

**}**

**return** 0**;**

**}**

void linearExecution**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**)** **{**

sum **=** 0**;**

minVal **=** INT32\_MAX**;**

**for** **(**int value**:** data**)** **{**

**if** **(**value **%** 10 **==** 0**)** **{**

sum **+=** value**;**

**if** **(**value **<** minVal**)** **{**

minVal **=** value**;**

**}**

**}**

**}**

**}**

void processSectionWithMutex**(**int start**,** int end**,** const vector**<**int**>** **&**data**,** long long **&**localSum**,** int **&**localMin**,** mutex **&**mtx**)** **{**

long long sum **=** 0**;**

int minVal **=** INT32\_MAX**;**

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

**if** **(**data**[**i**]** **%** 10 **==** 0**)** **{**

sum **+=** data**[**i**];**

**if** **(**data**[**i**]** **<** minVal**)** **{**

minVal **=** data**[**i**];**

**}**

**}**

**}**

lock\_guard lock**(**mtx**);**

localSum **+=** sum**;**

**if** **(**minVal **<** localMin**)** **{**

localMin **=** minVal**;**

**}**

**}**

void parallelWithMutex**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**,** int numThreads**)** **{**

sum **=** 0**;**

minVal **=** INT32\_MAX**;**

mutex mtx**;**

vector**<**thread**>** threads**;**

int chunkSize **=** data**.**size**()** **/** numThreads**;**

**for** **(**int t **=** 0**;** t **<** numThreads**;** **++**t**)** **{**

int start **=** t **\*** chunkSize**;**

int end **=** **(**t **==** numThreads **-** 1**)** **?** data**.**size**()** **:** start **+** chunkSize**;**

threads**.**emplace\_back**(**processSectionWithMutex**,** start**,** end**,** cref**(**data**),** ref**(**sum**),** ref**(**minVal**),** ref**(**mtx**));**

**}**

**for** **(**auto **&**th**:** threads**)** **{**

**if** **(**th**.**joinable**())** **{**

th**.**join**();**

**}**

**}**

**}**

void processSectionWithCAS**(**int start**,** int end**,** const vector**<**int**>** **&**data**,** atomic**<**long long**>** **&**atomicSum**,** atomic**<**int**>** **&**atomicMin**)** **{**

long long localSum **=** 0**;**

int localMin **=** INT32\_MAX**;**

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

**if** **(**data**[**i**]** **%** 10 **==** 0**)** **{**

localSum **+=** data**[**i**];**

**if** **(**data**[**i**]** **<** localMin**)** **{**

localMin **=** data**[**i**];**

**}**

**}**

**}**

atomicSum**.**fetch\_add**(**localSum**,** memory\_order\_relaxed**);**

int currentMin **=** atomicMin**.**load**(**memory\_order\_relaxed**);**

**while** **(**localMin **<** currentMin **&&** atomicMin**.**compare\_exchange\_weak**(**currentMin**,** localMin**,** memory\_order\_relaxed**))** **{**

**}**

**}**

void parallelWithCAS**(**const vector**<**int**>** **&**data**,** long long **&**sum**,** int **&**minVal**,** int numThreads**)** **{**

atomic**<**long long**>** atomicSum**(**0**);**

atomic**<**int**>** atomicMin**(**INT32\_MAX**);**

vector**<**thread**>** threads**;**

int chunkSize **=** data**.**size**()** **/** numThreads**;**

**for** **(**int t **=** 0**;** t **<** numThreads**;** **++**t**)** **{**

int start **=** t **\*** chunkSize**;**

int end **=** **(**t **==** numThreads **-** 1**)** **?** data**.**size**()** **:** start **+** chunkSize**;**

threads**.**emplace\_back**(**processSectionWithCAS**,** start**,** end**,** cref**(**data**),** ref**(**atomicSum**),** ref**(**atomicMin**));**

**}**

**for** **(**auto **&**th**:** threads**)** **{**

**if** **(**th**.**joinable**())** **{**

th**.**join**();**

**}**

**}**

sum **=** atomicSum**.**load**();**

minVal **=** atomicMin**.**load**();**

**}**

Запустимо програму та занесемо результати роботи в окремі таблиці.

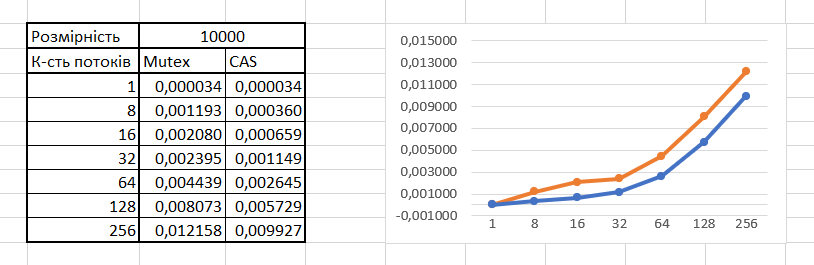


Рисунок 1 — Час роботи програми з вектором розмірності 10000.

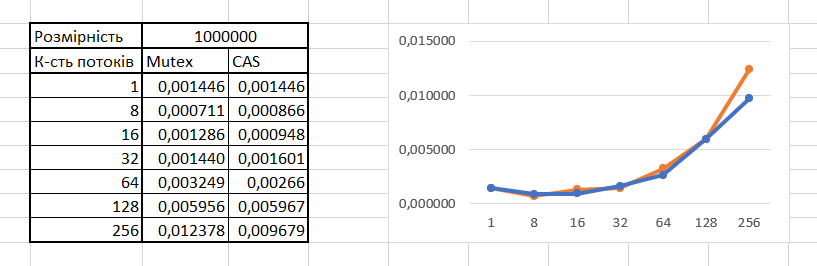


Рисунок 2 — Час роботи програми з вектором розмірності 1000000.

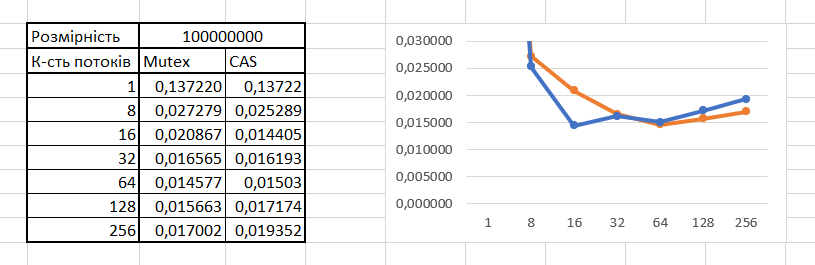


Рисунок 3 — Час роботи програми з вектором розмірності 100000000.

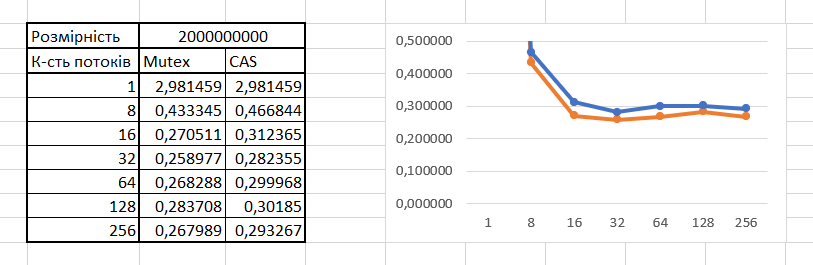


Рисунок 4 — Час роботи програми з вектором розмірності 2000000000.

**ВИСНОВКИ**

Було виконано лабораторну роботу, у якій створено програму, що вирішує задану задачу різною кількістю потоків з матрицями різної розмірності, використовуючи різні механізми синхронізації.

Знову-таки для даних малих розмірностей паралельність виконання потоками має низьку ефективність (лінійне виконання краще). Проте для більших розмірностей паралелізм вже необхідний.

Можна простежити закономірність, що зі збільшенням розмірності даних неблокуючий алгоритм втрачає свою ефективність у порівнянні з механізмами синхронізації (м’ютекси в даному випадку). В розмірності 10000 неблокуючий алгоритм краще при будь-якій кількості потоків. Натомість в розмірності 2000000000 ситуація прямо протилежна.

На мою думку, таке трапляється через складність неблокуючого алгоритму. А саме, якщо декілька потоків намагаються оновити якусь змінну (у нас це мінімум або ж сума), то операція може часто завершуватися помилкою та повторюватися знову і знову. Також в цьому процесі між потоками відбувається активний обмін кеш лініями, що теж впливає на швидкість.

Натомість м’ютекс хоч і потребує чіткого блокування змінної, виконання операцій з його участю є передбачуваним і стабільним. До того ж, з розвитком бібліотек програмування покращуються і механізми синхронізації, що ми і можемо побачити у результатах виконання лабораторної.

Також [посилання на репозиторій](https://github.com/nikk0308/Parallel_Calculations).