

**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  
Моренець Сергій Андрійович

Прийняв:

асистент Яременко В. С.

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 20:***

Знайти суму трьох максимальних елементів, і самі елементи.

***Як досягається справжня атомарність?***

1. Атомарні змінні (std::atomic) забезпечують безпечний доступ до даних без використання м'ютексів. Вони гарантують, що операції над змінними виконуються як єдиний неподільний крок. Це дозволяє уникнути станів гонки. Атомарні змінні підтримують базові операції (load, store) та арифметичні (fetch\_add, fetch\_sub), які виконуються без переривань.
2. Атомарні операції є основою для реалізації багатопотокових алгоритмів без блокувань. Вони виконуються без проміжних станів, тому інші потоки не можуть спостерігати часткове оновлення даних. Найпоширеніші операції:

* load() — зчитує поточне значення.
* store() — встановлює нове значення.
* compare\_exchange\_weak() та compare\_exchange\_strong() — порівнюють поточне значення із заданим та оновлюють його у випадку збігу.  
  Ці операції реалізуються на апаратному рівні, що робить їх ефективними, але складними для використання.

1. В C++ є кілька режимів упорядкування пам’яті (memory\_order), які визначають порядок виконання атомарних операцій:

* memory\_order\_relaxed — не гарантує жодного порядку, лише атомарність.
* memory\_order\_acquire — гарантує, що подальші читання не будуть виконані до завершення цієї операції.
* memory\_order\_release — запобігає перенесенню попередніх записів після цієї операції.
* memory\_order\_seq\_cst — забезпечує повну послідовність операцій.

Бар’єри пам’яті дозволяють контролювати порядок виконання команд, мінімізуючи накладні витрати.

1. На рівні процесора атомарні операції реалізуються за допомогою спеціальних інструкцій, таких як LOCK, CMPXCHG та LL/SC. Вони забезпечують апаратну атомарність, блокуючи шину пам'яті або використовуючи спеціальні регістри.

Виконання завдання:

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <vector>

#include <thread>

#include <mutex>

#include <atomic>

#include <chrono>

#define MAX\_ELEMENTS\_COUNT 3

using namespace std;

using namespace chrono;

vector<int> threadsCount = {4, 8, 16, 32, 64, 128, 256};

vector<int> arrSizes = {10000, 100000, 1000000, 10000000, 100000000, 1000000000};

vector<int> currentAnswers;

void arrayFilling (vector<int> & array, int arrSize) {

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

        array.push\_back(rand()); // is not limit for data

    }

}

void arraySorting (int \* array) {

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT - 1; i++) {

        for (int j = 0; j < MAX\_ELEMENTS\_COUNT - i - 1; j++) {

            if (array[j] > array[j + 1]) {

                swap(array[j], array[j + 1]);

            }

        }

    }

}

void atomicIntArraySorting(atomic<int> \*array) {

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT - 1; i++) {

        for (int j = 0; j < MAX\_ELEMENTS\_COUNT - i - 1; j++) {

            int j\_val = array[j].load();

            int j\_plus\_1\_val = array[j + 1].load();

            if (j\_val > j\_plus\_1\_val) {

                while (!array[j].compare\_exchange\_weak(j\_val, j\_plus\_1\_val)) {}

                while (!array[j + 1].compare\_exchange\_weak(j\_plus\_1\_val, j\_val)) {}

            }

        }

    }

}

void simpleAlgorithm (vector<int> &array) {

    vector<int> maxElements;

    maxElements.resize(MAX\_ELEMENTS\_COUNT);

    int maxElementsSum = 0;

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        maxElements[i] = array[i];

    }

    arraySorting(&maxElements[0]);

    for (int i = MAX\_ELEMENTS\_COUNT; i < array.size(); i++) {

        if (array[i] > maxElements[0]) {

            maxElements[0] = array[i];

            arraySorting(&maxElements[0]);

        }

    }

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        maxElementsSum += maxElements[i];

    }

}

void localMutexAlgorithm (vector<int> &array, vector<int> &maxElements, int startIndex, int endIndex, mutex &mtx) {

    vector<int> localMaxElements;

    localMaxElements.resize(MAX\_ELEMENTS\_COUNT);

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        localMaxElements[i] = array[i + startIndex];

    }

    arraySorting(&localMaxElements[0]);

    for (int i = MAX\_ELEMENTS\_COUNT + startIndex; i < endIndex; i++) {

        if (array[i] > localMaxElements[0]) {

            localMaxElements[0] = array[i];

            arraySorting(&localMaxElements[0]);

        }

    }

    lock\_guard<mutex> lock(mtx);

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        if (localMaxElements[localMaxElements.size() - 1 - i] > maxElements[0]) {

            maxElements[0] = localMaxElements[localMaxElements.size() - 1 - i];

            arraySorting(&maxElements[0]);

        }

        else {

            break;

        }

    }

}

void mutexAlgorithm (vector<int> &array, int threadsCount) {

    vector <int> maxElements;

    maxElements.resize(MAX\_ELEMENTS\_COUNT);

    int maxElementsSum = 0;

    mutex mtx;

    vector<thread> threads;

    int elementsPerThread = array.size() / threadsCount;

    int remainingElements = array.size() % threadsCount;

    int startIndex = 0;

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

        int endIndex = startIndex + elementsPerThread;

        if (i < remainingElements) {

            endIndex++;

        }

        threads.emplace\_back(localMutexAlgorithm, ref(array), ref(maxElements), startIndex, endIndex, ref(mtx));

        startIndex = endIndex;

    }

    for (int i = 0; i < threads.size(); i++) {

        threads[i].join();

    }

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        maxElementsSum += maxElements[i];

    }

}

void localCASAlgorithm(vector<int> &array, vector<atomic<int>> &maxElements, int startIndex, int endIndex) {

    vector<int> localMaxElements(MAX\_ELEMENTS\_COUNT);

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        localMaxElements[i] = array[i + startIndex];

    }

    arraySorting(&localMaxElements[0]);

    for (int i = MAX\_ELEMENTS\_COUNT + startIndex; i < endIndex; i++) {

        if (array[i] > localMaxElements[0]) {

            localMaxElements[0] = array[i];

            arraySorting(&localMaxElements[0]);

        }

    }

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        int localValue = localMaxElements[localMaxElements.size() - 1 - i];

        int currValue = maxElements[0].load();

        if (localValue > currValue) {

            while (!maxElements[0].compare\_exchange\_weak(currValue, localValue)) { }

                atomicIntArraySorting(&maxElements[0]);

            }

            else {

                break;

            }

    }

}

void CASAlgorithm (vector<int> &array, int threadsCount) {

    vector<atomic<int>> maxElements(MAX\_ELEMENTS\_COUNT);

    int maxElementsSum = 0;

    vector<thread> threads;

    int elementsPerThread = array.size() / threadsCount;

    int remainingElements = array.size() % threadsCount;

    int startIndex = 0;

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

        int endIndex = startIndex + elementsPerThread;

        if (i < remainingElements) {

            endIndex++;

        }

        threads.emplace\_back(localCASAlgorithm, ref(array), ref(maxElements), startIndex, endIndex);

        startIndex = endIndex;

    }

    for (int i = 0; i < threads.size(); i++) {

        threads[i].join();

    }

    for (int i = 0; i < MAX\_ELEMENTS\_COUNT; i++) {

        maxElementsSum += maxElements[i].load();

    }

}

void task () {

    for (int i = 0; i < arrSizes.size(); i++) {

        cout << "=========================" << endl;

        cout << "Array size: " << arrSizes[i] << endl;

        cout << "-------------------------" << endl;

        vector<int> array;

        arrayFilling(array, arrSizes[i]);

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

        simpleAlgorithm(array);

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

        auto duration = duration\_cast<milliseconds>(end - start);

        cout << "Simple algorithm: " << duration.count() << " ms" << endl;

        cout << "- - - - - - - - - - - - -" << endl;

        cout << "Threads\t| Mutex\t| CAS" << endl;

        cout << "- - - - - - - - - - - - -" << endl;

        for (int j = 0; j < threadsCount.size(); j++) {

            cout << threadsCount[j] << "\t| ";

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

            mutexAlgorithm(array, threadsCount[j]);

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

            duration = duration\_cast<milliseconds>(end - start);

            cout << duration.count() << " ms\t| ";

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

            CASAlgorithm(array, threadsCount[j]);

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

            duration = duration\_cast<milliseconds>(end - start);

            cout << duration.count() << " ms" << endl;

        }

        cout << "=========================" << endl << endl;

    }

}

int main() {

    srand(time(0));

    task();

    return 0;

}

***Порівняння результатів тестування:***

***Висновки***

З отримаманих результатів тестування можемо виділити такі особливості алгоритмів для даної задачі:

* Алгоритм з використанням паралелізації стає ефективнішим за лінійну реалізацію приблизно починаючи від 10кк елементів в масиві. Якщо б на кожній ітерації проходження масиву ми виконували складніші обчислення, то алгоритм з паралелізацією став би ефективнішим на меншій розмірності масиву.
* Серед розглянутих комбінацій розмірності масиву та кількості потоків алгоритми з використанням паралелізації є ефективнішими на меншій кількості потоків. Для алгоритму з м’ютексами це можна пояснити тим, що витрачається менше часу на блокування потоків, а для елементу з порівнянням-вставленням тим, що при меншій кількості потоків виникає менше конфліктів при оновленні атомарних змінних.
* Великої різниці між алгоритмами з використанням м’ютексів та CAS не помічено. На неграничних кількостях потоків обидва алгоритми ведуть себе схожим чином. Для обох алгоритмів найефективнішою кількістю потоків є 8 – 32 потоки.

Git: <https://github.com/m0renets/PC_Lab_2.git>