**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**

**імені Ігоря Сікорського»**

**Лабораторна робота № 1**

**з курсу «Паралельні обчислення»**

**на тему «Дослідження базових операцій з потоками виконання»**

**Виконав студент ІПСА групи ДА-22**

**Котляр Єгор Ярославович**

Київ – 2025

**Мета роботи**

Розглянути основні операції з потоками виконання, навчитися використовувати неблокуючу паралелізацію для вирішення найпростіших математичних задач, використовуючи обрану мову програмування. Навчитися досліджувати та оцінювати ефективність паралелізації алгоритму.

**Завдання**

1. Визначити основні характеристики ПК, котрі на думку студента впливають на ефективність виконання паралельних обчислень. Зафіксувати значення даних характеристик для ПК студента, та занести їх до протоколу роботи.

2. Створити або використати наявній механізм для заміру часу виконання програми, або інших параметрів, котрі студент вважає релевантними. Занести опис механізму до роботи.

3. Вирішити обрану за варіантом задачу, не використовуючи паралелізацію. Заміряти час вирішення задачі, або інші параметри, котрі студент вважає релевантними.

4. Вирішити оборану за варіантом задачу, використовуючи паралелізацію. Заміряти час вирішення задачі, або інші параметри, котрі студент вважає релевантними. Обґрунтувати вибір алгоритму паралелізації (розподілення даних між потоками), надати опис та обґрунтування в протоколі роботи.

5. Перевірити алгоритм на фіксованих кількостях потоків: 2-рази меншій, ніж кількість фізичних ядер, на кількості рівній фізичним ядрам, на кількості рівній логічних ядрам, на кількості більшій в 2, 4, 8, 16 разів ніж кількість логічних ядер.

6. Повторити пункт 5 з використанням різної розмірності даних.

7. Зробити таблиці та графіки залежності часу виконання завдання від кількості потоків для різної розмірності даних. Надати опис графіків, з причинами виникнення отриманих результатів в протоколі роботи.

8. Надати висновок, що повинен містити аналіз отриманих результатів.

**Варіант 14**: заповнити квадратну матрицю випадковими числами. На головній діагоналі розмістити максимальний елемент стовпчика.

**Хід виконання роботи**

За основні характеристики ПК я обрав системну плату та процесор:

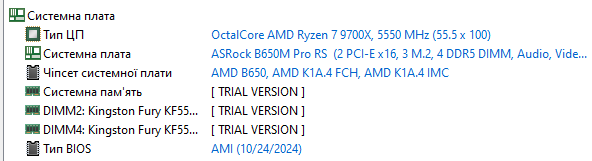


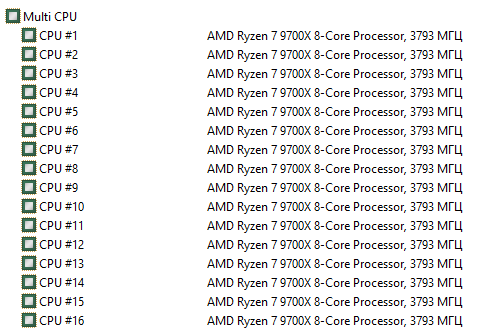
Рисунок 1 – Інформація системної плати.

Рисунок 2 – Кількість логічних ядер.

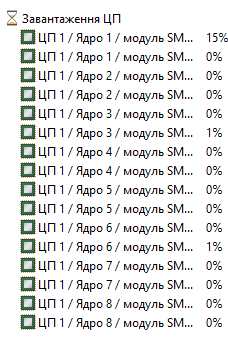


Рисунок 3 – Кількість фізичних ядер.

Як бачимо з рисунків мій процесор має 8 фізичних та 16 логічних ядер.

**Код програми:**

#include <windows.h>  
#include <iostream>  
#include <chrono>  
#include <thread>  
#include <vector>  
#include <iomanip>  
  
using namespace std;  
  
void printCPUInfo() {  
 SYSTEM\_INFO sysInfo;  
 GetSystemInfo(&sysInfo);  
  
 cout << "=== Інформація про процесор ===\n";  
  
 cout << "Архітектура процесора: ";  
 switch (sysInfo.wProcessorArchitecture) {  
 case PROCESSOR\_ARCHITECTURE\_AMD64:  
 cout << "x64 (AMD або Intel)\n";  
 break;  
 case PROCESSOR\_ARCHITECTURE\_INTEL:  
 cout << "x86\n";  
 break;  
 case PROCESSOR\_ARCHITECTURE\_ARM:  
 cout << "ARM\n";  
 break;  
 case PROCESSOR\_ARCHITECTURE\_ARM64:  
 cout << "ARM64\n";  
 break;  
 default:  
 cout << "Невідома архітектура\n";  
 break;  
 }  
  
 cout << "Логічних процесорів: " << sysInfo.dwNumberOfProcessors << "\n";  
 cout << "Розмір сторінки пам'яті: " << sysInfo.dwPageSize << " байт\n";  
 cout << "Мінімальна адреса додатку: " << sysInfo.lpMinimumApplicationAddress << "\n";  
 cout << "Максимальна адреса додатку: " << sysInfo.lpMaximumApplicationAddress << "\n";  
 cout << "\n";  
}  
  
void printMemoryInfo() {  
 MEMORYSTATUSEX statex;  
 statex.dwLength = sizeof(statex);  
  
 cout << "=== Інформація про пам'ять ===\n";  
 if (GlobalMemoryStatusEx(&statex)) {  
 double totalGB = statex.ullTotalPhys / (1024.0 \* 1024 \* 1024);  
 double availGB = statex.ullAvailPhys / (1024.0 \* 1024 \* 1024);  
 cout << "Загальна фізична пам'ять (RAM): " << fixed << setprecision(2)  
 << totalGB << " GB\n";  
 cout << "Доступна фізична пам'ять (RAM): " << fixed << setprecision(2)  
 << availGB << " GB\n";  
 } else {  
 cerr << "Помилка отримання інформації про пам'ять.\n";  
 }  
 cout << "\n";  
}  
  
vector<vector<int>> createRandomMatrix(int n) {  
 vector<vector<int>> mat(n, vector<int>(n));  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++) {  
 mat[i][j] = rand() % 1001;  
 }  
 }  
 return mat;  
}  
  
void nonParallelSolution(vector<vector<int>>& mat) {  
 int n = static\_cast<int>(mat.size());  
 for (int j = 0; j < n; j++) {  
 int maxVal = mat[0][j];  
 for (int i = 1; i < n; i++) {  
 if (mat[i][j] > maxVal)  
 maxVal = mat[i][j];  
 }  
 mat[j][j] = maxVal;  
 }  
}  
  
void parallelColumnMax(vector<vector<int>>& mat, int start, int end) {  
 int n = static\_cast<int>(mat.size());  
 for (int j = start; j < end; j++) {  
 int maxVal = mat[0][j];  
 for (int i = 1; i < n; i++) {  
 if (mat[i][j] > maxVal)  
 maxVal = mat[i][j];  
 }  
 mat[j][j] = maxVal;  
 }  
}  
  
void parallelSolution(vector<vector<int>>& mat, int numThreads) {  
 int n = static\_cast<int>(mat.size());  
 vector<thread> threads;  
 int columnsPerThread = n / numThreads;  
 int remainder = n % numThreads;  
 int start = 0;  
 for (int t = 0; t < numThreads; t++) {  
 int extra = (t < remainder) ? 1 : 0;  
 int end = start + columnsPerThread + extra;  
 threads.push\_back(thread(parallelColumnMax, ref(mat), start, end));  
 start = end;  
 }  
 for (auto& th : threads) {  
 th.join();  
 }  
}  
  
int main() {  
 SetConsoleOutputCP(CP\_UTF8);  
 printCPUInfo();  
 printMemoryInfo();  
  
 srand(GetTickCount());  
  
 vector<int> matrixSizes = {100, 1000, 10000, 50000, 100000};  
 vector<int> threadCounts = {4, 8, 16, 32, 64, 128, 256};  
  
 for (int n : matrixSizes) {  
 cout << "\n=== Розмір матриці: " << n << " x " << n << " ===\n";  
  
 // Створення початкової матриці (для даного розміру)  
 vector<vector<int>> mat = createRandomMatrix(n);  
  
 auto startTime = chrono::high\_resolution\_clock::*now*();  
 nonParallelSolution(mat);  
 auto endTime = chrono::high\_resolution\_clock::*now*();  
 chrono::duration<double> duration = endTime - startTime;  
 cout << "Послідовний час виконання: " << fixed << setprecision(6) << duration.count() << " секунд.\n";  
  
 for (int threads : threadCounts) {  
 vector<vector<int>> matParallel = mat;  
  
 auto startTime = chrono::high\_resolution\_clock::*now*();  
 parallelSolution(matParallel, threads);  
 auto endTime = chrono::high\_resolution\_clock::*now*();  
  
 chrono::duration<double> duration = endTime - startTime;  
 cout << "Паралельний час (потоків " << threads << "): " << fixed << setprecision(6) << duration.count() << " секунд.\n";  
 }  
 }  
  
 return 0;  
}

Перевіримо чи співпадає наведена характеристика системи в консолі з дійсністю:

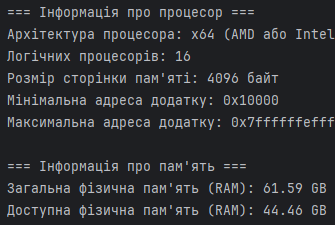
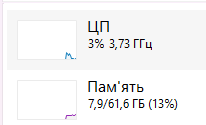
 

Рисунок 4 – Інформація системи з програми та диспетчеру завдань.

Отже, загальна фізична пам’ять та доступна фізична пам’ять співпадають.

Тепер подивимося на результати виконання завдання з матрицею, де використовується 4, 8, 16, 32, 64, 128 та 256 потоків:

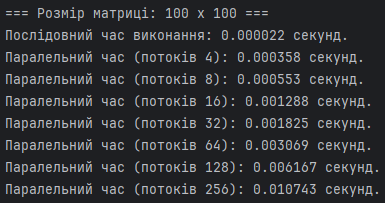


Рисунок 5 – Час для квадратної матриці розміру 100х100.

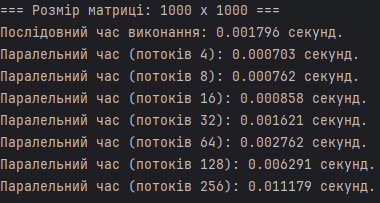


Рисунок 6 – Час для квадратної матриці розміру 1000х1000.

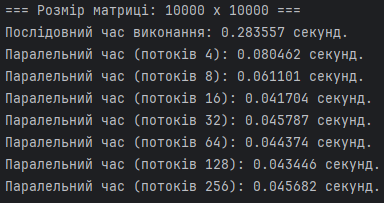


Рисунок 7 – Час для квадратної матриці розміру 10000х10000.

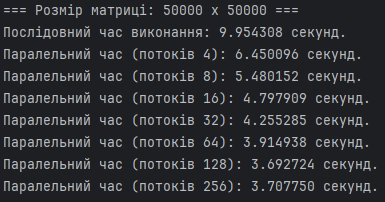


Рисунок 8 – Час для квадратної матриці розміру 50000х50000.

Зобразимо всі дані у вигляді таблиці та графіку:

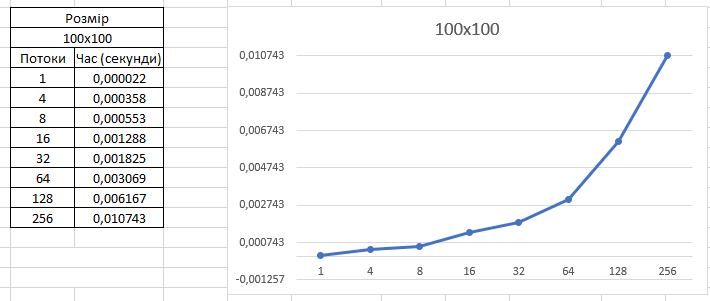


Рисунок 9 – Графік матриця розміром 100х100.

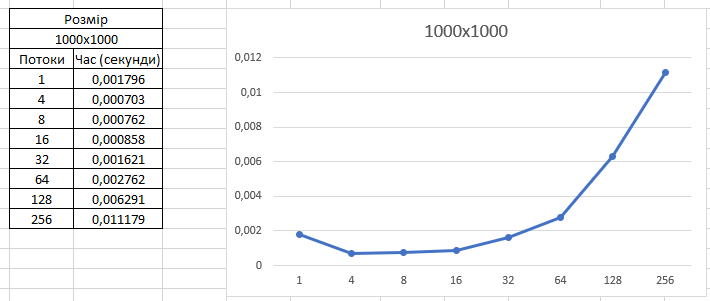


Рисунок 10 – Графік матриця розміром 1000х1000.

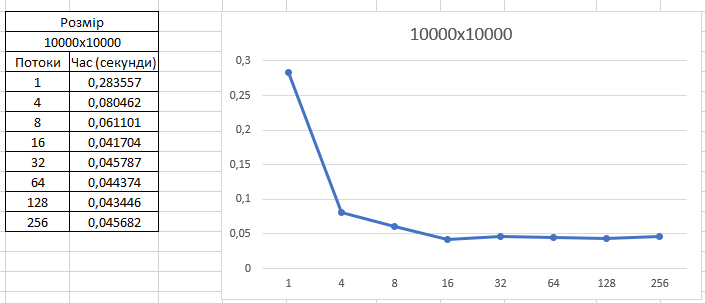


Рисунок 11 – Графік матриця розміром 10000х10000.

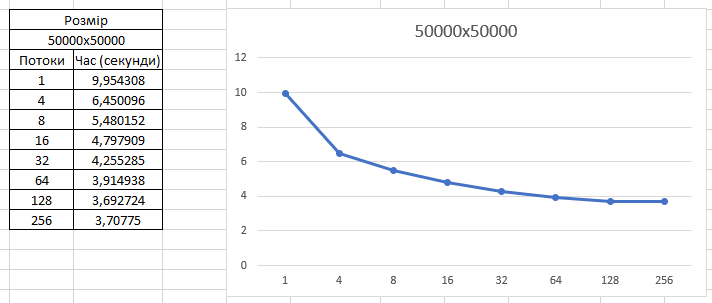


Рисунок 12 – Графік матриця розміром 50000х50000.

Як бачимо з рисунків графіків – чим більша матриця, тим кращий результат з використанням паралелізації. Це зумовлено тим, що для малих матриць обчислень дуже мало, тому витрати на створення потоків і їхню синхронізацію перевищують користь від паралелізації — і загальний час зростає. При збільшенні ж розміру матриці обчислювальна частина стає домінуючою, тож розподіл роботи між ядрами окупає витрати на паралелізацію і загальний час зменшується.

**Висновок**

У ході виконання лабораторної роботи було досліджено ефективність паралельної та послідовної реалізації обробки квадратної матриці. Було встановлено, що для малих розмірів матриць паралелізація не дає переваг, а іноді навіть погіршує результат через накладні витрати на створення та синхронізацію потоків. Натомість при збільшенні розміру матриці час виконання зменшується зі збільшенням кількості потоків, що свідчить про доцільність використання паралельних обчислень для задач із великою обчислювальною складністю. Таким чином, паралелізація є ефективною при значних обсягах даних.