

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«Київський політехнічний інститут ім. І. Сікорського»**

**Інститут Прикладного Системного Аналізу**

**Кафедра Системного Проектування**

Лабораторна робота №1

З дисципліни «Паралельні обчислення»

Виконав:

Ст. гр. ДА-12

Кракович Павло

**Лабораторна робота №1**

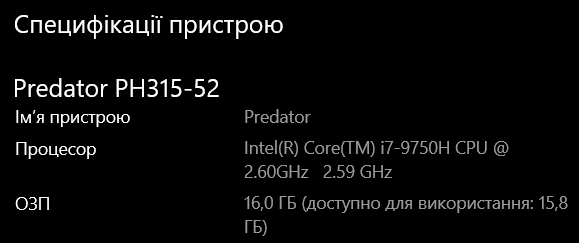
**Дослідження базових операцій з потоками виконання**

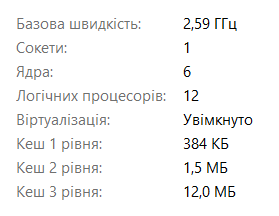
**Мета роботи:** Розглянути основні операції з потоками виконання, навчитися використовувати неблокуючу паралелізацію для вирішення найпростіших математичних задач, використовуючи обрану мову програмування. Навчитися досліджувати та оцінювати ефективність паралелізації алгоритму.

**Завдання:**

1. Визначити основні характеристики ПК, котрі вливають на ефективність виконання паралельних обчислень. Зафіксувати значення даних характеристик для ПК студента.
2. Створити механізм, котрий може бути використаний для заміру часу виконання програми, або інших параметрів, котрі студент вважає релевантними.
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 паралелізації. Заміряти час вирішення задачі, або інші параметри, котрі студент вважає релевантними. Обґрунтувати вибір алгоритму паралелізації.
5. Повторити пункт 4 з використання різної кількості процесів виконання. Обов’язково перевірити виконання задачі на фіксованих кількостях потоків: 2-рази меншій, ніж кількість фізичних ядер, на кількості рівній фізічним 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, в залежності від обраної задачі.
7. Заповнити таблицю й зробити графік часу виконання завдання від кількості потоків для різних розмірностей.

Характеристики пристрою:

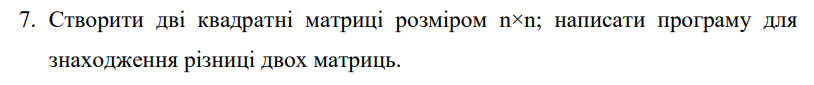




Основними для виконання завдання характеристиками є кількість ядер, потоків та їх швидкість.

Для заміру часу були використані методи, аналогічні прикладу у лабораторній роботі, а саме бібліотека chrono, яка дозволяє заміряти час за допомогою різниць часу, отриманого з часу системи.

Варіант завдання – №7



Для розбиття на паралелізм даних було використано айді потоку та їх кількість для визначення початкової та кінцевої точки обробки даних (віднімання). В залежності від кількості потоків, задача буде розбита на рівномірні частини між ними. В програмі не виконується копіювання, а реалізовано передачу посилання на контейнери

Табл. 1 – Виконання завдання за розмірності 3000 х 3000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Розмірність 3000 х 3000 | | | | |
| № | Використана паралелизація | Час створення потоку | Сер. час виконання потоку | Час виконання |
| 1 | Відсутня, один поток | 0,0001 | 0,0928 | 0,0937 |
| 2 | 3 потоки (1/2 ядер) | 0,0001 | 0,0321 | 0,0351 |
| 3 | 6 потоків | 0,0002 | 0,0191 | 0,0264 |
| 4 | 12 (рівне лог. ядрам) | 0,0003 | 0,0139 | 0,0159 |
| 5 | 24 | 0,0007 | 0,0074 | 0,0177 |
| 6 | 48 | 0,0016 | 0,0036 | 0,0184 |
| 7 | 96 | 0,0035 | 0,002 | 0,0197 |
| 8 | 192 | 0,0095 | 0,001 | 0,0238 |

Графік 1 – Виконання завдання за розмірності 3000 х 3000

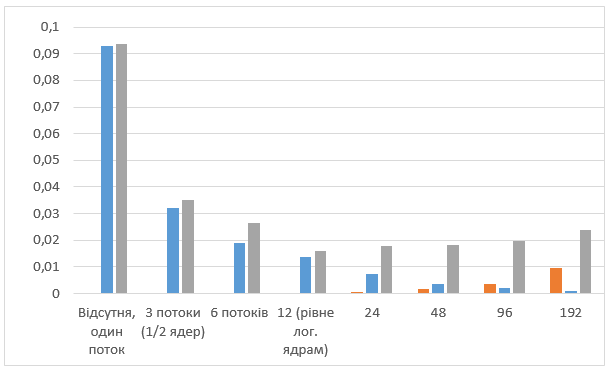
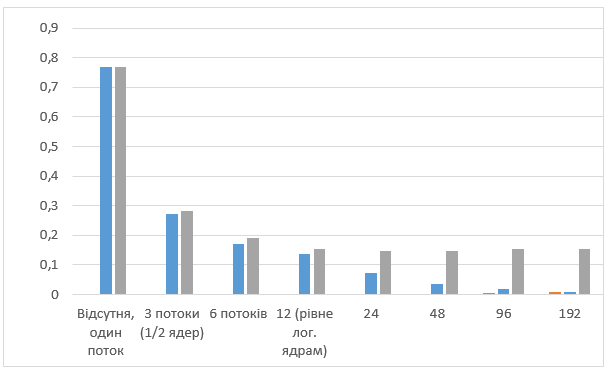


Табл. 2 – Виконання завдання за розмірності 9000 х 9000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Розмірність 9000 х 9000 | | | | |
| № | Використана паралелизація | Час створення потоку | Сер. час виконання потоку | Час виконання |
| 1 | Відсутня, один поток | 0,0001 | 0,7691 | 0,7701 |
| 2 | 3 потоки (1/2 ядер) | 0,0001 | 0,2737 | 0,2836 |
| 3 | 6 потоків | 0,0002 | 0,1709 | 0,1914 |
| 4 | 12 (рівне лог. ядрам) | 0,0003 | 0,1383 | 0,1538 |
| 5 | 24 | 0,0007 | 0,0713 | 0,1473 |
| 6 | 48 | 0,0016 | 0,0354 | 0,1463 |
| 7 | 96 | 0,0035 | 0,0177 | 0,1523 |
| 8 | 192 | 0,0095 | 0,0084 | 0,1529 |

Графік 2 – Виконання завдання за розмірності 9000 х 9000



Виконавши завдання за різної розмірності матриць можна впевнитись, що час створення потоків залишився незмінним, а середній час виконання потоку та час роботи завдання відносяться пропорційно до розмірності.

**Висновки**

Виконавши лабораторну роботу та проаналізувавши отримані результати можна зробити логічний висновок, що для оптимальної швидкості роботи програми слід використовувати кількіст потоків, рівну кілкості потоків процесору. Збільшення кількості віртуальних потоків на більшу трохи сповільнює програму загалом, оскільки потоки мають очікувати на можливість виконати свою частину завдання, хоча сама середня швидкість виконання, звісно, покращується.

Також, можна сказати, що час створення потоків незначним чином впливає на загальний час роботи. Цей вплив зменшується зі збільшенням розмірності (складності) завдання.

Оскільки робота була виконана в IDE Clion, не було простої можливості налаштувати конфігурацію, як в IDE VisualStudio, але запускаючи програму у режимі відладки можна спостерігати значне збільшення часу створення потоків, що може свідчити про те, що вимикається відповідна оптимізація для використання інструментів відладки.

Лістинг коду:

#include <iostream>  
#include <chrono>  
#include <thread>  
#include <vector>  
#include <cstdlib>  
#include <ctime>  
  
using std::chrono::nanoseconds;  
using std::chrono::duration\_cast;  
using std::chrono::high\_resolution\_clock;  
  
void printMatrix(const std::vector<std::vector<int>>& matrix)  
{  
 for (const auto& row : matrix) {  
 for (int value : row) {  
 std::cout << value << " ";  
 }  
 std::printf("\n");  
 }  
 std::printf("\n");  
}  
  
void fillRandomMatrix(std::vector<std::vector<int>>& matrix, int max\_value)  
{  
 int n = matrix.size();  
  
 for (int i = 0; i < n; ++i) {  
 for (int j = 0; j < n; ++j) {  
 matrix[i][j] = std::rand() % (max\_value + 1);  
 }  
 }  
}  
  
void compute(const std::size\_t thread\_id, const int n, std::vector<std::vector<int>>& result,  
 const std::vector<std::vector<int>>& matrix1, const std::vector<std::vector<int>>& matrix2,  
 const std::size\_t num\_threads, std::vector<double>& execution\_times)  
{  
 std::size\_t start\_row = thread\_id \* (n / num\_threads);  
 std::size\_t end\_row = (thread\_id == num\_threads - 1) ? n : (thread\_id + 1) \* (n / num\_threads);  
  
 auto payload\_begin = high\_resolution\_clock::now();  
  
 for (std::size\_t i = start\_row; i < end\_row; ++i) {  
 for (std::size\_t j = 0; j < n; ++j) {  
 result[i][j] = matrix1[i][j] - matrix2[i][j];  
 }  
 }  
  
 auto payload\_end = high\_resolution\_clock::now();  
 auto elapsed = duration\_cast<nanoseconds>(payload\_end - payload\_begin);  
 double elapsed\_seconds = elapsed.count() \* 1e-9;  
 execution\_times[thread\_id] = elapsed\_seconds;  
// std::printf("Thread %zu: Payload Time: %.3f seconds.\n", thread\_id, elapsed\_seconds);  
}  
  
int main()  
{  
 std::srand(std::time(0));  
 const int num\_threads = 192;  
 const int matrix\_size = 9000;  
  
 std::vector<std::vector<int>> result(matrix\_size, std::vector<int>(matrix\_size, 0));  
 std::vector<std::vector<int>> matrix1(matrix\_size, std::vector<int>(matrix\_size, 0));  
 std::vector<std::vector<int>> matrix2(matrix\_size, std::vector<int>(matrix\_size, 0));  
  
 fillRandomMatrix(matrix1, 9);  
 fillRandomMatrix(matrix2, 9);  
// printMatrix(matrix1);  
// printMatrix(matrix2);  
  
 std::vector<std::thread> threads;  
 std::vector<double> execution\_times(num\_threads, 0.0);  
  
 auto work\_begin = high\_resolution\_clock::now();  
 auto creation\_begin = high\_resolution\_clock::now();  
  
 for (int i = 0; i < num\_threads; i++) {  
 threads.emplace\_back(compute, i, matrix\_size, std::ref(result), std::ref(matrix1), std::ref(matrix2), num\_threads, std::ref(execution\_times));  
 }  
  
 auto creation\_end = high\_resolution\_clock::now();  
  
 for (auto& thread : threads) {  
 thread.join();  
 }  
  
 auto work\_end = high\_resolution\_clock::now();  
  
 double average\_execution\_time = 0.0;  
 for (const auto& time : execution\_times) {  
 average\_execution\_time += time;  
 }  
  
 auto elapsed\_creation = duration\_cast<nanoseconds>(creation\_end - creation\_begin);  
 double total\_creation\_time = elapsed\_creation.count() \* 1e-9;  
 std::printf("Creation time: %.4f seconds.\n", total\_creation\_time);  
  
 average\_execution\_time /= num\_threads;  
 std::printf("Average payload time for all threads: %.4f seconds.\n", average\_execution\_time);  
  
 auto elapsed\_work = duration\_cast<nanoseconds>(work\_end - work\_begin);  
 double total\_work\_time = elapsed\_work.count() \* 1e-9;  
 std::printf("Work time: %.4f seconds.\n", total\_work\_time);  
// std::printf("Result matrix:\n");  
// printMatrix(result);  
 return 0;  
}