**КПІ ім. Ігоря Сікорського**

**Інститут прикладного системного аналізу**

**Кафедра Системного проектування**

Лабороторна робота - 1

з дисципліни «Паралельні обчислення»

Виконав:

Студент групи ДА-12

Коваленко Олександр

Київ – 2024

**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 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:** 6. Створити дві квадратні матриці розміром n×n; написати програму для знаходження суми двох матриць.

**Хід роботи**

1. Процесор - Intel(R) Core(TM) i3 CPU 530 @ 2.93GHz 2.93 GHz

2 ядра 4 потоки

Оперативна пам’ять – 8 гб 1600 МГц

2. Я вирішив використовувати бібліотеку std::chrono, оскільки вона є стандартною і надійною, і має простий інтерфейс. Вона також дозволяє вимірювати час з високою точністю.

3. В якості контейнера для даних(матриці) використовується std::vector <short> matrix1(n \* n); він має розмір n \* n. Для розподілення даних між потоками необхідно поділити розмір на кількість потоків: n \* n/thread\_num – це кількість даних, які будуть оброблювати один потік. В потокі викликається функція яка приймає посилання на 2 матріци та індекси початку та кінця роботи для потоку. Індекси (початку,кінця) шукаються як (i \* data\_num, (i + 1) \* data\_num), де data\_num - це кількість даних, які будуть оброблювати один потік, а i – це номер потуку в циклу for.

4. Табличка часу виконання для різних розмірів матриці і кількості потоків.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Розмір матриці | | | | | |
| Кількість потоків |  | 1000 | 5000 | 10000 | 20000 | 40000 |
| 1 | 0,019 | 0,091 | 0,321 | 1,134 | 11,713 |
| 2 | 0,017 | 0,061 | 0,27 | 0,966 | 8,125 |
| 4 | 0,024 | 0,066 | 0,15 | 0,6104 | 4,455 |
| 8 | 0,029 | 0,0608 | 0,1576 | 0,38421 | 3,057 |
| 16 | 0,039 | 0,0945 | 0,17328 | 0,595 | 3,268 |
| 32 | 0,0905 | 0,1043 | 0,2315 | 0,5905 | 4,318 |
| 64 | 0,1427 | 0,1747 | 0,30715 | 0,824 | 5,698 |
| 128 | 0,2817 | 0,2857 | 0,4808 | 0,9211 | 6,0156 |

Графіки залежності часу від кількості потоків

Графіки залежності часу від кількості даних(розміру матриці)

Кількість потоків вибиралося з умов лабораторної роботи: Перевірити алгоритм на фіксованих кількостях потоків: 2-рази меншій (тобто 1), ніж кількість фізичних ядер, на кількості рівній фізичним ядрам( 2 ), на кількості рівній логічних ядрам( 4 ) , на кількості більшій в 2, 4, 8, 16 разів ніж кількість логічних ядер. ( і 8, 16, 32, 64, та я ще виришив перевірити 128 потоків)

5. На маленькій кількості даних ( n = 1000) кількість потоків від 1 до 16 не сильно впливає на час роботи, але коли потоків стає більше час створення і час перемикання контексту має досить великий вплив на час роботи.

На кількості даних ( n = 5000) картина приблизна така сама, тільки тепер кількість потоків від 1 до 8. Час перемикання контексту буде забирати багато часу, а так як в порівнянні з 1 випадком данних стало в 25 разів більше, то і перемикань стало в стільки ж разів більше (на великій кількості потоків), що сутєво впливає на час виконання.

На кількості даних ( n = 10000) час роботи найменьший є коли потоків 4,8 – тут картина приблизно така сама як у випадку 2.

На кількості даних ( n = 20000 та n = 40000 ) час роботи найменьший коли потоків 8 – це може бути пов’язано з тим, що програма не може забрати всю потужність комп’ютера, ос має свої потоки які працюють паралельно з програмою, тому коли потоків в 2 рази більше ніж логічних ядер, менше шансів, що ос забере собі необхідні ресурси і потоків не настільки багато, щоб часто було перемикання контексту виконання.

Час залежить від часу приблизно лінійно на розмірах від 1000 до 20 000, а на розмірі в 40 000 є велике сповільння, бо не вистачає ооперативної пам’яті бо 2 матриці такого розміру будуть займати приблизно 6,5 гб.

6. В програмі є проблема з когерентністю кеш рядків, дані які оброблює один потік можуть бути з даними іншого потоку в одному кеш рядку.

7. Висновок: була написана многопоточна програма для знаходження суми двох матриць на с++. В якості контейнера для матриці був обраний vector, тому що він щберігає дані в купі, що є важливим для багатопотчної програми, бо в стеку неможливо створити великий контейнер, крім того в векторі є можливість швидко звертатися до елементів, які лежать різних в різних рядках на відміну від реалізації **int\*\*m(**m[2] = new int\*), бо коли будемо створювати масиви всередені масивів нема гарантіі, що вони будуть лежати близько один к одному. Було розроблено алгоритм поділу даних між потокома: було вибраний саме такий алгоритм(дані ділимо на приблизно однакові розміри даних і для кожного потоку вони свої і ідуть в vector один за одним, наприклад є масив 10 на 10 і 4 потоки, тобто всього 100 елементів, на кожний потік по 25, 1 потік буде оброблювати дані з 0 по 25), бо він є зручний і зрозумілий, просто знайти індекси для кожного потоку. Сама функція, яку передають в потік дуже проста: в циклі for з індексу початку до індексу кінця сумує елементи матриць. Був підрахован час створення та виконання потоків за допомогою бібліотеки chrono. Програма краще всього себе показує на кількості потоків , які дорівнює 4 та 8 (кількість логічних ядер та в число 2 рази більше).

**Додаток**

**Код:**

#include <iostream>

#include <thread>

#include <chrono>

#include <vector>

const int n = 40000;

const int thread\_num =128;

const int data\_num = n\*n / thread\_num;

void calculate\_amount(std::vector<short>& matrix1, std::vector<short>& matrix2, int start, int finish)

{

for (int i = start; i < finish; i++)

{

matrix1[i] += matrix2[i];

}

}

void calculate\_amount1(int \*a, int \*b, int num)

{

for (int i = 0; i < num; i++)

{

\*a = \*a + \*b;

a = a + 1;

b = b + 1;

}

}

void fill\_matrix(std::vector<short>& matrix)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

matrix[n \* i + j] = j;

}

}

}

void print\_matrix(std::vector<short>& matrix)

{

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

std::cout << matrix[n\*i+j] << " ";

}

std::cout << "\n";

}

}

int main()

{

//int\*\* matrix1 = new int\*[n];

std::vector<short> matrix1(n \* n);

fill\_matrix(matrix1);

// print\_matrix(matrix1);

// std::cout << "\n\n";

//int\*\* matrix2 = new int\* [n];

std::vector<short> matrix2(n \* n);

fill\_matrix(matrix2);

// print\_matrix(matrix2);

// std::cout << "\n\n";

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

std::vector<std::thread> threads;

threads.reserve(thread\_num);

for (int i = 0; i < thread\_num-1; i++)

{

threads.emplace\_back(calculate\_amount, std::ref(matrix1), std::ref(matrix2), i \* data\_num, (i + 1) \* data\_num);

// threads.emplace\_back(calculate\_amount1, &(matrix1[i \* data\_num]), &(matrix2[i \* data\_num]), data\_num);

}

threads.emplace\_back(calculate\_amount, std::ref(matrix1), std::ref(matrix2), (thread\_num - 1) \* data\_num, n\*n);

auto finish\_creation = std::chrono::high\_resolution\_clock::now();

// threads[thread\_num-1] = std::thread(calculate\_amount, matrix1, matrix2, (thread\_num - 1 )\* data\_num, n);

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

for (int i = 0; i < thread\_num; i++)

{

threads[i].join();

}

auto finish\_working = std::chrono::high\_resolution\_clock::now();

auto creating\_time = std::chrono::duration\_cast<std::chrono::nanoseconds>(finish\_creation - start\_creation);

auto working\_time = std::chrono::duration\_cast<std::chrono::nanoseconds>(finish\_working - start\_working);

auto all\_time = std::chrono::duration\_cast<std::chrono::nanoseconds>(finish\_working - start\_creation);

std::printf(" Creation time: %.7f seconds.\n", creating\_time.count() \* 1e-9);

std::printf(" Function time: %.5f seconds.\n", working\_time.count() \* 1e-9);

std::printf(" All time: %.5f seconds.\n", all\_time.count() \* 1e-9);

// std::cout << "\n\n";

// print\_matrix(matrix1);

return 0;

}