КПІ ім. Ігоря Сікорського

Інститут прикладного системного аналізу

Кафедра Системного проектування

Лабораторна робота №1

З курсу «Паралельні обчисення»

З теми: «Дослідження базових операцій з потоками виконання»

Виконав:

Студент 3-го курсу

групи ДА-21

Сімков Микита

Варіант 20

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.

**Хід роботи**

1. При визначенні важливих характеристик обов'язково потрібно взяти: всі характеристики виконавчих юнітів (CPU, GPU): тактова частота, кількість ядер, кількість потоків, розмір L1, L2, L3 кешів, архітектура паралельної обробки (CMT, SMT) – для CPU, кількість CUDA/Stream процесорів, об’єм і тип відеопам’яті (VRAM, GDDR6, HBM2), пропускна здатність пам’яті – для GPU. Також впливають параметри оперативної пам'яті: об'єм пам'яті дає працювати з алгоритмами з більшою просторовою складністю, пропускна здатність та час затримки пришвидшують отримання даних.

В даній роботі обробкою даних буде займатись CPU тож основний вплив будуть мати характеристики саме процесора :

1. AMD Ryzen 5 5600X має такі основні характеристики:
2. Кількість ядер / потоків: 6 ядер / 12 потоків
3. Тип багатопотоковості: SMT
4. Базова частота: 3.7 ГГц
5. Максимальна частота Boost: 4.6 ГГц
6. Кеш-пам’ять: L2 – 3 МБ, L3 – 32 МБ
7. Підтримка інструкцій: AVX2, FMA3, SSE4a, SHA
8. Архітектура: Zen 3
9. Техпроцес: 7 нм

Також через великий об’єм матриць на виконання впливають характеристики оперативної пам’яті, а саме:

Частота:

1. 2 планки по 2400
2. 2 планки по 3200

Та об’єм (щоб не потрібно було використовувати файл довантаження):

Всі 4 планки мають об’єм 8GB.

2. Для вимірювання часу виконання коду було обрано chrono, оскільки цей метод забезпечує високу точність і є стандартним інструментом C++ для отримання точних часових замірів. Для замірів використання процесора та пам’яті (що є не менш важливими параметрами в оптимізації) буде використано вбудований в Visual Studio diagnostic tools

Замір часу виконується тільки для тієї частини коду яка відповідає за обчислення та відрізняється у 2 реалізацій (матриці повністю ідентичні та створюються окремо)

3. Рішення без паралелізації виглядає дуже простим

void defaultprocessMatrix(vector<vector<short int>>& matrix) {

    for (int i = 0; i < SIZE; ++i) {

        int minIndex = 0;

        for (int j = 1; j < SIZE; ++j) {

            if (matrix[i][j] < matrix[i][minIndex]) {

                minIndex = j;

            }

        }

        swap(matrix[i][SIZE - 1 - i], matrix[i][minIndex]);

    }

}

Простий перебір всіх n рядків за n дій кожен, що дає асимптотику

Заміри часу будуть зроблені для 500, 10000 та 50000 елементів, по 2 рази кожен. Всі дані про час виконання будуть занесені в таблиці, також якщо профайлер покаже якісь корисні в аналізі дані вони теж будуть додані до протоколу.

4-6. Розподілення потоків по рядкахя робив за таким принципом: якщо в нас є n потоків, то кожен i-тий потік буде обробляти рядки: і, i+n, i+2n і так до кінця матриці. Вважаю цей підхід релевантним, бо технічно ми реалізуємо не ,,паралельне програмування,, а ,,паралельне обчислення,, бо в моєму завданні дані, з якими працюють потоки взагалі не пересікаються (різні рядки матриці). Тож така реалізація рівномірно розподілить навантаження серед потоків.

void processRows(vector<vector<short int>>& matrix, int threadID, int numThreads) {

    for (int i = threadID; i < SIZE; i += numThreads) {

        int minIndex = 0;

        for (int j = 1; j < SIZE; ++j) {

            if (matrix[i][j] < matrix[i][minIndex]) {

                minIndex = j;

            }

        }

        swap(matrix[i][SIZE - 1 - i], matrix[i][minIndex]);

    }

}

// Функція для запуску паралельної обробки

void processMatrixParallel(vector<vector<short int>>& matrix, int numThreads) {

    vector<thread> threads;

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

        threads.emplace\_back(processRows, ref(matrix), i, numThreads);

    }

    for (auto& t : threads) {

        t.join();

    }

}

Створено функцію, що запускає потоки та функцію яку кожен запущений потік має виконати. Кожному потоку дається майже рівна кількість рядків для обробки. По аналогії до попереднього пункту заміри будуть виконані для 500, 10000 та 50000 елементів для 3, 6, 12, 24, 48... потоків.

Почнемо з запису результатів послідовної реалізації:

Для дебаг моду з профайлером:

Результати:

При 1000: 15мс 16мс.

При 10000: 1547мс, 1534мс + 194мб

При 50000: 38252мс, 37983мс + 4.7GB пам’яті.

Тепер можна замірити в релізі:

При 1000 елементів: стабільно 0мс.

При 10000: 50мс, 52мс.

При 50000: 1329мс, 1361мс.

Кожен запуск потребував рівно 8% CPU.

Як можна побачити, для дебагу, отримані дані співпадають з теоретичними якщо взяти, що дій 1 потік робить за 15-16мс, то буде в разів довше, тобто 1500-1600мс, що повністю відповідає замірам. Таке співпадіння спричинене тим, що 1 потік – 8% від всієї потужності процесора (1/12) працював майже протягом усього виконання на всі 8%.

Тепер буду тестувати паралельну реалізацію в реліз моді. Для початку для 3 потоків

3 потоки:

При 1000 елементів: 2мс, 3мс.

При 10000 елементів: 17мс, 17мс.

При 50000 елементів: 477мс, 454мс.

Використовує стільки ж пам’яті як і в послідовному.

З процесором трохи цікавіше. Для малих даних (1000, 10000 елементів) процесор ледве ледве встигає почати розгін і в піку використовується +-10% (бо потоки які створюються раніше можуть завершити роботу ще до того як створяться всі інші). Для 50000 елементів часу на розгін достатньо і в піку він використвоує 25% процесора, тобто ¼ всіх потоків.

6 потоків:

При 1000 елементів: 2мс, 3мс.

При 10000 елементів: 12мс, 14мс.

При 50000 елементів: 323мс, 292мс.

Все аналогічно до 3 потоків, проте час на обробку зменшився. Аналогічно, для 50000 елементів процесор встигає дійти до 50% використання.

12 потоків:

При 1000 елементів: 4мс, 6мс.

При 10000 елементів: 13мс, 12мс.

При 50000 елементів: 232мс, 223мс.

Ситуацій повністю повторилась для великих обчислень. Час ще скоротився, проте тепер відчутні пролагування під час обробки і прцоесор в якийсь момент встає в 100%. Проте що стосується 1000 елементів, тепер створення потоку виконується довше ніж виконання цього потоку.

24 потоки:

При 1000 елементів: 7мс, 7мс.

При 10000 елементів: 15мс, 16мс.

При 50000 елементів: 240мс, 228мс.

Прослідковуються ті ж тенденції що і для 12 потоків, проте для 24 потоків процесор навантажувався максимум на 64%, хоча це, напевне можна списати на випадковість.

48 потоків:

При 1000 елементів: 15мс, 11мс.

При 10000 елементів: 21мс, 20мс.

При 50000 елементів: 233мс, 234мс.

Прослідковуються ті ж тенденції.

96 потоків:

При 1000 елементів: 31мс, 31мс.

При 10000 елементів: 33мс, 32мс.

При 50000 елементів: 262мс, 243мс.

Прослідковуються ті ж тенденції.

192 потоки:

При 1000 елементів: 48мс, 46мс.

При 10000 елементів: 54мс, 51мс.

При 50000 елементів: 290мс, 272мс.

Прослідковуються ті ж тенденції.

7.

Табличка результатів:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | один | Фіз/2 | фіз | фіз \* 2 | фіз \* 4 | фіз \* 8 | фіз \* 16 |
| 1000 | 0 | 2.5 | 2.5 | 5 | 7 | 31 | 47 |
| 10000 | 51 | 17 | 13 | 12.5 | 15.5 | 32.5 | 52.5 |
| 50000 | 1340 | 465 | 307.5 | 227.5 | 234 | 252.5 | 281 |

A graph with lines and numbers

AI-generated content may be incorrect.

8.

Підсумовуючи: Один потік миттєво виконав обрахунок для маленької матриці та перегнав всі інші реалізації. Це пов’язано з тим що маленьку матрицю можна досить швидко обробити, а створення потоку та перемикання між ними займає якісь мілісекунди. Тож для 1000 елементів - чим потоків більше, тим гірше результат, бо вони створюються значно довше ніж працюють.

Для інших же тестів кількість даних достатня, щоб побачити головний плюс паралелизації. Для 10000 елементів вже прослідковується явний приріст в швидкодії аж в 4 рази. При цьому процесор не встигав розгонятись до всіх 100% навантаження, думаю це зумовлено тим, що потоки, які створюються першими виконують всю роботу швидше ніж програма встигає створити всі потоки. Як можна побачити з таблиці, тут також прослідковується тенденція, що завелика кількість потоків радше сповільнює програму.

Для 50000 елементів, вже можна було побачити всю потужність паралелизації. Процесор іноді забивався на всі 100% та можна було побачити провисання. Це хороший знак, адже це означає що наша програма використала для обчислень всі логічні ядра одразу. Тут, аналогічно, оптимальна кількість потоків знаходиться в межах кількості логічних ядер.

Щодо всіх обрахунків: можна зробити вивід, щоб оптимальна кількість потоків це +- кількість логічних ядер комп’ютера, оскільки тоді комп’ютер і використовує все навантаженн, бо при малій кількості потоків комп’ютер просто не використовує всю свою потужність, а при завеликій кількості забагато накладних витрат на перемикання контексту та конкуренцію за ресурси.

Щодо отриманих результатів, не всюди можна чітко виділити де саме оптимальна кілкьість ядер оскільки:

1. Впливає фактор випадковості: процесор може в якийсь момент бути зайняти якимось випадковим процесом в системі і пізніше почати обробляти нашу програму, що додасть зайві долі секунди до вимірів
2. В моєму випадку також впливає низька частота оперативної пам’яті: для завдання де є дуже велика матрция але з нею потрібно зробити всього дій, навіть мій процесор з доволі великим розміром кешу буде витрачати велику кількість часу на банальне отримання даних з пам’яті.
3. Також обчислння можуть бути сповільнені архітектурою паралелизації: так як в моєму процесорі архітектура SMT, потоки при повному завантаженні можуть працювати повільніше, так як йде більша боротьба за ресурси комп’ютера і деякі потоки можуть простоювати, що збільшує час виконання при такому рівномірному розподілі навантаження, але це компенсуєтьсфя повільною оперативкою.