МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА

Лабораторна робота №6

“Алгоритм Дейкстри”

Виконав:
студент групи ПМі-31
Дудинець Олександр

Львів 2024

Мета роботи

Метою цієї лабораторної роботи було розробити програму для знаходження найкоротших шляхів від заданої вершини до всіх інших вершин у зваженому графі за допомогою алгоритму Дейкстри. Завдання включало реалізацію як послідовного, так і паралельного варіантів алгоритму. Також потрібно було обчислити час виконання кожного алгоритму, визначити прискорення та ефективність паралельного алгоритму для різної кількості потоків та розмірів графів.

Опис ключових частин коду

1. **Генерація графу (generateGraphParallel)**
    a. Паралельно створює випадковий зважений граф у вигляді матриці суміжності.
    b. Реалізація:
        i. Створюється двовимірний масив adjMatrix розміром numVertices x numVertices.
        ii. Кожен елемент adjMatrix[i][j] представляє вагу ребра між вершинами i та j.
2. **Послідовна реалізація алгоритму Дейкстри (dijkstraSequential)**
    a. Знаходить найкоротші шляхи від заданої вершини до всіх інших вершин у графі.
    b. Реалізація:
        i. Ініціалізація:
            1. Масив dist[] заповнюється значеннями INT_MAX (нескінченність), окрім початкової вершини, для якої встановлюється 0.
            2. Масив sptSet[] використовується для відстеження оброблених вершин.
        ii. Основний цикл:
            1. Повторюється для всіх вершин:
                a. Вибирається вершина u з мінімальною відстанню, яка ще не оброблена.
                b. Вершина u позначається як оброблена.
                c. Для всіх сусідніх вершин v оновлюється dist[v], якщо шлях через u коротший.

3. **Паралельна реалізація алгоритму Дейкстри (dijkstraParallel)**
    a. Прискорює виконання алгоритму шляхом паралельної обробки релаксації ребер (процес оновлення відстані до вершини, якщо знайдено коротший шлях через іншу вершину).
    b. Реалізація:
        i. Ініціалізація:
            1. Аналогічна послідовному алгоритму.
        ii. Основний цикл:
            1. Після вибору вершини u з мінімальною відстанню, робота з релаксації ребер розподіляється між потоками.
            2. Кожен потік обробляє свою підмножину вершин v та оновлює dist[v], якщо знаходить коротший шлях через u.
            3. Використовується mutex для захисту оновлення масиву dist[], запобігаючи, так званим, race conditions.
4. **Використання mutex**
    a. Забезпечує синхронізований доступ до спільного ресурсу dist[].
    b. Реалізація:
        i. Перед оновленням dist[v], потік захоплює mutex за допомогою lock_guard<mutex> lock(distMutex);.
        ii. Це гарантує, що лише один потік може оновлювати dist[] у певний момент часу.
5. **Перевірка еквівалентності результатів (areArraysEqual)**
    a. Перевіряє, чи масиви відстаней, отримані послідовним та паралельним алгоритмами, є ідентичними.
    b. Реалізація:
        i. Порівнює відповідні елементи двох масивів dist[] по всіх індексах i.
6. **Обчислення прискорення та ефективності**
    a. Функція calculateSpeedup: Обчислює прискорення як відношення часу виконання послідовного алгоритму до часу паралельного.
    b. Функція calculateEfficiency: Обчислює ефективність як відношення прискорення до кількості потоків.

7. **Аналіз (benchmark)**
    a. Запускає обидва алгоритми, вимірює час їх виконання та аналізує результати.
    b. Реалізація:
        i. Генерація графу:
            1. Використовується паралельна генерація графу для пришвидшення процесу при великих розмірах, оскільки послідовна генерація великих графів (50 000+ вершин) займала багато часу.
        ii. Виконання алгоритмів:
            1. Виконує послідовний та/або паралельний алгоритми залежно від параметрів.
            2. Вимірює час виконання кожного алгоритму.
            3. Виводить результати та обчислює прискорення й ефективність.
8. **Параметри командного рядка:**
    a. Формат: <numVertices> <threads> <sourceNode> [runSequential] [runParallel].
    b. Пояснення:
        i. numVertices — кількість вершин у графі.
        ii. threads — кількість потоків для паралельного алгоритму.
        iii. sourceNode — початкова вершина для алгоритму Дейкстри.
        iv. runSequential — прапорець для запуску послідовного алгоритму (1 — запустити, 0 — пропустити).
        v. runParallel — прапорець для запуску паралельного алгоритму.

Аналіз результатів

Програма була протестована на графах з 100 000, 50 000, 10 000 вершин з використанням 16, 9, 8, 500 потоків. Тестування проводилося на комп'ютері з процесором Apple M1 Pro (8 високопродуктивних ядер і 2 енергоефективні ядра).

Обидва алгоритми (послідовний та паралельний) були запущені на однакових графах, що дозволило порівняти результати та переконатися у їхній коректності. Результати обчислення найкоротшого шляху для обох варіантів алгоритмів були однаковими.

Граф з 100 000 вершин:

- **8 потоків:**

```
~/Developer/University/Parallel-Computing-Course-Assignments/lab-6 main ?2 ·····
❯ ./run.sh 100000 8 0
Benchmark for Dijkstra algorithm with 100000 nodes and 8 threads:

=====================

- Generating graph...

Graph generated in 32394ms

Press any key to continue...

=====================

- Running sequential algorithm:
   - Time: 80482ms

- Running parallel algorithm:
   - Time: 38974ms

=====================

- Summary:
   - Nodes: 100000
   - Threads: 8
   - Source Node: 0
   - Sequential time: 80482ms
   - Parallel time: 38974ms
   - Speedup: 2.06502x
   - Efficiency: 25% (took 38974ms vs 10060ms ideal)
   - Shortest paths length equal: Yes
```

Прискорення складає приблизно 2x, що свідчить про ефективне використання паралельних обчислень.

Ефективність 25% вказує на те, що не весь потенціал паралелізації використаний, через витрати на синхронізацію та управління потоками.

- **16 потоків:**

```
- Summary:
   - Nodes: 100000
   - Threads: 16
   - Source Node: 0
   - Sequential time: 83683ms
   - Parallel time: 43792ms
   - Speedup: 1.91092x
   - Efficiency: 11% (took 43792ms vs 5230ms ideal)
   - Shortest paths length equal: Yes
```

Збільшення кількості потоків до 16 не призвело до значного покращення, а ефективність знизилася до 11%.

- **500 потоків:**

```
- Summary:
   - Nodes: 100000
   - Threads: 500
   - Source Node: 0
   - Sequential time: 82990ms
   - Parallel time: 744273ms
   - Speedup: 0.111505x
   - Efficiency: 0% (took 744273ms vs 165ms ideal)
   - Shortest paths length equal: Yes
```

Використання 500 потоків призвело до значного погіршення продуктивності.

Витрати ресурсів на створення та управління великою кількістю потоків перевищують будь-яку вигоду від паралелізації.

Ефективність практично нульова, що свідчить про недоцільність використання такої кількості потоків.

Граф з 10 000 вершин:

- **8 потоків:**

```
- Summary:
   - Nodes: 10000
   - Threads: 8
   - Source Node: 0
   - Sequential time: 549ms
   - Parallel time: 1156ms
   - Speedup: 0.474913x
   - Efficiency: 5% (took 1156ms vs 68ms ideal)
   - Shortest paths length equal: Yes
```

  На малих графах паралельний алгоритм працює повільніше за послідовний.

  Витрати на управління потоками не компенсуються прискоренням від паралельної обробки.

  Ефективність низька через невелику кількість обчислень у порівнянні з накладними витратами.

- **16 потоків:**

```
- Summary:
   - Nodes: 10000
   - Threads: 16
   - Source Node: 0
   - Sequential time: 556ms
   - Parallel time: 2172ms
   - Speedup: 0.255985x
   - Efficiency: 1% (took 2172ms vs 34ms ideal)
   - Shortest paths length equal: Yes
```

  Подальше збільшення кількості потоків призводить до ще більшого зниження продуктивності.

Граф з 50 000 вершин та некратна кількість потоків:

- **9 потоків:**

```
- Summary:
   - Nodes: 50000
   - Threads: 9
   - Source Node: 0
   - Sequential time: 16099ms
   - Parallel time: 12424ms
   - Speedup: 1.2958x
   - Efficiency: 14% (took 12424ms vs 1788ms ideal)
   - Shortest paths length equal: Yes
```

Алгоритм працює коректно при некратній кількості потоків.

Залишкові вершини рівномірно розподіляються між потоками, забезпечуючи баланс навантаження.

Прискорення є, але ефективність все ще обмежена накладними витратами.

Граф з 500 вершин:

- **8 потоків:**

```
- Summary:
   - Nodes: 500
   - Threads: 8
   - Source Node: 0
   - Sequential time: 2ms
   - Parallel time: 53ms
   - Speedup: 0.0377358x
   - Efficiency: 0% (took 53ms vs 0ms ideal)
   - Shortest paths length equal: Yes
```

На малих графах паралельний алгоритм працює повільніше за послідовний.

Витрати на управління потоками не компенсуються прискоренням від паралельної обробки.

Ефективність низька через невелику кількість обчислень у порівнянні з накладними витратами.

Загальні висновки з результатів:

Ефективність паралельного алгоритму залежить від розміру графу та кількості потоків.

Оптимальна кількість потоків зазвичай відповідає кількості фізичних або логічних ядер процесора.

Збільшення кількості потоків понад апаратні можливості призводить до зниження продуктивності через контекстне перемикання та накладні витрати.

Для малих графів паралельний алгоритм може бути менш ефективним через значні накладні витрати відносно загального часу виконання.

Алгоритм коректно працює при некратній кількості потоків, що підтверджується ідентичністю результатів та відсутністю помилок.

Висновок

У цій лабораторній роботі були реалізовані як послідовний, так і паралельний варіанти алгоритму Дейкстри для знаходження найкоротших шляхів у зваженому графі. Паралельний алгоритм продемонстрував значне прискорення на великих графах при оптимальній кількості потоків, що підтверджує доцільність використання паралельних обчислень для задач такого типу.

Під час роботи я поглибив свої знання з паралельного програмування на мові C++, зокрема з використанням механізмів синхронізації mutex. Я також ознайомився з особливостями паралелізації алгоритмів, які мають послідовні залежності.

Було вирішено паралельно генерувати граф, оскільки послідовна генерація великих графів (50 000+ вершин) займала багато часу.

Однак ефективність паралельного алгоритму була нижчою за очікувану на графах менших за 50 000 вершин через витрати на управління потоками та обмеження, пов'язані з послідовними частинами алгоритму.