

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА

Лабораторна робота №7
“Алгоритм Прима”

Виконав:
студент групи ПМі-31
Дудинець Олександр

Львів 2024

Мета роботи

Метою цієї лабораторної роботи було розробити програму для побудови мінімального кісткового дерева у зваженому зв'язному неорієнтованому графі за допомогою алгоритму Прима. Завдання включало реалізацію як послідовного, так і паралельного варіантів алгоритму, а також оцінку часу виконання кожного з них, прискорення та ефективності 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. Спочатку забезпечується зв'язність графа шляхом створення випадкового кісткового дерева, а потім додаються додаткові ребра.
- b. Реалізація:
 - i. Створюється двовимірний масив `graph` розміром $n \times n$, де n – кількість вершин у графі.
 - ii. З'єднуються вершини для забезпечення зв'язності.
 - iii. Додаються додаткові ребра для випадкових пар вершин.

2. Послідовна реалізація алгоритму Прима (`primSequential`)

- a. Знаходить мінімальне кісткове дерево, починаючи з заданої вершини.
- b. Реалізація:
 - i. Ініціалізуються масиви `key` (для відстеження найменшої ваги) та `inMST` (для відмітки вершин, що вже включені до MST).
 - ii. Вибирається вершина з найменшим значенням ключа, яка ще не в MST.
 - iii. Для кожної сусідньої вершини оновлюється ключ та батько, якщо знайдено меншу вагу.

3. Паралельна реалізація алгоритму Прима (`primParallel`)

- a. Подібно до послідовного алгоритму, але кожна ітерація розподіляється між потоками.
- b. Потоки виконують паралельний пошук мінімальної ваги у своїх діапазонах.

- с. Для оновлення масивів `key` та `parent` використовується механізм синхронізації за допомогою `mutex`.

4. Перевірка еквівалентності результатів (`areArraysEqual`)

- а. Перевіряє, чи масиви відстаней, отримані послідовним та паралельним алгоритмами, є ідентичними.
- б. Реалізація:
 - і. Порівнює відповідні елементи двох масивів по всіх індексах `i`.

5. Обчислення прискорення та ефективності

- а. Функція `calculateSpeedup`: Обчислює прискорення як відношення часу виконання послідовного алгоритму до часу паралельного.
- б. Функція `calculateEfficiency`: Обчислює ефективність як відношення прискорення до кількості потоків.

6. Аналіз (`benchmark`)

- а. Запускає обидва алгоритми, вимірює час їх виконання та аналізує результати.
- б. Реалізація:
 - і. Генерує граф.
 - ii. Виконує послідовний та/або паралельний алгоритми залежно від параметрів.
 - iii. Вимірює час виконання кожного алгоритму.
 - iv. Виводить результати та обчислює прискорення й ефективність.

7. Параметри командного рядка:

- а. Формат: `<numVertices> <threads> <sourceNode> [runSequential] [runParallel]`.
- б. Пояснення:
 - і. `numVertices` — кількість вершин у графі.
 - ii. `threads` — кількість потоків для паралельного алгоритму.
 - iii. `sourceNode` — початкова вершина для алгоритму Прима.
 - 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 різних розмірів (від 500 до 100 000 вершин) з використанням різної кількості потоків (від 3 до 16). Тестування проводилося на комп'ютері з процесором Apple M1 Pro (8 високопродуктивних ядер і 2 енергоефективні ядра).

Обидва алгоритми (послідовний та паралельний) були запуснені 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:

```
- Summary:
  - Nodes: 100000
  - Threads: 8
  - Source Node: 3
  - Sequential time: 61030ms
  - Parallel time: 52521ms
  - Speedup: 1.16201x
  - Efficiency: 14% (took 52521ms vs 7628ms ideal)
  - MST equal: Yes
```

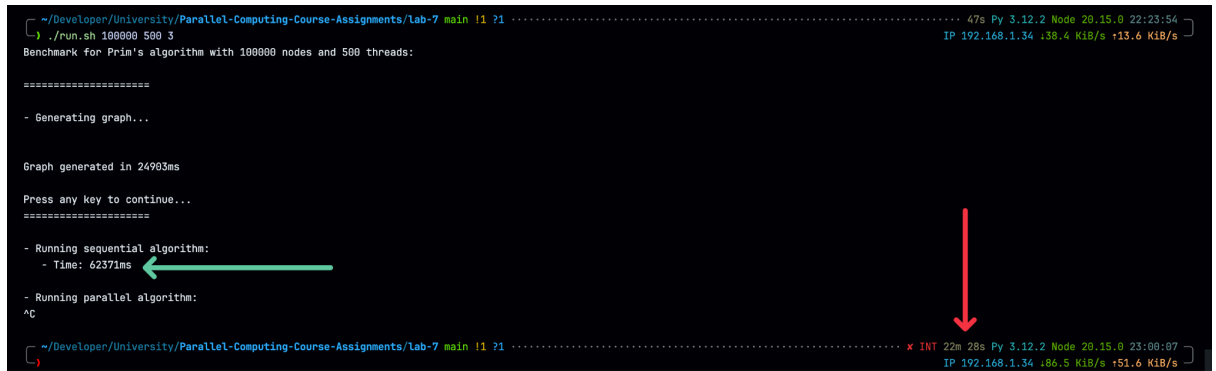
Прискорення складає 1.16x, що свідчить про певне покращення, проте ефективність становить лише 14%. Це пов'язано з великими накладними витратами на синхронізацію потоків.

- 16 потоків:

```
- Summary:
  - Nodes: 100000
  - Threads: 16
  - Source Node: 3
  - Sequential time: 60102ms
  - Parallel time: 61895ms
  - Speedup: 0.971032x
  - Efficiency: 6% (took 61895ms vs 3756ms ideal)
  - MST equal: Yes
```

Збільшення кількості потоків не призвело до значного покращення, ефективність знизилася до 6%, що свідчить про те, що додаткові потоки не використовуються ефективно.

- **500 потоків:**



```
~/Developer/University/Parallel-Computing-Course-Assignments/Lab-7 main |1 ?1 ..... 47s Py 3.12.2 Node 20.15.0 22:23:54
) ./run.sh 100000 500 3 IP 192.168.1.34 +38.4 KiB/s +13.6 KiB/s
Benchmark for Prim's algorithm with 100000 nodes and 500 threads:

=====
- Generating graph...

Graph generated in 24903ms

Press any key to continue...
=====
- Running sequential algorithm:
  - Time: 62371ms ←
- Running parallel algorithm:
  ^C
x INT 22m 28s Py 3.12.2 Node 20.15.0 23:00:07
IP 192.168.1.34 +86.5 KiB/s +51.6 KiB/s
```

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 50 000 вершин:

- 8 потоків:

```
- Summary:  
  - Nodes: 50000  
  - Threads: 8  
  - Source Node: 3  
  - Sequential time: 15725ms  
  - Parallel time: 18530ms  
  - Speedup: 0.848624x  
  - Efficiency: 10% (took 18530ms vs 1965ms ideal)  
  - MST equal: Yes
```

Паралельний алгоритм працює гірше за послідовний, оскільки витрати на синхронізацію та управління потоками перевищують вигоду від паралелізації.

- 16 потоків:

```
- Summary:  
  - Nodes: 50000  
  - Threads: 16  
  - Source Node: 3  
  - Sequential time: 15287ms  
  - Parallel time: 25480ms  
  - Speedup: 0.599961x  
  - Efficiency: 3% (took 25480ms vs 955ms ideal)  
  - MST equal: Yes
```

Подальше збільшення кількості потоків призводить до ще більшого зниження продуктивності.

Граф з 500 вершин:

- 8 потоків:

```
- Summary:  
  - Nodes: 500  
  - Threads: 8  
  - Source Node: 3  
  - Sequential time: 2ms  
  - Parallel time: 148ms  
  - Speedup: 0.0135135x  
  - Efficiency: 0% (took 148ms vs 0ms ideal)  
  - MST equal: Yes
```

Паралельний алгоритм показує значно гірші результати порівняно з послідовним, оскільки для малих графів накладні витрати на управління потоками стають домінуючими.

Загальні висновки з результатів

Ефективність паралельного алгоритму залежить від розміру графу та кількості потоків. На малих графах паралельна обробка не виправдана через великі накладні витрати.

- Оптимальна кількість потоків зазвичай відповідає кількості фізичних ядер процесора. Використання більшої кількості потоків може навіть знизити продуктивність.
- Для великих графів паралельний алгоритм може показати суттєве прискорення. Проте потрібно забезпечити баланс між кількістю потоків та накладними витратами на їх синхронізацію.
- У моєму випадку, межа між ефективністю паралельного і послідовного алгоритмів була, приблизно, на графах розміром 75000 вершин. Графи з більшим розміром обраховувались швидше паралельно, а з меншим – навпаки.

```
- Summary:  
  - Nodes: 75000  
  - Threads: 8  
  - Source Node: 3  
  - Sequential time: 34889ms  
  - Parallel time: 33413ms  
  - Speedup: 1.04417x  
  - Efficiency: 13% (took 33413ms vs 4361ms ideal)  
  - MST equal: Yes
```

Висновок

У цій роботі були реалізовані послідовний та паралельний варіанти алгоритму Прима для побудови мінімального кісткового дерева. Паралельний алгоритм продемонстрував покращення продуктивності на великих графах, що підтверджує доцільність використання паралельних обчислень у задачах такого типу. Однак ефективність паралельного алгоритму зменшується при надмірному збільшенні кількості потоків або на малих графах.