ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА
Факультет прикладної математики та інформатики

Паралельні та розподілені обчислення
ЛАБОРАТОРНА РОБОТА №7

Тема: «Алгоритм Прима».

Виконала:
Ст. *Пелещак Вероніка*
ПМІ-35с

2025

Тема: Алгоритм Прима.

Завдання: Для зваженого зв'язного неорієнтованого графа G, використовуючи алгоритм Прима, з довільно заданої вершини a побудувати мінімальне кістякове дерево. Для різної розмірності графів та довільного вузла a порахувати час виконання програми без потоків та при заданих k потоках розпаралелення.

Опис програми:

1. generateGraph():

Генерує випадковий зважений зв’язний неорієнтований граф заданої кількості вершин.

- Створюється двовимірний вектор **graph[V][V]**, заповнений нулями.
- Використовується генератор випадкових чисел **mt19937** і розподіл **uniform_int_distribution**.
- Спочатку створюється ланцюг, який з’єднує всі вершини послідовно, щоб граф був **зв’язним**.
- Потім додаються випадкові ребра між вершинами з певною ймовірністю, щоб зробити граф більш щільним.
- Матриця ваг є **симетричною**, бо граф **неорієнтований**.

2. sequentialPrim():

Реалізація **послідовного алгоритму Прима** для побудови мінімального кістякового дерева (MST).

- Ініціалізуються допоміжні вектори:
    - **parent** — зберігає попередника кожної вершини в MST;
    - **key** — мінімальні ваги ребер до MST;
    - **inMST** — логічний масив, який позначає вершини, що вже у MST.
- На кожній ітерації:
    - Знаходиться вершина u з мінімальним ключем серед тих, що не в MST.
    - Додається ребро **parent[u]–u** до MST.
    - Оновлюються ваги (key) сусідніх вершин, якщо знайдено меншу вагу.
- Після проходу по всіх вершинах формується вектор mst, який містить усі ребра мінімального кістякового дерева.

3. parallelPrim():

Реалізація **паралельного алгоритму Прима** з використанням потоків **(std::thread)**.

- Основні масиви **(parent, key, inMST)** такі самі, як у послідовній версії.
- Для кожної ітерації алгоритму створюється декілька потоків, кожен із яких обробляє певний діапазон вершин і визначає **локальний мінімум ваги ребра**.
- Результати потоків **синхронізуються** за допомогою **mutex**, що дає змогу коректно знайти вершину з найменшим ключем.

- Після вибору вершини **u** оновлюються значення **ключів (key)** для сусідніх вершин, і формується структура мінімального остовного дерева (mst).

4. measureTime():

Вимірює час виконання будь-якої функції.

- Використовується **chrono::high_resolution_clock** для вимірювання часу початку та завершення виконання.
- Обчислюється різниця в секундах і повертається у вигляді **double**.

5. main():

Основна функція, яка керує процесом тестування алгоритмів.

- Задаються розміри графів **(8000, 10000, 12500)** і кількість потоків **(4 і 8)**.
- Для кожного розміру:
    1. Генерується випадковий граф.
    2. Виконується **послідовний алгоритм** і вимірюється його час.
    3. Виконується **паралельний алгоритм** для різної кількості потоків.
    4. Обчислюється **прискорення (Speedup)** та **ефективність (Efficiency)**.

Результат:

```
---------------------------------------
Розмір графа: 8000 вершин
Час послідовного виконання: 2.01819 секунд
Час паралельного виконання (4 потоків): 1.93867 секунд
Speedup: 1.04102
Effieciency: 0.260254
Час паралельного виконання (8 потоків): 2.01094 секунд
Speedup: 1.0036
Effieciency: 0.12545
---------------------------------------
---------------------------------------
Розмір графа: 10000 вершин
Час послідовного виконання: 3.16065 секунд
Час паралельного виконання (4 потоків): 2.69428 секунд
Speedup: 1.1731
Effieciency: 0.293274
Час паралельного виконання (8 потоків): 2.86981 секунд
Speedup: 1.10135
Effieciency: 0.137668
---------------------------------------
---------------------------------------
Розмір графа: 12500 вершин
Час послідовного виконання: 4.86913 секунд
Час паралельного виконання (4 потоків): 3.9386 секунд
Speedup: 1.23626
Effieciency: 0.309065
Час паралельного виконання (8 потоків): 4.1005 секунд
Speedup: 1.18745
Effieciency: 0.148431
---------------------------------------
```

Висновок: У програмі реалізовано дві версії **алгоритму Прима — послідовну** та **паралельну**, що дозволяє порівняти їхню продуктивність. Паралельна версія використовує потоки **(std::thread)** і м’ютекс **(std::mutex)** для коректного визначення вершини з мінімальним ключем. Реалізовано також **вимірювання часу**, **обчислення прискорення** та **ефективності**. Результати показують, що паралельні обчислення скорочують час виконання для великих графів.