Київський національний університет імені Тараса Шевченка

**Лабораторна робота**

**З навчального курсу «Розподілене та паралельне програмування»**

Виконав:

студент 4 курсу

факультету кібернетики

спеціальність «Комп’ютерні науки»

групи ТТП-42

Божевський Іван Петрович

**Київ 2025**

**Зміст**

**Постановка задачі**

Метою данної лабораторної роботи є реалізація Алгоритм Флойда-Варшаллаза допомогою різних підходів до паралелізації та розподілення обчислень, а також порівняння цих методів.

**Опис технологій**

**MPI (Message Passing Interface)**

MPI (Message Passing Interface) — це стандарт для обміну повідомленнями в паралельних та розподілених обчисленнях. Він забезпечує інтерфейс для комунікації між процесами, які можуть виконуватись як на одному комп'ютері, так і на багатьох вузлах у кластері. MPI дозволяє організовувати ефективний обмін даними між процесами за допомогою:

* Передачі повідомлень (send/receive),
* Глобальних операцій (наприклад, broadcast, reduce, scatter, gather),
* Синхронізації процесів.

MPI є особливо корисною технологією у випадках, коли задачі розбиваються між кількома процесами, що працюють незалежно та мають окрему пам’ять (distributed memory). У Python реалізація MPI можлива через бібліотеку mpi4py, яка забезпечує зручний інтерфейс до функціоналу стандарту MPI.

**OpenMP (Open Multi-Processing)**

OpenMP (Open Multi-Processing) — це API для багатопотокового програмування в середовищі з розділеною пам’яттю (shared memory). Вона дозволяє ефективно розпаралелювати обчислення між потоками одного процесу, використовуючи інструкції у вигляді директив (pragma) у C/C++ або коментарів у Fortran.

Основні можливості OpenMP:

* Розпаралелювання циклів (паралельні for-loops),
* Виконання секцій коду в кількох потоках,
* Синхронізація за допомогою конструкцій critical, barrier, atomic тощо.

У Python часткова підтримка OpenMP реалізується через використання Numba — JIT-компілятора, який дозволяє компілювати функції Python до машинного коду з можливістю використання OpenMP-подібного паралелізму через декоратори (наприклад, @njit(parallel=True)).

**Опис алгоритмe**

Алгоритм Флойда-Варшалла — це класичний алгоритм для знаходження найкоротших шляхів між усіма парами вершин у графі. Він належить до категорії динамічного програмування і використовує ітеративний підхід для покращення шляхів на кожному етапі. Ось детальний опис цього алгоритму:

Вхід:

1. Граф: Неорієнтований або орієнтований граф, де кожне ребро має вагу (може бути і невід'ємною, і від'ємною, але без від'ємних циклів). Граф представлений матрицею суміжності, де елементи матриці — це ваги ребер, а відсутність ребра між двома вершинами може позначатися нескінченністю або дуже великим числом.
2. Матриця відстаней: На початковому етапі матриця відстаней dist[i][j] містить:
   * Вагу ребра між вершинами i та j, якщо таке ребро існує.
   * Нескінченність (або дуже велике число), якщо ребра між вершинами немає.
   * Нулі на діагоналі (тобто для будь-якої вершини відстань до себе дорівнює нулю).

Алгоритм:

Алгоритм базується на принципі динамічного програмування. Основна ідея полягає в тому, щоб поступово покращувати відстані між всіма парами вершин за допомогою трьох вкладених циклів:

1. Ініціалізація: Спочатку ми маємо матрицю відстаней, де для кожної пари вершин записана відстань або нескінченність, якщо пряме з'єднання відсутнє.
2. Основний цикл:
   * Для кожної вершини k (розглядається як посередник) перевіряються всі пари вершин i та j.
   * Якщо шлях через k скорочує відстань між вершинами i і j, то оновлюється матриця відстаней:

dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j])

Інакше значення dist[i][j] залишається незмінним.

1. Завершення: Після виконання всіх ітерацій для кожної вершини, матриця dist містить найкоротші відстані між усіма парами вершин.

**Результати та порівняння**

Реалізовано генератор графів для стовреня тестових даних різної розмірності. Для наступних тестів використовуємо розмірність від 100 вершин до 1000 з кроком 100 вершин, всього 10 тестів на метод та алгоритм.

Для MPI запускаємо з параметром -n 8, тобто 8 ядер.

|  |  |  |  |
| --- | --- | --- | --- |
| Розмірність Графу | Час(секунди)  Звичайний алгоритм | Час(секунди)  MPI | Час(секунди)  OpenMP |
| 100 | 0.673811 | 0.047636 | 0.001001 |
| 200 | 6.311242 | 0.337727 | 0.001868 |
| 300 | 20.925933 | 2.811157 | 0.003328 |
| 400 | 52.036167 | 7.705416 | 0.006160 |
| 500 |  | 5.328368 | 0.009507 |
| 600 |  | 9.006365 | 0.014002 |
| 700 |  | 13.805203 | 0.019006 |
| 800 |  | 20.792219 | 0.027003 |
| 900 |  | 28.963339 | 0.039512 |
| 1000 |  | 39.294063 | 0.049997 |

Для звичайного алгоритму проведено тільки 4 перші тести, оскільки вже для 400 вершин час 52 секунди, що в рази бльше інших методів.

Посилання на репозиторій: https://github.com/LironeA/RPP\_Lab1

**Висновки**

У ході виконання лабораторної роботи було реалізовано та протестовано алгоритми роботи з графами з використанням трьох підходів: звичайної (послідовної) реалізації, паралельної з використанням MPI, а також паралельної реалізації з OpenMP. Результати експериментів засвідчили суттєве зростання ефективності при використанні паралельних підходів, особливо OpenMP, у порівнянні з традиційним послідовним виконанням.

Аналізуючи отримані дані, можна зробити такі висновки:

1. **Звичайна реалізація** демонструє експоненційне зростання часу виконання із збільшенням розмірності графа. Наприклад, для графа з 400 вузлів час виконання становить понад 52 секунди, а для 300 – майже 21 секунду. Це свідчить про низьку масштабованість при збільшенні обсягів вхідних даних.
2. **MPI** показав значне покращення продуктивності. Навіть при збільшенні розміру графа до 1000 вузлів, час виконання не перевищив 40 секунд. Порівняно з послідовною реалізацією, MPI забезпечив прискорення в декілька разів, особливо помітне на середніх розмірах графів (наприклад, у 20 разів швидше для графа розміром 300).
3. **OpenMP** виявився найбільш ефективним підходом серед трьох. Час виконання залишався дуже малим навіть для великих графів: усього ~0.067 секунд для графа з 1000 вузлів. Це свідчить про ефективне використання багатоядерності на рівні потоків, що забезпечує максимальну швидкодію для задач такого типу.

У результаті, можна зробити висновок, що для задач обробки великих графів оптимальним підходом є використання OpenMP, завдяки його високій продуктивності та мінімальному часу виконання. MPI також демонструє хорошу ефективність, особливо в розподілених середовищах. Звичайна реалізація є доцільною лише для малих графів або при відсутності можливості використання паралельних технологій.