

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: Алгоритм Флойда-Воршелла та його паралельна реалізація на платформі Node.js

|  |  |
| --- | --- |
| **Керівник**:  професор Стеценко Інна Вячеславівна  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Мєшков Андрій Ігорович  студент групи ІП-15  залікова книжка № 1522  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «11» квітня 2024 р.  Інна СТЕЦЕНКО |

**Київ – 2024**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

Факультет Інформатики та обчислювальної техніки

(повна назва)

Кафедра Інформатики та програмної інженерії

(повна назва)

Дисципліна Технології паралельних обчислень

(повна назва)

Курс 3 Група ІП 15 Семестр 2

**ЗАВДАННЯ**

**НА КУРСОВУ РОБОТУ СТУДЕНТУ**

*Мєшков Андрій Ігорович*

(прізвище, ім’я, по батькові)

|  |
| --- |
| **1. Тема роботи** «Алгоритм Флойда-Воршелла та його паралельна реалізація на платформі Node.js» |

**2. Термін подання студентом роботи** «11» *квітня 2024 року*

**3.** **Вихідні дані до роботи**

|  |
| --- |
| *Пояснювальна записка* |
|  |

**4.** **Зміст пояснювальної записки**

|  |
| --- |
| *1) Опис алгоритму та його відомих паралельних реалізацій* |
| *2) Розробка паралельного алгоритму та аналіз його швидкодії* |
| *3) Вибір програмного забезпечення для розробки паралельних обчислень* |
| *та його короткий опис* |
| *4) Розробка паралельного алгоритму з використанням обраного програмного* |
| *забезпечення: проєктування, реалізація, тестування* |
| *5) Дослідження ефективності паралельних обчислень алгоритму* |

**5. Перелік графічного матеріалу**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

**6. Дата видачі завдання** *«20» лютого 2024 року*

**АНОТАЦІЯ**

**Структура та обсяг роботи.** Пояснювальна записка курсової роботи складається з 5 розділів, містить 5 рисунків, 2 таблиці, 5 джерел.

У даній курсовій роботі досліджується алгоритм Флойда-Воршелла, який використовується для пошуку найкоротших шляхів у зваженому графі. Основна увага приділяється його паралельній реалізації з використанням середовища виконання Node.js.

Розглядаються основні принципи роботи алгоритму, його оптимізації та впровадження у середовищі Node.js для ефективного використання багатопроцесорних систем.

У роботі представлені результати експериментів з реалізованим алгоритмом, що демонструють його продуктивність та масштабованість при обробці великих обсягів даних. Дана робота є актуальною у контексті розвитку паралельних обчислювань та оптимізації алгоритмів для великих обсягів даних.

КЛЮЧОВІ СЛОВА: ПАРАЛЕЛЬНИЙ АЛГОРИТМ, NODE.JS, ФЛОЙД-ВОРШЕЛЛ, БАГАТОПОТОЧНІСТЬ.

ЗМІСТ

[**ЗАВДАННЯ** 2](#_Toc162896387)

[ВСТУП 5](#_Toc162896388)

[1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 6](#_Toc162896389)

[1.1 Теоретичні основи алгоритму Флойда-Воршелла 6](#_Toc162896390)

[1.2 Відомі паралельні реалізації алгоритму Флойда-Воршелла 8](#_Toc162896391)

[2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 11](#_Toc162896392)

[3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 15](#_Toc162896393)

[4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 17](#_Toc162896394)

[5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ 22](#_Toc162896395)

[ВИСНОВКИ 25](#_Toc162896396)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 26](#_Toc162896397)

[ДОДАТКИ 27](#_Toc162896398)

[Додаток А. ТЕКСТИ ПРОГРАМНОГО КОДУ 27](#_Toc162896399)

# ВСТУП

У сучасному світі, де обсяги даних зростають експоненційно, а вимоги до швидкості їх обробки стають все більш жорсткими, паралельні обчислення набувають особливої актуальності. Використання багатоядерних процесорів, кластерів та обчислювальних хмар дозволяє значно прискорити обробку даних і розв'язування складних алгоритмічних задач. Серед таких задач важливе місце займає пошук найкоротших шляхів у графах, який має широке застосування від комп'ютерних мереж і систем GPS-навігації до оптимізації логістичних ланцюгів та аналізу соціальних мереж.

Алгоритм Флойда-Воршелла є класичним рішенням для знаходження найкоротших шляхів між усіма парами вершин у зваженому графі. Водночас, при обробці великих графів, цей алгоритм вимагає значних обчислювальних ресурсів, що робить його ідеальним кандидатом для паралельної обробки.

Ця курсова робота присвячена розробці та аналізу паралельної реалізації алгоритму Флойда-Воршелла з використанням Node.js, середовища виконання, яке дозволяє ефективно використовувати можливості сучасних багатопроцесорних систем. Метою роботи є не лише демонстрація потенційного прискорення виконання алгоритму за допомогою паралельної обробки, але й аналіз можливих проблем та обмежень, які виникають при такій реалізації. Ключовою задачею є розробка ефективної моделі паралельного виконання, яка б могла бути масштабована для обробки великих графів, а також порівняння продуктивності паралельної реалізації з традиційним секвенційним підходом.

Таким чином, актуальність даної курсової роботи обумовлена не тільки широким спектром застосування алгоритму Флойда-Воршелла, але й постійним пошуком способів оптимізації обчислень за допомогою використання паралельних алгоритмів і технологій. Результати дослідження можуть бути корисними для розробників програмного забезпечення, науковців та інженерів, які працюють у галузі обчислювальної техніки та оптимізації алгоритмів.

# 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

У цьому розділі даної курсової роботи розглядається алгоритм Флойда-Воршелла, який є класичним методом в області теорії графів для знаходження найкоротших шляхів між усіма парами вершин у зваженому графі. Особлива увага приділяється не лише теоретичним основам алгоритму, його математичній моделі та алгоритмічним крокам, але й аналізу існуючих паралельних реалізацій, що дозволяють значно знизити час обчислень за рахунок використання сучасних багатопроцесорних систем та обчислювальних кластерів.

## 1.1 Теоретичні основи алгоритму Флойда-Воршелла

Алгоритм Флойда-Воршалла, названий на честь його творців Роберта Флойда та Стівена Уоршалла, є фундаментальним алгоритмом в інформатиці та теорії графів. Він використовується для пошуку найкоротших шляхів між усіма парами вузлів у зваженому графі. Цей алгоритм є високоефективним і може обробляти графи як з додатними, так і з від’ємними вагами ребер, що робить його універсальним інструментом для вирішення широкого кола проблем мережі та з’єднання.

Алгоритм Флойда-Воршалла порівнює багато можливих шляхів через граф між кожною парою вершин. Він гарантовано знаходить усі найкоротші шляхи та може робити це за допомогою порівнянь на графіку, навіть якщо на графіку можуть бути ребра. Це робиться шляхом поступового покращення оцінки на найкоротшому шляху між двома вершинами, поки оцінка не стане оптимальною.

Розглянемо граф *G* з вершинами *V*, пронумерованими від 1 до *N*. Далі розглянемо функцію shortestPath(*i, j, k*), яка повертає довжину найкоротшого можливого шляху (якщо такий існує) від *i* до *j*, використовуючи лише вершини з набору {1,2,…, *k* } як проміжні точки уздовж спосіб. Тепер, маючи цю функцію, наша мета — знайти довжину найкоротшого шляху від кожного *i* до кожного *j* за допомогою будь-якої вершини в {1,2,…, *N*}. За визначенням, це значення shortestPath(*i, j, N*) , яке ми знайдемо рекурсивно. [1]

Зауважте, що shortestPath(*i, j, k*) має бути меншим або дорівнювати shortestPath(*i, j, k*−1): ми маємо більше гнучкості, якщо нам дозволено використовувати вершину *k*. Якщо shortestPath(*i, j, k*) насправді менший за shortestPath(*i, j, k*−1) , то повинен існувати шлях від *i* до *j* з використанням вершин {1,2,…, *k*}, який є коротшим ніж будь-який такий шлях, який не використовує вершину *k*. Оскільки негативних циклів немає, цей шлях можна розкласти як:

(1) шлях від *i* до *k*, який використовує вершини {1,2,…, *k*−1}, за якими йде

(2) шлях від *k* до *j*, який використовує вершини {1,2,…, *k*−1}.

І, звісно, це мають бути найкоротші такі шляхи, інакше ми могли б ще більше зменшити довжину. Іншими словами, ми прийшли до рекурсивної формули 1.1.

(1.1)

Тим часом базовий випадок задано формулою 1.2.

(1.2)

де позначає вагу ребра від до , якщо воно існує, і ∞ (нескінченність) в іншому випадку.

Ці формули є основою алгоритму Флойда-Воршалла. Алгоритм працює так, що спочатку обчислюється shortestPath(*i, j, k*) для всіх пар (*i, j*) для *k* =0 , потім *k* =1 , потім *k* =2 і так далі. Цей процес триває до тих пір, поки не буде *k* = *N*, і ми знайдемо найкоротший шлях для всіх (*i, j*) пар, використовуючи будь-які проміжні вершини. [2]

Нижче наведено псевдокод для цієї базової версії.

let dist be a |V| × |V| array of minimum distances initialized to ∞

for each edge (u, v) do

dist[u][v] ← w(u, v)

for each vertex v do

dist[v][v] ← 0

for k from 1 to |V|

for i from 1 to |V|

for j from 1 to |V|

if dist[i][j] > dist[i][k] + dist[k][j]

dist[i][j] ← dist[i][k] + dist[k][j]

end if

## 1.2 Відомі паралельні реалізації алгоритму Флойда-Воршелла

Алгоритм Флойда-Воршелла використовується для пошуку найкоротших шляхів між усіма парами вершин у спрямованому та зваженому графі. Основна мета алгоритму полягає в тому, щоб визначити мінімальну відстань між будь-якими двома вершинами у графі.

У цьому контексті кожна вершина розглядається як проміжна вершина "k", і алгоритм перевіряє, чи можна скоротити відстань між вершинами "i" та "j", проходячи через вершину "k". Це виражається у формулі 1.3:

(1.3)

Для оновлення значення A[i,j] потрібні значення A[i,k]та A[k, j].

При k=1, можна спостерігати наступне:

- Для A[0, 2] потрібні значення A[0, 1] та A[1, 2] .

- Для A[1, 2] потрібні значення A[1, 1] та A[1, 2] .

- Для A[2, 2] потрібні значення A[2, 1] та A[1, 2] .

- Для A[3, 2] потрібні значення A[3, 1] та A[1, 2] .

Це означає, що для певного k , значення A[k][j] потрібні всім елементам стовпця j .

Аналогічно, можна визначити наступне:

- Для A[0, 0] потрібні значення A[0, 1] та A[1, 0] .

- Для A[0, 1] потрібні значення A[0, 1] та A[1, 1] .

- Для A[0, 2] потрібні значення A[0, 1] та A[1, 2] .

- Для A[0, 3] потрібні значення A[0, 1] та A[1, 3] .

Під час ітерації k зовнішнього циклу, кожний елемент рядка k матриці A має бути переданий кожному завданню в тому ж стовпці, що і цей елемент. Це також відноситься і до елементів стовпця k матриці A , які повинні бути передані кожному завданню в тому ж рядку, що і цей елемент.

Таким чином, для забезпечення ефективної реалізації алгоритму використовується поділ даних матриці за допомогою двовимірного відображення блоків:

- Усі елементи матриці розбиваються на квадрати однакового розміру, кожен з яких призначається окремому процесору.

- Для матриці розміром та *p* процесорів, кожен процесор обчислює частину матриці відстаней розміром .

**Приклад**. Паралельна реалізація алгоритму Флойда-Воршелла використовує двовимірне відображення блоків для оптимізації обчислень. Наприклад, у графі з 16 вершинами, розділеному на 4 процесори, кожен процесор обробляє частину графу розміром 4 4 для швидшого знаходження найкоротших шляхів. елементів розподілено між *p* процесорамів знаходимо за формулою 1.4.

(1.4)

Критична умова рівномірного розподілу даних у формулі 1.5.

(1.5)

Псевдокод паралельного алгоритму:

func Floyd\_All\_Pairs\_Parallel ()

for k := 1 to *n* do

Each process , that has a segment of the k-th row of ,

broadcasts it to the processes;

Each process , that has a segment of the k-th column of , broadcasts it to the , processes;

Each process waits to receive the needed segments;

Each process computes its part of the matrix;

Дану модель розв’язку ми надалі будемо використовувати у розробці алгоритму.[3]

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У цьому розділі ми розробимо та проаналізуємо швидкодію послідовного алгоритму. Спершу представимо алгоритм у вигляді спрощеного псевдокоду:

function floydWarshall(graph):

nodes = length of graph

dist = graph

for each node k in nodes:

for each node i in nodes:

for each node j in nodes:

if dist[i][k] + dist[k][j] < dist[i][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

Алгоритм Floyd-Warshall правильно знаходить найкоротші шляхи між усіма парами вершин у зваженому графі. Цю правильність можна довести математично індукцією.

- Базовий Крок: На початку, дистанції між вершинами відомі для прямих шляхів між ними.

- Крок Індукції: Припустимо, що після кроку k, dist[i][j] містить довжину найкоротшого шляху між вершинами i та j за участю вершин від 1 до k.

- Під час кожного кроку алгоритму, він оновлює dist[i][j], якщо виявляється, що шлях через вершину k коротший за попередній відомий шлях.

- Коли алгоритм завершується (після n кроків для графу з n вершинами), dist[i][j] буде містити довжину найкоротшого шляху між усіма парами вершин.

Таким чином, алгоритм Floyd-Warshall є правильним, оскільки він перебирає всі можливі пари вершин та уточнює найкоротші шляхи за O() часу.

Але також можна перевірити правильність знаючи правильну відповідь. Для цього побудуємо функцію яка порівняє відповідь алгоритму з очікуваним результатом. Напишемо псевдокод:

function checkFloydWarshall(graph, expectedDistances)

result = graph

for i = 0 to length of graph - 1

for j = 0 to length of graph - 1

if result[i][j] != expectedDistances[i][j]

return false

return true

Для перевірки візьмемо граф, який зображено на рисунку 2.1.

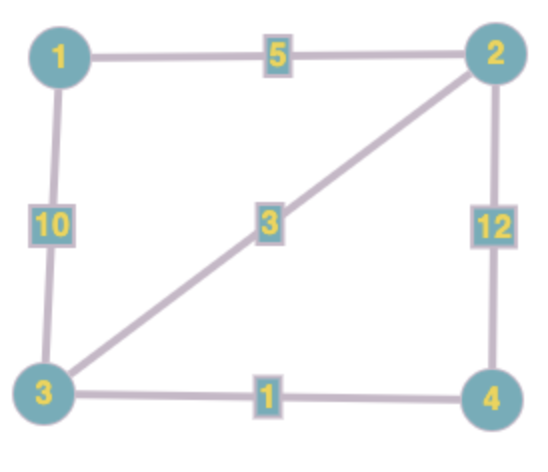


Рисунок 2.1. Перевірочний граф

Матрицю відстаней було визначено вручну за допомогою онлайн-ресурсів, що дало наступний результат:

0, 5, 8, 9,

5, 0, 3, 4,

8, 3, 0, 1,

9, 4, 1, 0.

Точність алгоритму було перевірено шляхом порівняння отриманих відстаней з очікуваними результатами. Результати цієї перевірки представлено на рисунку 2.2.



Рисунок 2.2. Результат перевірки

Альтернативні методи валідації можуть включати застосування того ж чи іншого алгоритму для пошуку найкоротших шляхів, однак це порушує питання надійності таких алгоритмів для перевірки. В таких випадках достатньою є проста ручна валідація.

Для проведення аналізу швидкодії алгоритму, ми можемо створити ненаправлені графи різної кількості вершин і порівняти час виконання алгоритму для них. Давайте розглянемо приклад. Створемо граф за допомогою функції, яка представлена у псевдокоді:

function generateRandomGraph(numNodes, density = 0.5, maxWeight = 10)

edges = empty list

for i = 0 to numNodes - 1

for j = i + 1 to numNodes - 1

if random() < density

weight = floor(random() \* maxWeight) + 1

add [i, j, weight] to edges

add [j, i, weight] to edges

return createGraph(numNodes, edges)

Проведення аналізу швидкодії для різної кількості вершин

Після реалізації алгоритму, мовою програмування JavaScript (код наведений у Додатку А), ми можемо провести аналіз швидкодії алгоритму(рис. 2.3).

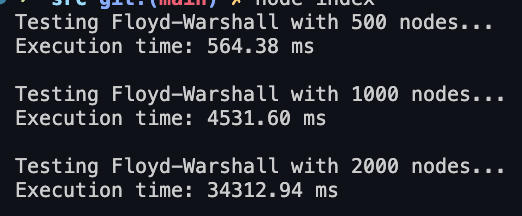


Рисунок 2.3. Результат виконання

Цей код генерує ненаправлені графи з різною кількістю вершин, вимірює час виконання алгоритму Флойда-Уоршела для кожного графа, і виводить цей час у секундах.

Аналізуючи вивід, можна побачити, як час виконання збільшується зі збільшенням кількості вершин у графі. Такий аналіз допоможе з'ясувати, як швидко алгоритм працює для різних розмірів вхідних даних. Надалі, ці дані буде порівняно з даними паралельного алгоритму, та буде виміряно прискорення.

# 3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

Для імплементації паралельних обчислень в рамках цього проекту було обрано середовище Node.js у комбінації з модулем worker\_threads. Цей вибір підкріплений низкою факторів, що вказують на високу придатність цього інструментарію для реалізації масштабованих і водночас ефективних паралельних обчислювальних задач.

Node.js, як відомо, це середовище виконання JavaScript на стороні сервера, яке надає масштабованість та високу продуктивність за рахунок подійно-орієнтованої архітектури та неблокуючих вводу-виводу операцій. Ці характеристики роблять Node.js ідеальним для розробки аплікацій, де потрібна швидка обробка великої кількості одночасних з'єднань або запитів.[4]

Модуль worker\_threads, введений у Node.js, дозволяє використовувати мультипоточність у середовищі, що традиційно було орієнтовано на однопотокову виконання. Worker\_threads надає API для створення нових потоків (worker'ів), що можуть виконувати JavaScript або WebAssembly код паралельно головному потоку. Це забезпечує значне поліпшення продуктивності для задач, що вимагають інтенсивних обчислень, дозволяючи Node.js додаткам ефективно використовувати багатоядерні процесори.[5]

Використання worker\_threads у Node.js має кілька переваг:

1. Покращена продуктивність: Можливість паралельного виконання обчислювально важких задач без блокування головного потоку, що забезпечує краще використання апаратних ресурсів.

2. Гнучкість та контроль: Розробники можуть детально керувати поведінкою кожного worker'а, включаючи ініціалізацію, передачу даних та завершення роботи, що надає високий рівень контролю над паралельними обчисленнями.

3. Сумісність з сучасними стандартами: Worker\_threads є частиною стандарту HTML для Web Workers, що забезпечує легкість інтеграції та використання у широкому спектрі Node.js аплікацій.

Окрім технічних переваг, вибір Node.js та worker\_threads також визначається широкою підтримкою спільноти та наявністю обширних бібліотек та інструментів, що спрощують розробку та тестування складних аплікацій.

Загалом, використання Node.js у поєднанні з worker\_threads для паралельних обчислень у цьому проекті є виваженим та обґрунтованим вибором, що відповідає сучасним.

# 4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

Вибір реалізації паралельних обчислень з використанням процесорів в даному проекті дозволяє досягти значного прискорення обчислень, особливо для завдань, які вимагають інтенсивної обробки даних та великої кількості одночасних обчислень.

Одним з головних аргументів за використанням паралельних обчислень з процесорами є можливість одночасної обробки різних частин даних різними процесорами. У випадку алгоритму Флойда-Уоршелла, де необхідно оновлювати матрицю відстаней для всіх пар вершин у кожному кроці, паралельний підхід дозволяє кожному процесору виконувати обчислення для окремих частин матриці одночасно. Це призводить до значного збільшення продуктивності та скорочення часу виконання алгоритму.

Крім того, використання Node.js для реалізації паралельних алгоритмів також має свої переваги. У порівнянні з іншими мовами програмування, Node.js має декілька обмежень у вбудованих можливостях для паралельних обчислень. Проте, модуль worker\_threads у Node.js вирішує це обмеження, надаючи зручний інтерфейс для створення та керування паралельними потоками. Це дає можливість ефективно використовувати паралельні потоки без додаткових бібліотек або середовищ.

Для втілення паралельного алгоритму Флойда-Уоршелла у середовищі Node.js було створено функцію parallelFloydWarshall. Ця функція отримує на вхід матрицю відстаней distanceMatrix та кількість процесорів `numProcessors`, які будуть використовуватись для паралельних обчислень.

Для забезпечення паралельності використовувалися об'єкти `SharedArrayBuffer` та `Int32Array`, які дозволяли кожному потоку отримувати доступ до одного і того ж розділеного масиву даних. Дані у масиві були представлені у вигляді чисел, де нульові значення відповідали відсутності зв'язку між вершинами, а значення `Infinity` вказували на відсутність прямого зв'язку.

Для кожного процесора було обчислено частини матриці, над якими виконувалися операції пошуку найкоротших відстаней. Ці частини матриці були розподілені між потоками з використанням `ThreadPool`, який забезпечував ефективне використання ресурсів процесора. Код наведено у додатку А.

Для перевірки коректності та ефективності реалізованого паралельного алгоритму було проведено ручне тестування. Основні етапи тестування включали:

- Було підготовлено тестові дані з відомими відстанями між вершинами графу, це дані, які були використанні у розділі 2.

- Результати роботи паралельного алгоритму були порівняні з очікуваними значеннями. Алгоритм працює.

**Вибір оптимальних параметрів (кількість процесорів).** Оптимальна кількість процесорів для паралельного алгоритму Флойда-Уоршелла була визначена на основі ретельного тестування. Для цього було використано набір різноманітних даних різних розмірів:

**Розмірність даних**: Для визначення оптимальної кількості процесорів були обрані три розмірності вхідних даних: 1000x1000, 2000x2000, 3000x3000 матриць відстаней.

**Кількість процесорів**: Для кожної з розмірностей даних було підготовлено масив кількості процесорів, які використовувались у тестуванні. Використані значення: 4, 16, 25, 64, 100 процесорів. Також було передбачено валідацію значень процесорів, так як ділення кількості вершин на корінь кількості процесорів не може мати залишку.

Усі графи, які використовувалися у тестуванні, були ненапрямлені.

Для усіх тестових даних були використані цілі позитивні числа, де нуль відповідає відсутності зв'язку між вершинами, а значення "нескінченність" вказує на відсутність прямого зв'язку між вершинами. В реальності алгоритм може працювати і на інших нецілих та негативних даних.

Результати тестування та обрана оптимальна кількість процесорів представлені на Рисунку 4.1.



Рисунок 4.1. Результат виконання

Запишемо дані в таблицю 4.1.

Таблиця 4.1. – Результати тестувань - час, мікросекунди

|  |  |  |  |
| --- | --- | --- | --- |
| Кількість процесорів | Кількість вершин графу | | |
| 1000 | 2000 | 3000 |
| 4 | 3556 | 27014 | 90051 |
| 16 | 3612 | 25824 | 85004 |
| 25 | 3749 | 25942 | 94239 |
| 64 | 4270 | 26655 | 87153 |
| 100 | 4814 | 27226 | 87290 |

Результати тестування паралельного алгоритму Флойда-Уоршелла засвідчили, що оптимальна кількість процесорів значно впливає на швидкість виконання алгоритму та його ефективність у роботі з різними розмірами графів.

**Оптимальна Кількість Процесорів**.

1. Графи розміром 1000x1000:

- Найкращий результат показав використання 4 процесорів з часом 3556 ms.

2. Графи розміром 2000x2000:

- Найкращі результати було досягнуто з 16 процесорами, що зайняли 25824 ms.

3. Графи розміром 3000x3000:

- Оптимальним вибором виявилося використання 16 процесорів з часом 85004 ms.

З цих результатів випливає, що 16 процесорів – найоптимальніше число при тому, що нам важливіші результати на більших даних (тому не обрано 4).

**Підвищення Швидкості з Більшою Кількістю Процесорів**. Найефективніше використання 16 процесорів показало добрі результати на всіх, окрім перших найменших даних. Це пояснюється можливістю розділення роботи на більші кількість паралельних потоків, що дозволяє прискорити обчислення.

**Різна Ефективність для Різних Розмірів Графів**. Зауважимо, що оптимальна кількість процесорів може змінюватися в залежності від розміру графу. Наприклад, для менших графів (1000x1000) було доцільно використовувати менше процесорів, тоді як для більших графів (2000x2000, 3000x3000) ефективнішим виявилося використання більшої кількості процесорів.

**Вплив на Швидкодію у Залежності від Розміру Графу**. У деяких випадках збільшення кількості процесорів не завжди призводило до зменшення часу виконання. Це може пояснюватися додатковими витратами на обмін даними та координацію між процесорами.

Таким чином, оптимальна кількість процесорів для паралельного алгоритму Флойда-Уоршелла залежить від розміру вхідних даних та може бути різною для різних випадків. У даному дослідженні виявлено, що при розмірах матриць 2000x2000 та 3000x3000 оптимальною кількістю процесорів є 16, тоді як для 1000x1000 найбільш підходящою виявилася кількість 4 процесорів. Для подальшого вирахування прискорення буде використано саме 16 процесорів.

Це дослідження вказує на важливість вибору оптимальної кількості процесорів для паралельних обчислень, що дозволяє досягти максимальної швидкодії та ефективності виконання алгоритмів на великих обсягах даних.

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

В цьому розділі буде проведено експеримент з дослідженням ефективністю паралельних обчислень відносно послідовних. Для цього потрібно виміряти час дії обох реалізацій та знайти прискорення за формулою 5.1:

(5.1)

Для проведення експерименту з дослідження ефективності паралельних обчислень відносно послідовних були встановлені наступні умови:

1. Кількість Повторень: Кожен експеримент повторили тричі для кожного розміру графу. Такий підхід дозволяє отримати більш стабільні та надійні результати шляхом усереднення.

2. Параметри Розмірів Графів: В експерименті використовувалися графи різних розмірів: 1000, 2000, 3000 вершин. Це дозволяє перевірити реакцію алгоритму на різні обсяги вхідних даних.

3. Кількість Процесорів: Оптимальнішим числом виявилося 16 процесорів. Даний висновок зроблено в Розділі 4.

4. Вимірювання Часу: Час виконання кожного алгоритму вимірювався з використанням вбудованих засобів Nodejs, що надають точність до мікросекунд. А саме performance.

5. Технічні засоби системи експерименту:

* тип процесору: Intel Core i5;
* об‘єм ОЗП: 8 Гб;
* підключення до мережі Інтернет зі швидкістю від 20 мегабіт;

6. Програмне забезпечення: Алгоритм написано мовою Javascript на платформі Nodejs в інтегрованому середовищі розробки VSCode. Було використано бібліотеки: weather threads, url та path.

7. Дані: У дослідженні використовувалися ненаправлені графи з довжиною до 10, вага шляху – ціле позитивно число.

Проведений експеримент дозволяє визначити оптимальну кількість процесорів для паралельного алгоритму Флойда-Уоршелла та встановити його ефективність порівняно з послідовною реалізацією. Отримані результати дозволяють розуміти, як алгоритм поводиться при різних обсягах даних та кількостях паралельних обчислень.

Використовуючи повтори, треба вирахувати середній час дії кожної реалізації за формулою 5.2.

(5.2)

Виведемо результати на рисунку 5.1.

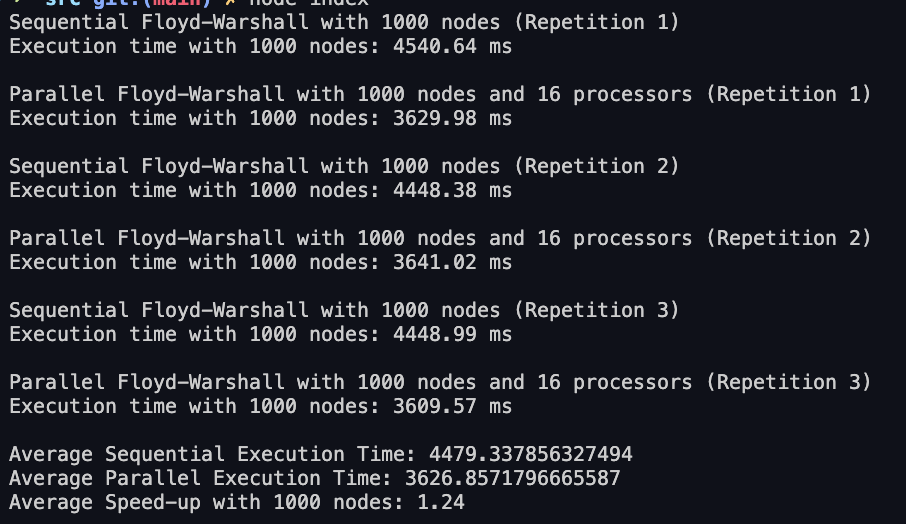


Рисунок 5.1. Демонстрація обчислення з повторами

Занесемо середні значення та прискорення у таблицю 5.1 та порівняємо кінцеві результати.

Таблиця 5.1. – Порівняння часу роботи алгоритмів

|  |  |  |  |
| --- | --- | --- | --- |
| Кількість вершин графа | Час послідовного алгоритму, мікросекунд | Час паралельного алгоритму, мікросекунд | Прискорення алгоритму |
| 1000 | 4479 | 3626 | 1.24 |
| 2000 | 35277 | 25641 | 1.37 |
| 3000 | 127783 | 87057 | 1.47 |

Загалом, ми бачимо, що паралельний алгоритм показує зростання прискорення зі збільшенням кількості вершин у графі. Це може бути пов'язано з тим, що при більшому розмірі графа є більше обчислень, які можна розподілити між різними процесорами. Таким чином, використання паралельних обчислень стає більш ефективним і призводить до значного прискорення у порівнянні з послідовним алгоритмом. Найбільше прискорення (1.47) було досягнуто при розмірі графа 3000 вершин. Це може свідчити про те, що в цьому випадку паралельне виконання алгоритму дозволяє краще використовувати ресурси системи та ефективно розподіляти завдання між процесорами, що призводить до збільшення продуктивності.

# ВИСНОВКИ

У даній роботі був розглянутий алгоритм Флойда-Уоршелла, який використовується для знаходження найкоротших шляхів в графі між усіма парами вершин. Цей алгоритм є класичним методом для розв'язання задачі найкоротших шляхів і зазвичай застосовується в графічних моделях, маршрутизації мереж та інших областях.

Для реалізації паралельної версії алгоритму було використано мову програмування JavaScript та платформу Node.js. Для забезпечення паралельності була використана бібліотека `worker\_threads`, яка дозволяє створювати та керувати потоками в середовищі Node.js. Додатково використовувались бібліотеки `url` та `path` для роботи з URL-адресами та шляхами файлів в процесі реалізації.

Умови досягнення найкращого прискорення для паралельного алгоритму залежать від розміру графу та кількості доступних процесорів. За результатами експериментів з 1000, 2000 та 3000 вершинами графу було виявлено, що найбільше прискорення досягається при використанні 16 процесорів.

Рекомендації до параметрів алгоритму для досягнення найбільшого прискорення включають вибір оптимальної кількості процесорів, яка залежить від розміру графу, а також ефективне розподілення роботи між ними. Для графів менших розмірів, може бути корисним використання меншої кількості процесорів для запобігання зайвого навантаження.

Таким чином, паралельна реалізація алгоритму Флойда-Уоршелла на платформі Node.js виявляється ефективним способом розв'язання задачі найкоротших шляхів в графі, особливо при використанні оптимальної кількості процесорів в залежності від розміру вхідного графу.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cormen T. H., Leiserson C.E., Rivest R. L. Introduction to Algorithms. 1990.
2. Kenneth H. R. Discrete Mathematics and Its Applications, 5th Edition. 2003.
3. Gautam A. Parallel Implementation Of Floyd-Warshall Algorithm. New York, 2019.
4. Документація Node.js. URL: <https://nodejs.org/en>
5. Документація модуля worker\_threads // Node.js documentation. URL: <https://nodejs.org/api/worker_threads.html>

# ДОДАТКИ

## Додаток А. ТЕКСТИ ПРОГРАМНОГО КОДУ

Код програми також наведено у даному репозиторії: <https://github.com/IP15-MieshkovAndrii/Year3.2/tree/main/Course%20work/src>

Файл floydWarshall.js

export const floydWarshall = (graph) => {

let nodes = graph.length;

let dist = graph;

for (let k = 0; k < nodes; k++) {

for (let i = 0; i < nodes; i++) {

for (let j = 0; j < nodes; j++) {

if ((dist[i][k] + dist[k][j]) < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

return dist;

}

Файл graph.js

export const createGraph = (numNodes, edges) => {

const graph = Array.from({ length: numNodes }, () => Array(numNodes).fill(Infinity));

for (let i = 0; i < numNodes; i++) {

graph[i][i] = 0;

}

edges.forEach(([src, dest, weight]) => {

graph[src][dest] = weight;

});

return graph;

};

Файл checkFloydWarshall.js

export const checkFloydWarshall = (graph, expectedDistances) => {

const result = graph;

for (let i = 0; i < graph.length; i++) {

for (let j = 0; j < graph.length; j++) {

if (result[i][j] !== expectedDistances[i][j]) {

return false;

}

}

}

return true;

};

Файл parallelFloydWarshall.js

import { ThreadPool } from './threadPool.js';

import { fileURLToPath } from 'url';

import { dirname, join } from 'path';

export async function parallelFloydWarshall(distanceMatrix, numProcessors) {

const nodes = distanceMatrix.length;

const sharedArrayBuffer = new SharedArrayBuffer(nodes \* nodes \* Int32Array.BYTES\_PER\_ELEMENT);

const sharedArray = new Int32Array(sharedArrayBuffer);

const INF = 1e9;

if(nodes%Math.sqrt(numProcessors) !== 0){

console.log("Invalid number of processors!")

return distanceMatrix;

}

sharedArray.set(distanceMatrix.flat());

for (let i = 0; i < nodes; i++) {

for (let j = 0; j < nodes; j++) {

if (i === j) {

sharedArray[i \* nodes + j] = 0;

} else if (sharedArray[i \* nodes + j] === 0){

sharedArray[i \* nodes + j] = INF;

}

}

}

const pool = new ThreadPool(numProcessors, join(dirname(fileURLToPath(import.meta.url)), 'worker.js'));

const chunkSize = nodes / Math.sqrt(numProcessors);

const tasks = [];

for (let i = 0; i < nodes; i += chunkSize) {

for (let j = 0; j < nodes; j += chunkSize) {

tasks.push({

sharedArray,

rowStart: i,

rowEnd: Math.min(i + chunkSize, nodes),

colStart: j,

colEnd: Math.min(j + chunkSize, nodes),

});

}

}

for (let i = 0; i < tasks.length; i++) {

const taskPromises = [];

taskPromises.push(pool.submit(tasks[i]));

await Promise.all(taskPromises);

}

await pool.shutdown();

const resultMatrix = [];

for (let i = 0; i < nodes; i++) {

resultMatrix.push(Array.from(sharedArray.slice(i \* nodes, (i + 1) \* nodes)));

resultMatrix[i] = resultMatrix[i].map((value) => (value === INF ? Infinity : value));

}

return resultMatrix;

}

Файл ThreadPool.js

import { Worker } from 'worker\_threads';

export class ThreadPool {

constructor(size, path) {

this.size = size;

this.workers = [];

this.tasks = [];

this.shuttingDown = false;

for (let i = 0; i < size; i++) {

this.workers.push(new Worker(path));

}

}

submit(task) {

return new Promise((resolve, reject) => {

if (this.shuttingDown) {

reject(new Error('ThreadPool is shutting down'));

return;

}

if (this.workers.length === 0) {

this.tasks.push({ task, resolve, reject });

} else {

const worker = this.workers.pop();

const handleMessage = (result) => {

worker.removeListener('message', handleMessage);

resolve(result);

this.workers.push(worker);

if (this.tasks.length > 0) {

const { task, resolve, reject } = this.tasks.shift();

this.submit(task).then(resolve).catch(reject);

}

};

worker.on('message', handleMessage);

worker.postMessage(task);

}

});

}

async shutdown() {

if (this.shuttingDown) {

return Promise.resolve();

}

this.shuttingDown = true;

const terminationPromises = this.workers.map(worker => new Promise((resolve, reject) => {

worker.once('exit', () => resolve());

worker.terminate();

}));

return Promise.all(terminationPromises).then(() => {

this.workers = [];

});

}

}

Файл Worker.js

import { parentPort } from 'worker\_threads';

parentPort.on('message', (task) => {

const { sharedArray, rowStart, rowEnd, colStart, colEnd } = task;

const nodes = Math.sqrt(sharedArray.length);

for (let k = 0; k < nodes; k++) {

for (let i = rowStart; i < rowEnd; i++) {

for (let j = colStart; j < colEnd; j++) {

const currentDist = sharedArray[i \* nodes + j];

const newDist = sharedArray[i \* nodes + k] + sharedArray[k \* nodes + j];

sharedArray[i \* nodes + j] = Math.min(currentDist, newDist);

}

}

}

parentPort.postMessage('done');

});

Файл speed.js

import { floydWarshall } from "./floydWarshall.js";

import { createGraph } from "./graph.js";

import { parallelFloydWarshall } from "./parallelFloydWarshall.js";

import { performance } from 'perf\_hooks';

const measureExecutionTimeP = (callback) => {

return new Promise((resolve, reject) => {

const startTime = performance.now();

callback().then(() => {

const endTime = performance.now();

resolve(endTime - startTime);

}).catch(reject);

});

};

const measureExecutionTime = (callback) => {

const startTime = performance.now();

callback();

const endTime = performance.now();

return endTime - startTime;

};

export const generateRandomGraph = (numNodes, density = 0.5, maxWeight = 10) => {

const edges = [];

for (let i = 0; i < numNodes; i++) {

for (let j = i + 1; j < numNodes; j++) {

if (Math.random() < density) {

const weight = Math.floor(Math.random() \* maxWeight) + 1;

edges.push([i, j, weight]);

edges.push([j, i, weight]);

}

}

}

return createGraph(numNodes, edges);

};

export const speedFloydWarshall = (graphSizes) => {

for (const numNodes of graphSizes) {

const graph = generateRandomGraph(numNodes, 0.3);

console.log(`Testing Floyd-Warshall with ${numNodes} nodes...`);

const executionTime = measureExecutionTime(() => {

floydWarshall(graph);

});

console.log(`Execution time: ${executionTime.toFixed(2)} ms\n`);

}

};

export const speedParallelFloydWarshall = async (graphSizes, processors) => {

for (const numNodes of graphSizes) {

for (const numProcessors of processors) {

const graph = generateRandomGraph(numNodes, 0.3);

console.log(`Testing Parallel Floyd-Warshall with ${numNodes} nodes and ${numProcessors} processors...`);

const executionTime = await measureExecutionTimeP(async () => {

await parallelFloydWarshall(graph, numProcessors);

});

console.log(`Execution time with ${numProcessors} processors: ${executionTime.toFixed(2)} ms\n`);

}

}

};

export const speedUp = async (graphSizes, numProcessors, repetitions) => {

for (const numNodes of graphSizes) {

let sequentialExecutionTimes = [];

let parallelExecutionTimes = [];

const graph = generateRandomGraph(numNodes, 0.3);

for (let i = 0; i < repetitions; i++) {

console.log(`Sequential Floyd-Warshall with ${numNodes} nodes (Repetition ${i + 1})`);

const sequentialExecutionTime = measureExecutionTime(() => {

floydWarshall(graph);

});

sequentialExecutionTimes.push(sequentialExecutionTime);

console.log(`Execution time with ${numNodes} nodes: ${sequentialExecutionTime.toFixed(2)} ms\n`);

console.log(`Parallel Floyd-Warshall with ${numNodes} nodes and ${numProcessors} processors (Repetition ${i + 1})`);

const executionTime = await measureExecutionTimeP(async () => {

await parallelFloydWarshall(graph, numProcessors);

});

parallelExecutionTimes.push(executionTime);

console.log(`Execution time with ${numNodes} nodes: ${executionTime.toFixed(2)} ms\n`);

}

const avgSequentialExecutionTime = sequentialExecutionTimes.reduce((a, b) => a + b, 0) / repetitions;

console.log(`Average Sequential Execution Time: ${avgSequentialExecutionTime}`)

const avgParallelExecutionTime = parallelExecutionTimes.reduce((a, b) => a + b, 0) / repetitions;

console.log(`Average Parallel Execution Time: ${avgParallelExecutionTime}`)

const speedUp = avgSequentialExecutionTime / avgParallelExecutionTime;

console.log(`Average Speed-up with ${numNodes} nodes: ${speedUp.toFixed(2)}\n`);

}

}