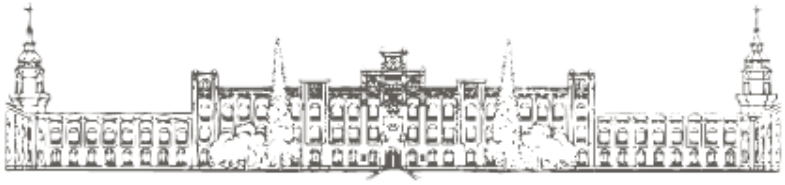


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: Алгоритм Флойда-Уоршелла пошуку найкоротших шляхів у графі та його паралельна реалізація мовою C#

Керівник:
PhD, ст.викл. А.Ю.Дифучин

«Допущено до захисту»
_____

«___» _____ 2024 р.

Виконавець:
Бондарчук Анастасія Олександрівна
студент групи ІП-12
залікова книжка № ІП-1204
_____

«16» травня 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Антон ДИФУЧИН

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, додати посилання на використані джерела інформації. Зробити висновок щодо актуальності виконаного дослідження.

2. Виконати розробку послідовного алгоритму у відповідності до варіанту

завдання та обраного програмного забезпечення для реалізації. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.

3. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.

4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.

5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.

6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень.

7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 29 сторінок, 20 рисунків, 4 таблиці, 6 посилань

Курсова робота присвячена алгоритму Флойда-Уоршелла [1, с.655-662] та його паралельній реалізації мовою програмування C# [2].

Алгоритм Флойда-Уоршелла є класичним алгоритмом для знаходження найкоротших шляхів в графі. У роботі досліджуються різні підходи до реалізації цього алгоритму, зокрема класичний варіант та реалізація з використанням багатопоточних технологій.

Для реалізації паралельної версії алгоритму використовуються механізм паралельного програмування у мові програмування C#, такий як клас Parallel [3].

Алгоритми порівнюються за швидкодією та правильністю результатів при обробці як малих, так і великих графів, тестуються на випадково заданих даних різної розмірності.

Результати дослідження можуть бути корисними для розробників, які працюють з великими обсягами даних та шукають оптимальні методи обробки цих даних для швидкодії та масштабованості.

Ключові слова: ПАРАЛЕЛЬНИЙ АЛГОРИТМ, ГРАФ, ПОШУК НАЙКОРОТШИХ ШЛЯХІВ, ФЛОЙД, УОРШЕЛЛ, C#, TASK PARALLEL LIBRARY, CLASS PARALLEL, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

ЗМІСТ

ВСТУП

Сьогодні, коли обсяги даних щохвилини зростають, швидкість їх обробки визначає успіх у різних галузях. В цьому контексті використання паралельних обчислень набуває все більшого значення. Сучасні комп'ютери, з їх багатоядерними процесорами та підтримкою багатопоточності, відкривають нові можливості для одночасної обробки даних у кількох потоках.

Паралельні обчислення дозволяють ефективно розподіляти обчислювальні завдання між різними ресурсами, що сприяє зменшенню часу виконання завдань та оптимальному використанню потенційних можливостей обчислювальних систем. Сьогодні паралельні та розподілені обчислення стали необхідною складовою у багатьох областях, що забезпечує їх поширення та популярність.

Однією з важливих задач, яку можна вирішити за допомогою паралельних обчислень, є пошук найкоротших шляхів у графі. Алгоритм Флойда-Уоршелла є ефективним засобом розв'язання цієї задачі, а його паралельна реалізація відкриває нові можливості для швидкого пошуку найкоротших шляхів у графі, особливо при великих обсягах даних.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм Флойда-Уоршелла – це ефективний алгоритм для пошуку найкоротших шляхів у зваженому графі, який може мати як додатні ваги, так і від'ємні. Основна ідея алгоритму полягає в тому, щоб шукати найкоротші шляхи між будь-якою парою вершин, проходячи через усі інші вершини графу.

Проблема: пошук усіх найкоротших шляхів та їх довжин(ваг) між усіма вершинами графу.

Вхідні дані: матриця (двовимірний масив) суміжності, що задає граф.

Вихідні дані: матриця суміжності з довжинами найкоротших шляхів; матриця проміжних вершин для кожного шляху.

Для реалізації паралельної версії алгоритму необхідно обрати спосіб його розпаралелювання. Тому для початку розглянемо оригінальну послідовну версію алгоритму Флойда-Уоршелла.

1.1 Послідовний алгоритм

Порядок дій:

1) Ініціалізація матриці суміжності таким чином, щоб вона відображала ваги ребер між вершинами графа.
2) Прохід по всіх вершинах і оновлення ребер зменшеними відстанями між вершинами та оновлення проміжної вершини, якщо це можливо.
3) Після завершення першої ітерації алгоритму, отримуємо найкоротші відстані між усіма парами вершин, які можуть проходити через одну і ту ж проміжну вершину, а також поточні проміжні вершини у знайдених шляхах.
4) Повторення процесу для всіх проміжних вершин, поки не знайдено найкоротші шляхи між усіма парами вершин.

1.2 Паралельні реалізації

Звичайний алгоритм Флойда-Уоршелла складається з трьох вкладених циклів, що працюють над матрицею відстаней. Однак для паралельної реалізації цього алгоритму існують різні підходи:

Паралелізація за зовнішнім циклом:

Суть: У цьому варіанті паралелізації кожен ітераційний крок зовнішнього циклу, що відповідає за проходження чергової вершини k через всі інші вершини, обробляється паралельно.

Паралелізація по рядках(стовпцях):

Суть: У цьому варіанті паралелізації кожен рядок(стовпець) матриці відстаней обробляється окремим обчислювальним потоком або процесом. Кожен потік відповідає за обчислення найкоротших відстаней від однієї вершини до всіх інших.

У кожного з цих підходів є свої переваги та недоліки, і вибір конкретного підходу залежить від характеристик системи, на якій відбувається паралельне обчислення, та від специфіки задачі.

Також для реалізації паралельного алгоритму використовують засоби MPI, проте дослідження стандарту MPI виходить за рамки теми курсової роботи.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Псевдокод послідовного алгоритму

Псевдокод:

```
1. procedure FloydWarshall(graph[][], vertices, path[][])
//заповнення матриці довжин шляхів даними з матриці суміжності графа
2. distance[][] = graph[][]
//цикл проходу по усіх проміжних вершинах
4.     for k from 1 to vertices do
//цикл проходу по усіх рядках
5.         for i from 1 to vertices do
//цикл проходу по усіх стовпцях
6.             for j from 1 to vertices do
//оновлення відстані та проміжної вершини, якщо шлях через вершину k коротший за
поточний від i до j
7.                 if distance[i][k] + distance[k][j] < distance[i][j] then
8.                    distance[i][j] = distance[i][k] + distance[k][j]
9.                    path[i][j] = path[i][k]
10.            endfor
11.        endfor
12.    endfor
//повернення оновленої матриці шляхів між вершинами
13. return distance[][]
14. end procedure
```

2.2 Реалізація послідовного алгоритму

Для реалізації безпосередньо алгоритму за заданим псевдокодом створено метод FloydWarshallOriginal() (рис. 2.1) та додаткові методи для ініціалізації графу та початкових матриць.

Метод FloydWarshallOriginal() приймає на вхід матрицю довжин шляхів, яка відповідає початковій матриці суміжності заданого графу, та початкову матрицю проміжних вершин. Алгоритм виконує потрійний цикл проходу по усіх вершинах та оновлює довжини мінімізуючи їх сумою шляхів через проміжні вершини, якщо це можливо, а також змінює в проміжні вершини, якщо шлях через обрану вершину виявився коротшим за наявний.

```
static double[,] FloydWarshallOriginal(double[,] dist, int[,] pathMatrix)
{
    int size = dist.GetLength(dimension: 0);

    for (int k = 0; k < size; k++)
    {
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (!double.IsPositiveInfinity(dist[i, k]) && !double.IsPositiveInfinity(dist[k, j]) &&
                    dist[i, k] + dist[k, j] < dist[i, j])
                {
                    dist[i, j] = dist[i, k] + dist[k, j];
                    pathMatrix[i, j] = pathMatrix[i, k];
                }
            }
        }
    }

    return dist;
}
```

Рисунок 2.1 – Реалізація послідовного алгоритму

Умова всередині циклів перевіряє довжини шляхів від поточних вершин до проміжної на наявність (при значення Infinity шлях відсутній), а також визначає чи є шлях через проміжну вершину менший за поточний. Якщо умова виконується, то відбувається заміна поточного шляху між вершинами на новий та зміна проміжної вершини між поточними вершинами на ту, яка веде до вершини k.

2.3 Тестування роботи послідовного алгоритму

Для тестування роботи алгоритму було створено методи:

- GenerateRandomGraph() – генерація випадкового графу (рис. 2.2);
- CreatePathMatrix() – створення матриці проміжних вершин (рис. 2.3);
- PrintMatrix() – виведення матриці на екран (рис. 2.4);
- PrintAllPaths() та PrintPath() – для пошуку усіх шляхів за допомогою матриці проміжних вершин та їх виведення (рис. 2.5, рис. 2.6).

```
1 usage
static double[,] GenerateRandomGraph(int size)
{
    Random rand = new Random();
    double[,] graph = new double[size, size];
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            int currentRand = rand.Next(-10, 10);
            if (i == j)
                graph[i, j] = 0;
            else if (currentRand <= -5)
            {
                graph[i, j] = double.PositiveInfinity;
            }
            else
            {
                graph[i, j] = rand.Next(maxValue: 100);
            }
        }
    }

    return graph;
}
```

Рисунок 2.2 – Генерація випадкового графу

```
static int[,] CreatePathMatrix(double[,] graph)
{
    int size = graph.GetLength(dimension: 0);
    int[,] pathMatrix = new int[size, size];
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            pathMatrix[i, j] = (double.IsPositiveInfinity(graph[i, j])) ? -1 : j;
        }
    }

    return pathMatrix;
}
```

Рисунок 2.3 – Створення матриці проміжних вершин

```
static void PrintMatrix(double[,] dist)
{
    int size = dist.GetLength(dimension: 0);
    Console.Write("\t");
    for (int i = 0; i < size; i++)
    {
        Console.Write($"[{i}]\t");
    }

    Console.WriteLine();
    for (int i = 0; i < size; ++i)
    {
        Console.Write($"[{i}]\t");
        for (int j = 0; j < size; ++j)
        {
            if (double.IsPositiveInfinity(dist[i, j]))
                Console.Write("INF\t");
            else
                Console.Write(dist[i, j] + "\t");
        }

        Console.WriteLine();
    }
}
```

Рисунок 2.4 – Виведення матриці в консоль

```
static void PrintAllPaths(int[,] pathMatrix)
{
    int size = pathMatrix.GetLength(dimension: 0);
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i != j)
            {
                List<int> path = new List<int> { i };
                Console.Write($"Path from [{i}] to [{j}]: ");
                PrintPath(from: i, to: j, pathMatrix, path);
            }
        }
    }
}
```

Рисунок 2.5 – Виведення усіх шляхів

```
2 usages
static void PrintPath(int from, int to, int[,] pathMatrix, List<int> path)
{
    if (from == to)
    {
        Console.WriteLine(string.Join(" -> ", path));
    }
    else if (pathMatrix[from, to] == -1)
    {
        Console.WriteLine("No path.");
    }
    else
    {
        path.Add(pathMatrix[from, to]);
        PrintPath(from: pathMatrix[from, to], to, pathMatrix, path);
    }
}
}
```

Рисунок 2.6 – Пошук та виведення шляху між двома вершинами

Для початку протестуємо алгоритм на невеликих обсягах даних, аби перевірити правильність його роботи.

Введемо розмір графу 5 та подивимось на згенеровану матрицю суміжності (рис. 2.7).

```
Enter the size of the matrix: 5
        [0]     [1]     [2]     [3]     [4]
[0]     0       18      18      INF     INF
[1]     50      0       50      18      69
[2]     24      INF     0       32      INF
[3]     31      INF     77      0       59
[4]     20      93      66      14      0
```

Рисунок 2.7 – Згенерована матриця суміжності графа

За згенерованою матрицею заповнимо граф на онлайн-ресурсі Visualizations of Graph Algorithms [4] (рис. 2.8).

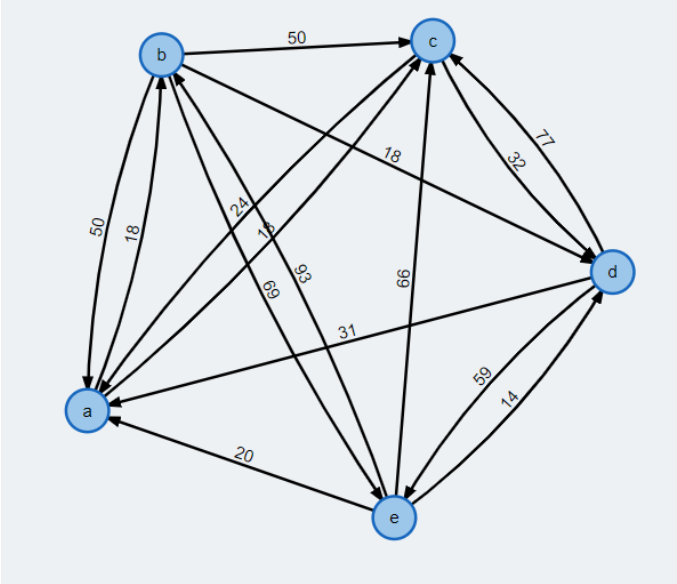


Рисунок 2.8 – Зображення графа, який відповідає згенерованій матриці

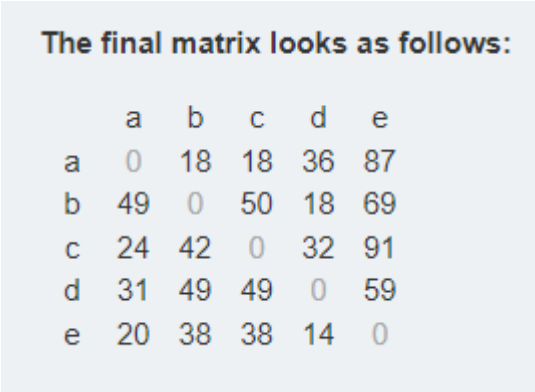
Перевіримо, чи співпадають згенерована програмним кодом матриця та матриця з інтернет-ресурсу (рис. 2.9).

The original matrix looks as follows:

	a	b	c	d	e
a	0	18	18	∞	∞
b	50	0	50	18	69
c	24	∞	0	32	∞
d	31	∞	77	0	59
e	20	93	66	14	0

Рисунок 2.9. – Матриця суміжності для створеного графу

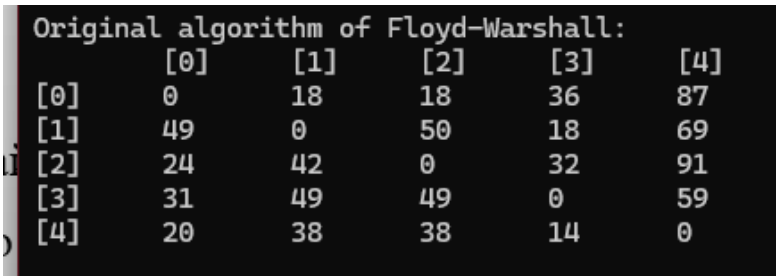
Запустивши алгоритм в онлайн-калькуляторі на виконання, отримуємо результат (рис. 2.10), який можемо порівняти із результатом, який отримали після виконання програмного коду (рис. 2.11).



The final matrix looks as follows:

	a	b	c	d	e
a	0	18	18	36	87
b	49	0	50	18	69
c	24	42	0	32	91
d	31	49	49	0	59
e	20	38	38	14	0

Рисунок 2.10 – Фінальний вигляд матриці в онлайн-калькуляторі

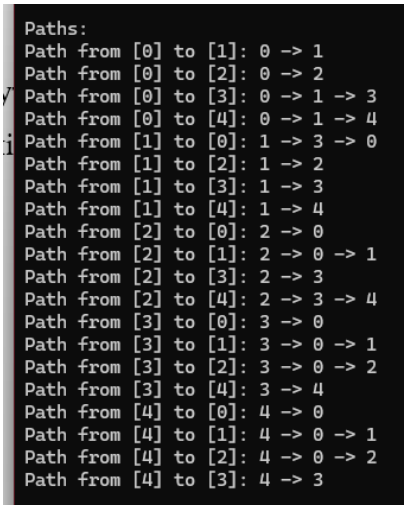


```
Original algorithm of Floyd-Warshall:
        [0]     [1]     [2]     [3]     [4]
[0]     0       18      18      36      87
[1]     49      0       50      18      69
[2]     24      42      0       32      91
[3]     31      49      49      0       59
[4]     20      38      38      14      0
```

Рисунок 2.11 – Фінальний вигляд матриці після виконання програмного коду

Бачимо, що результаті однакові. Отже послідовний алгоритм працює коректно.

Маємо змогу також переглянути усі шляхи між вершинами (рис. 2.12) та перевірити їх правильність, проаналізувавши зображення графу (рис. 2.8).



```
Paths:
Path from [0] to [1]: 0 -> 1
Path from [0] to [2]: 0 -> 2
Path from [0] to [3]: 0 -> 1 -> 3
Path from [0] to [4]: 0 -> 1 -> 4
Path from [1] to [0]: 1 -> 3 -> 0
Path from [1] to [2]: 1 -> 2
Path from [1] to [3]: 1 -> 3
Path from [1] to [4]: 1 -> 4
Path from [2] to [0]: 2 -> 0
Path from [2] to [1]: 2 -> 0 -> 1
Path from [2] to [3]: 2 -> 3
Path from [2] to [4]: 2 -> 3 -> 4
Path from [3] to [0]: 3 -> 0
Path from [3] to [1]: 3 -> 0 -> 1
Path from [3] to [2]: 3 -> 0 -> 2
Path from [3] to [4]: 3 -> 4
Path from [4] to [0]: 4 -> 0
Path from [4] to [1]: 4 -> 0 -> 1
Path from [4] to [2]: 4 -> 0 -> 2
Path from [4] to [3]: 4 -> 3
```

Рисунок 2.12 – Виведення усіх шляхів між вершинами

2.4 Аналіз швидкодії послідовного алгоритму

Часові результати роботи послідовного алгоритму представлено в таблиці 2.1.

Таблиця 2.1 – Час виконання послідовного алгоритму Флойда-Уоршелла в залежності від розміру початкової матриці (графу).

№ п/п	Розмірність матриці (графу), к-сть вершин	Швидкість виконання алгоритму, мкс
1	5	313,7
2	20	656,8
3	50	11 929,6
4	100	11 461,0
5	200	90 101,6
6	500	1 438 759,7
7	1000	12 704 156,2
8	1500	57 552 987,4

Виведемо отримані результати на графіку для наочності (рис. 2.13).

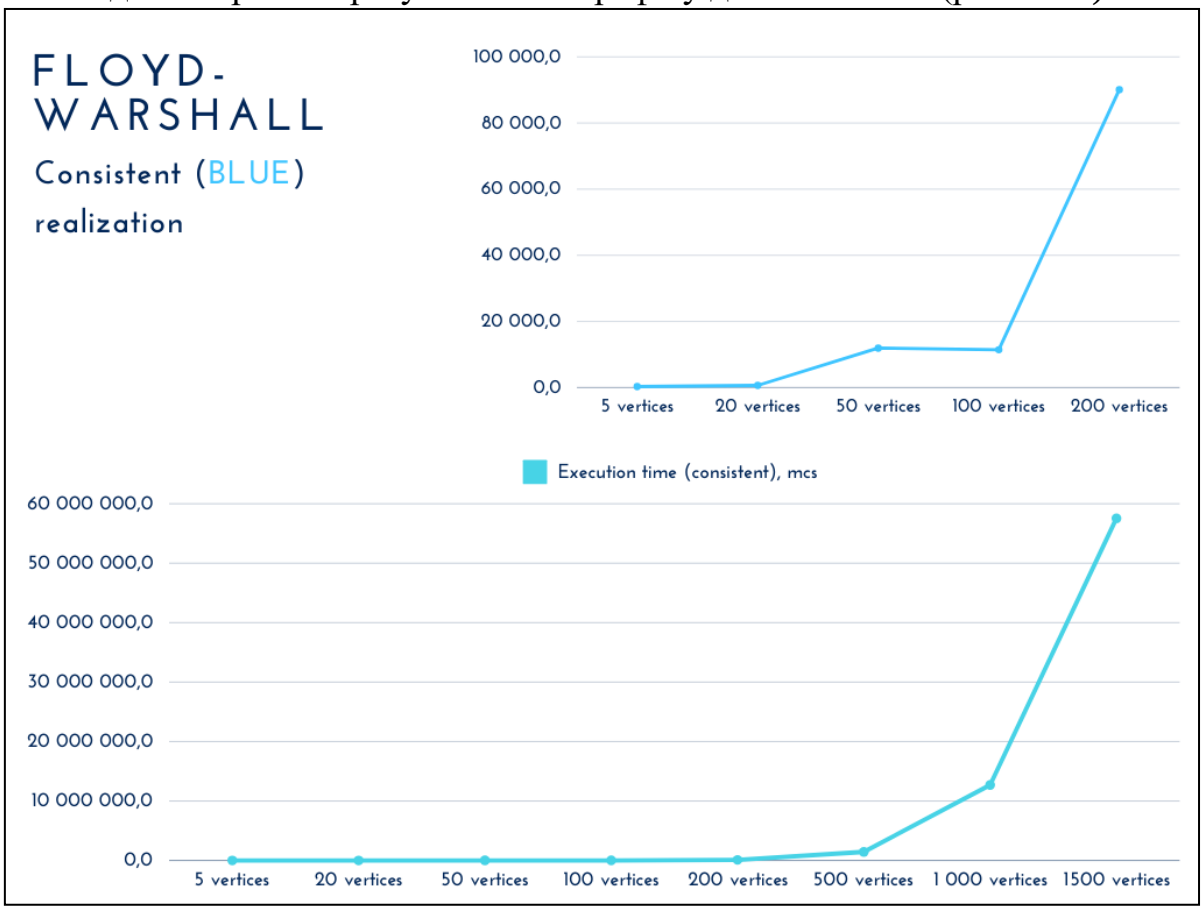


Рисунок 2.13 – Графік швидкодії послідовного алгоритму

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

Для розробки паралельної реалізації алгоритму Флойда-Уоршелла пошуку найкоротших шляхів у графі було обрано мову програмування C# з її засобами паралельного програмування. C# надає широкі можливості для розробки паралельних програм.

C# забезпечує інтеграцію з іншими технологіями Microsoft: часто використовується для розробки програмного забезпечення для платформ Microsoft, таких як Windows, .NET Framework або .NET Core. Тому в майбутньому розроблюваний програмний засіб можна буде легко інтегрувати в інші види програмного забезпечення.

Вибір програмного забезпечення для розробки паралельних обчислень у мові програмування C# може включати в себе використання таких інструментів, як бібліотека TPL (Task Parallel Library) [5] та клас Parallel.

TPL (Task Parallel Library) – це бібліотека в мові програмування C#, яка надає вбудовані інструменти для паралельного програмування. TPL дозволяє легко створювати та керувати паралельними завданнями (tasks), автоматично розподіляти їх на доступні потоки виконання та ефективно використовувати потоки пула.

Клас Parallel – це один із компонентів TPL, який дозволяє виконувати паралельні ітерації, обробку масивів та інші паралельні операції зручним і простим способом. Клас Parallel автоматично розподіляє роботу між доступними потоками виконання, що спрощує розробку паралельних програм.

Обираючи ці засоби для розробки алгоритму Флойда-Уоршелла, можна отримати значну перевагу у швидкості розробки та ефективності виконання завдань завдяки вбудованим можливостям паралельного програмування, що надаються TPL та класом Parallel.

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

4.1 Проєктування паралельної реалізації алгоритму

Проаналізувавши послідовний алгоритм, бачимо, що його безпосередня складність полягає у наявності потрійного вкладеного циклу, який і викликає сповільнення та навантаження системи під час обрахунків на великих обсягах даних. Отже варто розглядати можливість розпаралелення одного з циклів.

Одним з найбільш вдалих варіантів буде забезпечення багатопоточного виконання циклу обходу по рядках, тобто розділення матриці на частини по одному рядку або декількох рядках.

Порядок дій:

1) Ініціалізація матриці суміжності таким чином, щоб вона відображала ваги ребер між вершинами графа.
2) Обрання однієї проміжної вершини для порівняння.
3) Паралельне виконання обчислення внутрішнього циклу (по стовпцях) для кожного рядка (групи рядків).
4) Після завершення першої ітерації алгоритму, отримуємо найкоротші відстані між усіма парами вершин, які можуть проходити через одну і ту ж проміжну вершину, а також поточні проміжні вершини у знайдених шляхах.
5) Повторення процесу для всіх проміжних вершин, поки не знайдено найкоротші шляхи між усіма парами вершин.

4.2 Реалізація паралельного алгоритму Флойда-Уоршелла пошуку найкоротших шляхів у графі з використанням класу Parallel

Для реалізації паралельної версії алгоритму створено метод FloydWarshallParallel() (рис. 4.1), який виконує паралельне обчислення алгоритму відповідно до псевдокоду, наданого при описі послідовного алгоритму, та з використанням багатопоточної технології класу Parallel бібліотеки TPL.

Метод FloydWarshallParallel() приймає на вхід матрицю довжин шляхів, яка відповідає початковій матриці суміжності заданого графу, та початкову матрицю проміжних вершин. Усі дії, виконувані в алгоритмі, та їх суть повністю відповідають послідовному алгоритму, але цикл проходу по рядках замінений на паралельний варіант виконання ітерацій циклу Parallel.For [6].

Оскільки матриця ділиться на рядки (групи рядків), які обробляються паралельно, та кожен прохід по рядках та стовпцях відбувається всередині циклу по усіх вершинах k, то кожен потік має вільний доступ лише до своєї частини матриці, не змінюючи при цьому усі інші її елементи. Отже необхідності у створенні локальних копій матриць та зайвій синхронізації процесу немає.

```
1 usage
static double[,] FloydWarshallParallel(double[,] dist, int[,] pathMatrix)
{
    int size = dist.GetLength(dimension: 0);

    for (int k = 0; k < size; k++)
    {
        Parallel.For(fromInclusive: 0, toExclusive: size, body: i :int =>
        {
            for (int j = 0; j < size; j++)
            {
                if (!double.IsPositiveInfinity(dist[i, k]) && !double.IsPositiveInfinity(dist[k, j]) &&
                    dist[i, k] + dist[k, j] < dist[i, j])
                {
                    dist[i, j] = dist[i, k] + dist[k, j];
                    pathMatrix[i, j] = pathMatrix[i, k];
                }
            }
        });
    }


    return dist;
}
```

Рисунок 4.1 – Реалізація паралельного алгоритму

4.3 Тестування роботи паралельного алгоритму

Для тестування алгоритму використовуються вже раніше описані методи. Застосуємо їх та перевіримо правильність роботи паралельного алгоритму на тих самих вхідних даних (рис. 4.2).

```
Parallel algorithm of Floyd-Warshall (by splitting the matrix into rows):
        [0]     [1]     [2]     [3]     [4]
[0]     0       18      18      36      87
[1]     49      0       50      18      69
[2]     24      42      0       32      91
[3]     31      49      49      0       59
[4]     20      38      38      14      0

Paths:
Path from [0] to [1]: 0 -> 1
Path from [0] to [2]: 0 -> 2
Path from [0] to [3]: 0 -> 1 -> 3
Path from [0] to [4]: 0 -> 1 -> 4
Path from [1] to [0]: 1 -> 3 -> 0
Path from [1] to [2]: 1 -> 2
Path from [1] to [3]: 1 -> 3
Path from [1] to [4]: 1 -> 4
Path from [2] to [0]: 2 -> 0
Path from [2] to [1]: 2 -> 0 -> 1
Path from [2] to [3]: 2 -> 3
Path from [2] to [4]: 2 -> 3 -> 4
Path from [3] to [0]: 3 -> 0
Path from [3] to [1]: 3 -> 0 -> 1
Path from [3] to [2]: 3 -> 0 -> 2
Path from [3] to [4]: 3 -> 4
Path from [4] to [0]: 4 -> 0
Path from [4] to [1]: 4 -> 0 -> 1
Path from [4] to [2]: 4 -> 0 -> 2
Path from [4] to [3]: 4 -> 3
```

Рисунок 4.2 – Результати виконання паралельного алгоритму

Бачимо, що результати співпадають з послідовним алгоритмом та результатами обробки матриці через онлайн-калькулятор. Можемо стверджувати, що алгоритм працює правильно.

Оскільки на великих обсягах даних наочно порівнювати результати практично неможливо, створимо також методи для програмного порівняння результатів роботи двох варіацій алгоритму:

– CompareLengthResults() – порівняння матриць довжин найкоротших шляхів (рис. 4.3);
– ComparePathResults() – порівняння матриць проміжних вершин (рис. 4.4).

```
static void CompareLengthResults(double[,] originalResult, double[,] parallelResult)
{
    int size = originalResult.GetLength(dimension: 0);
    int compareCounter = 0;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (Math.Abs(originalResult[i, j] - parallelResult[i, j]) > 0)
            {
                compareCounter++;
            }
        }
    }

    Console.WriteLine($"There are {compareCounter} difference between original and parallel algorithms' lengths.");
}
```

Рисунок 4.3 – Порівняння матриць довжин шляхів

```
1 usage
static void ComparePathResults(int[,] originalPath, int[,] parallelPath)
{
    int size = originalPath.GetLength(dimension: 0);
    int compareCounter = 0;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (Math.Abs(originalPath[i, j] - parallelPath[i, j]) > 0)
            {
                compareCounter++;
            }
        }
    }

    Console.WriteLine($"There are {compareCounter} difference between original and parallel algorithms' paths.");
}
```

Рисунок 4.4 – Порівняння матриць проміжних вершин

Застосувавши описані методи бачимо, що різниці між результатами роботи обох алгоритмів дійсно немає (рис. 4.5).

```
Comparing results between lengths:
There are 0 difference between original and parallel algorithms' lengths.

Comparing results between paths:
There are 0 difference between original and parallel algorithms' paths.
```

Рисунок 4.5 – Порівняння результатів роботи двох алгоритмів

4.4 Аналіз швидкодії паралельного алгоритму

Часові результати роботи паралельного алгоритму представлено в таблиці 4.1.

Таблиця 4.1 – Час виконання паралельного алгоритму Флойда-Уоршелла в залежності від розміру початкової матриці (графу).

№ п/п	Розмірність матриці (графу), к-сть вершин	Швидкість виконання алгоритму, мкс
1	5	2669,1
2	20	4791,2
3	50	11 188,6
4	100	13 039,7
5	200	52 260,6
6	500	648 140,7
7	1000	6 225 137,9
8	1500	32 925 539,6

Виведемо отримані результати на графік для наочності (рис. 4.6).

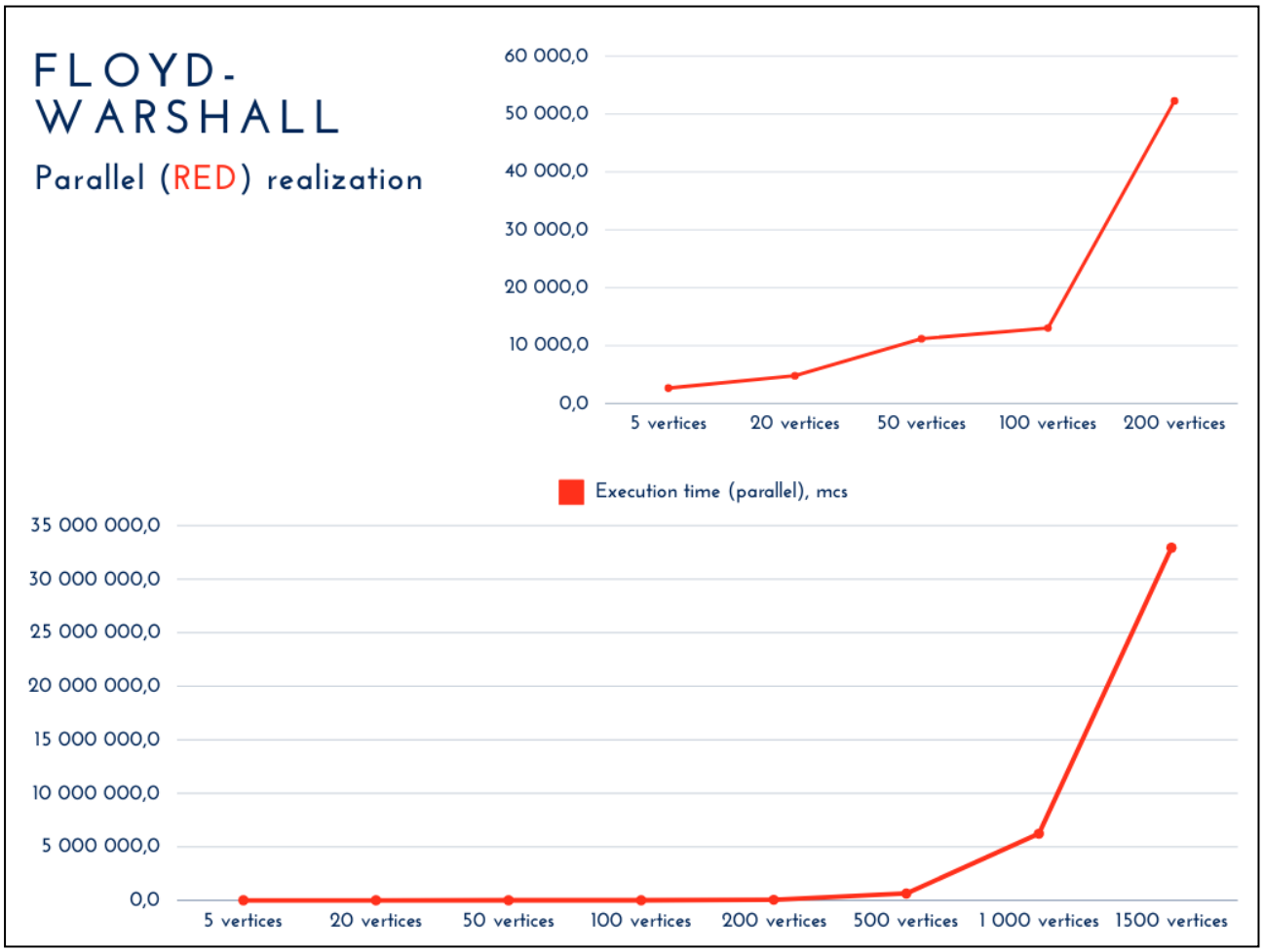


Рисунок 4.6 – Графік швидкодії паралельного алгоритму

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Дослідження послідовного та паралельного алгоритмів виконувались на апаратному забезпеченні з наступними характеристиками:

- 11th Gen Intel(R) Core(TM) i7-1165G7 @2.80GHz 1.69GHz (4 ядра)
- Обсяг оперативної пам’яті: 16 Gb

Для встановлення ефективності послідовного та паралельного алгоритмів було виконано декілька експериментів, результати яких представлено в таблиці 5.1.

Таблиця 5.1. – Виміри часу для послідовного та паралельного алгоритму на різних обсягах вхідних даних.

№ п/п	Кількість елементів	Час послідовного алгоритму, мікросекунд	Час паралельного алгоритму, мікросекунд
1	5	313,7	2 669,1
2	20	656,8	4 791,2
3	50	11 929,6	11 188,6
4	100	11 461,0	13 039,7
5	200	90 101,6	52 260,6
6	500	1 438 759,7	648 140,7
7	1000	12 704 156,2	6 225 137,9
8	1500	57 552 987,4	32 925 539,6

Виведемо отримані результати на графік та порівняємо їх (рис. 5.1).

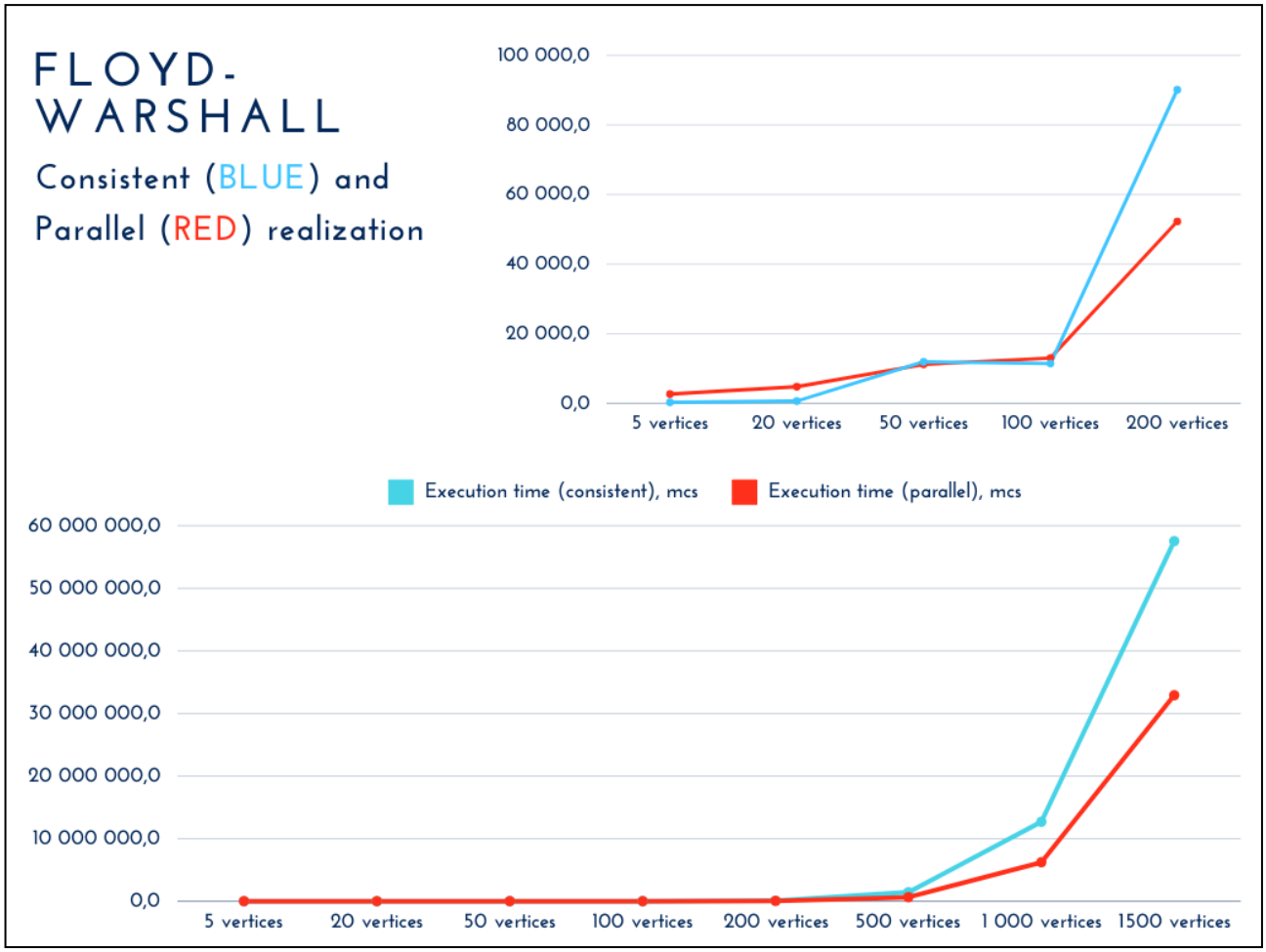


Рисунок 5.1 – Порівняльний графік швидкості роботи двох реалізацій алгоритму

На графіку наочно видно, що швидкість роботи паралельного алгоритму на достатньо великих обсягах даних помітно більша, а час роботи, відповідно, менший.

Обчислимо прискорення паралельного алгоритму відносно послідовного (табл. 5.2).

Таблиця 5.2 – Прискорення паралельного алгоритму відносно послідовного.

№ п/п	Кількість елементів	Прискорення паралельного алгоритму відносно послідовного
1	5	0,117
2	20	0,137
3	50	1,066

Продовження таблиці 5.2:

№ п/п	Кількість елементів	Прискорення паралельного алгоритму відносно послідовного
4	100	0,878
5	200	1,724
6	500	2,219
7	1000	2,040
8	1500	1,747

Як бачимо у таблиці, вже починаючи з 200 вершин прискорення становить 1,7, що задовольняє умову «>1,2».

Можна дійти висновку, що на графах 200+ елементів (а це приблизно 8 мільйонів ітерацій) паралельна реалізація алгоритму Флойда-Уоршелла пошуку найкоротших шляхів у графі має швидкісну перевагу над його послідовною версією.

ВИСНОВКИ

В ході виконання курсової роботи було проведено дослідження алгоритму Флойда-Уоршелла, який призначений для знаходження найкоротших шляхів у графі. Цей алгоритм є ефективним і широко використовується у різних сферах, де необхідно визначити оптимальний маршрут між вершинами графа.

Було досліджено та реалізовано паралельний варіант алгоритму за допомогою мови програмування C# та її засобів роботи з багатопоточністю. Зокрема було використано технології бібліотеки TPL, а саме клас Parallel та його метод Parallel.For, який слугує для паралельного виконання ітерацій циклу.

Використання багатопоточності при розробці алгоритму дозволило розподілити обчислення між різними потоками і, як наслідок, прискорити виконання обчислень на великих обсягах даних практично вдвічі.

В ході експериментів було доведено доцільність використання паралельних технологій при обрахунках на графах з великою кількістю вершин (>200).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Introduction to Algorithms Fourth Edition. The MIT Press, 2022. URL: https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf.

2. C# language documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/.

3. Data Parallelism (Task Parallel Library). URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library.

4. Visualizations of Graph Algorithms. URL: https://algorithms.discrete.ma.tum.de.

5. Task Parallel Library (TPL). URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl.

6. How to: Write a Simple Parallel.For Loop. URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-for-loop.

ДОДАТКИ

Додаток А. Код програми

GitHub репозиторій: https://github.com/NastasaBondarchuck/FloydWarshall

```
using System.Diagnostics;

class FloydWarshall
{
    static void Main()
    {
        Console.Write("Enter the size of the matrix: ");
        int size = int.Parse(Console.ReadLine()!);

        double[,] graph = GenerateRandomGraph(size);
        PrintMatrix(graph);

        int[,] originalPaths = CreatePathMatrix(graph);
        Console.WriteLine("\nOriginal algorithm of Floyd-Warshall:");
        double[,] dist = new double[size, size];
        Array.Copy(graph, dist, graph.Length);
        Stopwatch originalWatch = Stopwatch.StartNew();
        double[,] originalResult = FloydWarshallOriginal(dist, originalPaths);
        originalWatch.Stop();
        PrintMatrix(originalResult);
        Console.WriteLine("\nPaths:");
        PrintAllPaths(originalPaths);

        int[,] parallelPaths = CreatePathMatrix(graph);
        Console.WriteLine("\nParallel algorithm of Floyd-Warshall (by splitting
the matrix into rows):");
        dist = new double[size, size];
        Array.Copy(graph, dist, graph.Length);
        Stopwatch parallelWatch = Stopwatch.StartNew();
        double[,] parallelResult = FloydWarshallParallel(dist, parallelPaths);
        parallelWatch.Stop();
        PrintMatrix(parallelResult);
        Console.WriteLine("\nPaths:");
        PrintAllPaths(parallelPaths);

        Console.WriteLine("\nComparing results between lengths:");
        CompareLengthResults(originalResult, parallelResult);
        Console.WriteLine("\nComparing results between paths:");
        ComparePathResults(originalPaths, parallelPaths);

        Console.WriteLine("\nExecution time:");
        Console.WriteLine("Original algorithm: " + originalWatch.Elapsed + ".");
        Console.WriteLine("Parallel algorithm: " + parallelWatch.Elapsed + ".");

        Console.Read();
    }
    static double[,] GenerateRandomGraph(int size)
    {
        Random rand = new Random();
        double[,] graph = new double[size, size];
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                int currentRand = rand.Next(-10, 10);
                if (i == j)
```

```
                    graph[i, j] = 0;
                else if (currentRand <= -5)
                {
                    graph[i, j] = double.PositiveInfinity;
                }
                else
                {
                    graph[i, j] = rand.Next(100);
                }
            }
        }

        return graph;
    }
    static int[,] CreatePathMatrix(double[,] graph)
    {
        int size = graph.GetLength(0);
        int[,] pathMatrix = new int[size, size];
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                pathMatrix[i, j] = (double.IsPositiveInfinity(graph[i, j])) ? -1
: j;
            }
        }

        return pathMatrix;
    }

    static double[,] FloydWarshallOriginal(double[,] dist, int[,] pathMatrix)
    {
        int size = dist.GetLength(0);

        for (int k = 0; k < size; k++)
        {
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    if (!double.IsPositiveInfinity(dist[i, k]) &&
!double.IsPositiveInfinity(dist[k, j]) &&
                        dist[i, k] + dist[k, j] < dist[i, j])
                    {
                        dist[i, j] = dist[i, k] + dist[k, j];
                        pathMatrix[i, j] = pathMatrix[i, k];
                    }
                }
            }
        }

        return dist;
    }

    static double[,] FloydWarshallParallel(double[,] dist, int[,] pathMatrix)
    {
        int size = dist.GetLength(0);

        for (int k = 0; k < size; k++)
        {
            Parallel.For(0, size, i =>
            {
                for (int j = 0; j < size; j++)
```

```
                {
                    if (!double.IsPositiveInfinity(dist[i, k]) &&
!double.IsPositiveInfinity(dist[k, j]) &&
                        dist[i, k] + dist[k, j] < dist[i, j])
                    {
                        dist[i, j] = dist[i, k] + dist[k, j];
                        pathMatrix[i, j] = pathMatrix[i, k];
                    }
                }
            });
        }


        return dist;
    }

    static void PrintMatrix(double[,] dist)
    {
        int size = dist.GetLength(0);
        Console.Write("\t");
        for (int i = 0; i < size; i++)
        {
            Console.Write($"[{i}]\t");
        }

        Console.WriteLine();
        for (int i = 0; i < size; ++i)
        {
            Console.Write($"[{i}]\t");
            for (int j = 0; j < size; ++j)
            {
                if (double.IsPositiveInfinity(dist[i, j]))
                    Console.Write("INF\t");
                else
                    Console.Write(dist[i, j] + "\t");
            }

            Console.WriteLine();
        }
    }

    static void CompareLengthResults(double[,] originalResult, double[,]
parallelResult)
    {
        int size = originalResult.GetLength(0);
        int compareCounter = 0;
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (Math.Abs(originalResult[i, j] - parallelResult[i, j]) > 0)
                {
                    compareCounter++;
                }
            }
        }

        Console.WriteLine($"There are {compareCounter} difference between
original and parallel algorithms' lengths.");
    }

    static void ComparePathResults(int[,] originalPath, int[,] parallelPath)
    {
```

```
        int size = originalPath.GetLength(0);
        int compareCounter = 0;
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (Math.Abs(originalPath[i, j] - parallelPath[i, j]) > 0)
                {
                    compareCounter++;
                }
            }
        }

        Console.WriteLine($"There are {compareCounter} difference between
original and parallel algorithms' paths.");
    }

    static void PrintAllPaths(int[,] pathMatrix)
    {
        int size = pathMatrix.GetLength(0);
        for (int i = 0; i < size; i++)
        {
            for (int j = 0; j < size; j++)
            {
                if (i != j)
                {
                    List<int> path = new List<int> { i };
                    Console.Write($"Path from [{i}] to [{j}]: ");
                    PrintPath(i, j, pathMatrix, path);
                }
            }
        }
    }

    static void PrintPath(int from, int to, int[,] pathMatrix, List<int> path)
    {
        if (from == to)
        {
            Console.WriteLine(string.Join(" -> ", path));
        }
        else if (pathMatrix[from, to] == -1)
        {
            Console.WriteLine("No path.");
        }
        else
        {
            path.Add(pathMatrix[from, to]);
            PrintPath(pathMatrix[from, to], to, pathMatrix, path);
        }
    }
}
```