

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ

Лабораторна робота №3

з дисципліни «Високопродуктивні обчислення»

Тема роботи: «Реалізація за допомогою засобів OpenMP паралельних обчислень у комбінаторному методі визначення вектора пріоритетів (ваг об'єктів) на основі неповної матриці експертних парних порівнянь»

Виконала студентка

групи КН-31

Промоцька А.А.

Перевірів:

проф. Циганок В.В.

Київ – 2023

Мета: Освоїти алгоритмізацію паралельних обчислень та їх реалізацію за допомогою засобів OpenMP на практичному прикладі комбінаторного методу визначення вектора пріоритетів (ваг об'єктів) на основі неповної матриці експертних парних порівнянь. Набути практичних навичок створення ефективних паралельних програм з використанням технології OpenMP. Уміти використовувати стандартний інструментарій тестування створеного коду та оцінювати ефективність коду за множиною заданих критеріїв.

Хід роботи

Завдання №1

Представимо матрицю експертних парних порівнянь через дві матриці – матриця пар вершин, між якими існує ребро, та матриця ваг ребер:

```
vector<pair<int, int>> graphEdges = {
    {0, 1}, {0, 2}, {0, 3}, {0, 4}, {0, 5}, {0, 6},
    {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6},
    {2, 3}, {2, 4}, {2, 5}, {2, 6},
    {3, 4}, {3, 5}, {3, 6},
    {4, 5}, {4, 6},
    {5, 6}
};

vector <vector <double>> edgeWeights = {
    {1, 2, 3, 4, 2, 6, 7},
    {0.5, 1, 4, 3, 2, 4, 8},
    {0.33, 0.25, 1, 0.33, 2, 3, 4},
    {0.25, 0.33, 3, 1, 5, 7, 4},
    {0.5, 0.5, 0.5, 0.2, 1, 2, 5},
    {0.17, 0.25, 0.33, 0.14, 0.5, 1, 3},
    {0.14, 0.125, 0.25, 0.25, 0.2, 0.33, 1}
};
```

Таке представлення зумовлено обраним алгоритмом перебору усіх покривних дерев – алгоритм грубої сили.

Суть алгоритму полягає у тому, щоб отримати усі можливі комбінації шляху у графі та перевіряти комбінацію ребр на зв'язність – чи утворює вона покривне дерево.

Переваги цього алгоритму:

Гарантований результат: Гарантує, що усі покривні дерева графа будуть знайдені. Алгоритм ітерується через всі можливі комбінації ребер, тому не пропускає жодного покривного дерева.

Простота реалізації: Реалізація алгоритму грубої сили зазвичай досить проста. Вона може бути легко реалізована для графів невеликих розмірів.

Функція, в якій реалізований алгоритм грубої сили:

```
void generateAllSpanningTrees() {
    // Generate all possible subsets of edges of size V-1 (the number of edges in a
    // spanning tree)
    vector<bool> bitmask(graphEdges.size());
    fill(bitmask.end() - (V - 1), bitmask.end(), true);

    do {
        vector<pair<int, int>> edgeSubset;
        for (int i = 0; i < graphEdges.size(); ++i) {
            if (bitmask[i]) {
                edgeSubset.push_back(graphEdges[i]);
            }
        }

        // checking subset for being a spanning tree
        if (isSpanningTree(edgeSubset)) {
            spanningTrees.push_back(edgeSubset);
        }

    } while (next_permutation(bitmask.begin(), bitmask.end()));

    cout << "Number of trees: " << spanningTrees.size();
}
```

Функція, в якій реалізована перевірка комбінації ребер на зв'язність:

```
bool isSpanningTree(const vector<pair<int, int>>& edges) {
    vector<int> parent(V);
    iota(parent.begin(), parent.end(), 0);

    // Find root of the set that vertex i is in
    auto find = [&parent](int i) {
        while (i != parent[i]) {
            i = parent[i];
        }
        return i;
    };

    // Union of two sets. Returns false if x and y are already in the same set
    auto unionSets = [&parent, &find](int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX == rootY) {
            return false;
        }
        parent[rootY] = rootX;
        return true;
    };
};
```

```

    // Check if all edges can be added without forming a cycle
    for (const auto& edge : edges) {
        if (!unionSets(edge.first, edge.second)) {
            return false; // Cycle detected
        }
    }

    // Check if all vertices are connected
    int root = find(0);
    for (int i = 1; i < V; ++i) {
        if (find(i) != root) {
            return false; // Not all vertices are connected
        }
    }

    return true;
}

```

Вона використовує алгоритм пошуку з об'єднанням множин для перевірки, чи можна додати всі ребра без утворення циклу, і перевіряє, чи всі вершини з'єднані. Якщо підмножина задовольняє ці умови, вона є остовним деревом.

Для реалізації генерації матриць, що відповідають покривним деревам, був розроблений власний алгоритм реалізації транзитивного правила заповнення. Його суть полягає у тому, що коефіцієнт k обирається на основі найбільш заповненого стовпця або рядка матриці:

```

void makeSpanningTreeMatrixReflexive() {
    for (int i = 0; i < spanningTreeMatrix.size(); ++i) {
        for (int j = 0; j < spanningTreeMatrix[0].size(); ++j) {
            if (i == j)
                spanningTreeMatrix[i][j] = 1;
            else
                spanningTreeMatrix[i][j] = 0;
        }
    }
}

void generateMatrixOfSpanningTree(vector<pair<int, int>>& edgeSubset) {
    makeSpanningTreeMatrixReflexive();

    // Fill the matrix according to the rule of symmetry
    for (pair<int, int> verticesPair : edgeSubset) {
        spanningTreeMatrix[verticesPair.first][verticesPair.second] =
            edgeWeights[verticesPair.first][verticesPair.second];

        spanningTreeMatrix[verticesPair.second][verticesPair.first] =
            1.0 / spanningTreeMatrix[verticesPair.first][verticesPair.second];
    }

    // Fill missing elements with a transitive rule
    int numberOfMissingElements = V * (V - 1) - (edgeSubset.size() * 2);
    int maxEdges, numOfEdges, k;

    while (numberOfMissingElements > 0) {
        maxEdges = 0;

```

```

k = 0;

for (int i = 0; i < spanningTreeMatrix.size(); ++i) {
    numOfEdges = 0;

    for (int j = 0; j < spanningTreeMatrix[0].size(); ++j) {
        if (spanningTreeMatrix[i][j])
            numOfEdges++;
    }

    if (numOfEdges > maxEdges) {
        maxEdges = numOfEdges;
        k = i;
    }
}

for (int i = 0; i < spanningTreeMatrix.size(); ++i) {
    for (int j = 0; j < spanningTreeMatrix[0].size(); ++j) {
        if (i == k || k == j || i == j)
            continue;
        if (!spanningTreeMatrix[i][j] &&
            spanningTreeMatrix[i][k] &&
            spanningTreeMatrix[k][j])
        {
            spanningTreeMatrix[i][j] = spanningTreeMatrix[i][k] *
spanningTreeMatrix[k][j];
            numberOfMissingElements--;
        }
    }
}
}
}

```

Функція обчислення вектора пріоритетів за першим рядком матриці:

```

vector <double> calculatePriorityVectorForSpanningTree() {
    vector <double> priorityVector(V, 0);

    double sumOfRowelements = 0;

    for (int j = 0; j < spanningTreeMatrix[0].size(); ++j) {
        sumOfRowelements += spanningTreeMatrix[0][j];
    }

    for (int j = 0; j < spanningTreeMatrix[0].size(); ++j) {
        priorityVector[j] = spanningTreeMatrix[0][j] / sumOfRowelements;
    }

    return priorityVector;
}

```

Для знаходження результуючого вектора пріоритетів були розроблені функції знаходження середнього арифметичного:

```

void ArithmeticMean() {
    double sum;

    for (int j = 0; j < priorityVectors[0].size(); ++j) {
        sum = 0;
        for (int i = 0; i < priorityVectors.size(); ++i) {
            sum += priorityVectors[i][j];
        }
    }
}

```

```

        generalPriorityVector[j] = sum / spanningTrees.size();
    }
}

```

Та середнього геометричного:

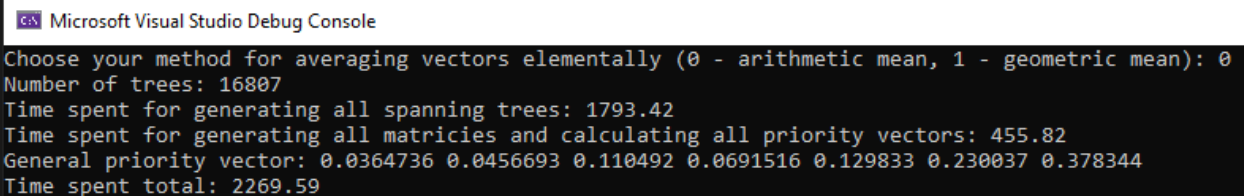
```

void GeometricMean() {
    double sum;

    for (int j = 0; j < priorityVectors[0].size(); ++j) {
        sum = 0;
        for (int i = 0; i < priorityVectors.size(); ++i) {
            sum += log(priorityVectors[i][j]);
        }

        generalPriorityVector[j] = exp(sum / priorityVectors.size());
    }
}

```



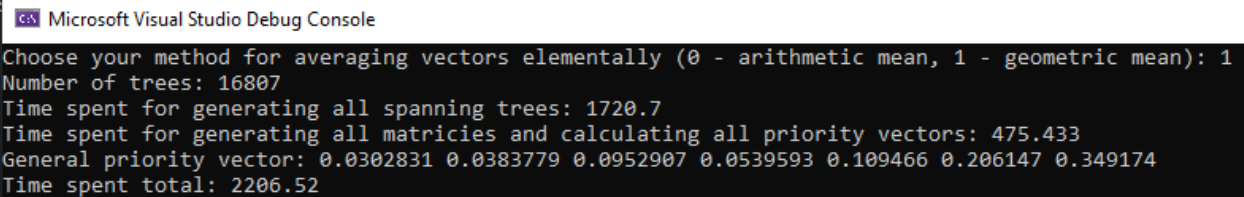
Microsoft Visual Studio Debug Console

```

Choose your method for averaging vectors elementally (0 - arithmetic mean, 1 - geometric mean): 0
Number of trees: 16807
Time spent for generating all spanning trees: 1793.42
Time spent for generating all matrices and calculating all priority vectors: 455.82
General priority vector: 0.0364736 0.0456693 0.110492 0.0691516 0.129833 0.230037 0.378344
Time spent total: 2269.59

```

Рисунок 1 – Результат роботи програми при середньому арифметичному



Microsoft Visual Studio Debug Console

```

Choose your method for averaging vectors elementally (0 - arithmetic mean, 1 - geometric mean): 1
Number of trees: 16807
Time spent for generating all spanning trees: 1720.7
Time spent for generating all matrices and calculating all priority vectors: 475.433
General priority vector: 0.0302831 0.0383779 0.0952907 0.0539593 0.109466 0.206147 0.349174
Time spent total: 2206.52

```

Рисунок 2 – Результат роботи програми при середньому геометричному

Завдання №2

У програмі доцільно розпаралелити два блоки:

1. Розрахунок матриці для кожного покривного дерева і обчислення вектора пріоритетів:

```

void BruteForceAlgorithm() {
    double startTime = omp_get_wtime();

    generateAllSpanningTrees();

    double endTime = omp_get_wtime();

    double elapsedTimeMilliseconds = (endTime - startTime) * 1000.0;

    cout << endl << "Time spent for generating all spanning trees: " <<
    elapsedTimeMilliseconds;

    startTime = omp_get_wtime();

#pragma omp parallel for
    for (int i = 0; i < spanningTrees.size(); ++i) {

```

```

        generateMatrixOfSpanningTree(spanningTrees[i]);
        priorityVectors.push_back(calculatePriorityVectorForSpanningTree());
    }

    endTime = omp_get_wtime();

    elapsedTimeMilliseconds = (endTime - startTime) * 1000.0;

    cout << endl << "Time spent for generating all matrices and calculating all
priority vectors: " << elapsedTimeMilliseconds;

    calculateGeneralPriorityVector();
}

```

2. Алгоритм заповнення матриць:

```

void generateMatrixOfSpanningTree(vector<pair<int, int>>& edgeSubset) {
    makeSpanningTreeMatrixReflexive();

    // Fill the matrix according to the rule of symmetry
    for (pair<int, int> verticesPair : edgeSubset)
    {
        spanningTreeMatrix[verticesPair.first][verticesPair.second] =
            edgeWeights[verticesPair.first][verticesPair.second];

        spanningTreeMatrix[verticesPair.second][verticesPair.first] =
            1.0 / spanningTreeMatrix[verticesPair.first][verticesPair.second];
    }

    //Fill missing elements with a transitive rule
    int numberOfMissingElements = V * (V - 1) - (edgeSubset.size() * 2);
    int maxEdges, numOfEdges, k;

    while (numberOfMissingElements > 0) {
        maxEdges = 0;
        k = 0;

#pragma omp parallel for reduction(max: maxEdges) shared(spanningTreeMatrix)
        for (int i = 0; i < V; ++i) {
            int numOfEdges = 0;
            for (int j = 0; j < V; ++j) {
                if (spanningTreeMatrix[i][j])
                    numOfEdges++;
            }
            if (numOfEdges > maxEdges) {
                maxEdges = numOfEdges;
                k = i;
            }
        }

#pragma omp parallel for shared(spanningTreeMatrix, numberOfMissingElements)
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                if (i == k || k == j || i == j)
                    continue;
                if (!spanningTreeMatrix[i][j] &&
                    spanningTreeMatrix[i][k] &&
                    spanningTreeMatrix[k][j])
                {
#pragma omp atomic
                    spanningTreeMatrix[i][j] = spanningTreeMatrix[i][k] *
spanningTreeMatrix[k][j];
#pragma omp atomic
                    numberOfMissingElements--;
                }
            }
        }
    }
}

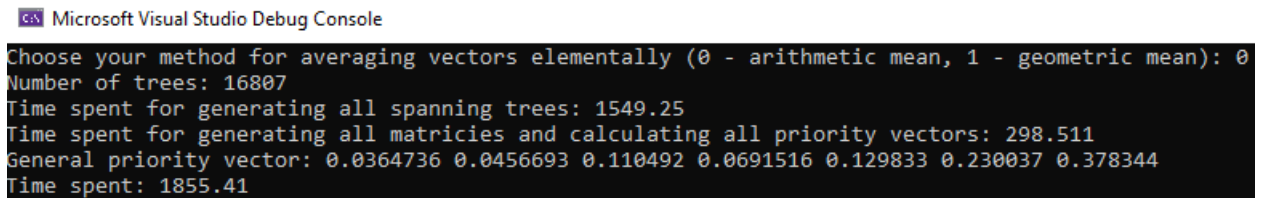
```

```

    }
  }
}

```

Алгоритм знаходження усіх покривних дерев виявилося недоцільно розпаралелювати при даній реалізації, оскільки збільшується час роботи програми.

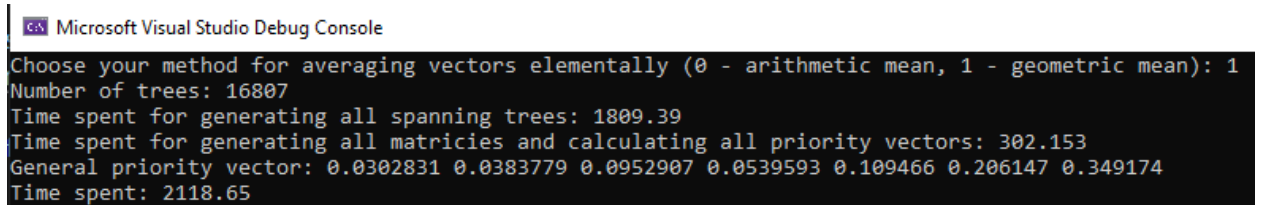


```

Microsoft Visual Studio Debug Console
Choose your method for averaging vectors elementally (0 - arithmetic mean, 1 - geometric mean): 0
Number of trees: 16807
Time spent for generating all spanning trees: 1549.25
Time spent for generating all matrices and calculating all priority vectors: 298.511
General priority vector: 0.0364736 0.0456693 0.110492 0.0691516 0.129833 0.230037 0.378344
Time spent: 1855.41

```

Рисунок 3 - Результат роботи програми при середньому арифметичному



```

Microsoft Visual Studio Debug Console
Choose your method for averaging vectors elementally (0 - arithmetic mean, 1 - geometric mean): 1
Number of trees: 16807
Time spent for generating all spanning trees: 1809.39
Time spent for generating all matrices and calculating all priority vectors: 302.153
General priority vector: 0.0302831 0.0383779 0.0952907 0.0539593 0.109466 0.206147 0.349174
Time spent: 2118.65

```

Рисунок 4 - Результат роботи програми при середньому геометричному

Висновок

На цій лабораторній роботі ми освоїли алгоритмізацію паралельних обчислень та їх реалізацію за допомогою засобів OpenMP на практичному прикладі комбінаторного методу визначення вектора на основі неповної матриці експертних парних порівнянь. Проведене дослідження продемонструвало, що при послідовній реалізації програма працює швидше при знаходженні результуючого вектора за допомогою середнього геометричного, тоді як при паралельній реалізації – за допомогою середнього арифметичного. Також було зроблено висновок, що розпаралелена програма працює ефективніше, навіть за рахунок паралелізації 2 блоків програми.