

Міністерство освіти і науки України
НТУУ «КПІ ім. Ігоря Сікорського»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Лабораторна робота №6
з дисципліни «Технології паралельних обчислень в
енергетичних комплексах»
Тема «Графові алгоритми»
Варіант №19

Студента 3-го курсу НН ІАТЕ гр. ТР-12

Ковальова Олександра

Перевірив: ас., Софієнко А. Ю.

Мета роботи. Розробити паралельні реалізації алгоритмів оброблення графів з допомогою технології MPI.

Завдання:

1. Напишіть програму, що реалізує паралельний алгоритм Прима
2. Напишіть програму, що реалізує паралельний алгоритм Флойда-Уоршелла.

Хід роботи

Алгоритм Прима:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

typedef struct DWeight {
    int weight;
    int node;
} DWeight;

typedef struct DEdge {
    int fromNode;
    int toNode;
    int weight;
} DEdge;

void fscanfEdgeList(FILE* file, int **adMatrix, int *nodesNmb) {
    int edgesNmb, node1, node2, weight, matrixSize, nodes;

    fscanf(file, "nodes-%d,edges-%d,weights", &nodes, &edgesNmb);
    *nodesNmb = nodes;

    matrixSize = nodes * nodes;
    *adMatrix = (int*) malloc(matrixSize * sizeof(int));

    for (int i = 0; i < matrixSize; (*adMatrix)[i] = 0, i++);

    for (int i = 0; i < edgesNmb; i++) {
        fscanf(file, "%d,%d,%d", &node1, &node2, &weight);
        (*adMatrix)[node1 * nodes + node2] = weight;
        (*adMatrix)[node2 * nodes + node1] = weight;
    }
}

void fprintfAdMatrix(FILE* file, int* adMatrix, int rowNmb, int colNmb) {
    int index = 0;
    for (int row = 0; row < rowNmb; row++) {
        for (int column = 0; column < colNmb; column++, index++) {
            fprintf(file, "%d ", adMatrix[index]);
        }
        fprintf(file, "\n");
    }
}

void fprintfDTable(FILE* file, DWeight* dTable, int rowNmb, int processId) {
    for (int row = 0; row < rowNmb; row++) {
        fprintf(file, "%d: %d| %d,%d\n", processId, row, dTable[row].node, dTable[row].weight);
    }
}

void fprintfDEdges(FILE* file, DEdge* edes, int nodeNmb, int rowNmb, double time_taken) {
    fprintf(file, "nodes-%d,edges-%d,weights\n", nodeNmb, rowNmb);
    for (int row = 0; row < rowNmb; row++) {
        fprintf(file, "%d,%d,%d\n", edes[row].fromNode, edes[row].toNode, edes[row].weight);
    }
    fprintf(file, "Done in: %.6f secs\n", time_taken);
}

void primPartitionMatrix(int *adMatrixFull, int nodesNmb, int processId, int processNmb, int
**adMatrixPartial, int *nodesProcesNmb, int *startNode) {
```

```

int matrixSize = nodesNmb * nodesNmb;
int lastId = processNmb - 1;

int middleNodes = (int) ceil((float) nodesNmb / (float) processNmb);
int middleSize = nodesNmb * middleNodes;

int lastNodes = nodesNmb - (middleNodes * lastId);
int lastSize = matrixSize - lastId * middleSize;
int lastCommProcessNmb, lastCommProcessId;

MPI_Comm lastComm;

*startNode = middleNodes * processId;

if (processNmb > 1) {
    MPI_Comm_split(MPI_COMM_WORLD, processId / lastId, processId, &lastComm);
    MPI_Comm_size(lastComm, &lastCommProcessNmb);
    MPI_Comm_rank(lastComm, &lastCommProcessId);

    if (processId != lastId) {
        *adMatrixPartial = (int*) malloc(middleSize * sizeof(int));
        *nodesProcesNmb = middleNodes;

        MPI_Scatter(adMatrixFull, middleSize, MPI_INT,
                    *adMatrixPartial, middleSize, MPI_INT,
                    0, lastComm);
    } else {
        *adMatrixPartial = (int*) malloc(lastSize * sizeof(int));
        *nodesProcesNmb = lastNodes;
    }

    if (processId == 0) {
        MPI_Send(adMatrixFull + (lastId * middleSize), lastSize, MPI_INT, lastId, 0, MPI_COMM_WORLD);
    }
    else if (processId == lastId) {
        MPI_Recv(*adMatrixPartial, lastSize, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
} else {
    *adMatrixPartial = (int*) malloc(matrixSize * sizeof(int));
    *nodesProcesNmb = nodesNmb;
    memcpy(*adMatrixPartial, adMatrixFull, matrixSize * sizeof(int));
}
}

void primPartitionDArray(int nodesNmbProcess, int nodesNmb, int firstNode, int* adMatrixPartial,
DWeight** dTable) {
    int weight;
    *dTable = (DWeight*) malloc(nodesNmbProcess * sizeof(DWeight));

    for (int row = 0; row < nodesNmbProcess; row++) {
        weight = adMatrixPartial[row * nodesNmb + firstNode];
        (*dTable)[row].weight = weight > 0 ? weight : INT_MAX;
        (*dTable)[row].node = firstNode;
    }
}

void primFindMinimum(int startNode, int nodesNmbProcess, DWeight* dTable, int* isAdded, DWeight
*local) {
    int globalNode = 0;
    int localWeight = 0;
    local->weight = INT_MAX;
    local->node = -1;

    for (int localNode = 0; localNode < nodesNmbProcess; localNode++) {
        globalNode = localNode + startNode;

        if (!isAdded[globalNode]) {
            localWeight = dTable[localNode].weight;

            if (localWeight != 0 && (local->node == -1 || local->weight > localWeight)) {
                local->weight = localWeight;
                local->node = globalNode;
            }
        }
    }
}

void primBroadcastSolution(int startNode, int nodesNmbProcess, DWeight *dTable, DWeight globalMin,
DEdge* edge) {

```

```

// Function broadcasts solution to every process.
int fromNode = -1, fromNodeGlobal;
MPI_Bcast(&globalMin, 1, MPI_2INTEGER, 0, MPI_COMM_WORLD);

// Finding responisble partition
if (startNode <= globalMin.node && globalMin.node < startNode + nodesNmbProcess) {
    // Only one process have fromNode value greater then -1
    fromNode = dTable[globalMin.node - startNode].node;
}

// FromNode value cannot be broadcasted along toNode,value tuple so separate
// Reduce and broadcast is needed
MPI_Reduce(&fromNode, &fromNodeGlobal, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Bcast(&fromNodeGlobal, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Saving solution in every process
edge->weight = globalMin.weight;
edge->toNode = globalMin.node;
edge->fromNode = fromNodeGlobal;
}

void primUpdateDArray(int *adMatrixPartial, int nodesNmb, int nodesNmbProcess, DEdge *edge, int
*isAdded, DWeight *dTable) {
    // Function updates D array after adding new node to tree

    int newWeight = 0;
    isAdded[edge->toNode] = 1;

    for (int row = 0; row < nodesNmbProcess; row++) {
        // Weight from new node
        newWeight = adMatrixPartial[row * nodesNmb + edge->toNode];

        // Check if new path to node is better then previous one
        if (dTable[row].weight > newWeight && newWeight > 0) {
            dTable[row].node = edge->toNode;
            dTable[row].weight = newWeight;
        }
    }
}

void primAlgorithm(int *adMatrix, int nodesNmb, int processId, int processNmb, DEdge* edges) {
    // Prim's Algorithm

    int *adMatrixPartial = 0; // chunk of adMatrix
    int *isNodeAdded = 0; // stores 1 if node is already in tree
    int nodesNmbProcess = 0; // nodes per process
    int firstNode = 0; // algorithm start node
    int edgesNmb = nodesNmb - 1; // edge count
    int startNode = 0; // start node number for partition

    DWeight *dTable = 0; // weights array
    DWeight localMin;
    DWeight globalMin;

    // Partitioning of adjacency matrix and distance array
    primPartitionMatrix(adMatrix, nodesNmb, processId, processNmb, &adMatrixPartial, &nodesNmbProcess,
&startNode);
    primPartitionDArray(nodesNmbProcess, nodesNmb, firstNode, adMatrixPartial, &dTable);

    // Algorithm initialization
    isNodeAdded = (int*) malloc(nodesNmb * sizeof(int));
    for (int i = 0; i < nodesNmb; isNodeAdded[i] = 0,i++);
    isNodeAdded[firstNode] = 1;

    // Main iteration
    for (int index = 0; index < edgesNmb; index++) {

        DEdge *edge = &edges[index];

        // 1. Each process P_i computes d_i = min{dTable}
        primFindMinimum(startNode, nodesNmbProcess, dTable, isNodeAdded, &localMin);

        // 2. Global minimum d is then obtained by using all-to-one reduction operation
        // and its stored in P_0 process. P_0 stores new vortex u
        MPI_Reduce(&localMin, &globalMin, 1, MPI_2INTEGER, MPI_MINLOC, 0, MPI_COMM_WORLD);

        // 3. Process P_0 broadcasts u one-to-all. The process responsible for u
        // marks u as belonging to tree.
        primBroadcastSolution(startNode, nodesNmbProcess, dTable, globalMin, edge);

        // 4. Each process updates the values od d[v] for its local vertices
    }
}

```

```

    primUpdatedArray(adMatrixPartial, nodesNmb, nodesNmbProcess, edge, isNodeAdded, dTable);
}

free(isNodeAdded);
free(adMatrixPartial);
free(dTable);
}

int main( int argc, char *argv[] )
{
    FILE *file;
    char *filename = argv[1];
    char *output_filename = argv[2];
    double time_taken;
    int processId, processNmb;
    int nodesNmb = 0;
    int *adMatrix = 0;
    DEdge *edges;

    // MPI Initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &processNmb);
    MPI_Comm_rank(MPI_COMM_WORLD, &processId);

    // Read edge list from file
    if (processId == 0) {
        file = fopen(filename, "r");
        if (file == 0) {
            printf("Cannot found input file %s!\n", filename);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        fscanfEdgeList(file, &adMatrix, &nodesNmb);
        fclose(file);
    }

    // Initialization
    MPI_Bcast(&nodesNmb, 1, MPI_INT, 0, MPI_COMM_WORLD);
    edges = (DEdge*) malloc((nodesNmb - 1) * sizeof(DEdge));

    // Starts Timer
    time_taken -= MPI_Wtime();

    // Prim's Algorithm
    primAlgorithm(adMatrix, nodesNmb, processId, processNmb, edges);

    // Stop the timer
    time_taken += MPI_Wtime();

    // Free allocated resources
    if(processId == 0) {
        if (argc == 2) {
            fprintfDEdges(stdout, edges, nodesNmb, nodesNmb - 1, time_taken);
        } else if (argc == 3) {
            FILE *file_ptr = fopen(output_filename, "w");

            if (file_ptr != NULL) {
                fprintfDEdges(file_ptr, edges, nodesNmb, nodesNmb - 1, time_taken);
                fclose(file_ptr);
            } else {
                printf("Error opening file %s\n", output_filename);
                free(adMatrix);
                free(edges);
                MPI_Finalize();
                return 0;
            }
        }

        free(adMatrix);
    }
    free(edges);

    MPI_Finalize();
    return 0;
}

```

Результати.

Послідовна версія:

```
----- NON-PARALLEL -----
Performance counter stats for 'mpirun --allow-run-as-root -np 1 ./out/exec.out ./graphs/input.csv ./results//non-parallel.txt':

    1,700.49 msec task-clock           #    0.873 CPUs utilized
         366      context-switches    #   215.232 /sec
           8      cpu-migrations      #    4.705 /sec
    201,354      page-faults         #   118.409 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    1.948251863 seconds time elapsed

    1.479318000 seconds user
    0.227788000 seconds sys

To check if the results are the same, the last 2 lines:
6110,9007,40
6689,7227,98
Done in: 1.452123 secs
```

Паралельна версія:

```
----- PARALLEL -----
Performance counter stats for 'mpirun --allow-run-as-root -np 4 ./out/exec.out ./graphs/input.csv ./results//parallel.txt':

    4,739.98 msec task-clock           #    3.212 CPUs utilized
         1,487      context-switches    #   313.714 /sec
           25      cpu-migrations      #    5.274 /sec
    206,416      page-faults         #    43.548 K/sec
<not supported>      cycles
<not supported>      instructions
<not supported>      branches
<not supported>      branch-misses

    1.475744832 seconds time elapsed

    4.388335000 seconds user
    0.357832000 seconds sys

To check if the results are the same, the last 2 lines:
6110,9007,40
6689,7227,98
Done in: 0.926544 secs
```

Алгоритм Флойда-Уоршелла:

```
#include <assert.h>
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define ROOT    0
#define MPI_TAG 1
#define TRUE    1
#define FALSE   0
#define INF     INT_MAX/2

#define MIN(A, B) (A < B) ? A : B

typedef struct {
    int rank;
    int row, col;
    int p, q;
    MPI_Comm comm;
    MPI_Comm row_comm;
    MPI_Comm col_comm;
} GRID_INFO;
```

```

void setup_grid(GRID_INFO *grid);
int check_fox(int p, int n);
int *read_mtrx(int n);
void send_sub_mtrx(int *mtrx, int n, int q);
void *process_mtrx(GRID_INFO *grid, double *time, int *mtrx_A, int n);
void floyd_warshall(int *_A, int *_B, int *_C, int n);
void fix_final_mtrx(int *mtrx, int *mtrx_F, int n, int q);
void print_mtrx(int *mtrx, int n);
void print_mtrx_to_file(FILE* file, int *mtrx, int n);

int main(int argc, char **argv) {
    char *output_filename = argv[1];

    MPI_Init(&argc, &argv);
    GRID_INFO grid;
    setup_grid(&grid);

    int n, *mtrx;
    if (grid.rank == ROOT) {
        if (!scanf("%d", &n)) {
            fprintf(stderr, "Error while reading input.\nAborting...\n");
            MPI_Abort(MPI_COMM_WORLD, 0);
            exit(1);
        }
        if (!check_fox(grid.p, n)) {
            fprintf(stderr, "Fox algorithm can't be applied with a matrix of size %d and %d
processes.\nAborting...\n", n, grid.p);
            MPI_Abort(MPI_COMM_WORLD, 0);
            exit(1);
        }
        mtrx = read_mtrx(n);
    }

    MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
    if (grid.rank == ROOT && grid.p > 1) send_sub_mtrx(mtrx, n, grid.q);

    const int m = n / grid.q;
    int *mtrx_A;
    if (grid.p > 1) {
        mtrx_A = (int *) malloc(m * m * sizeof(int));
        assert(mtrx_A != NULL);
        MPI_Recv(mtrx_A, m * m, MPI_INT, ROOT, MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
        mtrx_A = mtrx;
    }

    double time;
    int *mtrx_C = process_mtrx(&grid, &time, mtrx_A, n);
    if (grid.p > 1) free(mtrx_A);

    int *mtrx_F = malloc(n * n * sizeof(int));
    assert(mtrx_F != NULL);
    MPI_Gather(mtrx_C, m * m, MPI_INT, mtrx_F, m * m, MPI_INT, ROOT, MPI_COMM_WORLD);
    free(mtrx_C);

    if (grid.rank == ROOT) {
        fix_final_mtrx(mtrx, mtrx_F, n, grid.q);
        FILE *file_ptr = fopen(output_filename, "w");
        if (file_ptr != NULL) {
            print_mtrx_to_file(file_ptr, mtrx, n);
            fprintf(file_ptr, "\nExecution Time: %10.3lf milliseconds.\n", time * 1000);
        } else {
            printf("Error opening file %s\n", output_filename);
            free(mtrx_F);
            MPI_Comm_free(&grid.comm);
            MPI_Comm_free(&grid.row_comm);
            MPI_Comm_free(&grid.col_comm);
            MPI_Finalize();
            return 0;
        }
    }

    free(mtrx_F);
    MPI_Comm_free(&grid.comm);
    MPI_Comm_free(&grid.row_comm);
    MPI_Comm_free(&grid.col_comm);
    MPI_Finalize();
    return 0;
}

void setup_grid(GRID_INFO *grid) {

```

```

MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
MPI_Comm_rank(MPI_COMM_WORLD, &(grid->rank));

grid->q = sqrt(grid->p);

int dims[2] = { grid->q, grid->q };
int periods[2] = { 1, 1 };
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &(grid->comm));

int coords[2];
MPI_Comm_rank(grid->comm, &(grid->rank));
MPI_Cart_coords(grid->comm, grid->rank, 2, coords);
grid->row = coords[0];
grid->col = coords[1];

coords[0] = 0; coords[1] = 1;
MPI_Cart_sub(grid->comm, coords, &(grid->row_comm));
coords[0] = 1; coords[1] = 0;
MPI_Cart_sub(grid->comm, coords, &(grid->col_comm));
}

int check_fox(int p, int n) {
    int q = sqrt(p);
    if (q * q == p && n % q == 0) return TRUE;
    return FALSE;
}

int *read_mtrx(int n) {
    int (*mtrx)[n] = (int (*)[n]) malloc(n * n * sizeof(int));
    assert(mtrx != NULL);

    int i, j;
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if (!scanf("%d", &mtrx[i][j])) {
                fprintf(stderr, "Error while reading input.\nAborting...\n");
                MPI_Abort(MPI_COMM_WORLD, 0);
                exit(1);
            }
            if (mtrx[i][j] == 0 && i != j) mtrx[i][j] = INF;
        }
    }
    return (int *) mtrx;
}

void send_sub_mtrx(int *_mtrx, int n, int q) {
    int m = n / q;
    int (*mtrx)[n] = (int (*)[n]) _mtrx;
    int (*sub_mtrx)[m] = (int (*)[m]) malloc(m * m * sizeof(int));
    assert(sub_mtrx != NULL);

    int dst = 0;
    int i, j, k, l;
    for (k = 0; k < q; k++) {
        for (l = 0; l < q; l++) {
            for (i = 0; i < m; i++) {
                for (j = 0; j < m; j++) {
                    sub_mtrx[i][j] = mtrx[i + (k * m)][j + (l * m)];
                }
            }
            MPI_Send(sub_mtrx, m * m, MPI_INT, dst++, MPI_TAG, MPI_COMM_WORLD);
        }
    }
    free(sub_mtrx);
}

void *process_mtrx(GRID_INFO *grid, double *time, int *mtrx_A, int n) {
    int m = n / grid->q;
    int src = (grid->row + 1) % grid->q;
    int dst = (grid->row - 1 + grid->q) % grid->q;

    int *temp_A = (int *) malloc(m * m * sizeof(int));
    assert(temp_A != NULL);
    int *mtrx_B = (int *) malloc(m * m * sizeof(int));
    assert(mtrx_B != NULL);
    int *mtrx_C = (int *) malloc(m * m * sizeof(int));
    assert(mtrx_C != NULL);
    memcpy(mtrx_C, mtrx_A, m * m * sizeof(int));

    int iter;
    *time = MPI_Wtime();

```



```

    for (iter = 1; iter < n; iter <= 1) {
        memcpy(mtrx_B, mtrx_C, m * m * sizeof(int));
        int stage;
        for (stage = 0; stage < grid->q; stage++) {
            int bcast_root = (grid->row + stage) % grid->q;
            if (bcast_root == grid->col) {
                MPI_Bcast(mtrx_A, m * m, MPI_INT, bcast_root, grid->row_comm);
                floyd_warshall(mtrx_A, mtrx_B, mtrx_C, m);
            } else {
                MPI_Bcast(temp_A, m * m, MPI_INT, bcast_root, grid->row_comm);
                floyd_warshall(temp_A, mtrx_B, mtrx_C, m);
            }
            MPI_Sendrecv_replace(mtrx_B, m * m, MPI_INT, dst, MPI_TAG, src, MPI_TAG, grid->col_comm,
MPI_STATUS_IGNORE);
        }
        *time = MPI_Wtime() - *time;
        free(temp_A);
        free(mtrx_B);
        return mtrx_C;
    }

inline void floyd_warshall(int *_A, int *_B, int *_C, int n) {
    int (*A)[n] = (int (*)[n]) _A;
    int (*B)[n] = (int (*)[n]) _B;
    int (*C)[n] = (int (*)[n]) _C;

    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                C[i][j] = MIN(C[i][j], A[i][k] + B[k][j]);
}

void fix_final_mtrx(int *_mtrx, int *_mtrx_F, int n, int q) {
    int m = n / q;
    int (*mtrx)[n] = (int (*)[n]) _mtrx;
    int (*mtrx_F)[n] = (int (*)[n]) _mtrx_F;

    int count = 0;
    int i, j, k, l;
    int a = 0, b = 0;
    for (k = 0; k < q; k++) {
        for (l = 0; l < q; l++) {
            for (i = k * m; i < (k + 1) * m; i++) {
                for (j = l * m; j < (l + 1) * m; j++) {
                    mtrx[i][j] = mtrx_F[a][b] == INF ? 0 : mtrx_F[a][b];
                    b = ++b == n ? (++a && 0) : b;
                }
            }
        }
        count++;
    }
}

void print_mtrx(int *_mtrx, int n) {
    int (*mtrx)[n] = (int (*)[n]) _mtrx;

    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n - 1; j++) {
            printf("%d ", mtrx[i][j]);
        }
        printf("%d\n", mtrx[i][j]);
    }
    fflush(stdout);
}

void print_mtrx_to_file(FILE* file, int *_mtrx, int n) {
    int (*mtrx)[n] = (int (*)[n]) _mtrx;

    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n - 1; j++) {
            fprintf(file, "%d ", mtrx[i][j]);
        }
        fprintf(file, "%d\n", mtrx[i][j]);
    }
}

```

Результати.

```
alex@host ~/PCT/Lab6/Floyd-Warshall
$ ./start.sh

  L A B 6

Using file ./graphs/input30x30.txt...

----- NON-PARALLEL -----
To check if the results are the same, the last 100 bytes:
5 11 10 11 10 9 7 11 12 7 3 13 8 5 6 9 11 6 13 12 9 10 10
Execution Time:      0.721 milliseconds.

----- PARALLEL -----
To check if the results are the same, the last 100 bytes:
5 11 10 11 10 9 7 11 12 7 3 13 8 5 6 9 11 6 13 12 9 10 10
Execution Time:      0.249 milliseconds.
```

Контрольні запитання:

- 1) Наведіть визначення графа. Які основні способи використовуються для задання графів?

Граф – це абстрактна математична структура, що складається з вершин (вузлів) та ребер (зв'язків), що з'єднують ці вершини.

Основні способи задання графів це матриця суміжності та список суміжності.

- 2) Який сенс має задача пошуку мінімального кістякового дерева? Наведіть приклад використання задачі на практиці.

Задача пошуку мінімального кістякового дерева полягає в знаходженні підмножини ребер графа, що з'єднує всі його вершини, при цьому має мінімальну загальну вагу. Це важлива задача для оптимізації мережевих структур, наприклад, в телекомунікаціях для побудови оптимальних мереж передачі даних або в електроенергетиці для планування електромережі. Наприклад, у сфері телекомунікацій, пошук мінімального кістякового дерева може використовуватися для побудови оптимальних мереж передачі даних, де ребра представляють канали передачі, а вершини – вузли мережі. Оптимальне кістякове дерево допомагає забезпечити ефективний та економічний обмін даними між вузлами мережі.

- 3) Наведіть загальну схему алгоритму Прима. Якою є трудомісткість цього алгоритму?

Алгоритм Прима починає з довільної вершини і поступово додає до кістякового дерева ребра з найменшою вагою, які з'єднують поточне дерево з

вершинами, що ще не включені. Трудомісткість алгоритму Прима зазвичай є $O(E \log V)$, де E – кількість ребер, V – кількість вершин у графі.

4) В який спосіб можна розпаралелити алгоритм Прима?

Алгоритм Прима можна розпаралелити, розподіливши множину вершин між різними обчислювальними вузлами. Кожен вузол може обробляти свою підмножину вершин, обчислюючи найменші ребра, які з'єднують ці вершини з рештою графу. Після цього вузли можуть обмінюватись даними, щоб оновити кістякове дерево з урахуванням відомих ребер. Такий підхід дозволяє розпаралелити обчислення та зменшити час виконання алгоритму на паралельних обчислювальних системах.

5) У чому полягає задача пошуку всіх найкоротших шляхів?

Задача пошуку всіх найкоротших шляхів полягає у визначенні найкоротших відстаней між кожною парою вершин у графі.

6) Наведіть загальну схему алгоритму Флойда-Уоршелла. Якою є трудомісткість цього алгоритму?

а Алгоритм Флойда-Уоршелла використовує динамічне програмування для знаходження найкоротших шляхів між усіма парами вершин у напрямленому або ненаправленому зваженому графі.

Трудомісткість алгоритму Флойда-Уоршелла – $O(V^3)$, де V – кількість вершин у графі.

7) В який спосіб можна розпаралелити алгоритм Флойда-Уоршелла?

Алгоритм Флойда-Уоршелла можна розпаралелити, розподіливши обчислення між різними обчислювальними вузлами. Кожен вузол може обчислювати частину матриці найкоротших шляхів, а потім обмінюватись результатами з іншими вузлами для оновлення частини матриці. Такий підхід дозволяє розпаралелити обчислення та зменшити час виконання алгоритму на паралельних обчислювальних системах.