Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Кафедра цифрових технологій в енергетиці

ЗВІТ   
з виконання лабораторної роботи №6  
з дисципліни «Технології паралельних обчислень в енергетичних комплексах»

«Графові алгоритми»

Варіант 21

Виконав: студент групи ТР-12 Руденко В.І

Завдання:

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

Хід Роботи:

**Код програми Завдання 1 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);*

*printf("time spend: %.6f\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) {*

*int fromNode = -1, fromNodeGlobal;*

*MPI\_Bcast(&globalMin, 1, MPI\_2INTEGER, 0, MPI\_COMM\_WORLD);*

*if (startNode <= globalMin.node && globalMin.node < startNode + nodesNmbProcess) {*

*fromNode = dTable[globalMin.node - startNode].node;*

*}*

*MPI\_Reduce(&fromNode, &fromNodeGlobal, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);*

*MPI\_Bcast(&fromNodeGlobal, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);*

*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) {*

*int newWeight = 0;*

*isAdded[edge->toNode] = 1;*

*for (int row = 0; row < nodesNmbProcess; row++) {*

*newWeight = adMatrixPartial[row \* nodesNmb + edge->toNode];*

*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;*

*}*

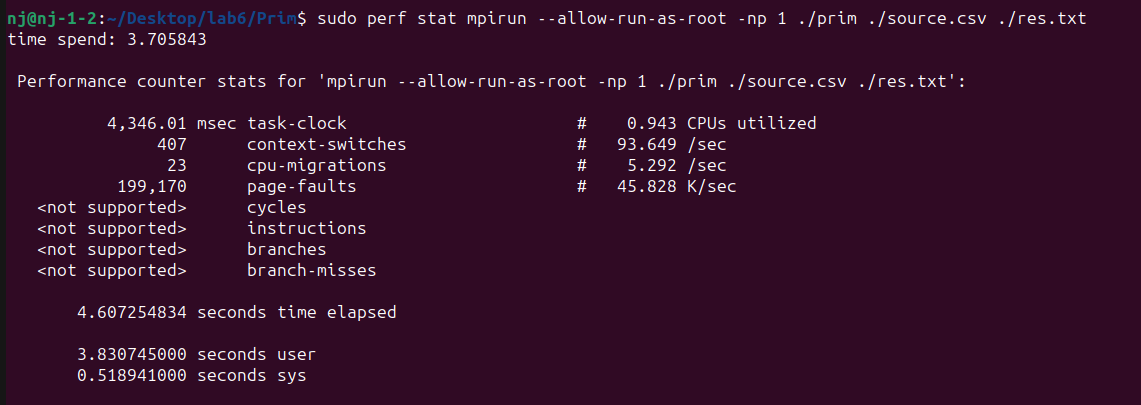


Table 2 Виконання завдання в однопоточному режимі

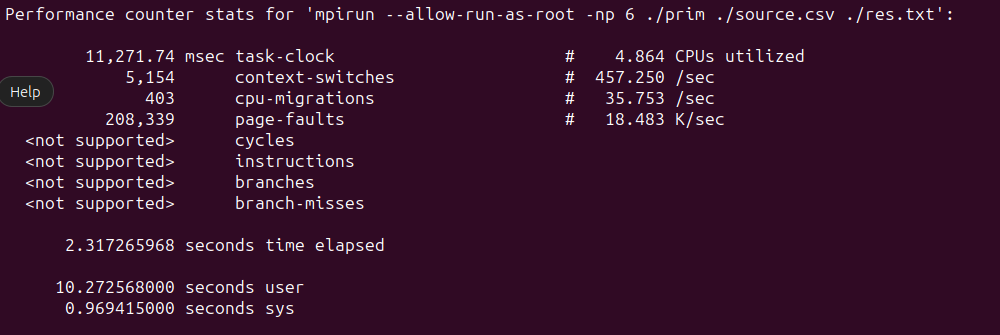


Table 4 Виконання Завдання а багатопоточному режимі

**Код програми Завдання 2 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);*

*printf("time spend: %.6f\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;*

*}*

*}*

*}*

*}*

*}*

*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]);*

*}*

*}*

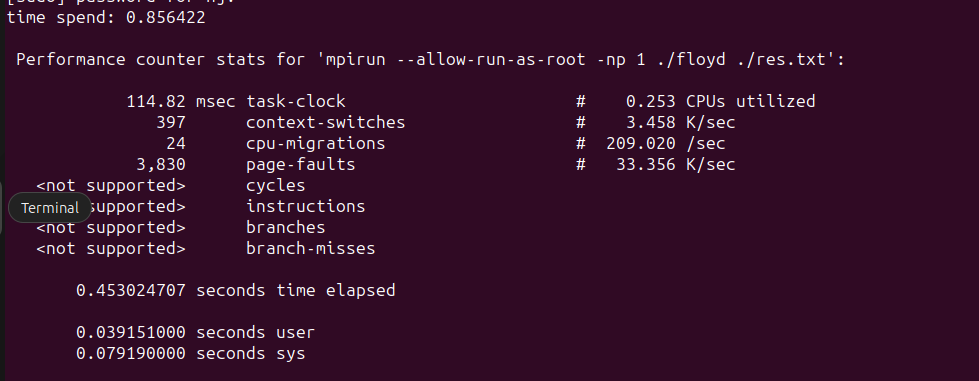


Table 2 Виконання завдання в однопоточному режимі

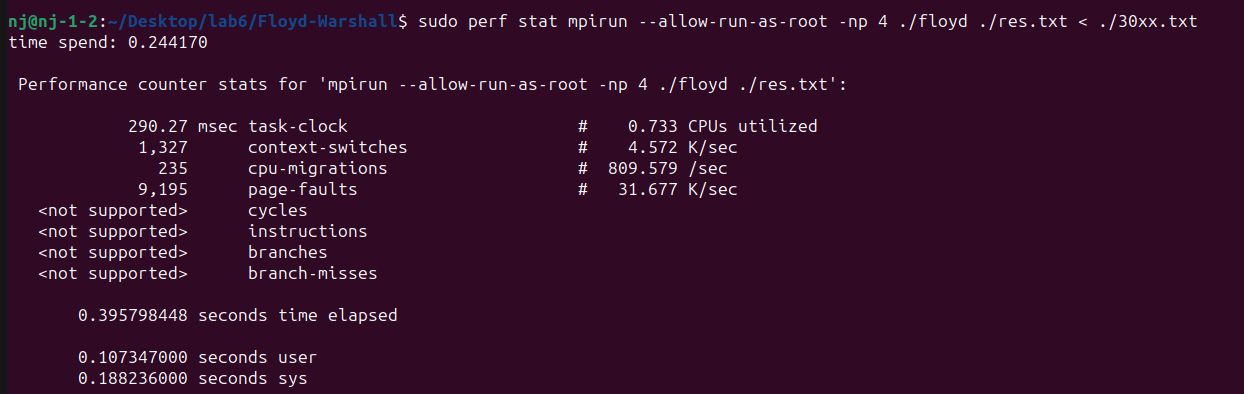


Table 4 Виконання Завдання а багатопоточному режимі

**Контрольні запитання:**

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.

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.

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 де E – кількість ребер, V – кількість вершин у графі.

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.

У чому полягає задача пошуку всіх найкоротших шляхів?

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

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

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

Трудомісткість алгоритму Флойда-Уоршелла – , де V – кількість вершин у графі.

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 обмінюватися результатами з іншими вузлами для оновлення частини матриці. Такий підхід дозволяє розпаралелити обчислення та зменшити час виконання алгоритму на паралельних обчислювальних системах.