**Міністерство освіти і науки України**

**Національний університет "Львівська політехніка"**

**Кафедра ЕОМ**

****

**Звіт з лабораторної роботи №1**

**з дисципліни “Паралельні та розподілені обчислення ”**

**Варіант 1**

**Виконав: студент .гр. КІ-34**

**Юхименко Д.Р**

**Прийняв:**

**Козак Н.Б.**

**Львів 2020 р.**

**Мета:** Ознайомитись з методами організації паралельного множення матриці на вектор та розробити паралельну програму з використанням технології MPI.

**Завдання:**

В результаті перемноження матриці *А* розмірності *m х n* і вектора *b*, що складається з *n* елементів, отримується вектор розміру *m*, кожен *i*-й елемент якого є результат скалярного множення i-того рядка матриці *А* (позначимо цей рядок *aі*) і вектора *b*:

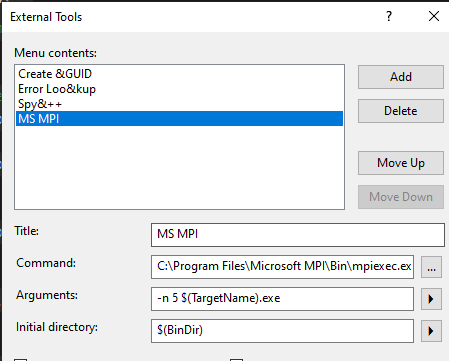
Тим самим отримання результуючого вектора *С* припускає повторення *m* однотипних операцій по множенню рядків матриці *A* і вектора *b*. Кожна така операція включає множення елементів рядка матриці і вектора *b* (n операцій) і подальше підсумовування отриманих множень (n - 1 операцій). Загальна кількість необхідних скалярних операцій є величина

*T1 = m·(2n-1).*

**Індивідуальне завдання:**



**Параметри запуску:**



**Код програми:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include <mpi.h>

#include <algorithm>

#include <iostream>

#include <fstream>

#define Dummy

using namespace std;

int Rows = 650;

int Columns = 80;

int ProcNum = 0; // Number of available processes

int ProcRank = 0; // Rank of current process

int GridSize; // Size of virtual processor grid

int GridCoords[2]; // Coordinates of current processor in grid

MPI\_Comm GridComm; // Grid communicator

MPI\_Comm ColComm; // Column communicator

MPI\_Comm RowComm; // Row communicator

/// Function for simple initialization of matrix elements

void DummyDataInitialization(double\* pAMatrix, double\* pBMatrix, int Size)

{

    int i, j; // Loop variables

    for (i = 0; i < Size; i++)

        for (j = 0; j < Size; j++) {

            pAMatrix[i \* Size + j] = 0;

            pBMatrix[i \* Size + j] = 0;

        }

    for (i = 0; i < Rows; i++)

        for (j = 0; j < Columns; j++) {

            pAMatrix[i \* Rows + j] = 1;

        }

    for (i = 0; i < Rows; i++)

        for (j = 0; j < 1; j++) {

            pBMatrix[i \* Rows + j] = 1;

        }

}

// Function for random initialization of matrix elements

void RandomDataInitialization(double\* pAMatrix, double\* pBMatrix,

    int Size) {

    int i, j; // Loop variables

    srand(unsigned(clock()));

    for (i = 0; i < Size; i++)

        for (j = 0; j < Size; j++) {

            pAMatrix[i \* Size + j] = 0;

            pBMatrix[i \* Size + j] = 0;

        }

    for (i = 0; i < Rows; i++)

        for (j = 0; j < Columns; j++) {

            pAMatrix[i \* Rows + j] = rand() / double(1000);

            pBMatrix[i \* Rows + j] = rand() / double(1000);

        }

}

// Function for formatted matrix output

void PrintMatrix(double\* pMatrix, int RowCount, int ColCount) {

    int i, j; // Loop variables

    for (i = 0; i < RowCount; i++) {

        for (j = 0; j < ColCount; j++)

            printf("%7.4f ", pMatrix[i \* ColCount + j]);

        printf("\n");

    }

}

// Function for matrix multiplication

void SerialResultCalculation(double\* pAMatrix, double\* pBMatrix,

    double\* pCMatrix, int Size) {

    int i, j, k; // Loop variables

    for (i = 0; i < Size; i++) {

        for (j = 0; j < Size; j++)

            for (k = 0; k < Size; k++)

                pCMatrix[i \* Size + j] += pAMatrix[i \* Size + k] \* pBMatrix[k \* Size + j];

    }

}

// Function for block multiplication

void BlockMultiplication(double\* pAblock, double\* pBblock,

    double\* pCblock, int Size) {

    SerialResultCalculation(pAblock, pBblock, pCblock, Size);

}

// Function for creating the two-dimensional grid communicator

// and communicators for each row and each column of the grid

void CreateGridCommunicators() {

    int DimSize[2]; // Number of processes in each dimension of the grid

    int Periodic[2]; // =1, if the grid dimension should be periodic

    int Subdims[2]; // =1, if the grid dimension should be fixed

    DimSize[0] = GridSize;

    DimSize[1] = GridSize;

    Periodic[0] = 0;

    Periodic[1] = 0;

    // Creation of the Cartesian communicator

    MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, DimSize, Periodic, 1, &GridComm);

    // Determination of the cartesian coordinates for every process

    MPI\_Cart\_coords(GridComm, ProcRank, 2, GridCoords);

    // Creating communicators for rows

    Subdims[0] = 0; // Dimensionality fixing

    Subdims[1] = 1; // The presence of the given dimension in the subgrid

    MPI\_Cart\_sub(GridComm, Subdims, &RowComm);

    // Creating communicators for columns

    Subdims[0] = 1;

    Subdims[1] = 0;

    MPI\_Cart\_sub(GridComm, Subdims, &ColComm);

}

// Function for memory allocation and data initialization

void ProcessInitialization(double\*& pAMatrix, double\*& pBMatrix,

    double\*& pCMatrix, double\*& pAblock, double\*& pBblock, double\*& pCblock,

    double\*& pTemporaryAblock, int& Size, int& BlockSize) {

    if (ProcRank == 0) {

        bool flag = false;

        Size = max(Rows, Columns);

        /\*do {

            printf("\nEnter the size of matrices: ");

            scanf("%d", &Size);

            if (Size % GridSize != 0) {

                printf("Size of matrices must be divisible by the grid size!\n");

            }

        } while (Size % GridSize != 0);//\*/

    }

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

    BlockSize = Size / GridSize;

    pAblock = new double[BlockSize \* BlockSize];

    pBblock = new double[BlockSize \* BlockSize];

    pCblock = new double[BlockSize \* BlockSize];

    pTemporaryAblock = new double[BlockSize \* BlockSize];

    for (int i = 0; i < BlockSize \* BlockSize; i++) {

        pCblock[i] = 0;

    }

    if (ProcRank == 0) {

        pAMatrix = new double[Size \* Size];

        pBMatrix = new double[Size \* Size];

        pCMatrix = new double[Size \* Size];

#ifdef Dummy

        DummyDataInitialization(pAMatrix, pBMatrix, Size);

#else

        RandomDataInitialization(pAMatrix, pBMatrix, Size);

#endif // Dummy

    }

}

// Function for checkerboard matrix decomposition

void CheckerboardMatrixScatter(double\* pMatrix, double\* pMatrixBlock,

    int Size, int BlockSize) {

    double\* MatrixRow = new double[BlockSize \* Size];

    if (GridCoords[1] == 0) {

        MPI\_Scatter(pMatrix, BlockSize \* Size, MPI\_DOUBLE, MatrixRow,

            BlockSize \* Size, MPI\_DOUBLE, 0, ColComm);

    }

    for (int i = 0; i < BlockSize; i++) {

        MPI\_Scatter(&MatrixRow[i \* Size], BlockSize, MPI\_DOUBLE,

            &(pMatrixBlock[i \* BlockSize]), BlockSize, MPI\_DOUBLE, 0, RowComm);

    }

    delete[] MatrixRow;

}

// Data distribution among the processes

void DataDistribution(double\* pAMatrix, double\* pBMatrix, double\*

    pMatrixAblock, double\* pBblock, int Size, int BlockSize) {

    // Scatter the matrix among the processes of the first grid column

    CheckerboardMatrixScatter(pAMatrix, pMatrixAblock, Size, BlockSize);

    CheckerboardMatrixScatter(pBMatrix, pBblock, Size, BlockSize);

}

// Function for gathering the result matrix

void ResultCollection(double\* pCMatrix, double\* pCblock, int Size,

    int BlockSize) {

    double\* pResultRow = new double[Size \* BlockSize];

    for (int i = 0; i < BlockSize; i++) {

        MPI\_Gather(&pCblock[i \* BlockSize], BlockSize, MPI\_DOUBLE,

            &pResultRow[i \* Size], BlockSize, MPI\_DOUBLE, 0, RowComm);

    }

    if (GridCoords[1] == 0) {

        MPI\_Gather(pResultRow, BlockSize \* Size, MPI\_DOUBLE, pCMatrix,

            BlockSize \* Size, MPI\_DOUBLE, 0, ColComm);

    }

    delete[] pResultRow;

}

// Broadcasting blocks of the matrix A to process grid rows

void ABlockCommunication(int iter, double\* pAblock, double\* pMatrixAblock,

    int BlockSize) {

    // Defining the leading process of the process grid row

    int Pivot = (GridCoords[0] + iter) % GridSize;

    // Copying the transmitted block in a separate memory buffer

    if (GridCoords[1] == Pivot) {

        for (int i = 0; i < BlockSize \* BlockSize; i++)

            pAblock[i] = pMatrixAblock[i];

    }

    // Block broadcasting

    MPI\_Bcast(pAblock, BlockSize \* BlockSize, MPI\_DOUBLE, Pivot, RowComm);

}

// Function for cyclic shifting the blocks of the matrix B

void BblockCommunication(double\* pBblock, int BlockSize) {

    MPI\_Status Status;

    int NextProc = GridCoords[0] + 1;

    if (GridCoords[0] == GridSize - 1) NextProc = 0;

    int PrevProc = GridCoords[0] - 1;

    if (GridCoords[0] == 0) PrevProc = GridSize - 1;

    MPI\_Sendrecv\_replace(pBblock, BlockSize \* BlockSize, MPI\_DOUBLE,

        NextProc, 0, PrevProc, 0, ColComm, &Status);

}

// Function for parallel execution of the Fox method

void ParallelResultCalculation(double\* pAblock, double\* pMatrixAblock,

    double\* pBblock, double\* pCblock, int BlockSize) {

    for (int iter = 0; iter < GridSize; iter++) {

        // Sending blocks of matrix A to the process grid rows

        ABlockCommunication(iter, pAblock, pMatrixAblock, BlockSize);

        // Block multiplication

        BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);

        // Cyclic shift of blocks of matrix B in process grid columns

        BblockCommunication(pBblock, BlockSize);

    }

}

// Test printing of the matrix block

void TestBlocks(double\* pBlock, int BlockSize, char str[]) {

    MPI\_Barrier(MPI\_COMM\_WORLD);

    if (ProcRank == 0) {

        printf("%s \n", str);

    }

    for (int i = 0; i < ProcNum; i++) {

        if (ProcRank == i) {

            printf("ProcRank = %d \n", ProcRank);

            PrintMatrix(pBlock, BlockSize, BlockSize);

        }

        MPI\_Barrier(MPI\_COMM\_WORLD);

    }

}

// Function for testing the matrix multiplication result

void TestResult(double\* pAMatrix, double\* pBMatrix, double\* pCMatrix,

    int Size) {

    double\* pSerialResult; // Result matrix of serial multiplication

    double Accuracy = 1.e-6; // Comparison accuracy

    int equal = 0; // =1, if the matrices are not equal

    int i; // Loop variable

    if (ProcRank == 0) {

        fstream out("result.txt", ios\_base::out | ios\_base::trunc);

        out << "A: " << endl;

        for (i = 0; i < Rows; i++)

        {

            for (int j = 0; j < Columns; j++) {

                out << pAMatrix[i \* Rows + j] << " ";

            }

            out << endl;

        }

        out << "B: " << endl;

        for (i = 0; i < Rows; i++)

        {

            for (int j = 0; j < 1; j++) {

                out << pBMatrix[i \* Rows + j] << " ";

            }

        }

        out << endl;

        out << "Result: " << endl;

        for (i = 0; i < Rows; i++)

        {

            for (int j = 0; j < 1; j++) {

                out << pCMatrix[i \* Rows + j] << " ";

            }

        }

        pSerialResult = new double[Size \* Size];

        for (i = 0; i < Size \* Size; i++) {

            pSerialResult[i] = 0;

        }

        BlockMultiplication(pAMatrix, pBMatrix, pSerialResult, Size);

        for (i = 0; i < Size \* Size; i++) {

            if (fabs(pSerialResult[i] - pCMatrix[i]) >= Accuracy)

                equal = 1;

        }

        out << endl;

        out << "Control: " << endl;

        for (i = 0; i < Rows; i++)

        {

            for (int j = 0; j < 1; j++) {

                out << pSerialResult[i \* Rows + j] << " ";

            }

        }

        if (equal == 1)

            printf("The results of serial and parallel algorithms are NOT"

                "identical. Check your code.");

        else

            printf("The results of serial and parallel algorithms are "

                "identical. ");

    }

}

// Function for computational process termination

void ProcessTermination(double\* pAMatrix, double\* pBMatrix,

    double\* pCMatrix, double\* pAblock, double\* pBblock, double\* pCblock,

    double\* pMatrixAblock) {

    if (ProcRank == 0) {

        delete[] pAMatrix;

        delete[] pBMatrix;

        delete[] pCMatrix;

    }

    delete[] pAblock;

    delete[] pBblock;

    delete[] pCblock;

    delete[] pMatrixAblock;

}

void main(int argc, char\* argv[]) {

    double\* pAMatrix; // First argument of matrix multiplication

    double\* pBMatrix; // Second argument of matrix multiplication

    double\* pCMatrix; // Result matrix

    int Size; // Size of matrices

    int BlockSize; // Sizes of matrix blocks

    double\* pAblock; // Initial block of matrix A

    double\* pBblock; // Initial block of matrix B

    double\* pCblock; // Block of result matrix C

    double\* pMatrixAblock;

    double Start, Finish, Duration;

    setvbuf(stdout, 0, \_IONBF, 0);

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);

    if (ProcNum != GridSize \* GridSize) {

        if (ProcRank == 0) {

            printf("Number of processes must be a perfect square \n");

        }

    }

    else {

        if (ProcRank == 0)

            printf("Parallel matrix multiplication program\n");

        // Creating the cartesian grid, row and column communcators

        CreateGridCommunicators();

        // Memory allocation and initialization of matrix elements

        ProcessInitialization(pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,

            pCblock, pMatrixAblock, Size, BlockSize);

        DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,

            BlockSize);

        // Execution of the Fox method

        ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,

            pCblock, BlockSize);

        // Gathering the result matrix

        ResultCollection(pCMatrix, pCblock, Size, BlockSize);

        TestResult(pAMatrix, pBMatrix, pCMatrix, Size);

        // Process Termination

        ProcessTermination(pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,

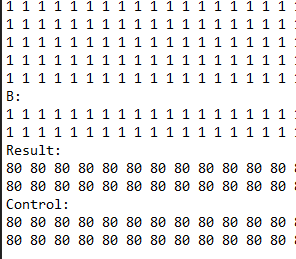
            pCblock, pMatrixAblock);

    }

    MPI\_Finalize();

}

**Результат виконання програми(вибірковий сегмент файлу з результатами):**



**Висновок:**

На цій лабораторній роботі я ознайомився з методами організації паралельного множення матриці на вектор та розробив паралельну програму з використанням технології MPI.