**Міністерство освіти і науки України**

**Національний університет «Львівська політехніка»**

**Кафедра ЕОМ**



Звіт

до лабораторної роботи № 4

з дисципліни «Паралельні та розподілені обчислення»

на тему: «Паралельні алгоритми множення матриці на вектор»

Виконав:

ст.гр. КІ-34

Мороз О.Ю.

Прийняв:

Козак Н.Б.

**Львів 2020**

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

**Завдання:**

Розробити програму для паралельного перемноження матриці на вектор заданого розміру з використанням МРІ. Тип розбиття – блокове. Кількість процесорів – 7.

Варіант №18:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ варіанту** | **Розмір матриці** | | **Тип розбиття** | **Кількість процесорів** |
| 2 | 70 | 1200 | блокове | 7 |

**Хід роботи:**

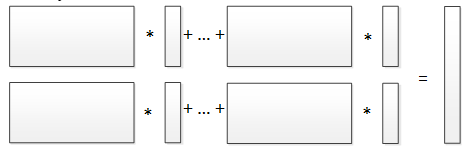
Виходячи з вище вказаних тверджень можливі розміри блоків

7 блоків: 10x1200

Розібємо дані блоки між процесорами



З даної схеми видно, що матриця розбита на 7 блоків однакового розміру та кількість елементів, які входять в усі блоки однакова.



Отже на кожний процесор буде переслано (70∙1200)/10+70+1200=9670 елементів.

Кількість операцій на кожному процесорі буде рівна:

70(2∙1200-1)= 167930

**Код програми:**

#include<stdio.h>

#include<stdlib.h>

#include <conio.h>

#include<mpi.h>

#include<time.h>

void ProcessInitialization(double\* &pMatrix, double\* &pVector, double\* &pResult, double\* &pProcRows, double\* &pProcResult, int &RowNum);

void DataDistribution(double\* pMatrix, double\* pProcRows, double\* pVector, int RowNum);

void ParallelResultCalculation(double\* pProcRows, double\* pVector, double\* pProcResult, int RowNum);

void ResultReplication(double\* pProcResult, double\* pResult, int RowNum);

void RandomDataInitialization(double \* &pMatrix, double \* &pVector);

void ProcessTermination(double \* pMatrix, double \* pVector, double \* pResult, double \* pProcRows, double \* pProcResult);

void SeperateMatrix(double\* pNewMatrix, double\* pMatrix, int \* pSendInd, int ColNum, int RowNum);

void AddSubMatrix(double \*pProcResult, int ColNum, int RowNum);

void Count(double \* &pMatrix, double \* &pVector, double \*\* &pMat, double \* &pRes);

int ProcNum, ProcRank;

int Row = 70, Col = 1200; // Розміри початкової матриці і вектора double\* pProcRows;

int ARow, ACol, RowNum1, ColNum1;

int bFlag = 0;

// Множення матриці на вектор - блокове розбиття

// (початковий і результуючий вектори дублюються між процесами)

void main(int argc, char\* argv[])

{

double\* pMatrix; // Перший аргумент - початкова матриця

double\* pVector; // Другий аргумент - початковий вектор

double\* pResult; // Результат множення матриці на вектор

double\* pProcRows;

double\* pProcResult;

double\*\* pMat;

double\* pRes;

int RowNum;

double Start, Finish, Duration;

MPI\_Init(&argc, &argv);

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

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

// Виділення пам'яті і ініціалізація початкових даних

ProcessInitialization(pMatrix, pVector, pResult, pProcRows, pProcResult, RowNum);

// Розподіл початкових даних між процесами

DataDistribution(pMatrix, pProcRows, pVector, RowNum);

// Паралельне виконання множення матриці на вектор

ParallelResultCalculation(pProcRows, pVector, pProcResult, RowNum);

// Збір результуючого вектора на всіх процесах

ResultReplication(pProcResult, pResult, RowNum);

MPI\_Finalize();

// Завершення процесу обчислень

Count(pMatrix, pVector, pMat, pRes);

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

printf("C[%d] = %6.2f\n", i, pMatrix[i] \* pVector[i]);

}\*/

ProcessTermination(pMatrix, pVector, pResult, pProcRows, pProcResult);

}

// Функція для виділення пам'яті і ініціалізації початкових даних

void ProcessInitialization(double\* &pMatrix, double\* &pVector, double\* &pResult, double\* &pProcRows, double\* &pProcResult, int &RowNum)

{

int Matrix;

int SubMatrix;

int i;

Matrix = Row \* Col;

SubMatrix = Matrix / ProcNum;

if (SubMatrix%Row == 0)

{

ColNum1 = SubMatrix / (Row / 2);

RowNum1 = Row / 2;

}

else

if (SubMatrix%Col == 0)

{

RowNum1 = SubMatrix / (Col / 2);

ColNum1 = Col / 2;

}

if (Row%RowNum1 == 0)

{

if (Col%ColNum1 == 0)

{

printf("%d blocks: %dx%d\n", ProcNum, RowNum1, ColNum1);

}

else

{

printf("%d blocks: %dx%d\n", ProcNum - 1, RowNum1, ColNum1);

printf("1 block: %dx%d\n", Row, ColNum1 / 2);

ARow = Row;

ACol = ColNum1 / 2;

bFlag = 1;

}

}

else

{

if (Col%ColNum1 == 0)

{

printf("%d blocks: %dx%d\n", ProcNum - 1, RowNum1, ColNum1);

printf("1 block: %dx%d\n", RowNum / 2, Col);

ARow = RowNum / 2;

ACol = Col;

bFlag = 1;

}

}

int RestRows; // Кількість рядків матриці, які ще

// не розподілені

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

RestRows = Row;

for (int i = 0; i < ProcRank; i++)

RestRows = RestRows - RestRows / (ProcNum - i);

RowNum = RestRows / (ProcNum - ProcRank);

pVector = new double[Col];

pResult = new double[Row];

pProcRows = new double[RowNum\*Col];

pProcResult = new double[RowNum];

if (ProcRank == 0) {

pMatrix = new double[Row\*Col];

RandomDataInitialization(pMatrix, pVector);

}

}

// Функція для розбиття початкових даних між процесами

void DataDistribution(double\* pMatrix, double\* pProcRows, double\* pVector, int RowNum)

{

int \*pSendNum; // Кількість елементів, що посилаються процесу

int \*pSendInd; // Індекс першого елементу даних

// посиланого процесу

int RestRows = Row; // Кількість рядків матриці, які ще

// не розподілені

MPI\_Bcast(pVector, Col, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Виділення пам'яті для зберігання тимчасових об'єктів

pSendInd = new int[ProcNum];

pSendNum = new int[ProcNum];

// Визначення положення рядків матриці, призначених

// кожному процесу

RowNum = (Row / ProcNum);

pSendNum[0] = RowNum \* Col;

pSendInd[0] = 0;

for (int i = 1; i < ProcNum; i++) {

RestRows -= RowNum;

RowNum = RestRows / (ProcNum - i);

pSendNum[i] = RowNum \* Col;

pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];

}

// Розсилка рядків матриці

MPI\_Scatterv(pMatrix, pSendNum, pSendInd, MPI\_DOUBLE, pProcRows,

pSendNum[ProcRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Звільнення пам'яті

delete[] pSendNum;

delete[] pSendInd;

}

// Функція для обчислення частини результуючого вектора

void ParallelResultCalculation(double\* pProcRows, double\* pVector, double\* pProcResult, int RowNum) {

int i, j;

for (i = 0; i < RowNum; i++) {

pProcResult[i] = 0;

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

pProcResult[i] += pProcRows[i\*Col + j] \* pVector[j];

}

AddSubMatrix(pProcResult, RowNum, ColNum1);

}

// Функція для збору результуючого вектора на всіх процесах

void ResultReplication(double\* pProcResult, double\* pResult, int RowNum)

{

int \*pReceiveNum; // Кількість елементів, що посилаються процесом

int \*pReceiveInd; // Індекс елементу даних в результуючому

// векторі

int RestRows = Row; // Кількість рядків матриці, які ще не

// розподілені

int i;

// Виділення пам'яті для тимчасових об'єктів

pReceiveNum = new int[ProcNum];

pReceiveInd = new int[ProcNum];

// Визначення положення блоків результуючого вектора

pReceiveInd[0] = 0;

pReceiveNum[0] = Row / ProcNum;

for (i = 1; i < ProcNum; i++) {

RestRows -= pReceiveNum[i - 1];

pReceiveNum[i] = RestRows / (ProcNum - i);

pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];

}

// Збір всього результуючого вектора на всіх процесах

MPI\_Allgatherv(pProcResult, pReceiveNum[ProcRank],

MPI\_DOUBLE, pResult, pReceiveNum, pReceiveInd,

MPI\_DOUBLE, MPI\_COMM\_WORLD);

// Звільнення пам'яті

delete[] pReceiveNum;

delete[] pReceiveInd;

}

void RandomDataInitialization(double \* &pMatrix, double \* &pVector)

{

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

{

if (i < Col)

pVector[i] = (double)(rand() % 10 + 1);

pMatrix[i] = (double)(rand() % 10 + 1);

}

}

void Count(double \* &pMatrix, double \* &pVector, double \*\* &pMat, double \* &pRes)

{

pRes = new double[Row];

pMat = new double\*[Row];

for (int i = 0; i < Row; i++)

pMat[i] = new double[Col];

for (int i = 0, k = 0; i < Row; ++i) {

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

pMat[i][j] = pMatrix[k++];

//printf("M[%d][%d] = %6.2f\n", i, j, pMat[i][j]);

}

}

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

pRes[i] = 0;

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

pRes[i] += pMat[i][j] \* pVector[j];

}

printf("k[%d] = %6.2f\n", i, pRes[i]);

}

printf("\n================================================\n");

delete[] pRes;

for (int count = 0; count < Row; count++)

delete[]pMat[count];

}

void ProcessTermination(double \* pMatrix, double \* pVector, double \* pResult, double \* pProcRows, double \* pProcResult)

{

for (int i = 0; i < Row; i++)

printf("C[%d] = %6.2f\n", i, pResult[i]);

delete[] pMatrix;

delete[] pVector;

delete[] pResult;

delete[] pProcRows;

delete[] pProcResult;

}

void SeperateBlocks(double\* &pNewMatrix, double\* pMatrix, int \* &pSendInd, int ColNum, int RowNum)

{

int l = 0;

if (bFlag && ARow == Row)

{

for (int m = 0; m < 2; m++)

for (int k = 0; k < ProcNum / 2; k++) {

for (int i = m \* RowNum; i < (m + 1)\*RowNum; i++)

for (int j = k \* ColNum; j < (k + 1)\*ColNum; j++, l++)

pNewMatrix[l] = pMatrix[i\*Col + j];

pSendInd[m + k] = (m + k)\*ColNum\*RowNum;

}

for (int i = 0; i < Row; i++)

for (int j = Col - ColNum / 2; j < Col; j++, l++)

pNewMatrix[l] = pMatrix[i\*Col + j];

pSendInd[ProcNum - 1] = (ProcNum - 1)\*ColNum\*RowNum;

}

if (bFlag && ACol == Col)

{

for (int m = 0; m < ProcNum / 2; m++)

for (int k = 0; k < 2; k++) {

for (int i = m \* RowNum; i < (m + 1)\*RowNum; i++)

for (int j = k \* ColNum; j < (k + 1)\*ColNum; j++, l++)

pNewMatrix[l] = pMatrix[i\*Col + j];

pSendInd[m + k] = (m + k)\*ColNum\*RowNum;

}

for (int i = Row - RowNum / 2; i < Row; i++)

for (int j = 0; j < Col; j++, l++)

pNewMatrix[l] = pMatrix[i\*Col + j];

pSendInd[ProcNum - 1] = (ProcNum - 1)\*ColNum\*RowNum;

}

if (!bFlag)

{

for (int m = 0; m < 2; m++)

for (int k = 0; k < ProcNum / 2; k++)

for (int i = m \* RowNum; i < (m + 1)\*RowNum; i++) {

for (int j = k \* ColNum; j < (k + 1)\*ColNum; j++, l++)

pNewMatrix[l] = pMatrix[i\*Col + j];

pSendInd[m + k] = (m + k)\*ColNum\*RowNum;

}

}

}

void AddSubMatrix(double \*pProcResult, int ColNum, int RowNum)

{

double \*pNewResult = new double[Col];

for (int m = Col; m < 2; m++)

for (int k = 0; k < ProcNum / 2; k++)

for (int i = m \* RowNum; i < (m + 1)\*RowNum; i++)

for (int j = k \* ColNum; j < (k + 1)\*ColNum; j++)

pNewResult[j] += pProcResult[i\*Col + j];

}

**Результати виконання програми:**

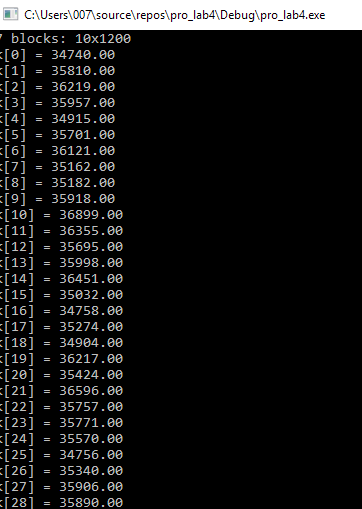


Рис.1. Результат виконання програми

**Висновок:** Дослідив можливості розв’язання різноманітних задач за допомогою паралельних алгоритмів. Навчився виділяти незалежні гілки обчислень та виконувати їх паралельно.