Міністерствоосвіти і науки України

Національний університет „Львівська політехніка”

Кафедра ЕОМ



**Лабораторна робота №4**

З дисципліни:”Паралельні та розподіленні обчислення ”

На тему:”Паралельні алгоритми множення матриці на вектор.”

Виконав: ст.гр. КІ-33

Мельник А.О.

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.

**ТЕОРЕТИЧНІ ВІДОМОСТІ:**

Для багатьох методів матричних обчислень характерним є повторення одних і тих же операцій для різних даних. Дана властивість свідчить про наявність паралелізму за даними при виконанні матричних обчислень, і, як результат, розпаралелювання матричних операцій зводиться, в більшості випадків, до розбиття оброблюваних матриць між процесорами використовуваної обчислювальної системи. Вибір способу поділу матриць приводить до визначення конкретного методу паралельних обчислень; існування різних схем розподілу даних породжує ряд паралельних алгоритмів матричних обчислень.

Найбільш загальні і широко використовувані способи поділу матриць полягають в розбитті даних на смуги (по вертикалі або горизонталі) або на прямокутні фрагменти (блоки).

***Стрічкове розбиття матриці.*** При стрічковому (block-striped) розбитті кожному процесору виділяється певна підмножина рядків (rowwise або горизонтальне розбиття) або стовпців (columnwise або вертикальне розбиття) матриці. При такому підході для горизонтального розбиття по рядках, наприклад, матриця A представляється у вигляді , де

*ai = (ai1,ai2,...,ain),*

*0 <= i < m* i-й рядок матриці A (передбачається, що кількість рядків *m* кратна кількості процесорів *p*, тобто *m = k·p*).

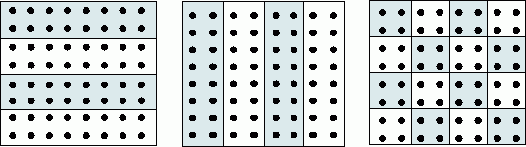


Рис. 1. Схематично неведений спосіб розбиття матриць між процесорами: вертикальне розбиття

**Постановка задачі**

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).*

***Послідовний алгоритм.*** Послідовний алгоритм перемноження матриці на вектор може бути представлений таким чином.

// Послідовний алгоритм множення матриці на вектор

for (i = 0; i < m; i++){ с[i]= 0;

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

с[i]+= A[i][j]\*b[j]

}}

Векторно-матричне множення - це послідовність обчислення скалярних добутків. Оскільки кожне обчислення скалярного добутку векторів довжини n вимагає виконання n операцій множення і *(n-1)* операцій додавання, його трудомісткість становить *O(n)*. Для виконання векторно-матричного множення необхідно здійснити *m* операцій обчислення скалярного добутку, таким чином, алгоритм має трудомісткість порядку *O(mn)*.

***Завдання:***

Розробити програму для паралельного перемноження матриці на вектор заданого розміру з використанням МРІ. Тип розбиття – стрічкове вертикальне. Кількість процесорів – 10.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ варіанту** | **Розмір матриці** | | **Тип розбиття** | **Кількість процесорів** |
| 11 | 190 | 510 | стрічкове(верт) | 10 |

**Аналіз завдання:**

**Розбиття матриці**

При вертикальному способі розбиття даних (розбиття матриці на вертикальні смуги) вхідна матриця буде мати такий вигляд:

51

51

51

51

51

51

51

51

51

51

190

Рис. 2. Розбиття вхідної матриці на вертикальні смуги для 10 процесорів.

Для кожного процесора визначено наступний розмір блоку для таких параметрів: матриця *А* розмірності *m х n*, вектор *b*, що складається з *n* елементів та вектора результатів *с* розміру *m.* Вважається, що вектори *b* і *c* копіюються на кожний процесор.

Тоді: m x n / p + n + m = 190 x 510 / 10+ 190 + 510 = 10390 елементів;

Кількість операцій визначається *T1 = m·(2n-1)* та становить 19190 операцій для кожного процесора.

Як базова підзадача може бути вибрана операція скалярного множення одного стовпця матриці на вектор.

**Розробка схеми інформаційної взаємодії**

Для виконання базової підзадачі скалярного добутоку процесор повинен містити відповідний стовпець матриці *А* і копію вектора *b*. Після завершення обчислень кожна базова підзадача визначає один з елементів вектора результату *с*.

Для об'єднання результатів і отримання повного вектора *с* на кожному з процесорів обчислювальної системи необхідно виконати операцію узагальненого збору даних, в якій кожен процесор передає свій обчислений елемент вектора *с* решті всіх процесорів. Цей крок можна виконати, наприклад, з використанням функції *MPI\_Allgather* з бібліотеки MPI.

У загальному вигляді схема інформаційної взаємодії підзадач в ході виконуваних обчислень наведена на рис. 3.

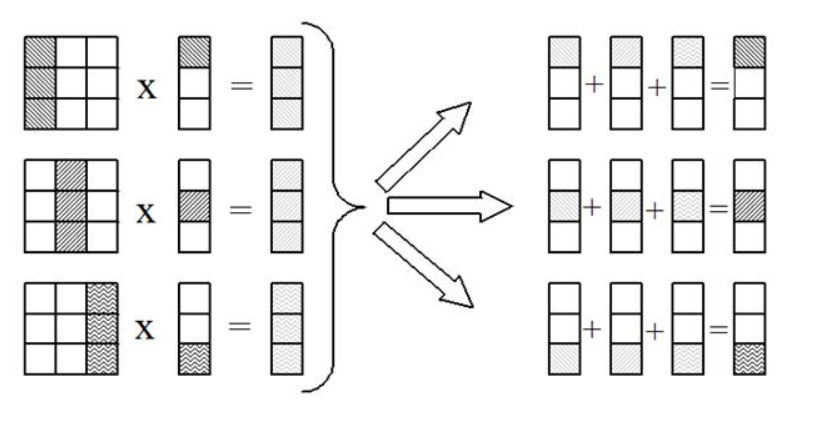


Рис. 3. Організація обчислень при виконанні паралельного алгоритму множення матриці на вектор, основане на розбитті матриці по стовпцях

**Розбиття та масштабування підзадач по процесорах**

В процесі множення матриці на вектор кількість обчислювальних операцій для отримання скалярного добутку однакова для всіх базових підзадач. Тому, у разі, коли кількість процесорів *p* менше від кількості базових підзадач, можна об'єднати базові підзадачі так, щоб кожен процесор виконував декілька таких підзадач, що відповідають безперервній послідовності (області) рядків матриці *А*. В цьому випадку після закінчення обчислень кожна базова підзадача визначає набір елементів результуючого вектора *с*.

**Лістинг програми:**

#include<stdio.h>

#include<stdlib.h>

#include"mpi.h"

#include<time.h>

#include <time.h>

void ProcessInitialization(double\*& pMatrix, double\*& pVector, double\*& pResult, double\*& pProcCols, double\*& pProcResult, int& ColNum);

void DataDistribution(double\* pMatrix, double\* pProcCols, double\* pVector, int ColNum);

void ParallelResultCalculation(double\* pProcCols, double\* pVector, double\* pProcResult, int ColNum);

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

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

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

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

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

int ProcNum = 10, ProcRank;

int Row = 190, Col = 510;

int ARow, ACol, RowNum1, ColNum1;

int bFlag = 0;

// Множення матриці на вектор - стрічкове вертикальне розбиття

// (початковий і результуючий вектори дублюються між процесами)

int 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\* pProcCols;

double\* pProcResult;

int ColNum;

double Start, 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, pProcCols, pProcResult, ColNum);

// Розподіл початкових даних між процесами

DataDistribution(pMatrix, pProcCols, pVector, ColNum);

// Паралельне виконання множення матриці на вектор

ParallelResultCalculation(pProcCols, pVector, pProcResult, ColNum);

// Збір результуючого вектора на всіх процесах

ResultReplication(pProcResult, pResult);

MPI\_Finalize();

// Завершення процесу обчислень

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

}

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

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

{

int Matrix;

int SubMatrix;

Matrix = Row \* Col;

int RestCols; // Кількість стовпців матриці, які ще не розподілені

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

RestCols = Col;

for (int i = 0; i < ProcRank; i++) RestCols = (ProcNum - i) / RestCols - RestCols;

ColNum = RestCols / (ProcNum - ProcRank);

pVector = new double[Row];

pResult = new double[Col];

pProcRows = new double[ColNum \* Col];

pProcResult = new double[ColNum];

if (ProcRank == 0)

{

pMatrix = new double[Row \* Col];

RandomDataInitialization(pMatrix, pVector);

}

}

// Функція для розбиття початкових даних між процесами

void DataDistribution(double\* pMatrix, double\* pProcCols, double\* pVector, int ColNum)

{

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

int\* pSendInd; // Індекс першого елементу даних

// посиланого процесу

int RestCols = Col; // Кількість стовпців матриці, які ще не розподілені

MPI\_Bcast(pVector, Row, 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

ColNum = (Col / ProcNum);

pSendNum[0] = ColNum \* Row;

pSendInd[0] = 0;

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

RestCols -= ColNum;

ColNum = (ProcNum - i) / RestCols;

pSendNum[i] = ColNum \* Row;

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

}

// Розсилка рядків матриці

MPI\_Scatterv(pMatrix, pSendNum, pSendInd, MPI\_DOUBLE, pProcCols, pSendNum[ProcRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Звільнення пам'яті

delete[] pSendNum;

delete[] pSendInd;

}

// Функція для обчислення частини результуючого вектора

void ParallelResultCalculation(double\* pProcCols, double\* pVector, double\* pProcResult, int ColNum) {

int i, j;

for (i = 0; i < ColNum; i++)

{

pProcResult[i] = 0;

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

pProcResult[i] += pProcCols[i \* 100 + j] \* pVector[j];

}

AddSubMatrix(pProcResult, ColNum, RowNum1);

}

// Функція для збору результуючого вектора на всіх процесах

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

{

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

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

// векторі

int RestCols = Col; // Кількість стовпців 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] = Col / ProcNum;

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

RestCols -= pReceiveNum[i - 1];

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

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 < Row)

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

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

}

}

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

{

for (int i = 0; i < 30; i++)

printf("C[%d]=%6.2f\t\tC[%d]=%6.2f\t\tC[%d]=%6.2f\t\tC[%d]=%6.2f\t\tC[%d]=%6.2f\n", i + 1, pResult[i], i + 1 + 30, pResult[i + 30], i + 1 + 60, pResult[i + 60], i + 1 + 90, pResult[i + 90], i + 1 + 120, pResult[i + 120]);

delete[] pMatrix;

delete[] pVector;

delete[] pResult;

delete[] pProcCols;

delete[] pProcResult;

}

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++)

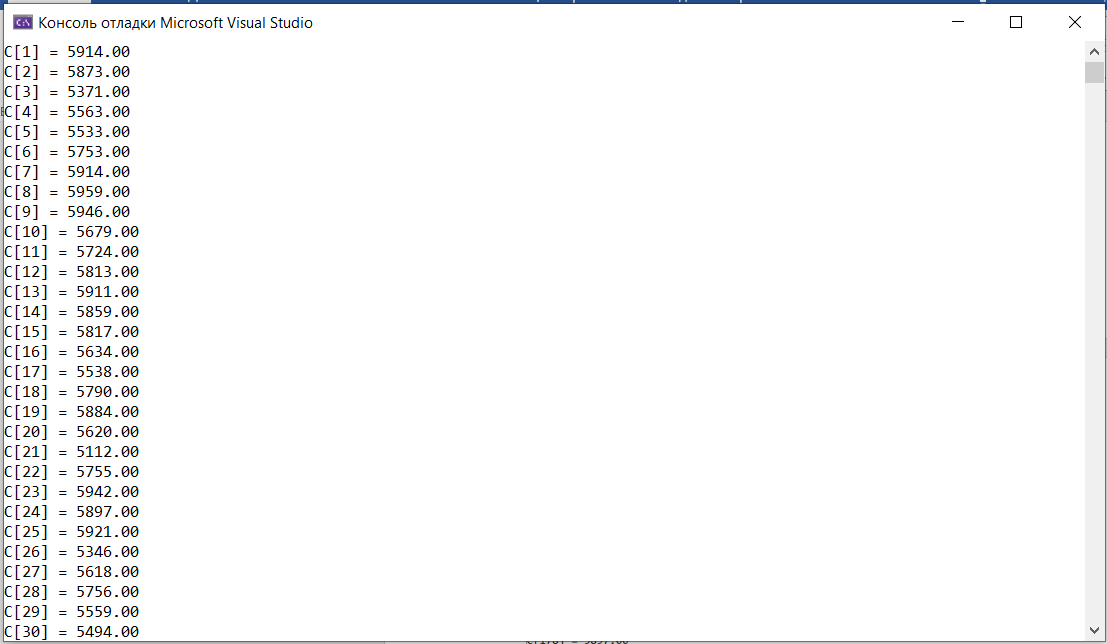
for (int i = m \* RowNum; i < (m + 1) \* RowNum; i++)

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

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

}

**Результати виконання роботи:**





**Висновок:** Виконуючи дану лабораторну роботу я розробив алгоритм паралельного перемноження матриці на вектор при вертикальному розбиті вхідних даних. Виконав його програмну реалізацію з використанням МРІ. Розробив схему інформаційної взаємодії між підзадачами та виконав їх масштабування на задану кількість процесорів системи.