**МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ**

**НУ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

**Кафедра ЕОМ**



**Звіт з лабораторної роботи №4**

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

**на тему: ” *ПАРАЛЕЛЬНІ АЛГОРИТМИ МНОЖЕННЯ МАТРИЦІ НА ВЕКТОР*”**

**Виконав: студент .гр. КІ-33**

**Фещенко З.-А.С.**

**Прийняв: викладач**

**Козак Н. Б.**

**Львів 2020 р.**

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

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

Матриці та операції над ними широко використовуються при математичному моделюванні найрізноманітніших процесів, явищ і систем. Матричні обчислення складають основу багатьох наукових і інженерних розрахунків.

Враховуючи важливість ефективного виконання матричних обчислень багато стандартних бібліотек програм містять процедури для різних матричних операцій. Об'єм програмного забезпечення для операцій над матрицями постійно збільшується - розробляються нові оптимальні структури даних для зберігання матриць спеціального типу (трикутних, стрічкових, розріджених тощо), створюються різні високоефективні машинно-залежні реалізації алгоритмів, проводяться теоретичні дослідження для пошуку швидших методів матричних обчислень.

Будучи обчислювально трудомісткими, матричні обчислення є класичною областю застосування паралельних обчислень. З одного боку, використання високопродуктивних багатопроцесорних систем дозволяє істотно підвищити складність завдань, які розв’язуються. З іншого боку, через своє достатньо просте формулювання матричні операції надають прекрасну можливість для демонстрації багатьох прийомів і методів паралельного програмування.

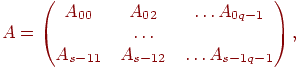
У даній лабораторній роботі аналізуються методи паралельних обчислень для операції векторно-матричного множення.

**1. Принципи розпаралелювання**

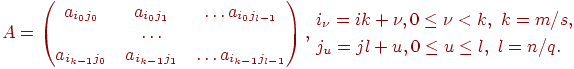
Для багатьох методів матричних обчислень характерним є повторення одних і тих же операцій для різних даних. Дана властивість свідчить про наявність паралелізму за даними при виконанні матричних обчислень, і, як результат, розпаралелювання матричних операцій зводиться, в більшості випадків, до розбиття 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).

*1.2. Блокове розбиття матриці.* При блоковому (chessboard block) розбитті матриця ділиться на прямокутні набори. Хай кількість процесорів складає p = s·q, кількість рядків матриці є кратним s, а кількість стовпців - кратним q, тобто m = k·s і n = l·q. Представимо початкову матрицю A у вигляді набору прямокутних блоків таким чином (3):

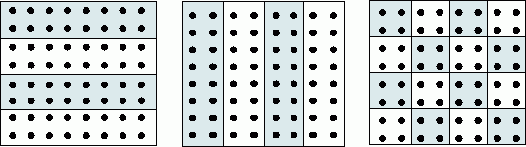
 (3)

де Aij - блок матриці, що складається з елементів:

 (4)

При такому підході доцільно, щоб обчислювальна система мала фізичну або, принаймні, логічну топологію процесорних граток з s рядків і q стовпців. При такому розбитті даних, сусідні в структурі граток процесори, обробляють суміжні блоки початкової матриці. Треба зазначити, що і для блокової схеми може бути застосоване циклічне чергування рядків і стовпців.

У лабораторній роботі розглядаються три паралельні алгоритми для множення квадратної матриці на вектор. Кожен підхід заснований на різному типі розбиття початкових даних (елементів матриці і вектора) між процесорами. Розбиття даних міняє схему взаємодії процесорів, тому кожен з представлених методів істотним чином відрізняється від решти.



а) б) в)

Рис. 1. Способи розбиття елементів матриці між процесорами

обчислювальної системи

На рис. 1 схематично неведені способи розбиття матриць між процесорами: а) горизонтальне розбиття, б) вертикальне розбиття та в) блокове розбиття матриці.

**2. 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*:

  (4)

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).* (5)

*2.1. 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)*.

*2.2. Розділення даних.*При виконанні паралельних алгоритмів перемноження матриці на вектор, окрім матриці *А*, необхідно розбити вектор *b* і вектор результату *с*. Елементи векторів можна продублювати, тобто скопіювати всі елементи вектора на всі процесори, складові багатопроцесорної обчислювальної системи, або розділити між процесорами. При блоковому розбитті вектора з n елементів кожен процесор обробляє безперервну послідовність із k елементів вектора (припускається, що розмірність вектора n без остачі ділиться на число процесорів, тобто n= k·p).

Пояснимо, чому дублювання векторів *b* і *с* між процесорами є допустимим рішенням (далі для простоти викладу вважатимемо, що *m=n*). Вектори *b* і *с* складаються з *n* елементів, тобто містять стільки ж даних, скільки і один рядок або один стовпець матриці. Якщо процесор зберігає рядок або стовпець матриці і одиночні елементи векторів *b* і *с*, то загальне число елементів, що зберігаються, має трудомісткість порядку *O(n)*. Якщо процесор зберігає рядок (стовпець) матриці і всі елементи векторів *b* і *с*, то загальна кількість елементів, що зберігаються, також порядку *O(n)*. Таким чином, при дублюванні і при розбитті векторів вимоги до об'єму пам'яті з одного класу складності.

**ЗАВДАННЯ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ варіанту** | **Розмір матриці** | | **Тип розбиття** | **Кількість процесорів** |
| 19 | 70 | 1200 | Блокове | 7 |

**ВИКОНАННЯ ЗАВДАННЯ**

Виходячи з вище вказаних тверджень можливі розміри блоків

7 blocks: 70x1200

Розібємо дані блоки між процессорами

70

1200

1200

1200

1200

1200

1200

1200

З даної схеми видно, що матриця розбита на 7 блоків однакового розміру.



Отже на кожний процесор буде переслано (490\*1200)/7=84000 елементів.

Кількість операцій на кожному процесорі буде рівна:

(1200/7\*490)(2\*490-1)= 82236000

**ТЕКСТ ПРОГРАМИ**

#include<iostream>

#include<stdio.h>

#include<stdlib.h>

#include<mpi.h>

#include<time.h>

#include <time.h>

using namespace std;

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);

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 - стрічкове горизонтальне 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;

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

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%Col == 0)

{

RowNum1 = SubMatrix / Col;

ColNum1 = Col;

}

if (Row%ProcNum == 0)

{

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

}

else

{

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

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

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 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++)

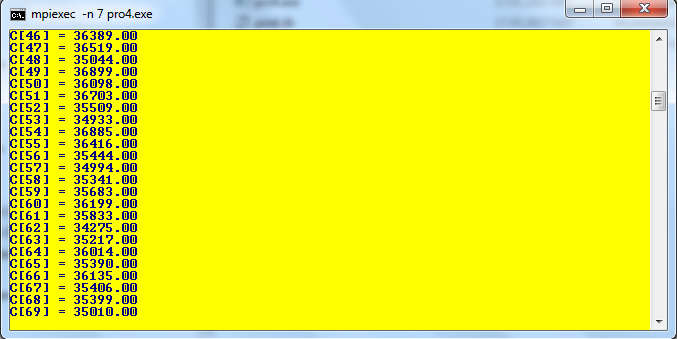
for (int k = 0; k<ProcNum / 2; k++)

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**

**ВИСНОВКИ**

Виконуючи лабораторну роботу розробив алгоритм паралельного перемноження матриці на вектор при блоковому 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. Обчислив кількість елементів та операцій для кожного процесора.