**Міністерство освіти і науки України**

**Національний університет "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**

**Кіндій В. А.**

**Прийняв: викладач**

**Козак Н. Б.**

**Львів 2020 р.**

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

**Завдання:**

1. Розробити схему інформаційної взаємодії підзадач при перемноженні матриць згідно заданого (варіанту) типу розбиття.

2. Розробити структурну схему алгоритму перемноження матриці на вектор для заданого розбиття.

3. Обчислити кількість операцій та розмір даних для кожного процесора.

4. Розробити програму для перемноження матриць з використанням МРІ.

5. Зробити висновок про ефективність застосування заданого типу розбиття для поставленої задачі.

6. Підготувати та захистити звіт.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ варіанту** | **Розмір матриці** | | **Тип розбиття** | **Кількість процесорів** |
| 5 | 140 | 70 | блокове | 7 |

**Хід роботи:**

Можливі розміри блоків розбитих на 7 блоків.

7 блоків: 140 x 10

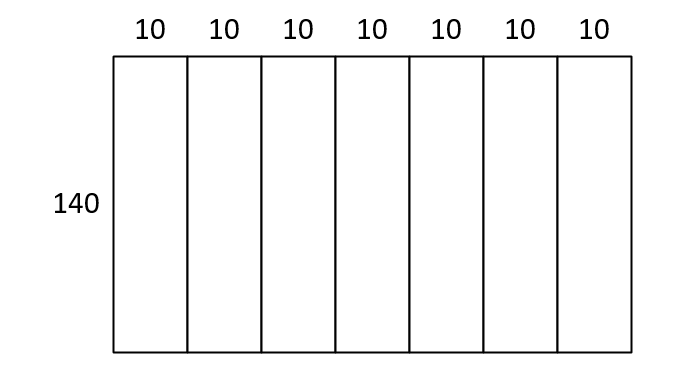
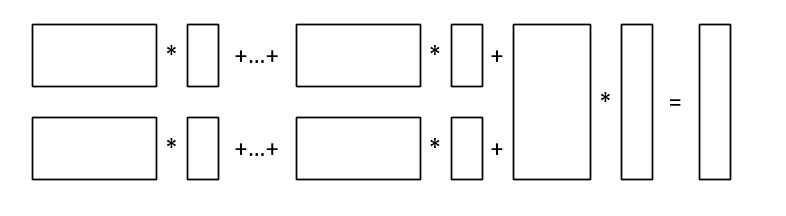


Рис 1. Схема розбиття матриці на блоки

З даної схеми видно що матриця розбита на 7 блоків однакового розміру, тільки остання підматриця має інші розміри, але кількість елементів, які входять у всі блоки однакова.



Отже на кожний процесор буде переслано (140\*70)/7 = 1400 елементів

Текст програми наведений в Додатку А.

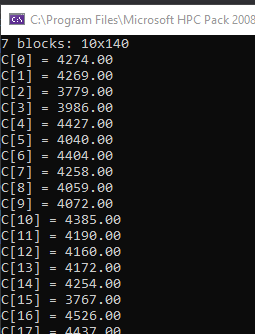


Рис 2. Результат виконання програми

**Висновок**: ознайомився з методом паралельного множення матриці на вектор та розробив паралельну програму з використанням технології MPI.

**Додаток А**

|  |
| --- |
| #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 = 140; // 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];  } |