Лабораторна робота №3

з дисципліни

Програмування комп’ютерних мереж

на тему:

**«Обмін даними за участю декількох задач в MPI»**

**Варіант №18**

|  |  |
| --- | --- |
| Виконав студент  групи КВ-64М  Подольський С. В.  залікова книжка № КВ6415 | Перевірив:  СтіренкоС. Г.  \_\_\_\_\_\_\_\_\_ |

# Мета роботи

# Вивчити види передач MPI <<один до багатьох>> та <<багато до багатьох>> та їх різновиди;

# Навчитися аналізувати алгоритми паралельних програм щодо необхідних типів передачі даних між паралельно виконуваними частинами.

# Завдання

Розробити програму залежно від варіанту:

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Метод** | **Розбиття даних** | **Організація задач** |
| 18 | Обчислення визначника | цикли стовпців | однорангова |

# Математичний апарат

**Обчислення визначника.**

LU-розклад може бути застосований до обчислення визначника матриці. За правилами знаходження визначника:

 (4.6)

Для будь-якої трикутної матриці визначник дорівнює добутку елементів головної діагоналі (дане твердження можна довести, застосувавши будь-який метод обчислення визначника), тому вираз (4.6) можна переписати у вигляді:

# 

**LU-розклад**

LU-розкладом (англ. LU-decomposition) прямокутної матриці  називається представлення матриці у вигляді добутку , де  – нижня трикутна матриця з головною діагоналлю з одиниць,  – верхня трикутна матриця.



**Алгоритм Дулітла для знаходження LU-розкладу.**

Зведення матриці до верхньотрикутного виду можливе аналогічно алгоритму Гауса. Фактично, необхідно занулили всі елементи матриці, для яких , де  – номер рядка,  – номер стовпця. Існують перетворення матриці, які не змінюють ранг та визначник матриці. До них належить операція алгебраїчної суми рядків з деяким коефіцієнтом. Таким чином, щоб занулити всі потрібні елементи першого стовпця необхідно з усіх рядків окрім першого відняти перший, помножений на відповідний коефіцієнт. Коефіцієнт може бути легко визначений з формули :



Збережемо використані коефіцієнти в матриці :



Аналогічно можна занулити елементи другого стовпця, для яких , віднімаючи з усіх рядків другий, помножений на відповідний коефіцієнт:





Процес необхідно продовжувати доти, поки всі елементи нижче головної діагоналі не стануть рівними нулю. Виконуючи такі перетворення послідовно при занулянні -го стовпця немає необхідності виконувати дії над елементами стовпців . Також при цьому не будуть змінюватися рядки . Таким чином можна побудувати наступний ітеративний алгоритм. На -му кроці обираємо ведучій елемент . Для кожного рядка  знаходимо коефіцієнт  та записуємо його у матрицю , після чого з рядка  матриці  віднімаємо поелементно рядок  помножений на . Зверніть увагу, що на кожному кроці алгоритм використовує матрицю , що була модифікована в процесі всіх попередніх кроків. Ітерація закінчується, після  кроків, де  – розмірність матриці. Після завершення ітерації діагональні елементи матриці  встановлюються рівними , а всі інші невстановлені – рівними . Таким чином маємо нижню трикутну матрицю  та верхню трикутну матрицю . Перевірити, що їх добуток дорівнює вихідній матриці  можна виконавши безпосереднє множення.

Формально алгоритм можна записати системою рівнянь для ітераційного процесу:

 (4.4)

Де також необхідно завершити обчислення  матриці у додатковому кроці за формулами

 (4.5)

**Існування та єдиність розкладу.**

LU-розклад існує тоді і тільки тоді, коли всі ведучі головні доповнювальні мінори матриці не дорівнюють нулю. Нагадаємо, що доповнювальним мінором  -го порядку матриці *A* називається визначник квадратної підматриці матриці  розмірністю *k*, отриманої вилученням з матриці одного або декількох рядків та стовпців. Головним мінором називається такий мінор, в якому номери вилучених рядків та стовпців збігаються. Ведучім називається мінор, якому відповідає прямокутна верхня ліва підматриця даної матриці *A*. Найпростішою перевіркою існування розкладу є порівняння на -тому кроці ітерації Дулітла значення  на рівність нулю. Якщо дорівнює, то LU-розклад даної матриці не існує. Розклад є єдиним, якщо головна діагональ матриці  складається з одиниць. Таким чином якщо LU-розклад існує, то він є єдиним. Також можна довести, що якщо розклад існує та визначник матриці не дорівнює нулю, то СЛАР з такими коефіцієнтами має єдиний розв'язок.

**Загальний вигляд алгоритму.**

Програма має обчислити матриці *L* та *U*. Нехай у загальному випадку маємо розмірність матриці , та систему з локальною пам'яттю на  процесорів.

Table 4.3: Перші кроки алгоритму при зберіганні по одному стовпцю

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Крок | Процесор | Процесор | Процесор | … | Процесор |
| 1 |  | – | – | … | – |
| 2 |  | | | | |
| 3 |  |  |  | … |  |
| 4 | – |  | – | … | – |
| 5 | – |  | | | |
| 6 | – |  |  | … |  |
| 7 | – | – |  | … | – |
| 8 | – | – |  | | |

Розглянемо випадок, коли в локальній пам'яті кожного процесора розміщено один стовпець матриці. На першому кроці процесор *P*1 обчислює всі значення коефіцієнтів . На другому – розсилає їх іншим процесорам. На третьому кроці всі процесори віднімають від кожного елемента в стовпці, що зберігається, значення першого помножене на . Після цих кроків елементи першого стовпця стали рівними нулю та їх подальша модифікація не потрібна. На четвертому кроці процесор *P*2 розраховує значення всіх коефіцієнтів  та на наступному розсилає їх процесорам . Далі аналогічно, з роботи виключаються процесори один за одним. Виконувані на кожному кроці дії наведені у табл. 4.3.

В даному випадку процесори починають простоювати після більшої кількості кроків, але проблема нерівномірності навантаження зберігається. Також з'являються кроки, в яких простоюють і останні процесори. Тому такий підхід не вирішує проблему ефективного використання.

В реальних випадках, як правило . Приймемо для спрощення . Тоді можливо запропонувати дві схеми розбиття матриці.

**Циклічна схема.**

В даній схемі в локальній пам'яті -го процесора зберігається кожний -тий рядок (або стовпець) починаючи з -го (тобто , , ,  ) Наведемо приклад такої схеми зберігання за рядками та за стовпцями для матриці розмірністю  в системі з локальною пам'яттю на  процесорів. Замість кожного елемента матриці вказано номер процесора, в пам'яті якого буде зберігатися даний елемент.



Після розрахунку значень першого рядка на першому кроці ітерації процесор 1 може продовжувати роботу над наступним своїм рядком. Після завершення обробки останнього рядка він має очікувати завершення обробки всіх рядків, але через обраний спосіб зберігання кожному з процесорів необхідно буде обробити лише один рядок. Швидкість даної схеми еквівалентна швидкості блочної лише у випадку . В усіх інших дана схема є більш ефективною, але потребує складнішої організації розсилки даних.

# Повний текст програми

/\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* Parallel determinant evaluation using MPI

\* Matrix LU decomposition and column cycles schema is used

\* Programming of computer networks Lab №3 (1st semester 2010)

\*

\* Author:

\* Podolsky Sergey

\* Group: KV-64M

\*

\* written: 09.12.2010

\* remarks: Compiled with Visual Studio 2010.

\* Debugged on Microsoft HPC Pack 2008

\* Tested on a cluster.

\*

\* Project definition:

\* MPI\_Lab3.cpp The entry point for the console application

\* stdafx.h Include file for standard system include files

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*/

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#include <mpi.h>

#define MATRIX\_INPUT\_FILE "matrix.txt"

#define RESULT\_FILE "det\_result.txt"

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

{

/\* Execution start time \*/

clock\_t start\_clock;

/\* MPI Initialization \*/

MPI\_Init(&argc, &argv);

/\* Get total number of processes and curent process rank \*/

int p, rank;

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

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

/\* Source matrix. Used in process 0 only \*/

double \*matrix;

/\* Source matrix size \*/

int n;

/\* Read matrix in process 0 \*/

if(rank == 0)

{

/\* Open file for reading \*/

FILE \*file = fopen(MATRIX\_INPUT\_FILE, "r");

if (file == NULL)

{

printf("Unable to open input file");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

/\* Read matrix size \*/

fscanf(file, "%d", &n);

/\* Check if matrix size is a multiple number of processors \*/

if (n % p != 0)

{

printf("Matrix size must be a multiple number of processors");

MPI\_Abort(MPI\_COMM\_WORLD, 2);

}

/\* Allocate matrix \*/

matrix = (double\*) calloc(n \* n, sizeof(double));

/\* Total count of matrix elements for each process \*/

int block\_size = n \* n / p;

/\* Fill matrix with elements from file in order of columns cycles \*/

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

// row number: i % n

// column number: i / n \* p % n + i / block\_size

fscanf(file, "%lf", &matrix[i % n \* n + i / n \* p % n + i / block\_size]);

/\* Close file \*/

fclose(file);

/\* Initialize start time value \*/

start\_clock = clock();

}

// Send matrix size to all processes

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

/\* Total count of matrix elements for each process \*/

int block\_size = n \* n / p;

double \*block = (double\*) calloc(block\_size, sizeof(double));

/\* Send a block of matrix elements to each process using "Cycles of rows" scattering \*/

if (rank == 0)

MPI\_Scatter(matrix, block\_size, MPI\_DOUBLE, block, block\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

else

MPI\_Scatter(NULL, 0, MPI\_DATATYPE\_NULL, block, block\_size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

/\* Perform matrix LU decomposition \*/

for (int pivot = 0; pivot < n - 1; pivot++)

{

/\* Array of coefficients for rows below pivot row\*/

const int l\_size = n - pivot - 1;

double\* l = (double\*) calloc(l\_size, sizeof(double));

/\* Determine process that contains pivot column \*/

if (rank == pivot % p)

{

/\* Offset of pivot column in the block \*/

int offset = pivot / p \* n;

/\* Check pivot for zero \*/

if (0 == block[offset + pivot])

{

printf("Zero pivot detected. Pivot element can not be zero");

MPI\_Abort(MPI\_COMM\_WORLD, 3);

}

/\* Fill array with coefficients values \*/

for (int i = 0; i < l\_size; i++)

l[i] = block[offset + pivot + i + 1] / block[offset + pivot];

}

/\* Send array of coefficients to all processes \*/

MPI\_Bcast(l, l\_size, MPI\_DOUBLE, pivot % p, MPI\_COMM\_WORLD);

/\* Subtract pivot row multiplied with corresponding coefficient from each row below pivot row

\* Accept changes only for columns to the right of the pivot column \*/

for (int column = rank > pivot ? 0 : (pivot - rank) / p; column < n / p; column++)

{

/\* Offset of current column in the block \*/

int offset = column \* n;

/\* Perform subtraction \*/

for (int i = 0; i < l\_size; i++)

block[offset + pivot + i + 1] -= l[i] \* block[offset + pivot];

}

}

/\* Calculate partial determinant value in each process using available columns \*/

double det = 1;

for (int pivot = rank; pivot < block\_size; pivot += n + p)

det \*= block[pivot];

/\* Write result to file \*/

if (rank == 0)

{

/\* Reduce final determinant value to process 0 \*/

MPI\_Reduce(MPI\_IN\_PLACE, &det, 1, MPI\_DOUBLE, MPI\_PROD, 0, MPI\_COMM\_WORLD);

/\* Calculate execution time in seconds\*/

clock\_t milliseconds = (clock() - start\_clock) \* 1000 / CLOCKS\_PER\_SEC;

/\* Create file \*/

FILE \*file = fopen(RESULT\_FILE, "w");

if (file == NULL)

{

printf("Failed to open output result file");

MPI\_Abort(MPI\_COMM\_WORLD, 4);

}

/\* Write result \*/

fprintf(file, "Determinant:\t%f\n", det);

fprintf(file, "Number of processes:\t%d\n", p);

fprintf(file, "Execution time:\t%ld milliseconds\n", milliseconds);

/\* Close file \*/

fclose(file);

}

else

/\* Reduce final determinant value from other processes \*/

MPI\_Reduce(&det, NULL, 1, MPI\_DOUBLE, MPI\_PROD, 0, MPI\_COMM\_WORLD);

return MPI\_Finalize();

}

# Вхідні дані (matrix.txt)

6

1 2 7 2 7 4

3 4 4 7 1 9

2 7 1 5 6 7

1 3 9 3 4 6

2 7 2 5 2 8

# 7 3 6 9 5 1

# Вихідні дані (det\_result.txt)

Determinant: 4892.000000

Number of processes: 3

Execution time: 0 milliseconds

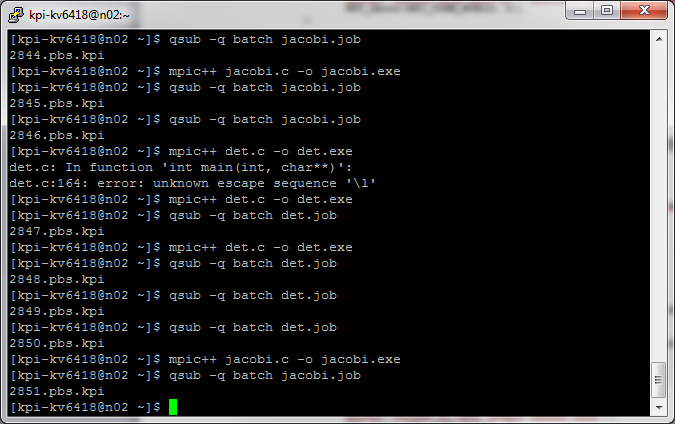


Рис. 1. Запуск програми на кластері