Лабораторна робота №4

з дисципліни

Програмування комп’ютерних мереж

на тему:

**«Користувацькі типи даних в MPI»**

**Варіант №18**

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

# Мета роботи

# Мета роботи:

# вивчити засоби середовища MPI для опису користувацьких типів і можливості щодо їх передачі;

# навчитися аналізувати алгоритм програми щодо типів даних, які будуть пересилатися між задачами;

# навчитися використовувати користувацькі типи разом з функціями обміну даними.

# Завдання

Реалізувати паралельну програму для розв'язання СЛАР методом Якобі або із застосуванням LU-розкладу.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Метод** | **Розбиття даних** | **Організація задач** |
| 18 | Розв'язання СЛАР методом Якобі | послідовний | майстер-робітник |

# Математичний апарат

**Метод Якобі**

Метод Якобі – ітеративний метод для розв'язання систем лінійних алгебраїчних рівнянь. СЛАР наступного виду:









можна записати у вигляді матричного рівняння:

 (4.1)

де:



Матрицю  можна представити у вигляді суми двох матриць  та , де  – діагональна матриця, а  містить нулі на головній діагоналі:



Тоді рівняння (4.1) можна переписати так:







Із відомого наближення розв'язку  можна отримати більш точне наближення 

 (4.2)

В якості початкового наближення  можна взяти одиничний вектор:



Нагадаємо, що обернену матрицю для діагональної  можна знайти тривіально:



Не важко впевнитись, що .

**Критерій зупинки ітерації.**

Звичайно для оцінки точності розв'язку обчислюють значення норми вектору нев'язки:



а в якості норми використовують евклідову норму:



Абсолютне значення норми вектору нев'язки відображає похибку наближення розв'язку. Але краще використовувати відносне значення, так як для матриць з малими коефіцієнтами навіть одиничний вектор в якості розв'язку може дати малу  (але звичайно одиничний вектор не є розв'язком), а для матриць з великими коефіцієнтами в деяких випадках неможливо отримати мале значення  через обмежену розрядну сітку машини.

Тому відношення норми вектору нев'язки до норми вектору  можна інтерпретувати як точність розв'язку. Тоді критерій зупинки ітерації:

 (4.3)

де *ε* – задана точність.

**Збіжність.**

Послідовність наближених значень розв'язків  є збіжною якщо матриця  має домінуючу головну діагональ, тобто:



Для деяких інших матриць метод Якобі також є збіжним.

**Загальний вигляд алгоритму.**

Програма має обрати початкове наближення  та за формулою (4.2) обчислювати наступні наближення до тих пір, доки не буде виконано критерій зупинки за формулою (4.3).

**Еквівалентні перетворення математичних формул.**

Перевірити критерій зупинки (4.3) – доволі дорога операція. Кількість обчислень, необхідна для обчислення за цією формулою, приблизно така ж, що і для обчислення наступного наближення. Тому, якщо реалізувати алгоритм у такому вигляді, на перевірку критерію запинки буде витрачено половину машинного часу. З цієї ситуації є два виходи:

• перетворити критерій зупинки та/або основну обчислювальну формулу так, щоб формули критерію зупинки використовували проміжні дані основного розрахунку;

• перевіряти критерій зупинки не після кожної ітерації, а після кожних .

Розглянемо критерій зупинки (4.3): знаменник є константним, його можна обчислити один раз після старту програми. Чисельник:

 (4.8)

Вираз у фігурних дужках є підвиразом основної формули (4.2). Тому під час обчислення наступного наближення  можна доволі дешево розрахувати нев'язку  для попереднього наближення . Якщо нев'язка для -го наближення задовольняє критерій зупинки, то необхідно взяти -те (а не останнє розраховане -е) наближення в якості розв'язку.

# Повний текст програми

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

\* Parallel system of linear equations solving using MPI

\* Jacobi method, contiguous MPI data type

\* and master-slave schema is used.

\* Programming of computer networks Lab №4 (1st semester 2010)

\*

\* Author:

\* Podolsky Sergey

\* Group: KV-64M

\*

\* written: 12.12.2010

\* remarks: Compiled with Visual Studio 2010.

\* Debugged on Microsoft HPC Pack 2008

\* Tested on a cluster.

\*

\* Project definition:

\* MPI\_Lab4.cpp The entry point for the console application

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

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

//#include "stdafx.h"

#include <stdio.h>

#include <cstdlib>

#include <time.h>

#include <mpi.h>

#include <math.h>

#define SIZE\_INPUT\_FILE "N.txt"

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

#define FREE\_MEMBERS\_INPUT\_FILE "B.txt"

#define RESULT\_FILE "X.txt"

/\* Roots precision \*/

#define EPS 0.0001

/\* Allowable number of iterations \*/

#define MAX\_INERATIONS 1000

/\* Rank of master process \*/

#define MASTER 0

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 current process rank \*/

int p, rank;

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

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

/\* Source matrix, free members vector. Used in master process only \*/

double \*a, \*b;

/\* Source matrix size \*/

int n;

/\* Read matrix in master process \*/

if (rank == MASTER)

{

/\* Open file of roots count for reading \*/

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

if (file == NULL)

{

printf("Unable to open input file ");

printf(SIZE\_INPUT\_FILE);

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

}

/\* Read count of roots \*/

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

/\* Close file \*/

fclose(file);

/\* 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);

}

/\* Open file of free members for reading \*/

file = fopen(FREE\_MEMBERS\_INPUT\_FILE, "r");

if (file == NULL)

{

printf("Unable to open input file ");

printf(FREE\_MEMBERS\_INPUT\_FILE);

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

}

// Allocate array of free members

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

// Read free members array from file

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

fscanf(file, "%lf", &b[i]);

/\* Close file \*/

fclose(file);

/\* Open file of matrix elements for reading \*/

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

if (file == NULL)

{

printf("Unable to open input file ");

printf(MATRIX\_INPUT\_FILE);

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

}

/\* Allocate matrix \*/

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

/\* Fill matrix with elements from file \*/

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

fscanf(file, "%lf", &a[i]);

/\* Close file \*/

fclose(file);

/\* Initialize start time \*/

start\_clock = clock();

}

/\* Broadcast roots count to all processes \*/

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

/\* Evaluate norm of free members vector in master process \*/

double b\_norm = 0;

if (rank == MASTER)

{

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

b\_norm += b[i] \* b[i];

b\_norm = sqrt(b\_norm);

}

/\* Broadcast norm of free members vector to all processes \*/

MPI\_Bcast(&b\_norm, 1, MPI\_DOUBLE, MASTER, MPI\_COMM\_WORLD);

/\* Row count per each process \*/

int rows\_count = n / p;

/\* Create row type as array of elements with the size of matrix \*/

MPI\_Datatype row\_type;

MPI\_Type\_contiguous(n, MPI\_DOUBLE, &row\_type);

MPI\_Type\_commit(&row\_type);

/\* Create column chunk type \*/

MPI\_Datatype chunk\_type;

MPI\_Type\_contiguous(rows\_count, MPI\_DOUBLE, &chunk\_type);

MPI\_Type\_commit(&chunk\_type);

/\* Allocate array for chunk of free members in each process \*/

double \*b\_chunk = (double\*) calloc(rows\_count, sizeof(double));

/\* Scatter free members vector to all processes \*/

if (rank == MASTER)

{

MPI\_Scatter(b, 1, chunk\_type, b\_chunk, 1, chunk\_type, MASTER, MPI\_COMM\_WORLD);

free(b);

}

else

MPI\_Scatter(NULL, 0, MPI\_DATATYPE\_NULL, b\_chunk, 1, chunk\_type, MASTER, MPI\_COMM\_WORLD);

/\* Allocate count of rows per each process \*/

double \*rows = (double\*) calloc(rows\_count \* n, sizeof(double));

/\* Scatter matrix as count of rows per process \*/

if (rank == MASTER)

{

MPI\_Scatter(a, rows\_count, row\_type, rows, rows\_count, row\_type, MASTER, MPI\_COMM\_WORLD);

free(a);

}

else

MPI\_Scatter(NULL, 0, MPI\_DATATYPE\_NULL, rows, rows\_count, row\_type, MASTER, MPI\_COMM\_WORLD);

/\* Vector of roots \*/

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

/\* Roots vector chunk\*/

double \*roots\_chunk = (double\*) calloc(rows\_count, sizeof(double));

/\* Initial roots values \*/

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

roots[i] = 1;

/\* Begin approximation iterations \*/

for (int iteration = 0; ; iteration++)

{

/\* Check for iterations limit \*/

if (rank == MASTER && iteration >= MAX\_INERATIONS)

{

printf("Iteration limit %d is reached", MAX\_INERATIONS);

MPI\_Abort(MPI\_COMM\_WORLD, 5);

}

/\* Partial residue of roots in the current process \*/

double residue = 0;

/\* Approximate each next root value \*/

for (int i = 0; i < rows\_count; i++)

{

double sum = 0;

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

sum += roots[j] \* rows[n \* i + j];

int j = rows\_count \* rank + i;

double r = b\_chunk[i] - sum;

roots\_chunk[i] = (r + roots[j] \* rows[i \* n + j]) / rows[n \* i + j];

residue += r \* r; // Error for current root

}

/\* Reduce total error \*/

// MPI\_Allreduce(MPI\_IN\_PLACE, &residue, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

if (rank == MASTER)

MPI\_Reduce(MPI\_IN\_PLACE, &residue, 1, MPI\_DOUBLE, MPI\_SUM, MASTER, MPI\_COMM\_WORLD);

else

MPI\_Reduce(&residue, NULL, 1, MPI\_DOUBLE, MPI\_SUM, MASTER, MPI\_COMM\_WORLD);

MPI\_Bcast(&residue, 1, MPI\_DOUBLE, MASTER, MPI\_COMM\_WORLD);

/\* Check stop criteria \*/

if (sqrt (residue) / b\_norm < EPS)

break;

/\* Gather all roots in all processes \*/

// MPI\_Allgather(roots\_chunk, 1, chunk\_type, roots, 1, chunk\_type, MPI\_COMM\_WORLD);

if (rank == MASTER)

MPI\_Gather(roots\_chunk, 1, chunk\_type, roots, 1, chunk\_type, MASTER, MPI\_COMM\_WORLD);

else

MPI\_Gather(roots\_chunk, 1, chunk\_type, NULL, 0, MPI\_DATATYPE\_NULL, MASTER, MPI\_COMM\_WORLD);

MPI\_Bcast(roots, 1, row\_type, MASTER, MPI\_COMM\_WORLD);

}

/\* Write results \*/

if (rank == MASTER)

{

/\* 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("Unable to open output file ");

printf(RESULT\_FILE);

MPI\_Abort(MPI\_COMM\_WORLD, 6);

}

/\* Write result \*/

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

fprintf(file, "x[%d] = \t%f\n", i, roots[i]);

fprintf(file, "\n");

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

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

/\* Close file \*/

fclose(file);

}

/\* Free resources \*/

MPI\_Type\_free(&row\_type);

MPI\_Type\_free(&chunk\_type);

free(b\_chunk);

free(roots);

free(roots\_chunk);

return MPI\_Finalize();

}

# Вхідні дані

**A.txt**

31 2 7 2 7 4

3 34 4 7 1 9

2 7 41 5 6 7

1 3 9 33 4 6

2 1 2 5 -13 1

7 3 6 9 5 32

**B.txt**

1

2

3

4

5

6

**N.txt**

6

# Вихідні дані

**X.txt**

x[0] = 0.056420

x[1] = -0.015547

x[2] = 0.074766

x[3] = 0.105215

x[4] = -0.311176

x[5] = 0.181620

Number of processes: 3

Execution time: 0 milliseconds

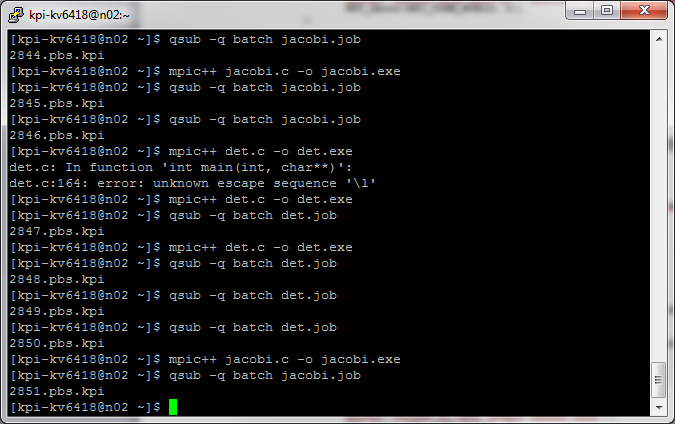


Рис. . Запуск програми на кластері