ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**КУРСОВОЙ ПРОЕКТ**

по дисциплине “Параллельные вычислительные технологии”

на тему

**Разработка параллельной MPI-программы вычисления определителя матрицы методом Гаусса**

|  |  |
| --- | --- |
| Выполнил студент | Сазонов Павел Евгеньевич |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС-641 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | доцент д.т.н. М.Г. Курносов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2018

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc533346110)

[АЛГОРИТМ ВЫЧИСЛЕНИЯ ОПРЕДЕЛИТЕЛЯ МЕТОДОМ ГАУССА 4](#_Toc533346111)

[ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ ВЫЧИСЛЕНИЯ ОПРЕДЕЛИТЕЛЯ МЕТОДОМ ГАУССА 5](#_Toc533346112)

[ЗАКЛЮЧЕНИЕ 9](#_Toc533346113)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 10](#_Toc533346114)

[ПРИЛОЖЕНИЕ 11](#_Toc533346115)

[1. Исходный код программы 11](#_Toc533346116)

# ВВЕДЕНИЕ

В рамках данного курсового проекта требуется разработать параллельную MPI-программу для вычисления определителя квадратной матрицы *A* размером *n*x*n,* методом Гаусса. Метод Гаусса сводится к последовательному исключению переменных, когда с помощью элементарных преобразований система уравнений приводится к равносильной системе треугольного или ступенчатого вида. Этот алгоритм имеет сложность O(*n*3). Эксперименты необходимо провести на кластере Jet (18 узлов: 2 х Intel Xeon E5420).

# АЛГОРИТМ ВЫЧИСЛЕНИЯ ОПРЕДЕЛИТЕЛЯ МЕТОДОМ ГАУССА

При помощи элементарных преобразований любую матрицу можно привести к верхнему (или нижнему) треугольному виду (метод Гаусса). Отсюда следует, что любую матрицу, используя элементарные преобразования, можно привести к треугольному виду, а затем вычислить определитель. [1]

Алгоритм состоит из двух шагов.

**1.** При помощи элементарных преобразований привести матрицу к треугольному или ступенчатому виду.

**2.** Вычислить определитель треугольной матрицы, перемножая её элементы, стоящие на главной диагонали.

Псевдокод последовательного метода нахождения определителя квадратной матрицы A, размером *n*x*n*.

//**Входные данные** : массив чисел *A*[0..*n*\**n*]

//**Выходные данные**: определитель *sum*

//прямой ход метода Гаусса

**for** *i* <- 0 **to** *n*-1 **do**

**for** *j* <- *i* + 1 **to** *n* **do**

*scaling* <- *A*[*j* \* *n* + *i*] / *A*[*i* \* *n* + *i*]

**for** *k* <- *i* **to** *n* **do**

*A*[*j* \* *n* + *k*] <- *A*[*j* \* *n* + *k*]- *scaling* \* *A*[*i* \* *n* + *k*]

//поиск определителя верхней треугольной матрицы

*sum* <- 1

**for** *i* <- 0 **to** *n* **do**

*sum* <- *sum* \* *A*[*i* \* *n* + *i*]

**return** sum

# ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ ВЫЧИСЛЕНИЯ ОПРЕДЕЛИТЕЛЯ МЕТОДОМ ГАУССА

Основная идея распараллеливания состоит в том, чтобы каждый процесс хранил в своей памяти n/P (P - количество процессов) строк, распределенных циклично для выравнивания вычислительной загрузки процессов. На *i*-том шаге прямого хода метода Гаусса процесс, в котором хранится *i*-тая строка, пересылает эту строку остальным процессам, после чего каждый процесс просчитывает свою часть матрицы. После выполнения прямого хода метода Гаусса, каждый процесс высчитывает свою часть определителя, и все процессы перемножают свои результаты.

Псевдокод параллельного метода нахождения определителя квадратной матрицы A, размером *n*x*n*.

//**Входные данные** : массив чисел *A*[0..*n*\**n*]

//**Выходные данные**: определитель *sum*

//*nrows* - количество строк массива A в памяти процесса

//*a*[*nrows* \* *n*] - часть матрицы A, хранящаяся в данном процессе

//*rows*[*nrows*] - номера строк, хранящихся в данном процессе

//прямой ход метода Гаусса

*row* <- 0

**for** *i* <- 0 **to** *n*-1 **do**

**if** *i* = *rows*[*row*]

MPI\_Bcast(&*a*[*row* \* *n*], *n*, MPI\_DOUBLE, *rank*, MPI\_COMM\_WORLD

**for** *j* <- *0* **to** *n* **do**

*tmp*[*j*] <- *a*[*row* \* *n* + *j*]

*row* <- *row* + 1

**else**

MPI\_Bcast(*tmp*, *n*, MPI\_DOUBLE, *i* % *commsize*, MPI\_COMM\_WORLD)

**for** *j* <- *row* **to** *nrows* **do**

*scaling* <- *a*[*j* \* *n* + *i*] / *tmp*[*i*]

**for** *k* <- *i* **to** *n* **do**

*a*[*j* \* *n* + *k*] <- *a*[*j* \* *n* + *k*] - *scaling* \* *tmp*[*k*]

//вычисление определителя треугольной матрицы

*sum* <- 1

**for** *i* <- 0 **to** *n*rows **do**

*sum* <- *sum* \* *a*[*i* \* *n* + *rows*[*i*]]

*gsum* <- 1

MPI\_Reduce(&*sum*, &*gsum*, 1, MPI\_DOUBLE, MPI\_PROD, 0, MPI\_COMM\_WORLD)

MPI\_Finalize()

**return** gsum

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Программа выполнялась на кластере Jet (18 узлов)

Таблица 1 – характеристики системы

|  |  |
| --- | --- |
| Системная плата | Intel S5000VSA (Серверная платформа Intel SR2520SAF) |
| Процессор | 2 x Intel Xeon E5420 (2,5 GHz; Intel-64) |
| Оперативная память | 8 GB (4 x 2GB PC-5300) |
| Жесткий диск | SATAII 500GB (Seagate 500Gb Barracuda) |
| Сетевая карта | 2 x Intel Gigabit Ethernet (Integrated Intel PRO/1000 EB, 80003ES2LAN Gigabit Ethernet Controller) 1 x Intel PRO/1000 MT Server Adapter (PWLA8490MT, 82572EI Gigabit Ethernet Controller) |

Библиотека MPICH версии 3.2.1

Версия операционной системы: Linux Fedora 20 x86\_64

Версия компилятора: gcc 7.0

В качестве показателя эффективности используется коэффициент ускорения.

T(*n*) - время выполнения последовательной программы.

(*n*) - время выполнения главного процесса параллельной программы при *p* процессах.

Таблица 2 – результаты эксперимента при размере матрицы 300х300

|  |  |  |
| --- | --- | --- |
| Время работы | Количество процессов | Ускорение |
| 0,07169 | 1 | 1 |
| 0,03763 | 2 | 1,90 |
| 0,02107 | 4 | 3,40 |
| 0,01507 | 8 | 4,75 |
| 0,03636 | 16 | 1,97 |
| 0,04162 | 32 | 1,72 |
| 0,03167 | 64 | 2,26 |

Таблица 3 – результаты экспериментов при размере матрицы 3000х3000

|  |  |  |
| --- | --- | --- |
| Время работы | Количество процессов | Ускорение |
| 120,91 | 1 | 1 |
| 92,114 | 2 | 1,31 |
| 62,762 | 4 | 1,92 |
| 27,938 | 8 | 4,32 |
| 10,357 | 16 | 11,67 |
| 5,4353 | 32 | 22,24 |
| 5,5291 | 64 | 21,86 |

# ЗАКЛЮЧЕНИЕ

В рамках данного курсового проекта была разработана параллельная MPI-программа для вычисления определителя матрицы методом Гаусса. Результаты экспериментов показали, что программа хорошо масштабируется до определенного количества процессов, зависящего от размеров исходной матрицы. Это связанно с тем, что на каждой итерации основного цикла прямого хода метода Гаусса, один из процессов, с помощью функции MPI\_Bcast, пересылает строку матрицы всем процессам. Пересылка строки между процессами, находящимися на разных вычислительных узлах, занимает больше времени, чем между процессами на одном узле и для различных матриц, существуют свои оптимальные размеры, при превышении которых ускорение падает.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Методы вычисления определителя [Электронный ресурс]// http://mathhelpplanet.com/static.php?p=metody-vychisleniya-opredelitelyei
2. MPI A Message-Passing Interface Standard[Электронный ресурс]//https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
3. MPICH [Электронный ресурс]//https://www.mpich.org/

# ПРИЛОЖЕНИЕ

# 1. Исходный код программы

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <mpi.h>

#include <time.h>

int get\_chunk(int total, int commsize, int rank){

int n = total;

int q = n / commsize;

if (n % commsize)

q++;

int r = commsize \* q - n;

int chunk = q;

if (rank >= commsize - r)

chunk = q - 1;

return chunk;

}

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

int n = 300;

int rank, commsize;

double t;

MPI\_Init(&argc, &argv);

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

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

if (rank == 0) {

t = MPI\_Wtime();

}

int nrows = get\_chunk(n, commsize, rank);

int \*rows = (int \*) malloc(sizeof(\*rows) \* nrows);

double \*a = (double \*) malloc(sizeof(\*a) \* nrows \* n);

double \*tmp = (double \*) malloc(sizeof(\*tmp) \* n);

for (int i = 0; i < nrows; i++) {

rows[i] = rank + commsize \* i;

srand(rows[i] \* n);

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

a[i \* n + j] = rand() % 100 + 1;

}

int row = 0;

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

if (i == rows[row]) {

MPI\_Bcast(&a[row \* n], n, MPI\_DOUBLE, rank, MPI\_COMM\_WORLD);

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

tmp[j] = a[row \* n + j];

row++;

}

else {

MPI\_Bcast(tmp, n, MPI\_DOUBLE, i % commsize, MPI\_COMM\_WORLD);

}

for (int j = row; j < nrows; j++) {

double scaling = a[j \* n + i] / tmp[i];

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

a[j \* n + k] -= scaling \* tmp[k];

}

}

double sum = 1;

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

sum \*= a[i \* n + rows[i]];

double gsum = 1;

MPI\_Reduce(&sum, &gsum, 1, MPI\_DOUBLE, MPI\_PROD, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

t = MPI\_Wtime() - t;

printf("Determinant (MPI): n %d, procs %d, time (sec) %.6f tglob = %f\n", n, commsize, t, gsum);

}

free(tmp);

free(rows);

free(a);

MPI\_Finalize();

return 0;

}