**Содержание**

[1 Алгоритм для параллельного решения СЛАУ с трехдиагональными матрицами 2](#_Toc146829445)

[2 Реализация 6](#_Toc146829446)

[2.1 С & MPI 6](#_Toc146829447)

[2.2 Python & MPI 8](#_Toc146829448)

[2.3 С & OpenMP 9](#_Toc146829449)

[2.4 С & Linux pthreads 10](#_Toc146829450)

[3 Тестирование 11](#_Toc146829451)

[4 Исследование 12](#_Toc146829452)

[4.1 Зависимость времени от размерности 12](#_Toc146829453)

[4.2 Зависимость времени С & MPI от числа процессов 12](#_Toc146829454)

[4.3 Зависимость времени Python & MPI от числа процессов 12](#_Toc146829455)

[4.4 Зависимость времени C & OpenMP от числа потоков 13](#_Toc146829456)

[4.5 Зависимость времени C & Linux pthreads от числа потоков 13](#_Toc146829457)

[4.6 Зависимость времени C (Python) & MPI от числа узлов 14](#_Toc146829458)

[Результаты работы 15](#_Toc146829459)

[Приложение 1 16](#_Toc146829460)

[Приложение 2 22](#_Toc146829461)

[Приложение 3 26](#_Toc146829462)

[Приложение 4 29](#_Toc146829463)

[Приложение 5 33](#_Toc146829464)

# **1 Алгоритм для параллельного решения СЛАУ с трехдиагональными матрицами**

Трехдиагональную матрицу можно представить в следующем виде:

Изображение выглядит как текст

Автоматически созданное описание

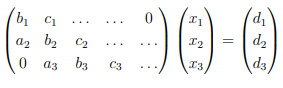
где ненулевые элементы расположены на главной диагонали и двух побочных кодиагоналях.

Базовым нахождением решения СЛАУ с трехдиагональной матрицы является: обнуление с помощью прямого хода сначала нижней кодиагонали, а потом обратным ходом обнуление верхней кодиагонали. После чего матрица системы примет диагональный вид, и неизвестные можно вычислить делением соответвутсвующего числа правой части на коэффициент стоящий при неизвестном.

Вариантом распраллеливания именно этих вычислений является: прохождение одновременно по нижней кодиагонали вниз с первого элемента, а по верхней кодиагонали вверх с нижнего элемента, в середине матрицы процессы обмениваются информацией и двигаются дальше. Однако этот алгоритм можно использовать только для двух процессов, что дает малый прирост в скорости.

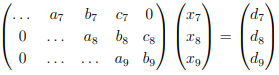
Существует способ модификации нахождения решения СЛАУ с трехдиагональной матрицей, который можно распараллелить на любое количестов процессов. Он состоит в следующем:

1. Пусть n = 9 (длина массивов а, b, c), а число процессов = 3. Тогда элементы трехдиагональной матрицы распределятся следующим образом по данным процессам:

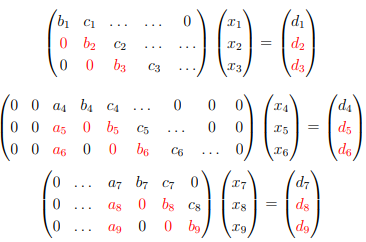


Изображение выглядит как стол

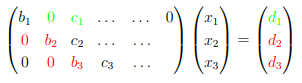
Автоматически созданное описание

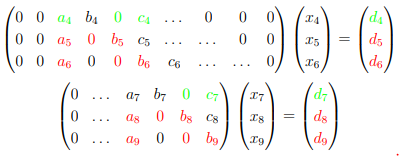


1. На каждом процессе происходит обнуление нижней кодиагонали с первого элемента на этом процессе, для чего на каждом процессе из первой строки вычитается нулевая с соответсвующим коэффициентом. После полного прохождения всеми получаем новые значения коэффициентов – отмеченных красным цветом:



1. Далее происходит обнуление элементов выше главной диагонали, начиная с предпредпоследнего, что приводит к следующему изменению коэффициентов – отмеченных зеленым цветом:

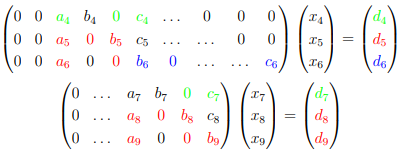




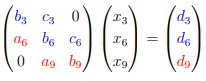
1. Теперь необходимо обнулить последние элементы верхней кодиагонали (т. е. c3 и c6), но проблема заключается в том, что для этого необходимо из строчки находящейся на нулевом процессе вычесть строчку первого процесса, а из строчки, находящейся на первом процессе, необходимо вычесть строчку, находящуюся на втором процессе (в общем случае необходимо вычитать из последней строки i-го процесса строку (i+1)-го процесса). Следовательно, необходимо произвести обмен данными. И после этого обмена проводится операция, приводящая к изменению следующих коэффициентов – отмеченных синим цветом:

Изображение выглядит как стол

Автоматически созданное описание



1. После этих действий, которые повлекли обмен маленькими сообщениями между соседними процессами, берутся элементы, находящиеся в последней строке блока данных каждого процесса, и собираются на нулевом процессе, где последовательно решается маленькая система:



Таким образом на нулевом процессе находится часть итогового вектора, в рассматриваемом примере это: x3, x6, x9.

1. Далее каждое из полученных значений передается соответствующему процессу (в рассматриваемом примере: x3 передается первому процессу, x6 второму, x9 третьему) и на каждом процессе находятся оставшиеся компоненты вектора решений (то есть на первом процессе находится x1 и x2; на втором x4 и x5; на третьем x7 и x8).
2. Далее полученные на каждом процессе кусочки вектора x объединяются и получается вектор решения СЛАУ с трехдиагональной матрицей.

# **2 Реализация**

## **2.1 С & MPI**

В первую очередь программная реализация была создана на языке C с использованием технологии MPI.

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу.

Были использованы следующие функции:

MPI\_Scatter() для рапределения элементов матрицы и вектора b по разным процессам;

MPI\_Scatterv() для распределения части значений вектора x, находимых на пятом шаге описанного алгоритма;

MPI\_Gather() для объединения частей вектора x в единое целое на нулевом узле;

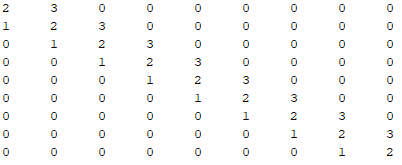
MPI\_Send() для отправки сообщений от нулевого процесса первому;

MPI\_Sendrecv() для отправки и получения сообщений всеми процессами, кроме нулевого и последнего;

MPI\_Recv() для получения сообщений последним процессом.

Количество процессов вводится при запуске программы.

Пример запуска и работы созданной реализации на C & MPI c матрицей, имеющей вид:



где N = 9 и вектор значений правой части СЛАУ:

b = [4; 4; 4; 4; 4; 4; 4; 4; 4].

Распараллеливание происходит на три процесса.





Изображение выглядит как текст

Автоматически созданное описание

Найденное решение x = [-598; 400; -66; -88; 82; -24; -10; 16; -6].

Для небольшой размерности матрицы программа выводит вектор решений x и его части, найденные на различных процессах.

Для больших размерностей значения вектора решений записываются в файл C\_MPI\_val.txt и выводится сообщение о записи:

Изображение выглядит как текст

Автоматически созданное описание

## **2.2 Python & MPI**

Для реализации программы на Python & MPI использовались те же функции MPI, что и для реализации С & MPI. Для запуска программы на языке Python создавалось виртуальное окружение exx:

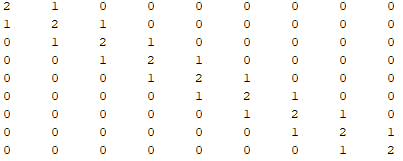
Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Представленный пример демонстрирует нахождение решения СЛАУ с трехдиагональной матрицей, имеющей вид:



где N = 9 и b = [1; 1; 1; 1; 1; 1; 1; 1; 1].

Найденное решение x = [0.5; 0; 0.5; 0; 0.5; 0; 0.5; 0; 0.5].

Распараллеливание происходит на три процесса.

При большой размерности матрицы найденное решение записывается в файл Python\_MPI\_val.txt:



## **2.3 С & OpenMP**

Следующей технологией, используемой для реализации параллельной программы, является OpenMP.

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой ведущий поток создаёт набор ведомых потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами.

Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — «прагм»:

#pragma omp parallel for shared() private() default().

Количество потоков, на которое происходит расспараллеливание может быть задано функцией:

omp\_set\_num\_threads(numthreads).

Для получения номера текущего потока используется функция:

omp\_get\_thread\_num().

Пример запуска и работы программы С & OpenMP:



Изображение выглядит как текст

Автоматически созданное описание

## **2.4 С & Linux pthreads**

Последней технологией, используемой для распараллеливания, является Pthreads.

Pthreads определяет набор типов и функций на Си. Для создания реализации использовались:

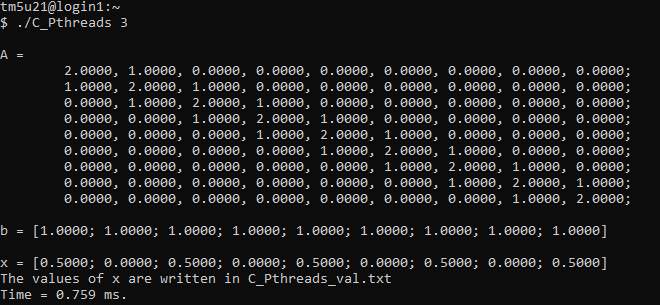
pthread\_create() – функция для создания нового потока;

pthread\_join() – функция, которая ожидает завершения потока и позволяет синхронизировать потоки;

pthread\_t — идентификатор потока.

Пример запуска и работы программы С & OpenMP:





# **3 Тестирование**

Получаемые значения вектора x с использованием различных технологий записываются в четыре текстовых файла, соответствующих данным технологиям и языкам:

* C\_MPI.txt
* Python\_MPI.txt
* C\_OpenMP.txt
* C\_Pthreads.txt

С помощью пакета MatLab вычиляется теоретическое значение вектора x и записывается в текстовый файл expected\_val.txt:

Изображение выглядит как текст

Автоматически созданное описание

Для тестирования полученных значений с использованием различных технологий был написан алгоритм, сравнивающий значения из файлов С\_MPI\_val.txt; Python\_MPI\_val.txt; C\_OpenMP\_val.txt; C\_Pthreads\_val.txt со значениями из файла expected\_val.txt.

При совпадении значений алгоритм выводит сообщение о прохождении теста: «COMPLETED», при несовпадении выводит сообщение о непрохождении теста: «FAILD».

Пример № 1 запуска, тестирующего алгоритма:

Изображение выглядит как текст

Автоматически созданное описание

Пример № 2 запуска, тестирующего алгоритма:

Изображение выглядит как текст

Автоматически созданное описание

В первом примере тест прошел только для Python & MPI, а во втором для всех реализаций.

# **4 Исследование**

## **4.1 Зависимость времени от размерности**

Было проведено исследование зависимости времени от размерности матрицы. Исследование проводилось при N = 10; N = 100 и N = 1000 для всех реализаций.

Результаты представлены в следующей таблице (время указано в сек.):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | С & MPI | Python & MPI | C & OpenMP | C & Linux pthreads |
| 10 | 0.00098 | 0.00849 | 0.000058 | 0.000908 |
| 100 | 0.000446 | 0.01839 | 0.000003 | 0.011025 |
| 1000 | 0.000651 | 0.03081 | 0.001079 | 0.458659 |

Можно сделать вывод о том, при большой размерности матрицы быстрее работает реализация на С & MPI.

## **4.2 Зависимость времени С & MPI от числа процессов**

В следующей таблице представлено время в секундах для 2; 5; 10; 20; 25 процессов при N = 1000; N = 10000 и N = 100000:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **2** | **5** | **10** | **20** | **25** |
| **1000** | 0.000339 | 0.00065 | 0.000792 | 0.001523 | 0.00141 |
| **10000** | 0.000882 | 0.001209 | 0.001128 | 0.001193 | 0.002005 |
| **100000** | 0.006337 | 0.004671 | 0.004079 | 0.00445 | 0.005077 |

По полученным данным можно сделать вывод о том, что для больших размерностей матриц (10000 и 100000) время уменьшается при увеличении числа процессов от 2 до 10.

## **4.3 Зависимость времени Python & MPI от числа процессов**

В следующей таблице представлено время в секундах для 2; 5; 10; 20; 25 процессов при N = 1000; N = 10000 и N = 100000:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **2** | **5** | **10** | **20** | **25** |
| **1000** | 0.008142 | 0.002603 | 0.001924 | 0.002162 | 0.036117 |
| **10000** | 0.053527 | 0.030916 | 0.029435 | 0.011512 | 0.009583 |
| **100000** | 0.521677 | 0.193474 | 0.108299 | 0.072087 | 0.04812 |

По полученным данным можно сделать вывод о том, что для больших размерностей матриц (10000 и 100000) время уменьшается при увеличении числа процессов от 2 до 25. Для матрицы размерностью 100000 увеличение процессов в два раза ведет к уменьшению времени почти в наполовину.

## **4.4 Зависимость времени C & OpenMP от числа потоков**

В следующей таблице представлено время в секундах для 2; 5; 10; 20; 25 процессов при N = 100; N = 1000 и N = 2000:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **2** | **5** | **10** | **20** | **25** |
| **100** | 0.0000029 | 0.0000029 | 0.0000028 | 0.0000027 | 0.0000026 |
| **1000** | 0.0010646 | 0.0012701 | 0.0011242 | 0.0011195 | 0.0010597 |
| **2000** | 0.0068917 | 0.0068873 | 0.0068829 | 0.0069258 | 0.0069489 |

По полученным данным можно сделать вывод о том, что для матрицы размерности 100 и 1000 увеличение потоков от 5 до 25 ведет к уменьшению времени работы.

## **4.5 Зависимость времени C & Linux pthreads от числа потоков**

В следующей таблице представлено время в секундах для 2; 5; 10; 20; 25 процессов при N = 100; N = 1000 и N = 2000:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **2** | **5** | **10** | **20** | **25** |
| **100** | 0.007363 | 0.010849 | 0.016837 | 0.03561 | 0.059188 |
| **1000** | 0.829966 | 0.447354 | 0.304066 | 0.324159 | 0.43022 |
| **2000** | 7.09444 | 2.97461 | 1.5847 | 1.08264 | 1.07958 |

По полученным данным можно сделать вывод о том, что для больших матриц (N = 2000) с увеличением числа потоков время уменьшается.

## **4.6 Зависимость времени C (Python) & MPI от числа узлов**

Запуск реализаций на нескольких узлах производился с помощью sbatch:

Изображение выглядит как текст

Автоматически созданное описание

В следующей таблице представлено время в секундах для C & MPI и Python & MPI при числе узлов от единицы до четырех:

|  |  |  |
| --- | --- | --- |
| Число узлов | С & MPI | Python & MPI |
| 1 | 0.000399 | 0.000761 |
| 2 | 0.000378 | 0.000793 |
| 3 | 0.000415 | 0.000704 |
| 4 | 0.000363 | 0.000743 |

# **Результаты работы**

В ходе проделанной работы:

1. был разработан параллельный алгоритм для решения СЛАУ (систем линейных уравнений) с трехдиагональными матрицами;
2. были созданы реализации параллельного решения СЛАУ с трехдиагональными матрицами с использованием языков С/Python и технологий MPI/OpenMP/Linux pthreads;
3. был разработан алгоритм для тестирования значений, получаемых при различных реализациях с теоретическими значениями, посчитанными с помощью MATLAB;
4. были проведены исследования получаемых результатов путем использования различных реализаций.

# **Приложение 1**

Реализация С & MPI

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/times.h>

#include <sys/time.h>

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

{

int rank, numprocs, j, i;

double \*arrA, \*arrB, \*arrC, \*arrD, \*arrA\_part, \*arrB\_part, \*arrC\_part, \*arrD\_part;

double \*temp\_array\_send, \*temp\_array\_recv;

double \*A\_extended, \*A\_extended\_a, \*A\_extended\_b, \*A\_extended\_c, \*A\_extended\_d;

double \*x\_temp, \*x\_part\_last, \*x\_part, \*x;

struct timeval etStart, etStop;

struct timezone dummyTz;

unsigned long long startTime, endTime;

FILE\* file = NULL;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

/\*a b c d - векторы одинаковой длины, описывающие систему(a1 = 0, сN = 0) \*/

int N = 9;

/\*каждому процессу достается 4 вектора меньшей длины\*/

int N\_part = N / numprocs;

if (rank == 0)

{

arrA = (double\*)calloc(N, sizeof(double));

for(j = 0; j < N; j++) arrA[j] = 1;

arrB = (double\*)calloc(N, sizeof(double));

for(j = 0; j < N; j++) arrB[j] = 2;

arrC = (double\*)calloc(N, sizeof(double));

for(j = 0; j < N; j++) arrC[j] = 1;

arrD = (double\*)calloc(N, sizeof(double));

for(j = 0; j < N; j++) arrD[j] = 1;

}

arrA\_part = (double\*)calloc(N\_part, sizeof(double));

arrB\_part = (double\*)calloc(N\_part, sizeof(double));

arrC\_part = (double\*)calloc(N\_part, sizeof(double));

arrD\_part = (double\*)calloc(N\_part, sizeof(double));

gettimeofday(&etStart, &dummyTz);

MPI\_Scatter(arrA, N\_part, MPI\_DOUBLE, arrA\_part, N\_part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(arrB, N\_part, MPI\_DOUBLE, arrB\_part, N\_part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(arrC, N\_part, MPI\_DOUBLE, arrC\_part, N\_part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(arrD, N\_part, MPI\_DOUBLE, arrD\_part, N\_part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if (rank > 0 || rank == 0)

{

for (int n = 1; n < N\_part; n++) {

double coef = arrA\_part[n]/arrB\_part[n-1];

arrA\_part[n] = -coef\*arrA\_part[n-1];

arrB\_part[n] = arrB\_part[n] - coef\*arrC\_part[n-1];

arrD\_part[n] = arrD\_part[n] - coef\*arrD\_part[n-1];

}

for (int n = N\_part-3; n > -1; n--) {

double coef = arrC\_part[n]/arrB\_part[n+1];

arrC\_part[n] = -coef\*arrC\_part[n+1];

arrA\_part[n] = arrA\_part[n] - coef\*arrA\_part[n+1];

arrD\_part[n] = arrD\_part[n] - coef\*arrD\_part[n+1];

}

}

//все процессы кроме 0-го будут отправлять данные

if (rank > 0)

{

temp\_array\_send = (double\*)calloc(4, sizeof(double));

temp\_array\_send[0] = arrA\_part[0];

temp\_array\_send[1] = arrB\_part[0];

temp\_array\_send[2] = arrC\_part[0];

temp\_array\_send[3] = arrD\_part[0];

}

//все процессы кроме последнего будут принимать данные

if (rank < numprocs - 1)

{

temp\_array\_recv = (double\*)calloc(4, sizeof(double));

}

//0-й процесс только принимает

if (rank == 0)

{

MPI\_Recv(temp\_array\_recv, 4, MPI\_DOUBLE, 1, 0, MPI\_COMM\_WORLD, &status);

}

//все процессы кроме нулевого и отправляют и получают сообщения

if (rank > 0 && rank < numprocs-1)

{

MPI\_Sendrecv(temp\_array\_send, 4, MPI\_DOUBLE, rank-1, 0,

temp\_array\_recv, 4, MPI\_DOUBLE, rank+1, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

}

//последниц процесс только отправляет

if (rank == numprocs-1)

{

MPI\_Send(temp\_array\_send, 4, MPI\_DOUBLE, numprocs-2, 0, MPI\_COMM\_WORLD);

}

if (rank < numprocs-1)

{

double coef = arrC\_part[N\_part-1]/temp\_array\_recv[1];

arrB\_part[N\_part-1] = arrB\_part[N\_part-1] - coef\*temp\_array\_recv[0];

arrC\_part[N\_part-1] = -coef\*temp\_array\_recv[2];

arrD\_part[N\_part-1] = arrD\_part[N\_part-1] - coef\*temp\_array\_recv[3];

}

//Массив на отправку: с каждого процесса последние элементы матрицы

if (rank > 0 || rank == 0)

{

temp\_array\_send = (double\*)calloc(4, sizeof(double));

temp\_array\_send[0] = arrA\_part[N\_part-1];

temp\_array\_send[1] = arrB\_part[N\_part-1];

temp\_array\_send[2] = arrC\_part[N\_part-1];

temp\_array\_send[3] = arrD\_part[N\_part-1];

}

//На нулевом процессе подготавливается рсширенная матрица системы

if (rank == 0)

{

A\_extended = (double\*)calloc(numprocs\*4, sizeof(double));

}

//Собираем столбцы в расширенную матрицу системы

MPI\_Gather(temp\_array\_send, 4, MPI\_DOUBLE, A\_extended, 4, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

x\_temp = (double\*)calloc(numprocs, sizeof(double));

//double x\_temp[numprocs];

A\_extended\_a = (double\*)calloc(numprocs, sizeof(double));

A\_extended\_b = (double\*)calloc(numprocs, sizeof(double));

A\_extended\_c = (double\*)calloc(numprocs, sizeof(double));

A\_extended\_d = (double\*)calloc(numprocs, sizeof(double));

if (rank == 0)

{

i = 0;

for(j = 0; j < numprocs\*4; j = j+4){

A\_extended\_a[i] = A\_extended[j];

i++;

}

i = 0;

for(j = 1; j < numprocs\*4; j = j+4){

A\_extended\_b[i] = A\_extended[j];

i++;

}

i = 0;

for(j = 2; j < numprocs\*4; j = j+4){

A\_extended\_c[i] = A\_extended[j];

i++;

}

i = 0;

for(j = 3; j < numprocs\*4; j = j+4){

A\_extended\_d[i] = A\_extended[j];

i++;

}

}

//Прямой ход метода Гуасса для нахождения вектора x\_temp (размер которого = количеству процессов)

if (rank == 0)

{

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

x\_temp[i] = 0;

}

//Обнуляем все элементы, лежащие ниже главной диагонали - прямой метод Гаусса

for (int n = 1; n < numprocs; n++){

double coef = A\_extended\_a[n]/A\_extended\_b[n-1];

A\_extended\_b[n] = A\_extended\_b[n] - coef\*A\_extended\_c[n-1];

A\_extended\_d[n] = A\_extended\_d[n] - coef\*A\_extended\_d[n-1];

}

//Обнуляем все элементы, лежащие выше главной диагонали - обратный ход Гаусса

for (int n = numprocs-2; n > -1; n--){

double coef = A\_extended\_c[n]/A\_extended\_b[n+1];

A\_extended\_d[n] = A\_extended\_d[n] - coef\*A\_extended\_d[n+1];

}

//Нахождение решения

for (int n = 0; n < numprocs; n++){

x\_temp[n] = A\_extended\_d[n]/A\_extended\_b[n];

}

}

int rcounts\_temp[numprocs];

int displs\_temp[numprocs];

//Подготовка массивов для работы со сдвигом в функции Scatter

if (rank == 0) {

rcounts\_temp[0] = 1;

displs\_temp[0] = 0;

for (int k = 1; k < numprocs; k++){

rcounts\_temp[k] = 2;

displs\_temp[k] = k - 1;

}

}

//Записываем в x\_part\_last нужное количество элементов для каждого процесса из x\_temp

if (rank == 0) {

x\_part\_last = (double\*)calloc(1, sizeof(double));

MPI\_Scatterv(x\_temp, rcounts\_temp, displs\_temp, MPI\_DOUBLE,

x\_part\_last, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

else{

x\_part\_last = (double\*)calloc(2, sizeof(double));

MPI\_Scatterv(x\_temp, rcounts\_temp, displs\_temp, MPI\_DOUBLE,

x\_part\_last, 2, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

//Создание конечных кусков массива x на каждом процессе

x\_part = (double\*)calloc(N\_part, sizeof(double));

if (rank == 0) {

if (N <= 20)

printf("Для процесса № %d x\_part = [ ", rank);

for (int n = 0; n < N\_part-1; n++){

x\_part[n] = (arrD\_part[n] - arrC\_part[n]\*x\_part\_last[0])/arrB\_part[n];

x\_part[N\_part-1] = x\_part\_last[0];

if (N <= 20)

printf("%0.4f; ", x\_part[n]);

}

if (N <= 20)

printf("%0.4f ]\n\n", x\_part[N\_part-1]);

}

else {

if (N <= 20)

printf("Для процесса № %d x\_part = [ ", rank);

for (int n = 0; n < N\_part-1; n++){

x\_part[n] = (arrD\_part[n] - arrA\_part[n]\*x\_part\_last[0] - arrC\_part[n]\*x\_part\_last[1])/arrB\_part[n];

x\_part[N\_part-1] = x\_part\_last[1];

if (N <= 20)

printf("%0.4f; ", x\_part[n]);

}

if (N <= 20)

printf("%0.4f ]\n\n", x\_part[N\_part-1]);

}

//Собираем все части вектора решений на 0-м процессе

x = (double\*)calloc(N, sizeof(double));

MPI\_Gather(x\_part, N\_part, MPI\_DOUBLE, x, N\_part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

gettimeofday(&etStop, &dummyTz);

if (rank == 0) {

if(N < 20){

printf("\nМассив решений x = [ ");

printf("%0.4f", x[0]);

for (int n = 1; n < N; n++)

{

printf("; %0.4f", x[n]);

}

printf(" ]\n\n");

}

//Запись в файл

file = fopen("C\_MPI\_val.txt", "w+b");

if (file == NULL) {

printf("Error in opening file... \n");

exit(-1);

}

fprintf(file, "%0.4f", x[0]);

for (int n = 1; n < N; n++) {

fprintf(file, "\n%0.4f", x[n]);

}

fclose(file);

printf('Значения вектора решений x записаны в файл C\_MPI\_val.txt');

startTime = (unsigned long long)etStart.tv\_sec \* 1000000 + etStart.tv\_usec;

endTime = (unsigned long long)etStop.tv\_sec \* 1000000 + etStop.tv\_usec;

/\* Выводим время работы параллельной части\*/

printf("\nTime = %g ms.\n",(float)(endTime - startTime)/(float)1000);

}

MPI\_Finalize();

return 0;

}

# **Приложение 2**

Реализация Python & MPI

from mpi4py import MPI

from numpy import empty, array, int32, float64, dot

import time

comm = MPI.COMM\_WORLD

rank = comm.Get\_rank()

numprocs = comm.Get\_size()

N = 9

N\_part = int(N / numprocs)

if rank == 0 :

arrA = empty(N, dtype=float64)

for j in range(0, N) : arrA[j] = 1

arrB = empty(N, dtype=float64)

for j in range(0, N) : arrB[j] = 2

arrC = empty(N, dtype=float64)

for j in range(0, N) : arrC[j] = 1

arrD = empty(N, dtype=float64)

for j in range(0, N) : arrD[j] = 1

else :

arrA = None

arrB = None

arrC = None

arrD = None

a\_part = empty(N\_part, dtype=float64);

b\_part = empty(N\_part, dtype=float64);

c\_part = empty(N\_part, dtype=float64);

d\_part = empty(N\_part, dtype=float64);

start = time.time()

comm.Scatter([arrA, N\_part, MPI.DOUBLE], [a\_part, N\_part, MPI.DOUBLE], root=0)

comm.Scatter([arrB, N\_part, MPI.DOUBLE], [b\_part, N\_part, MPI.DOUBLE], root=0)

comm.Scatter([arrC, N\_part, MPI.DOUBLE], [c\_part, N\_part, MPI.DOUBLE], root=0)

comm.Scatter([arrD, N\_part, MPI.DOUBLE], [d\_part, N\_part, MPI.DOUBLE], root=0)

for n in range(1, N\_part) :

coef = a\_part[n]/b\_part[n-1]

a\_part[n] = -coef\*a\_part[n-1]

b\_part[n] = b\_part[n] - coef\*c\_part[n-1]

d\_part[n] = d\_part[n] - coef\*d\_part[n-1]

for n in range(N\_part -3, -1, -1):

coef = c\_part[n]/b\_part[n+1]

c\_part[n] = -coef\*c\_part[n+1]

a\_part[n] = a\_part[n] - coef\*a\_part[n+1]

d\_part[n] = d\_part[n] - coef\*d\_part[n+1]

if rank > 0 :

temp\_array\_send = array([a\_part[0], b\_part[0],

c\_part[0], d\_part[0]],dtype=float64)

if rank < numprocs -1 :

temp\_array\_recv = empty(4, dtype=float64)

if rank == 0 :

comm.Recv([temp\_array\_recv , 4, MPI.DOUBLE], source=1,

tag=0, status=None)

if rank in range(1, numprocs -1) :

comm.Sendrecv(sendbuf=[temp\_array\_send , 4, MPI.DOUBLE],

dest=rank -1, sendtag=0,

recvbuf=[temp\_array\_recv , 4, MPI.DOUBLE],

source=rank+1, recvtag=MPI.ANY\_TAG , status=None)

if rank == numprocs -1 :

comm.Send([temp\_array\_send , 4, MPI.DOUBLE],

dest=numprocs -2, tag=0)

if rank < numprocs -1 :

coef = c\_part[N\_part -1]/temp\_array\_recv[1]

b\_part[N\_part -1] = b\_part[N\_part -1] - coef\*temp\_array\_recv[0]

c\_part[N\_part -1] = - coef\*temp\_array\_recv[2]

d\_part[N\_part -1] = d\_part[N\_part -1] - coef\*temp\_array\_recv[3]

temp\_array\_send = array([a\_part[N\_part -1], b\_part[N\_part -1], c\_part[N\_part -1], d\_part[N\_part -1]],dtype=float64)

if rank == 0:

A\_extended = empty(numprocs\*4, dtype=float64)

else:

A\_extended = None

comm.Gather([temp\_array\_send , 4, MPI.DOUBLE], [A\_extended , 4, MPI.DOUBLE], root=0)

x\_temp = empty(numprocs, dtype=float64)

A\_extended\_a = empty(numprocs, dtype=float64)

A\_extended\_b = empty(numprocs, dtype=float64)

A\_extended\_c = empty(numprocs, dtype=float64)

A\_extended\_d = empty(numprocs, dtype=float64)

if rank == 0:

i = 0

for j in range(0, numprocs\*4, 4) : A\_extended\_a[i] = A\_extended[j]; i = i+1

i = 0

for j in range(1, numprocs\*4, 4) : A\_extended\_b[i] = A\_extended[j]; i = i+1

i = 0

for j in range(2, numprocs\*4, 4) : A\_extended\_c[i] = A\_extended[j]; i = i+1

i = 0

for j in range(3, numprocs\*4, 4) : A\_extended\_d[i] = A\_extended[j]; i = i+1

#Находим x\_temp

if rank == 0:

#Обнуляем все элементы, лежащие ниже главной диагонали - прямой метод Гаусса

for n in range(1, numprocs):

coef = A\_extended\_a[n]/A\_extended\_b[n-1]

A\_extended\_b[n] = A\_extended\_b[n] - coef\*A\_extended\_c[n-1]

A\_extended\_d[n] = A\_extended\_d[n] - coef\*A\_extended\_d[n-1]

#Обнуляем все элементы, лежащие выше главной диагонали - обратный ход метода Гаусса

for n in range(numprocs-2, -1, -1):

coef = A\_extended\_c[n]/A\_extended\_b[n+1]

A\_extended\_d[n] = A\_extended\_d[n] - coef\*A\_extended\_d[n+1]

#Нахождение решения

for n in range(numprocs): x\_temp[n] = A\_extended\_d[n]/A\_extended\_b[n]

else:

x\_temp = None

if rank == 0:

rcounts\_temp = empty(numprocs, dtype=int32)

displs\_temp = empty(numprocs, dtype=int32)

rcounts\_temp[0] = 1

displs\_temp[0] = 0

for k in range(1, numprocs) :

rcounts\_temp[k] = 2

displs\_temp[k] = k - 1

else :

rcounts\_temp = None; displs\_temp = None

if rank == 0 :

x\_part\_last = empty(1, dtype=float64)

comm.Scatterv([x\_temp, rcounts\_temp, displs\_temp, MPI.DOUBLE], [x\_part\_last, 1, MPI.DOUBLE], root = 0)

else :

x\_part\_last = empty(2, dtype=float64)

comm.Scatterv([x\_temp, rcounts\_temp, displs\_temp, MPI.DOUBLE], [x\_part\_last, 2, MPI.DOUBLE], root = 0)

x\_part = empty(N\_part , dtype=float64)

if rank == 0 :

if (N < 20) :

print('Для процесса № %d x\_part = [ ' % rank, end = '');

for n in range(N\_part-1) :

x\_part[n] = (d\_part[n] - c\_part[n]\*x\_part\_last[0])/b\_part[n]

x\_part[N\_part -1] = x\_part\_last[0]

if (N < 20):

print('%.4f; ' % x\_part[n], end = '')

if (N < 20):

print('%.4f ]\n' % x\_part[N\_part-1])

else :

if (N < 20):

print('Для процесса № %d x\_part = [ ' % rank, end = '');

for n in range(N\_part-1) :

x\_part[n] = (d\_part[n] - a\_part[n]\*x\_part\_last[0] - c\_part[n]\*x\_part\_last[1])/b\_part[n]

x\_part[N\_part -1] = x\_part\_last[1]

if (N < 20):

print('%.4f; ' % x\_part[n], end = '')

if (N < 20):

print('%.4f ]\n' % x\_part[N\_part-1])

#Собираем все части вектора решений на 0-м процессе

x = empty(N, dtype=float64)

comm.Gather([x\_part, N\_part, MPI.DOUBLE], [x, N\_part, MPI.DOUBLE], root=0)

end = time.time()

if (rank == 0) :

if (N < 20):

print('\nМассив решений x = [ ', end = '')

print('%.4f;' % x[0], end = '')

for n in range(1, N-1) : print('%.4f;' % x[n], end = '')

print('%.4f ]\n' % x[N-1])

f = open('Python\_MPI\_val.txt', 'w')

f.write(str(round(x[0], 4)))

for n in range(1, N-1):

f.write('\n' + str(round(x[n], 4)))

f.close()

print('The values of x are wtitten in Python\_MPI\_val.txt')

#время выводится в секундах

print('\nTime = ', round((end - start), 6), end = ' s.\n')

# **Приложение 3**

Реализация С & OpenMP

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/times.h>

#include <sys/time.h>

FILE\* file = NULL;

int N = 9;

int numthreads;

volatile float A[1000][1000], b[1000], x[1000];

void initializeMatrix();

void printMatrix();

void parallelOpenMP();

void printX();

void backSubstitution();

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

struct timeval etStart, etStop;

struct timezone dummyTz;

unsigned long long startTime, endTime;

numthreads = atoi(argv[1]);

//создаем трехдиагональную матрицу

initializeMatrix();

//выводим матрицу, если N<20

printMatrix();

//время до параллельного алгоритма

gettimeofday(&etStart, &dummyTz);

parallelOpenMP();

//время после параллельного алгоритма

gettimeofday(&etStop, &dummyTz);

//выводим полученные значения,если N<20 (иначе просто записываем в С\_OpenP\_val.txt)

printX();

startTime = (unsigned long long)etStart.tv\_sec \* 1000000 + etStart.tv\_usec;

endTime = (unsigned long long)etStop.tv\_sec \* 1000000 + etStop.tv\_usec;

//выводим время в милисекундах

printf("\nTime = %g ms.\n",(float)(endTime - startTime)/(float)1000);

exit(0);

}

void initializeMatrix() {

int row, col;

for (col = 0; col < N; col++) {

for (row = 0; row < N; row++) {

if (col == row-1)

A[row][col] = 1;

if (col == row)

A[row][col] = 2;

if (col == row+1)

A[row][col] = 1;

if (col != row-1 && col != row && col != row+1)

A[row][col] = 0;

}

b[col] = 1.0;

x[col] = 0.0;

}

}

void printMatrix() {

int row, col;

if (N < 20) {

printf("\nA =\n\t");

for (row = 0; row < N; row++) {

for (col = 0; col < N; col++) {

printf("%0.4f%s", A[row][col], (col < N-1) ? ", " : ";\n\t");

}

}

printf("\nb = [");

for (col = 0; col < N; col++) {

printf("%0.4f%s", b[col], (col < N-1) ? "; " : "]\n");

}

}

}

void printX() {

int n;

if (N < 20) {

printf("\nx = [");

for (n = 0; n < N; n++) {

printf("%0.4f%s", x[n], (n < N-1) ? "; " : "]\n");

}

}

//Запись в файл

file = fopen("C\_OpenMP\_val.txt", "w+b");

if (file == NULL) {

printf("Error in opening file... \n");

exit(-1);

}

fprintf(file, "%0.4f", x[0]);

for (n = 1; n < N; n++) {

fprintf(file, "\n%0.4f", x[n]);

}

fclose(file);

printf("The values of x are wtitten in C\_OpenMP\_val.txt");

}

void backSubstitution(){

int norm, row, col;

for (row = N - 1; row >= 0; row--) {

x[row] = b[row];

for (col = N-1; col > row; col--) {

x[row] -= A[row][col] \* x[col];

}

x[row] /= A[row][row];

}

}

void parallelOpenMP() {

int norm, row, col;

float multiplier;

omp\_set\_num\_threads(numthreads);

for (norm = 0; norm < N - 1; norm++) {

#pragma omp parallel for shared(A,b,norm,N) private(row,multiplier,col) default(none)

for (row = norm + 1; row < N; row++) {

/\*int tid = omp\_get\_thread\_num();

printf("Hy from thread = %d\n", tid);\*/

multiplier = A[row][norm] / A[norm][norm];

for (col = norm; col < N; col++) {

A[row][col] -= A[norm][col] \* multiplier;

}

b[row] -= b[norm] \* multiplier;

}

}

backSubstitution();

}

# **Приложение 4**

Реализация С & Linux pthreads

#include <stdio.h>

#include <stdlib.h>

#include <sys/times.h>

#include <sys/time.h>

#include <pthread.h>

FILE\* file = NULL;

int N = 9;

int numberOfProcessors, numberOfThreadsPerProcessor, totalNumberOfThreads;

volatile float A[1000][1000], b[1000], x[1000];

void initializeMatrix();

void printInputs();

void printX();

void parallelThreadP();

void parallelThreadPEnoughThreadsToProcessAllRows();

void parallelThreadPLesserThreadsToProcessAllRows();

void \*rowFactorMultiplicationWithoutSkipLogic(struct ThreadParam \* threadParam);

void \*rowFactorMultiplicationWithSkipLogic(struct ThreadParam \* threadParam);

void backSubstitution();

void gauss();

struct ThreadParam{

int threadId;

int outerRow;

};

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

struct timeval etStart, etStop;

struct timezone dummyTz;

unsigned long long startTime, endTime;

numberOfThreadsPerProcessor = atoi(argv[1]);

numberOfProcessors = 1;

initializeMatrix();

printMatrix();

gettimeofday(&etStart, &dummyTz);

parallelThreadP();

gettimeofday(&etStop, &dummyTz);

printX();

startTime = (unsigned long long)etStart.tv\_sec \* 1000000 + etStart.tv\_usec;

endTime = (unsigned long long)etStop.tv\_sec \* 1000000 + etStop.tv\_usec;

printf("\nTime = %g ms.\n",(float)(endTime - startTime)/(float)1000);

exit(0);

}

void initializeMatrix() {

int row, col;

for (col = 0; col < N; col++) {

for (row = 0; row < N; row++) {

if (col == row-1)

A[row][col] = 1;

if (col == row)

A[row][col] = 2;

if (col == row+1)

A[row][col] = 1;

if (col != row-1 && col != row && col != row+1)

A[row][col] = 0;

}

b[col] = 1.0;

x[col] = 0.0;

}

}

void printMatrix() {

int row, col;

if (N < 20) {

printf("\nA =\n\t");

for (row = 0; row < N; row++) {

for (col = 0; col < N; col++) {

printf("%0.4f%s", A[row][col], (col < N-1) ? ", " : ";\n\t");

}

}

printf("\nb = [");

for (col = 0; col < N; col++) {

printf("%0.4f%s", b[col], (col < N-1) ? "; " : "]\n");

}

}

}

void printX() {

int n;

if (N < 20) {

printf("\nx = [");

for (n = 0; n < N; n++) {

printf("%0.4f%s", x[n], (n < N-1) ? "; " : "]\n");

}

}

//Запись в файл

file = fopen("C\_Pthreads\_val.txt", "w+b");

if (file == NULL) {

printf("Error in opening file... \n");

exit(-1);

}

fprintf(file, "%0.4f", x[0]);

for (n = 1; n < N; n++) {

fprintf(file, "\n%0.4f", x[n]);

}

fclose(file);

printf("The values of x are wtitten in C\_Pthreads\_val.txt");

}

void parallelThreadP(){

int norm, row, col;

float multiplier;

totalNumberOfThreads = ((N-1)>(numberOfProcessors\*numberOfThreadsPerProcessor))?(numberOfProcessors\*numberOfThreadsPerProcessor):(N-1);

if(totalNumberOfThreads == N-1){

parallelThreadPEnoughThreadsToProcessAllRows();

}else{

parallelThreadPLesserThreadsToProcessAllRows();

}

backSubstitution();

}

void parallelThreadPEnoughThreadsToProcessAllRows(){

pthread\_t pthreads[totalNumberOfThreads];

struct ThreadParam\* param = malloc(totalNumberOfThreads\* sizeof(struct ThreadParam));

int outerRow;

for(outerRow = 0 ; outerRow < N-1; outerRow++){

int threadIndex;

for(threadIndex = 0 ;threadIndex < totalNumberOfThreads; threadIndex++){

param[threadIndex].threadId = threadIndex;

param[threadIndex].outerRow = outerRow;

pthread\_create(&pthreads[threadIndex],NULL,rowFactorMultiplicationWithoutSkipLogic,&param[threadIndex]);

}

for(threadIndex = 0 ;threadIndex < totalNumberOfThreads; threadIndex++){

pthread\_join(pthreads[threadIndex],NULL);

}

}

free(param);

}

void parallelThreadPLesserThreadsToProcessAllRows(){

pthread\_t pthreads[totalNumberOfThreads];

struct ThreadParam \*param = malloc(totalNumberOfThreads\* sizeof(struct ThreadParam));

int outerRow;

for(outerRow = 0 ; outerRow < N-1; outerRow++){

int threadIndex;

for(threadIndex = 0 ;threadIndex < totalNumberOfThreads; threadIndex++){

param[threadIndex].threadId = threadIndex;

param[threadIndex].outerRow = outerRow;

pthread\_create(&pthreads[threadIndex],NULL,rowFactorMultiplicationWithSkipLogic,&param[threadIndex]);

}

for(threadIndex = 0 ;threadIndex < totalNumberOfThreads; threadIndex++){

pthread\_join(pthreads[threadIndex],NULL);

}

}

free(param);

}

void \*rowFactorMultiplicationWithoutSkipLogic(struct ThreadParam \* threadParam){

int col;

float multiplier;

int outerRow = threadParam->outerRow;

int innerRow = threadParam->threadId+outerRow+1;

if(innerRow < N) {

multiplier = A[innerRow][outerRow] / A[outerRow][outerRow];

for (col = outerRow; col < N; col++) {

A[innerRow][col] -= A[outerRow][col] \* multiplier;

}

b[innerRow] -= b[outerRow] \* multiplier;

}

}

void \*rowFactorMultiplicationWithSkipLogic(struct ThreadParam \* threadParam){

int innerRow, col;

float multiplier;

int startIndex = threadParam->threadId+1;

int outerRow = threadParam->outerRow;

for (innerRow = startIndex+outerRow; innerRow < N; innerRow+=totalNumberOfThreads) {

multiplier = A[innerRow][outerRow] / A[outerRow][outerRow];

for (col = outerRow; col < N; col++) {

A[innerRow][col] -= A[outerRow][col] \* multiplier;

}

b[innerRow] -= b[outerRow] \* multiplier;

}

}

void backSubstitution(){

int norm, row, col;

for (row = N - 1; row >= 0; row--) {

x[row] = b[row];

for (col = N-1; col > row; col--) {

x[row] -= A[row][col] \* x[col];

}

x[row] /= A[row][row];

}

}

# **Приложение 5**

Тесты

import numpy as np

array1 = []

array2 = []

array3 = []

array4 = []

array5 = []

with open("expected\_val.txt") as f1:

size = f1.readline()

array1.append([x for x in f1.readline().split()])

with open("C\_MPI\_val.txt") as f2:

size = f2.readline()

array2.append([x for x in f2.readline().split()])

if np.array\_equal(array1, array2):

print("C\_MPI\_TEST COMPLETED")

else:

print("C\_MPI\_TEST FAILED")

with open("Python\_MPI\_val.txt") as f3:

size = f3.readline()

array3.append([x for x in f3.readline().split()])

if np.array\_equal(array1, array3):

print("Python\_MPI\_TEST COMPLETED")

else:

print("Python\_MPI\_TEST FAILED")

with open("C\_OpenMP\_val.txt") as f4:

size = f4.readline()

array4.append([x for x in f4.readline().split()])

if np.array\_equal(array1, array4):

print("C\_OpenMP\_TEST COMPLETED")

else:

print("C\_OpenMP\_TEST FAILED")

with open("C\_Pthreads\_val.txt") as f5:

size = f5.readline()

array5.append([x for x in f5.readline().split()])

if np.array\_equal(array1, array5):

print("C\_Pthreads\_TEST COMPLETED")

else:

print("C\_Pthreads\_TEST FAILED")