Содержание

1 Алгоритм для параллельного решения СЛАУ с трехдиагональными матрицами

Трехдиагональную матрицу можно представить в следующем виде:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & \dots & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \dots \\ d_{n-1} \\ d_n \end{pmatrix}$$

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 a, b, c), а число процессов = 3. Тогда элементы трехдиагональной матрицы распределятся следующим образом по данным процессам:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & \dots & \dots \\ 0 & a_3 & b_3 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & a_6 & b_6 & c_6 & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} \dots & a_7 & b_7 & c_7 & 0 \\ 0 & \dots & a_8 & b_8 & c_8 \\ 0 & \dots & \dots & a_9 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

2) 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:

$$\begin{pmatrix} b_1 & c_1 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & \dots & \dots \\ 0 & 0 & b_3 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & c_6 & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & c_7 & 0 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

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

$$\begin{pmatrix} b_1 & 0 & c_1 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & \dots & \dots & \\ 0 & 0 & b_3 & c_3 & \dots & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & 0 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & c_6 & \dots & \dots & 0 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & 0 & c_7 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

4) 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:

$$\begin{pmatrix} b_1 & 0 & c_1 & 0 & \dots & \dots & 0 \\ 0 & b_2 & c_2 & 0 & \dots & \dots & \\ 0 & 0 & b_3 & 0 & 0 & c_3 & \dots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & a_4 & b_4 & 0 & c_4 & \dots & 0 & 0 & 0 \\ 0 & 0 & a_5 & 0 & b_5 & c_5 & \dots & \dots & 0 & 0 \\ 0 & 0 & a_6 & 0 & 0 & b_6 & 0 & \dots & \dots & c_6 \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} d_4 \\ d_5 \\ d_6 \end{pmatrix}$$

$$\begin{pmatrix} 0 & \dots & a_7 & b_7 & 0 & c_7 \\ 0 & \dots & a_8 & 0 & b_8 & c_8 \\ 0 & \dots & a_9 & 0 & 0 & b_9 \end{pmatrix} \begin{pmatrix} x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_7 \\ d_8 \\ d_9 \end{pmatrix}$$

5) 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:

$$\begin{pmatrix} b_3 & c_3 & 0 \\ a_6 & b_6 & c_6 \\ 0 & a_9 & b_9 \end{pmatrix} \begin{pmatrix} x_3 \\ x_6 \\ x_9 \end{pmatrix} = \begin{pmatrix} d_3 \\ d_6 \\ d_9 \end{pmatrix}$$

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

6) 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).
7) Далее полученные на каждом процессе кусочки вектора x объединяются и получается вектор решения СЛАУ с трехдиагональной матрицей.

2 Реализация

2.1 C & 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 с матрицей, имеющей вид:

2	3	0	0	0	0	0	0	0
1	2	3	0	0	0	0	0	0
0	1	2	3	0	0	0	0	0
0	0	1	2	3	0	0	0	0
0	0	0	1	2	3	0	0	0
0	0	0	0	1	2	3	0	0
0	0	0	0	0	1	2	3	0
0	0	0	0	0	0	1	2	3
0	0	0	0	0	0	0	1	2

где N = 9 и вектор значений правой части СЛАУ:

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

Распараллеливание происходит на три процесса.

```
$ module load mpi/openmpi/4.0.1/gcc/9
tm5u21@login1:~
```

```
tm5u21@login1:~
$ mpicc C_MPI.c
```

```
tm5u21@login1:~
$ mpirun -np 3 ./a.out
--------------------------------------------------------------------------
By default, for Open MPI 4.0 and later, infiniband ports on a device
are not used by default.  The intent is to use UCX for these devices.
You can override this policy by setting the btl_openib_allow_ib MCA parameter
to true.

  Local host:             login1
  Local adapter:          mlx4_0
  Local port:             1

--------------------------------------------------------------------------
--------------------------------------------------------------------------
WARNING: There was an error initializing an OpenFabrics device.

  Local host:   login1
  Local device: mlx4_0
--------------------------------------------------------------------------

Для процесса № 1 x_part = [ -88.000000; 82.000000; -24.000000 ]
Для процесса № 2 x_part = [ -10.000000; 16.000000; -6.000000 ]
Для процесса № 0 x_part = [ -598.000000; 400.000000; -66.000000 ]

Массив решений x = [ -598.000000; 400.000000; -66.000000; -88.000000; 82.000000; -24.000000; -10.000000; 16.000000; -6.000000 ]
[login1.cluster:03323] 2 more processes have sent help message help-mpi-btl-openib.txt / ib port not selected
[login1.cluster:03323] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
[login1.cluster:03323] 2 more processes have sent help message help-mpi-btl-openib.txt / error in device init
tm5u21@login1:~
```

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:

```
tm5u21@login1:~
$ vim C_MPI.c
tm5u21@login1:~
$ mpicc C_MPI.c
tm5u21@login1:~
$ mpirun -np 4 ./a.out
--------------------------------------------------------------------------
By default, for Open MPI 4.0 and later, infiniband ports on a device
are not used by default.  The intent is to use UCX for these devices.
You can override this policy by setting the btl_openib_allow_ib MCA parameter
to true.

  Local host:             login1
  Local adapter:          mlx4_0
  Local port:             1

--------------------------------------------------------------------------
--------------------------------------------------------------------------
WARNING: There was an error initializing an OpenFabrics device.

  Local host:   login1
  Local device: mlx4_0
--------------------------------------------------------------------------
The values of x are written in C_MPI_val.txt
Time = 0.547 ms.
```

2.2 Python & MPI

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

```
tm5u21@login1:~
$ cd python-envs
tm5u21@login1:~/python-envs
$ source exx/bin/activate
(exx) tm5u21@login1:~/python-envs
$ cd
(exx) tm5u21@login1:~
$ module load python/3.9
(exx) tm5u21@login1:~
$ module load mpi/openmpi/4.0.1/gcc/9
(exx) tm5u21@login1:~
$ mpirun -np 3 python3 ParallelProgram_Python.py
```

```
Для процесса № 0 x_part = [ 0.5000; 0.0000; 0.5000 ]

Для процесса № 1 x_part = [ 0.0000; 0.5000; 0.0000 ]

Для процесса № 2 x_part = [ 0.5000; 0.0000; 0.5000 ]



Массив решений x = [ 0.5000;0.0000;0.5000;0.0000;0.5000;0.0000;0.5000;0.0000;0.5000 ]

[login1.cluster:91873] 2 more processes have sent help message help-mpi-btl-openib.txt / ib port not selected
[login1.cluster:91873] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
[login1.cluster:91873] 2 more processes have sent help message help-mpi-btl-openib.txt / error in device init
(exx) tm5u21@login1:~
```

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

2	1	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0
0	1	2	1	0	0	0	0	0
0	0	1	2	1	0	0	0	0
0	0	0	1	2	1	0	0	0
0	0	0	0	1	2	1	0	0
0	0	0	0	0	1	2	1	0
0	0	0	0	0	0	1	2	1
0	0	0	0	0	0	0	1	2

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:

```
The values of x are written in Python_MPI_val.txt

Time =  0.000843 s.
```

2.3 C & 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 C & OpenMP:

```
$ gcc -fopenmp C_OpenMP.c -o C_OpenMP
tm5u21@login1:~
```

```
$ ./C_OpenMP 3

A =
        2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000;

b = [1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000]

x = [0.5000; 0.0000; 0.5000; 0.0000; 0.5000; 0.0000; 0.5000; 0.0000; 0.5000]
The values of x are written in C_OpenMP_val.txt
Time = 0.208 ms.
```

2.4 C & 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 C & OpenMP:

```
tm5u21@login1:~
$ gcc C_Pthreads.c -o C_Pthreads -lpthread -D DEBUG
```

```
tm5u21@login1:~
$ ./C_Pthreads 3

A =
        2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000, 0.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000, 1.0000;
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.0000;

b = [1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000; 1.0000]

x = [0.5000; 0.0000; 0.5000; 0.0000; 0.5000; 0.0000; 0.5000; 0.0000; 0.5000]
The values of x are written in C_Pthreads_val.txt
Time = 0.759 ms.
```

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:

```
x = A\B;
x = round(x, 4);
writematrix(x, 'expected_val.txt');
```

Для тестирования полученных значений с использованием различных технологий был написан алгоритм, сравнивающий значения из файлов C_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:

```
$ module load python/3.9
tm5u21@login1:~
$ python3 Testing.py
C_MPI_TEST COMPLETED
Python_MPI_TEST COMPLETED
C_OpenMP_TEST FAILED
C_Pthreads_TEST FAILED
```

Пример № 2 запуска, тестирующего алгоритма:

```
$ python3 Testing.py
C_MPI_TEST COMPLETED
Python_MPI_TEST COMPLETED
C_OpenMP_TEST COMPLETED
C_Pthreads_TEST COMPLETED
```

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	C & 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 C & MPI.

4.2 Зависимость времени C & 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:

```
$ sbatch core.sh
Submitted batch job 2696900
tm5u21@login1:~
$ cat out
/tmp/slurmd/job2696900/slurm_script: line 8: exx/bin/activate: Permission denied
--------------------------------------------------------------------------
By default, for Open MPI 4.0 and later, infiniband ports on a device
are not used by default.  The intent is to use UCX for these devices.
You can override this policy by setting the btl_openib_allow_ib MCA parameter
to true.

  Local host:              n02p008
  Local adapter:           mlx5_0
  Local port:              1

--------------------------------------------------------------------------
--------------------------------------------------------------------------
WARNING: There was an error initializing an OpenFabrics device.

  Local host:   n02p008
  Local device: mlx5_0
--------------------------------------------------------------------------
Значения вектора решений x записаны в файл Python_MPI_val.txt

Time =  0.000761
```

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

Число узлов	C & 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 C/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 C & 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, cN = 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 C & 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
C_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 C & 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*numberOfTh
readsPerProcessor):(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,&par
am[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")
```