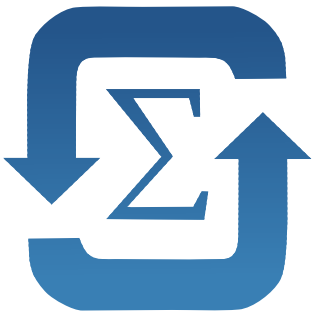
Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра параллельных вычислительных технологий

Расчётно-графическая работа
по дисциплине «Параллельное программирование»



ФАКУЛЬТЕТ:	ПМИ
ГРУППА:	ПМИ-61
СТУДЕНТ:	Ершов П.К.
БРИГАДА:	2
ПРЕПОДАВАТЕЛИ:	Городничев М.А. Щукин Г.А.

Новосибирск

2020

1. Цель работы

Реализация аналогов функций MPI_Send, MPI_Recv и MPI_Reduce с помощью операций с сокетами. Реализовать задачу решения СЛАУ методом простых итераций, используя новые функции, сравнить эффективность с MPI-реализацией.

2. Характеристики системы

Для тестирования MPI.

Описание системы	
Аппаратная конфигурация	Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz, ОЗУ: 12.0 ГБ
Программная конфигурация	ОС Ubuntu (64-bit), оперативная память 4848 МБ, компилятор gcc version 7.5.0 (Ubuntu 7.5.0- 3ubuntu1~18.04) Уровень оптимизации: O3

Для тестирования myMPI.

Описание системы	
Аппаратная конфигурация	Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz, ОЗУ: 12.0 ГБ
Программная конфигурация	ОС Windows (64-bit), оперативная память 4848 МБ, компилятор g++ (GCC) 9.2.0 (Windows 10) Уровень оптимизации: O3

3. Метод решения задачи

В качестве метода решения СЛАУ вида Ax = b был выбран метод Якоби как частный случай метода простой итерации.

Суть метода заключается в вычислении приближения к решению, так называемого вектора невязки.

Данный метод эффективен для матриц с диагональным преобладанием.

В данном решении была реализована поэлементная формула:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j}^{n} a_{ij}\, x_j^{(k)}\right), i = 1, \dots, n, \text{где: } n - \text{размерность системы},$$

$b -$ результиирующий вектор СЛАУ, $a_{ij} -$ элементы матрицы A,

$x_i^{(k+1)} -$ новый вектор невязки, $x_j^{(k)} -$ старый вектор невязки.

Таким образом, каждый новый процесс должен вычислять свою часть вектора невязки, для своей части матрицы A.

$$\begin{cases} 0: x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j}^{n} a_{ij}\, x_j^{(k)}\right) i = k, \dots, \frac{n}{size}; j = 1, \dots, n; \\ 1: x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j}^{n} a_{ij}\, x_j^{(k)}\right) i = k, \dots, \frac{n}{size}; j = 1, \dots, n; \\ \dots \\ \dots \\ x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j}^{n} a_{ij}\, x_j^{(k)}\right) i = k, \dots, \frac{n}{size}; j = 1, \dots, n; \end{cases},$$

где $k = rank * \frac{n}{size}, n -$ размерность системы, $size -$ число процессов,

$rank -$ номер процесса.

В конце каждого процесса необходимо собирать невязку в главный процесс и из него рассылать другим процессам части нового вектора. Так же необходимо собирать вычисленную в каждом процессе норму и выбирать максимальное значение, которое позже рассылается другим процессам. Если не учитывать норму, то каждый процесс будет делать лишние вычисления.

Разработка my_MPI.

В указанном задании необходимо было реализовать функции MPI_send, MPI_recv и MPI_reduce на сокетах. Были выбраны winsock для реализации.

MPI_mySend по факту представлен функцией send с параметрами:

SOCKET n_sock – номер сокета, на который отправляется сообщение.

const char * buf – указатель на сообщение, которое отправляется;

int size – размер сообщения в байтах;

int flag – флаг.

При запуске MPI_mySend ему передаётся номер сокета из вектора сокета. Также функция MPI_mySend получает указатель типа void, который при передаче в send преобразуется в char *, размер сообщения и тип данных в виде строковой переменной.

Вектор сокетов инициализируется при запуске функции Init. Эта же функция проверяет подключение и запускает сокет процесса на прослушивание.

```
void MPI_MySend(void *buf, int count, string type, int i)
```

`void *buf` – указатель на входное сообщение.

`int count` – размер сообщения.

`string type` – тип данных сообщения.

`int i` – номер сокета, он же номер процесса.

MPI_myRecv организован по похожему принципу с применением функции recv с параметрами:

SOCKET n_sock – номер сокета, на который отправляется сообщение.

const char * buf – указатель на сообщение;

int size – размер сообщения в байтах;

int flag – флаг.

```
void MPI_MyRecv(void *buf, int count, string type, int i)
```

`void *buf` – указатель на выходное сообщение.

`int count` – размер сообщения.

`string type` – тип данных сообщения.

`int i` – номер сокета, он же номер процесса.

Функция MPI_MyReduce представляет собой функцию сборки указанного сообщения из всех процессов в процесс-сервер.

Алгоритм работы данной функции следующий:

Если собственный ранг процесса равен рангу процесса-сервера, то запускается цикл сбора сообщений с помощью recv и выполнением указанной операции.

Если же ранг отличается от ранга сервера, то процесс отправляет с помощью send указанное сообщение.

```
void MPI_MyReduse(void *buf, void *send_b, int count, string type, string operation, int i)
```

`void *buf` – указатель на сообщение, которое отправляется.

`void *send_b` – указатель на память, в которую собираются сообщения.

`int count` – размер сообщения.

`string type` – тип данных.

`string operation` – тип операции.

`int i` – номер процесса-сервера.

4. Результаты тестирования

Для компиляции необходимо в папке приложения открыть терминал и ввести команду:

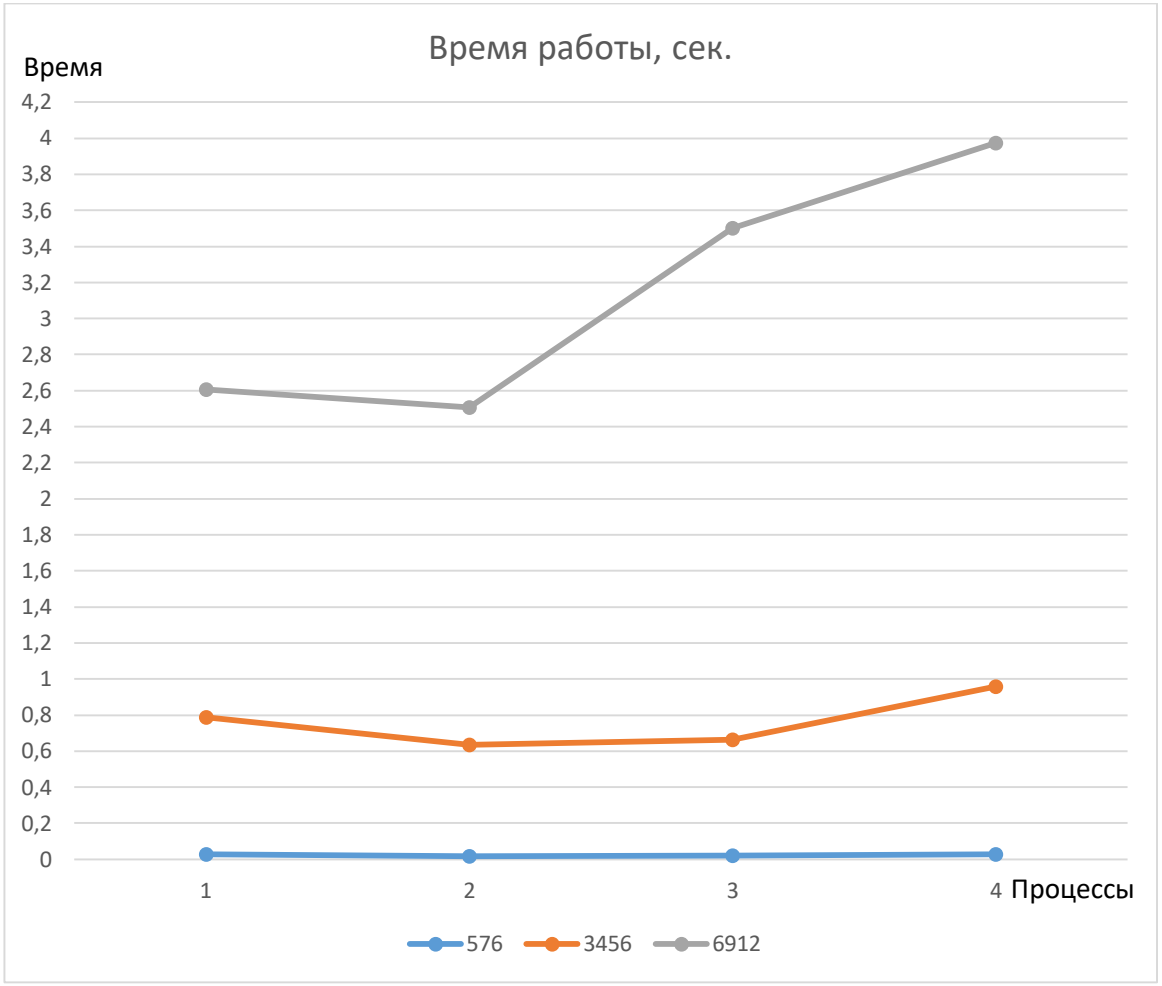
mpicc –o exe mpi_f.c mpi_Jacoby.c –lm

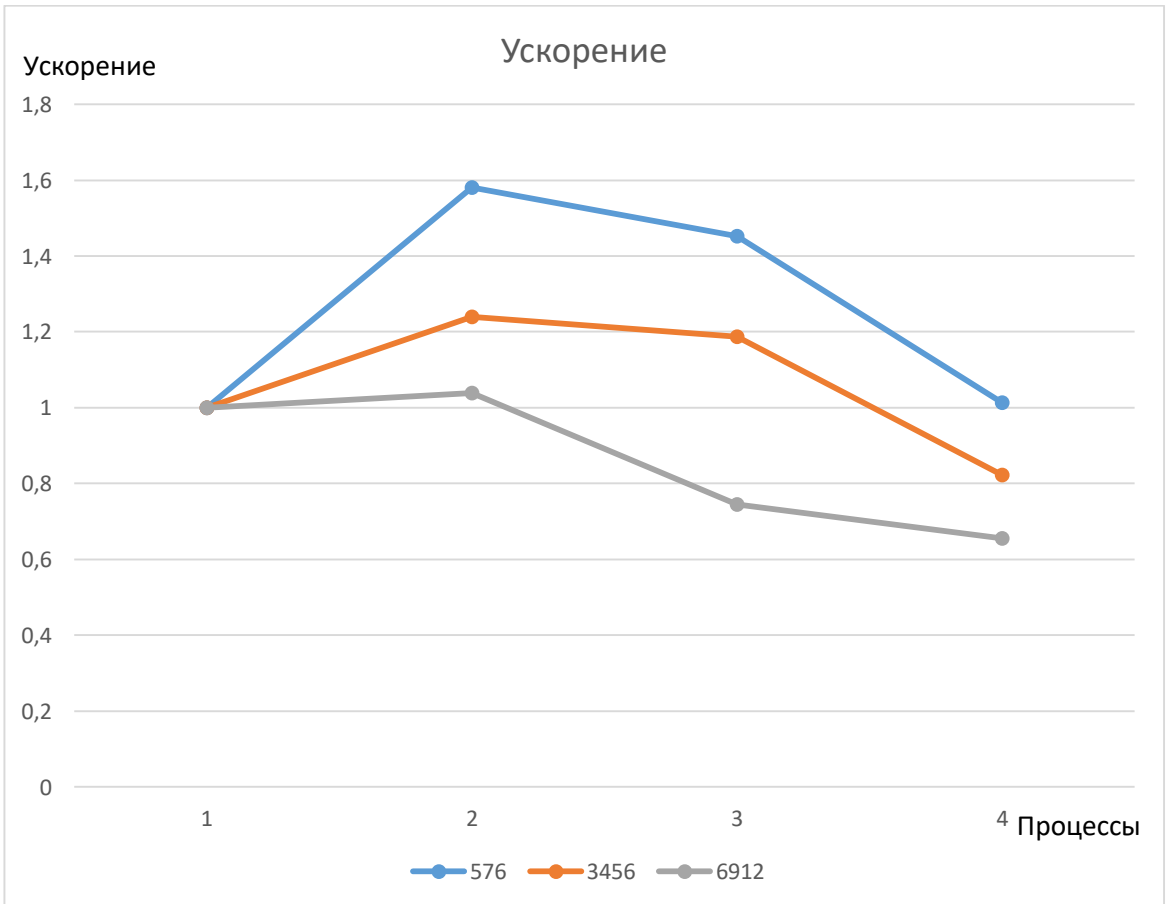
Для запуска необходимо ввести команду:

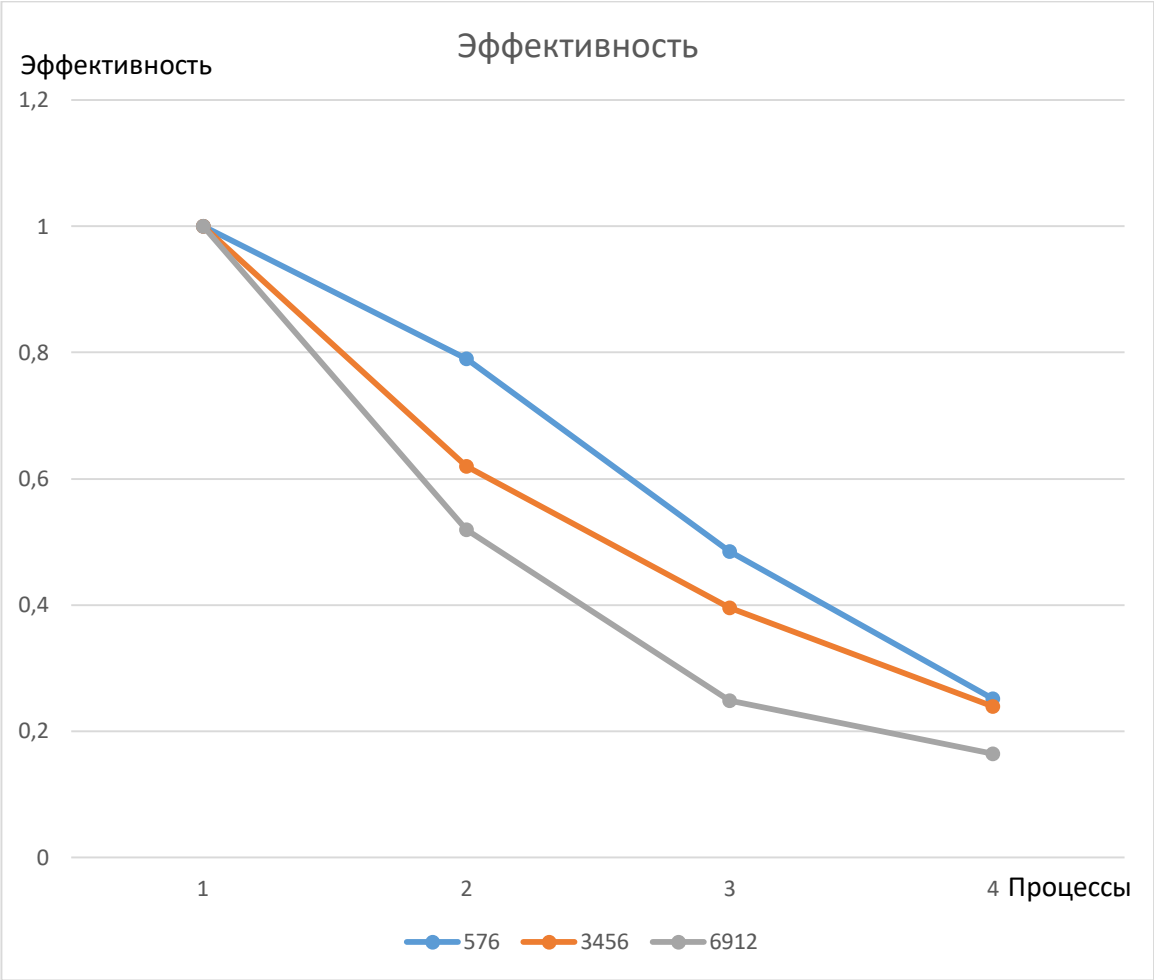
mpirun –np size exe n, где size – число потоков, n – размерность системы.

Программа на MPI.

Размерность задачи	Количество процессов	Время работы, сек.	Ускорение	Эффективность
576	1	0,028478	1	1
	2	0,018024	1,58	0,79
	3	0,019599	1,45303	0,48434
	4	0,028095	1,01363	0,25091
3456	1	0,787847	1	1
	2	0,635552	1,239626	0,619813
	3	0,663950	1,1866059	0,395535
	4	0,957703	0,82264	0,23942575
6912	1	2,604806	1	1
	2	2,508387	1,0384386	0,519219
	3	3,500817	0,7440566	0,2480188
	4	3,974648	0,6553469	0,163836


Время работы, сек.
Время
4,2
4
3,8
3,6
3,4
3,2
3
2,8
2,6
2,4
2,2
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1
2
3
4
Процессы
576
3456
6912


Ускорение
Ускорение
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
1
2
3
4
Процессы
576
3456
6912



Программа на my_MPI.

Для компиляции программы необходимо в командной строке, которая открыта в папке программы ввести следующее:

g++ my_mpi_f.cpp mpi_Jacoby.cpp -o mpi_Jacoby.exe -lwsock32

Для запуска каждого процесса необходимо отдельное окно командной строки.

Команда запуска программы:

mpi_Jacoby.exe rank size n

где rank – номер процесса, size – число процессов, n – размерность задачи.

Запускать окна необходимо в обратном порядке, т. е. процессе size – 1 является сервером, а все последующие процессы должны идти в порядке убывания.

Пример:

Для задачи с тремя процессами и матрицей 10 на 10 последовательность запуска будет следующей:

Первое окно:

mpi_Jacoby.exe 2 3 10

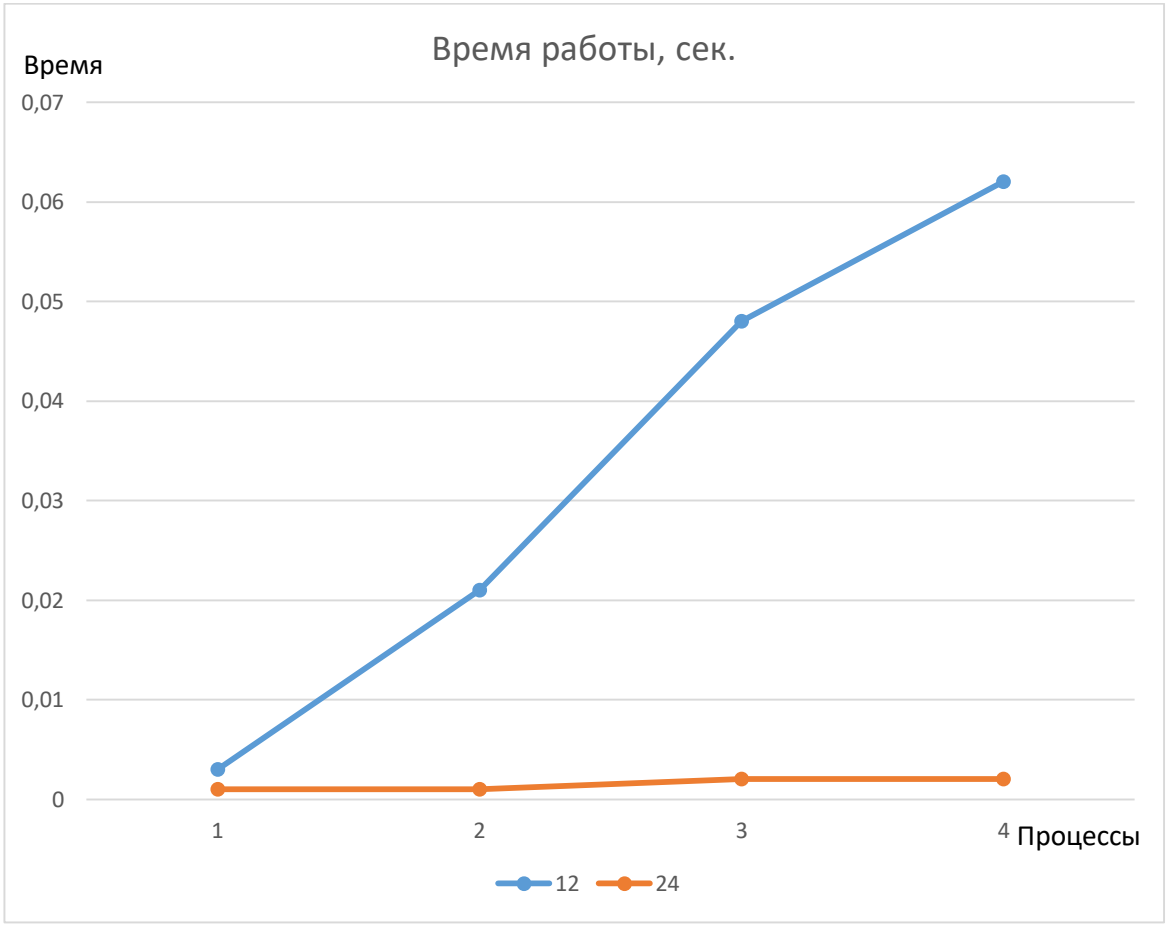
Второе окно:

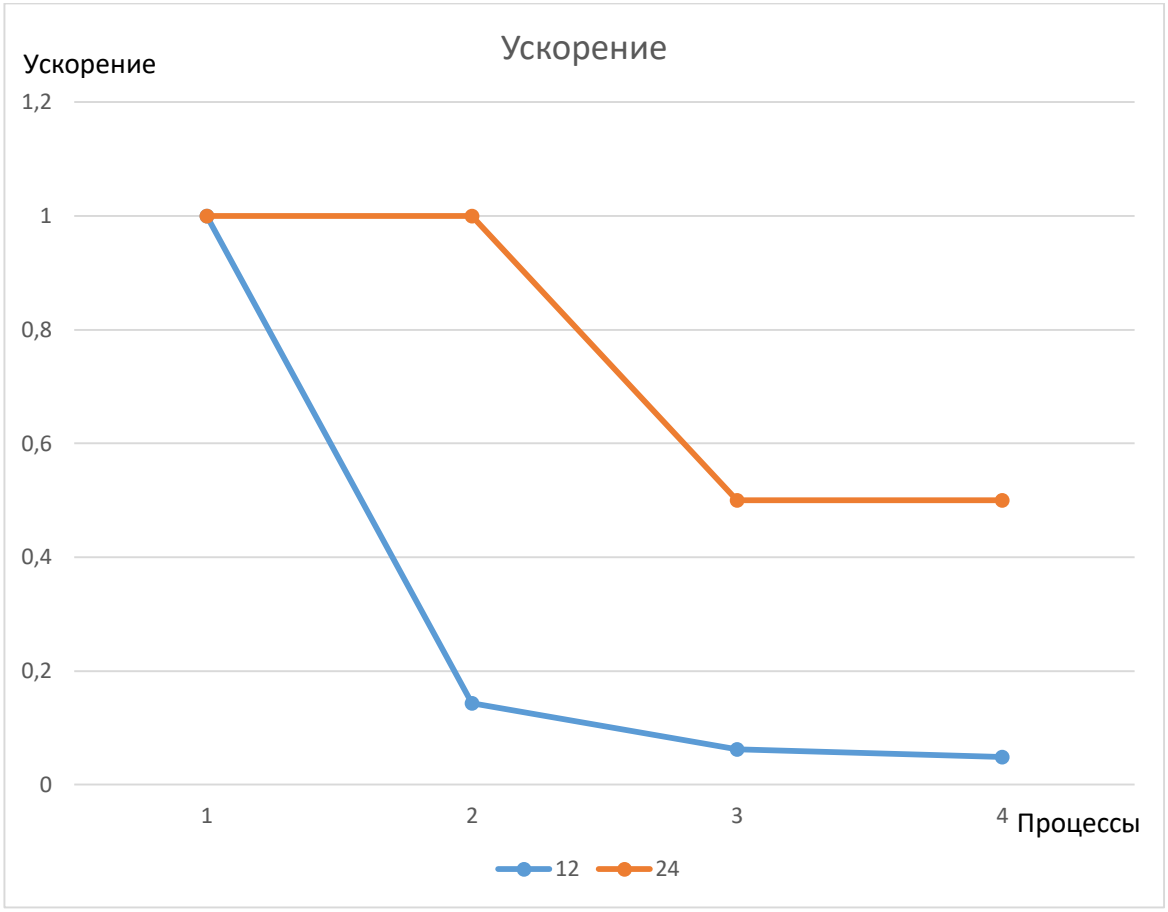
mpi_Jacoby.exe 1 3 10

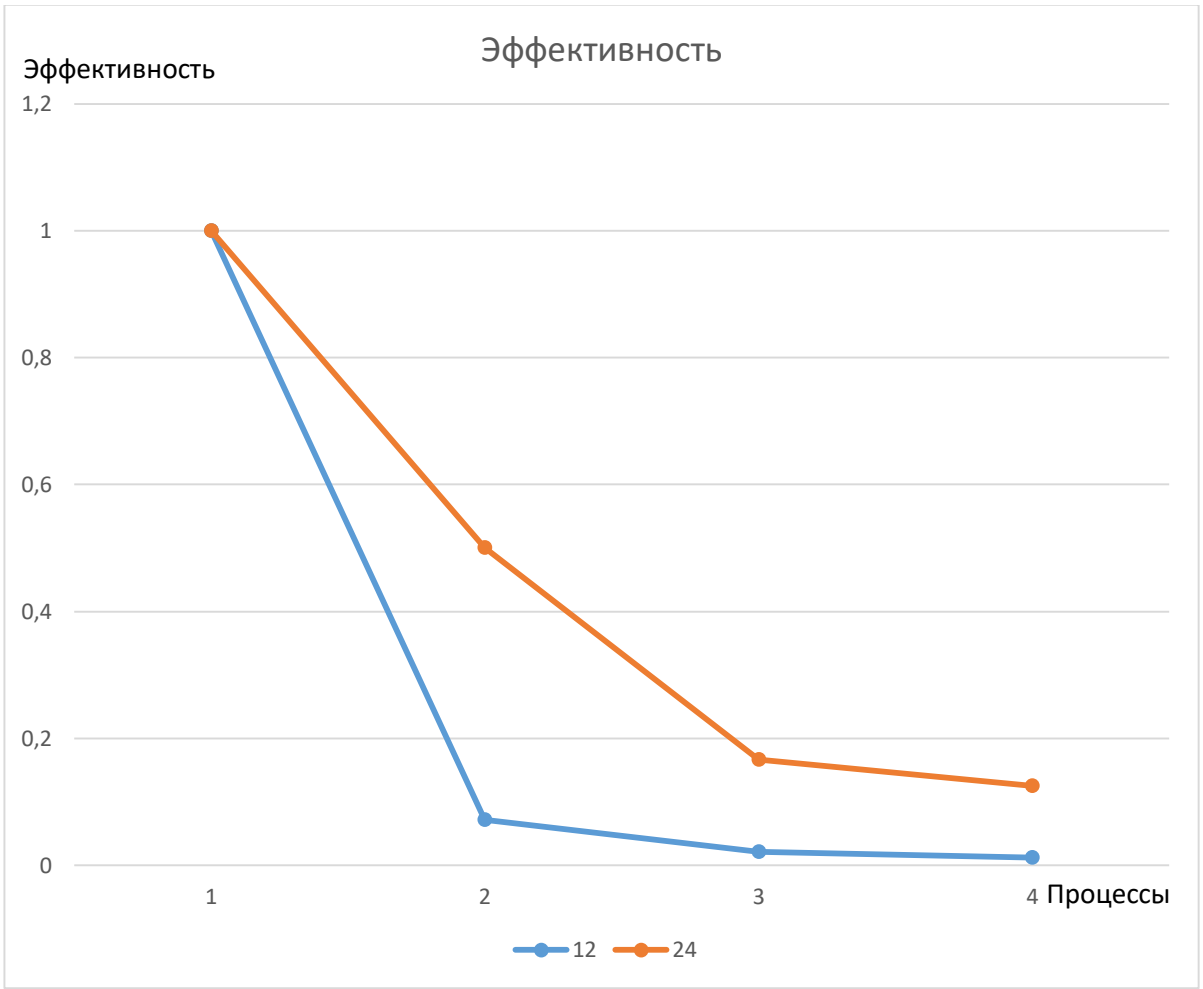
Третье окно:

mpi_Jacoby.exe 0 3 10

Размерность задачи	Количество процессов	Время работы, сек.	Ускорение	Эффективность
12	1	0,003	1	1
	2	0,021	0,142857	0,0714285
	3	0,048	0,0625	0,020833
	4	0,062	0,04838	0,012095
24	1	0,001	1	1
	2	0,001	1	0,5
	3	0,002	0,5	0,1666
	4	0,002	0,5	0,125




Ускорение
Ускорение
1,2
1
0,8
0,6
0,4
0,2
0
1
2
3
4
Процессы
12
24


Эффективность
Эффективность
1,2
1
0,8
0,6
0,4
0,2
0
1
2
3
4
Процессы
12
24

5. Выводы

5.1. Исследование MPI.

На графиках видно, что сначала, для двух потоков идёт ускорение работы, но позже оно падает. Вероятно, это связано с тем, что большое время занимает передача данных в каждый поток на каждой итерации цикла вычисления невязки. Однако, можно заметить, что падение ускорения с ростом размерности задачи уменьшается. Так, график ускорения для задачи с размерностью 6912 несколько более пологий, чем для задачи с размерностью 3456.

5.2. Исследование для my_MPI.

На полученных графиках хорошо видно, что увеличение размерности задачи всего в 2 раза уже повышает эффективность и ускорение. Из этого можно сделать вывод, что увеличение размерности задачи повышает производительность.

Приложение 1. Реализация метода Якоби на MPI.

mpi_f.h

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <mpi.h>

#define MAX_ITER 1000 // предельное количество итераций
#define eps 1.0E-5 // точность приближения
#define BILLION 1.0E+9

// Функция вычета разности между временными величинами
#define clocktimeDifference(start, stop)            \
    1.0 * (stop.tv_sec - start.tv_sec) +            \
    1.0 * (stop.tv_nsec - start.tv_nsec) / BILLION

float *take_mass(float *arr, int size, int N); // функция разделения массива

float *take_diag(float *arr, float *diag, int N); // выделение диагонали из матрицы

float *creat_matrix(float * mat, int n); // генерация матрицы

float *creat_vec(float * vec, int n); // генерация вектора

void print_equation(float *a, int n); // вывод матрицы

void print_vector(float *v, int n); // вывод вектора
```

mpi_f.c

```c
#include "mpi_f.h"

float *take_mass(float *arr, int size, int N) // разбиваем массив на под массивы
{
  int n = N / size; // размер части массива, котоаря попадёт в поток
  int k = 0;
  float *mass = (float*)malloc(sizeof(float) * size * N); // размер входного вектора умножить на число потоков

  for(int i = 0; i < size; i++)
  {
    for(int j = 0; j < N; j++) // зануляем значения выходного вектора
        mass[i * size + j] = 0;

    for(int j = k; j < n + k; j++) // заносим в массив только часть, которая попадает в указанную область
        mass[i * size + j] = arr[j];
    k += n; // увеличиваем чёстчит на размер части
  }
```

```
  return mass;
}

float *take_diag(float *arr, float *diag, int N) // выделяем диагональ из матрицы
{
    for(int i = 0; i < N; i++)
        diag[i] =  arr[i * N + i];
    return diag;
}

float *creat_matrix(float * mat, int n) // создаём случайную матрицу
{
    float sum; // парметр суммы элементов вне диагонали
    int seed = time(0) % 100; // параметр времни для получения сида рандомных значений

    srand(seed); // изменение случайных значений в зависимости от текущего времени
    for (int i = 0; i < n; i++) // общий цикл
    {
        for (int j = 0; j < n; j++)
        {
            mat[i * n + j] = rand() % 7; // получаем числа в диапазоне до 0 до 7
            if (rand() & 1) // меняем знак некоторых чисел чисел
                mat[i* n + j] *= -1;
        }
        sum = 0; // обнуляем сумму
        for (int j = 0; j < n; j++)
            if(i != j) // если элемент вне диагонали
                sum += abs(mat[i * n + j]); // заносим его в сумму
        if (mat[i* n + i] < sum) // если диагональный элемент меньше суммы элементво вне диагонали
            mat[i* n + i] += 2 * sum; // прибавляем к ниму удвоенную сумму внедиагональных элементов,
                                      // чтобы добиться диагонального преобладания в матрице
    }
        return mat;
}

float *creat_vec(float * vec, int n) // создаём случайный вектор
{
    int seed = time(0) % 100; // параметр времни для получения сида рандомных значений

    srand(seed); // изменение случайных значений в зависимости от текущего времени
    for (int i = 0; i < n; i++)
    {
        vec[i] = rand() % 10; // получаем значения от 0 до 10
        if (rand() & 1 && vec[i] != 0) // меняем знак у некоторых элементов вектора
            vec[i] *= -1;
    }
        return vec;
}

void print_equation(float *a, int n) // выводим матрицу
{
    printf("A*x = b\n");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            printf("%2d ", (int)(a[i * n + j]));
        printf("\n");
    }
    printf("\n");
}

void print_vector(float *v, int n) // выводим вектор
{
    for(int i = 0; i < n; i++)
        printf("%.2f ",v[i]);
    printf("\n");
}
```

mpi_Jacoby.c

```c
#include "mpi_f.h"

int main(int argc, char **argv)
{
    float *a; // матрица
    float *b; // вектор
    float *x; // невязка
    float *buf; // старое значение вектора невязки
    float *diag; // массив диагональных элементов
    float *loc_mat; // локальная часть матрицы
    int n; // размерность стистемы
    float *r; // массив для передачи частей массива
    float max; // максимальное значение нормы
    int tag = 1; // параметр передачи и приёма для MPI_Send и MPI_Recv
    int iter; // количество итераций
    int size; // количество потоков
    int rank; // ранг потока
    MPI_Status status;
    MPI_Init (&argc, &argv); // инициализация MPI
    MPI_Comm_size (MPI_COMM_WORLD, &size); // получение числа потоков
    MPI_Comm_rank (MPI_COMM_WORLD, &rank); // получение ранга потока
    n = atoi(argv[1]); // полчение размерности ситемы

    float send[n]; // инициализация массвиа для сборки всех частей вектора невязки между потоками
    a = (float *) malloc(sizeof (float) * n * n);
    b = (float *) malloc(sizeof (float) * n);
    x = (float *) malloc(sizeof (float) * n);
    diag = (float *) malloc(sizeof (float)* n);
    buf = (float *) malloc(sizeof (float) * n);
    loc_mat = (float *) malloc(sizeof (float) * n * n / size);

    for(int i = 0; i < n; i++) // обнуляем вектор сборки частей вектора невязки
        send[i] = 0;

    MPI_Barrier(MPI_COMM_WORLD); // инициализируем барьер, чтобы можно было корректно получить время выполнения
программы
    struct timespec start, stop;
        clock_gettime(CLOCK_MONOTONIC, &start); // получаем текущее время

    if(rank == 0)
    {
        a = creat_matrix(a, n); // получаем случайную матрицу A
        b = creat_vec(b, n);    // получаем случайный вектор b

        diag = take_diag(a, diag, n); // извлекаем из матрицы диагональ

        for (int i = 0; i < n; i++) // присваиваем вектору невязки x начальное приближение
            x[i] = 1;

        r = take_mass(b, size, n); // разбиваем случайный вектор b на части
        int t = n * n / size; // присваиваем параметру смещения начально указателя на
                              // элемент матрицы значений размера блока матрицы для каждого потока

        for(int i = 1; i < size; i++) // рассылаем всем потокам, кроме 0 начальные параметры
        {
            MPI_Send(x, n, MPI_FLOAT, i, tag, MPI_COMM_WORLD); // рассылваем вектор невязки
            MPI_Send(r + i * size, n, MPI_FLOAT, i, tag, MPI_COMM_WORLD); // рассылваем части вектора b
            MPI_Send(a + t, n * n / size, MPI_FLOAT, i, tag, MPI_COMM_WORLD); // рассылаем части матрицы
            MPI_Send(diag, n, MPI_FLOAT, i, tag, MPI_COMM_WORLD); // рассылаем диагональ
            t += n * n / size; // увеличивем параметр смещения
        }
        loc_mat = a; // присваеваем свою часть матрицы бля процесса 0
    }
    else
    {
        for(int i = 0; i < n; i++) // обнуляем вектор невязки в каждом потоке
            x[i] = 0;
        MPI_Recv (send, n, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // получаем вектор невязки
        MPI_Recv (b, n, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // получаем вектор b
        MPI_Recv (loc_mat, n * n / size, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // получаем часть матрицы
        MPI_Recv (diag, n, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // получаем диагональ

        for(int j = rank * n / size; j < n / size + rank * n/ size; j++) // переносим в вектор невязки те
элементы, которые нужны процессу
            x[j] = send[j];
```

```
        for(int i = 0; i < n; i++) // обнуляем вектор сборки невязки
            send[i] = 0;
    }

    float norm; // создаём параметр нормы
    float sum; // создаём парамерт суммы

    do // запускаем основной цикл метода Якоби
    {
        if(rank == 0) // для процесса 0
        {
            for(int i = 1; i < size; i++) // рассылаем обновлённый вектор невязки процесам
            {
                MPI_Send(x, n, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
            }
            for(int j = n / size; j < n; j++) // обнуляем часть вектора неваязки, котоаря не нужна для процесса,
                                              // чтобы при сборке результатов MPI_Reduce не накапливать ненужные
значения
                x[j] = 0;
        }
        else
        {
            for(int i = 0; i < n; i++) // обнуляем вектор невязки в каждом потоке
                x[i] = 0;
            MPI_Recv (send, n, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // получаем вектор невязки

            for(int j = rank * n / size; j < n / size + rank * n/ size; j++) // переносим в вектор невязки те
элементы, которые нужны процессу
                x[j] = send[j];
            for(int i = 0; i < n; i++) // обнуляем вектор сборки невязки
                send[i] = 0;
        }

        int p = rank * n / size; // задаём параметр смещения для правильно вычисления вектора невязки
        for(int i = 0; i < n / size; i++) // запускаем вычисления вектора невязки
        {
            sum = 0; // обнуляем сумму
            for(int j = 0; j < n; j++)
            {
                if ( loc_mat[i * n + j]!= diag[i + p])  // получаем результат умножение матрицы на вектор,
                    sum += (loc_mat[i * n + j] * x[j]); // исключая диагональные элементы
            }
            buf[i + p] = (b[i + p] - sum) / diag[i + p]; // получаем промежуточное значени вычитая сумму из
вектора b и деля её на диагональный элемент

            norm = fabs(x[p] - buf[p]); // вычисяем норму вектора
            for(int h = p; h < p + n / size; h++)
            {
                if(fabs(x[h] - buf[h]) > norm)
                    norm = fabs(x[h] - buf[h]);
                x[h] = buf[h]; // присваиваем новое значение вектору невязки
            }
        }

        MPI_Reduce(&norm, &max, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD); // получаем максимальное среди
процессов значение нормы

        if (rank == 0)
        {
            norm = max; // присваиваем значние максимума норм общей норме
            for (int i = 1; i != size; ++i) // и рассылваем всем процессам новую норму, чтобы они не выполнили
лишних вычислений
                MPI_Send(&max, 1, MPI_FLOAT, i, tag, MPI_COMM_WORLD);
        }
        else
            MPI_Recv(&norm, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status); // принимаем новую норму

        MPI_Reduce(x, send, n, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD); // собираем новый вектор невязки

        if(rank == 0)
        {
            for(int i = 0; i < n; i++) // обновляем значение вектора невязки
                x[i] = send[i];
        }

        iter++; // увеличивавем число итераций
    }
```

```
    while(norm > eps && iter < MAX_ITER); // продолжаем вычисления, пока норма больше параметра приближения
                                          // и пока не превышенно максимальное число итераций

    MPI_Barrier(MPI_COMM_WORLD); // ждём пока все процессы закончат работу

    clock_gettime(CLOCK_MONOTONIC, &stop); // завершаем замер времени

    if(rank == 0)
    {
        print_vector(b, n); // выводим вектор
        print_equation(a, n); // выводим матрицу
        for(int i = 0; i < n; i++)  // выводим результирующий веткор невязки
            printf("x[%d] = %0.9f \n", i, x[i]);
        printf("%d \n", iter); // выводим число итераций
        printf("Elapsed time: %lf\n", clocktimeDifference(start, stop)); // выводим время выполнения
    }

    // очищаем память
    free(a);
    free(b);
    free(x);
    MPI_Finalize(); // завершаем параллельную часть программы
    return 0;
}
```

Приложение 2. Реализация метода Якоби на myMPI.

my_mpi_f.h

```
#ifndef MY_MPI_F_H
#define MY_MPI_F_H
#pragma comment (lib, "ws2_32.lib")
#define _WINSOCK_DEPRECATED_NO_WARNINGS

#include <sys/types.h>
#include <WinSock2.h>
#include <winsock.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include <vector>
#include <cmath>
#include <ctime>
#include <locale.h>
#include <iostream>
#include <string>

#define MAX_ITER 1000 // предельное количество итераций
#define eps 1.0E-5 // точность приближения

using namespace std;
extern vector<SOCKET> sockets;
extern int socketRank;
extern int countOfProcess;

extern  HOSTENT *hostent;

void Init();

void MPI_MySend(void *buf, int count, string type, int i); // Отправка сообщения (указатель на данные, размер данных, тип - int,
куда отправить)

void MPI_MyRecv(void *buf, int count, string type, int i); // Приём сообщения (указатель на область памяти, размер получаемых
данных, от какого процесса записывать)


void MPI_MyReduse(void *buf, void *send_b, int count, string type, string operation, int i); // Сборка указанного сообщения в
указанном процессе
                                                                                             // (указатель на область памяти,
что передаём, указатель на область куда передаём,
                                                                                             // размер получаемых данных, тип
данных, операция, в какой процесс записывать)

double *creat_matrix(double * mat, int n); // создаём случайную матрицу

double *creat_vec(double * vec, int n); // создаём случайный вектор

double *take_diag(double *arr, double *diag, int N); // выделяем диагональ из матрицы

double *take_mass(double *arr, int size, int N); // разбиваем массив на под массивы
```

```
void print_equation(double *a, int n); // выводим матрицу

void print_vector(double *v, int n); // выводим вектор

#endif
```

my_mpi_f.cpp

```
#include "my_mpi_f.h"

vector<SOCKET> sockets;
int socketRank;
int countOfProcess;
HOSTENT *hostent;

void Init()
{
    int start_port = 1000;
    WORD version = MAKEWORD(2, 2);
    WSADATA wsaData;
    typedef unsigned long IPNumber;
    // Инициализация Winsock
    WSAStartup(version, (LPWSADATA)&wsaData);
    std::vector<SOCKADDR_IN> servers(countOfProcess);
    // Вектор сокетов для всех процессов
    sockets.resize(countOfProcess);
    //  Инициализация  сокетов
    for (int i = 0; i < servers.size(); i++)
    {
        servers[i].sin_family = PF_INET;
        hostent = gethostbyname("localhost");
        servers[i].sin_addr.s_addr = (*reinterpret_cast<IPNumber*>(hostent->h_addr_list[0]));
        servers[i].sin_port = htons(start_port + i);
        sockets[i] = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sockets[i] == INVALID_SOCKET)
        {
            std::cout << "unable to create socket" << socketRank << std::endl;
            return;
        }
    }

    if (socketRank == countOfProcess - 1)
    {
        printf("Socket port: %d\n", servers[socketRank].sin_port);
        int retVal = ::bind(sockets[socketRank], (LPSOCKADDR)&(servers[socketRank]), sizeof(servers[socketRank]));
        if (retVal == SOCKET_ERROR)
        {
            printf("Unable to bind\n");
            int error = WSAGetLastError();
            printf("%d\n", error);
            WSACleanup();
            system("pause");
            return;
        }

        int task = 0;
        retVal = listen(sockets[socketRank], 10);
        if (retVal == SOCKET_ERROR)
        {
            printf("Unable to listen\n");
            int error = WSAGetLastError();
            printf("%d", error);
            system("pause");
            return;
        }
        SOCKADDR_IN from;
        int fromlen = sizeof(from);
        int buf = 0;

        int *temp = new int[1];
        buf = accept(sockets[socketRank], (struct sockaddr*)&from, &fromlen);
        retVal = recv(buf, (char*)temp, sizeof(int), 0);
        printf("Connect %d process first for \n", temp[0]);
        sockets[temp[0]] = buf;
    }

    for (int i = socketRank + 1; i < countOfProcess; i++)
    {
        int retVal = connect(sockets[i], (LPSOCKADDR) &servers[i], sizeof(servers[i]));
        if (retVal == SOCKET_ERROR)
        {
            std::cout << "unable to connect" << std::endl;
            int error = WSAGetLastError();
            printf("%ld", error);
            return;
```

```
        }

        int *temp = new int[1];
        temp[0] = socketRank;
        retVal = send(sockets[i], (char*)temp, sizeof(int), 0);
        //perror("error");
        if (retVal == SOCKET_ERROR)
        {
            std::cout << "unable to recv" << std::endl;
            int error = WSAGetLastError();
            printf("%d\n", error);
            return;
        }
    }

    int flag = countOfProcess - 1;
    int def = 1;
    if (socketRank == countOfProcess - 1)
        def++;

    for (int i = socketRank - def; i >= 0; i--)
    {
        if (socketRank < flag)
        {
            int retVal = ::bind(sockets[socketRank], (LPSOCKADDR) & (servers[socketRank]), sizeof(servers[socketRank]));
            if (retVal == SOCKET_ERROR)
            {
                printf("Unable to bind\n");
                int error = WSAGetLastError();
                printf("%d\n", error);
                WSACleanup();
                system("pause");
                return;
            }

            int task = 0;
            retVal = listen(sockets[socketRank],  10);
            if (retVal == SOCKET_ERROR)
            {
                printf("Unable to listen\n");
                int error = WSAGetLastError();
                printf("%d", error);
                system("pause");
                return;
            }
        }
        flag--;
        SOCKADDR_IN from;
        int fromlen = sizeof(from);
        int buf = 0;
        int *temp = new int[1];

        buf = accept(sockets[socketRank], (struct sockaddr*)&from, &fromlen);
        int retVal = recv(buf, (char*)temp, sizeof(int), 0);
        printf("Connect %d process \n", temp[0]);
        sockets[temp[0]] = buf;
    }
    int retVal = 0;
    std::cout << "Connection made sucessfully" << std::endl;
}
// Отправка сообщения (указатель на данные, размер данных, тип - int, куда отправить)
void MPI_MySend(void *buf, int count, string type, int i)
{
    int size_;
    if (type == "MPI_INT")
        size_ = count * sizeof(int);
    if (type == "MPI_DOUBLE")
        size_ = count * sizeof(double);
    printf("");
    if (send(sockets[i], (char*)buf, size_, 0) == SOCKET_ERROR)
    {
        std::cout << "unable to send" << std::endl;
        int error = WSAGetLastError();
        printf("%d\n", error);
        return;
    }
}

// Приём сообщения (указатель на область памяти, размер получаемых данных, от какого процесса записывать)
void MPI_MyRecv(void *buf, int count, string type, int i)
{
    int size_;
    if (type == "MPI_INT")
        size_ = count * sizeof(int);
    if (type == "MPI_DOUBLE")
        size_ = count * sizeof(double);

    if (recv(sockets[i], (char*)(buf), size_, 0) == SOCKET_ERROR)
```

```
    {
        std::cout << "unable to recv" << std::endl;
        int error = WSAGetLastError();
        printf("%d\n", error);
        return;
    }
}

// Сборка указанного сообщения в указанном процессе
//(указатель на область памяти, что передаём, указатель на область куда передаём, размер получаемых данных, тип данных,
операция, в какой процесс записывать)
//*
void MPI_MyReduse(void *buf, void *send_b, int count, string type, string operation, int i)
{
    int size_;
    if (type == "MPI_INT")
        size_ = count * sizeof(int);
    if (type == "MPI_DOUBLE")
        size_ = count * sizeof(double);

    if(socketRank == i)
    {
        memcpy(send_b, buf, size_);
        void *u;
        for(int j = 0; j < countOfProcess; j++)
        {
            if(j != i)
            {
                if (recv(sockets[j], (char*)u, size_, 0) == SOCKET_ERROR)
                {
                    std::cout << "REDUCE unable to recv" << std::endl;
                    int error = WSAGetLastError();
                    printf("%d\n", error);
                    printf("%d\n", j);
                    return;
                }
                if(operation == "MPI_SUM")
                {
                    if (type == "MPI_INT")
                        for(int k = 0; k < count; k++)
                            ((int *)send_b)[k] += ((int *)u)[k];

                    if (type == "MPI_DOUBLE")
                        for(int k = 0; k < count; k++)
                            ((double *)send_b)[k] += ((double *)u)[k];
                }
                if(operation == "MPI_MAX")
                {
                    if (type == "MPI_INT")
                        if(*(int *)send_b < *(int *)u)
                            *(int *)send_b = *(int *)u;

                    if (type == "MPI_DOUBLE")
                        if(*(double *)send_b < *(double *)u)
                            *(double*)send_b = *(double*)u;
                }
            }
        }
    }
    else
    {
        printf("");
        if (send(sockets[i], (char*)buf, size_, 0) == SOCKET_ERROR)
        {
            std::cout << "REDUCE unable to send" << std::endl;
            int error = WSAGetLastError();
            printf("%d\n", error);
            return;
        }
    }
}

double *creat_matrix(double * mat, int n) // создаём случайную матрицу
{
    double sum; // парметр суммы элементов вне диагонали
    int seed = time(0) % 100; // параметр времни для получения сида рандомных значений

    srand(seed); // изменение случайных значений в зависимости от текущего времени
    for (int i = 0; i < n; i++) // общий цикл
    {
        for (int j = 0; j < n; j++)
        {
            mat[i * n + j] = rand() % 7; // получаем числа в диапазоне до 0 до 7
            if (rand() & 1) // меняем знак некоторых чисел чисел
                mat[i* n + j] *= -1;
        }
        sum = 0; // обнуляем сумму
        for (int j = 0; j < n; j++)
```

```
            if(i != j) // если элемент вне диагонали
                sum += abs(mat[i * n + j]); // заносим его в сумму
        if (mat[i* n + i] < sum) // если диагональный элемент меньше суммы элементво вне диагонали
            mat[i* n + i] += 2 * sum; // прибавляем к ниму удвоенную сумму внедиагональных элементов,
                                      // чтобы добиться диагонального преобладания в матрице
    }
         return mat;
}

double *creat_vec(double * vec, int n) // создаём случайный вектор
{
    int seed = time(0) % 100; // параметр времни для получения сида рандомных значений

    srand(seed); // изменение случайных значений в зависимости от текущего времени
    for (int i = 0; i < n; i++)
    {
        vec[i] = rand() % 10; // получаем значения от 0 до 10
        if (rand() & 1 && vec[i] != 0) // меняем знак у некоторых элементов вектора
            vec[i] *= -1;
    }
         return vec;
}

double *take_diag(double *arr, double *diag, int N) // выделяем диагональ из матрицы
{
    for(int i = 0; i < N; i++)
        diag[i] =  arr[i * N + i];
    return diag;
}

double *take_mass(double *arr, int size, int N) // разбиваем массив на под массивы
{
  int n = N / size; // размер части массива, котоаря попадёт в поток
  int k = 0;
  double *mass = (double*)malloc(sizeof(double) * size * N); // размер входного вектора умножить на число потоков

  for(int i = 0; i < size; i++)
  {
    for(int j = 0; j < N; j++) // зануляем значения выходного вектора
        mass[i * size + j] = 0;

    for(int j = k; j < n + k; j++) // заносим в массив только часть, которая попадает в указанную область
        mass[i * size + j] = arr[j];
    k += n; // увеличиваем чёстчит на размер части
  }
  return mass;
}

void print_equation(double *a, int n) // выводим матрицу
{
    printf("A*x = b\n");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            printf("%2d ", (int)(a[i * n + j]));
        printf("\n");
    }
    printf("\n");
}

void print_vector(double *v, int n) // выводим вектор
{
    for(int i = 0; i < n; i++)
        printf("%.2f ",v[i]);
    printf("\n");
}
```

mpi_Jacoby.cpp

```
#include "my_mpi_f.h"

int main(int argc, char **argv)
{
    socketRank = atoi(argv[1]);
    countOfProcess = atoi(argv[2]);
    int n = atoi(argv[3]);

    double *a; // матрица
    double *b; // вектор
    double *x; // невязка
    double *buf; // старое значение вектора невязки
    double *diag; // массив диагональных элементов
    double *loc_mat; // локальная часть матрицы
    double *r; // массив для передачи частей массива

    if (countOfProcess != 1)
```

```
        Init();
    a = (double *) malloc(sizeof (double) * n * n);
    b = (double *) malloc(sizeof (double) * n);
    x = (double *) malloc(sizeof (double) * n);
    diag = (double *) malloc(sizeof (double)* n);
    buf = (double *) malloc(sizeof (double) * n);
    loc_mat = (double *) malloc(sizeof (double) * n * n / countOfProcess);
    double *send = (double *) malloc(sizeof (double) * n);  // инициализация массвиа для сборки всех частей вектора невязки
между потоками

    clock_t start = clock();

    if(socketRank == countOfProcess - 1)
    {
        a = creat_matrix(a, n);
        b = creat_vec(b, n);    // получаем случайный вектор b
        diag = take_diag(a, diag, n); // извлекаем из матрицы диагональ

        for (int i = 0; i < n; i++) // присваиваем вектору невязки x начальное приближение
            x[i] = 1;

        r = take_mass(b, countOfProcess, n); // разбиваем случайный вектор b на части
        int t = 0;
        for(int i = 0; i < countOfProcess - 1; i++)
        {
            MPI_MySend(a + t, n * n / countOfProcess, "MPI_DOUBLE", i);
            MPI_MySend(b, n, "MPI_DOUBLE", i);
            MPI_MySend(x, n, "MPI_DOUBLE", i);
            MPI_MySend(diag, n, "MPI_DOUBLE", i);
            t +=  n * n / countOfProcess;
        }
        loc_mat = a + t; // присваеваем свою часть матрицы бля процесса 0
    }
    else
    {
        for (int i = 0; i < n; i++) // присваиваем вектору невязки x начальное приближение
            x[i] = 0;
        MPI_MyRecv(loc_mat,  n * n / countOfProcess, "MPI_DOUBLE", countOfProcess - 1);
        MPI_MyRecv(b, n, "MPI_DOUBLE", countOfProcess - 1);
        MPI_MyRecv(send, n, "MPI_DOUBLE", countOfProcess - 1);
        MPI_MyRecv(diag, n, "MPI_DOUBLE", countOfProcess - 1);
        for(int j = socketRank * n / countOfProcess; j < n / countOfProcess + socketRank * n / countOfProcess; j++) // переносим
в вектор невязки те элементы, которые нужны процессу
            x[j] = send[j];

        for(int i = 0; i < n; i++) // обнуляем вектор сборки невязки
            send[i] = 0;
    }


    double norm; // создаём параметр нормы
    double sum; // создаём парамерт суммы
    double max; // максимальное значение нормы
    int iter; // количество итераций
    do // запускаем основной цикл метода Якоби
    {
        double x_l[n];
        if(socketRank == countOfProcess - 1) // для процесса 0
        {

            for(int i = 0; i < countOfProcess - 1; i++) // рассылаем обновлённый вектор невязки процесам
            {
                MPI_MySend(x, n, "MPI_DOUBLE", i);
            }
            for(int j = 0; j < socketRank * n / countOfProcess; j++) // обнуляем часть вектора неваязки, котоаря не нужна для
процесса,
                                            // чтобы при сборке результатов MPI_Reduce не накапливать ненужные значения
                x[j] = 0;
        }
        else
            MPI_MyRecv(send,  n, "MPI_DOUBLE", countOfProcess - 1); // получаем вектор невязки

        for(int i = 0; i < n; i++) // обнуляем вектор невязки в каждом потоке
            x[i] = 0;
        int p = socketRank * n / countOfProcess; // задаём параметр смещения для правильно вычисления вектора невязки
        for(int i = 0; i < n / countOfProcess; i++) // запускаем вычисления вектора невязки
        {
            sum = 0; // обнуляем сумму
            for(int j = 0; j < n; j++)
            {
                if ( loc_mat[i * n + j] != diag[i + p])  // получаем результат умножение матрицы на вектор,
                {
                    double e = loc_mat[i * n + j];
                    double s = send[j];
                    printf("");
                    sum += (e * s); // исключая диагональные элементы
                }
            }
```

```
            buf[i + p] = (b[i + p] - sum) / diag[i + p]; // получаем промежуточное значени вычитая сумму из вектора b и деля её на диагональный элемент

            norm = fabs(x[p] - buf[p]); // вычисяем норму вектора

            for(int h = p; h < p + n / countOfProcess; h++)
            {
                if(fabs(x[h] - buf[h]) > norm)
                    norm = fabs(x[h] - buf[h]);
                x[h] = buf[h]; // присваиваем новое значение вектору невязки
            }
        }
        for(int i = 0; i < n; i++) // обнуляем вектор сборки невязки
            send[i] = 0;

        MPI_MyReduse(&norm, &max, 1, "MPI_DOUBLE", "MPI_MAX", countOfProcess - 1); // получаем максимальное среди процессов значение нормы

        if (socketRank == countOfProcess - 1)
        {
            norm = max; // присваиваем значние максимума норм общей норме

            for (int i = 0; i < countOfProcess - 1; ++i) // и рассылваем всем процессам новую норму, чтобы они не выполнили лишних вычислений
                MPI_MySend(&max, 1, "MPI_DOUBLE", i);
        }
        else
             MPI_MyRecv(&norm,  1, "MPI_DOUBLE", countOfProcess - 1); // получаем вектор невязки

        MPI_MyReduse(x, send, n, "MPI_DOUBLE", "MPI_SUM", countOfProcess - 1); // собираем новый вектор невязки

        if(socketRank == countOfProcess - 1)
        {
            for(int i = 0; i < n; i++) // обновляем значение вектора невязки
                x[i] = send[i];
        }

        iter++; // увеличивавем число итераций
    }
    while(norm > eps && iter < MAX_ITER); // продолжаем вычисления, пока норма больше параметра приближения

    clock_t end = clock();
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;

    if(socketRank == countOfProcess - 1)
    {
        print_vector(b, n); // выводим вектор
        print_equation(a, n); // выводим матрицу
        for(int i = 0; i < n; i++)  // выводим результирующий веткор невязки
            printf("x[%d] = %0.9f \n", i, x[i]);
        printf("%d \n", iter); // выводим число итераций
        printf("Elapsed time: %0.9f\n", seconds); // выводим время выполнения
    }
    free(a);
    free(b);
    free(x);
    free(diag);
    free(buf);
    free(send);

    WSACleanup();

    return 0;
}
```