# Министерство образования и науки Российской Федерации

## Федеральное государственное бюджетное образовательное учреждение высшего образования

## «Новосибирский государственный технический университет»

NSTU_Logo_blue

## Кафедра параллельных вычислительных технологий

### Расчётно-графическая работа по дисциплине «Параллельное программирование»

|  |  |  |
| --- | --- | --- |
| сигма градиент синий1 | 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.

1. **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: О3 |

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: О3 |

1. 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:

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

,

где

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.

1. 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 |

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 |

1. Выводы
   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.

* 1. 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**

#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**

#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**

#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;

}