НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра параллельных вычислительных технологий

ОТЧЕТ ПРИНЯТ

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_\_ /

(подпись) (расшифровка)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_ г.

**ОТЧЕТ**

по расчетно – графической задаче

Реализация аналогов функций MPI\_Send и MPI\_Recv с помощью операций с сокетами

по дисциплине Параллельное программирование

Студент гр. ПМИ-02

Сидоров Даниил Игоревич

Новосибирск-2024

1. **Цель работы**

Реализация аналогов функций MPI\_Send и MPI\_Recv с помощью операций с сокетами. Реализовать задачу умножения матрицы на вектор, используя новые функции, сравнить с MPI-реализацией.

1. **Описание задачи**

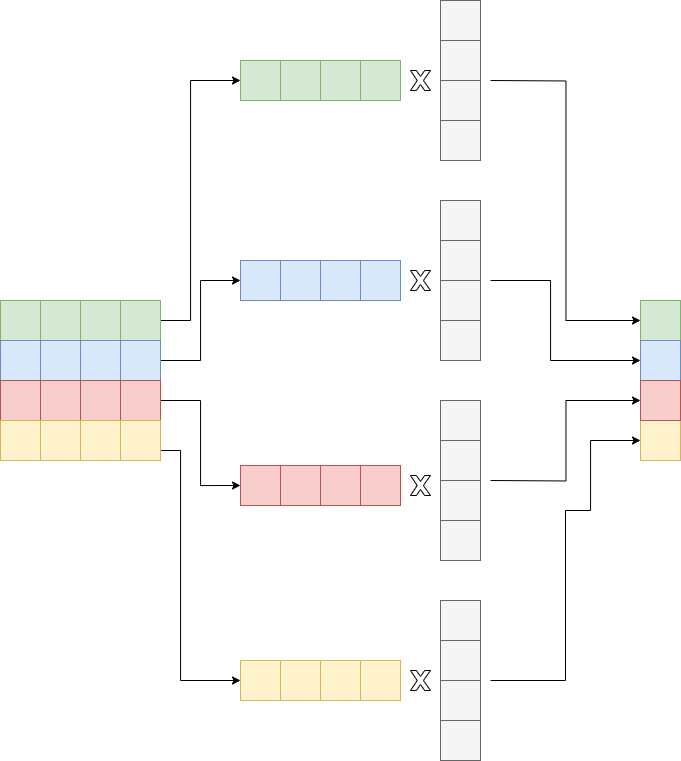
Для умножения матрицы на вектор в параллельном режиме воспользуемся ленточным разбиением матрицы. Реализуем две программы, одна будет использовать MPI, другая аналоги функций MPI\_Send и MPI\_Recv с помощью операций с сокетами.

Для сравнения двух реализаций, построим графики/таблицы ускорения и эффективности параллельной программы в зависимости от числа процессов, для разных размеров входных данных. Вычислим время коммуникации.

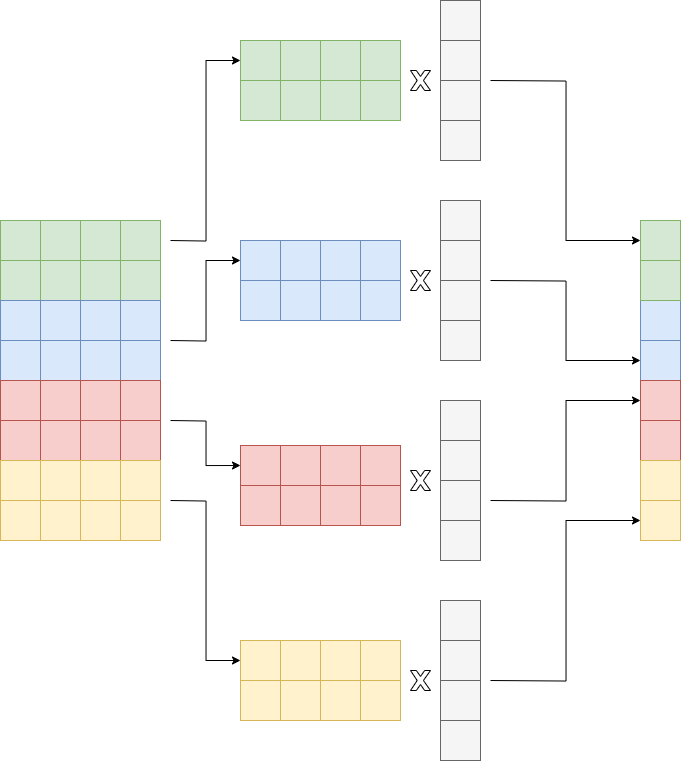
1. **Описание использованного алгоритма**

При ленточном разбиении каждому процессору выделяется то или иное подмножество строк (горизонтальное разбиение) или столбцов (вертикальное разбиение) матрицы. Разделение строк и столбцов на полосы в большинстве случаев происходит на непрерывной (последовательной) основе. Кроме части матрицы процессам необходимо передать вектор.

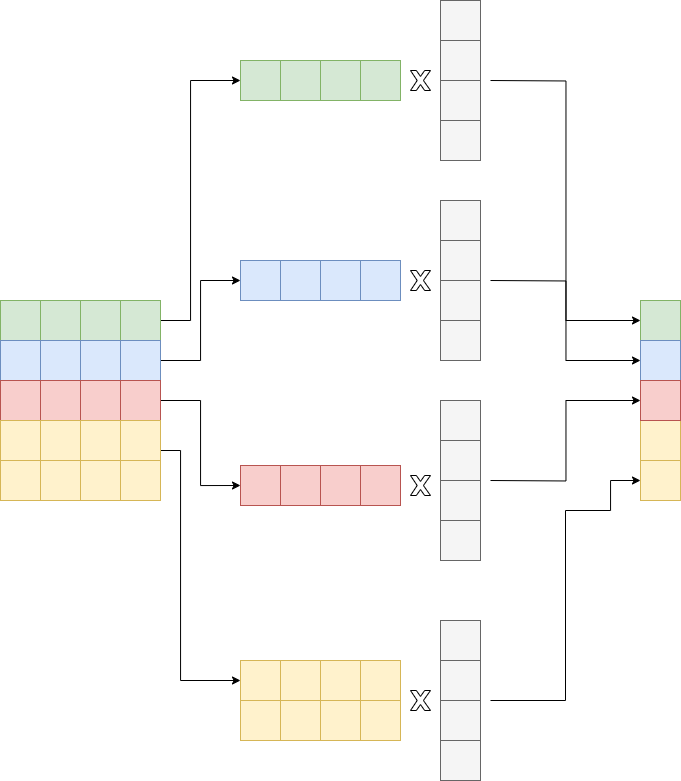
Пример №1 горизонтального разбиения матрицы в случае, когда количество строк равно количеству процессов:



Пример №2 горизонтального разбиения матрицы в случае, когда количество строк больше, но делится без остатка на количество процессов:



Пример №3 горизонтального разбиения матрицы в случае, когда количество строк больше, но не делится без остатка на количество процессов:



1. **Реализация аналогов функций MPI\_Send и MPI\_Recv**

Для реализации аналогов использовались функции winsock2.

MPI\_Send реализована функцией send, которая принимает следующие параметры:

* Дескриптор, определяющий подключенный сокет.
* Указатель на буфер, содержащий передаваемые данные.
* Длина (в байтах) данных в буфере, на которые указывает параметр buf.
* Набор флагов, указывающих способ вызова.

MPI\_Recv реализована функцией recv, которая принимает следующие параметры:

* Дескриптор, идентифицирующий подключенный сокет.
* Указатель на буфер для получения входящих данных.
* Длина (в байтах) буфера, на который указывает параметр buf.
* Набор флагов, влияющих на поведение этой функции.

1. **Параллельная реализация**

Опираясь на алгоритм умножения матрицы на вектор и горизонтального разбиения, выделим основные этапы программы:

1. **Создание матрицы и вектора**. Главный процесс создает матрицу и вектор.
2. **Разбиение матрицы**. Главный процесс с учетом количества процессов и размера матрицы определяет, сколько строк достанется каждому процессу. Возможны три варианта разбиения, которые были разобраны в пункте III.
3. **Отправка данных**. Главный процесс отправляет часть матрицы и вектор каждому процессу. Первую часть матрицы главный процесс оставляет себе.
4. **Ожидание получения**. Каждый процесс, исключая главный, ожидает получения данных от главного процесса.
5. **Расчет**. Каждый процесс, включая главный, производит умножение своей части матрицы на вектор.
6. **Отправка результата**. Процессы отправляют свои результаты расчета главного процессу.
7. **Прием результата**. Главный процесс получает и вставляет результаты других процессов в итоговый результирующий вектор, где уже хранится результат вычисления главного процесса. Вставка происходит согласно порядковому номеру процессов.

Отметим различие между двумя программами в начале работы и запуске программ. В реализации с использованием winsock2 требуется запустить несколько окон командной строки. Их количество равно количеству процессов. Первым запускается главный процесс, который инициирует работу сокетов.

1. **Результаты тестов**

**Использованная вычислительная система**

|  |  |
| --- | --- |
| Описание системы | |
| Аппаратная конфигурация | ЦП Intel Core i5-12500H @ 2.50GHz (4P и 8E ядер, 16 потоков, гипертрединг включен), ОЗУ 16 ГБ |
| Программная конфигурация | ОС Windows 11 |

**Время (в секундах) работы MPI при различных размерах задачи и количестве процессов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Количество процессов | | | |
| Размер матрицы | 1 | 2 | 3 | 4 |
|  | 0,288 | 0,242 | 0,209 | 0,206 |
|  | 1,182 | 0,962 | 0,847 | 0,786 |
|  | 2,697 | 2,186 | 1,97 | 1,854 |
|  | 3,081 | 2,655 | 2,266 | 2,211 |

**Время (в секундах) работы аналогов функций MPI при различных размерах задачи и количестве процессов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Количество процессов | | | |
| Размер матрицы | 1 | 2 | 3 | 4 |
|  | 0,343 | 0,311 | 0,255 | 0,231 |
|  | 1,34 | 1,128 | 1,007 | 0,995 |
|  | 3,125 | 2,751 | 2,449 | 2,368 |
|  | 3,814 | 3,296 | 3,034 | 2,838 |

**Ускорение и эффективность работы MPI при различных размерах задачи и количестве процессов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Число процессов | | | |
| Размер матрицы | Ускорение и эффективность | 1 | 2 | 3 | 4 |
|  |  | 1 | 1,190 | 1,378 | 1,398 |
|  | 1 | 0,595 | 0,459 | 0,350 |
|  |  | 1 | 1,229 | 1,396 | 1,504 |
|  | 1 | 0,614 | 0,465 | 0,376 |
|  |  | 1 | 1,234 | 1,369 | 1,455 |
|  | 1 | 0,617 | 0,456 | 0,364 |
|  |  | 1 | 1,160 | 1,360 | 1,393 |
|  | 1 | 0,580 | 0,453 | 0,348 |

**Ускорение и эффективность работы аналогов функций MPI при различных размерах задачи и количестве процессов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Число процессов | | | |
| Размер матрицы | Ускорение и эффективность | 1 | 2 | 3 | 4 |
|  |  | 1 | 1,103 | 1,345 | 1,485 |
|  | 1 | 0,551 | 0,448 | 0,371 |
|  |  | 1 | 1,188 | 1,331 | 1,347 |
|  | 1 | 0,594 | 0,444 | 0,337 |
|  |  | 1 | 1,136 | 1,276 | 1,320 |
|  | 1 | 0,568 | 0,425 | 0,330 |
|  |  | 1 | 1,157 | 1,257 | 1,344 |
|  | 1 | 0,579 | 0,419 | 0,336 |

**Ускорение работы MPI при различных размерах задачи и количестве процессов**

**Ускорение работы аналогов функций MPI при различных размерах задачи и количестве процессов**

**Эффективность работы MPI при различных размерах задачи и количестве процессов**

**Эффективность работы аналогов функций MPI при различных размерах задачи и количестве процессов**

Наблюдаем низкую эффективность. Для того, чтобы разобраться в причине этого, замерим время на коммуникации (вызов всех MPI функций и их аналогов) главного процесса. Посмотрим, как они меняются при разном числе процессов.

Возьмем размерность матрицы 15000 и проверим на функциях MPI:

|  |  |  |  |
| --- | --- | --- | --- |
| Число процессов | Общее время | Время на коммуникации | (общее время /Время на коммуникации) \* 100 |
| 2 | 2,17 | 0,33 | 15,207 % |
| 3 | 1,985 | 0,479 | 24,131 % |
| 4 | 1,852 | 0,534 | 28,834 % |

Возьмем размерность матрицы 15000 и проверим на аналогах функций MPI:

|  |  |  |  |
| --- | --- | --- | --- |
| Число процессов | Общее время | Время на коммуникации | (общее время /Время на коммуникации) \* 100 |
| 2 | 2,639 | 0,519 | 19,667 % |
| 3 | 2,45 | 0,715 | 29,184 % |
| 4 | 2,374 | 0,734 | 30,918 % |

Из представленных данных видно, что время, затраченное на коммуникации при увеличении числа процессов составляет все более значимую часть общего времени выполнения. С ростом числа процессов время расчетов падает, а время коммуникации растет, что объясняет слабый рост эффективности. Заметим, что аналоги функций MPI работают немного хуже оригинальных.

1. **Вывод**

Параллельная реализация может значительно ускорить вычисления, особенно для больших объемов данных. Однако, увеличение количества процессов не всегда приводит к пропорциональному уменьшению времени выполнения.

Проведенные тесты показывают, что вклад коммуникационного времени в общее время работы алгоритма увеличивается с увеличением числа процессов, что приводит к снижению эффективности параллельной программы.

Чтобы повысить эффективность параллельной программы возможно стоит попробовать асинхронные операции передачи данных или другие способы обеспечения параллелизма.

1. **Текст программы**

**MPI реализация:**

#include <mpi.h>

#include <cstdlib>

#include <vector>

#include <string>

#include <sstream>

#include <iostream>

#include <algorithm>

#include <time.h>

double communication = 0;

double fRand(double fMin, double fMax)

{

double f = (double)rand() / RAND\_MAX;

return fMin + f \* (fMax - fMin);

}

std::vector<double> make\_matrix(int size) {

std::vector<double> matrix(size \* size);

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

for (int j = 0; j < size; j++) {

matrix[i \* size + j] = fRand(0, 10000);

}

}

return matrix;

}

std::vector<double> make\_vector(int size) {

std::vector<double> vec(size);

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

vec[i] = fRand(0, 10000);

}

return vec;

}

void send\_str\_and\_vec(std::vector<double> matrix, std::vector<double> vec, int size, int matrix\_size) {

int rows\_per\_process = matrix\_size;

int remaining\_rows = 0;

if (size > 1)

{

rows\_per\_process = matrix\_size / size;

remaining\_rows = matrix\_size % size;

}

int current\_rows = rows\_per\_process;

int start\_row\_index = current\_rows \* matrix\_size;

for (int dest\_rank = 1; dest\_rank < size; dest\_rank++) {

if (dest\_rank == size - 1)

current\_rows += remaining\_rows;

clock\_t st = clock();

MPI\_Send(&matrix[start\_row\_index], current\_rows \* matrix\_size, MPI\_DOUBLE, dest\_rank, 123, MPI\_COMM\_WORLD);

MPI\_Send(&vec[0], vec.size(), MPI\_DOUBLE, dest\_rank, 456, MPI\_COMM\_WORLD);

clock\_t end = clock();

communication += (double)(end - st) / CLOCKS\_PER\_SEC;

start\_row\_index += current\_rows \* matrix\_size;

}

}

std::vector<double> recv\_string() {

MPI\_Status status;

MPI\_Probe(MPI\_ANY\_SOURCE, 123, MPI\_COMM\_WORLD, &status);

int src\_rank = status.MPI\_SOURCE;

int tag = status.MPI\_TAG;

int col\_size;

MPI\_Get\_count(&status, MPI\_DOUBLE, &col\_size);

std::vector<double> column(col\_size);

MPI\_Recv(&column[0], col\_size, MPI\_DOUBLE, src\_rank, tag, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

return column;

}

std::vector<double> recv\_vec() {

MPI\_Status status;

MPI\_Probe(MPI\_ANY\_SOURCE, 456, MPI\_COMM\_WORLD, &status);

int src\_rank = status.MPI\_SOURCE;

int tag = status.MPI\_TAG;

int vec\_size;

MPI\_Get\_count(&status, MPI\_DOUBLE, &vec\_size);

std::vector<double> vec(vec\_size);

MPI\_Recv(&vec[0], vec\_size, MPI\_DOUBLE, src\_rank, tag, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

return vec;

}

std::vector<double> recv\_res(int rank) {

MPI\_Status status;

MPI\_Probe(rank, 789, MPI\_COMM\_WORLD, &status);

int src\_rank = status.MPI\_SOURCE;

int tag = status.MPI\_TAG;

int res\_size;

MPI\_Get\_count(&status, MPI\_DOUBLE, &res\_size);

std::vector<double> res(res\_size);

MPI\_Recv(&res[0], res\_size, MPI\_DOUBLE, src\_rank, tag, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

return res;

}

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

MPI\_Init(&argc, &argv);

int rank, size;

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

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

int matrix\_size = atoi(argv[1]);

srand(time(NULL));

std::vector<double> matrix;

std::vector<double> vec;

std::vector<double> total\_result;

int rows\_per\_process = matrix\_size;

if (size > 1)

{

rows\_per\_process = matrix\_size / size;

}

std::vector<double> result(rows\_per\_process);

if (rank == 0) {

matrix = make\_matrix(matrix\_size);

vec = make\_vector(matrix\_size);

}

clock\_t start = clock();

if (rank == 0) {

send\_str\_and\_vec(matrix, vec, size, matrix\_size);

for (int i = 0; i < rows\_per\_process; ++i) {

for (int j = 0; j < matrix\_size; ++j) {

result[i] += matrix[i \* matrix\_size + j] \* vec[j];

}

}

total\_result.insert(total\_result.end(), result.begin(), result.end());

for (int src\_rank = 1; src\_rank < size; ++src\_rank) {

clock\_t st = clock();

result = recv\_res(src\_rank);

clock\_t end = clock();

communication += (double)(end - st) / CLOCKS\_PER\_SEC;

total\_result.insert(total\_result.end(), result.begin(), result.end());

}

}

else {

std::vector<double> column = recv\_string();

std::vector<double> vec = recv\_vec();

std::vector<double> result(column.size() / matrix\_size);

for (int i = 0; i < column.size() / matrix\_size; ++i) {

for (int j = 0; j < matrix\_size; ++j) {

result[i] += column[i \* matrix\_size + j] \* vec[j];

}

}

MPI\_Send(&result[0], result.size(), MPI\_DOUBLE, 0, 789, MPI\_COMM\_WORLD);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

clock\_t end = clock();

if (rank == 0)

{

double seconds = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("The time: %f seconds\n", seconds);

printf("The communication time: %f seconds\n", communication);

}

MPI\_Finalize();

return 0;

}

**Реализация c аналогами функций MPI:**

**socket.cpp:**

#include "mpi.h"

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

{

socketRank = atoi(argv[1]);

socketSize = atoi(argv[2]);

int matrix\_size = atoi(argv[3]);

srand(time(NULL));

std::vector<double> matrix(matrix\_size \* matrix\_size);

std::vector<double> vec(matrix\_size);

std::vector<double> total\_result;

int rows\_per\_process = matrix\_size;

int remaining\_rows = 0;

communication = 0;

if (socketRank == socketSize - 1)

{

matrix = make\_matrix(matrix\_size);

vec = make\_vector(matrix\_size);

}

if (socketSize > 1)

{

rows\_per\_process = matrix\_size / socketSize;

remaining\_rows = matrix\_size % socketSize;

Init();

}

std::vector<double> result(rows\_per\_process);

clock\_t start = clock();

if (socketRank == socketSize - 1)

{

send\_str\_and\_vec(matrix, vec, socketSize, matrix\_size);

for (int i = 0; i < rows\_per\_process; ++i) {

for (int j = 0; j < matrix\_size; ++j) {

result[i] += matrix[i \* matrix\_size + j] \* vec[j];

}

}

total\_result.insert(total\_result.end(), result.begin(), result.end());

for (int src\_rank = 0; src\_rank < socketSize - 1; ++src\_rank) {

if (src\_rank == socketSize - 2)

rows\_per\_process += remaining\_rows;

std::vector<double> result(rows\_per\_process);

clock\_t start = clock();

MPI\_MyRecv(result.data(), result.size(), "MPI\_DOUBLE", src\_rank);

clock\_t end = clock();

communication += (double)(end - start) / CLOCKS\_PER\_SEC;

total\_result.insert(total\_result.end(), result.begin(), result.end());

}

}

else

{

if (socketRank == socketSize - 2)

rows\_per\_process += remaining\_rows;

std::vector<double> column(rows\_per\_process \* matrix\_size);

std::vector<double> result(rows\_per\_process);

MPI\_MyRecv(column.data(), column.size(), "MPI\_DOUBLE", socketSize - 1);

MPI\_MyRecv(vec.data(), vec.size(), "MPI\_DOUBLE", socketSize - 1);

for (int i = 0; i < column.size() / matrix\_size; ++i) {

for (int j = 0; j < matrix\_size; ++j) {

result[i] += column[i \* matrix\_size + j] \* vec[j];

}

}

MPI\_MySend(result.data(), result.size(), "MPI\_DOUBLE", socketSize - 1);

}

clock\_t end = clock();

if (socketRank == socketSize - 1)

{

double seconds = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("The time: %f seconds\n", seconds);

printf("The communication time: %f seconds\n", communication);

}

WSACleanup();

return 0;

}

**mpi.cpp:**

#include "mpi.h"

std::vector<SOCKET> sockets;

int socketRank;

int socketSize;

HOSTENT\* hostent;

double communication;

void Init()

{

int start\_port = 8080;

WORD version = MAKEWORD(2, 2);

WSADATA wsaData;

typedef unsigned long IPNumber;

WSAStartup(version, (LPWSADATA)&wsaData);

std::vector<SOCKADDR\_IN> servers(socketSize);

sockets.resize(socketSize);

// 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 == socketSize - 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 \n", temp[0]);

sockets[temp[0]] = buf;

}

for (int i = socketRank + 1; i < socketSize; 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);

if (retVal == SOCKET\_ERROR)

{

std::cout << "unable to recv" << std::endl;

int error = WSAGetLastError();

printf("%d\n", error);

return;

}

}

int flag = socketSize - 1;

int def = 1;

if (socketRank == socketSize - 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;

}

void MPI\_MySend(void\* buf, int count, std::string type, int i)

{

int size\_;

if (type == "MPI\_INT")

size\_ = count \* sizeof(int);

if (type == "MPI\_DOUBLE")

size\_ = count \* sizeof(double);

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;

}

}

void MPI\_MyRecv(void\* buf, int count, std::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;

}

}

std::vector<double> make\_matrix(int size) {

std::vector<double> matrix(size \* size);

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

for (int j = 0; j < size; j++) {

matrix[i \* size + j] = fRand(0, 10000);

}

}

return matrix;

}

std::vector<double> make\_vector(int size) {

std::vector<double> vec(size);

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

vec[i] = fRand(0, 10000);

}

return vec;

}

void send\_str\_and\_vec(std::vector<double> matrix, std::vector<double> vec, int size, int matrix\_size) {

int rows\_per\_process = matrix\_size;

int remaining\_rows = 0;

if (size > 1)

{

rows\_per\_process = matrix\_size / size;

remaining\_rows = matrix\_size % size;

}

int current\_rows = rows\_per\_process;

int start\_row\_index = current\_rows \* matrix\_size;

for (int dest\_rank = 0; dest\_rank < size - 1; dest\_rank++) {

if (dest\_rank == size - 2)

{

current\_rows += remaining\_rows;

}

clock\_t start = clock();

MPI\_MySend(&matrix[start\_row\_index], current\_rows \* matrix\_size, "MPI\_DOUBLE", dest\_rank);

MPI\_MySend(vec.data(), vec.size(), "MPI\_DOUBLE", dest\_rank);

clock\_t end = clock();

communication += (double)(end - start) / CLOCKS\_PER\_SEC;

start\_row\_index += current\_rows \* matrix\_size;

}

}

double fRand(double fMin, double fMax)

{

double f = (double)rand() / RAND\_MAX;

return fMin + f \* (fMax - fMin);

}