|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
| Расчетно-графическая работа | | | |
| по дисциплине «Параллельное программирование» | | | |
| **ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВУЮЩИХ ПОТОКОВ** | | | |
|  | | | |
|  | Бригада 6 | ПМИ-71 Антонов С. |
|  | ПМИ-71 Арнольд э. |
|  | ПМИ-71 Кайль Д. |
|  |
|  |
| Преподаватель | Щукин Г. А.  Городничев М. А. |
|  | | | |
| Новосибирск | | | |

**Задание**

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

**Программа**

Makefile

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE -DMYMPI

CFLAGS  = -std=c99 -O2 -g $(DEFINES)

LIBS    = -lm

TARGET  = mympi

all: $(TARGET) cleanTemp

demo: DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE -DMYMPI -DRESULT

demo: all

mpi: $(TARGET)\_main.c

    mpicc $(TARGET)\_main.c -o mpi\_version

mpi\_demo: $(TARGET)\_main.c

    mpicc $(TARGET)\_main.c -DRESULT -o mpi\_version

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    gcc $(CFLAGS) -o $(TARGET)\_version $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET)\_main.c

$(TARGET).o: $(TARGET).c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET).c

cleanTemp:

    rm -rf \*.o

clean:

    rm -rf $(TARGET) $(TARGET)\_version mpi\_version

mympi.c

#include "mympi.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

/\* myMPI \*/

#define ARGS\_COUNT 5

#define ROOT\_RANK  0

/\* Server \*/

#define EMIT\_SIZE    10000

#define CLIENT\_COUNT 128        // Размер очереди подключений

#define SOCK\_ERR -1             // Код ошибки

#define SOCK\_PASS 0             // Код успеха

/\* Функция отлова ошибок \*/

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

/\* Тип отправленных/принятых данных \*/

enum DataType {

    DT\_MSG,                     // Сообщение

    DT\_BLOCK                    // Блокировка

};

/\* Заголовок отправленных/принятых данных \*/

typedef struct DataHeader {

    int src;                    // Номер процесса отправителя

    int tag;

    uint8\_t type;               // Тип заголовка

} DataHeader;

/\* Данные серверной части процесса \*/

struct MyMPIData {

    int\* send\_socks;

    int\* recv\_socks;

    int lock\_size;              // ROOT\_RANK данные:

};                              // количество блокировок

/\* Данные процесса \*/

struct MyMPIComm {

    const char\* hostname;       // Адрес

    uint16\_t port;              // Порт

    int rank;                   // Номер процесса

    int size;                   // Кол-во процессов

};

/\* Отправка данных на сокет \*/

static void dataSend(int sock, void\* data, size\_t data\_size) {

    if (data\_size) {

        const char\* data\_ptr = (const char\*)data;

        // Проверка данных на целостность

        size\_t send\_size = 0;

        while (send\_size < data\_size) {

            int bytes\_send = send(sock, data\_ptr, data\_size - send\_size, 0);

            if (bytes\_send == SOCK\_ERR)

                throwErr("Error: send data to dest!");

            data\_ptr += bytes\_send;

            send\_size += bytes\_send;

        }

    }

}

/\* Приём данных на сокет \*/

static void dataRecv(int sock, void\* data, size\_t data\_size) {

    if (data\_size) {

        char\* data\_ptr = (char\*)data;

        // Проверка данных на целостность

        size\_t recv\_size = 0;

        while (recv\_size < data\_size) {

            int bytes\_recv = recv(sock, data\_ptr, data\_size - recv\_size, 0);

            if (bytes\_recv == SOCK\_ERR)

                throwErr("Error: recv data from src!");

            data\_ptr += bytes\_recv;

            recv\_size += bytes\_recv;

        }

    }

}

/\* Отправка больших данных на сокет \*/

static void largeDataSend(int sock, void\* data, size\_t data\_size) {

    char\* data\_ptr = (char\*)data;

    if (EMIT\_SIZE < data\_size) {

        size\_t i;

        for (i = 0; i < data\_size - EMIT\_SIZE; i += EMIT\_SIZE)

            dataSend(sock, data\_ptr + i, EMIT\_SIZE);

        data\_ptr = data\_ptr + i;

        data\_size = data\_size % EMIT\_SIZE;;

    }

    dataSend(sock, data\_ptr, data\_size);

}

/\* Приём больших данных на сокет \*/

static void largeDataRecv(int sock, void\* data, size\_t data\_size) {

    char\* data\_ptr = (char\*)data;

    if (EMIT\_SIZE < data\_size) {

        size\_t i;

        for (i = 0; i < data\_size - EMIT\_SIZE; i += EMIT\_SIZE)

            dataRecv(sock, data\_ptr + i, EMIT\_SIZE);

        data\_ptr = data\_ptr + i;

        data\_size = data\_size % EMIT\_SIZE;;

    }

    dataRecv(sock, data\_ptr, data\_size);

}

/\* Инициализация сокетов для отправки сообщений \*/

static void sendSocksInit() {

    mympi\_data->send\_socks = (int\*)malloc(sizeof(int) \* mympi\_comm->size);

    if (mympi\_data->send\_socks == NULL)

        throwErr("Error: send socks out of memmory!");

    for (uint8\_t i = 0; i != mympi\_comm->size; ++i) {

        mympi\_data->send\_socks[i] = socket(AF\_INET, SOCK\_STREAM, 0);

        if (mympi\_data->send\_socks[i] == SOCK\_ERR)

            throwErr("Error: send socket open faild!");

        struct hostent\* hosten = gethostbyname(mympi\_comm->hostname);

        struct sockaddr\_in send\_addr;

        send\_addr.sin\_family = AF\_INET;

        send\_addr.sin\_addr.s\_addr = ((struct in\_addr\*)hosten->h\_addr\_list[0])->s\_addr;

        send\_addr.sin\_port = htons(mympi\_comm->port + i);

        if (i == mympi\_comm->rank) {

            int err = 0;

            int on = 1;

            err = setsockopt(mympi\_data->send\_socks[i], SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, (int\*)&on, sizeof(int));

            if (err == SOCK\_ERR)

                throwErr("Error: setsockopt failed!");

            err = bind(mympi\_data->send\_socks[i], (struct sockaddr\*)&send\_addr, sizeof(send\_addr));

            if (err == SOCK\_ERR)

                throwErr("Error: bind serv socket!");

            err = listen(mympi\_data->send\_socks[i], CLIENT\_COUNT);

            if (err == SOCK\_ERR)

                throwErr("Error: listen serv socket!");

        }

        else {

            while (connect(mympi\_data->send\_socks[i], (struct sockaddr\*)&send\_addr, sizeof(send\_addr)) != SOCK\_PASS);

            dataSend(mympi\_data->send\_socks[i], (void\*)&mympi\_comm->rank, sizeof(int));

        }

    }

}

/\* Инициализация сокетов для приёма \*/

static void recvSocksInit() {

    mympi\_data->recv\_socks = (int\*)malloc(sizeof(int) \* mympi\_comm->size);

    if (mympi\_data->recv\_socks == NULL)

        throwErr("Error: recv socks out of memmory!");

    for (uint8\_t i = 0; i != mympi\_comm->size - 1; ++i) {

        struct sockaddr\_in recv\_addr;

        socklen\_t recv\_len = sizeof(recv\_addr);

        int recv\_sock = 0;

        do {

            recv\_sock = accept(mympi\_data->send\_socks[mympi\_comm->rank], (struct sockaddr\*)&recv\_addr, (socklen\_t\*)&recv\_len);

            if (recv\_sock < SOCK\_ERR)

                throwErr("Error: recv sock accept!");

        } while (recv\_sock == SOCK\_ERR);

        int recv\_rank;

        dataRecv(recv\_sock, (void\*)&recv\_rank, sizeof(int));

        mympi\_data->recv\_socks[recv\_rank] = recv\_sock;

    }

}

/\* Инициализация работы myMPI \*/

void myMPIInit(int\* argc, char\*\* argv[]) {

    if (\*argc < ARGS\_COUNT)

        throwErr("Wrong number of arguments!\n"

                 "Enter: <hostname> <port> <rank> <number of processes>\n");

    mympi\_data = (struct MyMPIData\*)malloc(sizeof(struct MyMPIData));

    if (mympi\_data == NULL)

        throwErr("Error: mympi\_data out of memmory!");

    mympi\_comm = (struct MyMPIComm\*)malloc(sizeof(struct MyMPIComm));

    if (mympi\_comm == NULL)

        throwErr("Error: mympi\_comm out of memmory!");

    // Инициализация данных процесса

    mympi\_comm->hostname =  (\*argv)[1];

    mympi\_comm->port = atoi((\*argv)[2]);

    mympi\_comm->rank = atoi((\*argv)[3]);

    mympi\_comm->size = atoi((\*argv)[4]);

    // Очиста myMPI аргументов командой строки (сдвиг аргументов программы на их место)

    (\*argc) -= ARGS\_COUNT - 1;

    for (uint8\_t i = 1; i != \*argc; ++i)

        (\*argv)[i] = (\*argv)[i + ARGS\_COUNT - 1];

    // Инициализация данных серверной части

    mympi\_data->lock\_size = 0;

    sendSocksInit();

    recvSocksInit();

    myMPIBarrier();

}

/\* Отправка статуса блокировки указанному процессу \*/

static void lockSend(int dest) {

    if (dest == mympi\_comm->rank)

        throwErr("Error: send lock to yourself!");

    // Посылаем блокирующее сообщение

    DataHeader data\_header = (DataHeader){-1, -1, DT\_BLOCK};

    dataSend(mympi\_data->send\_socks[dest], (void\*)&data\_header, sizeof(DataHeader));

}

/\* Приём блокировок от других процессов \*/

static void lockRecv(int src) {

    if (src == mympi\_comm->rank)

        throwErr("Error: recv lock from yourself!");

    DataHeader data\_header;

    // Принимаем данные, пока не поступит блокирующее сообщение

    do {

        dataRecv(mympi\_data->recv\_socks[src], (void\*)&data\_header, sizeof(DataHeader));

    } while (data\_header.type != DT\_BLOCK);

}

/\* Добавить блокировку (увеличить общее количество блокировок) \*/

static void lockAdd() {

    ++mympi\_data->lock\_size;

}

/\* Очистить блокировки \*/

static void lockClear() {

    mympi\_data->lock\_size = 0;

}

/\* Барьерная синхронизация потоков \*/

void myMPIBarrier() {

    // Если поток основной

    if (mympi\_comm->rank == ROOT\_RANK) {

        // Принимаем блокировки от других потоков,

        // пока все, другие потоки, не будут заблокированы

        for (uint8\_t i = 0; i != mympi\_comm->size; ++i)

            if (i != ROOT\_RANK) {

                lockRecv(i);

                lockAdd();

            }

        // Когда все потоки заблокированы -- разблокируем их

        for (uint8\_t i = 0; i != mympi\_comm->size; ++i)

            if (i != ROOT\_RANK)

                lockSend(i);

    }

    else {

        // Если поток не основной --

        // посылаем сообщение о блокировке основному потоку

        lockSend(ROOT\_RANK);

        // Ждём сигнал для разблокировки от основного потока

        lockRecv(ROOT\_RANK);

    }

    // Очищаем блокировки

    lockClear();

}

/\* Завершение работы myMPI \*/

void myMPIFinalize() {

    myMPIBarrier();

    for (uint8\_t i = 0; i != mympi\_comm->size; ++i) {

        close(mympi\_data->send\_socks[i]);

        close(mympi\_data->recv\_socks[i]);

    }

    free(mympi\_data->send\_socks);

    free(mympi\_data->recv\_socks);

    free(mympi\_data);

    free(mympi\_comm);

}

/\* Получить номер процесса \*/

void myMPICommRank(int\* rank) {

    if (rank == NULL)

        throwErr("Error: rank null ptr!");

    \*rank = mympi\_comm->rank;

}

/\* Получить общее число процессов \*/

void myMPICommSize(int\* size) {

    if (size == NULL)

        throwErr("Error: size null ptr!");

    \*size = mympi\_comm->size;

}

/\* Получить порт \*/

void myMPICommPort(uint16\_t\* port) {

    if (port == NULL)

        throwErr("Error: port null ptr!");

    \*port = mympi\_comm->port + mympi\_comm->rank;

}

/\* Отправка данных процессу \*/

void myMPISend(void\* data, size\_t count, size\_t data\_type, int dest, int tag) {

    if (dest == mympi\_comm->rank)

        throwErr("Error: recv data from yourself!");

    DataHeader data\_header = (DataHeader){mympi\_comm->rank, tag, DT\_MSG};

    dataSend(mympi\_data->send\_socks[dest], (void\*)&data\_header, sizeof(DataHeader));

    largeDataSend(mympi\_data->send\_socks[dest], data, count \* data\_type);

}

/\* Приём данных процесса \*/

void myMPIRecv(void\* data, size\_t count, size\_t data\_type, int src, int tag) {

    if (src == mympi\_comm->rank)

        throwErr("Error: recv data from yourself!");

    DataHeader data\_header;

    // Ожидаем сообщение, пока номер и тэг сообщения не соответствует указанным

    do {

        dataRecv(mympi\_data->recv\_socks[src], (void\*)&data\_header, sizeof(DataHeader));

        // Проверяем тип сообщения. Если получена блокировка -- запоминаем её, для будущей проверки

        switch (data\_header.type) {

            case DT\_MSG:   largeDataRecv(mympi\_data->recv\_socks[src], data, count \* data\_type); break;

            case DT\_BLOCK: lockAdd();                                                           break;

            default:       throwErr("Error: unrecognized data type!");

        }

    } while (data\_header.src != src || data\_header.tag != tag);

}

Mympi.h

#ifndef MYMPI\_H

#define MYMPI\_H

#include <stdint.h>

#include <stddef.h>

/\* myMPI данные (данные одного процесса) \*/

struct MyMPIData\* mympi\_data;

struct MyMPIComm\* mympi\_comm;

void myMPIInit(int\* argc, char\*\* argv[]);

void myMPIBarrier();

void myMPIFinalize();

void myMPICommRank(int\* rank);

void myMPICommSize(int\* size);

void myMPICommPort(uint16\_t\* port);

void myMPISend(void\* data, size\_t count, size\_t data\_type, int dest, int tag);

void myMPIRecv(void\* data, size\_t count, size\_t data\_type, int src, int tag);

#endif

mympi\_main.c

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <time.h>

#ifdef MYMPI

#include "mympi.h"

#else

#include <mpi.h>

#endif

#define ARGS\_COUNT 3

// Номер главного процесса

#define ROOT\_RANK 0

#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

// Функция отлова ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

// Доступ к элементам матрицы

#define matAccess(mat, m, i, j) \

    ((mat)[(i) \* (m) + (j)])

// Создание массива длины size

static double\* arrayCreate(size\_t size) {

    double\* A = (double\*)malloc(sizeof(double) \* size);

    if (A == NULL)

        throwErr("Error: array out of memmory!");

    return A;

}

// Инициализация массива случайными числами

static void arrayRandInit(double\* A, size\_t size) {

    for (size\_t i = 0; i < size; ++i)

        A[i] = rand() % size;

}

// Функция умножения матрицы на вектор

static void matVecMult(double\* res, double\* mat, double\* vec, size\_t n, size\_t m) {

    if (res == NULL)

        throwErr("Error: res null ptr!");

    for (size\_t i = 0; i < n; ++i) {

        res[i] = 0;

        for (size\_t j = 0; j < m; ++j)

            res[i] += matAccess(mat, m, i, j) \* vec[j];

    }

}

int main(int argc, char\* argv[]) {

    srand(time(NULL));

    int rank = 0;

    int size = 1;

    // Инициализация MPI

    #ifdef MYMPI

    myMPIInit(&argc, &argv);

    myMPICommRank(&rank);

    myMPICommSize(&size);

    #else

    MPI\_Status status;

    MPI\_Init(&argc, &argv);

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

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

    #endif

    // Проверка входных аргументон

    if (argc < ARGS\_COUNT) {

        if (rank == ROOT\_RANK) {

            fprintf(stderr, "Wrong number of arguments!\n");

            fprintf(stderr, "Enter: <n> <m>\n");

        }

        #ifdef MYMPI

        myMPIFinalize();

        #else

        MPI\_Finalize();

        #endif

        exit(EXIT\_FAILURE);

    }

    // Размер задачи

    size\_t root\_n = atoi(argv[1]);

    size\_t root\_m = atoi(argv[2]);

    // Исходная матрица

    double\* root\_mat = NULL;

    // Размер обрабатываемого блока процессом

    size\_t n = 0;

    size\_t m = root\_m;

    // Данные процессов:

    // Часть исходной матрицы (Исходня матрица для ROOT\_RANK)

    double\* mat = NULL;

    // Исходный вектор

    double\* vec = NULL;

    // Часть результата (Конечный результат для ROOT\_RANK)

    double\* res = NULL;

    // Генерация данных (матрица m \* n, вектор m)

    if (rank == ROOT\_RANK) {

        root\_mat = arrayCreate(root\_n \* root\_m);

        arrayRandInit(root\_mat, root\_n \* root\_m);

        vec = arrayCreate(root\_m);

        arrayRandInit(vec, root\_m);

    }

    // Синхронизация процессов перед началом замера времени

    #ifdef MYMPI

    myMPIBarrier();

    #else

    MPI\_Barrier(MPI\_COMM\_WORLD);

    #endif

    struct timespec start, stop;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    if (rank == ROOT\_RANK) {

        // Если процесс главный --

        // Отправить вектор и его размер другим процессам

        for (uint8\_t i = 1; i < size; ++i) {

             #ifdef MYMPI

            myMPISend(&root\_m, 1, sizeof(size\_t), i, 0);

            myMPISend(vec, root\_m, sizeof(double), i, 0);

            #else

            MPI\_Send(&root\_m, 1, MPI\_UNSIGNED\_LONG\_LONG, i, 0, MPI\_COMM\_WORLD);

            MPI\_Send(vec, root\_m, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

            #endif

        }

        // Определение размеров блоков данных (строк для обработки)

        size\_t chunk\_size = root\_n / size;

        uint8\_t remainder = root\_n % size;

        size\_t shift = 0;

        for (uint8\_t i = 0; i < size; ++i) {

            size\_t row\_size = chunk\_size + (i < remainder ? 1 : 0);

            // Распределение строк матрицы между процессами

            if (i != ROOT\_RANK) {

                #ifdef MYMPI

                myMPISend(&row\_size, 1, sizeof(size\_t), i, 0);

                myMPISend(mat + shift, row\_size \* root\_m, sizeof(double), i, 0);

                #else

                MPI\_Send(&row\_size, 1, MPI\_UNSIGNED\_LONG\_LONG, i, 0, MPI\_COMM\_WORLD);

                MPI\_Send(mat + shift, row\_size \* root\_m, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

                #endif

            }

            else {

                // Если процесс главный -- обрабатывает часть исходной матрицы

                n = row\_size;

                mat = root\_mat;

            }

            shift += row\_size \* root\_m;

        }

    }

    else {

        // Если процесс не главный --

        // Принимает вектор и его размер

        // Принимает матрицу и её размер

        #ifdef MYMPI

        myMPIRecv(&m, 1, sizeof(size\_t), ROOT\_RANK, 0);

        vec = arrayCreate(m);

        myMPIRecv(vec, m, sizeof(double), ROOT\_RANK, 0);

        myMPIRecv(&n, 1, sizeof(size\_t), ROOT\_RANK, 0);

        mat = arrayCreate(n \* m);

        myMPIRecv(mat, n \* m, sizeof(double), ROOT\_RANK, 0);

        #else

        MPI\_Recv(&m, 1, MPI\_UNSIGNED\_LONG\_LONG, ROOT\_RANK, 0, MPI\_COMM\_WORLD, &status);

        vec = arrayCreate(m);

        MPI\_Recv(vec, m, MPI\_DOUBLE, ROOT\_RANK, 0, MPI\_COMM\_WORLD, &status);

        MPI\_Recv(&n, 1, MPI\_UNSIGNED\_LONG\_LONG, ROOT\_RANK, 0, MPI\_COMM\_WORLD, &status);

        mat = arrayCreate(n \* m);

        MPI\_Recv(mat, n \* m, MPI\_DOUBLE, ROOT\_RANK, 0, MPI\_COMM\_WORLD, &status);

        #endif

        root\_n = n;

    }

    // Умножение полученных строк матрицы на вектор. Получение локального результат

    res = arrayCreate(root\_n);

    matVecMult(res, mat, vec, n, m);

    if (rank == ROOT\_RANK) {

        // Если процесс главный --

        // Принимаем локальные результаты других процессов

        // Добавляем результы к результату ROOT\_RANK процесса

        size\_t shift = n;

        for (uint8\_t i = 1; i < size; ++i) {

            #ifdef MYMPI

            myMPIRecv(&n, 1, sizeof(size\_t), i, 0);

            myMPIRecv(res + shift, n, sizeof(double), i, 0);

            #else

            MPI\_Recv(&n, 1, MPI\_UNSIGNED\_LONG\_LONG, i, 0, MPI\_COMM\_WORLD, &status);

            MPI\_Recv(res + shift, n, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD, &status);

            #endif

            shift += n;

        }

    }

    else {

        // Если процесс не главный --

        // Посылаем локальный результат главному процессу

        #ifdef MYMPI

        myMPISend(&n, 1, sizeof(size\_t), ROOT\_RANK, 0);

        myMPISend(res, n, sizeof(double), ROOT\_RANK, 0);

        #else

        MPI\_Send(&n, 1, MPI\_UNSIGNED\_LONG\_LONG, ROOT\_RANK, 0, MPI\_COMM\_WORLD);

        MPI\_Send(res, n, MPI\_DOUBLE, ROOT\_RANK, 0, MPI\_COMM\_WORLD);

        #endif

    }

    // Ждём все процессы перед замером времени

    #ifdef MYMPI

    myMPIBarrier();

    #else

    MPI\_Barrier(MPI\_COMM\_WORLD);

    #endif

    clock\_gettime(CLOCK\_MONOTONIC, &stop);

    // Выводим результат работы

    if (rank == ROOT\_RANK) {

        #ifdef RESULT

        printf("matrix:\n");

        for (size\_t i = 0; i < root\_n; ++i) {

            for (size\_t j = 0; j < root\_m; ++j)

                printf("%g ", matAccess(root\_mat, root\_m, i, j));

            printf("\n");

        }

        printf("vector:\n");

        for (size\_t i = 0; i < m; ++i)

            printf("%g ", vec[i]);

        printf("\n");

        printf("result:\n");

        for (size\_t i = 0; i < root\_n; ++i)

            printf("%g ", res[i]);

        printf("\n");

        #endif

        printf("Elapsed time: %lf\n", clocktimeDifference(start, stop));

    }

    free(mat);

    free(vec);

    free(res);

    // Очищаем состояние MPI

    #ifdef MYMPI

    myMPIFinalize();

    #else

    MPI\_Finalize();

    #endif

    return 0;

}

Mympirun

#!/bin/bash

ROOT\_RANK=0

HOSTNAME="localhost"

PORT=8080

SIZE=1

POSITIONAL=()

while [[ $# -gt 0 ]]; do

    key="$1"

    case $key in

        -h|-host)

            HOSTNAME="$2"

            shift 2

        ;;

        -p|-port)

            PORT="$2"

            shift 2

        ;;

        -n|-np)

            SIZE="$2"

            shift 2

        ;;

        \*)

            POSITIONAL+=("$1")

            shift

        ;;

    esac

done

set -- "${POSITIONAL[@]}"

if [ -z $1 ]; then

    echo -e "mympirun a.out\n"

    echo "Arguments:"

    echo "-h <host>"

    echo "    - Name of host on which to run processes"

    echo "-p <port>"

    echo "    - Port on which to run processes"

    echo "-n <np>"

    echo "    - Specify the number of processes to use"

else

    for (( i=1; i < $SIZE; ++i )); do

        ./$1 $HOSTNAME $PORT $i $SIZE "${@:2}" &

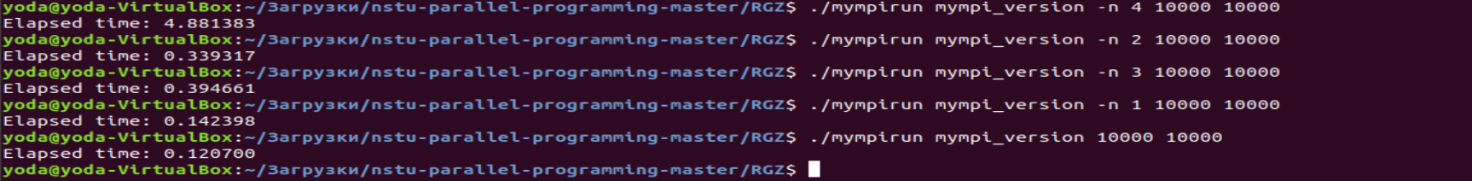
    done

    ./$1 $HOSTNAME $PORT $ROOT\_RANK $SIZE "${@:2}"

fi

Результат работы программы:

Mympi



Mpi

