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

I. 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.

II. Описание задачи

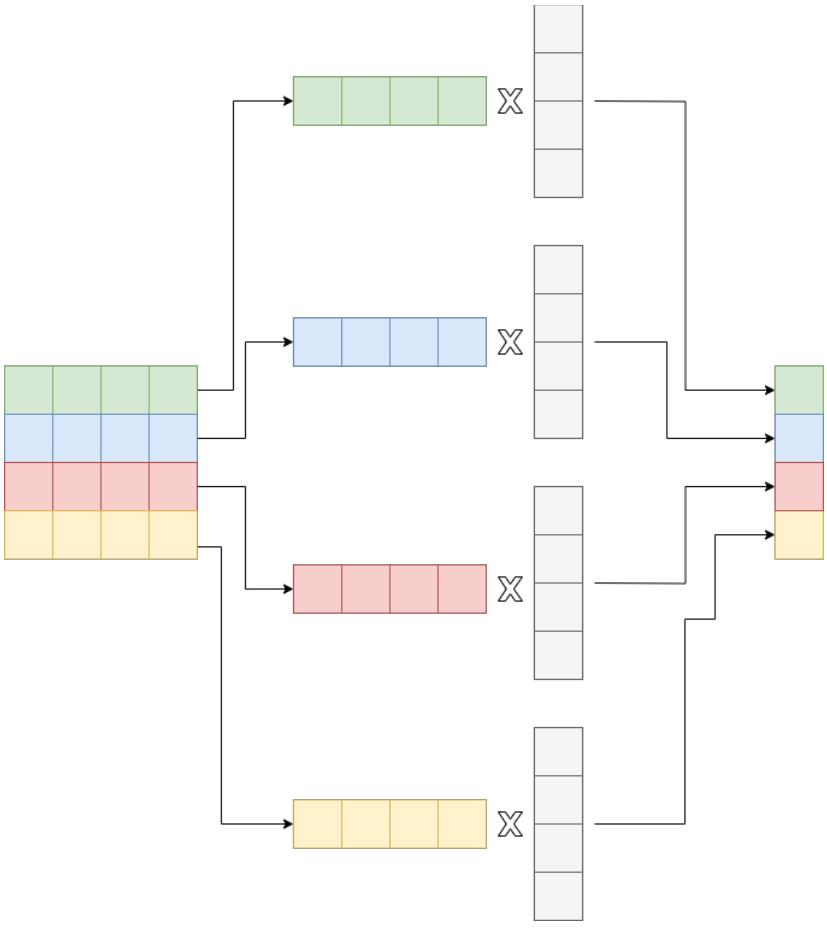
Для умножения матрицы на вектор в параллельном режиме воспользуемся ленточным разбиением матрицы. Реализуем две программы, одна будет использовать MPI, другая аналоги функций MPI_Send и MPI_Recv с помощью операций с сокетами.

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.

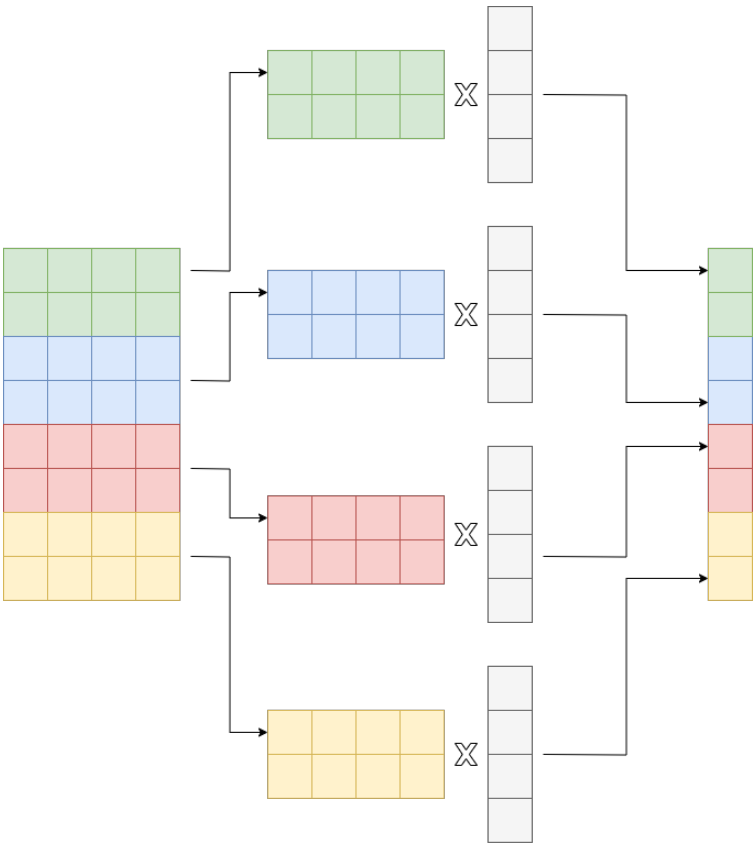
III. Описание использованного алгоритма

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.

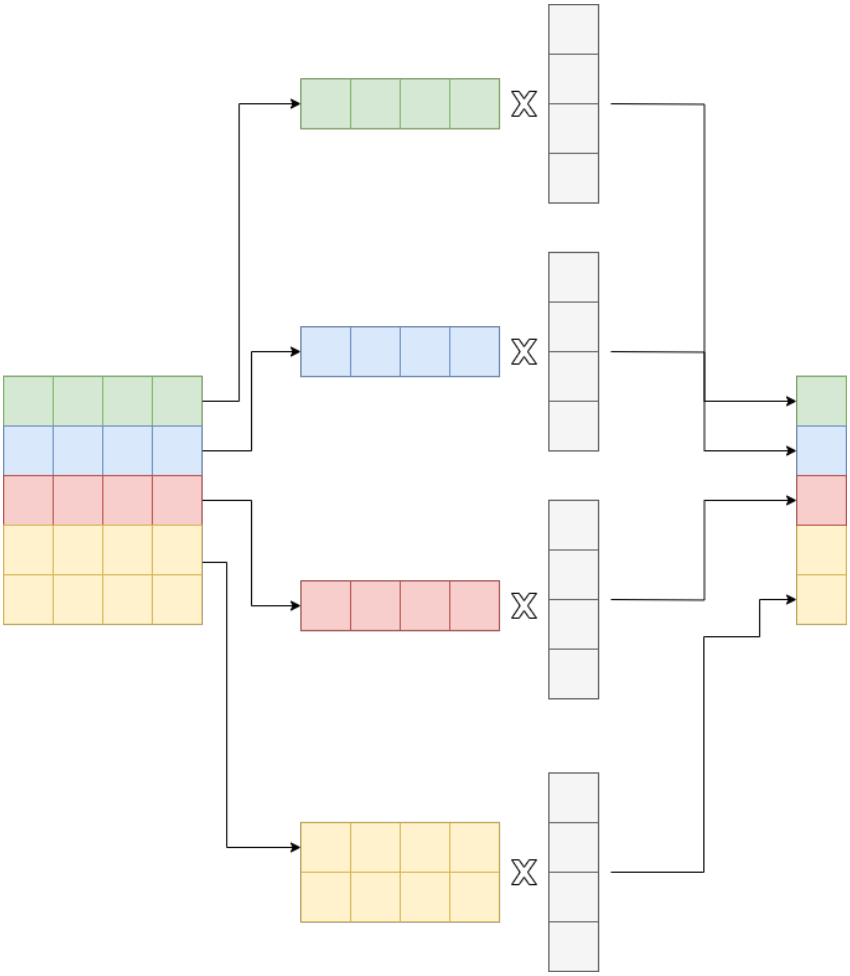
Пример №1 горизонтального разбиения матрицы в случае, когда количество строк равно количеству процессов:



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:



IV. 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.

V. 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.

VI. 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
5000	0,288	0,242	0,209	0,206
10000	1,182	0,962	0,847	0,786
15000	2,697	2,186	1,97	1,854
16000	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
5000	0,343	0,311	0,255	0,231
10000	1,34	1,128	1,007	0,995
15000	3,125	2,751	2,449	2,368
16000	3,814	3,296	3,034	2,838

Ускорение S и эффективность E работы MPI при различных размерах задачи и количестве процессов

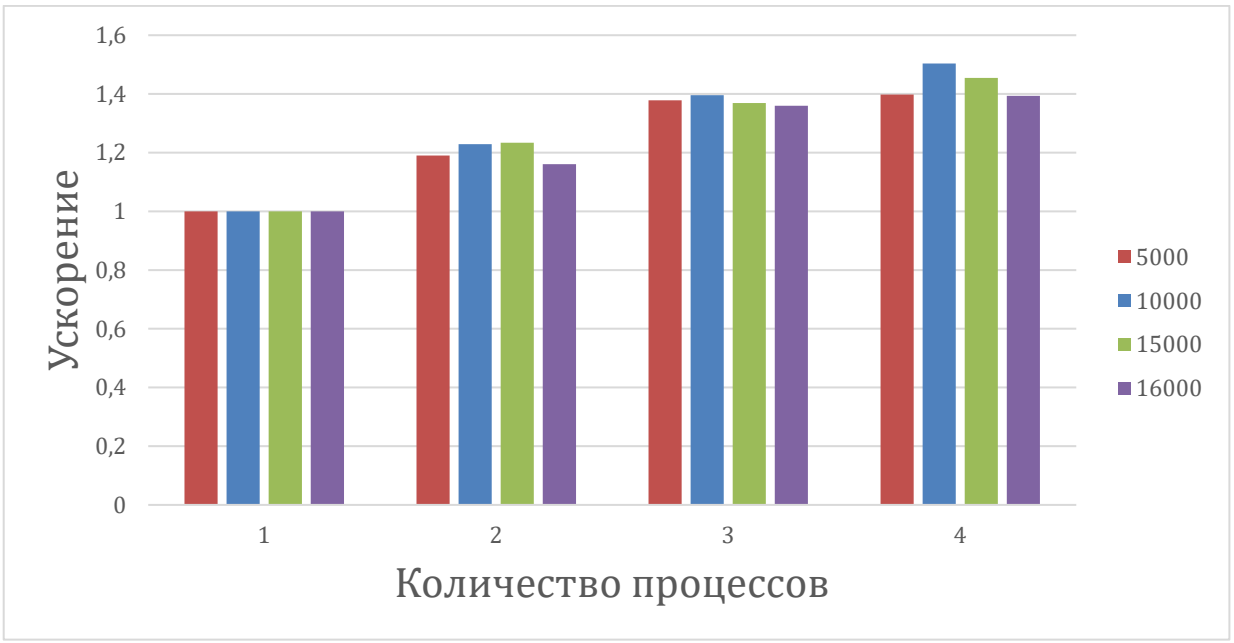
		Число процессов			
Размер матрицы	Ускорение и эффективность	1	2	3	4
5000	S	1	1,190	1,378	1,398

	E	1	0,595	0,459	0,350
10000	S	1	1,229	1,396	1,504
	E	1	0,614	0,465	0,376
15000	S	1	1,234	1,369	1,455
	E	1	0,617	0,456	0,364
16000	S	1	1,160	1,360	1,393
	E	1	0,580	0,453	0,348

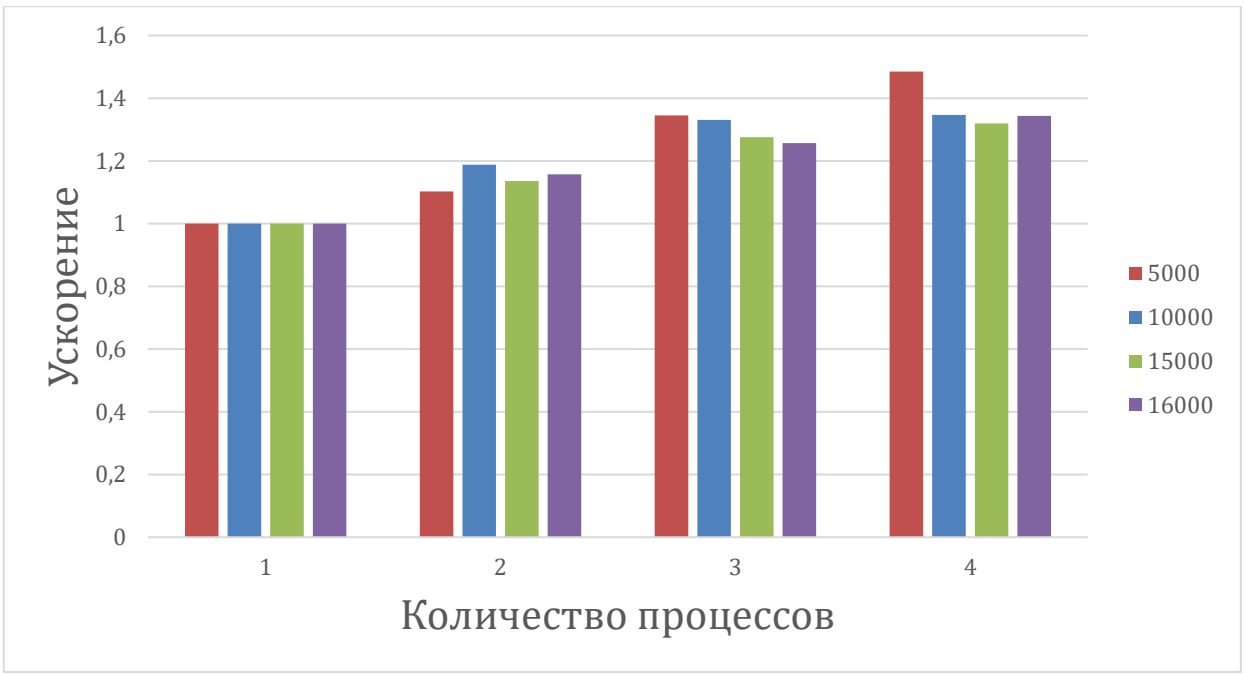
Ускорение S и эффективность E работы аналогов функций MPI при различных размерах задачи и количестве процессов

		Число процессов			
Размер матрицы	Ускорение и эффективность	1	2	3	4
5000	S	1	1,103	1,345	1,485
	E	1	0,551	0,448	0,371
10000	S	1	1,188	1,331	1,347
	E	1	0,594	0,444	0,337
15000	S	1	1,136	1,276	1,320
	E	1	0,568	0,425	0,330
16000	S	1	1,157	1,257	1,344
	E	1	0,579	0,419	0,336

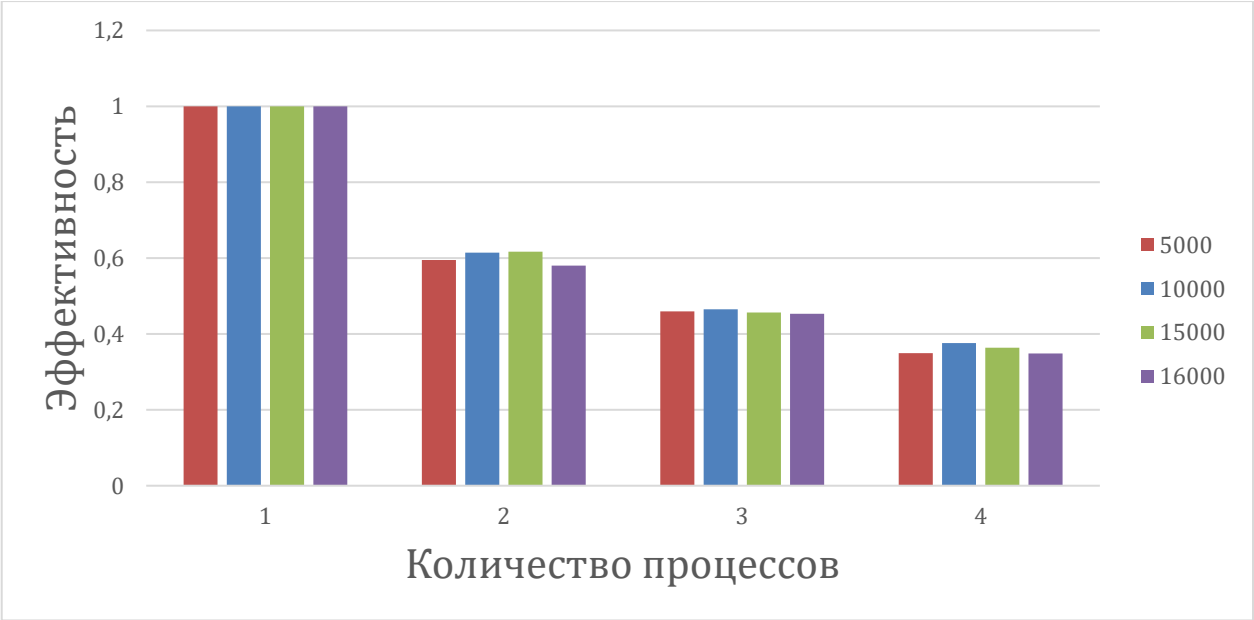
Ускорение работы MPI при различных размерах задачи и количестве процессов



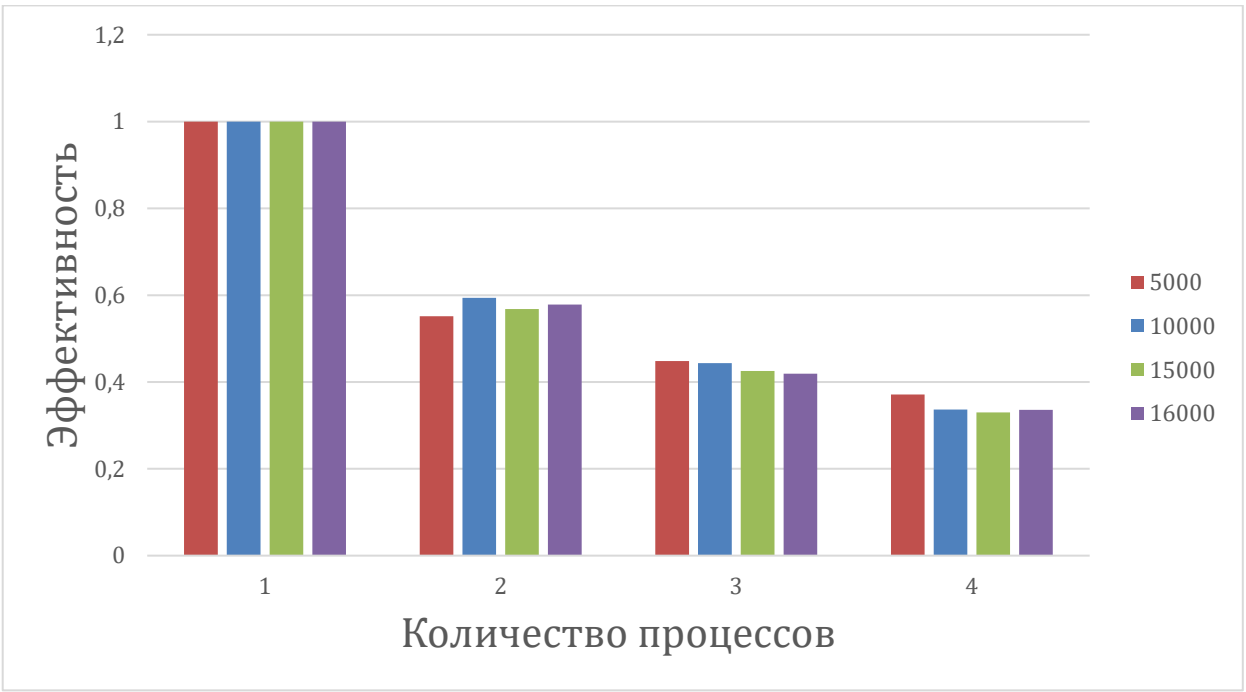
Ускорение работы аналогов функций MPI при различных размерах задачи и количестве процессов



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.

VII. 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.

VIII. 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 с 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);
}
```