МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ 1

«ОПРЕДЕЛЕНИЕ ВРЕМЕНИ РАБОТЫ ПРИКЛАДНЫХ ПРОГРАММ»

студента 2 курса, 24203 группы

Анисимова Льва Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент кафедры параллельных вычислений
Андрей Юрьевич Власенко

Новосибирск, 2026

СОДЕРЖАНИЕ

ЦЕЛЬ

Ознакомиться со стандартом MPI и коммуникациями типа точка-точка и коллективными коммуникациями, сравнить время, ускорение и эффективность работы программ, использующих разные виды коммуникаций.

ЗАДАНИЕ

I. Написать 3 программы, каждая из которых рассчитывает число s по двум данным векторам a и b равной длины N в соответствии со следующим двойным циклом:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
        s += a[i] * b[j];
```

a) последовательная программа

b) параллельная, использующая коммуникации типа точка-точка (MPI_Send, MPI_Recv)

c) параллельная, использующая коллективные коммуникации (MPI_Scatter, MPI_Reduce, MPI_Bcast)

II. Замерить время работы последовательной программы и параллельных на 2, 4, 8, 16, 24 процессах. Рекомендуется провести несколько замеров для каждого варианта запуска и выбрать минимальное время.

III. Построить графики времени, ускорения и эффективности.

IV. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

ОПИСАНИЕ РАБОТЫ

Написан и откомпилирован исходный код на языке C++ (Приложение 1), реализующий алгоритм вычисления суммы попарных произведений элементов двух векторов.

Экспериментальным путем подобран оптимальный размер векторов N = 70000 элементов, при котором время работы последовательной программы составляет приблизительно 40 секунд.

Разработана и скомпилирована программа на языке C++ (Приложение 2), использующая коммуникации типа «точка-точка». Взаимодействие между процессами организовано с помощью функций `MPI_Send` и `MPI_Recv`.

Создана и откомпилирована альтернативная MPI-программа (Приложение 3), использующая коллективные операции. Распределение данных осуществляется функцией `MPI_Scatterv`, сбор результатов — `MPI_Reduce`, а широковещательная рассылка параметров — `MPI_Bcast`. Код автоматизации тестирования программ можно найти в Приложении 4.

Проведены замеры времени выполнения для всех трех реализаций:

- Последовательная программа запускалась на одном процессе
- Параллельные программы тестировались на конфигурациях: 2, 4, 8, 16, 24 процессах
- Для каждой конфигурации выполнено по 5 запусков, выбран минимальный результат для исключения влияния случайных факторов

На основе полученных данных вычислены показатели параллельной эффективности. Результаты сведены в таблицу (Приложение 5). Графики представлены в Приложении 6.

ЗАКЛЮЧЕНИЕ

Изучены стандартные функции библиотеки MPI и два вида коммуникаций, использующихся в параллельных программах.

Сопоставлены графики времени, ускорения и эффективности. Метод коллективных коммуникаций работает быстрее и эффективнее метода «точка-точка», так как при коллективных коммуникациях задействованы сразу все процессы.

Приложение 1. Код программы №1

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
using namespace std;

int calculate_s(int N, vector<int> a, vector<int> b)
{
    int i = 0, j = 0, s = 0;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            s += a[i] * b[j];
        }
    }
    return s;
}

int main(int argc, char** argv)
{
    int N;
    ifstream fin("../../input.txt");
    fin >> N;

    vector<int> a(N), b(N);
    cout << "a = ";
    for (int i = 0; i < N; i++)
    {
        fin >> a[i];
        cout << a[i] << " ";
    }
    cout << endl;

    cout << "b = ";
    for (int i = 0; i < N; i++)
    {
        fin >> b[i];
        cout << b[i] << " ";
    }
    cout << endl;

    fin.close();

    {
        int s = 0;
        using namespace chrono;
        auto start = high_resolution_clock::now();

        s = calculate_s(N, a, b);

        auto end = high_resolution_clock::now();

        duration<double> duration = end - start;

        cout << "Result s = " << s << endl;
        cout << "Duration: " << duration.count() << " seconds" << endl;
    }

    return 0;
}
```

Приложение 2. Код программы №2

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <algorithm>

using namespace std;

// Функция вычисления частичной суммы
int calculate_s_partial(int N, const vector<int>& local_a, const vector<int>&
b)
{
    int s = 0;
    for (size_t i = 0; i < local_a.size(); i++)
    {
        for (int j = 0; j < N; j++)
        {
            s += local_a[i] * b[j];
        }
    }
    return s;
}

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 N = 0;
    vector<int> b;
    vector<int> local_a;

    // Процесс 0 читает данные
    if (rank == 0)
    {
        ifstream fin("../../input.txt");
        if (!fin.is_open())
        {
            cerr << "Error opening input.txt" << endl;
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }

        fin >> N;
        if (fin.fail() || N <= 0)
        {
            cerr << "Error reading N or invalid N" << endl;
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }

        vector<int> full_a(N);
        b.resize(N);

        for (int i = 0; i < N; i++)
        {
            fin >> full_a[i];
        }
```

```
        for (int i = 0; i < N; i++)
        {
            fin >> b[i];
        }

        fin.close();

        cout << "N = " << N << endl;
        cout << "Number of MPI processes: " << size << endl;

        // Делим вектор
        int base_chunk = N / size;
        int remainder = N % size;

        int start_0 = 0 * base_chunk + min(0, remainder);
        int end_0 = start_0 + base_chunk + (0 < remainder ? 1 : 0);
        int local_size_0 = end_0 - start_0;

        local_a.resize(local_size_0);
        copy(full_a.begin()    +    start_0,    full_a.begin()    +    end_0,
local_a.begin());

        // Рассылаем N всем процессам
        for (int dest = 1; dest < size; dest++)
        {
            MPI_Send(&N, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }

        // Рассылаем вектор b ЦЕЛИКОМ всем процессам
        for (int dest = 1; dest < size; dest++)
        {
            MPI_Send(b.data(), N, MPI_INT, dest, 1, MPI_COMM_WORLD);
        }

        // Рассылаем ЧАСТИ вектора a каждому процессу
        for (int dest = 1; dest < size; dest++)
        {
            int dest_start = dest * base_chunk + min(dest, remainder);
            int dest_end = dest_start + base_chunk + (dest < remainder ? 1 :
0);
            int dest_size = dest_end - dest_start;

            // Отправляем размер части
            MPI_Send(&dest_size, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);

            // Отправляем часть вектора
            MPI_Send(&full_a[dest_start],   dest_size,   MPI_INT,   dest,   3,
MPI_COMM_WORLD);
        }
    }
    else
    {
        // Получаем N
        MPI_Recv(&N, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Получаем вектор b
        b.resize(N);
        MPI_Recv(b.data(),     N,     MPI_INT,     0,     1,     MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        // Получаем размер своей части вектора a
        int local_size;
        MPI_Recv(&local_size,    1,    MPI_INT,    0,    2,    MPI_COMM_WORLD,
```

```
MPI_STATUS_IGNORE);
        local_a.resize(local_size);

        // Получаем свою часть вектора a
        MPI_Recv(local_a.data(), local_size, MPI_INT, 0, 3, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    // Синхронизируем
    MPI_Barrier(MPI_COMM_WORLD);

    // Начало замера времени
    chrono::high_resolution_clock::time_point start_time, end_time;
    if (rank == 0)
    {
        start_time = chrono::high_resolution_clock::now();
    }

    int local_s = calculate_s_partial(N, local_a, b);

    int global_s = 0;

    if (rank == 0)
    {
        global_s = local_s;

        for (int i = 1; i < size; i++)
        {
            int received_s;
            MPI_Recv(&received_s,   1,   MPI_INT,   i,   4,   MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            global_s += received_s;
        }
    }
    else
    {
        MPI_Send(&local_s, 1, MPI_INT, 0, 4, MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
    {
        end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> total_duration = end_time - start_time;

        cout << "\n=== RESULTS ===" << endl;
        cout << "Total sum s = " << global_s << endl;
        cout  <<  "Total  execution  time:  "  <<  total_duration.count()  <<  "
seconds" << endl;
        cout << "=================" << endl;
    }

    MPI_Finalize();
    return 0;
}
```

Приложение 3. Код программы №3

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <algorithm>

using namespace std;

int calculate_partial_sum(int N, const vector<int>& local_a, const
vector<int>& b)
{
    int s = 0;
    for (size_t i = 0; i < local_a.size(); i++)
    {
        for (int j = 0; j < N; j++)
        {
            s += local_a[i] * b[j];
        }
    }
    return s;
}

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 N = 0;
    vector<int> full_a;
    vector<int> b;
    vector<int> local_a;

    if (rank == 0)
    {
        ifstream fin("../../input.txt");
        if (!fin.is_open())
        {
            cerr << "Error opening input.txt" << endl;
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }

        fin >> N;
        if (fin.fail() || N <= 0)
        {
            cerr << "Error reading N or invalid N" << endl;
            MPI_Abort(MPI_COMM_WORLD, 1);
            return 1;
        }

        cout << "N = " << N << endl;
        cout << "Number of processes: " << size << endl;

        full_a.resize(N);
        b.resize(N);

        for (int i = 0; i < N; i++) fin >> full_a[i];
        for (int i = 0; i < N; i++) fin >> b[i];
```

```
        fin.close();
    }

    // Распределяем по процессам
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank != 0)
    {
        b.resize(N);
    }
    MPI_Bcast(b.data(), N, MPI_INT, 0, MPI_COMM_WORLD); // b-вектор целиком (требование)

    // MPI_Scatter требует одинаковое количество элементов для каждого процесса
    // Если N не делится нацело на size, то MPI_Scatterv

    int base_chunk = N / size;
    int remainder = N % size;

    int local_size = base_chunk + (rank < remainder ? 1 : 0);
    local_a.resize(local_size);

    // MPI_Scatterv требует counts и displs
    vector<int> counts(size); // массив количеств элементов, посылаемых процессам
    vector<int> displs(size);
    // массив смещений, i-ое значение определяет смещение i-го блока данных относительно начала sendbuf

    int current_displ = 0;
    for (int i = 0; i < size; i++)
    {
        counts[i] = base_chunk + (i < remainder ? 1 : 0);
        displs[i] = current_displ;
        current_displ += counts[i];
    }

    if (rank != 0)
    {
        full_a.resize(0);
    }

    MPI_Scatterv(
        rank == 0 ? full_a.data() : nullptr, // Отправной буфер
        counts.data(), // Сколько элементов каждому процессу
        displs.data(), // Смещения для каждого процесса
        MPI_INT,
        local_a.data(), // Приемный буфер
        local_size, // Сколько элементов получаем
        MPI_INT,
        0, // root
        MPI_COMM_WORLD
    );

    MPI_Barrier(MPI_COMM_WORLD);

    chrono::high_resolution_clock::time_point start_time, end_time;

    if (rank == 0)
    {
        start_time = chrono::high_resolution_clock::now();
    }
```

```
    int local_s = calculate_partial_sum(N, local_a, b);

    // Собираем по потокам
    int global_s = 0;
    MPI_Reduce(&local_s, &global_s, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
    {
        end_time = chrono::high_resolution_clock::now();
        chrono::duration<double> total_duration = end_time - start_time;

        cout << "\n=== RESULTS ===" << endl;
        cout << "Total sum s = " << global_s << endl;
        cout << "Total execution time: " << total_duration.count() << "
seconds" << endl;
        cout << "=================" << endl;
    }

    MPI_Finalize();
    return 0;
}
```

Приложение 4. Код скрипта создания input.txt и автозапуска

```
#!/bin/bash

# Тест с N=70000
echo "Generating input with N=70000..."
python3 -c "
N = 70000
print(N)
# Вектор a: 1, 2, 3, ..., N
print(' '.join(str(i+1) for i in range(N)))
# Вектор b: N, N-1, ..., 1
print(' '.join(str(N-i) for i in range(N)))
" > ../../input.txt

echo "Testing with N=70000..."
echo "======================"

for procs in 2 4 8 16 24
do
    echo -n "Processes: $procs -> "
    mpiexec -n $procs ./mpi_app 2>/dev/null | grep "Total execution
time" | cut -d: -f2
done
```

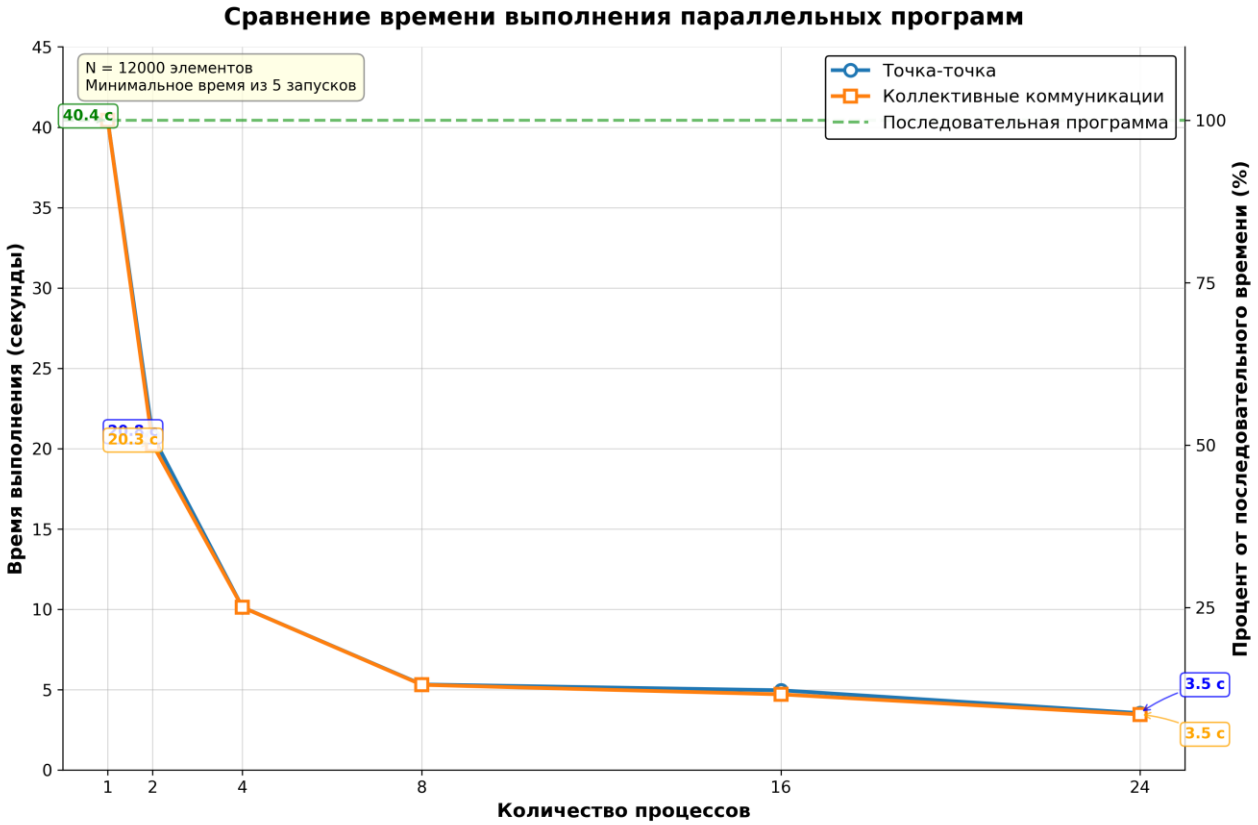
Приложение 5. Таблицы

	ВРЕМЯ *ТОЧКА-ТОЧКА*	**УСКОРЕНИЕ** *ТОЧКА-ТОЧКА*	**ЭФФЕКТИВНОСТЬ** *ТОЧКА-ТОЧКА*	***ВРЕМЯ ПОСЛЕДОВАТЕЛЬНОЙ:*** *40.4323*
2	20.7929	1.945	97.25%	
4	10.1324	3.991	99.77%	
8	5.32562	7.592	94.90%	
16	4.96459	8.144	50.90%	
24	3.54248	11.415	47.56%	

	ВРЕМЯ *КОЛЛЕКТИВ*	**УСКОРЕНИЕ** *КОЛЛЕКТИВ*	**ЭФФЕКТИВНОСТЬ** *КОЛЛЕКТИВ*	***ВРЕМЯ ПОСЛЕДОВАТЕЛЬНОЙ:*** *40.4323*
2	20.2618	1.996	99.80%	
4	10.1291	3.992	99.80%	
8	5.3021	7.627	95.34%	
16	4.7104	8.584	53.65%	
24	3.46103	11.684	48.68%	

Приложение 6. Графики

1. Время



2. Ускорение и эффективность