

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ЛАБОРАТОРНАЯ РАБОТА №1-2**  
**по дисциплине «Параллельные алгоритмы и системы»**  
**Тема: ЗАПУСК ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ И ПЕРЕДАЧА**  
**ДАННЫХ ПО ПРОЦЕССАМ**

Студент гр. 1307

\_\_\_\_\_

Виноградов А.С.

Преподаватель

\_\_\_\_\_

Санкт-Петербург

2025

## **Введение**

**Тема работы:** запуск параллельной программы и передача данных по процессам.

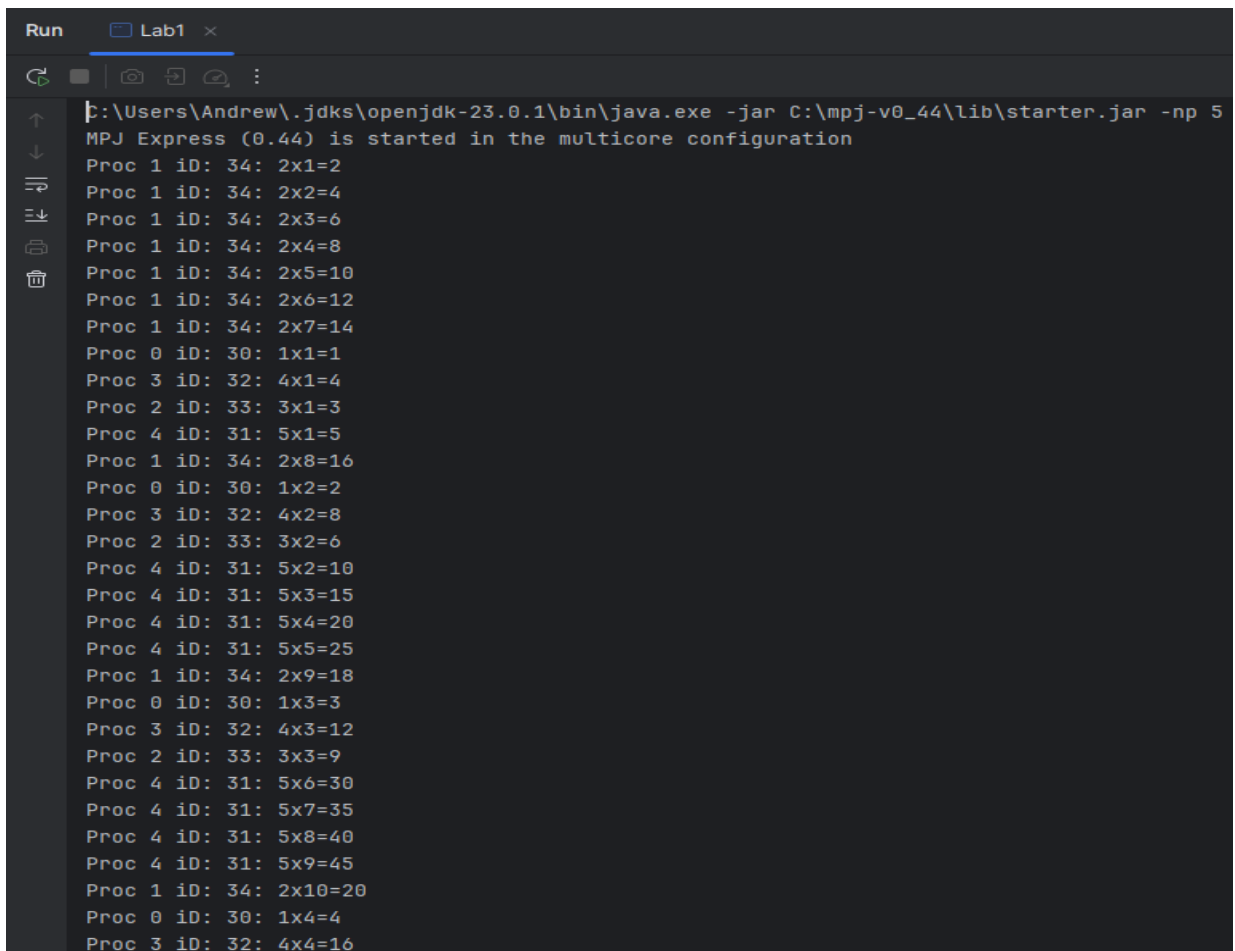
**Цель работы:** Освоить процесс запуска программы с применением библиотеки MPICH2. Научиться получать сведения о количестве запущенных процессов и номере отдельного процесса. Освоить функции передачи данных между процессами.

### **Лабораторная работа №1**

*Задания:*

- 1) Создать и запустить программу на 2-х процессах с применением функций `int MPI_Init( int* argc, char*** argv)` и `int MPI_Finalize( void )`.
- 2) Создать и запустить программу на 3-х процессах, Программа должна выводить на экран номер процесса и какой-либо идентификатор процесса.
- 3) Создать и запустить программу на n-х процессах печати таблицы умножения.

Для выполнения поставленной задачи, достаточно выполнить 3-е задание, в котором будут элементы как задания 1, так и задания 2. Программа будет протестирована на 5 процессах (-nr 5) и выведет таблицу умножения.



```
Run Lab1 x
C:\Users\Andrew\jdk-23.0.1\bin\java.exe -jar C:\mpj-v0_44\lib\starter.jar -np 5
MPJ Express (0.44) is started in the multicore configuration
Proc 1 iD: 34: 2x1=2
Proc 1 iD: 34: 2x2=4
Proc 1 iD: 34: 2x3=6
Proc 1 iD: 34: 2x4=8
Proc 1 iD: 34: 2x5=10
Proc 1 iD: 34: 2x6=12
Proc 1 iD: 34: 2x7=14
Proc 0 iD: 30: 1x1=1
Proc 3 iD: 32: 4x1=4
Proc 2 iD: 33: 3x1=3
Proc 4 iD: 31: 5x1=5
Proc 1 iD: 34: 2x8=16
Proc 0 iD: 30: 1x2=2
Proc 3 iD: 32: 4x2=8
Proc 2 iD: 33: 3x2=6
Proc 4 iD: 31: 5x2=10
Proc 4 iD: 31: 5x3=15
Proc 4 iD: 31: 5x4=20
Proc 4 iD: 31: 5x5=25
Proc 1 iD: 34: 2x9=18
Proc 0 iD: 30: 1x3=3
Proc 3 iD: 32: 4x3=12
Proc 2 iD: 33: 3x3=9
Proc 4 iD: 31: 5x6=30
Proc 4 iD: 31: 5x7=35
Proc 4 iD: 31: 5x8=40
Proc 4 iD: 31: 5x9=45
Proc 1 iD: 34: 2x10=20
Proc 0 iD: 30: 1x4=4
Proc 3 iD: 32: 4x4=16
```

Рисунок 1 - результат работы программы

*Краткое пояснение логики программы:*

Каждый процесс вычисляет свои локальные значения  $a_i$  и  $b_i$  (начиная с умножения на 1).

Процессы обмениваются данными ( $a_i$  и  $b_i$ ) по схеме (для избегания дедлока):

Процессы 0 и 3 сначала отправляют данные, затем получают.

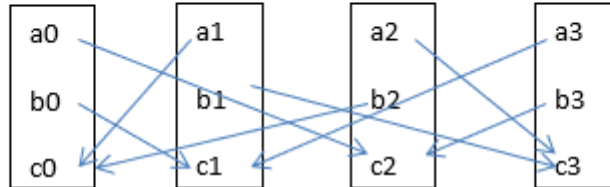
Процессы 1 и 2 сначала получают данные, затем отправляют.

После обмена каждый процесс вычисляет сумму полученных значений ( $received\_a[0] + received\_b[0]$ ).

## Лабораторная работа №2

## Задание

- 1) Запустить 4 процесса.
- 2) На каждом процессе создать переменные:  $a_i, b_i, c_i$ , где  $i$  – номер процесса. Инициализировать переменные. Вывести данные на печать.
- 3) Передать данные на другой процесс. Напечатать номера процессов и поступившие данные. Найти:  $c_0 = a_1 + b_2$ ;  $c_1 = a_3 + b_0$ ;  $c_2 = a_0 + b_3$ ;  $c_3 = a_2 + b_1$ .



5) Запустить  $n$  процессов и найти по вариантам (вариант 1) – Сумма нечетных элементов вектора.

```
C:\Users\Andrew\.jdk\openjdk-23.0.1\bin\java.exe ...
MPJ Express (0.44) is started in the multicore configuration
Proc 0 ID: 30 -> ai: 1, bi: 2
Proc 3 ID: 31 -> ai: 4, bi: 8
Proc 1 ID: 33 -> ai: 2, bi: 4
Proc 2 ID: 32 -> ai: 3, bi: 6
Proc 2 ID: 32 received: a=1, b=8 -> c2 = 9
Proc 1 ID: 33 received: a=4, b=2 -> c1 = 6
Proc 0 ID: 30 received: a=2, b=6 -> c0 = 8
Proc 3 ID: 31 received: a=3, b=4 -> c3 = 7
```

Рис. 2 – Результат работы Лаб2\_1

```
C:\Users\Andrew\.jdk\openjdk-23.0.1\bin\java.exe ...  
MPJ Express (0.44) is started in the multicore configuration  
Proc 0 (1 to 20) local sum: 100  
Proc 1 (21 to 40) local sum: 300  
Proc 3 (61 to 80) local sum: 700  
Proc 2 (41 to 60) local sum: 500  
Proc 4 (81 to 100) local sum: 900  
Total odd sum: 2500
```

Рис. 3 – Результат работы Лаб 2\_2 (1-100)

Краткое пояснение логики программы:

- Каждый процесс перебирает числа в своем диапазоне.
- Складываются только нечетные числа.
- Процессы (кроме rank 0) отправляют локальные суммы в rank 0.
- Главный процесс (rank 0) собирает суммы и вычисляет глобальную сумму.
- Каждый процесс выводит свою локальную сумму.
- rank 0 выводит итоговую сумму нечетных чисел.

Произведено исправление алгоритма, было исправлено:

- Убрано локальное создание данных в каждом процессе.
- Добавлено создание полного массива fullArray только на процессе с rank 0, где он заполняется значениями от start до end.
- Добавлены массивы sendcounts и displs для определения размеров и смещений частей массива для каждого процесса.

## Вывод

В процессе выполнения лабораторной работы были изучены базовые принципы использования библиотеки MPI (Message Passing Interface) для организации параллельных вычислений в среде Java. Были разработаны три программы, каждая из которых демонстрирует различные аспекты взаимодействия процессов при распределении вычислительных задач.

1. Используемые методы MPI MPI.Init(args): Инициализирует MPI-окружение.

Этот метод необходимо вызывать перед использованием любых других функций MPI.

MPI.COMM\_WORLD.Rank(): Возвращает уникальный номер (ранг) текущего процесса, который позволяет идентифицировать процесс и управлять его выполнением.

MPI.COMM\_WORLD.Size(): Определяет общее количество процессов, участвующих в выполнении программы.

MPI.COMM\_WORLD.Send(buffer, offset, count, datatype, destination, tag): Отправляет данные другому процессу. В параметрах указываются:

- buffer: массив с передаваемыми данными;
- offset: начальная позиция в массиве;
- count: количество передаваемых элементов;
- datatype: тип данных (например, MPI.INT);
- destination: ранг процесса-получателя;
- tag: идентификатор сообщения.

MPI.COMM\_WORLD.Recv(buffer, offset, count, datatype, source, tag): Принимает данные от другого процесса. Параметры аналогичны методу Send, но дополнительно указывается source — ранг процесса-отправителя.

MPI.Finalize(): Завершает работу MPI-окружения, освобождая выделенные ресурсы. Этот метод необходимо вызывать в конце программы.

2. Описание программ Lab1: Данная программа демонстрирует принцип параллельного вычисления таблицы умножения. Каждый процесс

рассчитывает определённые строки таблицы (например, процесс с рангом 0 отвечает за строки 1, 5, 9 и т. д.), используя шаг, равный количеству процессов (size), что позволяет равномерно распределить вычисления.

Lab2\_1: Программа реализует обмен данными между процессами согласно заданной схеме. Каждый процесс вычисляет собственные значения  $a_i$  и  $b_i$ , передаёт их другим процессам и получает данные от них. После обмена процесс суммирует полученные значения и выводит результат.

Lab2\_2: В данной программе реализуется вычисление суммы чисел в заданном диапазоне с распределением вычислений между процессами. Числовой диапазон делится на участки, обрабатываемые разными процессами. В случае, если количество процессов превышает число элементов в диапазоне, некоторые из них остаются без работы. Процесс с рангом 0 получает частичные суммы от остальных процессов и вычисляет итоговый результат.

3. Основные выводы Распределение вычислений: MPI позволяет эффективно делить вычислительные задачи между процессами, что особенно полезно для задач, разбиваемых на независимые части. В программе task2 показан механизм распределения диапазона чисел и обработка ситуаций, когда число процессов превышает количество задач.

Обмен данными: Методы Send и Recv позволяют организовывать передачу данных между процессами, что критически важно для координированных вычислений. В программе task1 демонстрируется передача данных между процессами в соответствии с заданной схемой.

Гибкость: MPI обеспечивает широкие возможности управления процессами, позволяя динамически перераспределять задачи и эффективно организовывать вычисления даже в условиях неравномерной загрузки.

Эффективность: Использование MPI ускоряет выполнение вычислений за счёт их параллельной обработки. Однако для достижения максимальной производительности важно правильно распределять задачи, избегая простаивания процессов.

4. Заключение В ходе выполнения лабораторной работы были разработаны и

протестированы программы, демонстрирующие ключевые возможности библиотеки MPI для организации параллельных вычислений. Изучены основные методы MPI, такие как Send, Recv, Rank, Size, и их применение для управления распределением задач и передачи данных между процессами. Приобретённые знания могут быть использованы в дальнейшем для решения более сложных задач, требующих высокопроизводительных вычислений.



## Приложение А

Лаб 1.

```
package vinandy.Lab1;

import mpi.MPI;

public class Lab1 {
    public static void main(String[] args) {
        MPI.Init(args);

        int Rank = MPI.COMM_WORLD.Rank();
        int Size = MPI.COMM_WORLD.Size();

        for (int i = Rank + 1; i <= 10; i += Size) {
            for (int j = 1; j <= 10; j++) {
                System.out.println("Proc " + Rank + " iD: " + Thread.currentThread().getId()
                    + ": " + i + "x" + j + "=" + (i * j));
            }
        }

        MPI.Finalize();
    }
}
```

Лаб 2\_1

```
package vinandy.Lab2;

import mpi.MPI;

public class Lab2_1 {
    public static void main(String[] args) {
        MPI.Init(args);

        int rank = MPI.COMM_WORLD.Rank();

        int ai = rank + 1;
        int bi = (rank + 1) * 2;

        System.out.println("Proc " + rank + " ID: " + Thread.currentThread().getId()
            + " -> ai: " + ai + ", bi: " + bi);

        int[] received_a = new int[1];
        int[] received_b = new int[1];

        int send_a, send_b;
        int receive_a, receive_b;

        switch (rank) {
            case 0:
                send_a = 2;
                send_b = 1;
                receive_a = 1;
                receive_b = 2;
                break;
            case 1:
                send_a = 0;
                send_b = 3;
        }
    }
}
```

```

        receive_a = 3;
        receive_b = 0;
        break;
    case 2:
        send_a = 3;
        send_b = 0;
        receive_a = 0;
        receive_b = 3;
        break;
    case 3:
        send_a = 1;
        send_b = 2;
        receive_a = 2;
        receive_b = 1;
        break;
    default:
        throw new IllegalStateException("Unexpected rank: " + rank);
}

if (rank == 0 || rank == 3) {
    MPI.COMM_WORLD.Send(new int[]{ai}, 0, 1, MPI.INT, send_a, 99);
    MPI.COMM_WORLD.Send(new int[]{bi}, 0, 1, MPI.INT, send_b, 99);
    MPI.COMM_WORLD.Recv(received_a, 0, 1, MPI.INT, receive_a, 99);
    MPI.COMM_WORLD.Recv(received_b, 0, 1, MPI.INT, receive_b, 99);
} else {
    MPI.COMM_WORLD.Recv(received_a, 0, 1, MPI.INT, receive_a, 99);
    MPI.COMM_WORLD.Recv(received_b, 0, 1, MPI.INT, receive_b, 99);
    MPI.COMM_WORLD.Send(new int[]{ai}, 0, 1, MPI.INT, send_a, 99);
    MPI.COMM_WORLD.Send(new int[]{bi}, 0, 1, MPI.INT, send_b, 99);
}

System.out.println("Proc " + rank + " ID: " + Thread.currentThread().getId()
    + " received: a=" + received_a[0]
    + ", b=" + received_b[0]
    + " -> c" + rank + " = " + (received_a[0] + received_b[0]));

    MPI.Finalize();
}
}

```

Лаб 2\_2

```

package vinandy.Lab2;

import mpi.MPI;

public class Lab2_2 {
    public static void main(String[] args) {
        MPI.Init(args);

        int rank = MPI.COMM_WORLD.Rank();
        int size = MPI.COMM_WORLD.Size();

        int start = Integer.parseInt(System.getProperty("start", "1"));
        int end = Integer.parseInt(System.getProperty("end", "100"));
        int range = end - start + 1;

        int[] fullArray = null;
        int localRange = range / size;
        int remainder = range % size;

        if (rank == 0) {

```

```

        fullArray = new int[range];
        for (int i = 0; i < range; i++) {
            fullArray[i] = start + i;
        }
    }

    int[] sendcounts = new int[size];
    int[] displs = new int[size];
    int offset = 0;

    for (int i = 0; i < size; i++) {
        if (i < remainder) {
            sendcounts[i] = localRange + 1;
        } else {
            sendcounts[i] = localRange;
        }
        displs[i] = offset;
        offset += sendcounts[i];
    }

    int localSize = sendcounts[rank];
    int[] localArray = new int[localSize];

    MPI.COMM_WORLD.Scatterv(fullArray, 0, sendcounts, displs, MPI.INT,
        localArray, 0, localSize, MPI.INT, 0);

    int localSum = 0;
    int localStart = localArray[0];
    int localEnd = localArray[localArray.length - 1];

    for (int i = 0; i < localSize; i++) {
        if (localArray[i] % 2 != 0) {
            localSum += localArray[i];
        }
    }

    System.out.println("Proc " + rank + " (" + localStart + " to " + localEnd + ") local sum: " + localSum);

    if (rank != 0) {
        MPI.COMM_WORLD.Send(new int[]{localSum}, 0, 1, MPI.INT, 0, 0);
    } else {
        int globalSum = localSum;
        for (int i = 1; i < size; i++) {
            if (i < range) {
                int[] receivedSum = new int[1];
                MPI.COMM_WORLD.Recv(receivedSum, 0, 1, MPI.INT, i, 0);
                globalSum += receivedSum[0];
            }
        }
        System.out.println("Total odd sum: " + globalSum);
    }

    MPI.Finalize();
}
}

```