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 _____ Николаев К.Д.

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.

Было принято решение реализовать только третье задание, так как оно полностью покрывает первые два. Это связано с тем, что в третьем задании можно гибко задавать количество процессов с помощью параметра -np N, что позволяет адаптировать программу под любые требования, включая те, что были в первых двух заданиях. Таким образом, третий вариант является универсальным решением.

```
java -jar "C:\mpj\lib\starter.jar" -np 2 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab1.Lab1
MPJ Express (0.44) is started in the multicore configuration
Process 0 ID: 21: 1x1=1
Process 0 ID: 21: 1x2=2
Process 0 ID: 21: 1x3=3
Process 0 ID: 21: 1x4=4
Process 1 ID: 22: 2x1=2
Process 1 ID: 22: 2x2=4
Process 1 ID: 22: 2x3=6
Process 0 ID: 21: 1x5=5
Process 0 ID: 21: 1x6=6
Process 0 ID: 21: 1x7=7
Process 1 ID: 22: 2x4=8
Process 1 ID: 22: 2x5=10
Process 0 ID: 21: 1x8=8
```

Рисунок 1 - результат работы программы (количество процессов = 2).

```
java -jar "C:\mpj\lib\starter.jar" -np 3 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab1.Lab1
MPJ Express (0.44) is started in the multicore configuration
Process 1 ID: 22: 2x1=2
Process 1 ID: 22: 2x2=4
Process 2 ID: 23: 3x1=3
Process 0 ID: 21: 1x1=1
Process 1 ID: 22: 2x3=6
Process 2 ID: 23: 3x2=6
Process 0 ID: 21: 1x2=2
Process 1 ID: 22: 2x4=8
Process 2 ID: 23: 3x3=9
Process 0 ID: 21: 1x3=3
Process 1 ID: 22: 2x5=10
Process 2 ID: 23: 3x4=12
Process 0 ID: 21: 1x4=4
Process 1 ID: 22: 2x6=12
```

Рисунок 2 - результат работы программы (количество процессов = 3).

```
java -jar "C:\mpj\lib\starter.jar" -np 4 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab1.Lab1
MPJ Express (0.44) is started in the multicore configuration
Process 1 ID: 24: 2x1=2
Process 1 ID: 24: 2x2=4
Process 2 ID: 22: 3x1=3
Process 3 ID: 23: 4x1=4
Process 0 ID: 21: 1x1=1
Process 1 ID: 24: 2x3=6
Process 2 ID: 22: 3x2=6
Process 3 ID: 23: 4x2=8
Process 0 ID: 21: 1x2=2
Process 1 ID: 24: 2x4=8
Process 2 ID: 22: 3x3=9
Process 3 ID: 23: 4x3=12
Process 0 ID: 21: 1x3=3
Process 1 ID: 24: 2x5=10
Process 2 ID: 22: 3x4=12
Process 3 ID: 23: 4x4=16
Process 0 ID: 21: 1x4=4
```

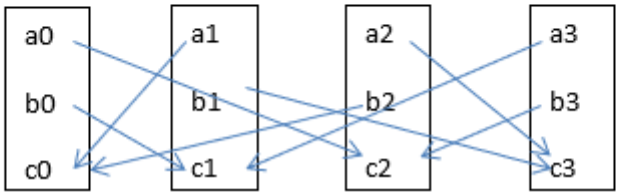
Рисунок 3 - результат работы программы (количество процессов = 4).

Краткое пояснение логики программы:

- Каждый процесс вычисляет свои локальные значения ai и bi.
- Процессы обмениваются данными (ai и bi) 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]).
- Результаты выводятся в консоль.

Лабораторная работа №2

Задания:

1) Запустить 4 процесса.
2) На каждом процессе создать переменные: ai,bi,ci, где I – номер процесса. Инициализировать переменные. Вывести данные на печать.
3) Передать данные на другой процесс. Напечатать номера процессов и поступившие данные. Найти: c0=a1+b2; c1=a3+b0; c2=a0+b3; c3=a2+b1.
4) 

5) Запустить n процессов и найти по вариантам (вариант 9) – Сумму элементов из заданного пользователем диапазона.

```
java -jar "C:\mpj\lib\starter.jar" -np 4 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab2.task1
MPJ Express (0.44) is started in the multicore configuration
Process 1 ID: 24 -> ai: 2, bi: 4
Process 2 ID: 23 -> ai: 3, bi: 6
Process 0 ID: 21 -> ai: 1, bi: 2
Process 3 ID: 22 -> ai: 4, bi: 8
Process 1 ID: 24 received: a=4, b=2 -> c1 = 6
Process 3 ID: 22 received: a=3, b=4 -> c3 = 7
Process 0 ID: 21 received: a=2, b=6 -> c0 = 8
Process 2 ID: 23 received: a=1, b=8 -> c2 = 9
```

Рисунок 4 - результат работы программы лаб2, задание 1.

```
java -jar "C:\mpj\lib\starter.jar" -np 4 -Dstart=100 -Dend=101 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab2.task2
MPJ Express (0.44) is started in the multicore configuration
Process 3 ID: 23 is idle (no range assigned).
Process 2 ID: 24 is idle (no range assigned).
Process 1 ID: 22: start-101 end-101 sum-101
Process 0 ID: 21: start-100 end-100 sum-100
Total sum: 201
```

Рисунок 5 - результат работы программы лаб2, задание 2 (количество процессов – 4, диапазон – 100-101).

```
java -jar "C:\mpj\lib\starter.jar" -np 4 -Dstart=100 -Dend=2000 -cp "D:\CODING\JavaProjects\parallel\target\classes" lebibop.lab2.task2
MPJ Express (0.44) is started in the multicore configuration
Process 1 ID: 24: start-576 end-1050 sum-386175
Process 3 ID: 23: start-1526 end-2000 sum-837425
Process 0 ID: 21: start-100 end-575 sum-160650
Process 2 ID: 22: start-1051 end-1525 sum-611800
Total sum: 1996050
```

Рисунок 6 - результат работы программы лаб2, задание 2 (количество процессов – 4, диапазон – 100-2000).

Краткое пояснение логики программы:

- Программа считывает параметры start и end (начало и конец диапазона чисел).

- Если количество процессов (size) больше, чем диапазон чисел (range), лишние процессы остаются idle (без работы).
- Диапазон чисел делится между процессами. Если диапазон не делится нацело, первые процессы получают на одно число больше.
- Каждый процесс вычисляет свою часть суммы чисел в своем диапазоне (localSum).
- Все процессы, кроме процесса с рангом 0, отправляют свои локальные суммы (localSum) процессу с рангом 0.
- Процесс с рангом 0 собирает все локальные суммы и вычисляет общую сумму (globalSum).
- Каждый процесс выводит свой диапазон и локальную сумму.
- Процесс с рангом 0 выводит общую сумму.

Вывод

В ходе выполнения лабораторной работы были изучены основы работы с библиотекой MPI (Message Passing Interface) для организации параллельных вычислений в Java. Были реализованы три программы, каждая из которых демонстрирует различные аспекты использования MPI для распределения задач между процессами.

1. Использованные методы MPI

MPI.Init(args):

Инициализирует MPI-окружение. Должен быть вызван перед использованием любых других MPI-функций.

MPI.COMM_WORLD.Rank():

Возвращает ранг (идентификатор) текущего процесса. Ранг используется для идентификации процесса и управления его поведением.

MPI.COMM_WORLD.Size():

Возвращает общее количество процессов, участвующих в выполнении программы.

MPI.COMM_WORLD.Send(buffer, offset, count, datatype, destination, tag):

Отправляет данные другому процессу. Параметры:

buffer: Массив данных для отправки.

offset: Смещение в массиве.

count: Количество элементов для отправки.

datatype: Тип данных (например, MPI.INT).

destination: Ранг процесса-получателя.

tag: Метка сообщения (используется для идентификации).

MPI.COMM_WORLD.Recv(buffer, offset, count, datatype, source, tag):

Получает данные от другого процесса. Параметры:

buffer: Массив для сохранения полученных данных.

offset: Смещение в массиве.

count: Количество элементов для получения.

datatype: Тип данных (например, MPI.INT).

source: Ранг процесса-отправителя.

tag: Метка сообщения (должна совпадать с тегом отправки).

MPI.Finalize():

Завершает работу MPI-окружения. Освобождает ресурсы, выделенные для MPI. Должен быть вызван в конце программы.

2. Описание программ

Lab1:

Программа демонстрирует параллельное вычисление таблицы умножения.

Каждый процесс вычисляет свою часть таблицы умножения (например, процесс с рангом 0 вычисляет строки 1, 5, 9 и т.д.).

Используется цикл с шагом, равным количеству процессов (size), чтобы распределить задачи между процессами.

Lab2.task1:

Программа организует обмен данными между процессами по заданной схеме.

Каждый процесс вычисляет свои значения ai и bi, отправляет их другим процессам и получает данные от них.

После обмена каждый процесс вычисляет сумму полученных значений и выводит результат.

Lab2.task2:

Программа вычисляет сумму чисел в заданном диапазоне, распределяя задачи между процессами.

Диапазон чисел делится между процессами. Если количество процессов больше, чем чисел в диапазоне, лишние процессы остаются без работы (idle).

Процесс с рангом 0 собирает результаты от всех процессов и вычисляет общую сумму.

3. Основные выводы

Распределение задач:

MPI позволяет эффективно распределять задачи между процессами, что особенно полезно для вычислений, которые можно разделить на независимые части. В программе task2 показано, как можно распределить диапазон чисел между процессами и обработать случаи, когда процессов больше, чем задач.

Обмен данными:

Методы Send и Recv позволяют организовать обмен данными между процессами. Это важно для задач, где процессы должны обмениваться промежуточными результатами. В программе task1 показан пример обмена данными между процессами по заданной схеме.

Гибкость:

MPI предоставляет гибкость в управлении процессами. Например, можно динамически распределять задачи между процессами и обрабатывать случаи, когда процессы остаются без работы.

Эффективность:

Использование MPI позволяет ускорить выполнение задач за счет параллельных вычислений. Однако важно правильно распределять задачи, чтобы избежать простоя процессов.

4. Заключение

В ходе выполнения лабораторной работы были успешно реализованы программы, демонстрирующие возможности MPI для организации параллельных вычислений. Были изучены основные методы MPI, такие как Send, Recv, Rank, Size, и их применение для распределения задач и обмена данными между процессами. Полученные навыки могут быть использованы для решения более сложных задач, требующих параллельных вычислений.

Приложение А

Lab1.java

```
package lebibop.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("Process " + rank + " ID: " + Thread.currentThread().getId()
                  + ": " + i + "x" + j + "=" + (i * j));
         }
      }

      MPI.Finalize();
   }
}
```

task1.java

```
package lebibop.lab2;

import mpi.MPI;

public class task1 {
   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("Process " + rank + " ID: " + Thread.currentThread().getId()
            + " -> ai: " + ai + ", bi: " + bi);
```

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

int send_a_to, send_b_to;
int recv_a_from, recv_b_from;

switch (rank) {
    case 0:
        send_a_to = 2;
        send_b_to = 1;
        recv_a_from = 1;
        recv_b_from = 2;
        break;
    case 1:
        send_a_to = 0;
        send_b_to = 3;
        recv_a_from = 3;
        recv_b_from = 0;
        break;
    case 2:
        send_a_to = 3;
        send_b_to = 0;
        recv_a_from = 0;
        recv_b_from = 3;
        break;
    case 3:
        send_a_to = 1;
        send_b_to = 2;
        recv_a_from = 2;
        recv_b_from = 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_to, 99);
    MPI.COMM_WORLD.Send(new int[]{bi}, 0, 1, MPI.INT, send_b_to, 99);
    MPI.COMM_WORLD.Recv(received_a, 0, 1, MPI.INT, recv_a_from, 99);
    MPI.COMM_WORLD.Recv(received_b, 0, 1, MPI.INT, recv_b_from, 99);
} else {
```

```
            MPI.COMM_WORLD.Recv(received_a, 0, 1, MPI.INT, recv_a_from, 99);
            MPI.COMM_WORLD.Recv(received_b, 0, 1, MPI.INT, recv_b_from, 99);
            MPI.COMM_WORLD.Send(new int[]{ai}, 0, 1, MPI.INT, send_a_to, 99);
            MPI.COMM_WORLD.Send(new int[]{bi}, 0, 1, MPI.INT, send_b_to, 99);
        }

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

        MPI.Finalize();
    }
}
```

task2.java

```
package lebibop.lab2;

import mpi.MPI;

public class task2 {
    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;

        if (rank >= range) {
            System.out.printf("Process %2d (ID: %3d) is idle (no range assigned).%n",
                    rank, Thread.currentThread().getId());
        } else {
            int[] localRange = new int[2];

            if (rank == 0) {
                int localRangeSize = range / size;
                int remainder = range % size;

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

```
        if (i >= range) {
            continue;
        }

        int localStart, localEnd;

        if (i < remainder) {
            localStart = start + i * (localRangeSize + 1);
            localEnd = localStart + localRangeSize;
        } else {
            localStart = start + (i * localRangeSize) + remainder;
            localEnd = localStart + localRangeSize - 1;
        }

        if (i == 0) {
            localRange[0] = localStart;
            localRange[1] = localEnd;
        } else {
            MPI.COMM_WORLD.Send(new int[]{localStart, localEnd}, 0, 2, MPI.INT, i, 0);
        }
    }
} else {
    MPI.COMM_WORLD.Recv(localRange, 0, 2, MPI.INT, 0, 0);
}

if (localRange[0] > localRange[1]) {
    int temp = localRange[0];
    localRange[0] = localRange[1];
    localRange[1] = temp;
}

int localSum = 0;
for (int i = localRange[0]; i <= localRange[1]; i++) {
    localSum += i;
}

System.out.printf("Process %2d (ID: %3d): start=%4d, end=%4d, sum=%d%n",
        rank, Thread.currentThread().getId(), localRange[0], localRange[1], localSum);

if (rank != 0) {
    MPI.COMM_WORLD.Send(new int[]{localSum}, 0, 1, MPI.INT, 0, 0);
} else {
```

```
            int[] allSums = new int[size];
            allSums[0] = 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);
                  allSums[i] = receivedSum[0];
               } else {
                  allSums[i] = 0;
               }
            }

            int globalSum = 0;
            for (int i = 0; i < size; i++) {
               globalSum += allSums[i];
            }

            System.out.printf("Total sum: %d", globalSum);
         }
      }

      MPI.Finalize();
   }
}
```