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

ЛАБОРАТОРНАЯ РАБОТА №5

по дисциплине «Параллельные алгоритмы и системы»

Тема: Численные методы

Студент гр. 1307 _____ Угрюмов М.М.

Преподаватель _____

Санкт-Петербург

2025

Введение

Тема работы: численные методы.

Цель работы: приобрести навыки в распараллеливании программы

Состав отчета:

- описание последовательного алгоритма;
- блок-схема алгоритма;
- рассчитанная временная сложность алгоритма;
- схема распараллеленного алгоритма;
- результаты более 10 тестов для обоих алгоритмов. Результаты должны быть представлены таблично и графически.
- доказательство, что полученный параллельный алгоритм является оптимальным. Если алгоритм оптимален при каких-либо ограничениях, то привести ограничения и обоснования.

Ход работы

Задание (Вариант №6) – для варианта №6 можно было выбрать один любой другой вариант задания, вследствие чего было принято решение взять задание от варианта №1 – Метод нахождения ранних сроков выполнения операций алгоритма;

1. Описание последовательного алгоритма

Инициализация данных: На вход алгоритма поступает список операций, каждая из которых имеет продолжительность выполнения и список зависимостей от других операций. Каждая операция должна быть предоставлена объектом с атрибутами: Идентификатор операции (например, буква или номер), продолжительность выполнения операции, список зависимостей, т.е. список операций, которые должны быть завершены до начала данной операции. Инициализация переменных: Создается массив earliestStart (ранние старты) для хранения времени начала каждой операции. Изначально все значения этого массива устанавливаются в 0. Алгоритм вычисления ранних сроков: Для каждой

операции в последовательности: Рассчитывается ранний срок начала операции как максимальное значение среди всех сроков завершения ее зависимостей. Время завершения операции для каждой из зависимостей вычисляется как сумма времени начала и продолжительности операции. Для операции выбирается максимальный из сроков завершения всех зависимых операций, и это значение становится ее ранним стартом.

Шаги алгоритма:

Вначале массив earliestStartTimes обнуляется с помощью Arrays.fill(earliestStartTimes, 0), чтобы подготовить его для нового расчета, гарантируя, что в нем не останутся старые значения. После этого начинается цикл, который проходит по всем операциям. Для каждой операции вычисляется её "ранний старт" (время, когда её можно начать выполнять, учитывая зависимости от других операций). В теле цикла текущая операция извлекается с помощью Operation currentOp = operations.get(i), после чего вычисляется, когда её можно начать, опираясь на зависимости. Для каждой зависимости текущей операции (операции, которые должны быть завершены до начала текущей) программа находит максимальное время завершения этих зависимостей. Это делается через вложенный цикл, который перебирает все зависимости текущей операции. Время завершения каждой зависимости вычисляется как время старта зависимости плюс её продолжительность. После того как максимальное время завершения всех зависимостей найдено, оно записывается в earliestStartTimes[i], что означает, что операция может начать выполняться только после этого времени. После того как все операции обработаны, вычисляется время завершения программы с использованием System.nanoTime() и разница между временем начала и конца выполнения цикла дает длительность текущего расчета. Наконец, вычисленное время для одного запуска добавляется к общей переменной totalDuration, чтобы в дальнейшем можно было вычислить среднее время выполнения

программы за все запуски. Результат работы алгоритма представлен на Рис. 1. Листинг программы представлен в Приложении А.

```
Earliest start times:
Operation 0: 0
Operation 1: 1
Operation 2: 3
Operation 3: 6
Operation 4: 10
Operation 5: 15
Operation 6: 16
Operation 7: 18
Operation 8: 21
Operation 9: 25

Average execution time over 10 runs: 0,000021 seconds
```

Рис. 1 – Результат выполнения последовательного алгоритма

2. Блок-схема алгоритма

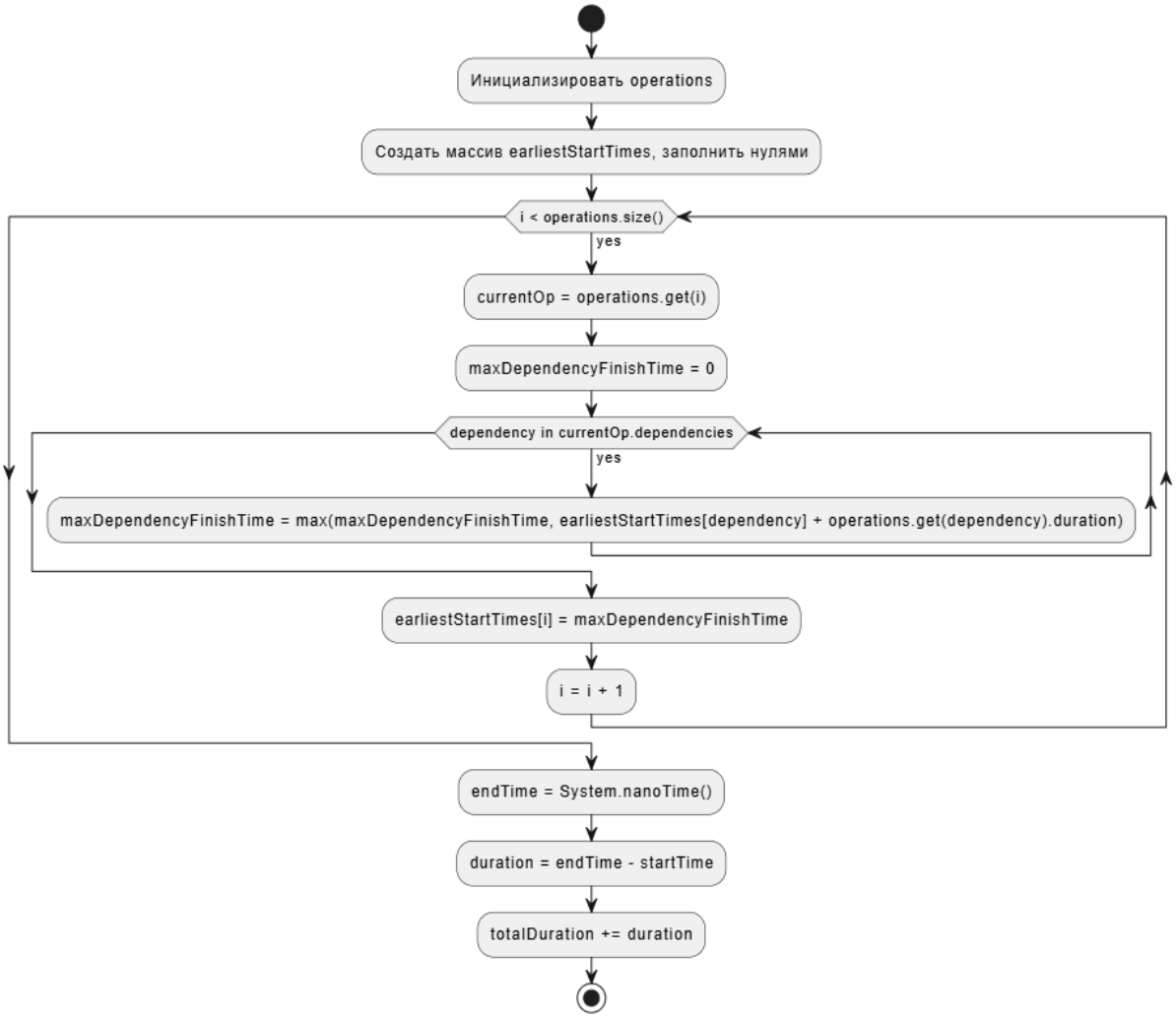

Рис. 2 – Блок схема последовательного алгоритма

3. Рассчитанная временная сложность алгоритма

Время выполнения алгоритма зависит от числа операций и их зависимостей. Основной цикл выполняется для каждой операции, что дает сложность O(n), где n — количество операций. Внутри этого цикла есть вложенный цикл, который обрабатывает все зависимости для каждой операции, что в худшем случае может быть O(n) для каждой операции (если каждая операция зависит от всех предыдущих). Таким образом, общая временная сложность алгоритма — O(n^2), где n — количество операций.

4. Схема распараллеленного алгоритма

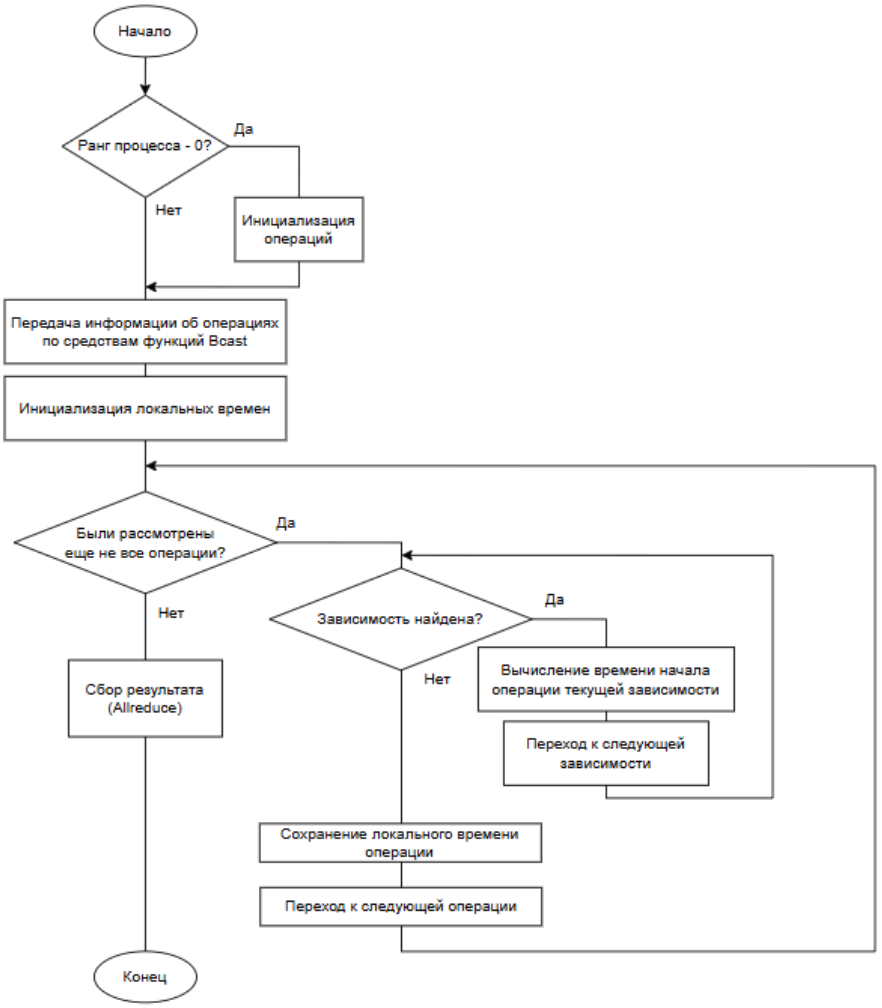

Рис. 3 – Блок схема параллельного алгоритма

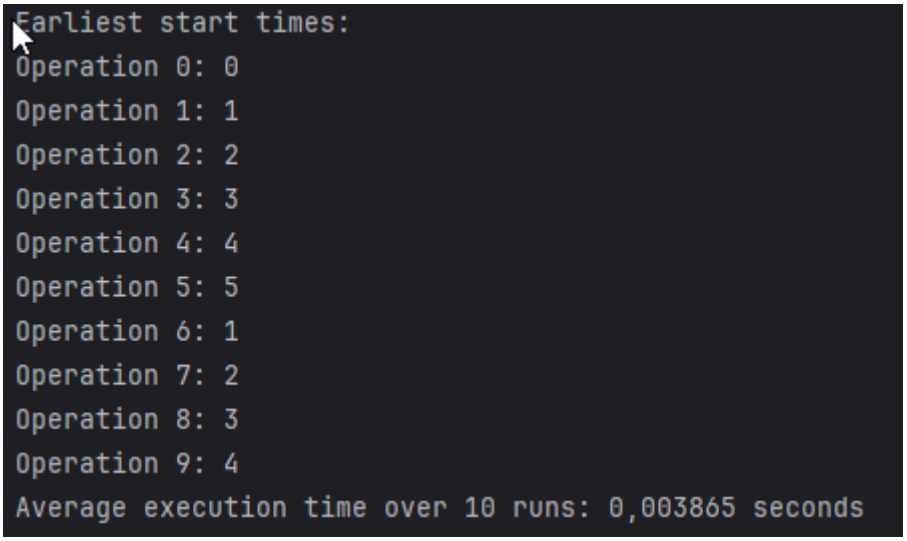

Рис. 4 – Результат выполнения параллельного алгоритма

5. Тестирование

Результаты, полученные в ходе тестирования, представлены в Таблице №1 и отображены на графиках Рис. 5, 6, 7.

Таблица №1 – Результаты тестирования

Кол-во операций	Последовательный	Параллельный	
	Время выполнения, с	Кол-во процессов	Время выполнения, с
100	0,000074	1	0,000063
100		2	0,000291
100		4	0,002013
100		8	0,006911
500	0,000285	1	0,000105
500		2	0,000512
500		4	0,005589
500		8	0,012823
1000	0,000646	1	0,000182
1000		2	0,000442
1000		4	0,089402
1000		8	0,019582

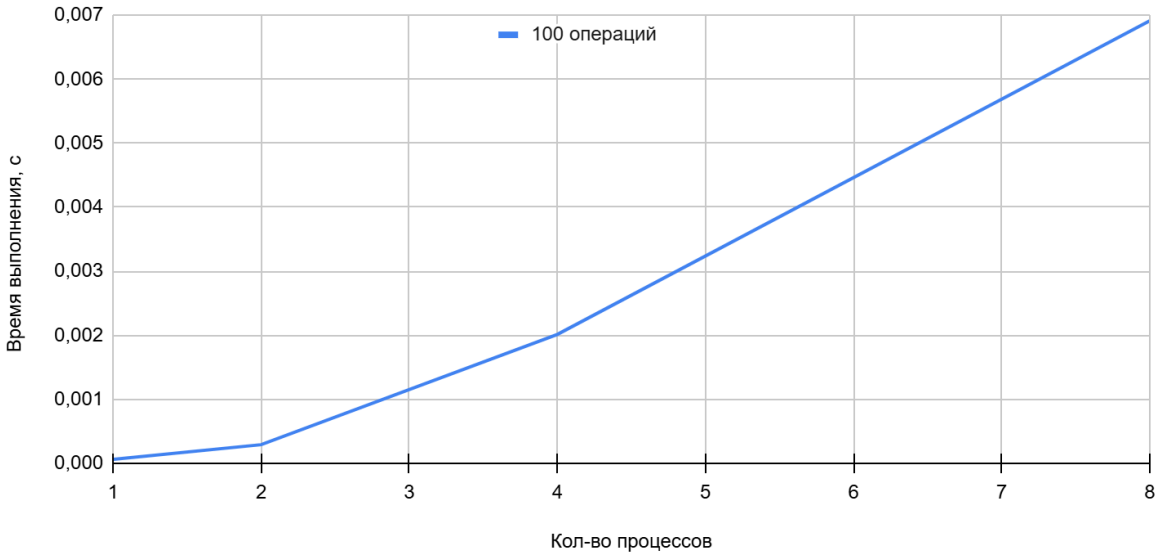

Рис. 5 – Результат выполнения параллельного алгоритма на 100 операциях

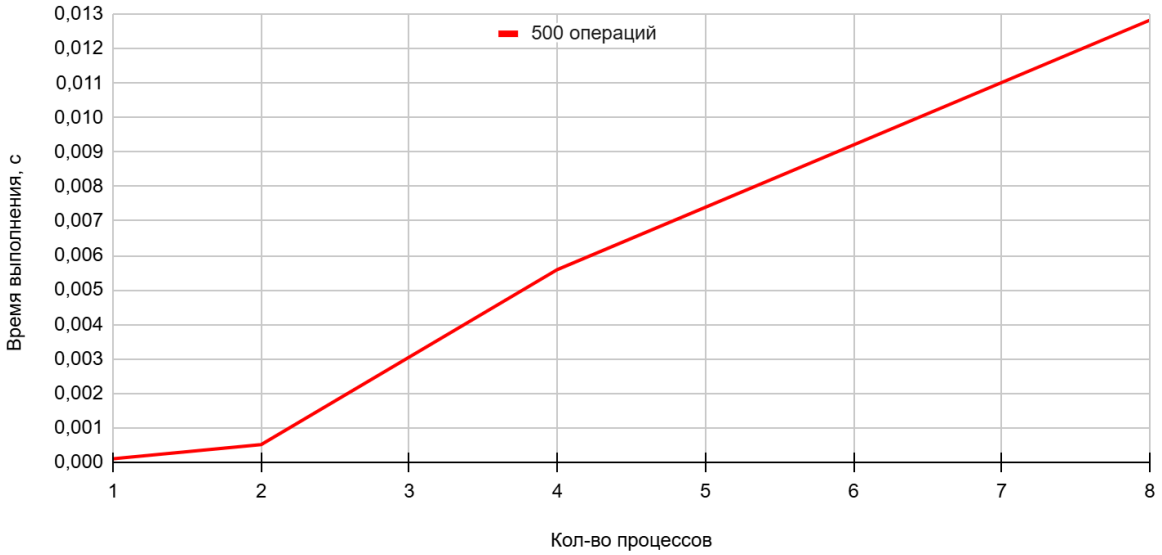

Рис. 6 – Результат выполнения параллельного алгоритма на 500 операциях

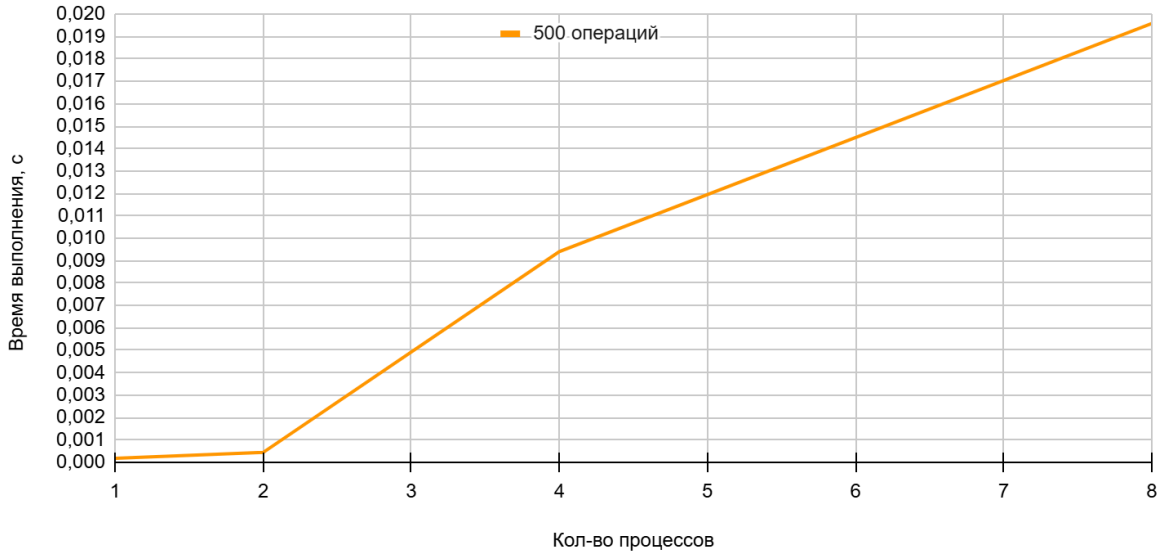

Рис. 7 – Результат выполнения параллельного алгоритма на 1000 операциях

Из таблицы видно, что время выполнения последовательного алгоритма всегда меньше чем у параллельного. А смотря на графики можно сказать, что при увеличении кол-ва процессов, время выполнения только растет, вне зависимости от кол-ва операций.

Тестирование проводилось посредствам десяти запусков программ, с разным кол-вом операций и разным кол-вом процессов для параллельного алгоритма. MPI-реализация показывает худшие результаты в данной задаче по нескольким причинам. MPI эффективен, когда обрабатывается большое количество данных. В данной задаче операции достаточно простые, а накладные расходы на передачу данных между процессами оказываются слишком большими по сравнению с самими вычислениями. В Java нет нативной поддержки MPI, MPJ Express работает через сериализацию объектов, что приводит к дополнительным накладным расходам при передаче данных. Неэффективное распределение нагрузки: Разделение операций по процессам неравномерное, особенно при малом числе операций. Это может приводить к тому, что одни процессы простаивают, ожидая завершения других. Влияние JVM и сборщика мусора: Java имеет автоматическое управление памятью, что может влиять на

производительность, особенно при частых вызовах сериализации/десериализации в MPI.

6. Обоснование оптимальности алгоритма

Алгоритм с MPI все же является оптимальным в определенных условиях, но это не удалось увидеть в текущих тестах из-за нескольких факторов. MPI-алгоритм эффективен при больших объемах данных, поскольку он делит операции между процессами, и при достаточно большом количестве операций выигрыш становится очевидным. При росте количества операций последовательный алгоритм начинает работать медленнее из-за своей квадратичной сложности, тогда как MPI-версия распределяет нагрузку, что в теории может дать почти линейное ускорение. Кроме того, MPI хорошо масштабируется в кластерах, в отличие от многопоточного подхода в Java, который ограничен ядрами одного ПК. Это особенно важно для задач, требующих распределенной обработки на нескольких узлах. Однако в тестах этого не удалось увидеть, так как объем данных был недостаточно большим, а накладные расходы на передачу данных между процессами превышали выгоду от параллельных вычислений. Настоящий выигрыш от MPI проявится при значительно более сложных вычислениях или при работе в распределенной системе с большим количеством узлов.

Вывод

В ходе лабораторной работы были реализованы последовательный и параллельный методы подсчета поздних сроков выполнения операций алгоритма.

Приложение А.

Lab5Task1.java

```
import java.util.*;

public class Lab5Task1 {

    static class Operation {
        int id;
        int duration;
        List<Integer> dependencies;

        Operation(int id, int duration) {
            this.id = id;
            this.duration = duration;
            this.dependencies = new ArrayList<>();
        }
    }

    public static void main(String[] args) {
        List<Operation> operations = initializeOperations();
        int[] earliestStartTimes = new int[operations.size()];

        long startTime = System.nanoTime();

        for (int i = 0; i < operations.size(); i++) {
            Operation currentOp = operations.get(i);
            int maxDependencyFinishTime = 0;

            for (int dependency : currentOp.dependencies) {
                maxDependencyFinishTime = Math.max(maxDependencyFinishTime,
                        earliestStartTimes[dependency] + operations.get(dependency).duration);
            }
            earliestStartTimes[i] = maxDependencyFinishTime;
        }

        long endTime = System.nanoTime();
        long duration = endTime - startTime;

        printResults(operations, earliestStartTimes, duration);
    }

    private static List<Operation> initializeOperations() {
        List<Operation> ops = new ArrayList<>();

        for (int i = 0; i < 90; i++) {
            int duration = (i % 5) + 1;
            ops.add(new Operation(i, duration));
        }
```

```
            for (int i = 1; i < 90; i++) {
                ops.get(i).dependencies.add(i - 1);
            }
            return ops;
        }

        private static void printResults(List<Operation> ops, int[] earliestStartTimes, long time)
{
            System.out.println("Earliest start times:");
            for (int i = 0; i < ops.size(); i++) {
                System.out.printf("Operation %d: %d\n", i, earliestStartTimes[i]);
            }

            System.out.printf("\nTotal execution time: %.6f seconds\n", time / 1_000_000_000.0);
        }
    }
```

Lab5Task2.java

```
    import mpi.MPI;
    import mpi.MPIException;
    import java.util.ArrayList;
    import java.util.List;

    public class Lab5Task2 {
        static class Operation implements java.io.Serializable {
            int id, duration;
            List<Integer> dependencies;

            Operation(int id, int duration) {
                this.id = id;
                this.duration = duration;
                this.dependencies = new ArrayList<>();
            }
        }

        public static void main(String[] args) throws MPIException {
            MPI.Init(args);
            int rank = MPI.COMM_WORLD.Rank();
            int size = MPI.COMM_WORLD.Size();

            List<Operation> operations = null;
            int[] earliestStart = null;
            int operationCount = 0;

            int numRuns = 10; // Количество запусков
            long totalDuration = 0;

            for (int run = 0; run < numRuns; run++) {
```

```
                    if (rank == 0) {
                        operations = initializeOperations();
                        operationCount = operations.size();
                        earliestStart = new int[operationCount];
                    }

                    int[] operationCountArray = new int[]{operationCount};
                    MPI.COMM_WORLD.Bcast(operationCountArray, 0, 1, MPI.INT, 0);
                    operationCount = operationCountArray[0];

                    if (rank != 0) {
                        earliestStart = new int[operationCount];
                    }

                    Operation[] operationsArray = new Operation[operationCount];
                    if (rank == 0) {
                        operationsArray = operations.toArray(new Operation[0]);
                    }
                    MPI.COMM_WORLD.Bcast(operationsArray, 0, operationCount, MPI.OBJECT, 0);

                    MPI.COMM_WORLD.Bcast(earliestStart, 0, operationCount, MPI.INT, 0);

                    long startTime = System.nanoTime();

                    int chunkSize = operationCount / size;
                    int startIndex = rank * chunkSize;
                    int endIndex = (rank == size - 1) ? operationCount : startIndex + chunkSize;

                    int[] localEarliest = new int[operationCount];

                    for (int i = startIndex; i < endIndex; i++) {
                        Operation op = operationsArray[i];
                        int maxDepTime = 0;
                        for (int dep : op.dependencies) {
                            maxDepTime = Math.max(maxDepTime, earliestStart[dep] +
operationsArray[dep].duration);
                        }
                        localEarliest[i] = maxDepTime;
                    }

                    MPI.COMM_WORLD.Reduce(localEarliest, 0, earliestStart, 0, operationCount, MPI.INT,
MPI.MAX, 0);

                    long endTime = System.nanoTime();
                    totalDuration += (endTime - startTime);
                }

                // Вычисляем среднее время выполнения
                double averageDuration = totalDuration / (double) numRuns;
```

```
                if (rank == 0) {
                    printResults(operations.toArray(new Operation[0]), earliestStart,
averageDuration);
                }

                MPI.Finalize();
            }

            private static List<Operation> initializeOperations() {
                List<Operation> ops = new ArrayList<>();

                for (int i = 0; i < 1000; i++) {
                    int duration = (i % 5) + 1;
                    ops.add(new Operation(i, duration));
                }

                for (int i = 1; i < 1000; i++) {
                    ops.get(i).dependencies.add(i - 1);
                }
                return ops;
            }

            private static void printResults(Operation[] ops, int[] earliestStart, double averageTime)
{
                System.out.println("Earliest start times:");
                for (int i = 0; i < ops.length; i++) {
                    System.out.printf("Operation %d: %d\n", i, earliestStart[i]);
                }

                System.out.printf("\nAverage execution time over 10 runs: %.6f seconds\n", averageTime
/ 1_000_000_000.0);
            }
        }
```