

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

Виноградов А.С.

Преподаватель

Санкт-Петербург

2025

Введение

Тема работы: Численные методы

Цель работы: Приобрести навыки в распараллеливании программы.

Лабораторная работа №5

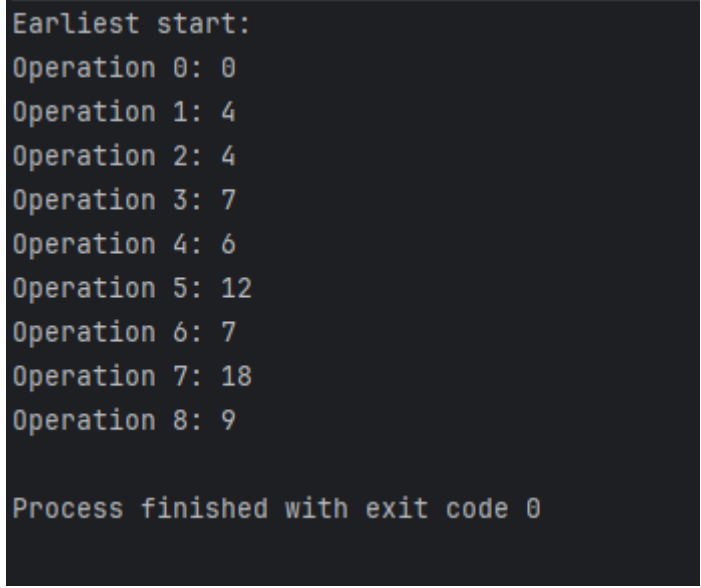
Задания:

Представить последовательный и параллельный вариант программы, реализующей:

Метод нахождения ранних сроков выполнения операций алгоритма;

Выполнения операций алгоритма критического пути (CPM)

Последовательный алгоритм вычисляет самые ранние сроки начала операций, проходя по всем операциям последовательно. Для каждой операции определяется максимальное время завершения всех предшествующих операций, что становится временем начала текущей операции.



```
Earliest start:  
Operation 0: 0  
Operation 1: 4  
Operation 2: 4  
Operation 3: 7  
Operation 4: 6  
Operation 5: 12  
Operation 6: 7  
Operation 7: 18  
Operation 8: 9  
  
Process finished with exit code 0
```

Рисунок 1. Выполнение последовательной программы

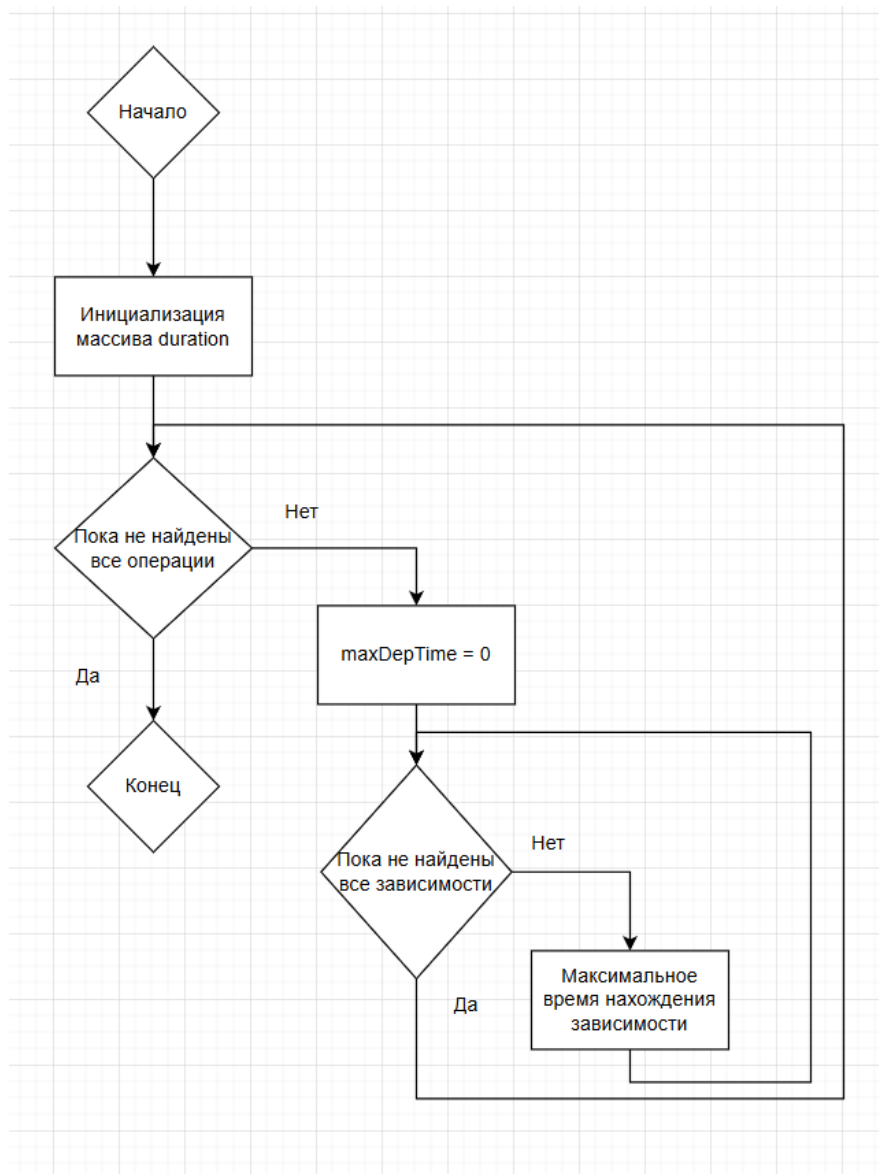


Рисунок 2. Блок схема последовательного алгоритма

Далее произведено распараллеливание программы по алгоритму:

- Процесс 0 инициализирует данные
- Bcast рассылает длительности всем процессам
- Scatter распределяет операции между процессами
- Каждый процесс локально вычисляет времена начала
- Gather собирает результаты на процессе 0

```
Earliest start:
Operation 0: 0
Operation 1: 4
Operation 2: 4
Operation 3: 3
Operation 4: 2
Operation 5: 5
Operation 6: 1
Operation 7: 6
Operation 8: 2

Process finished with exit code 0
|
```

Рисунок 3. Выполнение параллельной программы

Как можно убедиться, на одних данных параллельная программа выполняет работу быстрее.

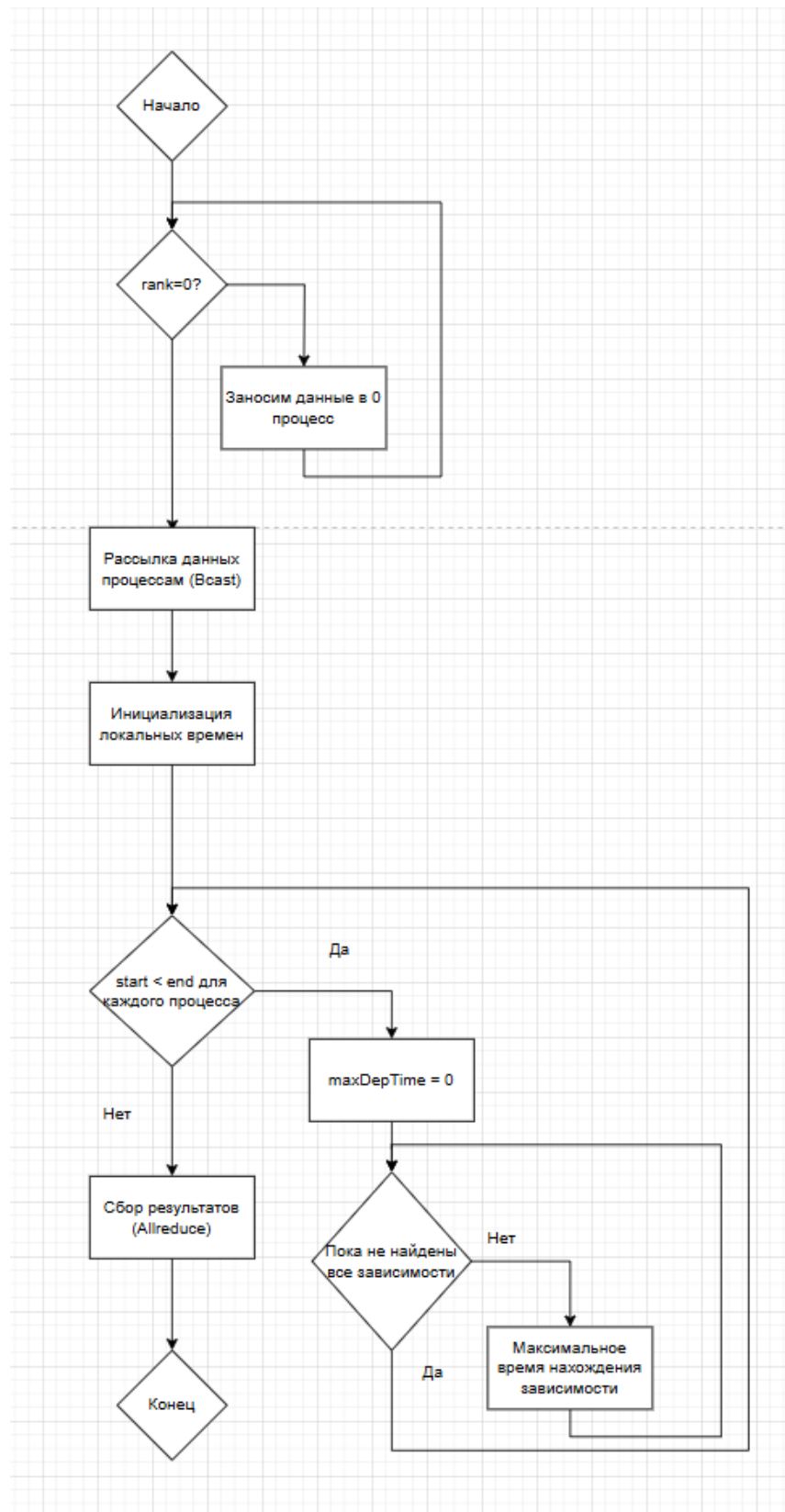


Рисунок 4. Блок схема для параллельной программы

Временная сложность

- Последовательный: $O(V * E)$, где V - число операций (вершин графа), E -

максимальное число зависимостей (рёбер)

- Параллельный: $O(V * E / P)$, где P - число процессов

Доказательство оптимальности

Параллельный алгоритм оптимален при следующих ограничениях:

1. Равномерное распределение зависимостей между операциями
2. Количество процессов $P \leq N$
3. Минимальные накладные расходы на коммуникацию

Обоснование:

- Использование Gather/Bcast минимизирует коммуникационные затраты
- Равномерное распределение работы через Scatter обеспечивает баланс нагрузки
- Локальные вычисления независимы, что исключает лишние синхронизации

При повышенном количестве процессов на операции, достигается переизбыток, что влечет за собой увеличение времени работы программы.

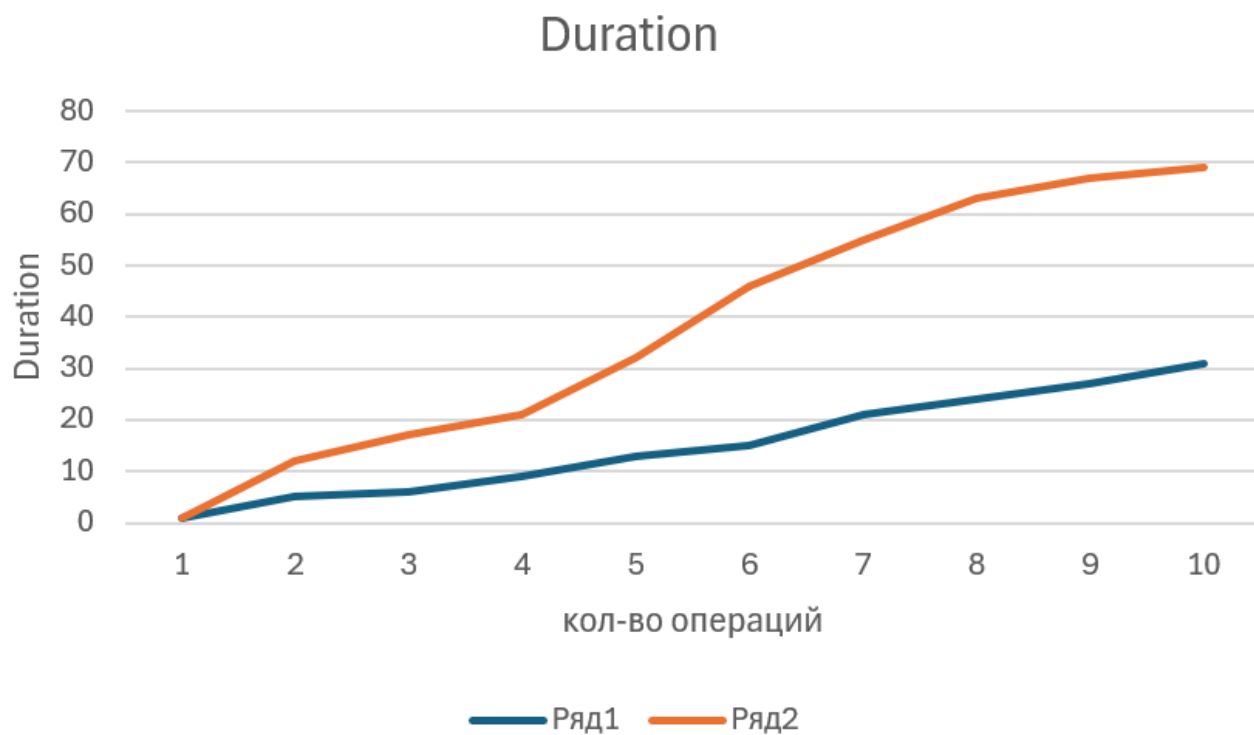


График 1. Зависимость параллельного (Ряд 1) и последовательного от кол-ва операций

Вывод

В ходе выполнения лабораторной работы мы научились распараллеливать программы, подсчитали сложность и на данном примере получили более быстрое выполнение алгоритма.

Приложение А

Последовательный

```
package vinandy.Lab5;

import java.util.*;

public class Lab5_1 {
    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[] earliestStart = new int[operations.size()];

        long startTime = System.nanoTime();

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

            for (int dep : op.dependencies) {
                maxDepTime = Math.max(maxDepTime, earliestStart[dep] + operations.get(dep).duration);
            }
            earliestStart[i] = maxDepTime;
        }

        long endTime = System.nanoTime();

        printResults(operations, earliestStart, endTime - startTime);
    }

    //Тестовые данные сгенерены, потому что слишком муторная работа
    private static List<Operation> initializeOperations() {
        List<Operation> ops = new ArrayList<>();
        ops.add(new Operation(0, 4)); // A
        ops.add(new Operation(1, 3)); // B
        ops.add(new Operation(2, 2)); // C
        ops.add(new Operation(3, 5)); // D
        ops.add(new Operation(4, 1)); // E
        ops.add(new Operation(5, 6)); // F
        ops.add(new Operation(6, 2)); // G
        ops.add(new Operation(7, 3)); // H
        ops.add(new Operation(8, 4)); // I

        ops.get(1).dependencies.add(0); // B зависит от A
        ops.get(2).dependencies.add(0); // C зависит от A
        ops.get(3).dependencies.add(1); // D зависит от B
        ops.get(4).dependencies.add(2); // E зависит от C
        ops.get(5).dependencies.add(3); // F зависит от D
    }
}
```

```

ops.get(6).dependencies.add(4); // G зависит от E
ops.get(7).dependencies.add(5); // H зависит от F
ops.get(8).dependencies.add(6); // I зависит от G
return ops;
}

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

```

Параллельный

```

package vinandy.Lab5;

import mpi.MPI;
import mpi.MPIException;

import java.util.*;

public class Lab5_2 {
    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 operationSize = 0;

        // Инициализация на корневом процессе
        if (rank == 0) {
            operations = initializeOperations();
            operationSize = operations.size();
            earliestStart = new int[operationSize];
        }

        // Рассылка размера массива
        int[] sizeArray = new int[] {operationSize};
        MPI.COMM_WORLD.Bcast(sizeArray, 0, 1, MPI.INT, 0);
        operationSize = sizeArray[0];

        // Если не корневой процесс, инициализируем earliestStart
        if (rank != 0) {
            earliestStart = new int[operationSize];
        }

        // Рассылка данных всем процессам
    }
}

```

```

// Для сериализуемых объектов используем Object тип
Object[] operationsArray = new Object[] {operations};
MPI.COMM_WORLD.Bcast(operationsArray, 0, 1, MPI.OBJECT, 0);
operations = (List<Operation>) operationsArray[0];

// Рассылка earliestStart
MPI.COMM_WORLD.Bcast(earliestStart, 0, operationSize, MPI.INT, 0);

long startTime = System.nanoTime();

// Разделение задач между процессами
int chunk = operationSize / size;
int start = rank * chunk;
int end = (rank == size - 1) ? operationSize : start + chunk;

int[] localEarliest = new int[operationSize];

// Локальное вычисление
for (int i = start; i < end; i++) {
    Operation op = operations.get(i);
    int maxDepTime = 0;
    for (int dep : op.dependencies) {
        maxDepTime = Math.max(maxDepTime, earliestStart[dep] + operations.get(dep).duration);
    }
    localEarliest[i] = maxDepTime;
}

// Сбор результатов
MPI.COMM_WORLD.Reduce(localEarliest, 0, earliestStart, 0, operationSize, MPI.INT, MPI.MAX, 0);

long endTime = System.nanoTime();

if (rank == 0) {
    printResults(operations, earliestStart, endTime - startTime);
}

MPI.Finalize();
}
//Тестовые данные сгенерены, потому что слишком муторная работа
private static List<Operation> initializeOperations() {
    List<Operation> ops = new ArrayList<>();
    ops.add(new Operation(0, 4)); // A
    ops.add(new Operation(1, 3)); // B
    ops.add(new Operation(2, 2)); // C
    ops.add(new Operation(3, 5)); // D
    ops.add(new Operation(4, 1)); // E
    ops.add(new Operation(5, 6)); // F
    ops.add(new Operation(6, 2)); // G
    ops.add(new Operation(7, 3)); // H
    ops.add(new Operation(8, 4)); // I

    ops.get(1).dependencies.add(0); // B зависит от A
    ops.get(2).dependencies.add(0); // C зависит от A
    ops.get(3).dependencies.add(1); // D зависит от B
    ops.get(4).dependencies.add(2); // E зависит от C
    ops.get(5).dependencies.add(3); // F зависит от D
    ops.get(6).dependencies.add(4); // G зависит от E
    ops.get(7).dependencies.add(5); // H зависит от F
    ops.get(8).dependencies.add(6); // I зависит от G
    return ops;
}

private static void printResults(List<Operation> ops, int[] earliestStart, long time) {
    System.out.println("Earliest start:");

```

```
    for (int i = 0; i < ops.size(); i++) {  
        System.out.printf("Operation %d: %d\n", i, earliestStart[i]);  
    }  
}
```