

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет «Информатика и вычислительная техника»

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

**Лабораторная работа №3**

по дисциплине «Основы теории принятия решений»

**Выполнил:**

студент уч. гр. ВМО41

Волкова Эмилия Юрьевна

**Проверил:**

проф. Кобак В. Г.

Ростов-на-Дону

2025

**Введение**

Предметом области исследования расписаний является круг задач проектирования и организационного управления в различных системах, в которых требуется найти наилучшее (оптимальное) значение выбранных критериев их функционирования с учетом имеющихся ограничений.

Программирование для многопроцессорных машинных систем связано с распараллеливанием и синхронизацией вычислений и организацией выполнения параллельных вычислительных процессов. Это выдвигает целый ряд сложных задач, среди которых весьма важными являются, расчет характеристик времени и количества операций, требующихся для выполнения параллельных программ, и построения расписаний (планов), выполнения параллельных программ на многопроцессорных и многомашинных вычислительных системах.

Модели параллельных программ и операционные характеристики процессов их выполнения служат основой для планирования параллельных вычислительных процессов, т.е. для построения расписаний указанных процессов. Расписания параллельных вычислительных процессов определяют порядок выполнения программы на вычислительной системе, включая распределение частей программы по процессам. С увеличением числа распределяемых частей программ и количества используемых процессоров сложность построения оптимальных расписаний обычно резко возрастает. Поэтому важное значение имеют простые в построении и удобные в реализации приближенные расписания параллельных вычислительных процессов, близкие к оптимальным с точки зрения времени выполнения параллельных программ.

**Постановка задачи**

В теории расписаний решение NP-полных задач с использованием точных методов эффективно только для малых размерностей. Однако, по мере увеличения размерности, сложность таких задач возрастает экспоненциально, что делает невозможным нахождение точного решения за разумное время. В таких случаях применяются приближенные алгоритмы, основанные на эвристических методах. Одним из таких методов является **алгоритм Плотникова–Зверева**, предназначенный для решения **неоднородной минимаксной задачи**.

Рассматривается система обслуживания, состоящая из N независимых устройств:

P = { p1, p2 , ... , pn}

На вход подается поток M независимых параллельных заданий:

T = { t1, t2, ... , tm}

Каждое задание ti может быть выполнено на любом устройстве pjp\_j, при этом время выполнения зависит от конкретного устройства и определяется матрицей

Tτ = { τ(ti, pj) }

где τ (ti, pj) — время обслуживания задания ti устройством pj.

При этом:

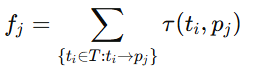
* Устройства **не идентичны**, то есть скорость их обработки может различаться.
* Каждое устройство pj может одновременно выполнять **не более одного задания**.
* **Не допускается прерывание** выполнения заданий.

Необходимо распределить задания между устройствами **так, чтобы минимизировать максимальное время обработки** среди всех устройств.

Критерий оптимальности выражается через **минимаксный критерий**:



где



— время завершения работы процессора pj.

**Алгоритм Плотникова–Зверева**

1. **Сортировка задач**.
   * Рассчитывается сумма элементов в каждой строке матрицы.
   * Строки сортируются по убыванию этой суммы.
   * Обозначим отсортированную матрицу как T\*.
2. **Инициализация массива загрузки**.
   * Создается массив L = [0, 0, …, 0] загрузки , где количество элементов равно числу устройств.
3. **Назначение заданий**.
   * Для каждой строки i матрицы T\*:
     + Определяется минимальный элемент  .
     + Индекс этого jmin элемента фиксируется.
4. **Обновление загрузки**.
   * Выбранный элемент добавляется к соответствующему устройству: 
5. **Повторение**.
   * Процесс повторяется для всех строк матрицы.

Формула минимаксного критерия:



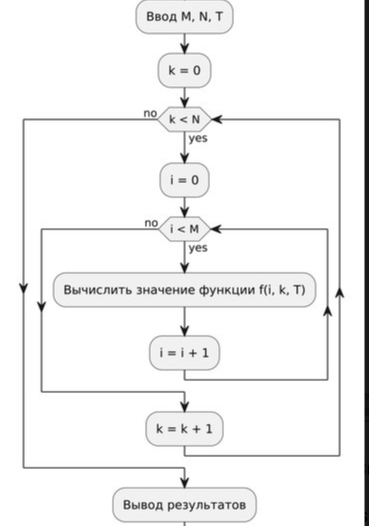


Рисунок 1 – Схема алгоритма алгоритма Плотникова-Зверева

**Метод минимальных элементов**

1. **Сортировка задач**.
   * Рассчитывается сумма элементов в каждой строке матрицы.
   * Строки сортируются по убыванию этой суммы.
   * Обозначим отсортированную матрицу как T\*.
2. **Инициализация массива загрузки**.
   * Создается массив L = [0, 0, …, 0] загрузки , где количество элементов равно числу устройств.
3. **Выбор минимума**.
   * Для каждой строки i выбирается минимальный элемент  и его индекс jmin.
4. **Обновление загрузки**.
   * Обновляется соответствующий элемент массива загрузки: 
5. **Повторение**.
   * Действия продолжаются до полного распределения заданий.

Формула критерия распределения:



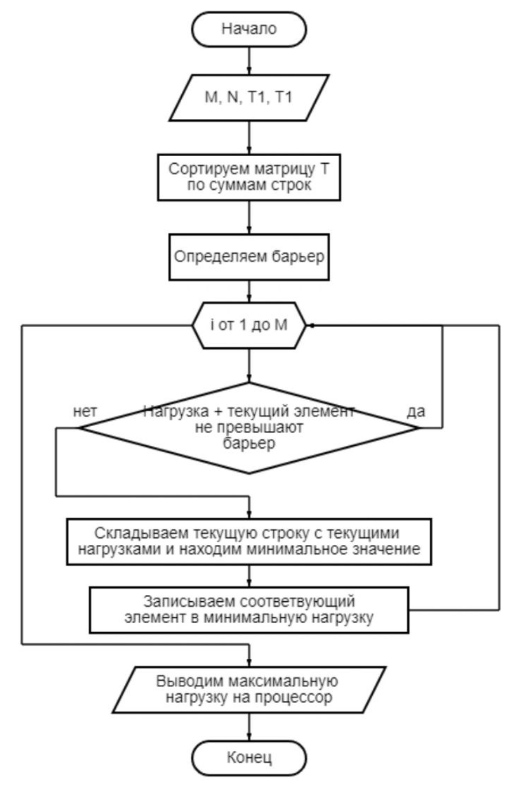


Рисунок 2 – Схема метода минимальных элементов (с барьером)

**Листинг программы**

import random

# Функция для метода критического пути (CMP)

def cmp\_schedule(tasks, processors):

    # Инициализируем список с нагрузкой на каждый процессор

    load = [0] \* processors

    # Распределяем задания по процессорам

    for task in tasks:

        # Назначаем задачу на процессор с минимальной текущей нагрузкой

        min\_index = load.index(min(load))

        load[min\_index] += task

    return load

# Функция для метода половинного деления (HDMT)

def hdmt\_schedule(tasks, processors):

    if processors % 2 != 0:

        raise ValueError("Количество процессоров должно быть четным")

    # Разделяем на два набора

    half\_size = processors // 2

    load\_a = [0] \* half\_size

    load\_b = [0] \* half\_size

    # Первый уровень - распределение по двум процессорам

    for i, task in enumerate(tasks):

        if i % 2 == 0:

            load\_a[i // 2 % half\_size] += task

        else:

            load\_b[i // 2 % half\_size] += task

    return load\_a + load\_b

# Основная программа

def main():

    num\_tasks = int(input("M: "))  # количество заданий

    processors = int(input("N: ")) # количество процессоров

    rand\_min = int(input("Min: "))

    rand\_max = int(input("Max: "))

    '''num\_massives = int(input("Кол-во массивов: ")) # количество списков тасков'''

    num\_massives = 100

    all\_cmp = []

    all\_cmp\_asc = []

    all\_cmp\_desc = []

    all\_hdmt = []

    all\_hdmt\_asc = []

    all\_hdmt\_desc = []

    cmp\_rand = 0

    cmp\_asc = 0

    cmp\_desc = 0

    hdmt\_rand = 0

    hdmt\_asc = 0

    hdmt\_desc = 0

    for i in range(num\_massives):

        tasks = []

        for i in range(num\_tasks):

            tasks.append(random.randint(rand\_min, rand\_max))

        tasks\_asc = sorted(tasks)

        task\_desc = sorted(tasks, reverse=True)

        result\_cmp = cmp\_schedule(tasks, processors)

        result\_cmp\_asc = cmp\_schedule(tasks\_asc, processors)

        result\_cmp\_desc = cmp\_schedule(task\_desc, processors)

        result\_hdmt = hdmt\_schedule(tasks, processors)

        result\_hdmt\_asc = hdmt\_schedule(tasks\_asc, processors)

        result\_hdmt\_desc = hdmt\_schedule(task\_desc, processors)

        all\_cmp.append(max(result\_cmp))

        all\_cmp\_asc.append(max(result\_cmp\_asc))

        all\_cmp\_desc.append(max(result\_cmp\_desc))

        all\_hdmt.append(max(result\_hdmt))

        all\_hdmt\_asc.append(max(result\_hdmt\_asc))

        all\_hdmt\_desc.append(max(result\_hdmt\_desc))

        max\_val\_cmp = min([max(result\_cmp), max(result\_cmp\_asc), max(result\_cmp\_desc)])

        if max\_val\_cmp == max(result\_cmp):

            cmp\_rand += 1

        elif max\_val\_cmp == max(result\_cmp\_asc):

            cmp\_asc += 1

        elif max\_val\_cmp == max(result\_cmp\_desc):

            cmp\_desc += 1

        max\_val\_hdmt = min([max(result\_hdmt), max(result\_hdmt\_asc), max(result\_hdmt\_desc)])

        if max\_val\_hdmt == max(result\_hdmt):

            hdmt\_rand += 1

        elif max\_val\_hdmt == max(result\_hdmt\_asc):

            hdmt\_asc += 1

        elif max\_val\_hdmt == max(result\_hdmt\_desc):

            hdmt\_desc += 1

    for i in [all\_cmp, all\_cmp\_asc, all\_cmp\_desc]:

        aver = sum(i)/len(i)

        i.append(aver)

    for i in [all\_hdmt, all\_hdmt\_asc, all\_hdmt\_desc]:

        aver = sum(i)/len(i)

        i.append(aver)

    print("-------Lab 1----------")

    print("CMP: ")

    print("Случайно: ", all\_cmp[len(all\_cmp)-1], cmp\_rand)

    print("По возрастанию: ", all\_cmp\_asc[len(all\_cmp\_asc)-1], cmp\_asc)

    print("По убыванию: ", all\_cmp\_desc[len(all\_cmp\_desc)-1], cmp\_desc)

    print("----------------------")

    print("HDMT")

    print("Случайно: ", all\_hdmt[len(all\_hdmt)-1], hdmt\_rand)

    print("По возрастанию: ", all\_hdmt\_asc[len(all\_hdmt\_asc)-1], hdmt\_asc)

    print("По убыванию: ", all\_hdmt\_desc[len(all\_hdmt\_desc)-1], hdmt\_desc)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Вычислительные эксперименты**

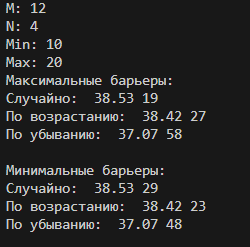


Рисунок 3 – Эксперимент №1

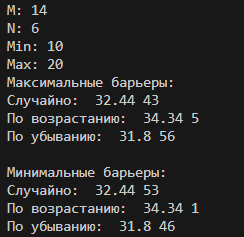


Рисунок 4 – Эксперимент №2

**Вывод**

Проанализировав данные, полученные входе экспериментов, можно сделать вывод, что алгоритм Пллотникова-Зверева является более эффективным, т.к. точность решения в данном случае несколько лучше, чем метод минимальных элементов.

**Литература**

1. Коффман Э.Г. “Теория расписания и вычислительные машины” – M.: “Наука”, 1987

2. Романовский И.В. “Алгоритмы решения экстремальных задач” – М.: “Наука”, 1977

3. Пашкеев С.Д., Минязов Р.И., Могилевский В.Д. “Машинные методы оптимизации в технике связи” – М.: “Связь”, 1976.