

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 №1**

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**

Имеется вычислительная система (ВС), состоящая из *N* несвязанных идентичных устройств (приборов, процессоров и т.п.)



На обслуживание в ВС поступает набор из *M* независимых параллельных заданий (работ)  известно время решения  задания  на любом из устройств. При этом каждое задание может выполняться на любом из устройств (процессоре), в каждый момент времени отдельный процессор обслуживает не более одного задания и выполнение задания не прерывается для передачи на другой процессор. Требуется найти такое распределение заданий по процессорам, при котором суммарное время выполнения заданий на каждом из процессоров было бы минимальным. Под расписанием следует понимать отображение , такое что, если , то говорят что задание , в расписании *R* назначенного на процессор . При сделанных выше допущениях, расписание можно представить разбиением множества заданий *T* на *N* непересекающихся подмножеств 

Критерий, используемый для минимизации времени завершения обслуживания заданий, является минимальным критерием и определяется в следующем виде: , где

- время завершения работы процессора .

**Алгоритмы списочных расписаний**

1. Алгоритм критического пути:

Широкое распространение получил простой и эффективный алгоритм списочного расписания: метод критического пути (method critical path), состоящий из следующих шагов:

Шаг 1. Упорядочиваем в порядке убывания веса множество заданий;

Шаг 2. Начальная загрузка всех устройств равна нулю, выбирается первое устройство (процессор) и первое задание из множества заданий , для всех .

Шаг 3. Выбираем устройство, на котором будет обрабатываться текущее задание, (процессор) с наименьшим среди всех процессоров (т.е. процессоров с минимальным временем загрузки на текущий момент).

Шаг 4. Назначить следующим заданием на процессор , .

Шаг 5. Если , то увеличить и перейти на Шаг 3. Иначе алгоритм заканчивается с построенным расписанием , общая длина которого равняется .

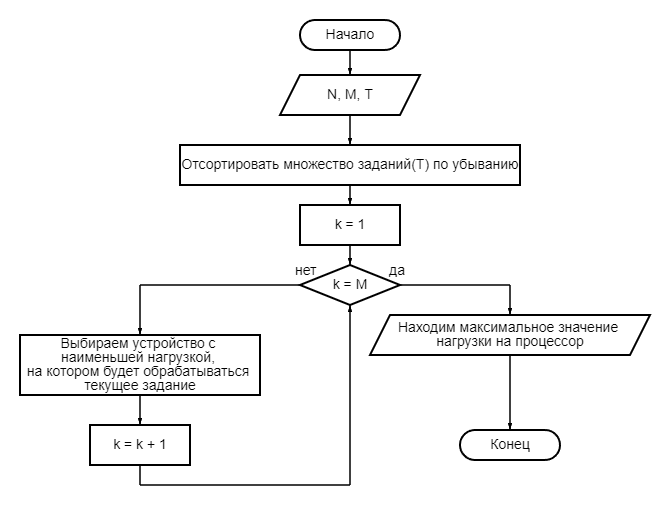


Рисунок 1 – Схема алгоритма критического пути

1. Алгоритм половинного деления множества задач:

Приведённый далее алгоритм является одной из модификаций алгоритма критического пути (CMP), но, алгоритм половинного деления состоит из двух уровней, и актуален только для чётного числа процессоров.

Как и в CMP, в алгоритме половинного деления (Half Division Multitude Tasks или HDMT) имеющуюся однородную матрицу упорядочить по убыванию. Далее, как и в алгоритме CMP, распределяем всё множество заданий, но, только, на два процессора -  и  (в этом заключается первый уровень). На втором уровне задания с процессоров  и  разбрасываются на две группы процессоров (по N/2 процессоров в каждой).

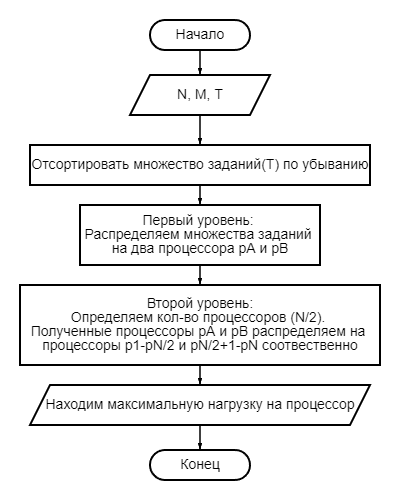


Рисунок 2 – Схема алгоритма половинного деления множества задач

**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():

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

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

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

all\_cmp = []

all\_cmp\_asc = []

all\_cmp\_desc = []

all\_hdmt = []

all\_hdmt\_asc = []

all\_hdmt\_desc = []

for i in range(num\_massives):

tasks = random.sample(range(1, 101), num\_tasks)

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))

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("----------------------")

print("CMP: ")

print("Случайно: ", all\_cmp[processors])

print("По возрастанию: ", all\_cmp\_asc[processors])

print("По убыванию: ", all\_cmp\_desc[processors])

print("----------------------")

print("HDMT")

print("Случайно: ", all\_hdmt[processors])

print("По возрастанию: ", all\_hdmt\_asc[processors])

print("По убыванию: ", all\_hdmt\_desc[processors])

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

main()

**Вычислительные эксперименты**

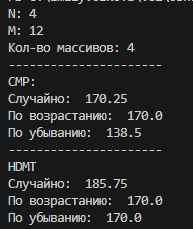


Рисунок 3 – Эксперимент №1

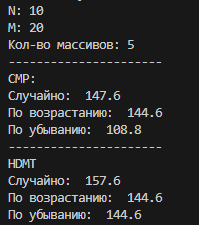


Рисунок 4 – Эксперимент №2

**Вывод**

Проанализировав данные, полученные входе экспериментов, можно сделать вывод, что модифицированный алгоритм HDMT является более эффективным, т.к. точность решения в данном случае несколько лучше, чем у CPM.

**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.